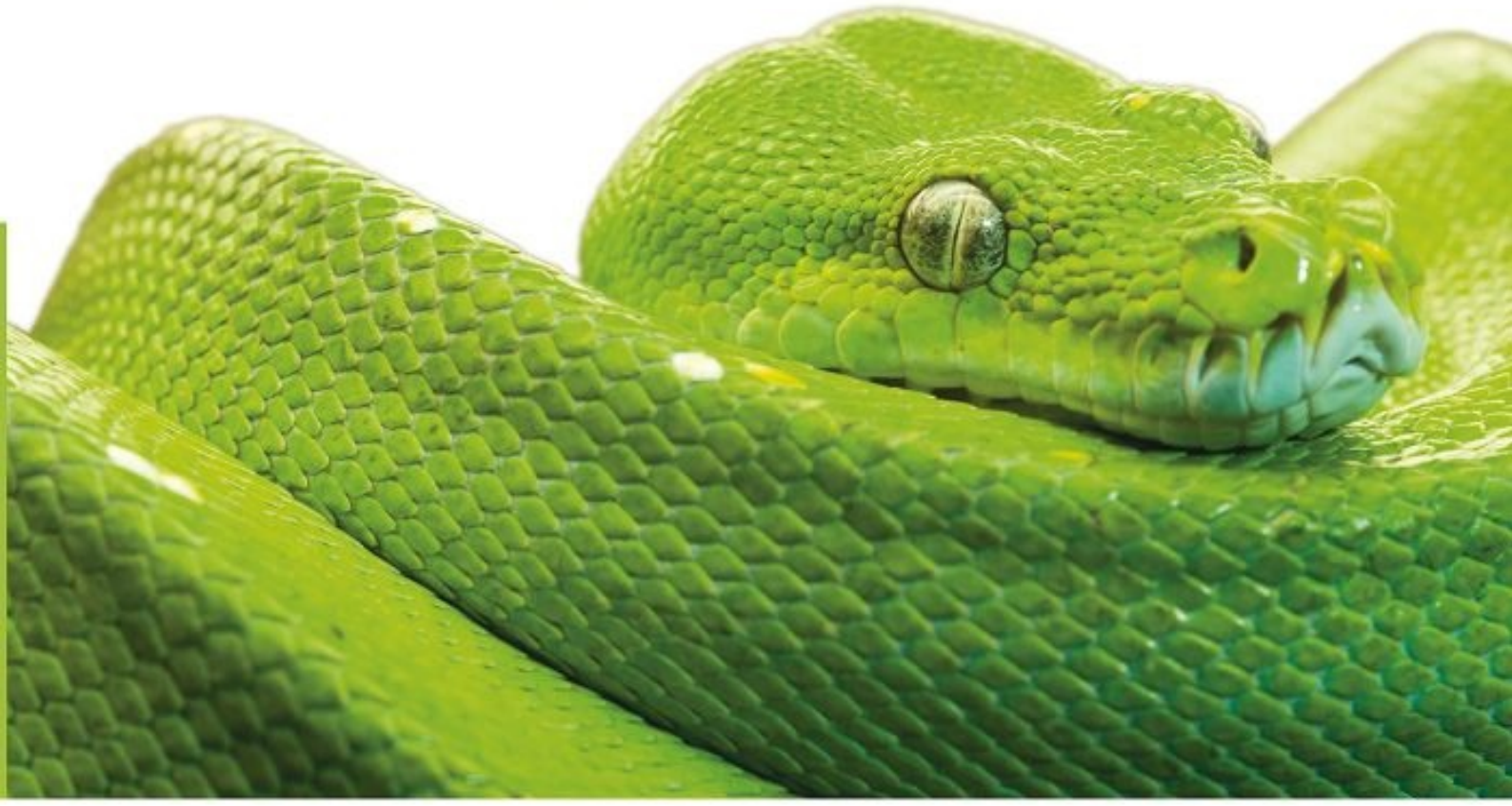
Mark Summerfield

Python

Programmazione avanzata

SCOPRIRE
i migliori pattern
per Python

MIGLIORARE
le prestazioni
con Cython



"Oggi dozzine di ingegneri di Google usano Python e continuiamo a ricercare persone capaci di lavorare con questo linguaggio."
– Peter Norvig

APOGEO

PYTHON
PROGRAMMAZIONE AVANZATA

Mark Summerfield

ISBN edizione cartacea: 9788850333028

L’edizione cartacea è in vendita nelle migliori librerie.

~

Sito web: www.apogeonline.com

Seguici su Twitter @apogeonline

Questo libro è dedicato a chi mette a disposizione software libero e open source in ogni parte del mondo
- la vostra generosità va a beneficio di tutti noi

Ho sviluppato software con Python per 15 anni, in vari campi. Nel tempo ho visto la nostra comunità maturare e crescere considerevolmente. Sono passati da tempo i giorni in cui dovevamo “vendere” Python ai nostri dirigenti per poterlo utilizzare nei progetti di lavoro. Oggi il mercato del lavoro per i programmatori in Python è forte. La partecipazione alle conferenze dedicate a Python è più che mai alta, a livello regionale, nazionale e internazionale. Progetti come OpenStack stanno spingendo il linguaggio in nuovi settori e nello stesso tempo contribuiscono ad attirare nuovi talenti nella nostra comunità. Grazie a una comunità forte e in continua espansione, oggi più che mai non mancano i libri dedicati a questo linguaggio.

Mark Summerfield è ben noto alla comunità di Python per i suoi scritti tecnici su Qt e Python. Un altro dei suoi libri, *Programming in Python 3*, è in cima al mio elenco di testi consigliati per apprendere il linguaggio, che spesso mi viene chiesto in qualità di organizzatore del gruppo di utenti di Atlanta, in Georgia. Anche questo nuovo libro entrerà nel mio elenco, ma per un pubblico un po’ diverso.

La maggior parte dei libri di programmazione si posiziona a due estremi di un intervallo che parte dalle guide introduttive a un linguaggio (o alla programmazione in generale) per arrivare ai testi più avanzati su argomenti molto specifici come lo sviluppo di applicazioni web, le applicazioni GUI o la bioinformatica. Mentre scrivevo *The Python Standard Library by Example*, volevo rivolgermi ai lettori che si trovavano tra questi due estremi - programmatori affermati e generalisti, già familiari con il linguaggio ma che volevano migliorare le loro competenze andando oltre le nozioni di base e senza limitarsi a un singolo campo di applicazione. Quando il mio editor mi ha chiesto un parere sulla proposta del libro di Mark, sono stato lieto di constatare che *Python in Practice* si rivolge allo stesso tipo di lettori.

È passato parecchio tempo da quando ho incontrato in un libro un’idea che fosse immediatamente applicabile a uno dei miei progetti, senza essere legata a uno specifico framework o a una particolare libreria. L’anno scorso ho lavorato su un sistema per la misurazione di servizi cloud OpenStack. Lungo il percorso, il team si è reso conto che i dati che stavamo raccogliendo per le fatturazioni potevano essere utili anche per altri scopi, come la reportistica e il monitoraggio, perciò abbiamo progettato il sistema in modo da inviarli a più consumatori facendo passare i campioni attraverso una pipeline di trasformazioni riutilizzabili.

Più o meno nello stesso tempo in cui il codice per la pipeline veniva messo a punto, sono stato coinvolto nella revisione tecnica di questo libro. Dopo aver letto i primi paragrafi della bozza del Capitolo 3 mi è stato subito chiaro che la nostra implementazione della pipeline era più complicata del necessario. La tecnica di concatenazione di coroutine illustrata da Mark è molto più elegante e facile da comprendere, perciò ho immediatamente inserito nella nostra roadmap la modifica del progetto per il prossimo ciclo di rilascio.

Questo libro è ricco di consigli ed esempi altrettanto utili, che vi aiuteranno a migliorare le vostre competenze. I generalisti come me troveranno descrizioni di diversi strumenti interessanti che magari non hanno mai incontrato prima. E per i programmatori più esperti, o per chi sta compiendo una fase di transizione dalla prima fase della carriera, questo libro aiuterà a riflettere sui problemi da una diversa prospettiva e fornirà tecniche per costruire soluzioni più efficaci.

Doug Hellmann
Senior Developer, DreamHost
Maggio 2013

Questo libro si rivolge ai programmatori in Python che desiderano ampliare e approfondire la loro conoscenza del linguaggio in modo da poter migliorare la qualità, l'affidabilità, la velocità, la facilità di manutenzione e di utilizzo dei loro programmi. Presenta numerosi esempi pratici e idee per migliorare la programmazione in Python.

Il libro è centrato su quattro temi chiave: i design pattern per scrivere codice in modo elegante ed efficace, la concorrenza e Cython (Python compilato) per aumentare la velocità di elaborazione, l'elaborazione di rete ad alto livello e la grafica.

Il volume *Design Patterns: Elements of Reusable Object-Oriented Software* (cfr. la bibliografia a fine libro per i dettagli) è stato pubblicato già nel 1995 e tuttavia continua a esercitare una forte influenza sulle pratiche di programmazione orientata agli oggetti.

Questo libro esamina tutti i design pattern nel contesto di Python, fornendo esempi per quelli ritenuti più utili, e spiegando perché alcuni sono irrilevanti per i programmatori. Questi pattern sono trattati nei Capitoli 1, 2 e 3.

Il GIL (*Global Interpreter Lock*) di Python evita che il codice sia eseguito su più di un core del processore alla volta.

NOTA

Questa limitazione si applica a CPython, l'implementazione di riferimento che la maggior parte dei programmatori in Python utilizza. Alcune implementazioni di Python non hanno questo limite, in particolare Jython - Python implementato in Java.

Questo ha fatto nascere il mito secondo cui Python non sarebbe in grado di gestire i thread o di trarre vantaggio dall'hardware multi-core. Per quanto riguarda l'elaborazione CPU-bound, la concorrenza può essere gestita utilizzando il modulo `multiprocessing`, che non è limitato dal GIL e può sfruttare al massimo tutti i core disponibili. In questo modo è facile ottenere i miglioramenti attesi nella velocità (circa proporzionali al numero di core). Anche per l'elaborazione I/O-bound possiamo utilizzare il modulo `multiprocessing`, oppure il modulo `threading`, o il modulo `concurrent.futures`. Se utilizziamo il threading per la concorrenza I/O-bound,

solitamente la latenza di rete prevale rispetto al carico aggiuntivo comportato da GIL, che perciò non costituisce un problema significativo nella pratica.

Sfortunatamente gli approcci di basso e medio livello alla concorrenza tendono a generare errori (in qualsiasi linguaggio). Possiamo evitare tali problemi scegliendo di non utilizzare lock espliciti e sfruttando i moduli `queue` e `multiprocessing` di Pyhton, oppure il modulo `concurrent.futures`. Vedremo come ottenere significativi miglioramenti delle prestazioni utilizzando la concorrenza di alto livello nel Capitolo 4.

A volte i programmatori utilizzano C, C++ o altri linguaggi compilati a causa di un altro mito: che Python sarebbe lento. Benché Python sia in generale più lento dei linguaggi compilati, su hardware moderno in realtà è spesso veloce quanto basta per la maggior parte delle applicazioni. E nei casi in cui Python non è veloce a sufficienza, possiamo comunque godere dei benefici di programmare in questo linguaggio e nello stesso tempo di ottenere un'esecuzione più veloce del codice.

Per velocizzare i programmi a lunga esecuzione possiamo utilizzare l'interprete Python PyPy (`http://pypy.org`), che dispone di un compilatore just-in-time in grado di fornire notevoli miglioramenti nelle prestazioni. Un altro modo per migliorare le performance è quello di utilizzare codice che sia eseguito velocemente quasi come il codice C compilato; nel caso dell'elaborazione CPU-bound si possono ottenere incrementi di velocità dell'ordine delle 100 volte. Il modo più facile per ottenere una velocità simile a quella del C è quello di utilizzare moduli Python che siano già scritti in C: per esempio il modulo `array` della libreria standard o il modulo esterno `numpy` per elaborare gli array con incredibile velocità e grande efficienza per quanto riguarda la memoria (anche array multidimensionali con `numpy`). Un altro modo è quello di eseguire una profilatura utilizzando il modulo `cProfile` della libreria standard per scoprire dove si trovano i colli di bottiglia, e poi scrivere codice per cui la velocità è un aspetto critico utilizzando Cython - che in sostanza fornisce una sintassi Python migliorata che viene compilata in C puro per la massima velocità in fase di runtime.

Naturalmente, a volte le funzionalità che ci servono sono già disponibili in una libreria in C o C++, o di un altro linguaggio che utilizza la convenzione di chiamata del C. Nella maggior parte dei casi esiste un modulo Python esterno che fornisce l'accesso alla libreria che ci serve - questi moduli si possono trovare nel Python Package Index (PyPI; `http://pypi.python.org`). Tuttavia, per i rari casi in cui un tale modulo non sia disponibile, si può usare il modulo `ctypes` della libreria standard per accedere alla funzionalità di librerie in C, come può fare il pacchetto esterno Cython.

L’uso di librerie C preesistenti può ridurre notevolmente i tempi di sviluppo, e solitamente consente di ottenere una notevole velocità di elaborazione. Cython e `ctypes` sono entrambi trattati nel Capitolo 5.

La libreria standard di Python fornisce una varietà di moduli per l’elaborazione di rete, dal modulo di basso livello `socket` a quello di livello intermedio `socketserver`, fino a quello di alto livello `xmlrpclib`. Benché l’elaborazione di rete di livello basso e intermedio abbia senso quando si effettua il porting del codice da un altro linguaggio, partendo subito in Python possiamo spesso evitare i dettagli di basso livello e concentrarsi su ciò che le nostre applicazioni di rete devono fare utilizzando moduli di alto livello. Nel Capitolo 6 vedremo come fare ciò utilizzando il modulo `xmlrpclib` della libreria standard e il modulo esterno `RPyC`, potente e facile da usare.

Quasi tutti i programmi devono fornire un’interfaccia utente di qualche tipo affinché sia possibile indicare che cosa fare. I programmi in Python possono essere scritti in modo da supportare interfacce utente a riga di comando, con il modulo `argparse`, e interfacce utente tipo terminale (per esempio su Unix usando il pacchetto esterno `urwid`; `http://excess.org/urwid`). Esistono anche molti ottimi framework web, dal semplice e leggero `bottle` (`http://bottlepy.org`) ai colossi come Django (`http://www.djangoproject.com`) e Pyramid (`http://www.pylonsproject.org`), che possono tutti essere utilizzati per fornire alle applicazioni un’interfaccia web. E naturalmente Python può essere usato per creare applicazioni GUI (*Graphical User Interface*).

Spesso si sente dire che le applicazioni GUI sarebbero destinate a scomparire in favore delle applicazioni web, ma non è ancora successo. In effetti le persone sembrano preferire le applicazioni GUI alle applicazioni web. Per esempio, quando è iniziato il boom degli smartphone all’inizio del ventunesimo secolo, gli utenti preferivano sempre utilizzare una “app” appositamente creata, anziché un browser e una pagina web per attività che svolgevano regolarmente. Esistono molti modi per programmare GUI in Python usando pacchetti esterni. Nel Capitolo 7 vedremo come creare applicazioni GUI moderne usando Tkinter, fornito con la libreria standard di Python.

La maggior parte dei computer moderni, tra cui i notebook e anche gli smartphone dispone di potenti strumenti grafici, spesso di una GPU (*Graphics Processing Unit*) separata in grado di gestire grafica 2D e 3D a livelli davvero notevoli. La maggior parte delle GPU supporta l’API OpenGL, e i programmatori in Python possono accedere a questa API attraverso pacchetti esterni. Nel Capitolo 8, vedremo come utilizzare OpenGL per la grafica 3D.

Questo libro è stato scritto allo scopo di mostrare come scrivere applicazioni Python migliori, che abbiano buone prestazioni e siano di facile manutenzione, oltre che di facile uso. Presuppone una certa conoscenza della programmazione in Python e va inteso come il tipo di libro a cui rivolgersi dopo aver imparato a programmare in Python, sfruttando la documentazione o altri libri - come *Programming in Python 3, Second Edition* (cfr. la bibliografia a fine libro per i dettagli). Questo libro intende fornire idee, ispirazione e tecniche pratiche che aiutino i lettori a raggiungere un livello più alto nella programmazione in Python.

Tutti gli esempi riportati sono stati testati con Python 3.3 (e ove possibile con Python 3.2 e Python 3.1) su Linux, OS X e Windows (nella maggior parte dei casi). Sono disponibili per il download presso il sito web dell'edizione inglese del libro, all'indirizzo `http://www.qtrac.eu/pipbook.html`, e dovrebbero funzionare con tutte le future versioni di Python 3.x.

Come tutti gli altri libri che ho scritto, anche questo ha tratto grande beneficio dai consigli, dall’aiuto e dall’incoraggiamento di altre persone: sono molto grado a tutte.

Nick Coghlan, core developer in Python dal 2005, ha fornito molte critiche costruttive, accompagnandole con idee e porzioni di codice per illustrare strade alternative e migliori. L’aiuto di Nick è stato preziosissimo per tutto il libro, e in particolare per migliorare i primi capitoli.

Doug Hellmann, esperto sviluppatore Python e autore, mi ha inviato molti commenti utili, sia sul progetto iniziale, sia sui vari capitoli. Doug mi ha dato idee ed è stato tanto gentile da scrivere la prefazione.

Due amici - Jasmin Blanchette e Trenton Schulz - entrambi esperti programmatori, con i loro diversi livelli di conoscenza di Python, sono per me ideali rappresentanti di molti dei lettori a cui questo libro si rivolge. I loro commenti sono stati utilissimi per migliorare e chiarire il testo e gli esempi in molti punti.

Sono felice di ringraziare il mio editor, Debra Williams Cauley, che ancora una volta mi ha fornito sostegno e aiuto concreto durante le lavorazioni.

Grazie anche a Elizabeth Ryan, che ha gestito così bene il processo di produzione, e alla correttrice di bozze Anna V. Popick, che ha svolto un eccellente lavoro.

E come sempre grazie a mia moglie, Andrea, per l’amore e il sostegno.

Mark Summerfield è un laureato in informatica con molti anni di esperienza nel settore del software, principalmente come programmatore e responsabile della documentazione. È il proprietario di Qtrac Ltd. (`http://www.qtrac.eu`), dove opera come autore indipendente, editor, consulente e formatore, specializzato nei linguaggi C++, Go e Python e nelle librerie Qt, PyQt e PySide.

Tra gli altri libri di Mark Summerfield:

- *Programming in Go* (2012, ISBN-13: 978-0-321-77463-7)
- *Advanced Qt Programming* (2011, ISBN-13: 978-0-321-63590-7)
- *Programming in Python 3* (prima edizione, 2009, ISBN-13: 978-0-13-712929-4; seconda edizione, 2010, ISBN-13: 978-0-321-68056-3)
- *Rapid GUI Programming with Python and Qt* (2008, ISBN-13: 978-0-13-235418-9)

Tra gli altri libri di Jasmin Blanchette e Mark Summerfield:

- *C++ GUI Programming with Qt 4* (prima edizione, 2006, ISBN-13: 978-0-13-187249-3; seconda edizione, 2008, ISBN-13: 978-0-13-235416-5)
- *C++ GUI Programming with Qt 3* (2004, ISBN-13: 978-0-13-124072-8)

I design pattern creazionali

I *design pattern* (o pattern di progettazione) creazionali riguardano la creazione di oggetti. Normalmente per creare gli oggetti si richiama il loro costruttore (cioè si richiama l'oggetto di classe con opportuni argomenti), ma talvolta serve una maggiore flessibilità nelle modalità di creazione, e proprio qui entrano in gioco i design pattern creazionali.

Per i programmatori in Python alcuni di questi pattern sono piuttosto simili tra loro, e alcuni di essi, come vedremo, in realtà non servono. Il fatto è che i design pattern originali sono stati creati principalmente per il linguaggio C++ e dovevano confrontarsi con alcune delle limitazioni di tale linguaggio, che però Python non ha.

Il pattern Abstract Factory

Il pattern Abstract Factory (letteralmente “fabbrica astratta”) è ideato per situazioni in cui vogliamo creare oggetti complessi costituiti da altri oggetti e gli oggetti composti sono tutti di un’unica particolare “famiglia”.

Per esempio, in un sistema GUI potremmo avere una factory astratta widget con tre factory di sottoclassi concrete: `MacWidgetFactory`, `XfceWidgetFactory` e `WindowsWidgetFactory`.

Tutte queste factory mettono a disposizione metodi per creare gli stessi oggetti (`make_button()`, `make_spinbox()` e così via), ma utilizzando lo stile appropriato per la piattaforma corrispondente. Questo ci consente di creare una funzione generica `create_dialog()` che riceve come argomento un’istanza di factory e produce una finestra di dialogo con il look and feel corrispondente, che può essere di OS X, Xfce o Windows.

Una factory classica

Per illustrare il pattern Abstract Factory esamineremo un programma che produce un semplice diagramma. Utilizzeremo due factory: una per produrre output di testo normale e l’altra per produrre output di tipo SVG (*Scalable Vector Graphics*). Entrambi gli output sono mostrati nella Figura 1.1. La prima versione del programma che esamineremo, `diagram1.py`, mostra il pattern nella sua forma pura, mentre la seconda versione, `diagram2.py`, sfrutta alcune funzionalità specifiche di Python per ottenere un codice leggermente più breve e pulito. Entrambe le versioni producono il medesimo output (tutti gli esempi del libro sono disponibili per il download all’indirizzo `http://www.qtrac.eu/pipbook.html`).

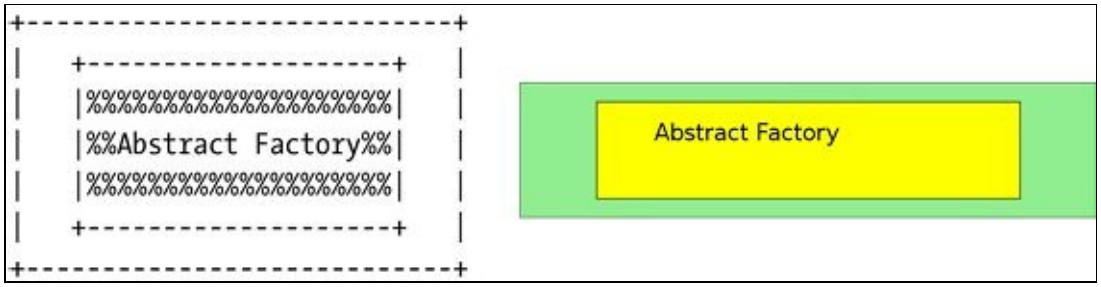


Figura 1.1 I diagrammi in formato di testo e SVG.

Iniziamo a esaminare il codice comune a entrambe le versioni, cominciando con la funzione `main()`.

```
def main():

        ...

        txtDiagram = create_diagram(DiagramFactory()) ❶

        txtDiagram.save(textFilename)

        svgDiagram = create_diagram(SvgDiagramFactory()) ❷

        svgDiagram.save(svgFilename)
```

Per prima cosa creiamo un paio di nomi di file (parte non mostrata), poi creiamo un diagramma utilizzando la factory di testo normale (default) ❶, e poi lo salviamo. Poi creiamo e salviamo lo stesso diagramma, ma questa volta usando una factory SVG (❷).

```
def create_diagram(factory):
     diagram = factory.make_diagram(30, 7)
     rectangle = factory.make_rectangle(4, 1, 22, 5, "yellow")
     text = factory.make_text(7, 3, "Abstract Factory")
     diagram.add(rectangle)
     diagram.add(text)
     return diagram
```

Questa funzione accetta come unico argomento una factory di diagramma e la utilizza per creare il diagramma richiesto. La funzione non conosce il tipo di factory che riceve, basta che supporti la nostra interfaccia per factory di diagramma. Esamineremo i metodi `make_…()` tra breve.

Dopo aver visto come si utilizzano le factory, possiamo esaminare le factory stesse. Riportiamo di seguito la factory di diagramma in formato di testo semplice (che costituisce anche la classe base per le factory):

```
class DiagramFactory:

     def make_diagram(self, width, height):
         return Diagram(width, height)

     def make_rectangle(self, x, y, width, height, fill="white", stroke="black"):
         return Rectangle(x, y, width, height, fill, stroke)

     def make_text(self, x, y, text, fontsize=12):
        return Text(x, y, text, fontsize)
```

Anche se si parla di pattern “astratti”, è normale che un’unica classe serva sia da classe base che fornisce l’interfaccia (l’astrazione) e sia da classe concreta di per sé. In effetti abbiamo seguito proprio questo approccio con la classe `DiagramFactory`.

Riportiamo ora le prime righe della factory del diagramma in formato SVG:

```
class SvgDiagramFactory(DiagramFactory):

     def make_diagram(self, width, height):
         return SvgDiagram(width, height)
     ...
```

L’unica differenza tra i due metodi `make_diagram()` è che `DiagramFactory.make_diagram()` restituisce un oggetto `Diagram`, mentre `SvgDiagramFactory.make_diagram()` restituisce un oggetto `SvgDiagram`. Questo schema vale del resto anche per gli altri due metodi della classe `SvgDiagramFactory` (non mostrati qui).

Vedremo tra breve che le implementazioni delle classi di testo normale `Diagram`, `Rectangle` e `Text` sono molto diverse da quelle delle classi `SvgDiagram`, `SvgRectangle` e `SvgText` - anche se ogni classe fornisce la stessa interfaccia (quindi `Diagram` e `SvgDiagram` hanno gli stessi metodi). Questo significa che non possiamo mescolare classi di famiglie diverse (come `Rectangle` e `SvgText`); si tratta di un vincolo applicato automaticamente dalle classi factory.

Gli oggetti di testo semplice `Diagram` mantengono i loro dati come elenco di elenchi di stringhe di un unico carattere (uno spazio oppure `+`, `|`, `-` e così via). Gli oggetti `Rectangle` e `Text` contengono un elenco di elenchi di stringhe costituite da un singolo carattere destinate a sostituire quelle nel diagramma nella posizione corrispondente (e procedendo verso destra e verso il basso).

```
class Text:

    def __init__(self, x, y, text, fontsize):
        self.x = x self.y = y
        self.rows = [list(text)]
```

Questa è la classe `Text` completa. Per il testo semplice basta rimuovere `fontsize`.

```
class Diagram:
    ...

    def add(self, component):
        for y, row in enumerate(component.rows):
            for x, char in enumerate(row):
                self.diagram[y + component.y][x + component.x] = char
```

Questo è il metodo `Diagram.add()`. Quando lo richiamiamo con un oggetto `Rectangle` o `Text` (il `component`), questo metodo itera su tutti i caratteri nell’elenco di elenchi di stringhe di un solo carattere (`component.rows`) del componente, e sostituisce i caratteri corrispondenti nel diagramma.

Il metodo `Diagram.__init__()` (non mostrato qui) ha già provveduto ad assicurare che il suo `self.diagram` sia un elenco di elenchi di caratteri spazio (per larghezza e altezza specificate) al momento della chiamata di `Diagram(width, height)`.

```
SVG_TEXT = """<text x="{x}" y="{y}" text-anchor="left" \
font-family="sans-serif" font-size="{fontsize}">{text}</text>"""

SVG_SCALE = 20

class SvgText:
```

```
    def __init__(self, x, y, text, fontsize):
        x *= SVG_SCALE
        y *= SVG_SCALE
        fontsize *= SVG_SCALE // 10
        self.svg = SVG_TEXT.format(**locals())
```

Questa è la classe `SvgText` completa, con le due classi su cui si basa (il nostro output SVG è piuttosto grezzo, ma serve allo scopo di illustrare questo pattern di progettazione; potete trovare moduli SVG di terze parti presso il Python Package Index, PyPI, all'indirizzo `http://pypi.python.org`). Tra l'altro, l'uso di `**locals()` ci evita di dover scrivere `SVG_TEXT.format(x=x, y=y, text=text, fontsize=fontsize)`. A partire da Python 3.2 potremmo scrivere `SVG_TEXT.format_map(locals())`, poiché il metodo `str.format_map()` provvede automaticamente al cosiddetto "spacchettamento" della mappa (cfr. il riquadro dedicato proprio a questo argomento più avanti in questo stesso capitolo).

```
class SvgDiagram:
    ...

    def add(self, component):
        self.diagram.append(component.svg)
```

Ogni istanza della classe `SvgDiagram` contiene un elenco di stringhe di `self.diagram`, ognuna delle quali è una porzione di testo SVG. In questo modo è facilissimo aggiungere nuovi componenti (per esempio di tipo `SvgRectangle` o `SvgText`).

Una factory astratta in stile Python

`DiagramFactory` e la sua sottoclasse `SvgDiagramFactory`, e le classi di cui si servono (`Diagram`, `SvgDiagram` e così via) funzionano benissimo e forniscono un'ottima esemplificazione del pattern di progettazione. Nondimeno, la nostra implementazione presenta qualche lacuna. In primo luogo, nessuna delle factory necessita di uno stato di per sé, perciò non è necessario creare istanze di factory. In secondo luogo, il codice per `SvgDiagramFactory` è quasi identico a quello di `DiagramFactory` - l'unica differenza è che restituisce istanze di `SvgText` anziché di `Text`, e così via, e questo appare come un'inutile duplicazione di codice. In terzo luogo, il nostro namespace di primo livello contiene tutte le classi: `DiagramFactory`, `Diagram`, `Rectangle`, `Text` e tutte le equivalenti SVG. Tuttavia, in realtà abbiamo bisogno di accedere soltanto alle due factory. Inoltre, siamo stati costretti a utilizzare un prefisso per i nomi di classi SVG (usando per esempio `SvgRectangle` anziché `Rectangle`) per evitare conflitti di nomi, cosa piuttosto sgradevole (una soluzione per evitare conflitti di nomi sarebbe quella di inserire

ciascuna classe in un proprio modulo, ma questo approccio non risolverebbe il problema della duplicazione di codice).

Nel seguito vedremo come porre rimedio alle lacune citate (il codice è riportato in `diagram2.py`).

La prima modifica che apporteremo consiste nell'annidare le classi `Diagram`, `Rectangle` e `Text` all'interno della classe `DiagramFactory`. Questo significa che ora si accede a tali classi con `DiagramFactory.Diagram` e così via. Possiamo anche annidare le classi equivalenti all'interno della classe `SvgDiagramFactory`; solo ora possiamo assegnare loro gli stessi nomi delle classi di testo normale, poiché non è più possibile un conflitto di nomi. Ecco un esempio: `SvgDiagramFactory.Diagram`. Abbiamo anche annidato le costanti su cui la classe si basa, perciò i nostri nomi di primo livello ora sono `main()`, `create_diagram()`, `DiagramFactory` e `SvgDiagramFactory`.

```
class DiagramFactory:

    @classmethod
    def make_diagramm(Class, width, height)
        return Class.Diagram(width, height)

    @classmethod
    def make_rectangle(Class, x, y, width, height, fill="white", stroke="black"):
        return Class.Rectangle(x, y, width, height, fill, stroke)

    @classmethod
    def make_text(Class, x, y, text, fontsize=12):
        return Class.Text(x, y, text, fontsize)

    ...
```

Qui è riportato l'inizio della nostra nuova classe `DiagramFactory`. I metodi `make_…()` ora sono tutti metodi di classe; questo significa che, quando tali metodi sono richiamati, il primo argomento è la classe (un po' come `self` che viene passato nei metodi normali). Perciò, in questo caso una chiamata di `DiagramFactory.make_text()` significa che come classe viene passata `DiagramFactory`, e viene creato e restituito un oggetto `DiagramFactory.Text`.

Questa modifica comporta anche che la sottoclasse `SvgDiagramFactory` che eredita da `DiagramFactory` non necessita di alcuno dei metodi `make_…()`. Per esempio, se richiamiamo `SvgDiagramFactory.make_rectangle()`, poiché `SvgDiagramFactory` non ha tale metodo, verrà richiamato al suo posto il metodo `DiagramFactory.make_rectangle()` della classe base, ma come `Class` sarà passata `SvgDiagramFactory`. Così verrà creato e restituito un oggetto `SvgDiagramFactory.Rectangle`.

```
def main():
    ...
    txtDiagram = create_diagram(DiagramFactory)
```

```
    txtDiagram.save(textFilename)

    svgDiagram = create_diagram(SvgDiagramFactory)
    svgDiagram.save(svgFilename)
```

Queste modifiche comportano anche il fatto che possiamo semplificare la nostra funzione `main()`, poiché non abbiamo più la necessità di creare istanze di factory.

La parte restante del codice è praticamente identica alla precedente, l'unica differenza è che, poiché ora le costanti e le classi non factory sono annidate all'interno delle factory, per accedervi dobbiamo usare il nome della factory.

```
class SvgDiagramFactory(DiagramFactory):
    ...
    class Text:

        def __init__(self, x, y, text, fontsize):
            x *= SvgDiagramFactory.SVG_SCALE
            fontsize *= SvgDiagramFactory.SVG_SCALE // 10
            self.svg = SvgDiagramFactory.SVG_TEXT.format(**locals())
```

Questa è la classe `Text` annidata in `SvgDiagramFactory` (equivalente alla classe `SvgText` di `diagram1.py`), che mostra come si deve accedere alle costanti annidate.

Il pattern Builder

Il pattern Builder è simile al pattern Abstract Factory nel senso che entrambi sono progettati per creare oggetti complessi composti da altri oggetti. Ciò che distingue il pattern Builder è che non si limita a fornire i metodi per costruire un oggetto complesso, ma contiene anche la rappresentazione dell'intero oggetto.

Questo pattern offre lo stesso livello di "composizione" del pattern Abstract Factory (gli oggetti complessi sono costruiti a partire da uno o più oggetti più semplici), ma è particolarmente adatto ai casi in cui la rappresentazione dell'oggetto complesso deve essere mantenuta separata dagli algoritmi di composizione.

Nel seguito mostriamo un esempio di uso del pattern Builder in un programma in grado di produrre form - che siano form web realizzati con HTML, o form GUI creati con Python e Tkinter. Entrambi i tipi di form operano in modo visuale e supportano l'inserimento di testo, ma i pulsanti non sono funzionanti (tutti gli esempi devono rispettare un equilibrio tra realismo e idoneità all'apprendimento, e di conseguenza alcuni, come quello descritto qui, prevedono soltanto funzionalità di base). I form sono mostrati nella Figura 1.2 e il codice sorgente è presente nel file `formbuilder.py`.

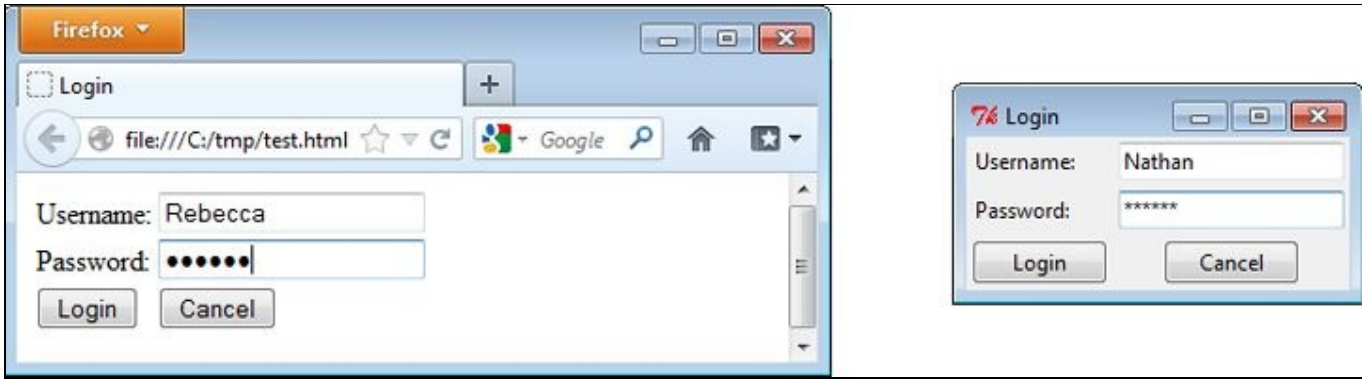


Figura 1.2 I form HTML e Tkinter su Windows.

Iniziamo esaminando il codice necessario per creare ciascun form, partendo con le chiamate di primo livello.

```
htmlForm = create_login_form(HtmlFormBuilder())
with open(htmlFilename, "w", encoding="utf-8") as file:
    file.write(htmlForm)

tkForm = create_login_form(TkFormBuilder())
with open(tkFilename, "w", encoding="utf-8") as file:
    file.write(tkForm)
```

Abbiamo così creato ciascun form e lo abbiamo scritto in un file appropriato. In entrambi i casi abbiamo usato la stessa funzione di creazione (`create_login_form()`), con un opportuno oggetto `builder` per i parametri.

```
def create_login_form(builder):
    builder.add_title("Login")
    builder.add_label("Username", 0, 0, target="username")
    builder.add_entry("username", 0, 1)
    builder.add_label("Password", 1, 0, target="password")
    builder.add_entry("password", 1, 1, kind="password")
    builder.add_button("Login", 2, 0)
    builder.add_button("Cancel", 2, 1)
    return builder.form()
```

Questa funzione consente di creare qualsiasi form HTML o Tkinter a piacere, e ogni altro tipo di form per cui disponiamo di un costruttore adatto. Il metodo `builder.add_title()` è usato per fornire un titolo al form, tutti gli altri sono utilizzati per aggiungere un widget in una posizione di riga e di colonna date.

Sia `HtmlFormBuilder` che `TkFormBuilder` ereditano da una classe base astratta, `AbstractFormBuilder`.

```
class AbstractFormBuilder(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def add_title(self, title):
        self.title = title

    @abc.abstractmethod
    def form(self):
        pass

    @abc.abstractmethod
    def add_label(self, text, row, column, **kwargs):
        pass
    ...
```

Qualsiasi classe che erediti da questa deve implementare tutti i metodi astratti. Abbiamo tralasciato i metodi astratti `add_entry()` e `add_button()` perché, a parte i nomi, sono identici al metodo `add_label()`. Tra l'altro, dobbiamo fare in modo che `AbstractFormBuilder` abbia una metaclasse `abc.ABCMeta` per consentire l'uso del decoratore `@abstractmethod` del modulo `abc` (cfr. il paragrafo del Capitolo 2 dedicato al pattern Decorator per ulteriori informazioni sui decoratori).

Se si assegna a una classe una metaclasse `abc.ABCMeta`, la classe non potrà essere istanziata, perciò dovrà essere usata come classe base astratta. Questo è particolarmente importante per il codice che viene trasferito, per esempio, da C++ o Java, ma comporta un certo sovraccarico di runtime. Tuttavia, molti programmatori in Python utilizzano un approccio più tranquillo: non utilizzano alcuna metaclasse e si limitano a indicare che la classe dovrebbe essere usata come classe base astratta.

```
class HtmlFormBuilder(AbstractFormBuilder):

    def __init _(self):
        self.title = "HtmlFormBuilder"
        self.items = {}

    def add_title(self, title):
        super().add_title(escape(title))
```

```
    def add_label(self, text, row, column, **kwargs):
        self.items[(row, column)] = ('<td><label for="{}">{}:</label></td>'
                .format(kwargs["target"], escape(text)))

    def add_entry(self, variable, row, column, **kwargs):
        html = """<td><input name="{}" type="{}" /></td>""".format(
                variable, kwargs.get("kind", "text"))
        self.items[(row, column)] = html
    ...
```

Questo è l'inizio della classe `HtmlFormBuilder`. Forniamo un titolo di default per il caso in cui il form sia creato senza titolo. Tutti i widget del form sono memorizzati in un dizionario `items` che utilizza chiavi di riga e colonna e il codice HTML dei widget.

Dobbiamo reimplementare il metodo `add_title()` poiché è astratto, ma dato che la versione astratta ha un'implementazione, possiamo semplicemente richiamare quest'ultima. In questo caso dobbiamo pre-elaborare il titolo usando la funzione `html.escape()` (o la funzione `xml.sax.saxutil.escape()` in Python 3.2 e versioni precedenti).

SPACCHETTAMENTO DI SEQUENZE E MAPPE

"Spacchettare" in gergo significa estrarre tutti gli elementi di una sequenza o mappa. Un caso semplice è quello in cui si vuole estrarre il primo o i primi elementi e poi il resto; per esempio:

```
first, second, *rest = sequence
```

In questo caso assumiamo che `sequence` abbia almeno tre elementi: `first == sequence[0]`, `second == sequence[1]` e `rest == sequence[2:]`.

Le applicazioni più comuni si presentano nelle chiamate di funzioni. Se abbiamo una funzione che richiede un certo numero di argomenti posizionali, o particolari argomenti parole chiave, possiamo usare lo spacchettamento per fornire tali argomenti. Per esempio:

```
args = (600, 900)
kwargs = dict(copies=2, collate=False)
print_setup(*args, **kwargs)
```

La funzione `print_setup()` richiede due argomenti posizionali (`width` e `height`) e accetta fino a due argomenti parole chiave opzionali (`copies` e `collate`). Anziché passare i valori direttamente, abbiamo creato una tupla `args` e un `dict kwargs`, e abbiamo usato lo scompattamento di sequenza (`*args`) e di mappa (`**kwargs`) per passare gli argomenti. L'effetto è identico a quello che avremmo ottenuto scrivendo `print_setup(600, 900, copies=2, collate=False)`.

Un'altra applicazione è quella in cui si creano funzioni che possono accettare qualsiasi numero di argomenti posizionali, o qualsiasi numero di argomenti parole chiave, o qualsiasi numero di entrambi. Per esempio:

```
def print_args(*args, **kwargs):
    print(args.__class__.__name__, args,
          kwargs.__class__.__name__, kwargs)

print_args()  # stampa: tuple () dict {}
print_args(1, 2, 3, a="A")  # stampa: tuple (1, 2, 3) dict {'a': 'A'}
```

La funzione `print_args()` accetta qualsiasi numero di argomenti posizionali o parole chiave. Al suo interno, `args` è di tipo `tuple` e `kwargs` è di tipo `dict`. Se volessimo passare questi argomenti a una funzione richiamata all'interno della funzione `print_args()`, potremmo naturalmente usare lo scompattamento nella chiamata (per esempio `function(*args, **kwargs)`). Un altro impiego

comune dello scompattamento di mappe è quello in cui si richiama il metodo `str.format()`, per esempio `s.format(**locals())`, anziché digitare manualmente tutti gli argomenti del tipo chiave=valore (per un esempio potete fare riferimento alla precedente discussione del metodo `SvgText__init__()`).

Il metodo `add_button()` (non mostrato) è simile come struttura agli altri metodi `add_…()`.

```
    def form(self):
        html = ["<!doctype html>\n<html><head><title>{}</title></head>"
                "<body>".format(self.title), '<form><table border="0">']
        thisRow = None
        for key, value in sorted(self.items.items()):
            row, column = key
            if thisRow is None:
                html.append(" <tr>")
            elif thisRow != row:
                html.append(" </tr>\n <tr>")
            thisRow = row
            html.append("    " + value)
        html.append(" </tr>\n</table></form></body></html>")
        return "\n".join(html)
```

Il metodo `HtmlFormBuilder.form()` crea una pagina HTML costituita da un `<form>`, all'interno del quale vi è una `<table>`, all'interno della quale vi sono righe e colonne di widget. Una volta che tutti i pezzi sono stati aggiunti all'elenco `html`, quest'ultimo viene restituito come singola stringa (con caratteri di nuova riga utilizzati come separatori per rendere il codice più leggibile).

```
class TkFormBuilder(AbstractFormBuilder):

    def __init__(self):
        self.title = "TkFormBuilder"
        self.statements = []

    def add_title(self, title):
        super().add_title(title)

    def add_label(self, text, row, column, **kwargs):
        name = self._canonicalize(text)
        create = """self.{}Label = ttk.Label(self, text="{}:")""".format(name, text)
        layout = """self.{}Label.grid(row={}, column={}, sticky=tk.W, \padx="0.75m",
                pady="0.75m")""".format(name, row, column)
        self.statements.extend((create, layout))

    ...
    def form(self):
        return TkFormBuilder.TEMPLATE.format(title=self.title,
        name=self._canonicalize(self.title, False),
        statements="\n        ".join(self.statements))
```

Questo è un estratto della classe `TkFormBuilder`. Memorizziamo i widget del form come elenco di istruzioni (cioè come stringhe di codice Python), con due istruzioni per widget.

La struttura del metodo `add_label()` è usata anche dai metodi `add_entry()` e `add_button()` (non riportati). Questi metodi ricevono un nome del widget espresso in forma canonica e poi creano due stringhe: `create`, con il codice per creare il widget, e `layout`,

con il codice per disporre il widget nel form. Infine, i metodi aggiungono le due stringhe all'elenco di istruzioni.

Il metodo `form()` è molto semplice: restituisce una stringa `TEMPLATE` parametrizzata con titolo e istruzioni.

```
    TEMPLATE = """#!/usr/bin/env python3
import tkinter as tk
import tkinter.ttk as ttk

class {name}Form(tk.Toplevel): ❶

    def __init__(self, master):
        super().__init__(master)
        self.withdraw()    # nasconde fino a quando è pronto a visualizzare
                             self.title("{title}") ❷
        {statements} ❸
        self.bind("<Escape>", lambda *args: self.destroy())
        self.deiconify()    # visualizza quando i widget sono creati e posizionati
        if self.winfo_viewable():
            self.transient(master)
        self.wait_visibility()
        self.grab_set()
        self.wait_window(self)

if __name__== "__main__":
    application = tk.Tk()
    window = {name}Form(application) ❹
    application.protocol("WM_DELETE_WINDOW", application.quit)
    application.mainloop()
"""
```

Il nome di classe univoco assegnato al form è basato sul titolo (per esempio `LoginForm`, ❶; ❹). Il titolo della finestra (per esempio "Login", ❷) è impostato subito, seguono le istruzioni per creare e disporre i widget del form (❸).

Il codice Python prodotto dall'uso del template può essere eseguito in forma autonoma grazie al blocco `if __name__ ...` alla fine.

```
    def _canonicalize(self, text, startLower=True):
        text = re.sub(r"\W+", "", text)
        if text[0].isdigit():
            return "_" + text
        return text if not startLower else text[0].lower() + text[1:]
```

Il codice del metodo `_canonicalize()` è incluso per completezza. Tra l'altro, anche se sembra che venga creata una regex nuova ogni volta che la funzione viene richiamata, nella pratica Python mantiene una cache interna piuttosto ampia di regex compilate, perciò, per la seconda chiamata e le successive, Python si limita a cercare la regex anziché ricompilarla.

NOTA

In questo libro si assume che il lettore conosca le espressioni regolari o regex e il modulo `re` di Python. I lettori che abbiano bisogno di apprendere questi argomenti possono consultare gratuitamente il PDF del Capitolo 13 del volume in lingua inglese *Programming in Python 3,*

Second Edition, capitolo dedicato proprio alle espressioni regolari. Cfr. `http://www.qtrac.eu/py3book.html`.

Il pattern Factory Method

Questo pattern è destinato all'uso quando si vuole scegliere quali classi istanziare al momento in cui è richiesto un oggetto. È utile di per sé, ma può essere sfruttato ancora meglio nei casi in cui non si conosce la classe in anticipo (per esempio quando la classe da usare dipende dal contenuto di un file o dall'input dell'utente).

In questo paragrafo esaminiamo un programma utilizzabile per creare una scacchiera. L'output è mostrato nella Figura 1.3 e le quattro varianti del codice sorgente sono riportate nei file `gameboard1.py`…`gameboard4.py`. Sfortunatamente, il supporto UTF-8 delle console Windows è abbastanza scarso, molti caratteri non sono disponibili anche se si utilizza la code page 65001. Perciò nel caso di Windows i programmi scrivono l'output su un file temporaneo e stampano il nome di file utilizzato. Pare che nessuno dei font Windows a spaziatura fissa standard disponga dei caratteri con i pezzi degli scacchi o della dama, presenti invece in molti font a spaziatura variabile. Il font DejaVu Sans, gratuito e open source, li contiene tutti (`dejavu-fonts.org`).

Vogliamo disporre di una classe astratta di tipo scacchiera di cui sia possibile creare sottoclassi per ottenere scacchiere specifiche per il gioco (scacchi, dama e così via). Ogni sottoclasse si riempirà con la disposizione iniziale dei pezzi. Ogni tipo unico di pezzo deve appartenere a una propria classe (per esempio `BlackDraught`, `WhiteDraught` per i pezzi della dama, `BlackChessBishop`, `WhiteChessKnight` e così via per i vari pezzi degli scacchi). Tra l'altro, per i singoli pezzi abbiamo usato nomi di classe come `WhiteDraught` anziché, per esempio, `WhiteChecker`, per mantenere una corrispondenza con i nomi inglesi utilizzati in Unicode per i caratteri corrispondenti.

Iniziamo esaminando il codice di primo livello che istanzia e stampa la scacchiera/damiera. Poi esamineremo le classi per il tavolo da gioco e alcuni dei pezzi, iniziando con quelle codificate esplicitamente. Poi vedremo alcune varianti che consentono di evitare la codifica esplicita e usare meno righe di codice.

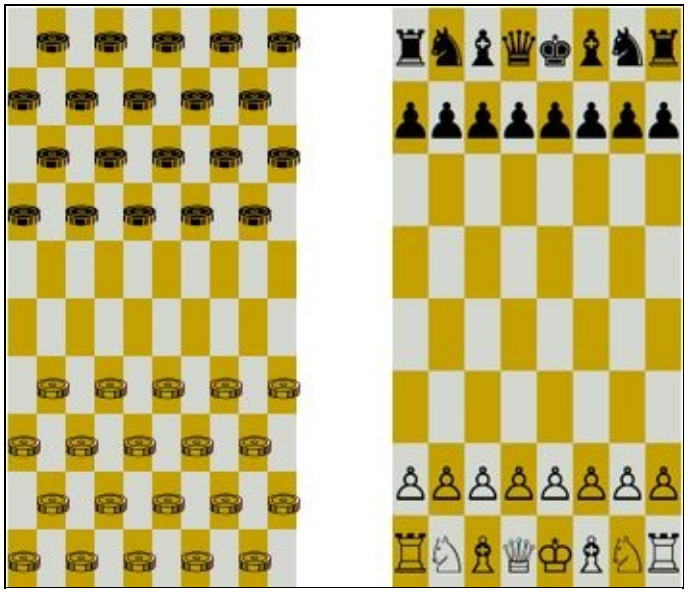


Figura 1.3 La damiera e la scacchiera su una console Linux.

```
def main():
    checkers = CheckersBoard()
    print(checkers)

    chess = ChessBoard()
    print(chess)
```

Questa funzione è comune a tutte le versioni del programma; crea ciascun tipo di tavolo da gioco e lo stampa sulla console, utilizzando il metodo `__str__()` di `AbstractBoard` per convertire la rappresentazione interna del tavolo in una stringa.

```
BLACK, WHITE = ("BLACK", "WHITE")

class AbstractBoard:

    def __init__(self, rows, columns):
        self.board = [[None for _ in range(columns)] for _ in range(rows)]
        self.populate_board()

    def populate_board(self):
        raise NotImplementedError()

    def __str__(self):
        squares = []
        for y, row in enumerate(self.board):
            for x, piece in enumerate(row):
                square = console(piece, BLACK if (y + x) % 2 else WHITE)
                squares.append(square)
            squares.append("\n")
    return "".join(squares)
```

Le costanti `BLACK` e `WHITE` sono usate qui per indicare il colore di sfondo di ciascuna casella. Nelle successive varianti saranno usate anche per indicare il colore di ciascun pezzo. Questa classe è ripresa da `gameboard1.py`, ma è identica in tutte le versioni.

Normalmente si specificano le costanti scrivendo: `BLACK, WHITE = range(2)`. Tuttavia, l’uso di stringhe risulta molto utile per il debugging in caso di messaggi di errore, e

non dovrebbe comportare problemi di velocità rispetto all'uso di interi grazie al meccanismo di controlli intelligente di Python.

Il tavolo da gioco è rappresentato da un elenco di righe di stringhe monocarattere, o `None` per le caselle vuote. La funzione `console()` (non mostrata qui, ma presente nel codice sorgente) restituisce una stringa che rappresenta il dato pezzo sul dato colore di sfondo. Su sistemi Unix-like questa stringa include codici di escape per colorare lo sfondo.

Avremmo potuto impostare `AbstractBoard` come classe formalmente astratta fornendole una metaclasse `abc.ABCMeta` (come abbiamo fatto in precedenza per `AbstractFormBuilder`). Tuttavia, in questo caso abbiamo scelto un approccio diverso, limitandoci a sollevare un'eccezione `NotImplementedError` per qualsiasi metodo che vogliamo sia reimplementato dalle sottoclassi.

```
class CheckersBoard(AbstractBoard):

    def __init__(self):
        super().__init__(10, 10)

    def populate_board(self):
        for x in range(0, 9, 2):
            for row in range(4):
                column = x + ((row + 1) % 2)
                self.board[row][column] = BlackDraught()
                self.board[row + 6][column] = WhiteDraught()
```

Questa sottoclasse è usata per creare una rappresentazione di una damiera internazionale 10 × 10. Il metodo `populate_board()` non è un metodo factory, poiché utilizza classi codificate in modo esplicito; è mostrato in questa forma come primo passo sulla strada che ci porterà a trasformarlo in un metodo factory.

```
class ChessBoard(AbstractBoard):

    def __init__(self):
        super().__init__(8, 8)

    def populate_board(self):
        self.board[0][0] = BlackChessRook()
        self.board[0][1] = BlackChessKnight()
        ...
        self.board[7][7] = WhiteChessRook()
        for column in range(8):
            self.board[1][column] = BlackChessPawn()
            self.board[6][column] = WhiteChessPawn()
```

Questa versione del metodo `populate_board()` di `ChessBoard`, esattamente come quella di `CheckersBoard`, non è factory, ma illustra come viene riempita la scacchiera.

```
class Piece(str):

     __slots__= ()
```

Questa classe serve da classe base per i pezzi. Avremmo potuto utilizzare semplicemente `str`, ma questo non ci avrebbe consentito di determinare se un oggetto sia un pezzo, per esempio usando `isinstance(x, Piece)`. L'uso di `__slots__ = ()` ci garantisce che le istanze non abbiano dati (tema che discuteremo più avanti nel paragrafo dedicato al pattern Flyweight del Capitolo 2).

```
class BlackDraught(Piece):

      __slots__= ()

    def __new__(Class):
        return super().__new__(Class, "\N{black draughts man}")

class WhiteChessKing(Piece):

     __slots__= ()

    def __new__(Class):
        return super().__new__(Class, "\N{white chess king}")
```

Queste due classi sono modelli per il pattern usato per tutte le classi dei pezzi. Ognuna è una sottoclasse immutabile di `Piece` (a sua volta sottoclasse di `str`) che è inizializzata con una stringa monocarattere contenente il carattere Unicode che rappresenta il pezzo corrispondente. In totale ci sono quattordici sottoclassi come queste, ognuna delle quale differisce dalle altre soltanto per il proprio nome e la stringa che contiene; naturalmente sarebbe meglio eliminare tutta questa duplicazione di codice.

```
    def populate_board(self):
        for x in range(0, 9, 2):
            for y in range(4):
                column = x + ((y + 1) % 2)
                    for row, color in ((y, "black"), (y + 6, "white")):
                        self.board[row][column] = create_piece("draught", color)
```

Questa nuova versione del metodo `CheckersBoard.populate_board()`, tratta dal file `gameboard2.py`, è un metodo factory, poiché dipende da una nuova funzione factory `create_piece()` anziché da classi codificate esplicitamente.

La funzione `create_piece()` restituisce un oggetto del tipo appropriato (per esempio `BlackDraught` o `WhiteDraught`) in base agli argomenti.

Questa versione del programma dispone di un analogo metodo denominato `ChessBoard.populate_board()`, non mostrato qui, che utilizza anch'esso colori e nomi di pezzi e la stessa funzione `create_piece()`.

```
def create_piece(kind, color):
    if kind == "draught":
        return eval("{}{}()".format(color.title(), kind.title()))
    return eval("{}Chess{}()".format(color.title(), kind.title()))
```

Questa funzione factory utilizza la funzione integrata `eval()` per creare istanze di classe. Per esempio, se gli argomenti sono `"knight"` e `"black"`, la stringa da sottoporre a `eval()` sarà `"BlackChessKnight()"`. Questo approccio funziona bene, ma è potenzialmente rischioso perché potrebbe consentire di sottoporre a `eval()` praticamente qualsiasi cosa. Vedremo più avanti una soluzione che prevede l'utilizzo della funzione integrata `type()`.

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
           name = name[:-4]
    exec("""\
class {}(Piece):

    __slots__ = ()

    def __new__(Class):
        return super().__new__(Class, "{}")""".format(name, char))
```

Anziché scrivere il codice per quattordici classi molto simili, in questo caso creiamo tutte le classi che ci servono con un unico blocco di codice.

La funzione `itertools.chain()` riceve uno o più iterabili e restituisce un singolo iterabile che itera sul primo iterabile passato, poi il secondo e così via. In questo caso abbiamo fornito due iterabili: il primo è una coppia di codici Unicode che punta ai pezzi neri e bianchi della dama, e il secondo è un oggetto `range` (un generatore, in effetti) per i pezzi neri e bianchi degli scacchi.

Per ciascun codice creiamo una stringa monocarattere (per esempio "♞") e poi un nome di classe basato sul nome Unicode del carattere (per esempio, "black chess knight", il cavallo nero, diventa `BlackChessKnight`).

Una volta ottenuti il carattere e il nome utilizziamo `exec()` per creare la classe che ci serve. Questo blocco di codice richiede poco più di dieci righe, contro le centinaia che servirebbero per creare tutte le classi una per una.

Sfortunatamente, tuttavia, l'uso di `exec()` è potenzialmente ancora più rischioso di `eval()`, perciò dobbiamo trovare una strada migliore.

```
DRAUGHT, PAWN, ROOK, KNIGHT, BISHOP, KING, QUEEN = ("DRAUGHT", "PAWN",
        "ROOK", "KNIGHT", "BISHOP", "KING", "QUEEN")

class CheckersBoard(AbstractBoard):
    ...

    def populate_board(self):
        for x in range(0, 9, 2):
            for y in range(4):
                column = x + ((y + 1) % 2)
                    for row, color in ((y, BLACK), (y + 6, WHITE)):
                        self.board[row][column] = self.create_piece(DRAUGHT, color)
```

Questo metodo `CheckersBoard.populate_board()` è tratto da `gameboard3.py`. Differisce dalla versione precedente perché il pezzo e il colore sono entrambi specificati usando costanti anziché letterali stringa (per ridurre i potenziali errori) e perché utilizza un nuovo metodo factory `create_piece()` per creare ciascun pezzo.

Un'implementazione alternativa di `CheckersBoard.populate_board()` è fornita nel file `gameboard4.py`; tale versione, non mostrata qui, utilizza una sottile combinazione di una list comprehension (descrizione di lista) e una coppia di funzioni `itertools`.

```
class AbstractBoard:

     __classForPiece = {(DRAUGHT, BLACK): BlackDraught,
                        (PAWN, BLACK): BlackChessPawn,
                        ...
                        (QUEEN, WHITE): WhiteChessQueen}
    ...
    def create_piece(self, kind, color):
        return AbstractBoard.__classForPiece[kind, color]()
```

Questa versione di `create_piece()`, anch'essa tratta da `gameboard3.py`, naturalmente, è un metodo di `AbstractBoard` ereditato da `CheckersBoard` e `ChessBoard`. Richiede due costanti e le cerca all'interno di un dizionario statico (a livello di classe) le cui chiavi sono coppie (tipo pezzo, colore) e i cui valori sono oggetti classe. Il valore ricercato, una classe, è immediatamente richiamato usando l'operatore di chiamata `()`, e viene restituita l'istanza di pezzo risultante.

Le classi del dizionario sarebbero potute essere codificate una per una (come in `gameboard1.py`) o create dinamicamente ma con qualche rischio (come in `gameboard2.py`). Ma per `gameboard3.py` le abbiamo create dinamicamente e in modo sicuro, senza usare `eval()` O `exec()`.

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
    name = name[:-4]
    new = make_new_method(char)
    Class = type(name, (Piece,), dict(__slots__=(), __new__=new))
    setattr(sys.modules[__name__], name, Class)  # Si può fare meglio!
```

Questo codice ha la stessa struttura complessiva del codice mostrato in precedenza per creare le quattordici sottoclassi di pezzi che servono al programma. Tuttavia, questa volta anziché usare `eval()` ed `exec()` abbiamo scelto un approccio più sicuro.

Una volta ottenuti carattere e nome creiamo una nuova funzione `new()` richiamando una funzione personalizzata `make_new_method()`. Poi creiamo una nuova classe utilizzando la funzione integrata `type()`.

Per creare una classe in questo modo dobbiamo passare il nome del tipo, una tupla delle sue classi base (in questo caso ce n'è solo una, `Piece`) e un dizionario degli attributi della classe. In questo caso abbiamo impostato l'attributo `__slots__` a una tupla vuota (per fare in modo che le istanze di classe non abbiano un `__dict__` privato, che non serve) e l'attributo metodo `__new__` alla funzione `new()` appena creata.

Infine, usiamo la funzione integrata `setattr()` per aggiungere al modulo corrente (`sys.modules[__name__]`) la nuova classe appena creata (`Class`) come attributo di nome `name` (per esempio `"WhiteChessPawn"`). In `gameboard4.py`, abbiamo scritto l'ultima riga di questa porzione di codice in un modo più elegante:

```
    globals()[name] = Class
```

In questo caso abbiamo recuperato un riferimento al `dict` di globali e abbiamo aggiunto una nuova voce la cui chiave è il nome contenuto in `name` e il cui valore è la `Class` appena creata. Il risultato è identico a quello ottenuto con la riga di `setattr()`in `gameboard3.py`.

```
def make_new_method(char):  # Necessario per creare un metodo nuovo ogni volta
    def new(Class):         # Non si può usare super() o super(Piece, Class)
        return Piece.__new__(Class, char)
    return new
```

Questa funzione è usata per creare una funzione `new()` che diventerà un metodo `__new__()` della classe. Non possiamo usare una chiamata di `super()` perché al momento in cui viene creata la funzione `new()` non esiste un contesto di classe a cui la funzione `super()` possa accedere.

Notate che la classe `Piece` descritta in precedenza non dispone di un metodo `__new__()`, ma la sua classe base (`str`) invece sì, perciò è quello il metodo che sarà effettivamente richiamato.

Tra l'altro, la riga `new = make_new_method(char)` del codice precedente e la funzione `make_new_method()` appena mostrata possono essere entrambe cancellate, purché la riga che richiama la funzione `make_new_method()` sia sostituita da quanto segue:

```
    new = (lambda char: lambda Class: Piece.__new__(Class, char))(char)
    new.__name__= "__new__"
```

In questo caso creiamo una funzione che crea una funzione e immediatamente richiama la funzione più esterna parametrizzata da `char` per resstituire una funzione `new()`. Questo codice è usato in `gameboard4.py`.

Tutte le funzioni `lambda` sono chiamate appunto “`lambda`”, cosa poco utile per il debugging. Perciò, in questo caso assgniamo esplicitamente alla funzione il nome che dovrebbe avere, una volta creata.

```
    def populate_board(self):

        for row, color in ((0, BLACK), (7, WHITE)):
            for columns, kind in (((0, 7), ROOK), ((1, 6), KNIGHT),
                    ((2, 5), BISHOP), ((3,), QUEEN), ((4,), KING)):
                for column in columns:
                    self.board[row][column] = self.create_piece(kind, color)

        for column in range(8):
            for row, color in ((1, BLACK), (6, WHITE)):
                self.board[row][column] = self.create_piece(PAWN, color)
```

Per completezza riportiamo di seguito il metodo `ChessBoard.populate_board()` tratto da `gameboard3.py` (e `gameboard4.py`). Su basa su costanti per colore e pezzo (che potrebbero essere fornite da un file o provenire dalla selezione di opzioni di menu, anziché essere codificate esplicitamente). Nella versione di `gameboard3.py` utilizza la funzione factory `create_piece()` mostrata in precedenza, ma per `gameboard4.py` abbiamo usato la variante finale di `create_piece()`.

```
def create_piece(kind, color):
    color = "White" if color == WHITE else "Black"
    name = {DRAUGHT: "Draught", PAWN: "ChessPawn", ROOK: "ChessRook",
            KNIGHT: "ChessKnight", BISHOP: "ChessBishop",
            KING: "ChessKing", QUEEN: "ChessQueen"}[kind]
    return globals()[color + name]()
```

Questa è la versione della funzione factory `create_piece()` in `gameboard4.py`. Utilizza le stesse costanti della versione di `gameboard3.py`, ma anziché mantenere un dizionario di oggetti di classe, trova dinamicamente la classe interessata nel dizionario restituito dalla funzione integrata `globals()`.

L’oggetto classe cercato viene immediatamente richiamato e viene restituita l’istanza del pezzo risultante.

Il pattern Prototype

Questo pattern è usato per creare nuovi oggetti creando un oggetto originale e poi modificando il clone.

Come abbiamo già visto, in particolare nel paragrafo precedente, Python supporta un'ampia varietà di modi per creare nuovi oggetti, anche quando i loro tipi sono noti soltanto al momento del runtime, e anche se conosciamo solo i nomi dei tipi.

```
class Point:

    __slots__ = ("x", "y")

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Data questa tipica classe `Point`, riportiamo sette modi per creare nuovi punti:

```
def make_object(Class, *args, **kwargs):
    return Class(*args, **kwargs)

point1 = Point(1, 2)
point2 = eval("{}({}, {})".format("Point", 2, 4))  # Risky
point3 = getattr(sys.modules[__name__], "Point")(3, 6)
point4 = globals()["Point"](4, 8)
point5 = make_object(Point, 5, 10)
point6 = copy.deepcopy(point5)
point6.x = 6
point6.y = 12
point7 = point1.__class__(7, 14)  # Avremmo potuto usare uno qualsiasi da point1 a point6
```

Il punto `point1` è creato in modo convenzionale (e statico) usando l'oggetto di classe `Point` come costruttore (in termini rigorosi un metodo `__init__()` è un inizializzatore e un metodo `__new__()` è un costruttore, tuttavia, poiché quasi sempre utilizziamo `__init__()` e raramente utilizziamo `__new__()`, parleremo in entrambi i casi di "costruttori"). Tutti gli altri punti vengono creati dinamicamente. I punti `point2`, `point3` e `point4` sono parametrizzati con il nome di classe. Come si vede bene dalla creazione del punto `point3` (e del punto `point4`), non è necessario usare una pericolosa `eval()` per creare istanze (come abbiamo fatto per `point2`). La creazione di `point4` è del tutto simile a quella di `point3`, ma usa una sintassi più elegante sfruttando la funzione integrata `globals()`. Il punto `point5` è creato usando una funzione generica `make_object()` che accetta un oggetto di classe e i relativi argomenti. Il punto `point6` è creato usando il classico approccio a prototipo: prima cloniamo un oggetto esistente, poi lo inizializziamo o configuriamo. Il punto `point7` è creato usando l'oggetto di classe `point1` più nuovi argomenti.

Il punto `point6` mostra che Python offre il supporto interno per la prototipazione usando la funzione `copy.deepcopy()`. Tuttavia, il punto `point7` mostra che Python può fare di meglio: anziché richiedere la clonazione di un oggetto esistente e la modifica del clone, Python ci fornisce l’accesso a qualsiasi oggetto di classe, in modo che possiamo creare un nuovo oggetto direttamente e in maniera molto più efficiente rispetto a quanto sia possibile con la clonazione.

Il pattern Singleton

Questo pattern è usato quando serve una classe che abbia una singola istanza, che sia l'unica e sola istanza a cui si accede dal programma.

Con alcuni linguaggi orientati agli oggetti, creare un singleton è particolarmente complicato, ma non è il caso di Python. Il Python Cookbook (`code.activestate.com/recipes/langs/python/`) fornisce una classe `Singleton` facile da usare e da cui qualsiasi classe può ereditare per divenire un singleton, oltre a una classe `Borg` che raggiunge lo stesso scopo in un modo diverso.

Tuttavia, il modo più semplice per ottenere la funzionalità dei singleton in Python è quello di creare un modulo con lo stato globale mantenuto in variabili private e accessibile mediante funzioni pubbliche. Nel Capitolo 7 vedremo un esempio `currency` in cui ci servirà una funzione che restituisca un dizionario di tassi di valuta (nome e valore del tasso di conversione). Potremmo voler richiamare la funzione più volte, ma nella maggior parte dei casi vogliamo che i tassi siano recuperati una volta sola. Possiamo ottenere questo scopo utilizzando il pattern Singleton.

```
_URL = "http://www.bankofcanada.ca/stats/assets/csv/fx-seven-day.csv"

def get(refresh=False):
    if refresh:
        get.rates = {}
    if get.rates:
        return get.rates
    with urllib.request.urlopen(_URL) as file:
        for line in file:
            line = line.rstrip().decode("utf-8")
            if not line or line.startswith(("#", "Date")):
                continue
            name, currency, *rest = re.split(r"\s*,\s*", line)
            key = "{} ({})".format(name, currency)
            try:
                get.rates[key] = float(rest[-1])
            except ValueError as err:
                print("error {}: {}".format(err, line))
    return get.rates
get.rates = {}
```

Questo è il codice per il modulo `currency/Rates.py` (come di consueto abbiamo escluso le varie importazioni). In questo caso creiamo un dizionario `rates` come attributo della funzione `Rates.get()`, che è il nostro valore privato. Quando viene richiamata per la prima volta la funzione pubblica `get()` (o se è richiamata con `refresh=True`), scarichiamo i tassi di conversione aggiornati; altrimenti restituiamo semplicemente i tassi scaricati l'ultima volta. Non è necessaria una classe, ma abbiamo un valore dati singleton - i tassi - e potremmo facilmente aggiungerne altri.

Tutti i pattern creazionali sono facili da implementare in Python. Il pattern Singleton può essere implementato direttamente usando un modulo, e il pattern

Prototype è ridondante (anche se comunque utilizzabile con il modulo `copy`), poiché Python fornisce accesso dinamico a oggetti di classe. I pattern creazionali più utili per Python sono Factory e Builder, che possono essere implementati in vari modi.

Una volta creati gli oggetti di base, spesso abbiamo la necessità di creare oggetti più complessi mettendo insieme o adattando altri oggetti. Vedremo come fare nel prossimo capitolo.

I design pattern strutturali

I design pattern strutturali si occupano principalmente di come gli oggetti possono essere composti tra loro per creare nuovi oggetti “più grandi”. Al centro dell’attenzione ci sono in particolare tre temi: adattare le interfacce, aggiungere le funzionalità e gestire collezioni di oggetti.

Il pattern Adapter

Il pattern Adapter in pratica è una tecnica per adattare un'interfaccia in modo che una classe possa usarne un altra (con un'interfaccia incompatibile) senza la necessità di apportare alcuna modifica alle classi in questione. Questo è utile per esempio quando vogliamo usare una classe che non può essere modificata, in un contesto per cui non è stata progettata in origine.

Supponiamo di avere una semplice classe `Page` che possa essere usata per eseguire il rendering di una pagina dati il titolo, i paragrafi di testo e un'istanza di una classe renderer (il codice di questo paragrafo è interamente tratto dall'esempio `render1.py`).

```
class Page:

    def __init__(self, title, renderer):
        if not isinstance(renderer, Renderer):
            raise TypeError("Expected object of type
                        Renderer, got {}".
                format(type(renderer).__name__))
        self.title = title
        self.renderer = renderer
        self.paragraphs = []
    def add_paragraph(self, paragraph):
        self.paragraphs.append(paragraph)

    def render(self):
        self.renderer.header(self.title)
        for paragraph in self.paragraphs:
            self.renderer.paragraph(paragraph)
        self.renderer.footer()
```

La classe `Page` non conosce la classe renderer e non se ne occupa, al di là del fatto che fornisce l'interfaccia per il rendering della pagina, cioè i tre metodi `header(str)`, `paragraph(str)` e `footer()`.

Vogliamo assicurarci che il renderer passato sia un'istanza di `Renderer`. Una soluzione semplice, ma poco soddisfacente, è questa: `assert isinstance(renderer, Renderer)`. Ci sono però due punti deboli. In primo luogo, viene generato un errore `AssertionError` anziché il previsto e più specifico `TypeError`. In secondo luogo, se l'utente esegue il programma con l'opzione `-O` ("ottimizza"), l'`assert` sarà ignorato e l'utente riceverà alla fine un `AttributeError` generato in un secondo tempo, all'interno del metodo `render()`. L'istruzione `if not isinstance(...)` usata nel codice genera correttamente un `TypeError` e opera indipendentemente dall'opzione `-O`.

Un problema apparente di questo approccio è che sembrerebbe che dobbiamo impostare tutti i renderer come sottoclassi di una classe base `Renderer`. Sarebbe certamente così se programmassimo in C++, e anche in Python potremmo creare tale classe base, tuttavia, il modulo `abc` (*abstract base class*) di Python ci offre

un’alternativa più flessibile che combina il vantaggio di una classe base astratta con la flessibilità del *duck typing*. Ciò significa che possiamo creare oggetti con la garanzia che soddisfino una particolare interfaccia (cioè che abbiano un’API specifica) ma senza la necessità che siano sottoclassi di una particolare classe base.

```
class Renderer(metaclass=abc.ABCMeta):

    @classmethod
    def __subclasshook__(Class, Subclass):
        if Class is Renderer:
            attributes = collections.ChainMap(*(Superclass.__dict__
                for Superclass in Subclass.__mro__))
        methods = ("header", "paragraph", "footer")
        if all(method in attributes for method in methods):
            return True
    return NotImplemented
```

La classe `Renderer` reimplementa il metodo speciale `__subclasshook__()`. Tale metodo è usato dalla funzione integrata `isinstance()` per determinare se l’oggetto fornito come primo argomento è una sottoclasse della classe (o di una qualsiasi tupla di classi) passata come secondo argomento.

Il codice è abbastanza sofisticato (e specifico di Python 3.3) perché utilizza la classe `collections.ChainMap()` (l’esempio `render1.py` e il modulo `Qtrac.py` usato da `render2.py` includono entrambi codice specifico di Python 3.3 e codice che funziona anche con versioni precedenti di Python 3). La spiegazione è riportata nel seguito, ma non è così importante comprendere tutto nei dettagli perché tutto il lavoro può essere svolto dal decoratore di classe `@Qtrac.has_methods` fornito con gli esempi del libro (e trattato più avanti in questo capitolo).

Il metodo speciale `__subclasshook__()` inizia controllando se l’istanza di classe su cui è richiamato (`Class`) è `Renderer`; se non lo è, restituisce `NotImplemented`. Questo significa che il comportamento `__subclasshook__` non è ereditato dalle sottoclassi. Abbiamo scelto questo approccio perché assumiamo che una sottoclasse stia aggiungendo nuovi criteri alla classe base astratta, anziché aggiungere un comportamento. Naturalmente possiamo comunque ereditare il comportamento, se vogliamo, semplicemente richiamando esplicitamente `Renderer.__subclasshook__()` nella nostra reimplementazione di `__subclasshook__()`.

Se è stato restituito `True` o `False`, il meccanismo della classe base astratta viene arrestato e viene restituito il valore `bool`. Tuttavia, restituendo `NotImplemented` consentiamo l’attuazione della normale funzionalità di ereditarietà (sottoclassi, sottoclassi di classi registrate esplicitamente, sottoclassi di sottoclassi).

Se la condizione dell'istruzione `if` è soddisfatta, iteriamo su ogni classe ereditata dalla `Subclass` (inclusa se stessa), restituita dal metodo speciale `__mro__()`, e accediamo al suo dizionario privato (`__dict__`). Questo fornisce una tupla di `__dict__` che suddividiamo immediatamente usando lo spacchettamento di sequenza (`*`), in modo che tutti i dizionari siano passati alla funzione `collections.ChainMap()`. Tale funzione accetta come argomenti un numero qualsiasi di mappe (come `dict`) e restituisce una vista a mappa singola, come se tutte fossero nella stessa mappa. Ora creiamo una tupla dei metodi che vogliamo controllare. Infine, iteriamo su tutti i metodi e controlliamo che ciascuno sia nella mappa `attributes` le cui chiavi sono i nomi di tutti i metodi e le proprietà della `Subclass` e di tutte le sue `Superclass`, e restituiamo `True` se tutti i metodi sono presenti.

Notate che controlliamo solo che la sottoclasse (o una qualsiasi delle sue classi base) abbia attributi i cui nomi corrispondano ai metodi richiesti, perciò anche una proprietà andrà bene. Se volessimo essere certi di trovare corrispondenza solo per i metodi, potremmo aggiungere al test `method in attributes` una clausola aggiuntiva `and callable(method)`; tuttavia, nella pratica raramente questo costituisce un problema, perciò non vale la pena di farlo.

Creare una classe con un metodo `__subclasshook__()` per fornire il controllo di interfaccia è molto utile, ma scrivere dieci righe di codice complesso per ognuna di tali classi, quando si differiscono soltanto per la classe base e i metodi supportati, è proprio un esempio di duplicazione di codice che è meglio evitare. Nel paragrafo seguente creeremo un decoratore di classe che ci consentirà di creare classi di controllo di interfaccia con soltanto un paio di righe di codice univoche (tra gli esempi è incluso anche il programma `render2.py` che fa uso di tale decoratore).

```
class TextRenderer:

    def __init__(self, width=80, file=sys.stdout):
        self.width = width
        self.file = file
        self.previous = False

    def header(self, title):
        self.file.write("{0:^{2}}\n{1:^{2}}\n".format(title,
                "=" * len(title), self.width))
```

Questo è l'inizio di una semplice classe che supporta l'interfaccia per il rendering della pagina. Il metodo `header()` scrive il titolo dato centrato nella larghezza data, e sulla riga seguente scrive un carattere `=` sotto ciascun carattere del titolo in questione.

```
    def paragraph(self, text):
        if self.previous:
            self.file.write("\n")
        self.file.write(textwrap.fill(text, self.width))
```

```
        self.file.write("\n")
        self.previous = True

    def footer(self):
        pass
```

Il metodo `paragraph()` utilizza il modulo `textwrap` della libreria standard di Python per scrivere il paragrafo dato, con la larghezza data. Usiamo il booleano `self.previous` per assicurarci che ogni paragrafo sia separato dal precedente con una riga vuota. Il metodo `footer()` in questo caso non fa nulla, ma deve essere presente perché fa parte dell'interfaccia di rendering della pagina.

```
class HtmlWriter:

    def __init__(self, file=sys.stdout):
        self.file = file

    def header(self):
        self.file.write("<!doctype html>\n<html>\n")

    def title(self, title):
        self.file.write("<head><title>{}</title></head>\n".format(
                escape(title)))

    def start_body(self):
        self.file.write("<body>\n")

    def body(self, text):
        self.file.write("<p>{}</p>\n".format(escape(text)))

    def end_body(self):
        self.file.write("</body>\n")

    def footer(self):
        self.file.write("</html>\n")
```

La classe `HtmlWriter` può essere usata per scrivere una semplice pagina HTML e si occupa di eseguire l'escape con la funzione `html.escape()` (oppure con la funzione `xml.sax. saxutil.escape()` in Python 3.2 o versioni precedenti).

Benché questa classe disponga di metodi `header()` e `footer()`, essi hanno comportamenti diversi da quelli promessi dall'interfaccia del renderer di pagina. Perciò, a differenza di quanto abbiamo fatto con `TextRenderer`, non possiamo passare un `HtmlWriter` come renderer di pagina a un'istanza di `Page`.

Una soluzione sarebbe quella di creare una sottoclasse di `HtmlWriter` e fornirle i metodi dell'interfaccia per il renderer di pagina. Sfortunatamente questo approccio è fragile, perché la classe risultante conterrà un misto di metodi di `HtmlWriter` e di metodi di interfaccia del renderer di pagina. Una soluzione molto migliore è quella di creare un *adattatore*, ovvero una classe che aggreghi la classe che ci serve, che fornisca l'interfaccia richiesta e che gestisca tutte le opportune intermediazioni. La Figura 2.1 mostra come tale classe adattatore rientri nel contesto del progetto.

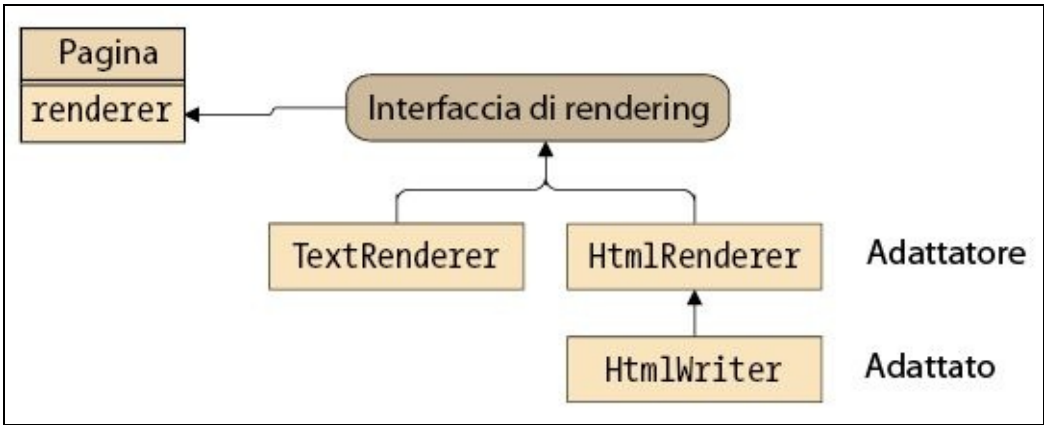


Figura 2.1 Una classe adattatore per il rendering di pagina inquadrata nel contesto del progetto.

```
class HtmlRenderer:

    def __init__(self, htmlWriter):
        self.htmlWriter = htmlWriter

    def header(self, title):
        self.htmlWriter.header()
        self.htmlWriter.title(title)
        self.htmlWriter.start_body()

    def paragraph(self, text):
        self.htmlWriter.body(text)

    def footer(self):
        self.htmlWriter.end_body()
        self.htmlWriter.footer()
```

Questa è la nostra classe adattatore. Riceve un `htmlWriter` di tipo `HtmlWriter` al momento della costruzione e fornisce i metodi dell'interfaccia per il rendering di pagina. Tutto il lavoro effettivo è delegato a `HtmlWriter`, perciò la classe `HtmlRenderer` è semplicemente un wrapper che fornisce una nuova interfaccia per la classe `HtmlWriter` esistente.

```
textPage = Page(title, TextRenderer(22))
textPage.add_paragraph(paragraph1)
textPage.add_paragraph(paragraph2)
textPage.render()

htmlPage = Page(title, HtmlRenderer(HtmlWriter(file)))
htmlPage.add_paragraph(paragraph1)
htmlPage.add_paragraph(paragraph2)
htmlPage.render()
```

Qui vediamo due esempi che mostrano la creazione di istanze della classe `Page` con i loro renderer personalizzati. In questo caso abbiamo specificato per `TextRenderer` una larghezza di default di 22 caratteri e abbiamo fornito alla classe `HtmlWriter` usata dall'adattatore `HtmlRenderer` un file aperto su cui scrivere (la creazione di questo file non è mostrata qui) che ridefinisce l'impostazione di default `sys.stdout`.

Il pattern Bridge

Questo pattern è usato nei casi in cui vogliamo separare un'astrazione (per esempio un'interfaccia o un algoritmo) dal modo in cui è implementata.

L'approccio convenzionale, senza ricorrere al pattern Bridge, prevedrebbe di creare una o più classi base astratte e poi di fornire due o più implementazioni concrete per ognuna di esse. Con il pattern Bridge, invece, si creano due gerarchie di classi indipendenti: quella "astratta" che definisce le operazioni (per esempio l'interfaccia e gli algoritmi di alto livello) e quella concreta che fornisce le implementazioni che verranno richiamate dalle operazioni astratte. La classe "astratta" aggrega un'istanza di una delle classi di implementazione concrete, e tale istanza fa da ponte tra l'interfaccia astratta e le operazioni concrete.

Potremmo dire che nel paragrafo precedente dedicato al pattern Adapter la classe `HtmlRenderer` utilizzava il pattern Bridge, poiché aggregava un `HtmlWriter` per provvedere al rendering.

Per l'esempio di questo paragrafo, supponiamo di voler creare una classe per disegnare grafici a barre usando un particolare algoritmo, affidando però ad altre classi il rendering effettivo dei grafici. Esamineremo un programma che fornisce questa funzionalità e che utilizza il pattern Bridge: `barchart1.py`.

```
class BarCharter:

    def __init__(self, renderer):
        if not isinstance(renderer, BarRenderer):
            raise TypeError("Expected object of type BarRenderer, got {}".
                    format(type(renderer).__name__))
        self.__renderer = renderer

    def render(self, caption, pairs):
        maximum = max(value for _, value in pairs)
        self.__renderer.initialize(len(pairs), maximum)
        self.__renderer.draw_caption(caption)
        for name, value in pairs:
            self.__renderer.draw_bar(name, value)
        self.__renderer.finalize()
```

La classe `BarCharter` implementa nel suo metodo `render()` un algoritmo per il disegno di grafici a barre che si basa sull'implementazione del renderer e sulla corrispondenza con una particolare interfaccia per i grafici a barre. Tale interfaccia richiede i metodi `initialize(int, int)`, `draw_caption(str)`, `draw_bar(str, int)` e `finalize()`.

Come abbiamo già fatto nel paragrafo precedente, utilizziamo un test `isinstance()` per assicurarci che l'oggetto `renderer` passato supporti l'interfaccia che ci serve, senza richiedere che i renderer delle barre abbiano una particolare classe base. Tuttavia, anziché creare una classe di dieci righe di codice come abbiamo fatto

precedentemente, questa volta creiamo la nostra classe di controllo dell'interfaccia con due sole righe:

```
@Qtrac.has_methods("initialize", "draw_caption", "draw_bar", "finalize")
class BarRenderer(metaclass=abc.ABCMeta): pass
```

Questo codice crea una classe `BarRenderer` con la necessaria metaclasse per lavorare con il modulo `abc`. Tale classe viene poi passata alla funzione `Qtrac.has_methods()`, che restituisce un decoratore di classe. Questo decoratore aggiunge poi alla classe un metodo di classe `__subclasshook__()` personalizzato, che controlla i metodi dati ogni volta che viene passato un `BarRenderer` come tipo per una chiamata di `isinstance()` (per i lettori che non hanno familiarità con i decoratori di classe è utile consultare il paragrafo dedicato a questo argomento in questo capitolo).

```
def has_methods(*methods):
    def decorator(Base):
        def __subclasshook__(Class, Subclass):
            if Class is Base:
                attributes = collections.ChainMap(*(Superclass.__dict__
                    for Superclass in Subclass.__mro__))
                if all(method in attributes for method in methods):
                    return True
            return NotImplemented
        Base.__subclasshook__= classmethod(__subclasshook__)
        return Base
    return decorator
```

La funzione `has_methods()` del modulo `Qtrac.py` cattura i metodi richiesti e crea una funzione decoratore di classe, che poi restituisce. Il decoratore crea una funzione `__subclasshook__()` e poi la aggiunge alla classe base come metodo di classe, usando la funzione integrata `classmethod()`. Il codice della funzione personalizzata `__subclasshook__()` è in sostanza identico a quello che abbiamo esaminato precedentemente in questo capitolo, ma questa volta, anziché utilizzare una classe base codificata esplicitamente, usiamo la classe decorata (`Base`), e invece di un insieme di nomi di metodi codificati esplicitamente, usiamo quelli passati al decoratore di classe (`methods`).

È anche possibile ottenere la stessa funzionalità di controllo dei metodi mediante l'eredità da una generica classe base astratta. Per esempio:

```
class BarRenderer(Qtrac.Requirer):
    required_methods = {"initialize", "draw_caption", "draw_bar", "finalize"}
```

Questa porzione di codice è tratta da `barchart3.py`. La classe `Qtrac.Requirer` (non mostrata qui, ma inclusa in `Qtrac.py`) è una classe base astratta che esegue gli stessi controlli del decoratore `@has_methods`.

```
def main():
    pairs = (("Mon", 16), ("Tue", 17), ("Wed", 19), ("Thu", 22),
                ("Fri", 24), ("Sat", 21), ("Sun", 19))
```

```
    textBarCharter = BarCharter(TextBarRenderer())
    textBarCharter.render(„Forecast 6/8", pairs)
    imageBarCharter = BarCharter(ImageBarRenderer())
    imageBarCharter.render("Forecast 6/8", pairs)
```

Questa funzione `main()` imposta alcuni dati, crea due grafici a barre, ognuna con una diversa implementazione del renderer ed esegue il rendering dei dati. Gli output sono mostrati nella Figura 2.2, mentre interfaccia e classi sono illustrate nella Figura 2.3.

```
class TextBarRenderer:

    def __init__(self, scaleFactor=40):
        self.scaleFactor = scaleFactor
```

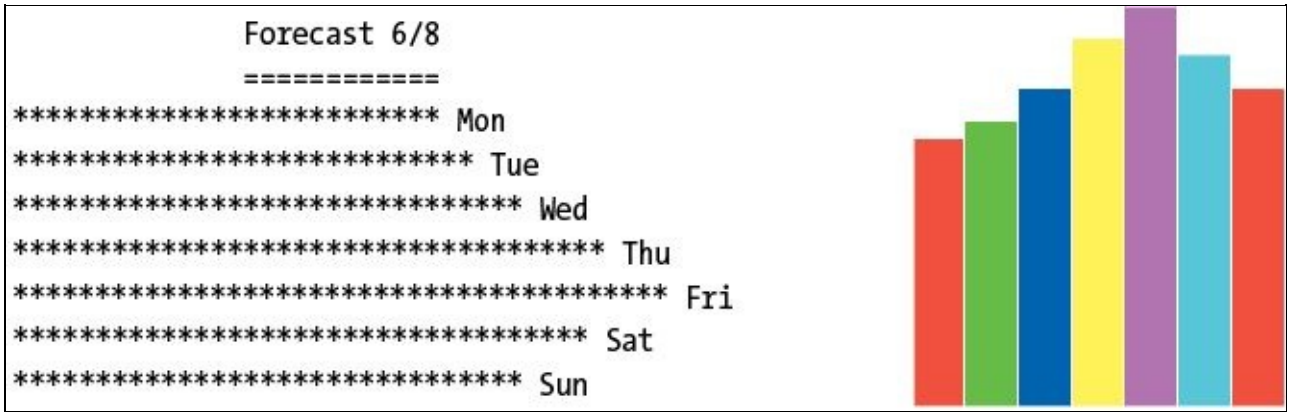


Figura 2.2 Esempi di grafici a barre realizzati con testi e immagini.

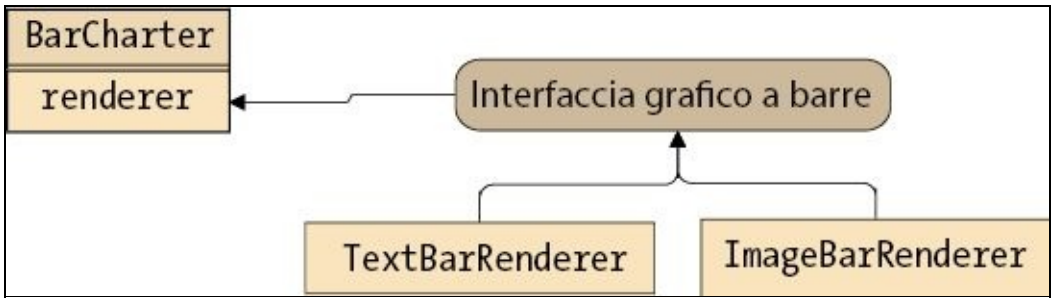


Figura 2.3 Interfaccia e classi per la creazione del grafico a barre.

```
    def initialize(self, bars, maximum):
        assert bars > 0 and maximum > 0
        self.scale = self.scaleFactor / maximum

    def draw_caption(self, caption):
        print("{0:^{2}}\n{1:^{2}}".format(caption, "=" * len(caption),
        self.scaleFactor))

    def draw_bar(self, name, value):
        print("{} {}".format("*" * int(value * self.scale), name))

    def finalize(self):
pass
```

Questa classe implementa l’interfaccia per la creazione del grafico a barre ed esegue il rendering del testo su `sys.stdout`. Naturalmente sarebbe facile fare in modo che il file di output sia definibile dall’utente, e, per i sistemi Unix-like, usare i caratteri Unicode per disegnare riquadri e i colori per ottenere un output più elegante.

Notate che, benché il metodo `finalize()` di `TextBarRenderer` non faccia nulla, deve comunque essere presente per soddisfare l'interfaccia del renderer di grafici a barre.

Anche se la libreria standard di Python è molto ampia, ha un'unica lacuna sorprendente per la sua importanza: non contiene un package per lettura e scrittura di immagini bitmap e vettoriali. Una soluzione è quella di usare una libreria esterna, per esempio una libreria multiformato come Pillow (`http://github.com/python-imaging/Pillow`) oppure una per un formato specifico, o anche un toolkit GUI. Un'altra soluzione è quella di creare una libreria personalizzata per la gestione di immagini, come vedremo più avanti nel Capitolo 3, e precisamente nel paragrafo con il caso di studio finale. Se si è disposti a limitarsi alle immagini GIF (più PNG con Tk/Tcl 8.6), si può usare Tkinter.

NOTA

Notate che la gestione di immagini in Tkinter deve essere svolta nel thread principale (la GUI). Per una gestione di immagini concorrente occorre utilizzare un altro approccio, come vedremo più avanti nel primo paragrafo del Capitolo 4.

In `barchart1.py`, la classe `ImageBarRenderer` utilizza il modulo `cyImage` (o, come alternativa, il modulo `Image`). Quando la differenza non sia rilevante parleremo semplicemente di modulo `Image` per indicare entrambi. Questi moduli sono forniti con il codice degli esempi di questo libro e sono esaminati più avanti (`Image` nel Capitolo 3, nel paragrafo del caso di studio finale, e `cyImage` nel Capitolo 5, e precisamente nel paragrafo dedicato a Cython). Per completezza, gli esempi includono anche `barchart2.py`, una versione di `barchart1.py` che utilizza Tkinter al posto di `cyImage` o `Image`; tuttavia, nel libro non è riportato il codice di tale versione.

Poiché `ImageBarRenderer` è più complesso di `TextBarRenderer`, esamineremo separatamente i suoi dati statici e poi i suoi metodi, uno per uno.

```
class ImageBarRenderer:
    COLORS = [Image.color_for_name(name) for name in ("red", "green",
              "blue", "yellow", "magenta", "cyan")]
```

Il modulo `Image` rappresenta i pixel utilizzando interi senza segno a 32 bit in cui sono codificati quattro componenti di colore: alfa (trasparenza), rosso (*red*), verde (*green*) e blu (*blue*). Il modulo fornisce la funzione `Image.color_for_name()` che accetta un nome di colore - un nome `rgb.txt` X11 (per esempio `"sienna"`) o un nome in stile HTML (per esempio `"#A0522D"`) e restituisce l'intero senza segno corrispondente.

Ora creiamo un elenco di colori da usare per le barre del nostro grafico.

```
    def __init__(self, stepHeight=10, barWidth=30, barGap=2):
        self.stepHeight = stepHeight
```

```
        self.barWidth = barWidth
        self.barGap = barGap
```

Questo metodo consente all'utente di impostare alcune preferenze che influenzano il modo in cui le barre del grafico saranno colorate.

```
    def initialize(self, bars, maximum):
        assert bars > 0 and maximum > 0
        self.index = 0
        color = Image.color_for_name("white")
        self.image = Image.Image(bars * (self.barWidth + self.barGap),
                maximum * self.stepHeight, background=color)
```

Questo metodo (e quelli che seguono) deve essere presente perché fa parte dell'interfaccia per il renderer del grafico a barre. Qui creiamo una nuova immagine la cui dimensione è proporzionale al numero di barre e alle relative larghezze e altezze massime, e che inizialmente è colorata di bianco.

La variabile `self.index` è usata per tenere traccia della barra a cui siamo arrivati (contando a partire da 0).

```
    def draw_caption(self, caption):
        self.filename = os.path.join(tempfile.gettempdir(),
                    re.sub(r"\W+", "_", caption) + ".xpm")
```

Il modulo `Image` non offre supporto per disegnare testi, perciò utilizziamo la caption fornita come base per il nome del file di immagine.

Il modulo `Image` supporta due formati di immagine: XBM (`.xbm`) per immagini monocromatiche e XPM (`.xpm`) per immagini a colori (se è installato il modulo PyPNG, cfr. `http://pypi.python.org/pypi/pypng`, il modulo `Image` supporta anche il formato PNG, `.png`). In questo caso abbiamo scelto il formato di XPM a colori, poiché il nostro grafico a barre è a colori e questo formato è sempre supportato.

```
    def draw_bar(self, name, value):
        color = ImageBarRenderer.COLORS[self.index %
                len(ImageBarRenderer.COLORS)]
        width, height = self.image.size
        x0 = self.index * (self.barWidth + self.barGap)
        x1 = x0 + self.barWidth
        y0 = height - (value * self.stepHeight)
        y1 = height - 1
        self.image.rectangle(x0, y0, x1, y1, fill=color)
        self.index += 1
```

Questo metodo sceglie un colore dalla sequenza `COLORS` (utilizzando a rotazione gli stessi colori se ci sono più barre che colori disponibili), poi calcola le coordinate della barra corrente (`self.index`), ovvero i vertici superiore sinistro e inferiore destro, e indica all'istanza `self.image` (di tipo `Image.Image`) di disegnare un rettangolo utilizzando le coordinate e il colore di riempimento specificati. Poi l'indice viene incrementato per passare alla barra seguente.

```
    def finalize(self):
```

```
        self.image.save(self.filename)
        print("wrote", self.filename)
```

Con questo codice salviamo l’immagine e informiamo di questo fatto l’utente.

`TextBarRenderer` e `ImageBarRenderer` hanno implementazioni radicalmente diverse. Tuttavia, entrambi possono essere usati come ponte per fornire un’implementazione concreta della classe `BarCharter`.

Il pattern Composite

Questo pattern è progettato per supportare la gestione uniforme di oggetti di una gerarchia, indipendentemente dal fatto che contengano altri oggetti (parte della gerarchia) o meno. Tali oggetti sono detti *compositi*. Nell'approccio classico, gli oggetti compositi hanno la stessa classe base per oggetti singoli e per collezioni di oggetti. Sia gli oggetti compositi sia gli altri normalmente hanno gli stessi metodi core, e gli oggetti compositi hanno metodi aggiuntivi per supportare l'aggiunta, la rimozione e l'iterazione sui loro oggetti figli.

Il pattern Composite è spesso usato nei programmi di disegno, come Inkscape, per supportare operazioni di raggruppamento e separazione. In questi casi il pattern è utile perché, quando l'utente seleziona componenti da raggruppare o da separare, alcuni di essi potrebbero essere elementi singoli (per esempio un rettangolo), mentre altri potrebbero essere compositi (per esempio un viso costituito da molte forme diverse).

Per vedere un esempio concreto, esaminiamo una funzione `main()` che crea alcuni elementi singoli e altri compositi, e poi li stampa tutti. Il codice è tratto da `stationery1.py`, ed è seguito dall'output.

```
def main():
    pencil = SimpleItem("Pencil", 0.40)
    ruler = SimpleItem("Ruler", 1.60)
    eraser = SimpleItem("Eraser", 0.20)
    pencilSet = CompositeItem("Pencil Set", pencil, ruler, eraser)
    box = SimpleItem("Box", 1.00)
    boxedPencilSet = CompositeItem("Boxed Pencil Set", box, pencilSet)
    boxedPencilSet.add(pencil)
    for item in (pencil, ruler, eraser, pencilSet, boxedPencilSet):
            item.print()
```

```
$0.40 Pencil
$1.60 Ruler
$0.20 Eraser
$2.20 Pencil Set
        $0.40 Pencil
        $1.60 Ruler
        $0.20 Eraser
$3.60 Boxed Pencil Set
    $1.00 Box
    $2.20 Pencil Set
        $0.40 Pencil
        $1.60 Ruler
        $0.20 Eraser
$0.40 Pencil
```

Ogni `SimpleItem` ha un nome e un prezzo, mentre ogni `CompositeItem` ha un nome e un numero qualsiasi di `SimpleItem`, o `CompositeItem`; gli elementi compositi possono essere annidati a qualsiasi livello. Il prezzo di un elemento composito è la somma dei prezzi degli elementi in esso contenuti.

Nel nostro esempio, un astuccio da scuola è costituito da una matita, un righello e

una gomma. La gerarchia è illustrata nella Figura 2.4.

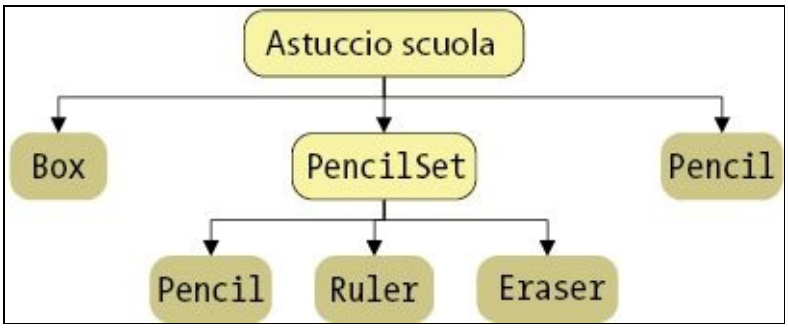


Figura 2.4 Una gerarchia di elementi compositi e non.

Esamineremo due diverse implementazioni del pattern Composite: la prima che usa l'approccio classico e la seconda che usa una singola classe per rappresentare oggetti composti e non.

Una classica gerarchia di elementi compositi e non

L'approccio classico prevede una classe base per tutti i tipi di elementi (compositi e non) e una classe base astratta aggiuntiva per i compositi. La gerarchia di classi è mostrata nella Figura 2.5. Iniziamo esaminando la classe base `AbstractItem`.

```
class AbstractItem(metaclass=abc.ABCMeta):

    @abc.abstractproperty
    def composite(self):
        pass

    def __iter__(self):
        return iter([])
```

Vogliamo che per tutte le sottoclassi si sappia se siano composite o non, e che siano tutte iterabili, con un comportamento di default che prevede la restituzione di un iteratore su una sequenza vuota.

Poiché la classe `AbstractItem` ha almeno un metodo o proprietà astratta, non possiamo creare oggetti `AbstractItem` (tra l'altro, a partire da Python 3.3 è possibile scrivere `@property @abstractmethod def method(...): ...` anziché `@abstractproperty def method(...): ...`).

```
class SimpleItem(AbstractItem):

    def __init__(self, name, price=0.00):
        self.name = name
        self.price = price

    @property
    def composite(self):
        return False
```

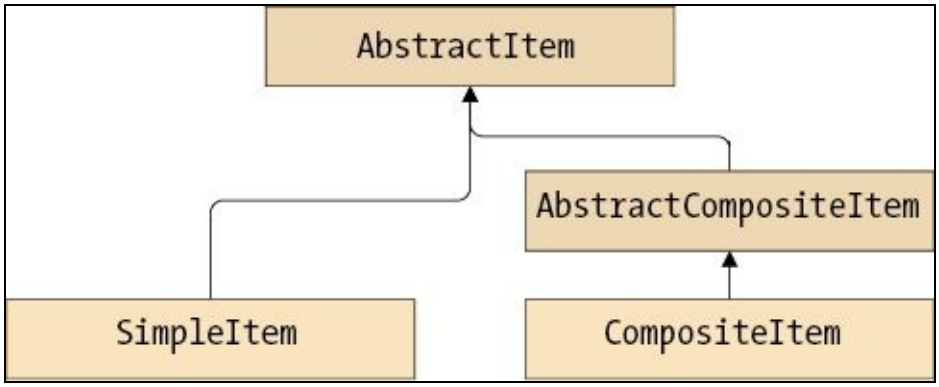


Figura 2.5 Una gerarchia di classi composite e non.

La classe `SimpleItem` è usata per gli elementi non compositi. In questo esempio, i `SimpleItem` hanno proprietà `name` e `price`.

Poiché `SimpleItem` eredita da `AbstractItem`, deve reimplementare tutte le proprietà e i metodi astratti, che in questo caso si riducono alla proprietà `composite`. Poiché il metodo `__iter__()` di `AbstractItem` non è astratto e non lo reimplementiamo, otteniamo la versione della classe base che restituisce un iteratore su una sequenza vuota. Tutto ciò ha senso perché i `SimpleItem` non sono compositi, e così abbiamo comunque la possibilità di gestire `SimpleItem` e `CompositeItem` in modo uniforme (almeno per quanto riguarda l'iterazione); per esempio, passando una combinazione di tali elementi a una funzione come `itertools.chain()`.

```
    def print(self, indent="", file=sys.stdout):
        print("{}${:.2f} {}".format(indent, self.price, self.name),
                    file=file)
```

Abbiamo fornito un metodo `print()` per facilitare la stampa di elementi compositi e non, con gli elementi annidati che utilizzano livelli successivi di rientro.

```
class AbstractCompositeItem(AbstractItem):

    def __init__(self, *items):
        self.children = []
        if items:
            self.add(*items)
```

Questa classe serve da classe base per i `CompositeItem` e fornisce i meccanismi per l'aggiunta, la rimozione e l'iterazione di elementi compositi. Non è possibile istanziare elementi `AbstractCompositeItem`, perché la classe eredita la proprietà astratta `composite` ma non fornisce un'implementazione per essa.

```
    def add(self, first, *items):
        self.children.append(first)
        if items:
            self.children.extend(items)
```

Questo metodo accetta uno o più elementi (sia `SimpleItem` sia `CompositeItem`) e li aggiunge a un elenco di figli. Non avremmo potuto scartare il parametro `first` e utilizzare soltanto `*items`, perché ciò avrebbe consentito di aggiungere zero elementi, cosa che, per quanto in questo caso non comporti problemi, potrebbe probabilmente mascherare un errore logico nel codice utente (per maggiori informazioni sullo spacchettamento, per esempio su `*items`, cfr. il riquadro dedicato a questo tema nel Capitolo 1). Tra l'altro, non sono presenti controlli per impedire i riferimenti circolari, per esempio allo scopo di evitare l'aggiunta di un elemento composito a se stesso.

Più avanti implementeremo questo metodo usando una sola riga di codice.

```
    def remove(self, item):
        self.children.remove(item)
```

Per la rimozione di elementi abbiamo usato un approccio semplice che ci consente di rimuovere un solo elemento alla volta. Naturalmente un elemento rimosso potrebbe essere composito, e in quel caso la sua rimozione comporterebbe la rimozione di tutti i suoi elementi figli, e così via.

```
    def __iter__(self):
        return iter(self.children)
```

Implementando il metodo speciale `__iter__()` consentiamo l'iterazione sugli elementi figli di un elemento composto in cicli `for`, comprehension e generatori. In molti casi scriveremmo il corpo del metodo come `for item in self.children: yield item`, ma poiché `self.children` è una sequenza (una lista), possiamo usare allo scopo la funzione integrata `iter()`.

```
class CompositeItem(AbstractCompositeItem):

    def __init__(self, name, *items):
        super().__init__(*items) self.name = name

    @property
    def composite(self):
        return True
```

Questa classe è usata per elementi compositi concreti. Ha la proprietà `name`, ma affida tutto il lavoro di gestione dei compositi (aggiunta, rimozione e iterazione di elementi figli) alla classe base. Si possono creare istanze di `CompositeItem` perché la classe fornisce un'implementazione della proprietà astratta `composite`, e non vi sono altre proprietà o metodi astratti.

```
    @property
    def price(self):
        return sum(item.price for item in self)
```

Questa proprietà di sola lettura è sottile: calcola il prezzo di un elemento composito accumulando la somma dei prezzi dei suoi elementi figli (e dei figli di questi, nel caso di elementi figli compositi) in modo ricorsivo, usando un generatore di espressioni come argomento per la funzione integrata `sum()`.

L'espressione `for item in self` comporta che Python richiami effettivamente `iter(self)` per ottenere un iteratore per `self`. Questo dà luogo a una chiamata del metodo speciale `__iter__()`, e tale metodo restituisce un iteratore su `self.children`.

```
    def print(self, indent="", file=sys.stdout):
        print("{}${:.2f} {}".format(indent, self.price, self.name), file=file)
        for child in self:
            child.print(indent + "    ")
```

Anche qui abbiamo fornito un comodo metodo `print()`, anche se, sfortunatamente, la prima istruzione è semplicemente una copia del corpo del metodo `SimpleItem.print()`.

In questo esempio, le classi `SimpleItem` e `CompositeItem` sono progettate per soddisfare la maggior parte dei casi d'uso, tuttavia è possibile creare delle sottoclassi (o sottoclassi delle loro classi base astratte) ove si preferisca una gerarchia a grana più fine.

Le classi `AbstractItem`, `SimpleItem`, `AbstractCompositeItem` e `CompositeItem` mostrate qui funzionano tutte perfettamente, tuttavia il codice sembra più lungo del necessario e non ha un'interfaccia uniforme, poiché i compositi hanno metodi (`add()` e `remove()`) che gli altri non hanno. Affronteremo questo problema nel seguito.

Una singola classe per elementi (non) compositi

Le quattro classi descritte nel paragrafo precedente (due astratte e due concrete) non forniscono un'interfaccia completamente uniforme perché soltanto i compositi supportano i metodi `add()` e `remove()`. Se siamo disponibili ad accettare un piccolo carico aggiuntivo di lavoro, un attributo di lista vuota per elementi non compositi e un `float` per elementi compositi, possiamo utilizzare una singola classe per rappresentare sia elementi compositi, sia elementi non compositi. Così inoltre abbiamo il vantaggio di un'interfaccia del tutto uniforme, poiché ora possiamo richiamare `add()` e `remove()` su qualsiasi elemento, non solo sui compositi, e ottenere un comportamento adeguato.

Nel seguito creiamo una nuova classe `Item` che può essere composita o meno, senza necessità di altre classi. Il codice riportato in questo paragrafo è tratto da `stationery2.py`.

```
class Item:

    def __init__(self, name, *items, price=0.00):
        self.name = name
        self.price = price
        self.children = [] if items:
        self.add(*items)
```

Gli argomenti di `__init__()` non sono elegantissimi, ma in questo caso non ci sono problemi perché, come vedremo tra breve, non ci aspettiamo che `Item()` sia richiamato per creare elementi.

Ogni elemento deve avere un nome e anche un prezzo, per cui forniamo un valore di default. Inoltre, un elemento può avere zero o più elementi figli (`*items`), che sono memorizzati in `self.children`, una lista vuota per elementi non compositi.

```
    @classmethod
    def create(Class, name, price):
        return Item(name, price=price)

    @classmethod
    def compose(Class, name, *items):
        return Item(name, *items)
```

Invece di creare elementi richiamando oggetti di classe, abbiamo fornito due metodi di classe factory che accettano argomenti opportuni e restituiscono un `Item`. Ora, quindi, anziché scrivere `SimpleItem("Ruler", 1.60)` e `CompositeItem("Pencil Set", pencil, ruler, eraser)`, scriviamo `Item.create("Ruler", 1.60)` e `Item.compose("Pencil Set", pencil, ruler, eraser)`. E naturalmente, ora tutti i nostri elementi sono dello stesso tipo: `Item`. Ovviamente gli utenti possono sempre usare `Item()` direttamente, se preferiscono; per esempio con `Item("Ruler", price=1.60)` e `Item("Pencil Set", pencil, ruler, eraser)`.

```
def make_item(name, price):
    return Item(name, price=price)

def make_composite(name, *items):
    return Item(name, *items)
```

Abbiamo anche fornito due funzioni factory che operano esattamente come i metodi di classe. Tali funzioni sono comode quando utilizziamo i moduli. Per esempio, se la nostra classe `Item` fosse nel modulo `Item.py`, potremmo sostituire, per esempio, `Item.Item.create("Ruler", 1.60)` con `Item.make_item("Ruler", 1.60)`.

```
    @property
    def composite(self):
        return bool(self.children)
```

Questa proprietà è diversa dalla versione precedente, perché ogni elemento potrebbe essere composito o meno. Per la classe `Item`, un elemento composito è tale se la sua lista `self.children` non è vuota.

```
    def add(self, first, *items):
        self.children.extend(itertools.chain((first,), items))
```

Il metodo `add()` è leggermente diverso rispetto alla versione precedente, in questo caso infatti abbiamo adottato un approccio che dovrebbe essere più efficiente. La funzione `itertools.chain()` accetta qualsiasi numero di iterabili e restituisce un singolo iterabile che è in effetti la concatenazione di tutti gli iterabili passati al metodo.

Questo metodo può essere richiamato su qualsiasi elemento, che sia composito o meno. Nel caso di elementi non compositi, la chiamata fa diventare l'elemento composito.

Un effetto collaterale causato dalla trasformazione di un elemento non composito in uno composito è che il prezzo di tale elemento viene nascosto, perché diventa la somma dei prezzi dei suoi elementi figli. Sono naturalmente possibili scelte di progettazione diverse, che prevedano per esempio di mantenere il prezzo.

```
    def remove(self, item):
        self.children.remove(item)
```

Se si rimuove l'ultimo figlio di un elemento composito, tale elemento diventa non composito. Tale trasformazione presenta un aspetto particolare: il prezzo dell'elemento diventa il valore del suo attributo privato `self.__price`, anziché la somma dei prezzi dei suoi elementi figli (che non esistono più). Impostiamo un prezzo iniziale per tutti gli elementi nel metodo `__init__()` per assicurarci che tutto questo meccanismo funzioni sempre.

```
    def __iter__(self):
        return iter(self.children)
```

Questo metodo restituisce un iteratore su una lista composita di elementi figli, oppure, nel caso di un elemento non composito, su una sequenza vuota.

```
    @property
    def price(self):
        return (sum(item.price for item in self) if self.children else
                    self.__price)

    @price.setter
    def price(self, price):
        self.__price = price
```

La proprietà `price` deve funzionare sia per elementi compositi (dove è la somma dei prezzi degli elementi figli) sia per i non compositi (dove è il prezzo dell'elemento).

```
    def print(self, indent="", file=sys.stdout):
        print("{}${:.2f} {}".format(indent, self.price, self.name),
                    file=file)
    for child in self:
        child.print(indent + "    ")
```

Anche questo metodo deve funzionare per compositi e non, benché il codice sia identico a quello del metodo `CompositeItem.print()` descritto nel paragrafo precedente. Quando iteriamo su un elemento non composito, il metodo restituisce un iteratore su una sequenza vuota, perciò non vi è rischio di ricorsione infinita quando iteriamo sui figli di un elemento.

La flessibilità di Python consente di creare classi composite e non in modo semplice, come classi separate per minimizzare il carico in termini di spazio di memorizzazione, o come singola classe per fornire un’interfaccia completamente uniforme.

Vedremo un’altra variante del pattern Composite nel Capitolo 3, e precisamente nel paragrafo dedicato al pattern Command.

Il pattern Decorator

In generale, un *decoratore* è una funzione che accetta una funzione come unico argomento e restituisce una nuova funzione con lo stesso nome dell'originale ma con funzionalità potenziate. I decoratori sono spesso usati dai framework (per esempio i framework web) per facilitare l'integrazione di funzioni al loro interno.

Il pattern Decorator è talmente utile che Python offre un supporto integrato per esso. In Python i decoratori possono essere applicati a funzioni e metodi. Inoltre, Python supporta anche i decoratori di classe, funzioni che accettano come unico argomento una classe e restituiscono una nuova classe con lo stesso nome dell'originale ma con funzionalità aggiuntive. I decoratori di classe possono talvolta essere utilizzati come alternativa alla creazione di sottoclassi.

La funzione integrata `property()` di Python può essere usata come decoratore, come abbiamo già visto (per esempio nelle proprietà `composite` e `price` descritte nel paragrafo precedente). La libreria standard di Python include alcuni decoratori integrati. Per esempio, il decoratore di classe `@functools.total_ordering` può essere applicato a una classe che implementa i metodi speciali `__eq__()` e `__lt__()` (che forniscono gli operatori di confronto `==` e `<`); in questo modo la classe viene sostituita da una nuova versione che include tutti gli altri metodi speciali di confronto, cosicché la classe decorata supporta l'intera gamma di operatori di confronto (`<`, `<=`, `==`, `!=`, `=>` e `>`).

Un decoratore può accettare una sola funzione, un solo metodo o una sola classe come suo unico argomento, perciò in teoria non è possibile parametrizzare i decoratori. Nondimeno, nella pratica non c'è alcuna limitazione, poiché, come vedremo, possiamo creare factory decoratori parametrizzati che possono restituire una funzione decoratore, la quale a sua volta può essere usata per decorare una funzione, un metodo o una classe.

I decoratori di funzioni e metodi

Tutti i decoratori di funzioni (e di metodi) hanno la stessa struttura generale: per prima cosa creano una funzione wrapper (che in questo libro chiamiamo sempre `wrapper()`); all'interno del wrapper dovremmo richiamare la funzione originale, ma prima della chiamata siamo liberi di svolgere qualsiasi attività di pre-elaborazione, e di acquisire il risultato, oltre a eseguire qualsiasi attività di pre-elaborazione anche dopo la chiamata. E siamo liberi di restituire ciò che vogliamo: il risultato originale, un risultato modificato o qualsiasi altra cosa desideriamo. Infine, restituiamo la

funzione wrapper come risultato del decoratore, e questa funzione sostituisce quella originale usando lo stesso nome.

Per applicare un decoratore a una funzione, un metodo o una classe si scrive un simbolo `@` (“at”, “presso”) allo stesso livello di rientro dell’istruzione `def` o `class`, e subito dopo il nome del decoratore in questione. È possibile impilare i decoratori, cioè applicare un decoratore a una funzione decorata, e così via, come illustrato nella Figura 2.6; tra breve vedremo un esempio.

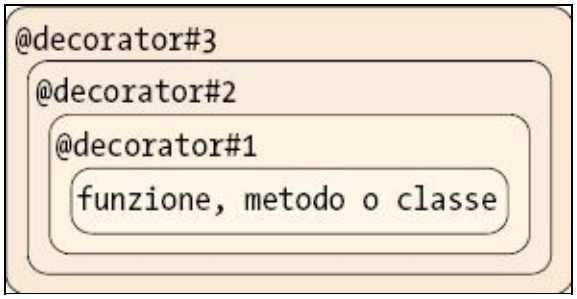


Figura 2.6 Decoratori in pila.

```
@float_args_and_return
def mean(first, second, *rest):
    numbers = (first, second) + rest
    return sum(numbers) / len(numbers)
```

In questo caso abbiamo usato il decoratore `@float_args_and_return` (verrà descritto tra breve) per decorare la funzione `mean()`. La funzione `mean()` non decorata accetta due o più argomenti numerici e restituisce la loro media come `float`, mentre la funzione `mean()` decorata, che chiameremo `mean()` perché va a sostituire l’originale, può accettare due o più argomenti di qualsiasi tipo che saranno convertiti in un `float`. Senza il decoratore, la chiamata `mean(5, "6", "7.5")` avrebbe sollevato un `TypeError`, perché non possiamo sommare valori `int` e `str`, ma con la versione decorata non ci sono problemi, poiché `float("6")` e `float("7.5")` producono numeri validi.

Tra l’altro, la sintassi del decoratore è un caso che rientra nel cosiddetto “zucchero sintattico”, ovvero conta più che altro per una maggiore leggibilità e facilità d’uso. Avremmo potuto scrivere il codice precedente come segue:

```
def mean(first, second, *rest):
    numbers = (first, second) + rest
    return sum(numbers) / len(numbers)
mean = float_args_and_return(mean)
```

In questo caso abbiamo creato la funzione senza un decoratore e poi l’abbiamo sostituita con una versione decorata richiamando noi stessi il decoratore. L’uso dei decoratori è molto comodo, ma talvolta è necessario richiamarli direttamente. Vedremo un esempio verso la fine di questo paragrafo, quando richiameremo il

decoratore integrato `@property` nella funzione `ensure()`. Lo abbiamo fatto anche in precedenza quando abbiamo richiamato il decoratore integrato `@classmethod` nella funzione `has_methods()`.

```
def float_args_and_return(function):
    def wrapper(*args, **kwargs):
        args = [float(arg) for arg in args]
        return float(function(*args, **kwargs))
    return wrapper
```

La funzione `float_args_and_return()` è un decoratore di funzione, perciò accetta come unico argomento una singola funzione. Per convenzione le funzioni wrapper accettano `*args` e `**kwargs`, cioè qualsiasi argomento (cfr. il riquadro del Capitolo 1 dedicato allo spacchettamento di sequenze e mappe). Qualsiasi vincolo sugli argomenti sarà gestito dalla funzione originale, perciò dobbiamo assicurarci che siano passati tutti gli argomenti.

In questo esempio, all'interno della funzione wrapper sostituiamo gli argomenti posizionali passati con un elenco di numero in virgola mobile, poi richiamiamo la funzione originale con gli `*args` eventualmente modificati e ne convertiamo il risultato in un `float`, che poi restituiamo.

Una volta che è stato creato il wrapper, lo restituiamo come risultato del decoratore. Sfortunatamente, nel modo in cui è scritto il codice, l'attributo `__name__` della funzione decorata restituita è impostato a `"wrapper"` anziché al nome della funzione originale, e non c'è docstring, anche se la funzione originale ce l'ha. Perciò la sostituzione non è perfetta. Per superare questa lacuna, la libreria standard di Python include il decoratore `@functools.wraps`, che può essere usato per decorare una funzione wrapper all'interno di un decoratore e garantisce che gli attributi `__name__` e `__doc__` contengano il nome e la docstring della funzione originale.

```
def float_args_and_return(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        args = [float(arg) for arg in args]
        return float(function(*args, **kwargs))
    return wrapper
```

Questa è un'altra versione del decoratore, che utilizza il decoratore `@functools.wraps` per garantire che la funzione `wrapper()` creata all'interno del decoratore abbia il proprio attributo `__name__` correttamente impostato al nome della funzione passata (per esempio `"mean"`) e che abbia la docstring della funzione originale (che in questo esempio è vuota). È meglio usare sempre `@functools.wraps`, perché così si garantisce che nei traceback i nomi delle funzioni decorate siano riportati correttamente (invece di

essere tutti impostati a `"wrapper"`) e che possiamo accedere alle docstring delle funzioni originali.

```
@statically_typed(str, str, return_type=str)
def make_tagged(text, tag):
    return "<{0}>{1}</{0}>".format(tag, escape(text))

@statically_typed(str, int, str) # Accetterà qualsiasi tipo di valore di ritorno
def repeat(what, count, separator):
    return ((what + separator) * count)[:-len(separator)]
```

La funzione `statically_typed()` usata per decorare le funzioni `make_tagged()` e `repeat()` è una factory di decoratori, cioè una funzione di creazione di decoratori. Non è un decoratore di per sé, perché non accetta una funzione, un metodo o una classe come suo unico parametro. Ma in questo caso dobbiamo parametrizzare il decoratore, poiché vogliamo specificare il numero e i tipi di argomenti posizionali che una funzione decorata può accettare (e opzionalmente specificare il tipo del suo valore di ritorno), e questi variano da una funzione all'altra. Abbiamo quindi creato una funzione `statically_typed()` che accetta i parametri che ci servono - un tipo per argomento posizionale e una parola chiave opzionale per specificare il tipo di valore restituito - e restituisce un decoratore.

Perciò, quando Python incontra `@statically_typed(…)` nel codice, richiama la funzione con i dati argomenti e poi utilizza la funzione restituita come decoratore per la funzione che segue (in questo esempio, `make_tagged()` o `repeat()`).

La creazione di factory di decoratori prevede un determinato schema. Per prima cosa creiamo una funzione decoratore, e all'interno di essa creiamo una funzione wrapper; quest'ultima segue lo stesso schema precedente. Come di consueto, al termine del wrapper, viene restituito il risultato della funzione originale (eventualmente modificato o sostituito). E al termine della funzione decoratore viene restituito il wrapper. Alla fine, al termine della factory di decoratori, viene restituito il decoratore.

```
def statically_typed(*types, return_type=None):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            if len(args) > len(types):
                raise ValueError("too many arguments")
            elif len(args) < len(types):
                raise ValueError("too few arguments")
            for i, (arg, type_) in enumerate(zip(args, types)):
                if not isinstance(arg, type_):
                    raise ValueError("argument {} must be of type {}"
                                     .format(i, type_.__name__))
            result = function(*args, **kwargs)
            if (return_type is not None and
                not isinstance(result, return_type)):
                raise ValueError("return value must be of type {}".format(
                             return_type.__name__))
            return result
```

```
        return wrapper
    return decorator
```

In questo caso iniziamo creando una funzione decoratore. L'abbiamo chiamata `decorator()`, ma il nome non conta. All'interno della funzione decoratore creiamo il wrapper, esattamente come abbiamo fatto in precedenza. In questo caso particolare il wrapper è importante, perché verifica il numero e i tipi di tutti gli argomenti posizionali prima di richiamare la funzione originale, e poi controlla il tipo del risultato se è stato specificato un tipo di valore restituito. E al termine, restituisce il risultato.

Una volta creato il wrapper, il decoratore lo restituisce. E poi, proprio alla fine, viene restituito il decoratore stesso. Perciò, quando Python raggiunge per esempio la riga `@statically_typed(str, int, str)` nel codice sorgente, richiama la funzione `statically_typed()`, che restituirà la funzione `decorator()` creata, dopo aver catturato gli argomenti passati alla funzione `statically_typed()`. Ora, tornando a `@`, Python esegue la funzione `decorator()` restituita, passandole la funzione che segue (che può essere una funzione creata con l'istruzione `def` o la funzione restituita da un altro decoratore). In questo caso la funzione è `repeat()`, perciò essa viene passata come unico argomento a `decorator()`. La funzione `decorator()` ora crea una nuova funzione `wrapper()` con parametri che descrivono lo stato catturato (cioè gli argomenti forniti alla funzione `statically_typed()`) e restituisce il wrapper, che Python poi utilizza per sostituire la funzione `repeat()` originale.

Notate che la funzione `wrapper()` creata quando viene richiamata la funzione `decorator()` creata dalla funzione `statically_typed()` ha catturato parte dello stato di configurazione, in particolare la tupla `types` e la parola chiave `return_type`. Quando una funzione o un metodo cattura lo stato in questo modo, si dice che è una chiusura. Il supporto offerto da Python per le chiusure consente di creare funzioni factory, decoratori e factory di decoratori parametrizzate.

L'uso di un decoratore per forzare il controllo del tipo statico degli argomenti, e opzionalmente del valore restituito da una funzione, potrebbe sembrare attraente a chi si avvicina a Python provenendo da un linguaggio a tipizzazione statica (per esempio C, C++ o Java), ma così si introduce una penalizzazione in termini di prestazioni, meno rilevante per i linguaggi compilati. Inoltre, il controllo dei tipi nel caso di un linguaggio a tipizzazione dinamica non è propriamente nello stile di Python, anche se evidenzia la flessibilità del linguaggio (e se vogliamo una tipizzazione statica al momento della compilazione possiamo usare Cython, come vedremo nel Capitolo 5).

È probabilmente più utile la validazione dei parametri, come vedremo nel paragrafo seguente.

I pattern per scrivere decoratori sono abbastanza semplici da usare, anche se potrebbero richiedere un po' di tempo per abituarsi. Nel caso di un decoratore di funzione o metodo senza parametri, basta creare una funzione decoratore che crei e restituisca un wrapper. Questo pattern è mostrato dal decoratore `@float_args_and_return` che abbiamo visto in precedenza e dal decoratore `@Web.ensure_logged_in` che vedremo tra breve. Nel caso di un decoratore parametrizzato, create una factory di decoratori che crei un decoratore (che a sua volta crei un wrapper), seguendo il pattern usato per la funzione `statically_typed()`.

```
@application.post("/mailinglists/add")
@Web.ensure_logged_in
def person_add_submit(username):
    name = bottle.request.forms.get("name")
    try:
        id = Data.MailingLists.add(name)
        bottle.redirect("/mailinglists/view")
    except Data.Sql.Error as err:
        return bottle.mako_template("error", url="/mailinglists/add",
                    text="Add Mailinglist", message=str(err))
```

Questa porzione di codice è tratta da un'applicazione web per la gestione di mailing list che utilizza il framework web `bottle` (`bottlepy.org`). Il decoratore `@application.post` è fornito dal framework ed è usato per associare una funzione con un URL.

Per questo particolare esempio vogliamo soltanto che gli utenti possano accedere alla pagina `mailinglists/add` se hanno effettuato il login e siano reindirizzati alla pagina `login` se non lo hanno fatto. Anziché inserire in ogni funzione che produca una pagina web lo stesso codice per verificare se l'utente ha effettuato il login, abbiamo creato il decoratore `@Web.ensure_logged_in`, che gestisce questa attività ed evita di inserire codice per il login in altre funzioni dove non serve direttamente.

```
def ensure_logged_in(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        username = bottle.request.get_cookie(COOKIE,
                secret=secret(bottle.request))
        if username is not None:
            kwargs["username"] = username
            return function(*args, **kwargs)
        bottle.redirect("/login")
    return wrapper
```

Quando l'utente effettua il login al sito, il codice della pagina `login` verifica nome utente e password, e se sono validi, imposta un cookie nel browser dell'utente che resta in vita per una singola sessione.

Quando l'utente richiede una pagina la cui funzione associata è protetta dal decorate `@ensure_logged_in`, come la funzione `person_add_submit()` della pagina `mailinglists/add`, viene richiamata la funzione `wrapper()` definita qui. Il wrapper inizia tentando di recuperare il nome utente dal cookie; se fallisce, l'utente non ha effettuato il login, perciò lo reindirizziamo alla pagina `login` dell'applicazione web. Ma se l'utente ha effettuato il login, aggiungiamo il nome utente agli argomenti parole chiave, e restituiamo il risultato della chiamata della funzione originale. Ciò significa che, quando viene richiamata la funzione originale, si può assumere che l'utente abbia effettuato il login con successo ed è possibile accedere al nome utente.

I decoratori di classi

Capita abbastanza spesso di creare classi con molte proprietà di lettura-scrittura. Tali classi spesso hanno molto codice duplicato (o quasi) per le routine set-get. Per esempio, supponiamo di avere una classe `Book` che contenga il titolo di un libro, il codice ISBN, il prezzo e la quantità. Ci servirebbero quattro decoratori `@property`, tutti con praticamente lo stesso codice (per esempio `@property def title(self): return title`), e anche quattro metodi di impostazione, ognuno con il proprio codice di validazione, anche se il codice per validare le proprietà di prezzo e quantità sarebbe identico, a parte i valori minimo e massimo accettati. Se le classi come queste sono molte, la quantità di codice duplicato o quasi può diventare enorme.

Fortunatamente in Python grazie ai decoratori è possibile eliminare tutta questa duplicazione di codice. Per esempio, precedentemente in questo capitolo abbiamo usato un decoratore di classi per creare classi personalizzate di controllo dell'interfaccia, senza dover duplicare dieci righe di codice ogni volta (cfr. il paragrafo dedicato al pattern Bridge). E ora presentiamo un altro esempio, con l'implementazione di una classe `Book` che include quattro proprietà completamente validate (più una proprietà calcolata di sola lettura):

```
@ensure("title", is_non_empty_str)
@ensure("isbn", is_valid_isbn)
@ensure("price", is_in_range(1, 10000))
@ensure("quantity", is_in_range(0, 1000000))
class Book:

    def __init__(self, title, isbn, price, quantity):
        self.title = title
        self.isbn = isbn
        self.price = price
        self.quantity = quantity

    @property
    def value(self):
        return self.price * self.quantity
```

`self.title`, `self.isbn` e così via sono tutte proprietà, perciò gli assegnamenti che hanno luogo nel metodo `__init__()` sono tutte validate dal corrispondente metodo di impostazione. Ma anziché dover scrivere manualmente il codice per creare queste proprietà con i loro metodi get-set, abbiamo usato quattro volte un decoratore di classe che fornisce tutta la funzionalità che ci serve.

La funzione `ensure()` accetta due parametri, un nome di proprietà e una funzione di validazione, e restituisce un decoratore di classe, che viene poi applicato alla classe che segue.

In questo caso viene creata la classe `Book`, poi viene effettuata la prima chiamata di `ensure()` con `quantity`, poi viene applicato il decoratore di classe restituito. Così la classe `Book` ora dispone di una proprietà aggiuntiva `quantity`. Poi si effettua la chiamata di `ensure()` con `price`, e dopo l'applicazione del decoratore di classe restituito, la classe `Book` dispone di proprietà `quantity` e `price`. Questo processo viene ripetuto altre due volte, finché otteniamo una versione finale della classe `Book` con tutte e quattro le proprietà.

Sembra di procedere all'indietro, ma in realtà avviene quanto segue:

```
ensure("title", is_non_empty_str)( # Pseudo-code
    ensure("isbn", is_valid_isbn)(
        ensure("price", is_in_range(1, 10000))(
            ensure("quantity", is_in_range(0, 1000000))(class Book: ...))))
```

L'istruzione `class Book` deve essere eseguita per prima, perché l'oggetto di classe risultante è necessario come parametro per la chiamata di `ensure()` con `quantity`, e l'oggetto di classe restituito da questa è necessario per la precedente, e così via.

Notate che prezzo e quantità utilizzano entrambi la stessa funzione di validazione, ma con parametri diversi. In effetti `is_in_range()` è una funzione factory che crea e restituisce una nuova funzione `is_in_range()` con i valori minimo e massimo codificati esplicitamente.

Come vedremo tra breve, il decoratore di classe restituito dalla funzione `ensure()` aggiunge una proprietà alla classe. Questa routine di impostazione della proprietà richiama la funzione di validazione per la proprietà data e le passa due argomenti: il nome della proprietà e il nuovo valore da assegnarle. Il validatore non deve fare nulla se il valore è valido, mentre altrimenti deve sollevare un'eccezione (per esempio `ValueError`). Prima di esaminare l'implementazione di `ensure()`, vediamo un paio di validatori.

```
def is_non_empty_str(name, value):
    if not isinstance(value, str):
```

```
            raise ValueError("{} must be of type str".format(name))
        if not bool(value):
            raise ValueError("{} may not be empty".format(name))
```

Questo validatore è usato per la proprietà `title` della classe `Book` al fine di assicurarsi che il titolo sia una stringa non vuota. Come mostrano gli errori `ValueError`, il nome della proprietà è utile per i messaggi di errore.

```
def is_in_range(minimum=None, maximum=None):
    assert minimum is not None or maximum is not None def is_in_range(name, value):
    if not isinstance(value, numbers.Number):
        raise ValueError("{} must be a number".format(name))
            if minimum is not None and value < minimum:
        raise ValueError("{} {} is too small".format(name, value))
    if maximum is not None and value > maximum:
        raise ValueError("{} {} is too big".format(name, value))
    return is_in_range
```

Questa è una funzione factory che crea una nuova funzione di validazione, la quale controlla che il valore fornito sia un numero (usando la classe astratta `numbers.Number`) e che tale numero rientri nell'intervallo previsto. Una volta creato, il validatore viene restituito.

```
def ensure(name, validate, doc=None):
    def decorator(Class):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            validate(name, value)
            setattr(self, privateName, value)
        setattr(Class, name, property(getter, setter, doc=doc))
        return Class
    return decorator
```

La funzione `ensure()` crea un decoratore di classe parametrizzato con un nome di proprietà, una funzione di validazione e una docstring opzionale. Perciò, ogni volta che un decoratore di classe restituito da `ensure()` è usato per una particolare classe, questa viene arricchita con l'aggiunta di una nuova proprietà.

La funzione `decorator()` riceve una classe come unico argomento. Tale funzione inizia creando un nome privato; il valore della proprietà viene registrato in un attributo con tale nome (quindi, nell'esempio di `Book` il valore della proprietà `self.title` verrà registrato nell'attributo privato `self.__title`). Poi, la funzione crea una funzione di lettura che restituirà il valore registrato nell'attributo con il nome privato. La funzione integrata `getattr()` accetta un oggetto e un nome di attributo e restituisce il valore dell'attributo, o solleva un `AttributeError`. La funzione poi crea una funzione di impostazione che richiama la funzione `validate()` e poi (se `validate()` non ha sollevato eccezioni) imposta il valore registrato nell'attributo con il nome privato al nuovo valore. La funzione integrata `setattr()` accetta un oggetto, un nome di attributo e un

valore e imposta il valore dell'attributo al valore fornito, creando un nuovo attributo se necessario.

Una volta create le funzioni di lettura e impostazione, queste sono utilizzate per creare una nuova proprietà che è aggiunta come attributo alla classe passata, con il nome di proprietà (pubblico) fornito, utilizzando la funzione integrata `setattr()`. La funzione integrata `property()` accetta una funzione di lettura, e opzionalmente funzioni di impostazione, eliminazione e una docstring, e restituisce una proprietà. Può anche essere usata come decoratore di metodo, come abbiamo visto. La classe modificata viene poi restituita dalla funzione `decorator()`, e la stessa funzione `decorator()` viene restituita dalla funzione `ensure()`.

Uso di un decoratore di classe per aggiungere proprietà

Nel precedente esempio dovevamo utilizzare il decoratore di classe `@ensure` per ogni attributo che volessimo validare. Alcuni programmatori in Python non amano "impilare" molti decoratori di classe in questo modo, ma preferiscono combinare un singolo decoratore di classe con attributi nel corpo di una classe, in modo da ottenere un codice più leggibile.

```
@do_ensure
class Book:

    title = Ensure(is_non_empty_str)
    isbn = Ensure(is_valid_isbn)
    price = Ensure(is_in_range(1, 10000))
    quantity = Ensure(is_in_range(0, 1000000))

    def __init__(self, title, isbn, price, quantity):
        self.title = title
        self.isbn = isbn
        self.price = price
        self.quantity = quantity

    @property
    def value(self):
        return self.price * self.quantity
```

Questa è una nuova versione della classe `Book` che utilizza un decoratore di classe `@do_ensure` in combinazione con istanze di `Ensure`. Ogni `Ensure` accetta una funzione di validazione e il decoratore di classe `@do_ensure` sostituisce ogni istanza di `Ensure` con una proprietà validata che ha lo stesso nome. Tra l'altro, le funzioni di validazione (`is_non_empty_str()` e così via) sono le stesse mostrate in precedenza.

```
class Ensure:

    def __init__(self, validate, doc=None):
        self.validate = validate
        self.doc = doc
```

Questa piccola classe è usata per memorizzare la funzione di validazione che verrà usata per l'impostazione della proprietà e, opzionalmente, nella docstring. Per esempio, l'attributo `title` della classe `Book` inizialmente è un'istanza di `Ensure`, ma una volta creata la classe `Book`, il decoratore `@do_ensure` sostituisce ogni `Ensure` con una proprietà. Perciò, l'attributo `title` alla fine diventa una proprietà `title` (la cui impostazione utilizza la funzione di validazione dell'istanza di `Ensure` originale).

```
def do_ensure(Class):
    def make_property(name, attribute):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            attribute.validate(name, value)
            setattr(self, privateName, value)
        return property(getter, setter, doc=attribute.doc)
    for name, attribute in Class.__dict__.items():
        if isinstance(attribute, Ensure):
            setattr(Class, name, make_property(name, attribute))
    return Class
```

Questo decoratore di classe si compone di tre parti. Nella prima parte definiamo una funzione annidata (`make_property()`). La funzione accetta un nome (per esempio `"title"`) e un attributo di tipo `Ensure`, e crea e restituisce una proprietà che registra il suo valore in un attributo privato (per esempio `"__title"`). Inoltre, quando si accede alla funzione di impostazione della proprietà, questa richiama la funzione di validazione. Nella seconda parte iteriamo su tutti gli attributi della classe e sostituiamo ogni `Ensure` con una nuova proprietà. Nella terza parte restituiamo la classe modificata.

Una volta che il decoratore ha terminato, nella classe decorata ognuno degli attributi `Ensure` è stato sostituito da una proprietà validata con lo stesso nome.

In teoria avremmo potuto evitare la funzione annidata e inserire semplicemente tale codice dopo il test `if isinstance()`. Tuttavia, nella pratica questo approccio non funziona a causa di problemi con il binding posticipato, perciò è essenziale disporre di una funzione separata in questo caso. Questo problema non è raro quando si creano decoratori o factory di decoratori, ma usando una funzione separata, eventualmente annidata, solitamente lo si risolve.

Uso di un decoratore di classe invece di creare sottoclassi

A volte creiamo una classe base con alcuni metodi o dati unicamente per fare in modo di poterne creare sottoclassi più volte. In questo modo si evita di dover duplicare i metodi o i dati e non ci sono problemi qualora si debbano creare altre

sottoclassi. Tuttavia, se i metodi o i dati ereditati non vengono mai modificati nelle sottoclassi, è possibile usare un decoratore di classe per ottenere lo stesso scopo.

Per esempio, più avanti utilizzeremo una classe base `Mediated` che fornisce un attributo dati `self.mediator` e un metodo `on_change()` (cfr. il paragrafo dedicato al pattern Mediator nel Capitolo 3). Questa classe è ereditata da due classi, `Button` e `Text`, che utilizzano i dati e il metodo ma senza modificarli.

```
class Mediated:

    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
```

Questa è la classe base tratta da `mediator1.py`. Per l'ereditarietà si usa la sintassi consueta, cioè `class Button(Mediated): ...` e `class Text(Mediated): ...`, ma poiché nessuna sottoclasse avrà mai bisogno di modificare il metodo ereditato `on_change()`, possiamo usare un decoratore di classe al posto di creare sottoclassi.

```
def mediated(Class):
    setattr(Class, "mediator", None)
    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
    setattr(Class, "on_change", on_change)
    return Class
```

Questo codice è tratto da `mediator1d.py`. Il decoratore di classe è applicato come qualunque altro decoratore, cioè con `@mediated class Button: ...` e `@mediated class Text: ...` Le classi decorate hanno esattamente lo stesso comportamento della versione con sottoclassi.

I decoratori di funzioni e di classi sono una caratteristica di Python molto potente e nello stesso tempo molto facile da usare. E come abbiamo visto, i decoratori di classe possono talvolta essere usati come alternativa alla creazione di sottoclassi. La creazione di decoratori è una forma semplice di metaprogrammazione, e i decoratori di classi spesso possono essere usati al posto di forme di metaprogrammazione più complesse, come le metaclassi.

Il pattern Façade

Questo pattern è usato per presentare un'interfaccia uniforme e semplificata a un sottosistema la cui interfaccia è troppo complessa o troppo di basso livello per poter essere usata in modo comodo.

La libreria standard di Python fornisce moduli per gestire file compressi con gzip, tarball e zip, ma tutti hanno interfacce diverse. Supponiamo di voler essere in grado di accedere ai nomi contenuti in un file d'archivio, e di estrarne i file, usando un'interfaccia semplice e uniforme. Una soluzione è quella di utilizzare il pattern Façade per fornire una semplicissima interfaccia di alto livello che demandi la maggior parte del lavoro alla libreria standard.

La Figura 2.7 mostra l'interfaccia che vogliamo fornire agli utenti (una proprietà `filename` e metodi `names()` e `unpack()`) e le interfacce per cui forniamo una "facciata". Un'istanza di `Archive` conterrà un nome di file d'archivio, e soltanto quando vengono richiesti i nomi dell'archivio, o di spacchettare l'archivio, aprirà effettivamente il file d'archivio (il codice riportato in questo paragrafo è tratto da `Unpack.py`).

```
class Archive:

    def __init__(self, filename):
        self._names = None
        self._unpack = None
        self._file = None
        self.filename = filename
```

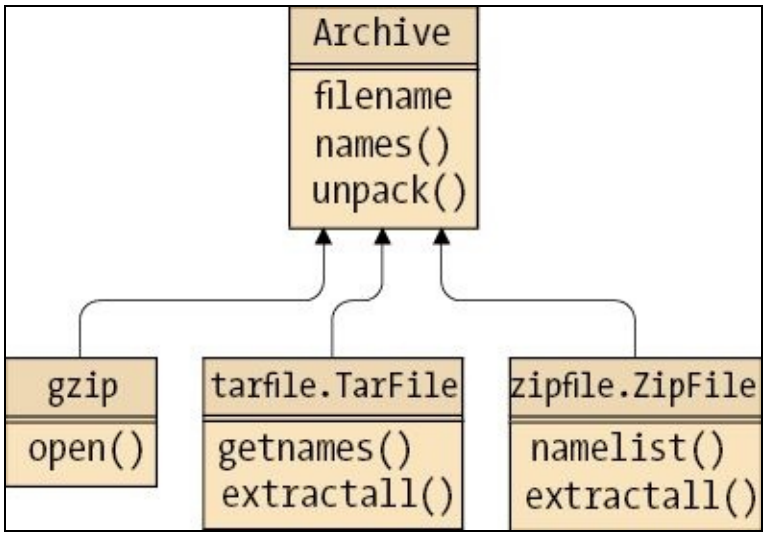


Figura 2.7 La facciata Archive.

La variabile `self._names` dovrebbe contenere un callable (letteralmente "invocabile" o "richiamabile") che restituirà una lista dei nomi dell'archivio. Similmente, la variabile `self._unpack` serve a contenere un callable che estrarrà tutti i file dell'archivio nella directory corrente. `self._file` serve a contenere un oggetto file che sia stato

aperto sull’archivio. E infine, `self.filename` è una proprietà di lettura-scrittura che contiene il nome di file dell’archivio.

```
    @property
    def filename(self):
        return self.__filename

    @filename.setter
    def filename(self, name):
        self.close()
        self.__filename = name
```

Se l’utente modifica il nome di file (per esempio con `archive.filename = newname`), il file di archivio corrente viene chiuso (se è aperto). Tuttavia non apriamo immediatamente il nuovo archivio, perché la classe `Archive` utilizza la lazy evaluation e quindi apre l’archivio soltanto quando necessario.

```
    def close(self):
        if self._file is not None:
            self._file.close()
        self._names = self._unpack = self._file = None
```

In teoria, gli utenti della classe `Archive` dovrebbero richiamare il metodo `close()` quando hanno terminato di lavorare con un’istanza. Il metodo chiude l’oggetto file (se è aperto) e imposta le variabili `self._names`, `self._unpack` e `self._file` a `None` per invalidarle.

Abbiamo impostato la classe `Archive` come context manager (come vedremo tra breve), perciò nella pratica gli utenti non avranno bisogno di richiamare `close()`, purché utilizzino la classe in un’istruzione `with`. Per esempio:

```
with Archive(zipFilename) as archive:
    print(archive.names())
    archive.unpack()
```

Qui creiamo un `Archive` per un file zip, stampiamo i suoi nomi sulla console e poi estraiamo tutti i file nella directory corrente. E poiché l’`archive` è un context manager, `archive.close()` viene richiamato automaticamente quando l’`archive` esce dall’ambito dell’istruzione `with`.

```
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.close()
```

Questi due metodi sono sufficienti per fare di `Archive` un context manager. Il metodo `__enter__()` restituisce `self` (un’istanza di `Archive`), che è assegnato alla variabile dell’istruzione `with … as`. Il metodo `__exit__()` chiude l’oggetto file dell’archivio (se è

aperto) e poiché restituisce (implicitamente) `None`, qualsiasi eccezione che sia sollevata viene propagata normalmente.

```
    def names(self):
        if self._file is None:
            self._prepare()
        return self._names()
```

Questo metodo restituisce un elenco dei nomi di file nell'archivio, aprendo l'archivio in questione e impostando `self._names` e `self._unpack` a callable appropriati (usando `self._prepare()`) se non è già aperto.

```
def unpack(self):
if self._file is None:
self._prepare()
self._unpack()
```

Questo metodo estrae tutti i file dell'archivio, ma come vedremo, solo se tutti i loro nomi sono "sicuri".

```
    def _prepare(self):
        if self.filename.endswith((".tar.gz", ".tar.bz2", ".tar.xz", ".zip")):
            self._prepare_tarball_or_zip()
        elif self.filename.endswith(".gz"):
            self._prepare_gzip()
        else:
            raise ValueError("unreadable: {}".format(self.filename))
```

Questo metodo delega la preparazione a metodi opportuni. Per file tarball e zip il codice necessario è molto simile, perciò la preparazione avviene nello stesso metodo. I file compressi con gzip invece sono gestiti in modo diverso, perciò hanno un metodo separato.

I metodi di preparazione devono assegnare dei callable alle variabili `self._names` e `self._unpack`, in modo che possano essere richiamati nei metodi `names()` e `unpack()` che abbiamo appena visti.

```
    def _prepare_tarball_or_zip(self):
        def safe_extractall():
            unsafe = []
            for name in self.names():
                if not self.is_safe(name):
                            unsafe.append(name)
            if unsafe:
                raise ValueError("unsafe to unpack: {}".format(unsafe))
            self._file.extractall()
        if self.filename.endswith(".zip"):
            self._file = zipfile.ZipFile(self.filename)
            self._names = self._file.namelist
            self._unpack = safe_extractall
        else: # Termina con.tar.gz, .tar.bz2 o .tar.xz
            suffix = os.path.splitext(self.filename)[1]
            self._file = tarfile.open(self.filename, "r:" + suffix[1:])
            self._names = self._file.getnames
            self._unpack = safe_extractall
```

Questo metodo inizia creando una funzione annidata `safe_extractall()` che controlla tutti i nomi dell'archivio e solleva un `ValueError` se vi sono nomi "non sicuri", secondo

quanto definito nel metodo `is_safe()`. Se tutti i nomi sono sicuri, viene richiamato il metodo `tarball.TarFile.extractall()` oppure il metodo `zipfile.ZipFile.extractall()`.

A seconda dell'estensione di file dell'archivio, apriamo un file `tarball.TarFile` o `zipfile.ZipFile` e lo assegniamo a `self._file`. Poi impostiamo `self._names` al metodo bound corrispondente (`namelist()` o `getnames()`) e `self._unpack` alla funzione `safe_extractall()` appena creata. Questa funzione è una chiusura che ha catturato `self` e può quindi accedere a `self._file` e richiamare il metodo `extractall()` appropriato (cfr. il riquadro dedicato ai metodi bound e unbound).

```
def is_safe(self, filename):
    return not (filename.startswith(("/", "\\")) or
        (len(filename) > 1 and filename[1] == ":" and
        filename[0] in string.ascii_letter) or
        re.search(r"[.][.][/\\]", filename))
```

METODI BOUND E UNBOUND

Un *metodo bound* è un metodo già associato a un'istanza della sua classe. Supponiamo di avere una classe `Form` con un metodo `update_ui()`. Ora, se scriviamo `bound = self.update_ui` all'interno di uno dei metodi della classe `Form`, a `bound` viene assegnato un riferimento al metodo `Form.update_ui()` collegato a una particolare istanza del form (`self`). Un metodo bound può essere richiamato direttamente, per esempio con `bound()`.

Un metodo *unbound* è un metodo privo di un'istanza associata. Per esempio, se scriviamo `unbound = Form.update_ui`, a `unbound` è assegnato un riferimento al metodo `Form.update_ui()`, ma senza alcun collegamento a una particolare istanza. Ciò significa che, se vogliamo richiamare il metodo unbound, dobbiamo fornire un'istanza appropriata come primo argomento. Per esempio, `form = Form() ; unbound(form)` (in termini rigorosi, Python 3 è privo di metodi unbound, perciò `unbound` in realtà è l'oggetto funzione sottostante, anche se questo fa differenza soltanto in alcuni casi particolari di metaprogrammazione).

Un file di archivio creato in maniera "maligna" potrebbe, al momento della sua estrazione, andare a sovrascrivere file di sistema importanti, sostituendoli con codice errato o pericoloso. Alla luce di ciò, non si dovrebbero mai aprire archivi contenenti file con percorsi assoluti o componenti di percorsi relativi, e si dovrebbero sempre aprire i file con accesso di utente non privilegiato (quindi mai come root o Administrator).

Questo metodo restituisce `False` se il nome di file fornito inizia con uno slash o un backslash (come avviene in un percorso assoluto), o contiene `../` oppure `..\` (un percorso relativo che potrebbe condurre ovunque), oppure con `D:`, dove `D` è una lettera di unità a disco Windows.

In altre parole, ogni nome di file con percorso assoluto o componenti relativi è considerato non sicuro. Per qualsiasi altro nome di file il metodo restituisce `True`.

```
    def _prepare_gzip(self):
        self._file = gzip.open(self.filename)
        filename = self.filename[:-3]
        self._names = lambda: [filename]
        def extractall():
            with open(filename, "wb") as file:
                        file.write(self._file.read())
        self._unpack = extractall
```

Questo metodo fornisce un oggetto file aperto per `self._file` e assegna opportuni callable a `self._names` e `self._unpack`. Per la funzione `extractall()`, dobbiamo noi stessi leggere e scrivere i dati.

Il pattern Façade può essere molto utile per creare interfacce semplificate e molto comode. Il vantaggio è che in questo modo siamo isolati dai dettagli di basso livello; lo svantaggio è che potremmo dover rinunciare a un controllo più fine. Tuttavia, una “facciata” non nasconde o rimuove le funzionalità sottostanti, perciò possiamo sempre scegliere di utilizzare la facciata per la maggior parte del tempo, e passare alle classi di livello inferiore quando ci serve maggiore controllo.

I pattern Façade e Adapter a prima vista sono simili. La differenza è che una facciata fornisce un’interfaccia semplice sopra una più complessa, mentre un adattatore fornisce un’interfaccia standardizzata sopra un’altra interfaccia (non necessariamente complessa). I due pattern possono essere usati insieme. Per esempio, potremmo definire un’interfaccia per gestire file di archivio (tarball, zip, file `.cab` di Windows e così via), usare un adattatore per ogni formato, e disporre sopra tutto ciò una facciata in modo che gli utenti non debbano preoccuparsi di quale particolare formato di file sia utilizzato.

Il pattern Flyweight

Il pattern Flyweight (letteralmente “peso mosca”, si richiama a una categoria del pugilato per atleti molto leggeri) è progettato per gestire un gran numero di oggetti relativamente piccoli, di cui molti sono duplicati di altri. È implementato rappresentando ciascun oggetto univoco una sola volta, e condividendo tale istanza univoca ogni volta che sia necessario.

Python assume un approccio “leggero” in modo naturale, perché utilizza riferimenti a oggetti. Per esempio, se avessimo un lungo elenco di stringhe con molti duplicati, memorizzando riferimenti a oggetti (cioè variabili) anziché stringhe letterali, potremmo risparmiare molto spazio in memoria.

```
red, green, blue = "red", "green", "blue"
x = (red, green, blue, red, green, blue, red, green)
y = ("red", "green", "blue", "red", "green", "blue", "red", "green")
```

Nel codice precedente, la tupla `x` memorizza 3 stringhe usando 8 riferimenti a oggetti. La tupla `y` memorizza 8 stringhe usando 8 riferimenti a oggetti, poiché in realtà tale codice è una versione con sintassi più leggibile di `_anonymous_item0 = "red", ..., _anonymous_item7 = "green"; y = (_anonymous_item0, ... _anonymous_item7)`.

Probabilmente il modo più semplice per trarre vantaggio dal pattern Flyweight in Python è quello di usare un `dict`, in cui ciascun oggetto univoco è registrato come un valore identificato da una chiave univoca. Per esempio, se volessimo creare molte pagine HTML con font specificati da stili CSS (*Cascading Style Sheets*), anziché creare un nuovo font ogni volta che ne serve uno, potremmo creare in anticipo tutti quelli che ci servono (oppure crearli quando richiesto) e mantenerli in un `dict`. Poi, ogni volta che ci serve un font, potremmo ricavarlo dal `dict`. In questo modo ciascun singolo font sarebbe creato una sola volta, indipendentemente dal numero di volte in cui è usato.

In alcune situazioni potremmo avere un gran numero di oggetti non necessariamente piccoli, di cui la maggior parte sono diversi dagli altri. Un modo facile per ridurre lo spazio occupato in memoria in questi casi è quello di utilizzare `__slots__`.

```
class Point:

    __slots__= ("x", "y", "z", "color")

    def __init__(self, x=0, y=0, z=0, color=None):
        self.x = x
        self.y = y
        self.z = z
        self.color = color
```

Questa è una semplice classe `Point` che contiene una posizione tridimensionale e un colore. Grazie agli `__slots__`, nessun `Point` ha il proprio `dict` privato (`self.__dict__`). Tuttavia, questo significa anche che non è possibile aggiungere alcun attributo arbitrario ai singoli punti (questa classe è tratta da `pointstore1.py`.)

Su una macchina di test sono serviti circa 2 secondi e mezzo per creare una tupla di un milione di questi punti, e il programma (che faceva poco altro) occupava 183 MiB (mebibyte) di memoria RAM. Senza gli slot, questo programma è stato eseguito in una frazione di secondo in meno, ma occupava 312 MiB di memoria RAM.

Per default Python sacrifica sempre la memoria in nome della velocità, ma possiamo invertire questo approccio, se la nostra situazione lo richiede.

```
class Point:

    __slots__= ()
    __dbm = shelve.open(os.path.join(tempfile.gettempdir(), "point.db"))
```

Questo è l'inizio della nostra seconda classe `Point` (tratta da `pointstore2.py`). Utilizza un database DBM (chiave-valore) memorizzato in un file su disco per registrare i dati. Un riferimento a oggetto per il DBM è memorizzato nella variabile statica (a livello di classe) `Point.__dbm`. Tutti i `Point` condividono lo stesso file DBM sottostante. Iniziamo aprendo il file DBM. Il comportamento di default del modulo `shelve` prevede di creare automaticamente tale file, se non esiste già (vedremo più avanti come assicurarci che il file DBM sia chiuso in maniera appropriata).

Il modulo `shelve` esegue il *pickling* (serializza) del valore che registriamo e l'*unpickling* (deserializza) i valori che recuperiamo (il formato di pickling di Python non è sicuro, perché il processo di unpickling esegue codice Python arbitrario. Alla luce di ciò non dovremmo mai usare pickle provenienti da fonti non sicure, o per dati a cui sia possibile accedere da accesso non protetto. In alternativa, se vogliamo usare i pickle in tali circostanze, dovremmo applicare apposite misure di sicurezza, quali checksum e cifratura).

```
    def __init__(self, x=0, y=0, z=0, color=None):
        self.x = x
        self.y = y
        self.z = z
        self.color = color
```

Questo metodo è esattamente identico a quello di `pointstore1.py`, ma dietro le quinte i valori sono assegnati al file DBM sottostante.

```
    def __key(self, name):
        return   "{:X}:{}".format(id(self), name)
```

Questo metodo fornisce la stringa chiave per ognuno degli attributi `x`, `y`, `z` e `color` di `Point`. La chiave è costituita dall'ID dell'istanza (un numero univoco restituito dalla funzione integrata `id()`) in esadecimale e dal nome dell'attributo. Per esempio, se avessimo un `Point` con ID di 3 954 827, il suo attributo `x` sarebbe registrato con la chiave `"3C588B:x"`, il suo attributo `y` con la chiave `"3C588B:y"` e così via.

```
def __getattr__(self, name):
    return Point.__dbm[self.__key(name)]
```

Questo metodo è richiamato ogni volta che si accede a un attributo `Point` (per esempio con `x = point.x`). Chiavi e valori dei database DBM devono essere `bytes`. Fortunatamente i moduli DBM di Python accettano sia chiavi `str` sia chiavi `bytes`, convertendo le prime in `bytes` mediante la codifica di default (UTF-8) dietro le quinte. E se usiamo il modulo `shelve` (come abbiamo fatto qui), possiamo registrare qualsiasi valore serializzabile che vogliamo, affidandoci al modulo `shelve` per la conversione in e da `bytes` come richiesto.

Qui otteniamo dunque la chiave appropriata e recuperiamo il valore corrispondente. E grazie al modulo `shelve`, il valore recuperato è convertito da (pickled) `bytes` al tipo impostato in origine (per esempio in un `int` o in `None` per il colore di un punto).

```
def __setattr__(self, name, value):
    Point.__dbm[self.__key(name)] = value
```

Ogni volta che viene impostato un attributo di `Point` (per esempio con `point.y = y`), viene richiamato questo metodo. Qui otteniamo la chiave appropriata e impostiamo il suo valore, affidandoci al modulo `shelve` per convertire il valore in (pickled) `bytes`.

```
atexit.register(__dbm.close)
```

Al termine della classe `Point` registriamo il metodo `close()` del DBM da richiamare quando il programma termina, usando la funzione `register()` del modulo `atexit`.

Sulla nostra macchina di test è servito circa un minuto per creare un database di un milione di punti, ma il programma occupava soltanto 29 MiB di memoria RAM (più un file su disco di 361 MiB), rispetto ai 183 MiB di memoria RAM della prima versione. Benché il tempo richiesto per riempire il DBM sia considerevole, una volta eseguito questo compito, la velocità di ricerca dovrebbe essere maggiore, poiché la maggior parte dei sistemi operativi memorizzerà nella cache un file utilizzato di frequente.

Il pattern Proxy

Utilizziamo il pattern Proxy quando vogliamo che un oggetto prenda il posto di un altro. Nel libro *Design Patterns* sono presentati quattro casi d'uso. Il primo è un proxy remoto in cui un oggetto locale fa da proxy per un oggetto remoto. La libreria RPyC è un esempio perfetto di questo caso: consente di creare oggetti su un server e proxy per tali oggetti su uno o più client (questa libreria è presentata nel Capitolo 6). Il secondo caso è quello di un proxy virtuale che ci consente di creare oggetti leggeri al posto di oggetti pesanti, che vengono creati soltanto quando sono effettivamente necessari. Vedremo un esempio in questo paragrafo. Il terzo caso è quello di un proxy di protezione che fornisce diversi livelli di accesso a seconda dei diritti di accesso su un client. Il quarto è un riferimento intelligente che "svolge azioni aggiuntive quando si accede a un oggetto". Possiamo utilizzare lo stesso approccio di codifica per tutti i proxy, anche se il comportamento del quarto caso potrebbe essere ottenuto anche mediante un descrittore (per esempio, sostituendo un oggetto con una proprietà usando il decoratore `@property`).

NOTA

I descrittori sono trattati nel volume *Programming in Python 3*, seconda edizione (cfr. la bibliografia selezionata per i dettagli) e nella documentazione online: `http://docs.python.org/3/reference/datamodel.html#descriptors`.

Questo pattern può anche essere usato per il test di unità. Per esempio, se abbiamo la necessità di testare un codice che accede a una risorsa non sempre disponibile, o una classe che è in fase di sviluppo ma ancora incompleta, potremmo creare un proxy per la risorsa o classe che fornisce l'interfaccia completa, ma con degli stub per le funzionalità che mancano. Questo approccio è talmente utile che Python 3.3 include la libreria `unittest.mock` che consente di creare oggetti mock e di aggiungere stub al posto di metodi mancanti (cfr. `http://docs.python.org/py3k/library/unittest.mock.html`).

Per l'esempio di questo paragrafo assumeremo di dover creare più immagini in modo speculativo, dove alla fine ne viene utilizzata soltanto una. Abbiamo un modulo `Image` e un modulo `cyImage` quasi equivalente ma più veloce (è trattato nel Capitolo 3 e nel Capitolo 5), ma questi moduli creano le loro immagini in memoria.

Poiché ci serve soltanto una delle immagini create in modo speculativo, sarebbe meglio creare proxy di immagini leggere e procedere con la creazione di un'immagine reale soltanto quando conosciamo davvero quella che ci serve.

L'interfaccia della classe `Image.Image` è costituita da dieci metodi oltre al costruttore:

`load()`, `save()`, `pixel()`, `set_pixel()`, `line()`, `rectangle()`, `ellipse()`, `size()`, `subsample()` e `scale()`

(non sono inclusi alcuni metodi statici di comodo che sono disponibili anche come funzioni modulo, quali `Image.Image.color_for_name()` e `Image.color_for_name()`).

Per la nostra classe proxy implementeremo soltanto il sottoinsieme di metodi di `Image.Image` che sono sufficienti per i nostri scopi. Iniziamo esaminando come viene utilizzato il proxy. Il codice è tratto da `imageproxy1.py`; l'immagine prodotta è mostrata nella Figura 2.8.

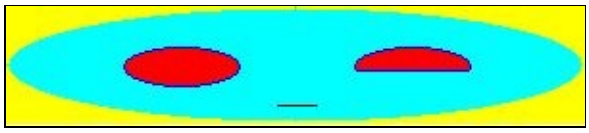


Figura 2.8 Un'immagine disegnata.

```
YELLOW, CYAN, BLUE, RED, BLACK = (Image.color_for_name(color)
    for color in ("yellow", "cyan", "blue", "red", "black"))
```

Per prima cosa creiamo alcune costanti di colore usando la funzione `color_for_name()` del modulo `Image`.

```
image = ImageProxy(Image.Image, 300, 60)
image.rectangle(0, 0, 299, 59, fill=YELLOW)
image.ellipse(0, 0, 299, 59, fill=CYAN)
image.ellipse(60, 20, 120, 40, BLUE, RED)
image.ellipse(180, 20, 240, 40, BLUE, RED)
image.rectangle(180, 32, 240, 41, fill=CYAN)
image.line(181, 32, 239, 32, BLUE)
image.line(140, 50, 160, 50, BLACK)
image.save(filename)
```

Qui creiamo un proxy di immagine, passando la classe di immagine che vogliamo utilizzare. Poi disegniamo su di esso, e al termine salviamo l'immagine risultante. Questo codice funzionerebbe anche se avessimo creato l'immagine usando `Image.Image()` anziché `ImageProxy()`; tuttavia, usando un proxy, l'immagine reale non viene creata finché non viene richiamato il metodo `save()`, perciò il costo di creare l'immagine prima di salvarla è estremamente basso (in termini di memoria e di elaborazione), e se alla fine scarteremo l'immagine senza salvarla, non avremo perso molto. Nel caso in cui si utilizza `Image.Image`, invece, si rischia di pagare un prezzo molto alto in anticipo (creando effettivamente un array di valore di colore con determinate dimensioni) e di svolgere parecchio lavoro di elaborazione al momento del disegno (impostare ciascun pixel in un rettangolo colorato, oltre a calcolare quali impostare), anche se alla fine scarteremo l'immagine in questione.

```
class ImageProxy:
    def __init__(self, ImageClass, width=None, height=None, filename=None):
        assert (width is not None and height is not None) or \
                filename is not None
        self.Image = ImageClass
        self.commands = []
        if filename is not None:
            self.load(filename)
```

```
        else:
            self.commands = [(self.Image, width, height)]
    def load(self, filename):
        self.commands = [(self.Image, None, None, filename)]
```

La classe `ImageProxy` può assumere il posto di una `Image.Image` (o di qualunque altra classe di immagine passata a essa che supporti l'interfaccia `Image`), purché l'interfaccia incompleta fornita dal proxy sia sufficiente. Una `ImageProxy` non memorizza un immagine, ma semplicemente un elenco di tuple di comando, dove la prima voce di ogni tupla è una funzione o un metodo unbound e le rimanenti sono gli argomenti da passare quando la funzione o il metodo viene richiamato.

Quando viene creata una `ImageProxy`, occorre specificare larghezza e altezza (per creare una nuova immagine con le dimensioni specificate) o un nome di file. Se si specifica un nome di file, in esso sono memorizzati gli stessi comandi di una chiamata di `ImageProxy.load()`: il costruttore `Image.Image()` e, come argomenti, `None` e `None` per larghezza e altezza e il nome di file. Notate che, se `ImageProxy.load()` è richiamata in un secondo tempo, tutti i comandi precedenti sono scartati e il comando di caricamento diventa il primo e l'unico comando in `self.commands`. Se si forniscono una larghezza e un'altezza, viene registrato il costruttore `Image.Image()` insieme con tali valori come argomenti.

Se si richiama un qualsiasi metodo non supportato (per esempio `pixel()`), tale metodo non viene trovato e Python fa automaticamente ciò che desideriamo: solleva un `AttributeError`. Un approccio alternativo per la gestione di metodi per cui non è possibile utilizzare proxy consiste nel creare un'immagine reale non appena viene richiamato uno di tali metodi, e da lì in poi usare l'immagine reale (il programma `imageproxy2.py`, incluso tra il codice degli esempi ma non riportato qui, assume questo approccio).

```
    def set_pixel(self, x, y, color):
        self.commands.append((self.Image.set_pixel, x, y, color))

    def line(self, x0, y0, x1, y1, color):
        self.commands.append((self.Image.line, x0, y0, x1, y1, color))

    def rectangle(self, x0, y0, x1, y1, outline=None, fill=None):
        self.commands.append((self.Image.rectangle, x0, y0, x1, y1,
                    outline, fill))

    def ellipse(self, x0, y0, x1, y1, outline=None, fill=None):
        self.commands.append((self.Image.ellipse, x0, y0, x1, y1,
                    outline, fill))
```

L'interfaccia di disegno della classe `Image.Image` è costituita da quattro metodi: `line()`, `rectangle()`, `ellipse()` e `set_pixel()`. La nostra classe `ImageProxy` supporta pienamente

questa interfaccia, ma invece di eseguire questi comandi, si limita ad aggiungerli, insieme con i loro argomenti, all'elenco `self.commands`.

```
    def save(self, filename=None):
        command = self.commands.pop(0)
        function, *args = command
        image = function(*args)
        for command in self.commands:
            function, *args = command
            function(image, *args)
        image.save(filename)
        return image
```

Soltanto se scegliamo di salvare l'immagine, dobbiamo creare effettivamente un'immagine vera e pagare il prezzo in termini di elaborazione e memoria. Il modo in cui è progettata `ImageProxy` comporta che il primo comando è sempre quello che crea una nuova immagine (che sia una di larghezza e altezza date, o una caricata da file). Perciò trattiamo in modo speciale quel primo comando salvandone il valore di ritorno, che sappiamo sarà una `Image.Image` (o una `cyImage.Image`). Poi iteriamo sui comandi rimanenti, richiamandoli uno per uno e passando l'`image` come primo argomento (`self`), poiché sono in realtà chiamate di metodi unbound. Alla fine salviamo l'immagine usando il metodo `Image.Image.save()`.

Il metodo `Image.Image.save()` non ha un valore di ritorno (anche se può sollevare un'eccezione nel caso si verifichi un errore). Tuttavia abbiamo modificato leggermente la sua interfaccia per `ImageProxy` restituendo l'`Image.Image` che è stata creata, nel caso in cui fosse necessaria per un'ulteriore elaborazione. Questa modifica non dovrebbe comportare problemi, poiché se il valore di ritorno è ignorato (come sarebbe se richiamassimo `Image.Image.save()`), sarà scartato. Il programma `imageproxy2.py` non richiede tale modifica, poiché ha una proprietà `image` di tipo `Image.Image` che forza la creazione dell'immagine (se non è già stata creata) quando vi si accede.

Registrando i comandi, come abbiamo fatto in questo esempio, si ottiene la possibilità di un adattamento che supporti operazioni di annullamento; per ulteriori informazioni su questo argomento potete consultare il paragrafo dedicato al pattern Command nel Capitolo 3 e quello dedicato al pattern State sempre nel Capitolo 3.

I design pattern strutturali possono essere tutti implementati in Python. I pattern Adapter e Façade facilitano il riutilizzo di classi in nuovi contesti, e il pattern Bridge consente di incorporare la sofisticata funzionalità di una classe all'interno di un'altra. Il pattern Composite facilita la creazione di gerarchie di oggetti, anche se in Python non si ha spesso tale necessità poiché spesso è sufficiente utilizzare i dizionari `dict`. Il pattern Decorator è talmente utile che il linguaggio Python lo supporta direttamente ed estende il concetto alle classi. Il fatto che Python utilizzi riferimenti a oggetti

comporta che il linguaggio stesso impieghi una variante del pattern Flyweight. Infine, il pattern Proxy è particolarmente facile da implementare in Python. Con i design pattern si va oltre la creazione di oggetti di base e complessi e si entra nel regno dei comportamenti: il modo in cui oggetti singoli o gruppi di oggetti possono svolgere varie attività. Nel prossimo capitolo esamineremo i pattern comportamentali.

I design pattern comportamentali

I pattern comportamentali si occupano di come svolgere i vari compiti, ovvero di algoritmi e di interazioni tra oggetti. Forniscono strumenti potenti per riflettere e organizzare i calcoli, e come alcuni dei pattern trattati nei due capitoli precedenti, alcuni di essi sono supportati direttamente dalla sintassi integrata di Python.

Il ben noto motto del linguaggio di programmazione Perl è: “Esistono più modi per farlo”, mentre nello Zen di Python di Tim Peters: “Deve esistere uno - e preferibilmente un solo - modo ovvio di farlo” (per vedere lo Zen di Python, digitate `import this` a un prompt interattivo di Python). Tuttavia, come in qualsiasi linguaggio di programmazione, talvolta esistono due o più modi per fare qualcosa Python, soprattutto dopo l’introduzione delle comprehension (per esempio, usare una comprehension o un ciclo `for`) e dei generatori (per esempio, usare un’espressione generatore o una funzione con un’istruzione `yield`). E come vedremo in questo capitolo, il supporto di Python per le coroutine aggiunge un nuovo modo per compiere determinate attività.

Il pattern Chain of Responsibility

Questo pattern è stato progettato per disaccoppiare il mittente di una richiesta dal destinatario che elabora la richiesta in questione. Perciò, invece di una chiamata diretta da una funzione a un'altra, la prima funzione invia una richiesta a una catena di ricevitori; il primo ricevitore della catena può gestire la richiesta e interrompere la catena (non inoltrando la richiesta), oppure inoltrare la richiesta al ricevitore che lo segue nella catena. Il secondo ricevitore ha le stesse scelte a disposizione, e così via finché si raggiunge l'ultimo ricevitore (che può scegliere se scartare la richiesta o sollevare un'eccezione).

Supponiamo di avere un'interfaccia utente che riceve eventi da gestire. Alcuni degli eventi provengono dall'utente (per esempio gli eventi del mouse e della tastiera), altri provengono dal sistema (per esempio gli eventi del timer). Nei due paragrafi seguenti esamineremo un approccio convenzionale per creare una catena di gestione di eventi, e poi un altro approccio basato su pipeline che fa uso di coroutine.

Una catena convenzionale

In questo paragrafo esamineremo una catena di gestione di eventi convenzionale in cui ciascun evento ha una corrispondente classe di gestione di eventi.

```
handler1 = TimerHandler(KeyHandler(MouseHandler(NullHandler())))
```

Ecco come si potrebbe impostare la catena utilizzando quattro classi handler separate. La catena è illustrata nella Figura 3.1. Poiché scartiamo gli eventi non gestiti, avremmo potuto passare semplicemente `None` (nulla) come argomento di `MouseHandler`.

L'ordine in cui creiamo gli handler non dovrebbe contare, perché ognuno gestisce soltanto gli eventi del tipo per cui è progettato.

```
while True:
    event = Event.next()
    if event.kind == Event.TERMINATE:
        break
    handler1.handle(event)
```

Gli eventi sono normalmente gestiti in un ciclo. In questo caso usciamo dal ciclo e terminiamo l'applicazione nel caso di un evento `TERMINATE`; altrimenti, passiamo l'evento alla catena di gestione.

```
handler2 = DebugHandler(handler1)
```

Qui abbiamo creato un nuovo handler (anche se avremmo potuto semplicemente assegnarlo a `handler1`). Questo handler deve essere il primo nella catena, poiché è utilizzato per “spiare” gli eventi che passano nella catena e per fornire report su di essi, ma non per gestirli (perciò inoltra ogni evento che riceve).

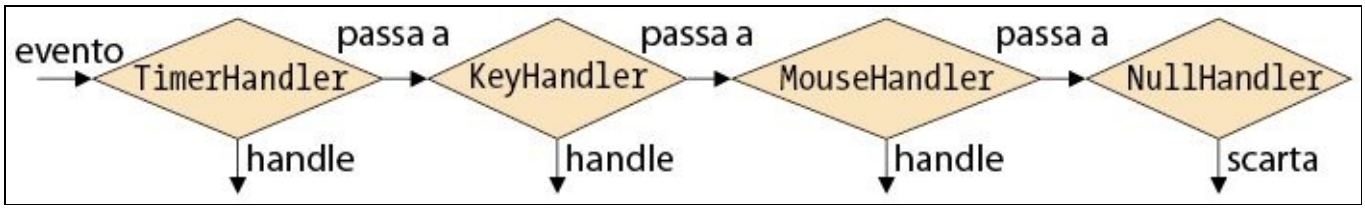


Figura 3.1 Una catena di gestione di eventi.

Ora possiamo richiamare `handler2.handle(event)` nel nostro ciclo, e oltre ai normali gestori di eventi avremo un output di debugging che consente di visualizzare gli eventi ricevuti.

```
class NullHandler:

    def __init__(self, successor=None):
        self.__successor = successor

    def handle(self, event):
        if self.__successor is not None:
            self.__successor.handle(event)
```

Questa classe serve da classe base per i nostri gestori di eventi e fornisce l’infrastruttura per la gestione di eventi. Se un istanza è creata con un handler successore, allora quando tale istanza riceve un evento, si limita a inoltrarlo nella catena al successore. Tuttavia, se non vi è un successore, abbiamo deciso di scartare semplicemente l’evento. Questo è l’approccio standard adottato nella programmazione di GUI (*Graphical User Interface*), anche se avremmo potuto facilmente registrare in un log o sollevare un’eccezione per eventi non gestiti (per esempio se il nostro programma fosse un server).

```
class MouseHandler(NullHandler):

    def handle(self, event):
        if event.kind == Event.MOUSE:
            print("Click:    {}".format(event))
        else:
            super().handle(event)
```

Poiché non abbiamo reimplementato il metodo `__init__()`, verrà usato quello della classe base, perciò la variabile `self.__successor` sarà creata correttamente.

Questa classe handler gestisce soltanto gli eventi a cui è interessata (cioè quelli del tipo `Event.MOUSE`) e passa tutti gli altri al proprio successore nella catena (se ne esiste uno).

Le classi `KeyHandler` e `TimerHandler` (nessuna delle quali è riportata qui) hanno esattamente la stessa struttura di `MouseHandler`, da cui differiscono soltanto per il tipo di eventi a cui rispondono (`Event.KEYPRESS` ed `Event.TIMER` in questo caso) e per le operazioni di gestione che svolgono (stampano messaggi diversi).

```
class DebugHandler(NullHandler):

    def __init__(self, successor=None, file=sys.stdout):
        super().__init__(successor)
        self.__file = file

    def handle(self, event):
        self.__file.write("*DEBUG*: {}\n".format(event))
        super().handle(event)
```

La classe `DebugHandler` è diversa dagli altri handler perché non gestisce mai alcun evento, e deve essere la prima nella catena. Se riceve un file o un oggetto tipo file a cui indirizzare i suoi report, e quando si verifica un evento, questa classe fornisce un report sull'evento e poi lo inoltra.

Una catena basata su coroutine

Un *generatore* è una funzione o metodo che ha una o più espressioni `yield` al posto di `return`. Ogni volta che si raggiunge un'espressione `yield`, viene prodotto il valore risultante e la funzione o metodo viene sospesa con tutto il suo stato intatto. A questo punto la funzione ha "ceduto" il processore (al ricevitore del valore che ha prodotto), perciò, per quanto sia sospesa, non blocca l'esecuzione. Poi, quando la funzione o metodo è utilizzata nuovamente, l'esecuzione riprende dall'istruzione che segue la `yield`. Perciò i valori sono estratti da un generatore iterando su di esso (per esempio, usando `for value in generator:`) oppure richiamando `next()` su di esso.

Una coroutine utilizza la stessa espressione `yield` come generatore, ma ha un comportamento diverso. Una coroutine esegue un ciclo infinito ed entra in sospensione alla sua prima (o unica) espressione `yield`, in attesa che le sia inviato un valore. Se e quando viene inviato un valore, la coroutine lo riceve come valore della sua espressione `yield`, quindi può svolgere tutte le elaborazioni desiderate e, quanto termina, entra in ciclo e di nuovo si sospende in attesa che un valore arrivi alla sua successiva espressione `yield`. Perciò i valori vengono inseriti in una coroutine richiamando i metodi `send()` o `throw()` della coroutine in questione.

In Python, qualsiasi funzione o metodo che contiene una `yield` è un generatore. Tuttavia, utilizzando un decoratore `@coroutine`, e un ciclo infinito, possiamo trasformare un generatore in una coroutine (abbiamo trattato i decoratori e in

particolare il decoratore `@functools.wraps` nel Capitolo 2, e per la precisione nel paragrafo dedicato al pattern Decorator).

```
def coroutine(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        generator = function(*args, **kwargs)
        next(generator)
        return generator
    return wrapper
```

Il wrapper richiama la funzione generatore soltanto una volta e cattura il generatore prodotto nella variabile `generator`. Questo generatore è in realtà la funzione originale con i suoi argomenti e le variabili locali catturate come stato. Poi il wrapper fa avanzare il generatore (solo una volta, usando la funzione integrata `next()`) per l'esecuzione fino alla prima espressione `yield`. Il generatore, con il suo stato catturato, viene poi restituito. La funzione generatore restituita è una coroutine, pronta a ricevere un valore nella sua prima (o unica) espressione `yield`.

Se richiamiamo un generatore, riprenderà l'esecuzione dal punto in cui l'ha lasciata (cioè continuerà dopo l'ultima o unica espressione `yield`). Invece, se inviamo un valore a una coroutine (usando la sintassi `generator.send(value)` di Python), questo valore sarà ricevuto all'interno della coroutine come risultato dell'espressione `yield` corrente, e l'esecuzione riprenderà da quel punto.

Poiché possiamo sia ricevere che inviare valori alle coroutine, queste possono essere utilizzate per creare delle pipeline, tra cui rientrano le catene di gestione di eventi. Inoltre, non abbiamo la necessità di fornire un'infrastruttura di successori, poiché possiamo usare la sintassi per i generatori di Python.

```
pipeline = key_handler(mouse_handler(timer_handler()))
```

In questo caso creiamo la nostra catena (`pipeline`) utilizzando un gruppo di chiamate di funzione annidate. Ogni funzione richiamata è una coroutine, e ciascuna viene eseguita fino alla sua prima (o unica) espressione `yield`, sospendendo l'esecuzione in quel punto, pronta per essere utilizzata di nuovo o per ricevere un valore. Perciò la pipeline viene creata immediatamente.

Anziché avere un handler null, non passiamo nulla all'ultimo handler della catena. Vedremo come funziona il meccanismo quando esamineremo una tipica coroutine handler (`key_handler()`).

```
while True:
    event = Event.next()
    if event.kind == Event.TERMINATE:
        break
    pipeline.send(event)
```

Come per l’approccio convenzionale, una volta che la catena è pronta a gestire eventi, li gestiamo in un ciclo. Poiché ciascuna funzione handler è una coroutine (una funzione generatore), ha un metodo `send()`. Perciò, ogni volta che abbiamo un evento da gestire, lo inviamo alla pipeline. In questo esempio, il valore sarà inviato prima alla coroutine `key_handler()`, che gestirà l’evento o lo inoltrerà altrove. Come in precedenza, l’ordine degli handler spesso non conta.

```
pipeline = debug_handler(pipeline)
```

Questo è il solo caso in cui è importante quale ordine usiamo per un handler. Poiché la coroutine `debug_handler()` serve a “spiare” gli eventi e si limita a inoltrarli, deve essere il primo handler nella catena. Con questa nuova pipeline, possiamo nuovamente elaborare il ciclo di eventi, inviando ciascuno alla pipeline mediante `pipeline.send(event)`.

```
@coroutine
def key_handler(successor=None):
    while True:
        event = (yield)
        if event.kind == Event.KEYPRESS:
            print("Press:     {}".format(event))
        elif successor is not None:
            successor.send(event)
```

Questa coroutine accetta una coroutine successore a cui inviare (o `None`) e inizia l’esecuzione di un ciclo infinito. Il decoratore `@coroutine` garantisce che il `key_handler()` sia eseguito fino alla sua espressione `yield`, perciò, quando si crea la catena `pipeline`, questa funzione ha raggiunto la sua espressione `yield` ed è bloccata, in attesa che lo `yield` produca un valore (inviato). Naturalmente è bloccata soltanto la coroutine, non il programma nel suo complesso.

Una volta che un valore viene inviato a questa coroutine, direttamente o da un’altra coroutine nella pipeline, viene ricevuto come valore `event`. Se l’evento è di un tipo che la coroutine è in grado di gestire (cioè di tipo `Event.KEYPRESS`), viene gestito - in questo esempio viene semplicemente stampato - e non ulteriormente inoltrato. Se, invece, l’evento non è del tipo giusto per questa coroutine, e purché esista una coroutine successore, viene inviato a quest’ultima. Se non c’è un successore, e l’evento non è gestito qui, viene semplicemente scartato.

Dopo la gestione, l’invio o lo scarto di un evento, la coroutine ritorna all’inizio del ciclo `while`, e poi, ancora una volta, attende che lo `yield` produca un valore inviato nella pipeline.

Le coroutine `mouse_handler()` e `timer_handler()` (non mostrate qui), hanno esattamente la stessa struttura di `key_handler()`; le sole differenze sono il tipo di evento gestito e i messaggi stampati.

```
@coroutine
def debug_handler(successor, file=sys.stdout):
    while True:
        event = (yield)
        file.write("*DEBUG*: {}\n".format(event))
        successor.send(event)
```

Il `debug_handler()` attende di ricevere un evento, ne stampa i dettagli e lo invia alla successiva coroutine per la gestione.

Le coroutine utilizzano lo stesso meccanismo dei generatori, ma lavorano in modo molto diverso. Con un normale generatore, estraiamo i valori uno alla vota (per esempio con `for x in range(10):`). Con le coroutine, invece, inseriamo i valori uno alla volta usando `send()`. Grazie a questa versatilità di Python è possibile esprimere molti tipi diversi di algoritmi in un modo molto pulito e naturale. Per esempio, la catena basata su coroutine mostrata in questo paragrafo è stata implementata utilizzando una quantità di codice molto minore rispetto alla catena convenzionale mostrata in precedenza.

Vedremo ancora in azione le coroutine quando tratteremo il pattern Mediator, più avanti in questo stesso capitolo.

Il pattern Chain of Responsibility può naturalmente essere applicato in molti altri contesti diversi da quelli illustrati qui. Per esempio, potremmo utilizzarlo per gestire le richieste in arrivo su un server.

Il pattern Command

Questo pattern è usato per incapsulare comandi come oggetti. Ciò consente, per esempio, di costruire una sequenza di comandi da eseguire in un secondo tempo, o di creare comandi annullabili. Abbiamo già visto un impiego semplice del pattern Command nell'esempio di `ImageProxy` (cfr. il Capitolo 2), e in questo paragrafo facciamo un passo in avanti, creando classi per singoli comandi annullabili e per macro annullabili (sequenze di comandi annullabili).

Iniziamo esaminando un codice che utilizza il pattern Command, poi esamineremo le classi utilizzate (`UndoableGrid` e `Grid`) e il modulo `Command` che fornisce l'infrastruttura per azioni-annullamenti.

```
grid = UndoableGrid(8, 3)     # (1) Vuota
redLeft = grid.create_cell_command(2, 1, "red")
redRight = grid.create_cell_command(5, 0, "red")
redLeft()                     # (2) Celle rosse
redRight.do()                 # Oppure: redRight()
greenLeft = grid.create_cell_command(2, 1, "lightgreen")
greenLeft()                   # (3) Cella verde
rectangleLeft = grid.create_rectangle_macro(1, 1, 2, 2, "lightblue")
rectangleRight = grid.create_rectangle_macro(5, 0, 6, 1, "lightblue")
rectangleLeft()               # (4) Quadrati blu
rectangleRight.do()           # Oppure: rectangleRight()
rectangleLeft.undo()          # (5) Annulla sinistra blu
SquaregreenLeft.undo()        # (6) Annulla sinistra verde
rectangleRight.undo()         # (7) Annulla destra blu
redLeft.undo()                # (8) Annulla celle rosse
redRight.undo()
```

La Figura 3.2 rappresentata la griglia come codice HTML otto volte. La prima mostra la griglia come appare appena creata (cioè quando è vuota); ognuna delle successive immagini mostra lo stato della griglia dopo la creazione e la chiamata di ciascun comando o macro (direttamente o usando il metodo `do()`) e dopo ogni chiamata di `undo()`.

```
class Grid:

    def __init__(self, width, height):
        self.__cells = [["white" for _ in range(height)]
                                 for _ in range(width)]

    def cell(self, x, y, color=None):
        if color is None:
            return self.__cells[x][y]
        self.__cells[x][y] = color

    @property
    def rows(self):
        return len(self.__cells[0])

    @property
    def columns(self):
        return len(self.__cells)
```

Questa classe `Grid` è simile a una classe di immagine, contiene un elenco di liste di nomi di colori.

Il metodo `cell()` serve sia per operazioni di lettura (quando l'argomento `color` è `None`) sia per l'impostazione (quando è fornito un `color`). Le proprietà di sola lettura `rows` e `columns` restituiscono le dimensioni della griglia.

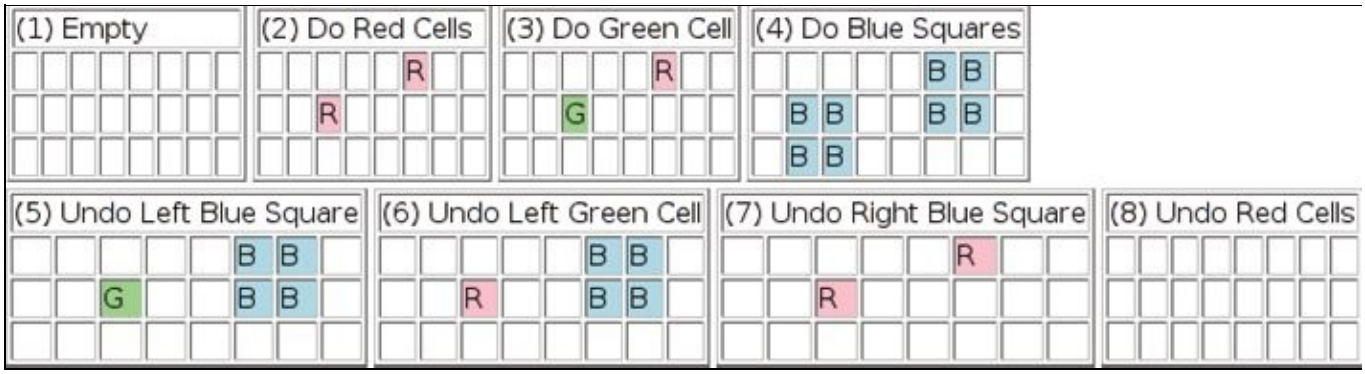


Figura 3.2 Una griglia con azioni e annullamenti.

```
class UndoableGrid(Grid):

    def create_cell_command(self, x, y, color):
        def undo():
            self.cell(x, y, undo.color)
        def do():
            undo.color = self.cell(x, y)   # Sottile!
            self.cell(x, y, color)
    return Command.Command(do, undo, "Cell")
```

Per fare in modo che `Grid` supporti comandi annullabili, abbiamo creato una sottoclasse che aggiunge due metodi, il primo dei quali è mostrato qui.

Ogni comando deve essere del tipo `Command.Command` o `Command.Macro`. Il primo utilizza dei callable di azione e annullamento e una descrizione opzionale, il secondo ha una descrizione opzionale e può avere qualsiasi numero di `Command.Command`.

Nel metodo `create_cell_command()` riceviamo la posizione e il colore della cella da impostare e poi creiamo le due funzioni richieste per creare un `Command.Command`. Entrambi i comandi si limitano a impostare il colore della cella data.

Naturalmente, nel momento in cui sono create le funzioni `do()` e `undo()`, non possiamo sapere quale sarà il colore della cella subito prima che sia applicato il comando `do()`, perciò non sappiamo a quale colore riportarla in caso di annullamento. Abbiamo risolto questo problema recuperando il colore della cella all'interno della funzione `do()` - nel momento in cui viene richiamata - e impostandolo come attributo della funzione `undo()`. Soltanto allora impostiamo il colore nuovo. Notate che il meccanismo funziona perché la funzione `do()` è una chiusura che non si limita a

catturare i parametri `x`, `y` e `color` nel proprio stato, ma anche la funzione `undo()` che è appena stata creata.

Una volta create le funzioni `do()` e `undo()`, creiamo un nuovo `Command.Command` che le incorpora, con in più una semplice descrizione, e restituisce il comando al chiamante.

```
    def create_rectangle_macro(self, x0, y0, x1, y1, color):
        macro = Command.Macro("Rectangle")
        for x in range(x0, x1 + 1):
            for y in range(y0, y1 + 1):
                macro.add(self.create_cell_command(x, y, color))
        return macro
```

Questo è il secondo metodo di `UndoableGrid` per creare comandi con possibilità di annullamento. Crea una macro che a sua volta creerà un rettangolo con le coordinate specificate. Per colorare ogni cella, viene creato un comando apposito utilizzando l'altro metodo della classe (`create_cell_command()`), e tale comando è aggiunto alla macro. Una volta che sono stati aggiunti tutti i comandi, viene restituita la macro.

Come vedremo, comandi e macro supportano entrambi i metodi `do()` e `undo()`. Poiché supportano gli stessi metodi, e le macro contengono comandi, la relazione tra comandi e macro è una variante del pattern Composite (cfr. il Capitolo 2).

```
class Command:

    def __init__(self, do, undo, description=""):
        assert callable(do) and callable(undo)
        self.do = do
        self.undo = undo
        self.description = description

    def __call__(self):
        self.do()
```

Un `Command.Command` si aspetta di ricevere due callable: il primo è il comando "do" e il secondo è il comando "undo" (la funzione `callable()` è integrata in Python 3.3; per le versioni precedenti si può creare una funzione equivalente: `def callable(function): return isinstance(function, collections.Callable)`).

Per eseguire un `Command.Command` è sufficiente richiamarlo (grazie alla nostra implementazione del metodo speciale `__call__()`) oppure, in modo equivalente, richiamare il suo metodo `do()`. Il comando può essere annullato richiamando il suo metodo `undo()`.

```
class Macro:

    def __init__(self, description=""):
        self.description = description
        self.__commands = []
    def add(self, command):
        if not isinstance(command, Command):
            raise TypeError("Expected object of type Command, got {}".
                            format(type(command).__name__))
```

```
        self.__commands.append(command)

    def __call__(self):
        for command in self.__commands:
            command()

    do = __call__

    def undo(self):
        for command in reversed(self.__commands):
            command.undo()
```

La classe `Command.Macro` è utilizzata per incapsulare una sequenza di comandi che dovrebbero essere eseguiti tutti - o annullati tutti - in una singola operazione (parliamo di macro eseguite in una singola operazione, ma in realtà tale operazione non è atomica dal punto di vista della concorrenza, benché possa essere resa tale se si utilizzano lock opportuni). `Command.Macro` presenta la stessa interfaccia di `Command.Command`: metodi `do()` e `undo()`, e la possibilità di chiamata diretta. In più, le macro forniscono un metodo `add()` attraverso il quale è possibile aggiungere comandi `Command.Command`.

Nel caso delle macro, i comandi devono essere annullati in ordine inverso. Per esempio, supponiamo di aver creato una macro e di aver aggiunto i comandi A, B e C. Quando eseguiamo la macro (cioè la richiamiamo direttamente, o richiamiamo il suo metodo `do()`), essa esegue A, poi B e poi C. Perciò, quando richiamiamo `undo()`, dobbiamo eseguire i metodi `undo()` per C, poi per B e poi per A.

In Python le funzioni, i metodi bound e altri callable sono oggetti di classe che possono essere inoltrati e memorizzati in strutture dati quali `list` e `dict`. Per questo si tratta di un linguaggio ideale per l’implementazione del pattern Command. E il pattern stesso può essere utilizzato con grande profitto, come abbiamo visto qui, fornendo funzionalità di esecuzione e annullamento, oltre alla possibilità di supportare le macro e l’esecuzione differita.

Il pattern Interpreter

Questo pattern formalizza due requisiti comuni: fornire un mezzo con cui gli utenti possano inserire valori non stringa nelle applicazioni, e consentire agli utenti di programmare applicazioni.

Al livello di base, un'applicazione riceve stringhe dall'utente - o da altri programmi - e tali stringhe devono essere interpretate (e forse eseguite) in maniera appropriata. Supponiamo per esempio di ricevere dall'utente una stringa che dovrebbe rappresentare un intero. Un modo facile (ma poco saggio) per ottenere il valore dell'intero è il seguente: `i = eval(userCount)`. Questo approccio è pericoloso, perché, anche se speriamo che la stringa sia qualcosa di assolutamente innocente come `"1234"`, potrebbe anche essere qualcosa del tipo `"os.system('rmdir /s /q C:\\\\')"`.

In generale, se ci è fornita una stringa che dovrebbe rappresentare un valore di un tipo di dati specifico, possiamo usare Python per ottenere tale valore in modo diretto e sicuro.

```
try:
    count = int(userCount)
    when = datetime.datetime.strptime(userDate, "%Y/%m/%d").date()
except ValueError as err:
print(err)
```

In questa porzione di codice Python cerca in modo sicuro di effettuare il parsing di due stringhe, una in `int` e l'altra in un `datetime.date`.

Talvolta abbiamo la necessità di andare oltre l'interpretazione di singole stringhe in valori. Per esempio, potremmo avere l'esigenza di fornire a un'applicazione una calcolatrice, o di consentire agli utenti di creare brevi porzioni di codice da applicare a dati. Un approccio diffuso per risolvere questi problemi consiste nel creare un DSL (*Domain Specific Language*). Questi linguaggi possono essere creati direttamente con Python, per esempio scrivendo un parser ricorsivo discendente. Tuttavia, è molto più semplice utilizzare una libreria di parsing esterna come PLY (`http://www.dabeaz.com/ply`), PyParsing (`http://pyparsing.wikispaces.com`) o una delle molte altre disponibili (il parsing, anche con l'utilizzo di PLY e PyParsing, è un argomento trattato nel libro *Programming in Python 3, Second Edition*, dello stesso autore; cfr. la bibliografia per i dettagli).

Se ci troviamo in un ambiente in cui possiamo considerare fidati gli utenti delle nostre applicazioni, possiamo fornire loro l'accesso allo stesso interprete Python. L'IDE (*Integrated Development Environment*) IDLE, incluso in Python, fa

esattamente questo, anche se esegue il codice utente in un processo separato, in modo che eventuali crash non abbiano effetti sul sistema.

Valutazione di espressioni con eval()

La funzione integrata `eval()` valuta una singola stringa come un'espressione (con accesso a qualsiasi contesto globale o locale decidiamo di specificare) e restituisce il risultato. Questo è sufficiente per costruire la semplice applicazione `calculator.py` che esamineremo nel seguito. Iniziamo esaminando alcune interazioni.

```
$ ./calculator.py
Enter an expression (Ctrl+D to quit):    65
A=65
ANS=65
Enter an expression (Ctrl+D to quit):    72
A=65, B=72
ANS=72
Enter an expression (Ctrl+D to quit):    hypotenuse(A, B)
name 'hypotenuse' is not defined
Enter an expression (Ctrl+D to quit):    hypot(A, B)
A=65, B=72, C=97.0
ANS=97.0
Enter an expression (Ctrl+D to quit):    ^D
```

L'utente ha inserito due lati di un triangolo rettangolo e poi ha utilizzato la funzione `math.hypot()` (dopo aver commesso un errore) per calcolare l'ipotenusa. Dopo l'inserimento di ciascuna espressione, il programma `calculator.py` stampa le variabili create fin lì (e accessibili all'utente) e il valore dell'espressione corrente (abbiamo indicato il testo inserito dall'utente in grassetto, dando per scontata la pressione di Invio o Return al termine di ciascuna riga, e Ctrl+D con `^D`).

Per fare in modo che la calcolatrice sia il più possibile comoda, il risultato di ciascuna espressione è memorizzato in una variabile, cominciando con `A`, poi `B` e così via, e ricominciando con `A` se si raggiunge `Z`. Inoltre, abbiamo importato tutte le funzioni e le costanti del modulo `math` (per esempio `hypot()`, `e`, `pi`, `sin()` e così via) nel namespace della calcolatrice, in modo che l'utente possa accedervi senza qualificarle (per esempio scrivendo `cos()` anziché `math.cos()`).

Se l'utente inserisce una stringa che non può essere valutata, la calcolatrice stampa un messaggio di errore e poi ripete il prompt; tutto il contesto esistente rimane intatto.

```
def main():
    quit = "Ctrl+Z,Enter" if sys.platform.startswith("win") else "Ctrl+D"
    prompt = "Enter an expression ({} to quit): ".format(quit)
    current = types.SimpleNamespace(letter="A")
    globalContext = global_context()
    localContext = collections.OrderedDict()
    while True:
        try:
            expression = input(prompt)
            if expression:
```

```
            calculate(expression, globalContext, localContext, current)
    except EOFError:
        print()
        break
```

Abbiamo usato EOF (*End Of File*) per indicare che l'utente ha terminato. Ciò significa che la calcolatrice può essere usata in una pipeline di shell, accettando l'input reindirizzato da un file, oltre all'input interattivo dell'utente.

Abbiamo la necessità di tenere traccia del nome della variabile corrente (`A` o `B` o…) in modo da poterla aggiornare ogni volta che viene eseguito un calcolo. Tuttavia non possiamo semplicemente passarla come una stringa, perché le stringhe sono copiate e non possono essere modificate. Una soluzione scadente è quella di usare una variabile globale, mentre una soluzione migliore e molto più comune consiste nel creare una lista di un solo elemento, per esempio `current = ["A"]`. Tale lista può essere passata come `current` e la stringa può essere letta o modificata utilizzando `current[0]` per accedervi.

Per questo esempio abbiamo scelto un approccio più moderno, creando un piccolo namespace con un singolo attributo (`letter`) il cui valore è `"A"`. Possiamo passare liberamente l'istanza del namespace `current`, e poiché ha un attributo `letter`, possiamo leggere o modificare il valore di tale attributo utilizzando la sintassi `current.letter`.

La classe `types.SimpleNamespace` è stata introdotta in Python 3.3. Con le versioni precedenti, si può ottenere un effetto equivalente scrivendo `current = type("_", (), dict(letter="A"))()`. Così si crea una nuova classe denominata `_` con un singolo attributo denominato `letter` con valore iniziale `"A"`. La funzione integrata `type()` restituisce il tipo di un oggetto, se è richiamata con un solo argomento, oppure crea una nuova classe se si fornisce un nome di classe, una tupla di classi base e un dizionario di attributi. Se passiamo una tupla vuota, la classe base sarà `object`. Poiché non ci serve la classe ma solo un'istanza, dopo aver richiamato `type()`, richiamiamo immediatamente la classe stessa (da qui le parentesi in più) per restituire la sua istanza che assegniamo a `current`.

Python può fornire il contesto globale corrente usando la funzione integrata `globals()`, la quale restituisce un `dict` che possiamo modificare (per esempio aggiungendovi voci, come abbiamo visto precedentemente nel Capitolo 1). Python può anche fornire il contesto locale utilizzando la funzione integrata `locals()`, benché il `dict` restituito da questa funzione non debba essere modificato.

Vogliamo fornire un contesto globale arricchito con le costanti e le funzioni del modulo `math` e un contesto locale inizialmente vuoto. Mentre il contesto globale deve

essere un `dict`, il contesto locale può essere fornito come un `dict` o qualsiasi altro oggetto di corrispondenza. In questo caso abbiamo scelto di utilizzare un `collections.OrderedDict`, cioè un dizionario con le voci ordinate, come contesto locale.

Poiché la calcolatrice può essere usata in modo interattivo, abbiamo creato un ciclo di eventi che termina quando si incontra `EOF`. All'interno del ciclo richiediamo l'input dell'utente con un prompt (indicando anche come uscire) e, se l'utente inserisce un testo qualsiasi, richiamiamo la nostra funzione `calculate()` per svolgere i calcoli e stampare i risultati.

```
import math

def global_context():
    globalContext = globals().copy()
    for name in dir(math):
        if not name.startswith("_"):
            globalContext[name] = getattr(math, name)
    return globalContext
```

Questa funzione ausiliaria inizia creando un `dict` locale con moduli, funzioni e variabili globali del programma, poi itera su tutte le costanti e le funzioni pubbliche del modulo `math` e, per ciascuna, aggiunge il proprio nome non qualificato al `globalContext` e imposta come suo valore la costante o funzione effettiva del modulo `math` a cui si riferisce. Per esempio, quando il nome è `"factorial"`, viene aggiunto come una chiave nel `globalContext`, e il suo valore è impostato come funzione `math.factorial()` (come riferimento). Questo consente agli utenti della calcolatrice di usare nomi non qualificati.

Un approccio più semplice sarebbe stato quello di eseguire `from math import *` e poi utilizzare direttamente `globals()`, senza la necessità del `dict globalContext`. Un tale approccio va bene probabilmente per il modulo `math`, ma l'approccio da noi adottato in questo caso offre un controllo più fine, che potrebbe essere più adatto per altri moduli.

```
def calculate(expression, globalContext, localContext, current):
    try:
        result = eval(expression, globalContext, localContext)
        update(localContext, result, current)
        print(", ".join(["{}={}".format(variable, value)
                for variable, value in localContext.items()]))
        print("ANS={}".format(result))
    except Exception as err:
        print(err)
```

Questa è la funzione in cui chiediamo a Python di valutare l'espressione stringa utilizzando i dizionari di contesto globale e locale che abbiamo creato. Se `eval()` ha successo, aggiorniamo il contesto locale con il risultato e stampiamo le variabili e il risultato stesso. Se si verifica un'eccezione, la stampiamo. Poiché abbiamo usato un

`collections.OrderedDict` per il contesto locale, il metodo `items()` restituisce gli elementi in ordine di inserimento, senza la necessità di effettuare un ordinamento esplicito (se avessimo usato un `dict` normale avremmo dovuto scrivere `sorted(localContext.items())`).

Benché sia solitamente sconsigliato di utilizzare l'eccezione generica `Exception`, in questo caso appare ragionevole, perché l'espressione dell'utente potrebbe sollevare qualsiasi tipo di eccezione.

```
def update(localContext, result, current):
    localContext[current.letter] = result
    current.letter = chr(ord(current.letter) + 1)
    if current.letter > "Z":    # Supportiamo soltanto 26 variabili
        current.letter = "A"
```

Questa funzione assegna il risultato alla successiva variabile nella sequenza ciclica `A` … `Z` `A` … `Z` … Ciò significa che, dopo che l'utente ha inserito 26 espressioni (considerando le lettere dell'alfabeto inglese), il risultato dell'ultima è impostato come valore di `Z`, e poi il risultato della successiva sovrascriverà il valore di `A`, e così via.

La funzione `eval()` valuterà qualsiasi espressione di Python. Questo aspetto è potenzialmente pericoloso, se l'espressione è ricevuta da una fonte non sicura. Un'alternativa è quella di utilizzare la funzione della libreria standard `ast.literal_eval()`, più restrittiva ma anche più sicura.

Valutazione del codice con exec()

La funzione integrata `exec()` può essere utilizzata per eseguire porzioni arbitrarie di codice Python. A differenza di `eval()`, `exec()` non è limitata a una singola espressione e restituisce sempre `None`. Il contesto può essere passato a `exec()` nello stesso modo di `eval()`, tramite dizionari globali e locali. I risultati possono essere recuperati da `exec()` attraverso il contesto locale passato.

In questo paragrafo esaminiamo il programma `genome1.py`, che crea una variabile `genome` (una stringa di lettere A, C, G e T disposte a caso) ed esegue otto porzioni di codice utente con il genoma nel contesto del codice.

```
context = dict(genome=genome, target="G[AC]{2}TT", replace="TCGA")
execute(code, context)
```

Questa porzione di codice mostra la creazione del dizionario `context` con alcuni dati su cui il codice dell'utente potrà lavorare e l'esecuzione di un oggetto `Code` (`code`) dell'utente con il contesto fornito.

```
TRANSFORM, SUMMARIZE = ("TRANSFORM", "SUMMARIZE")

Code = collections.namedtuple("Code", "name code kind")
```

Ci aspettiamo che il codice utente sia fornito nella forma di tuple con nome `Code`, con un nome descrittivo, il codice stesso (sotto forma di stringa) e un tipo, `TRANSFORM` o `SUMMARIZE`. Quando viene eseguito, il codice utente dovrebbe creare un oggetto `result` o un oggetto `error`. Se il tipo di codice è `TRANSFORM`, il `result` dovrebbe essere una nuova stringa genoma, mentre se è `SUMMARIZE`, il `result` dovrebbe essere un numero. Naturalmente cercheremo di fare in modo che il nostro codice sia sufficientemente robusto da saper gestire codice utente che non dovesse soddisfare questi requisiti.

```
def execute(code, context):
    try:
        exec(code.code, globals(), context)
        result = context.get("result")
        error = context.get("error")
        handle_result(code, result, error)
    except Exception as err:
        print("'{}' raised an exception: {}\n".format(code.name, err))
```

Questa funzione esegue la chiamata di `exec()` sul codice utente, usando il contesto globale del programma e il contesto locale fornito. Poi cerca di recuperare gli oggetti `result` ed `error`, uno dei quali dovrebbe essere stato creato dal codice utente, e li passa alla funzione personalizzata `handle_result()`.

Esattamente come nell'esempio di `eval()` nel paragrafo precedente, abbiamo usato l'eccezione `Exception`, anche se normalmente si consiglia di evitarla, perché il codice utente potrebbe sollevare qualsiasi tipo di eccezione.

```
def handle_result(code, result, error):
    if error is not None:
        print("'{}' error: {}".format(code.name, error))
    elif result is None:
        print("'{}' produced no result".format(code.name))
    elif code.kind == TRANSFORM:
        genome = result
        try:
            print("'{}' produced a genome of length {}".format(code.name,
                  len(genome)))
        except TypeError as err:
            print("'{}' error: expected a sequence result: {}".format(
                  code.name, err))
    elif code.kind == SUMMARIZE:
        print("'{}' produced a result of {}".format(code.name, result))
print()
```

Se l'oggetto `error` non è `None`, viene stampato; altrimenti, se `result` è `None`, stampiamo un messaggio che indica che non è stato prodotto alcun risultato. Se abbiamo un `result` e il tipo specificato per il codice utente è `TRANSFORM`, assegniamo `result` a `genome`, e in questo caso stampiamo semplicemente la nuova lunghezza del genoma. Il blocco `try ... except` serve a proteggere il programma da un eventuale errore nel codice utente

(per esempio la restituzione di un valore singolo anziché di una stringa o un'altra sequenza per un tipo `TRANSFORM`). Se il tipo del `result` è `SUMMARIZE`, stampiamo semplicemente una riga di riepilogo contenente il risultato.

Il programma `genome1.py` ha otto elementi `Code`: i primi due (che vedremo tra breve) producono risultati corretti, il terzo presenta un errore di sintassi, il quarto restituisce un errore, il quinto non fa nulla, nel sesto il tipo è impostato in modo errato, il settimo richiama `sys.exit()` e l'ottavo non viene mai raggiunto perché il settimo termina il programma. L'output è riportato di seguito.

```
$ ./genome1.py
'Count' produced a result of 12

'Replace' produced a genome of length 2394

'Exception Test' raised an exception: invalid syntax (<string>, line 4)

'Error Test' error: 'G[AC]{2}TT' not found

'No Result Test' produced no result

'Wrong Kind Test' error: expected a sequence result: object of type 'int' has no len()
```

Come si vede chiaramente dall'output, poiché il codice utente viene eseguito nello stesso interprete del programma stesso, può terminare o mandare in crash quest'ultimo.

```
Code("Count",
"""
import re
matches = re.findall(target, genome)
if matches:
    result = len(matches)
else:
    error = "'{}' not found".format(target)
""", SUMMARIZE)
```

Questo è l'elemento di codice "Count". Il codice fa molto più di quanto sarebbe possibile in una singola espressione del tipo che `eval()` sarebbe in grado di gestire. Le stringhe `target` e `genome` sono prese dall'oggetto `context` passato come contesto locale di `exec()`, ed è in questo stesso oggetto `context` è contenuta implicitamente qualsiasi nuova variabile (come `result` ed `error`).

```
Code("Replace",
"""

import re
result, count = re.subn(target, replace, genome)
if not count:
    error = "no '{}' replacements made".format(target)
""", TRANSFORM)
```

L'elemento di codice "Replace" esegue una semplice trasformazione sulla stringa `genome`, sostituendo le sottostringhe non sovrapposte che corrispondono all'espressione regolare `target` con la stringa `replace`.

La funzione `re.subn()` (e il metodo `regex.subn()`) esegue sostituzioni esattamente nello stesso modo di `re.sub()` (e `regex.sub()`). Tuttavia, mentre la funzione (e metodo) `sub()` restituisce una stringa in cui sono state effettuate tutte le sostituzioni, la funzione (e metodo) `subn()` restituisce sia la stringa, sia un conteggio delle sostituzioni effettuate.

Le funzioni `execute()` e `handle_result()` del programma `genome1.py` sono facili da comprendere e da usare, ma da un punto di viste il programma è fragile: se il codice utente va in crash, o semplicemente richiama `sys.exit()`, il nostro programma terminerà. Nel paragrafo seguente esamineremo una soluzione per questo problema.

Valutazione del codice con un sottoprocesso

Una possibile soluzione al problema di eseguire codice utente senza rischiare di compromettere l'applicazione è quella di eseguirlo in un processo separato. I programmi `genome2.py` e `genome3.py` di questo paragrafo mostrano come possiamo eseguire un interprete Python in un sottoprocesso, fornirgli un programma da eseguire attraverso lo standard input e recuperare i risultati dallo standard output.

Abbiamo fornito ai programmi `genome2.py` e `genome3.py` esattamente gli stessi otto elementi di codice del programma `genome1.py`. Di seguito è riportato l'output di `genome2.py` (quello di `genome3.py` è identico):

```
$ ./genome2.py
'Count' produced a result of 12

'Replace' produced a genome of length 2394

'Exception Test' has an error on line 3
    if genome[i] = "A":
                 ^
SyntaxError: invalid syntax

'Error Test' error: 'G[AC]{2}TT' not found

'No Result Test' produced no result

'Wrong Kind Test' error: expected a sequence result: object of type 'int' has
no len()

'Termination Test' produced no result

'Length' produced a result of 2406
```

Notate che, anche se il settimo elemento `Code` richiama `sys.exit()`, il programma `genome2.py` continua l'esecuzione, limitandosi a indicare che quella porzione di codice non ha prodotto alcun risultato, e poi passando all'esecuzione del codice "Length" (il programma `genome1.py` è stato terminato dalla chiamata di `sys.exit()`, perciò l'ultima sua riga di output era `"...error: expected a sequence…"`).

Un altro punto da notare è che `genome2.py` produce descrizioni di errori molto migliori (per esempio nell'errore di sintassi del codice "Exception Test").

```
context = dict(genome=genome, target="G[AC]{2}TT", replace="TCGA")
execute(code, context)
```

La creazione del contesto e l'esecuzione del codice utente in esso si effettua esattamente allo stesso modo che abbiamo visto per il programma `genome1.py`.

```
def execute(code, context):
    module, offset = create_module(code.code, context)
    with subprocess.Popen([sys.executable, "-"], stdin=subprocess.PIPE,
            stdout=subprocess.PIPE, stderr=subprocess.PIPE) as process:
        communicate(process, code, module, offset)
```

Questa funzione crea una stringa di codice (`module`) contenente il codice dell'utente più del codice di supporto che vedremo tra breve. L'`offset` è il numero di righe che abbiamo aggiunto prima del codice utente, e che ci aiuterà a fornire numeri di riga precisi nei messaggi di errore. La funzione poi avvia un sottoprocesso in cui esegue una nuova istanza dell'interprete Python, il cui nome è in `sys.executable`, e il cui argomento `"-"` (trattino) significa che l'interprete si aspetterà di eseguire codice Python inviato al suo `sys.stdin` (la funzione `subprocess.Popen()` ha aggiunto il supporto per i context manager, cioè l'istruzione `with`, in Python 3.2).

L'interazione con il processo, che include anche l'invio del codice `module`, è gestita dalla nostra funzione personalizzata `communicate()`.

```
def create_module(code, context):
    lines = ["import json", "result = error = None"]
    for key, value in context.items():
        lines.append("{} = {!r}".format(key, value))
    offset = len(lines) + 1
    outputLine = "\nprint(json.dumps((result, error)))"
    return "\n".join(lines) + "\n" + code + outputLine, offset
```

Questa funzione crea un elenco di righe che formeranno un nuovo modulo Python destinato all'esecuzione da parte di un interprete Python in un sottoprocesso. La prima riga importa il modulo `json` che utilizzeremo per restituire risultati al processo di avvio (cioè al programma `genome2.py`). La seconda riga inizializza le variabili `result` e `error` per assicurarsi che esistano. Poi aggiungiamo una riga per ognuna delle variabili contestuali. Infine, memorizziamo `result` e `error` (che il codice utente potrebbe aver modificato) all'interno di una stringa usando JSON (*JavaScript Object Notation*). Vi sarà la stampa su `sys.stdout` dopo l'esecuzione del codice utente.

```
UTF8 = "utf-8"

def communicate(process, code, module, offset):
    stdout, stderr = process.communicate(module.encode(UTF8))
    if stderr:
        stderr = stderr.decode(UTF8).lstrip().replace(", in <module>", ":")
```

```
        stderr = re.sub(", line (\d+)",
                lambda match: str(int(match.group(1)) - offset), stderr)
        print(re.sub(r'File."[^"]+?"', "'{}' has an error on line "
                .format(code.name), stderr))
        return
    if stdout:
        result, error = json.loads(stdout.decode(UTF8))
        handle_result(code, result, error)
        return
    print("'{}' produced no result\n".format(code.name))
```

La funzione `communicate()` inizia inviando il codice di modulo creato in precedenza all'interprete Python del sottoprocesso per l'esecuzione, e poi si ferma in attesa che siano prodotti i risultati. Una volta che l'interprete ha terminato l'esecuzione, il suo standard output e il suo standard error output sono raccolti nelle nostre variabili locali `stdout` e `stderr`. Notate che tutte le comunicazioni hanno luogo utilizzando byte grezzi, da qui la necessità di codificare la stringa `module` in UTF-8.

Se in output è riportato un errore (cioè se è stata sollevata un'eccezione, o se viene scritto qualcosa su `sys.stderr`), sostituiamo il numero di riga segnalato (che tiene conto delle righe che abbiamo aggiunto prima del codice dell'utente) con il numero di riga effettivo nel codice utente, e sostituiamo il testo `"File "<stdin>""` con il nome dell'oggetto `Code`, poi stampiamo il testo dell'errore come una stringa.

La chiamata di `re.sub()` cattura le cifre del numero di riga con `(\d+)` e le sostituisce con il risultato della chiamata della funzione `lambda` fornita come suo secondo argomento (spesso forniamo una stringa come secondo argomento, ma in questo caso abbiamo bisogno di svolgere dei calcoli). La funzione `lambda` converte le cifre in un intero e sottrae l'offset, quindi restituisce il nuovo numero di riga come stringa che va a sostituire l'originale. In questo modo ci si assicura che il numero di riga del messaggio di errore corrisponda al codice dell'utente, indipendentemente dal numero di righe che abbiamo inserito prima della riga corrispondente al momento di creare il modulo che abbiamo inviato all'interprete.

Se non sono presenti errori in output, ma c'è standard output, decodifichiamo i byte di output in una stringa (che dovrebbe essere in formato JSON) e ne effettuiamo il parsing per ottenere oggetti Python, in questo caso una tupla di due elementi: un `result` e un `error`. Poi richiamiamo la nostra funzione personalizzata `handle_result()` (tale funzione è identica in `genome1.py`, `genome2.py` e `genome3.py`, ed è stata già riportata precedentemente)

Il codice utente del programma `genome2.py` è identico a quello di `genome1.py`, anche se per `genome2.py` forniamo un codice di supporto aggiuntivo prima e dopo il codice utente. L'uso del formato JSON per restituire i risultati è sicuro e comodo, ma limita i

tipi di dati che possono essere restituiti (per esempio il tipo di `result`) a `dict`, `list`, `str`, `int`, `float`, `bool` o `None`, e `dict` o `list` possono contenere soltanto oggetti di questi tipi.

Il programma `genome3.py` è quasi identico a `genome2.py`, ma restituisce i risultati in un pickle. Questo significa che è possibile utilizzare la maggior parte dei tipi di dati di Python.

```
def create_module(code, context):
    lines = ["import pickle", "import sys", "result = error = None"]
    for key, value in context.items():
        lines.append("{} = {!r}".format(key, value))
    offset = len(lines) + 1
    outputLine = "\nsys.stdout.buffer.write(pickle.dumps((result, error)))"
    return "\n".join(lines) + "\n" + code + outputLine, offset
```

Questa funzione è molto simile alla versione di `genome2.py`. Una differenza secondaria è che dobbiamo importare `sys`, mentre la differenza principale è che, mentre i metodi `loads()` e `dumps()` del modulo `json` operano su `str`, le funzioni equivalenti del modulo `pickle` operano su `bytes`. Perciò, in questo caso dobbiamo scrivere i byte grezzi direttamente sul buffer di `sys.stdout`, per evitare che siano erroneamente codificati.

```
def communicate(process, code, module, offset):
    stdout, stderr = process.communicate(module.encode(UTF8))
    ...
    if stdout:
        result, error = pickle.loads(stdout)
        handle_result(code, result, error)
        return
```

Il metodo `communicate()` del programma `genome3.py` è identico a quello di `genome2.py` eccetto per la riga con la chiamata del metodo `loads()`. Per i dati JSON abbiamo dovuto decodificare i byte in una stringa `str` UTF-8, ma qui lavoriamo direttamente sui byte grezzi.

Utilizzando `exec()` per eseguire porzioni di codice Python arbitrarie ricevute dall'utente o da altri programmi, si consente a tale codice di accedere a tutta la potenza dell'interprete Python, e alla sua intera libreria standard. Ed eseguendo il codice utente in un interprete Python separato all'interno di un sottoprocesso, siamo in grado di proteggere il nostro programma da eventuali crash o terminazioni. Tuttavia, non siamo realmente in grado di impedire al codice utente di attuare comportamenti maligni. Per eseguire codice non fidato dovremmo utilizzare qualche tipo di sandbox, per esempio quella fornita dall'interprete Python PyPy (`http://pypy.org`).

Per alcuni programmi, il fatto di arrestarsi rimanendo in attesa che il codice utente completi la sua esecuzione potrebbe essere accettabile, ma si corre il rischio di

un’attesa “eterna” nel caso in cui il codice utente avesse un bug (per esempio un ciclo infinito). Una possibile soluzione sarebbe quella di creare il sottoprocesso in un thread separato e usare un timer nel thread principale. Se il timer scade, potremmo forzare la terminazione del sottoprocesso e segnalare il problema all’utente. La programmazione concorrente è introdotta nel Capitolo 4.

Il pattern Iterator

Questo pattern fornisce un modo per accedere sequenzialmente agli elementi di una collezione o di un oggetto aggregato senza esporre i dettagli interni dell'implementazione corrispondente. È talmente utile che Python ne ha integrato il supporto, e fornisce metodi speciali che possiamo implementare nelle nostre classi per fare in modo che supportino l'iterazione.

L'iterazione può essere supportata soddisfacendo il protocollo sequenza, oppure utilizzando la forma con due argomenti della funzione integrata `iter()`, o ancora, rispettando il protocollo iteratore. Vedremo degli esempi di tutti questi approcci nei paragrafi che seguono.

Iteratori per protocollo sequenza

Un modo per fornire supporto di iteratori alle nostre classi è quello di fare in modo che supportino il protocollo sequenza. Questo significa che dobbiamo implementare un metodo speciale `__getitem__()` che possa accettare un argomento indice intero che inizia da zero e che sollevi un'eccezione `IndexError` se non è possibile alcuna ulteriore iterazione.

```
for letter in AtoZ():
    print(letter, end="")
print()

for letter in iter(AtoZ()):
    print(letter, end="")
print()

ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Queste due porzioni di codice creano un oggetto `AtoZ()` e poi iterano su di esso. L'oggetto prima restituisce la stringa di un solo carattere `"A"`, poi `"B"` e così via fino a `"Z"`. L'oggetto potrebbe essere stato reso iterabile in molti modi, ma in questo caso abbiamo fornito un metodo `__getitem__()`, come vedremo tra breve.

Nel secondo ciclo abbiamo usato la funzione integrata `iter()` per ottenere un iteratore a un'istanza della classe `AtoZ`. Naturalmente ciò non è necessario in questo caso, ma come vedremo tra breve (e in altri punti del libro), `iter()` trova le sue applicazioni.

```
class AtoZ:

    def __getitem__(self, index):
        if 0 <= index < 26:
```

```
            return chr(index + ord("A"))
        raise IndexError()
```

Questa è la classe `AtoZ` completa. Abbiamo fornito un metodo `__getitem__()` che soddisfa il protocollo sequenza. Quando un oggetto di questa classe viene iterato, alla ventisettesima iterazione solleva un `IndexError`. Se questo si verifica all'interno di un ciclo `for`, l'eccezione viene scartata e il ciclo viene terminato in modo pulito; l'esecuzione quindi riprende dalla prima istruzione dopo il ciclo.

Iteratori con la funzione iter() a due argomenti

Un altro modo per fornire supporto dell'iterazione è quello di usare la funzione integrata `iter()`, passandole due argomenti invece di uno solo. Quando si usa questa forma, il primo argomento dev'essere un callable (una funzione, un metodo bound o qualsiasi altro oggetto callable), e il secondo argomento deve essere un valore sentinella. Quando si utilizza questa forma, il callable viene richiamato a ciascuna iterazione (senza argomenti) e l'iterazione si arresta soltanto se il callable solleva un'eccezione `StopIteration` o se restituisce il valore sentinella.

```
for president in iter(Presidents("George Bush"), None):
    print(president, end=" * ")
print()

for president in iter(Presidents("George Bush"), "George W. Bush"):
    print(president, end=" * ")
print()

George Bush * Bill Clinton * George W. Bush * Barack Obama *
George Bush * Bill Clinton *
```

La chiamata di `Presidents()` crea un'istanza della classe `Presidents` e, grazie all'implementazione del metodo speciale `__call__()`, tali istanze sono callable. Perciò, in questo caso creiamo un oggetto `Presidents` che è un callable (come richiesto dalla forma con due argomenti della funzione integrata `iter()`) e forniamo come sentinella `None`. È necessario fornire una sentinella, anche `None`, affinché Python sappia che deve utilizzare la funzione `iter()` con due argomenti e non la versione con un argomento solo.

Il costruttore `Presidents` crea un callable che restituirà a turno ciascun presidente degli Stati Uniti, a partire da George Washington o, opzionalmente, dal presidente che indichiamo. In questo caso abbiamo indicato di partire da George Bush. Alla prima iterazione abbiamo usato come sentinella `None` per indicare "vai fino all'ultimo", che al momento in cui scriviamo è Barack Obama. Alla seconda iterazione abbiamo fornito il nome di un presidente come sentinella; questo significa

che il callable invierà in output ogni presidente dal primo fino a quello che precede la sentinella.

```
class Presidents:

    __names = ("George Washington", "John Adams", "Thomas Jefferson",
               ...
               "Bill Clinton", "George W. Bush", "Barack Obama")

    def __init__(self, first=None):
        self.index = (-1 if first is None else
                Presidents.__names.index(first) - 1)

    def __call__(self):
        self.index += 1
        if self.index < len(Presidents.__names):
            return Presidents.__names[self.index]
        raise StopIteration()
```

La classe `Presidents` mantiene un elenco statico (valido sulla classe) `__names` con i nomi di tutti i presidenti degli Stati Uniti. Il metodo `__init__()` imposta l'indice iniziale a uno meno il primo presidente in elenco o il presidente specificato dall'utente.

Le istanze di qualsiasi classe che implementi il metodo speciale `__call__()` sono callable. E quando viene richiamata una tale istanza, viene eseguito questo metodo `__call__()` (nei linguaggi che non supportano l'uso di funzioni come oggetti di prima classe, le istanze callable sono chiamate *funtori*).

Nel metodo speciale `__call__()` di questa classe restituiamo il nome del successivo presidente in elenco, oppure solleviamo un'eccezione `StopIteration`. Nella prima iterazione, in cui la sentinella era `None`, non si è mai raggiunta la sentinella (poiché `__call__()` non restituisce mai `None`), ma l'iterazione si è arrestata senza problemi perché, una volta esauriti i presidenti, abbiamo sollevato l'eccezione `StopIteration`. Nella seconda iterazione, invece, non appena il presidente sentinella è stato restituito alla funzione integrata `iter()`, la funzione stessa ha sollevato un'eccezione `StopIteration` per terminare il ciclo.

Iteratori del protocollo iteratore

Probabilmente il modo più facile per fornire supporto agli iteratori nelle nostre classi è fare in modo che supportino il protocollo iteratore. Questo protocollo richiede che una classe implementi il metodo speciale `__iter__()` e che tale metodo restituisca un oggetto iteratore.

L'oggetto iteratore deve avere un proprio metodo `__iter__()` che restituisce l'iteratore stesso e un metodo `__next__()` che restituisce l'elemento successivo, oppure

sollevi un'eccezione `StopIteration` se non vi sono altri elementi. Il ciclo `for` e l'istruzione `in` di Python utilizzano questo protocollo dietro le quinte. Un modo semplice per implementare un metodo `__iter__()` è quello di renderlo un generatore, oppure di fare in modo che restituisca un generatore, perché i generatori soddisfano i protocollo di iterazione (cfr. l'inizio di questo stesso capitolo per ulteriori informazioni sui generatori).

Nel seguito creiamo una semplice classe *bag* (letteralmente classe "borsa", ma chiamata anche *multiset*). Un bag è una classe collezione che è simile a un `set` ma consente la presenza di elementi duplicati. Un esempio è mostrato nella Figura 3.3. Naturalmente faremo in modo che il bag sia iterabile, e mostreremo tre modi per farlo. Tutto il codice è ripreso da `Bag1.py` eccetto dove è indicato altrimenti.

```
class Bag:

    def __init__(self, items=None):
        self.__bag = {}
        if items is not None:
            for item in items:
                self.add(item)
```

I dati del bag sono memorizzati nel dizionario privato `self.__bag`. Le chiavi del dizionario possono essere qualunque hashable (cioè elementi del bag), e i valori sono conteggi (per esempio di quanti elementi si trovano nella borsa). Gli utenti possono aggiunti alcuni elementi iniziali a un bag appena creato, se lo desiderano.

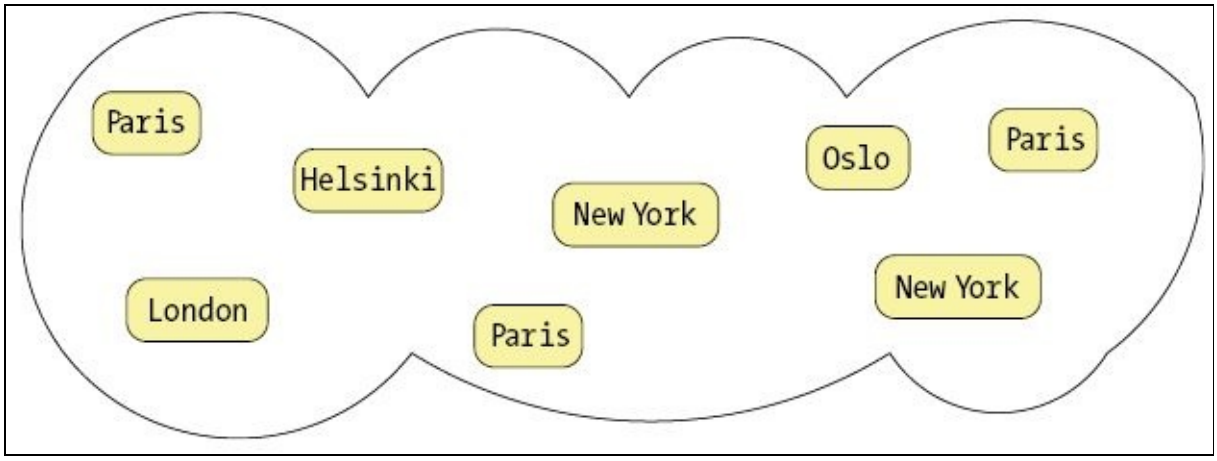


Figura 3.3 Un bag è una collezione non ordinata di valori con duplicati permessi.

```
    def add(self, item):
        self.__bag[item] = self.__bag.get(item, 0) + 1
```

Poiché `self.__bag` non è un `collections.defaultdict`, dobbiamo prestare attenzione a incrementare soltanto un elemento già esistente, altrimenti genereremmo un'eccezione `KeyError`. Utilizziamo il metodo `dict.get()` per recuperare il conteggio di un elemento esistente, o `0` se tale elemento non esiste, e impostare il dizionario in

modo che abbia un elemento con questo numero più 1, creando l'elemento se necessario.

```
def __delitem__(self, item):
    if self.__bag.get(item) is not None:
        self.__bag[item] -= 1
        if self.__bag[item] <= 0:
            del self.__bag[item]
    else:
        raise KeyError(str(item))
```

Se si fa un tentativo di eliminare un elemento che non si trova nel bag, solleviamo un `KeyError` contenente l'elemento nella forma di stringa. D'altra parte, se l'elemento si trova nel bag, iniziamo a ridurne il conteggio. Se il totale scende a zero o meno, lo cancelliamo dal bag.

Non abbiamo implementato i metodi speciali `__getitem__()` o `__setitem__()`, perché nessuno di essi ha senso per i bag (infatti i bag non sono ordinati). Utilizziamo invece `bag.add()` per aggiungere elementi, `del bag[item]` per eliminarli e `bag.count(item)` per verificare quanti elementi di un tipo particolare sono contenuti nel bag.

```
def count(self, item):
    return self.__bag.get(item, 0)
```

Questo metodo restituisce semplicemente il numero di occorrenze del dato elemento che sono presenti nel bag, o zero se l'elemento non è nel bag. Un'alternativa del tutto ragionevole sarebbe quella di sollevare un `KeyError` per i tentativi di contare un elemento che non è nel bag, cosa che si può fare semplicemente modificando il corpo del metodo in `return self.__bag[item]`.

```
def __len__(self):
    return sum(count for count in self.__bag.values())
```

Questo metodo è piuttosto sottile, perché dobbiamo contare tutti gli elementi duplicati nel bag, separatamente. A questo scopo, iteriamo su tutti i valori del bag (cioè contiamo i suoi elementi) e li sommiamo utilizzando la funzione integrata `sum()`.

```
def __contains__(self, item):
    return item in self.__bag
```

Questo metodo restituisce `True` se il bag contiene almeno uno dei dati elementi (poiché se un elemento è nel bag, il suo conteggio è almeno 1); altrimenti restituisce `False`.

Abbiamo visto ormai tutti i metodi del bag tranne quelli per il supporto dell'iterazione. Per prima cosa esamineremo il metodo `Bag.__iter__()` del modulo `Bag1.py`.

```
def __iter__(self):   # Crea inutilmente una lista di elementi!
    items = []
    for item, count in self.__bag.items():
        for _ in range(count):
            items.append(item)
    return iter(items)
```

Questo metodo rappresenta un primo tentativo. Costruisce una lista di elementi - quanti ne sono indicati da `count` - e poi restituisce un iteratore per la lista. Nel caso di un bag grande, questo potrebbe portare alla creazione di una lista molto grande, cosa piuttosto inefficiente, perciò esamineremo due approcci migliori.

```
def __iter__(self):
    for item, count in self.__bag.items():
        for _ in range(count):
            yield item
```

Questo codice è tratto dal modulo `Bag2.py` ed è l'unico metodo diverso rispetto alla classe `Bag` di `Bag1.py`.

In questo caso iteriamo sugli elementi del bag, recuperando ciascuno di essi e il suo conteggio, e restituendo ciascun elemento `count` volte. C'è un lieve sovraccarico per rendere questo metodo un generatore, ma è indipendente dal numero di elementi; naturalmente non è necessario creare elenchi separati, perciò questo metodo è molto più efficiente della versione di `Bag1.py`.

```
def __iter__(self):
    return (item for item, count in self.__bag.items()
        for _ in range(count))
```

Questa è la versione del metodo `Bag.__iter__()` del modulo `Bag3.py`. È praticamente identica alla versione del modulo `Bag2.py`, ma invece di impostare il metodo come generatore, restituisce un'espressione generatore.

Anche se le implementazioni di bag riportate in questo libro funzionano perfettamente, ricordate che la libreria standard ha la propria implementazione: `collections.Counter`.

Il pattern Mediator

Questo pattern fornisce un mezzo per creare un oggetto - il *mediatore* - in grado di incapsulare le interazioni tra altri oggetti. Ciò consente di ottenere interazioni tra oggetti che non hanno una diretta conoscenza l'uno dell'altro. Per esempio, se si fa clic su un oggetto pulsante, esso deve semplicemente indicarlo al mediatore; spetta a quest'ultimo notificare il fatto a qualsiasi altro oggetto che possa essere interessato al clic. Un mediatore per un form con del testo e qualche pulsante, e alcuni metodi associati, è illustrato nella Figura 3.4.

Il pattern Mediator è di grande utilità nella programmazione di GUI. In effetti, tutti i toolkit GUI disponibili per Python (come Tkinter, PyQt/PySide, PyGObject, wxPython) forniscono qualcosa di equivalente. Vedremo alcuni esempi con Tkinter nel Capitolo 7.

Nel seguito esamineremo due approcci all'implementazione di un mediatore. Il primo è piuttosto convenzionale, mentre il secondo utilizza delle coroutine. Entrambi fanno uso di classi `Form`, `Button` e `Text` (di cui vedremo le implementazioni) per un toolkit di interfaccia utente fittizio.

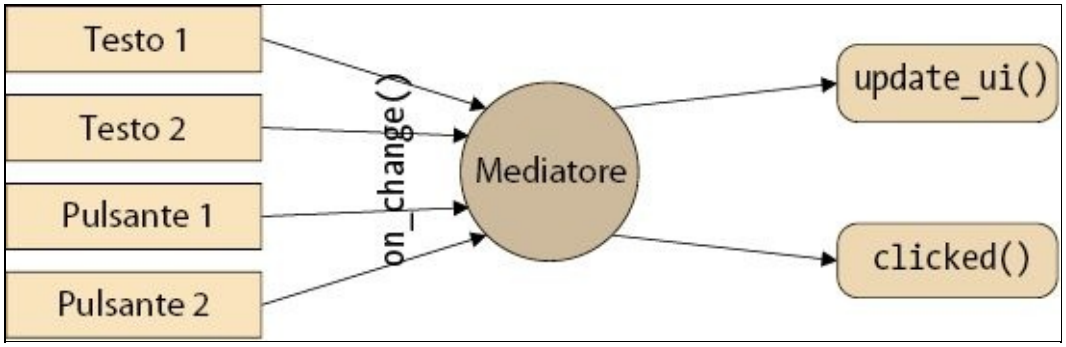


Figura 3.4 Il mediatore per un widget di un form.

Un mediatore convenzionale

In questo paragrafo creiamo un mediatore convenzionale, ovvero una classe in grado di orchestrare le interazioni, in questo caso per un form. Tutto il codice riportato qui è tratto dal programma `mediator1.py`.

```
class Form:

    def __init__(self):
        self.create_widgets()
        self.create_mediator()
```

Come la maggior parte delle funzione e dei metodi presentati in questo libro, anche questo metodo è stato spietatamente rimodellato.

```
    def create_widgets(self):
        self.nameText = Text()
        self.emailText = Text()
        self.okButton = Button("OK")
        self.cancelButton = Button("Cancel")
```

Questo form ha due widget per l'inserimento di testo, destinati al nome e all'indirizzo email dell'utente, e due pulsanti: *OK* e *Cancel*. Naturalmente, in un'interfaccia utente vera dovremmo includere widget etichetta e provvedere a una disposizione accurata, ma questo esempio ci serve soltanto per illustrare il pattern Mediator, perciò non ci occupiamo di questi aspetti. Esamineremo tra breve le classi `Text` e `Button`.

```
    def create_mediator(self):
        self.mediator = Mediator(((self.nameText, self.update_ui),
            (self.emailText, self.update_ui),
            (self.okButton, self.clicked),
            (self.cancelButton, self.clicked)))
        self.update_ui()
```

Creiamo un singolo oggetto mediatore per l'intero form. Questo oggetto riceve una o più coppie widget-callable, che descrivono le relazioni che il mediatore deve supportare. In questo caso tutti i callable sono metodi bound (cfr. il riquadro dedicato ai metodi bound e unbound nel Capitolo 2). In questo caso indichiamo che, se il testo di uno dei widget per l'inserimento di testo cambia, occorre richiamare il metodo `Form.update_ui()`; e se uno dei pulsanti viene premuto con un clic, occorre richiamare il metodo `Form.clicked()`. Dopo aver creato il mediatore, richiamiamo il metodo `update_ui()` per inizializzare il form.

```
    def update_ui(self, widget=None):
        self.okButton.enabled = (bool(self.nameText.text) and
                                 bool(self.emailText.text))
```

Questo metodo abilita il pulsante *OK* se entrambi i widget per l'inserimento di testo contengono del testo; altrimenti lo disabilita. Naturalmente va richiamato ogni volta che il testo di uno dei due widget viene modificato.

```
    def clicked(self, widget):
        if widget == self.okButton:
            print("OK")
        elif widget == self.cancelButton:
            print("Cancel")
```

Questo metodo deve essere richiamato ogni volta che si fa clic su un pulsante. In un'applicazione reale farebbe qualcosa di più interessante che stampare il testo del pulsante premuto.

```
class Mediator:

    def __init__(self, widgetCallablePairs):
        self.callablesForWidget = collections.defaultdict(list)
        for widget, caller in widgetCallablePairs:
            self.callablesForWidget[widget].append(caller)
            widget.mediator = self
```

Questo è il primo dei due metodi della classe `Mediator`. Vogliamo creare un dizionario le cui chiavi sono widget e i cui valori sono liste di uno o più callable. A questo scopo utilizziamo un dizionario di default. Quando accediamo a un elemento di un dizionario di default, se l'elemento non è presente nel dizionario in questione, viene creato e aggiunto con il valore creato dal callable fornito al dizionario. In questo caso abbiamo fornito al dizionario un oggetto `list`, che, quando è richiamato, restituisce una nuova lista vuota. Perciò, la prima volta che un particolare widget viene cercato nel dizionario, viene inserito un nuovo elemento con il widget come chiave e una lista vuota come valore, e aggiungiamo immediatamente il chiamante alla lista. E ogni volta successiva che un widget viene cercato nel dizionario, il chiamante viene aggiunto alla lista esistente. Inoltre impostiamo l'attributo del mediatore per il widget (creandolo, se necessario) al mediatore attuale (`self`).

Il metodo aggiunge i metodi bound nell'ordine in cui appaiono nelle coppie; se non ci interessasse l'ordine potremmo passare `set` anziché `list` al momento di creare il dizionario di default, e utilizzare `set.add()` al posto di `list.append()` per aggiungere i metodi bound.

```
    def on_change(self, widget):
        callables = self.callablesForWidget.get(widget)
        if callables is not None:
            for caller in callables:
                caller(widget)
        else:
            raise AttributeError("No on_change() method registered for {}"
                    .format(widget))
```

Ogni volta che un oggetto mediato, cioè ogni widget passato a un `Mediator`, presenta un cambiamento dello stato, dovrebbe richiamare il metodo `on_change()` del proprio mediatore. Tale metodo poi recupera e richiama ogni metodo bound associato al widget.

```
class Mediated:

    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
```

Questa è una classe di comodo progettata per essere ereditata dalle classi mediate. Mantiene un riferimento all'oggetto mediatore, e se viene richiamato il suo metodo `on_change()`, richiama il metodo `on_change()` del mediatore, parametrizzato con il widget (cioè con `self`, il widget che ha subito un cambiamento dello stato).

Poiché questo metodo della classe base non è mai modificato in alcuna delle sottoclassi, potremmo sostituire la classe base con un decoratore di classe, come abbiamo visto in precedenza nel Capitolo 2.

```
class Button(Mediated):

    def __init__(self, text=""):
        super().__init__()
        self.enabled = True
        self.text = text

    def click(self):
        if self.enabled:
        self.on_change()
```

Questa classe `Button` eredita `Mediated`. In questo modo il pulsante ha un attributo `self.mediator` e un metodo `on_change()` che deve essere richiamato quando si verifica un cambiamento dello stato, per esempio quando il pulsante viene premuto con un clic.

Perciò, in questo esempio, una chiamata di `Button.click()` provoca una chiamata di `Button.on_change()` (ereditato da `Mediated`), che provoca una chiamata del metodo `on_change()` del mediatore, che a sua volta richiamerà il metodo o i metodi associati al pulsante, in questo caso il metodo `Form.clicked()`, con il pulsante come argomento `widget`.

```
class Text(Mediated):

    def __init__(self, text=""):
        super().__init__()
        self.__text = text

    @property
    def text(self):
        return self.__text

    @text.setter
    def text(self, text):
        if self.text != text:
        self.__text = text
        self.on_change()
```

Strutturalmente, la classe `Text` è identica alla classe `Button` ed eredita anch’essa `Mediated`.

Per qualsiasi widget (pulsante, casella di inserimento di testo e così via), purché sia impostato come sottoclasse di `Mediated` e richiami `on_change()` per ogni cambiamento dello stato, possiamo lasciare a `Mediator` il compito di occuparsi delle interazioni. Naturalmente, quando creiamo il `Mediator`, dobbiamo anche registrare i widget e i metodi associati che vogliamo siano richiamati. Ciò significa che tutti i widget di un form sono accoppiati in modo lasco, evitando relazioni dirette e potenzialmente fragili.

Un mediatore basato su coroutine

Un mediatore può essere considerato come una pipeline che riceve messaggi (chiamate di `on_change()`) e li passa agli oggetti interessati. Come abbiamo già visto precedentemente, le coroutine possono essere utilizzate per fornire queste caratteristiche. Tutto il codice riportato qui è tratto dal programma `mediator2.py`, e tutto il codice non riportato qui è identico a quello riportato nel paragrafo precedente e tratto dal programma `mediator1.py`.

L'approccio adottato qui è diverso da quello scelto nel paragrafo precedente. In precedenza abbiamo associato coppie di widget e metodi, e ogni volta che il widget notificava una modifica, il mediatore richiamava i metodi associati.

In questo caso, a ogni widget è fornito un mediatore che in sostanza è una pipeline di coroutine. Ogni volta che un widget subisce un cambiamento dello stato, invia se stesso nella pipeline, e spetta ai componenti di quest'ultima (cioè le coroutine) decidere se vogliono intraprendere un'azione in risposta a un cambiamento del widget.

```
def create_mediator(self):
    self.mediator = self._update_ui_mediator(self._clicked_mediator())
    for widget in (self.nameText, self.emailText, self.okButton,
                   self.cancelButton):
        widget.mediator = self.mediator
    self.mediator.send(None)
```

Per la versione con coroutine non abbiamo bisogno di una classe mediatore separata, ma creiamo una pipeline di coroutine; in questo caso con due componenti: `self._update_ui_mediator()` e `self._clicked_mediator()` (sono tutti metodi di `Form`).

Una volta ottenuta la pipeline, impostiamo l'attributo `mediator` di ciascun widget alla pipeline in questione, e alla fine inviamo `None` lungo la pipeline. Poiché nessun widget è `None`, in questo modo non scattano azioni specifiche dei widget, ma vengono eseguite azioni a livello del form (come l'abilitazione o la disabilitazione del pulsante *OK* in `_update_ui_mediator()`).

```
@coroutine
def _update_ui_mediator(self, successor=None):
    while True:
        widget = (yield)
        self.okButton.enabled = (bool(self.nameText.text) and
                                 bool(self.emailText.text))
        if successor is not None:
            successor.send(widget)
```

Questa coroutine fa parte della pipeline (il decoratore `@coroutine` è stato illustrato e discusso in precedenza in questo capitolo).

Ogni volta che un widget riporta un cambiamento, viene passato alla pipeline ed è restituito dall'espressione `yield` nella variabile `widget`. Quando si tratta di abilitare o disabilitare il pulsante *OK*, lo facciamo a prescindere da quale widget sia stato modificato (dopo tutto potrebbe anche darsi che non sia cambiato alcun widget, che `widget` sia `None`, e quindi che il form sia semplicemente in fase di inizializzazione). Dopo la gestione del pulsante, la coroutine passa il widget modificato alla coroutine successiva nella catena (se esiste).

```
    @coroutine
    def _clicked_mediator(self, successor=None):
        while True:
            widget = (yield)
            if widget == self.okButton:
                print("OK")
            elif widget == self.cancelButton:
                print("Cancel")
            elif successor is not None:
                successor.send(widget)
```

Questa coroutine della pipeline si occupa soltanto dei clic sui pulsanti *OK* e *Cancel*. Se uno di questi due pulsanti è il widget modificato, la coroutine lo gestisce, altrimenti passa il widget alla coroutine successiva, se esiste.

```
class Mediated:

    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.send(self)
```

Le classi `Button` e `Text` sono identiche a quelle di `mediator1.py`, ma la classe `Mediated` presenta una piccola modifica: se viene richiamato il metodo `on_change()`, invia il widget cambiato (`self`) nella pipeline che fa da mediatore.

Come abbiamo detto nel paragrafo precedente, la classe `Mediated` potrebbe essere sostituita con un decoratore di classe. Tra gli esempi di questo libro è inclusa una versione di questo esempio denominata `mediator2d.py` in cui viene fatto proprio questo (cfr. il Capitolo 2).

Il pattern Mediator può anche essere variato in modo da fornire funzionalità di multiplexing, cioè comunicazioni molti a molti tra oggetti. Per informazioni al riguardo, consultate i paragrafi di questo capitolo dedicati al pattern Observer e al pattern State.

Il pattern Memento

Questo pattern offre un mezzo per salvare e ripristinare lo stato di un oggetto senza violare l'incapsulamento.

Python offre direttamente il supporto per questo pattern: possiamo usare il modulo `pickle` per serializzare e deserializzare oggetti Python arbitrari (con alcuni vincoli, per esempio non possiamo serializzare un oggetto file). In effetti Python può serializzare `None`, `bool`, `bytearray`, `bytes`, `complex`, `float`, `int` e `str`, oltre a `dict`, `list` e `tuple` che contengono soltanto oggetti serializzabili (incluse le collezioni), funzioni di primo livello, classi di primo livello e istanze di classi di primo livello personalizzate il cui `__dict__` sia serializzabile. In pratica, oggetti della maggior parte delle classi personalizzate. È anche possibile ottenere lo stesso effetto utilizzando il modulo `json`, ma questo supporta soltanto i tipi di base di Python, oltre a dizionari e liste (abbiamo visto esempi di utilizzo di `json` e `pickle` precedentemente in questo capitolo, nel paragrafo dedicato alla valutazione del codice mediante un sottoprocesso).

Anche nei rari casi in cui incontriamo una limitazione a ciò che può essere serializzato, possiamo sempre aggiungere un supporto personalizzato per la serializzazione, per esempio implementando i metodi speciali `__getstate__()` e `__setstate__()` ed eventualmente il metodo `__getnewargs__()`. Similmente, se vogliamo utilizzare il formato JSON con le nostre classi personalizzate, possiamo estendere il codificatore e il decodificatore del modulo `json`.

Potremmo anche creare un nostro formato e nostri protocolli, ma non ha molto senso farlo, dato il ricco supporto offerto da Python per questo pattern.

La deserializzazione o unpickling in sostanza comporta l'esecuzione di codice Python arbitrario, perciò non è consigliato deserializzare pickle ricevuti da fonti non sicure, come supporti fisici o connessioni di rete. In tali casi l'uso di JSON è più sicuro, oppure possiamo usare checksum e cifratura con la serializzazione, in modo da assicurarci che il pickle non sia stato alterato.

Il pattern Observer

Questo pattern supporta relazioni di dipendenza molti a molti tra oggetti, tali che, quando un oggetto cambia di stato, tutti gli oggetti in relazione con esso ricevono una notifica. Probabilmente l'espressione più comune di questo pattern e delle sue varianti è il paradigma MVC (*Model/View/Controller*), in cui un modello rappresenta i dati, una o più viste visualizzano tali dati, e uno o più controller mediano tra l'input (per esempio l'interazione con l'utente) e il modello. E qualsiasi modifica al modello si riflette automaticamente sulle viste associate.

Una comune semplificazione dell'approccio MVC è quella di usare un modello/vista in cui le viste visualizzano i dati e fanno anche da mediatori per l'input; in sostanza, viste e controller sono riuniti insieme. Dal punto di vista del pattern Observer, questo significa che le viste sono osservatori del modello, e che il modello è il soggetto osservato.

In questo paragrafo creeremo un modello che rappresenta un valore con un minimo e un massimo (come una barra di scorrimento, o un cursore a scorrimento, o un controllo di temperatura), e due osservatori separati (viste) per il modello: una per l'output del valore del modello quando cambia (una sorta di barra di avanzamento HTML) e un'altra per mantenere una cronologia delle modifiche (con valori e timestamp). Ecco un'esecuzione di esempio del programma `observer.py`:

```
$ ./observer.py > /tmp/observer.html
0 2013-04-09 14:12:01.043437
7 2013-04-09 14:12:01.043527
23 2013-04-09 14:12:01.043587
37 2013-04-09 14:12:01.043647
```

I dati di cronologia sono inviati a `sys.stderr` e il codice HTML a `sys.stdout`, che abbiamo reindirizzato in un file HTML. La pagina HTML è mostrata nella Figura 3.5. Il programma invia in output quattro tabelle HTML di una sola riga, la prima quando il modello (vuoto) è osservato la prima volta, e poi ogni volta che il modello viene modificato. La Figura 3.6 illustra l'architettura modello/vista del nostro esempio.

L'esempio di questo paragrafo, `observer.py`, utilizza una classe base `Observed` per fornire la funzionalità per aggiungere, rimuovere e notificare osservatori. La classe `SliderModel` fornisce un valore con un minimo e un massimo, ed eredita la classe `Observed` in modo che possa supportare l'osservazione. E poi abbiamo due viste che osservano il modello: `HistoryView` e `LiveView`. Naturalmente esamineremo tutte queste classi, ma per prima cosa esaminiamo la funzione `main()` del programma per vedere

come sono utilizzate e come è stato ottenuto l'output mostrato in precedenza e nella Figura 3.5.

```
def main():
    historyView = HistoryView()
    liveView = LiveView()
    model = SliderModel(0, 0, 40)    # minimo, valore, massimo
    model.observers_add(historyView, liveView)    # liveView produce output
    for value in (7, 23, 37):
    model.value = value    # liveView produce output
    for value, timestamp in historyView.data:
        print("{:3} {}".format(value, datetime.datetime.fromtimestamp(
                timestamp)), file=sys.stderr)
```

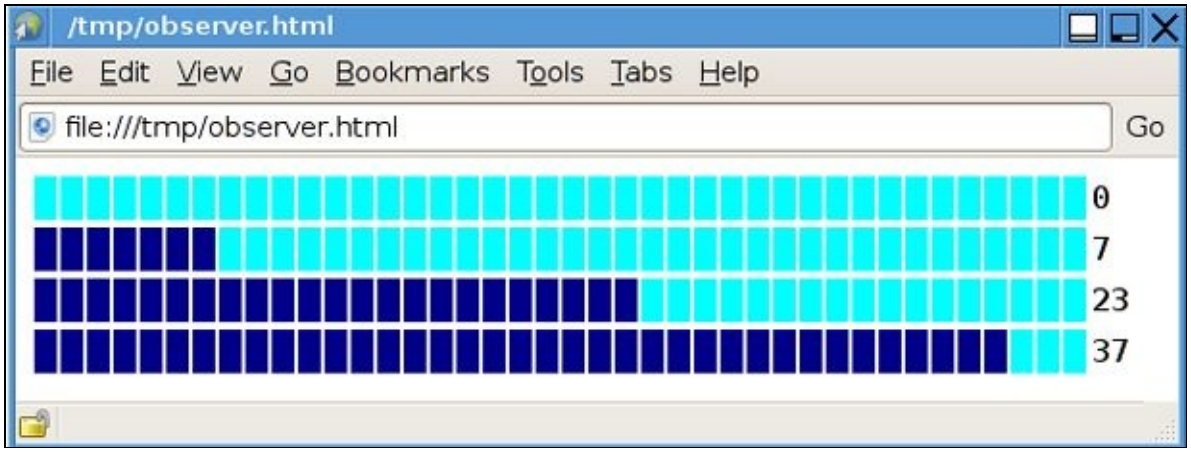


Figura 3.5 L'output HTML dell'esempio quando il modello cambia.

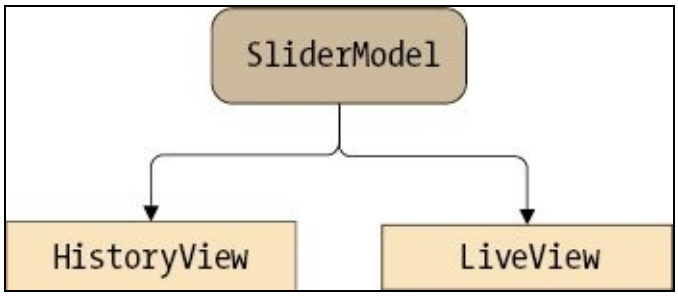


Figura 3.6 Un modello e due viste.

Iniziamo creando le due viste, poi creiamo un modello con valore minimo 0, valore corrente 0 e valore massimo 40. Poi aggiungiamo le due viste come osservatori del modello. Non appena viene aggiunto il `LiveView` come osservatore, produce il suo primo output, e non appena viene aggiunto `HistoryView`, registra il primo valore accompagnato dal timestamp. Poi aggiorniamo il valore del modello tre volte, e a ciascun aggiornamento `LiveView` esegue l'output di una nuova tabella HTML di una sola riga e `HistoryView` registra il valore e il timestamp.

Al termine stampiamo l'intera cronologia su `sys.stderr` (cioè la console). La funzione `datetime.datetime.fromtimestamp()` accetta un *timestamp* (numero di secondi trascorsi dal valore di epoch restituito da `time.time()`) e restituisce un oggetto

`datetime.datetime` equivalente. Il metodo `str.format()` è in grado di eseguire l'output di `datetime.datetime`s in formato ISO-8601.

```
class Observed:

    def __init__(self):
        self.__observers = set()

    def observers_add(self, observer, *observers):
        for observer in itertools.chain((observer,), observers):
            self.__observers.add(observer)
            observer.update(self)

    def observer_discard(self, observer):
        self.__observers.discard(observer)

    def observers_notify(self):
        for observer in self.__observers:
            observer.update(self)
```

Questa classe è progettata per essere ereditata da modelli o da qualunque altra classe che voglia supportare l'osservazione. La classe `Observed` mantiene un insieme di oggetti di osservazione. Ogni volta si aggiunge un oggetto, viene richiamato il suo metodo `update()` per inizializzarlo con lo stato attuale del modello. Poi, ogni volta che il modello cambia di stato dovrebbe richiamare il suo metodo ereditato `observers_notify()`, in modo che possa essere richiamato il metodo `update()` di ogni osservatore per assicurarsi che ciascun osservatore (cioè ogni vista) rappresenti il nuovo stato del modello.

Il metodo `observers_add()` merita particolare attenzione. Vogliamo accettare uno o più osservatori da aggiungere, ma utilizzando semplicemente `*observers` sarebbe consentito inserirne zero o più. Perciò richiediamo almeno un osservatore (`observer`) e ne accettiamo zero o più in aggiunta (`*observers`). Avremmo potuto ottenere lo stesso effetto utilizzando la concatenazione di tuple (per esempio con `for observer in (observer,) + observers:`), ma abbiamo preferito utilizzare la più efficiente funzione `itertools.chain()`. Come si è osservato precedentemente nel Capitolo 2, questa funzione accetta qualsiasi numero di iterabili e restituisce un singolo iterabile che è in effetti la concatenazione di tutti gli iterabili passati.

```
class SliderModel(Observed):

    def __init__(self, minimum, value, maximum):
        super().__init__()
        # Devono esistere prima dell'uso dei metodi di impostazione delle proprietà
        self.__minimum = self.__value = self.__maximum = None
        self.minimum = minimum
        self.value = value
        self.maximum = maximum

    @property
    def value(self):
        return self.__value

    @value.setter
```

```
    def value(self, value):
        if self.__value != value:
            self.__value = value
            self.observers_notify()
    ...
```

Questa è la particolare classe modello per questo esempio, ma naturalmente potrebbe essere qualsiasi tipo di modello. Ereditando `Observed`, la classe acquisisce un insieme privato di osservatori (inizialmente vuoto) e i metodi `observers_add()`, `observer_discard()` e `observers_notify()`. Ogni volta che lo stato del modello cambia, per esempio quando il suo valore viene modificato, deve richiamare il suo metodo `observers_notify()` affinché ogni osservatore possa reagire di conseguenza.

La classe ha anche proprietà `minimum` e `maximum` di cui non abbiamo riportato il codice; sono strutturalmente identiche alla proprietà `value` e, naturalmente, anche i loro metodi di impostazione richiamano `observers_notify()`.

```
class HistoryView:

    def __init__(self):
        self.data = []

    def update(self, model):
        self.data.append((model.value, time.time()))
```

Questa vista è un osservatore del modello, poiché fornisce un metodo `update()` che accetta il modello osservato come suo unico argomento oltre a `self`. Ogni volta che viene richiamato il metodo `update()`, aggiunge una coppia valore-timestamp alla propria lista `self.data`, mantenendo così una cronologia di tutte le modifiche applicate al modello.

```
class LiveView:

    def __init__(self, length=40):
        self.length = length
```

Questa è un’altra vista che osserva il modello. La lunghezza `lenght` è il numero di celle usate per rappresentare il valore del modello in una tabella HTML di una sola riga.

```
def update(self, model):
    tippingPoint = round(model.value * self.length /
            (model.maximum - model.minimum))
    td = '<td style="background-color: {}"> </td>'
    html = ['<table style="font-family: monospace" border="0"><tr>']
    html.extend(td.format("darkblue") * tippingPoint)
    html.extend(td.format("cyan") * (self.length - tippingPoint))
    html.append("<td>{}</td></tr></table>".format(model.value))
    print("".join(html))
```

Quando il modello è osservato per la prima volta, e ogni volta che viene successivamente aggiornato, viene richiamato questo metodo, che invia in output una tabella HTML di una sola riga con `self.length` celle per rappresentare il modello,

usando il colore azzurro cyan per le celle vuote e blu scuro per quelle riempite. Determina quante celle ci sono e di quale tipo calcolando il tipping point tra le celle piene (se ve ne sono) e quelle vuote.

Il pattern Observer è ampiamente utilizzato nella programmazione di GUI e anche nel contesto di altre architetture a elaborazione di eventi, come simulazioni e server. Tra gli esempi citiamo i trigger di database, il sistema di signaling di Django, il meccanismo di segnali e slot del framework GUI Qt, e molte applicazioni di WebSockets.

Il pattern State

Questo pattern è progettato per fornire oggetti il cui comportamento cambia quando cambia il loro stato.

Per illustrare questo design pattern creeremo una classe multiplexer che ha due stati, e in cui il comportamento dei suoi metodi cambia in base allo stato in cui si trova un'istanza del multiplexer. Quando il multiplexer è nel suo stato attivo, può accettare "connessioni" - cioè coppie nome evento-callback - dove il callback è un qualsiasi callable Python (per esempio un `lambda`, una funzione, un metodo bound e così via). Dopo che le connessioni sono state effettuate, ogni volta che un evento è inviato al multiplexer, vengono richiamati i callback associati (purché il multiplexer sia nel suo stato attivo). Se il multiplexer è dormiente, la chiamata dei suoi metodi non ha alcun effetto.

Per mostrare il multiplexer in uso creeremo alcune funzioni di callback che contano il numero di eventi che ricevono e li connettono a un multiplexer attivo. Poi invieremo alcuni eventi casuali al multiplexer, e quindi stamperemo i conteggi accumulati dai callback. Tutto il codice è tratto dal programma `multiplexer1.py`, e l'output per un'esecuzione di esempio è mostrato di seguito:

```
$ ./multiplexer1.py
After 100 active events: cars=150 vans=42 trucks=14 total=206
After 100 dormant events: cars=150 vans=42 trucks=14 total=206
After 100 active events: cars=303 vans=83 trucks=30 total=416
```

Dopo aver inviato al multiplexer attivo un centinaio di eventi a caso, cambiamo lo stato del multiplexer in dormiente, e poi inviamo un altro centinaio di eventi, che dovrebbero essere tutti ignorati. Poi riportiamo il multiplexer nello stato attivo e inviamo altri eventi; in questo caso dovrebbero essere richiamati i callback associati.

Iniziamo esaminando il modo in cui è realizzato il multiplexer, come sono effettuate le connessioni e come sono inviati gli eventi. Poi esamineremo le funzioni di callback e la classe di eventi. Infine esamineremo il multiplexer stesso.

```
totalCounter = Counter()
carCounter = Counter("cars")
commercialCounter = Counter("vans", "trucks")

multiplexer = Multiplexer()
for eventName, callback in (("cars", carCounter),
        ("vans", commercialCounter), ("trucks", commercialCounter)):
    multiplexer.connect(eventName, callback)
    multiplexer.connect(eventName, totalCounter)
```

Qui iniziamo creando alcuni contatori. Le istanze di queste classi sono callable, perciò possono essere usate ovunque sia richiesta una funzione (per esempio un

callback). Mantengono un conteggio indipendente per nome fornito, o se anonime (come `totalCounter`) mantengono un singolo conteggio.

Poi creiamo un nuovo multiplexer (che all'inizio è attivo per default). Poi connettiamo le funzioni callback agli eventi. I nomi di eventi a cui siamo interessati sono tre: "cars", "vans" e "trucks". La funzione `carCounter()` è connessa all'evento "cars", la funzione `commercialCounter()` è connessa agli eventi "vans" e "trucks"; la funzione `totalCounter()` è connessa a tutti e tre gli eventi.

```
for event in generate_random_events(100):
    multiplexer.send(event)
print("After 100 active events: cars={} vans={} trucks={} total={}"
        .format(carCounter.cars, commercialCounter.vans,
                commercialCounter.trucks, totalCounter.count))
```

In questa porzione di codice generiamo un centinaio di eventi casuali e li inviamo uno per uno al multiplexer. Se, per esempio, un evento è "cars", il multiplexer richiamerà le funzioni `carCounter()` e `totalCounter()`, passando l'evento come unico argomento per ciascuna chiamata. Similmente, se l'evento è "vans" o "trucks", vengono richiamate le funzioni `commercialCounter()` e `totalCounter()`.

```
class Counter:

    def __init__(self, *names):
        self.anonymous = not bool(names)
        if self.anonymous:
            self.count = 0
        else:
            for name in names:
                if not name.isidentifier():
                    raise ValueError("names must be valid identifiers")
                setattr(self, name, 0)
```

Se non sono forniti nomi, viene creata un'istanza di un contatore anonimo il cui conteggio è mantenuto in `self.count`. Altrimenti, sono creati contatori indipendenti per il nome o i nomi passati utilizzando la funzione integrata `setattr()`. Per esempio, all'istanza `carCounter` viene assegnato un attributo `self.cars` e all'istanza `commercialCounter` vengono assegnati gli attributi `self.vans` e `self.trucks`.

```
    def __call__(self, event):
        if self.anonymous:
            self.count += event.count
        else:
            count = getattr(self, event.name)
            setattr(self, event.name, count + event.count)
```

Quando viene richiamata un'istanza di `Counter`, la chiamata è passata a questo metodo speciale. Se il contatore è anonimo (per esempio `totalCounter`), `self.count` viene incrementato, altrimenti cerchiamo di recuperare l'attributo contatore corrispondente al nome di evento. Per esempio, se il nome dell'evento è `"trucks"`, impostiamo `count` al

valore di `self.trucks`. Poi aggiorniamo il valore dell'attributo con la somma del vecchio conteggio più quello del nuovo evento.

Poiché non abbiamo fornito un valore di default per la funzione integrata `getattr()`, se l'attributo non esiste (per esempio `"truck"`), il metodo solleverà un `AttributeError`. Ciò garantisce anche che non creeremo un attributo dal nome sbagliato per errore, perché in quei casi la chiamata di `setattr()` non viene mai raggiunta.

```
class Event:

    def __init__(self, name, count=1):
        if not name.isidentifier():
            raise ValueError("names must be valid identifiers")
        self.name = name
        self.count = count
```

Questa è l'intera classe `Event`. È molto semplice, perché ci serve semplicemente come parte dell'infrastruttura per illustrare il pattern State che è esemplificata dalla classe `Multiplexer`. Tra l'altro, il `Multiplexer` offre anche un esempio per il pattern Observer (trattato in precedenza in questo capitolo).

Uso di metodi sensibili allo stato

Per gestire lo stato all'interno di una classe possiamo scegliere tra due approcci principali. Uno è quello di usare metodi sensibili allo stato, come vedremo qui, e l'altro è quello di usare metodi specifici dello stato, come vedremo più avanti in questo stesso capitolo.

```
class Multiplexer:

    ACTIVE, DORMANT = ("ACTIVE", "DORMANT")

    def __init__(self):
        self.callbacksForEvent = collections.defaultdict(list)
        self.state = Multiplexer.ACTIVE
```

La classe `Multiplexer` ha due stati (o modi): `ACTIVE` e `DORMANT`. Quando un'istanza di `Multiplexer` è nello stato `ACTIVE`, i suoi metodi sensibili allo stato lavorano utilmente, ma quando è nello stato `DORMANT`, tali metodi sono inattivi. Ci assicuriamo che, quando viene creato un nuovo `Multiplexer`, si avvii nello stato `ACTIVE`.

Le chiavi di dizionario `self.callbacksForEvent` sono nomi di eventi, e i valori sono liste di callable.

```
    def connect(self, eventName, callback):
        if self.state == Multiplexer.ACTIVE:
            self.callbacksForEvent[eventName].append(callback)
```

Questo metodo è utilizzato per creare un'associazione tra un evento con nome e un callback. Se il dato nome di evento non è già nel dizionario, il fatto che `self.callbacksForEvent` sia un dizionario di default garantisce che sarà creato un elemento con il nome di evento come chiave e una lista vuota come valore, che verrà poi restituita. E se il nome di evento è già nel dizionario, ne sarà restituita la lista. Perciò, in entrambi i casi otteniamo una lista a cui possiamo poi aggiungere il nuovo callback (abbiamo trattato precedentemente i dizionari di default, nel paragrafo di questo capitolo dedicato al pattern Mediator).

```
def disconnect(self, eventName, callback=None):
    if self.state == Multiplexer.ACTIVE:
        if callback is None:
            del self.callbacksForEvent[eventName]
        else:
            self.callbacksForEvent[eventName].remove(callback)
```

Se questo metodo è richiamato senza specificare un callback, interpretiamo il fatto come la volontà dell'utente di disconnettere tutti i callback associati al dato nome di evento. Altrimenti, rimuoviamo soltanto il callback specificato dalla lista di callback del nome di evento.

```
def send(self, event):
    if self.state == Multiplexer.ACTIVE:
        for callback in self.callbacksForEvent.get(event.name, ()):
            callback(event)
```

Se un evento è inviato al multiplexer, e se il multiplexer è attivo, questo metodo itera su tutti i callback associati all'evento dato (potrebbe non essercene alcuno) e, per ciascuno, lo richiama con l'evento come argomento.

Uso di metodi specifici dello stato

Il programma `multiplexer2.py` è quasi identico a `multiplexer1.py`, l'unica differenza è che la classe `Multiplexer` utilizza metodi specifici dello stato anziché metodi sensibili allo stato come nel paragrafo precedente. La classe `Multiplexer` ha gli stessi due stati e lo stesso metodo `__init__()` dell'esempio precedente, ma ora l'attributo `self.state` è una proprietà.

```
@property
def state(self):
    return (Multiplexer.ACTIVE if self.send == self.__active_send
            else Multiplexer.DORMANT)
```

Questa versione del multiplexer non memorizza lo stato come tale, ma lo calcola verificando se uno dei metodi pubblici è stato impostato a un metodo privato attivo o passivo, come vedremo nel seguito.

```
    @state.setter
    def state(self, state):
        if state == Multiplexer.ACTIVE:
            self.connect = self.__active_connect
            self.disconnect = self.__active_disconnect
            self.send = self.__active_send
        else:
            self.connect = lambda *args: None
            self.disconnect = lambda *args: None self.
            send = lambda *args: None
```

Ogni volta che lo stato cambia, il metodo di impostazione della proprietà `state` imposta il multiplexer in modo che abbia un insieme di metodi appropriati per lo stato in questione. Per esempio, se lo stato è `DORMANT`, le versioni `lambda` anonime dei metodi sono assegnate ai metodi pubblici.

```
def __active_connect(self, eventName, callback):
self.callbacksForEvent[eventName].append(callback)
```

Qui abbiamo creato un metodo attivo privato: questo o un metodo `lambda` anonimo viene assegnato al metodo pubblico corrispondente. Non abbiamo mostrato i metodi attivi privati per la disconnessione o l'invio, perché seguono lo stesso schema. Il punto fondamentale da tenere presente è che nessuno di questi metodi controlla lo stato dell'istanza (poiché vengono chiamati soltanto nello stato appropriato), e questo li semplifica e li rende più veloci.

Naturalmente è facile mettere a punto una versione del `Multiplexer` basata su coroutine, ma poiché abbiamo già visto alcuni esempi di coroutine, rinunciamo a mostrarne un altro qui (tuttavia, il programma `multiplexer3.py` incluso tra gli esempi del libro mostra un approccio al multiplexing basato su coroutine).

Anche se abbiamo usato il pattern State per un multiplexer, avere oggetti con informazioni di stato (o modali) è abbastanza comune in molti contesti.

Il pattern Strategy

Questo pattern fornisce un mezzo per incapsulare un insieme di algoritmi che possono essere usati in modo intercambiabile, in base alle esigenze dell'utente.

Per esempio, in questo paragrafo creeremo due diversi algoritmi per disporre una lista contenente un numero arbitrario di elementi all'interno di una tabella con un numero di righe specificato. Un algoritmo produrrà in output del codice HTML; la Figura 3.7 mostra i risultati per tabelle con due, tre e quattro righe. L'altro algoritmo produrrà testo semplice, e i risultati (per tabelle di quattro e cinque righe) sono mostrati qui:

```
$ ./tabulator3.py
...
+------------------+------------------+------------------+
| Nikolai Andriano | Matt Biondi      | Bjørn Dæhlie     |
| Birgit Fischer   | Sawao Kato       | Larisa Latynina  |
| Carl Lewis       | Michael Phelps   | Mark Spitz       |
| Jenny Thompson   |                  |                  |
+------------------+------------------+------------------+

+------------------+------------------+
| Nikolai Andrianov | Matt Biondi      |
| Bjørn Dæhlie      | Birgit Fischer   |
| Sawao Kato        | Larisa Latynina  |
| Carl Lewis        | Michael Phelps   |
| Mark Spitz        | Jenny Thompson   |
+------------------ +------------------+
```

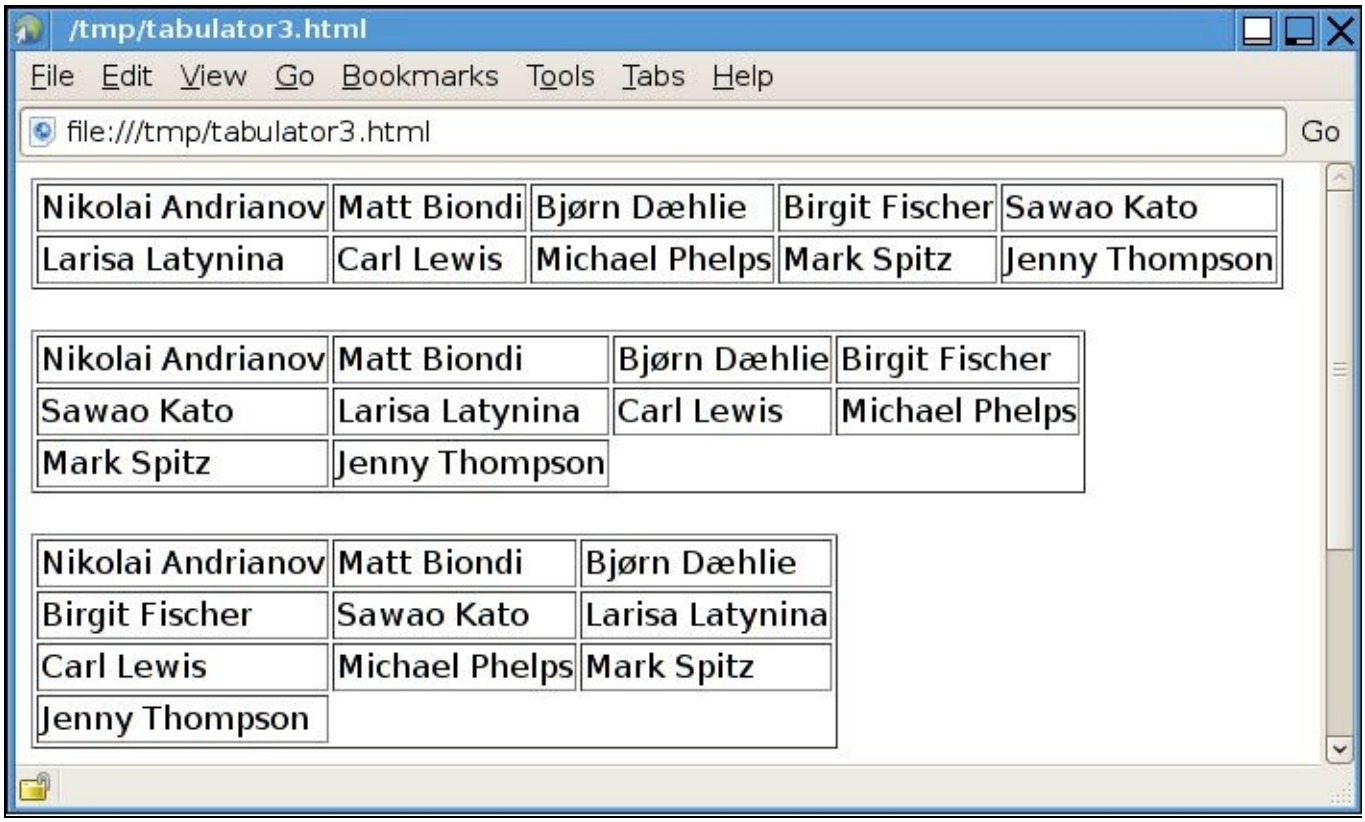


Figura 3.7 La tabella HTML risultato del programma tabulatore.

Esistono vari approcci che potremmo adottare per la parametrizzazione mediante algoritmo. Uno ovvio è quello di creare una classe `Layout` che accetti un'istanza `Tabulator`, la quale realizzi la disposizione tabellare appropriata. Il programma `tabulator1.py` (non riportato qui) utilizza questo approccio. Un raffinamento, per tabulatori che non hanno la necessità di mantenere lo stato, è quello di utilizzare metodi statici e passare la classe tabulatore anziché un'istanza per fornire l'algoritmo; tale approccio è utilizzato nel programma `tabulator2.py` (anch'esso non riportato qui).

Nel seguito mostreremo una tecnica più semplice e ancora più raffinata: una classe `Layout` che accetta una funzione di tabulazione che implementa l'algoritmo desiderato.

```
WINNERS = ("Nikolai Andrianov", "Matt Biondi", "Bjørn Dæhlie",
        "Birgit Fischer", "Sawao Kato", "Larisa Latynina", "Carl Lewis",
        "Michael Phelps", "Mark Spitz", "Jenny Thompson")

def main():
    htmlLayout = Layout(html_tabulator)
    for rows in range(2, 6):
        print(htmlLayout.tabulate(rows, WINNERS))
    textLayout = Layout(text_tabulator)
    for rows in range(2, 6):
        print(textLayout.tabulate(rows, WINNERS))
```

In questa funzione creiamo due oggetti `Layout`, ognuno parametrizzato da una diversa funzione di tabulazione. Per ciascun layout stampiamo una tabella con due righe, con tre righe, con quattro righe e con cinque righe.

```
class Layout:

    def __init__(self, tabulator):
        self.tabulator = tabulator

    def tabulate(self, rows, items):
        return self.tabulator(rows, items)
```

Questa classe supporta un solo algoritmo: quello di tabulazione. La funzione che implementa tale algoritmo dovrebbe accettare un numero di righe e una sequenza di elementi e restituire i risultati in forma tabulata.

In effetti potremmo ridurre ulteriormente le dimensioni di questa classe; ecco la versione di `tabulator4.py`.

```
class Layout:

    def __init__(self, tabulator):
        self.tabulate = tabulator
```

In questo caso l'attributo `self.tabulate` è un callable (la funzione di tabulazione passata). Le chiamate mostrate in `main()` funzionano esattamente come per le classi `Layout` di `tabulator3.py` e `tabulator4.py`.

Benché gli effettivi algoritmi di disposizione in tabella non siano importanti per il design pattern di per sé, ne esaminiamo brevemente uno a titolo di completezza.

```
def html_tabulator(rows, items):
    columns, remainder = divmod(len(items), rows)
    if remainder:
        columns += 1
    column = 0
    table = ['<table border="1">\n']
    for item in items:
        if column == 0:
            table.append("<tr>")
        table.append("<td>{}</td>".format(escape(str(item))))
        column += 1
        if column == columns:
            table.append("</tr>\n")
        column %= columns
    if table[-1][-1] != "\n":
    table.append("</tr>\n")
    table.append("</table>\n")
    return "".join(table)
```

Per entrambe le funzioni di tabulazione dobbiamo calcolare il numero di colonne necessario per inserire tutti gli elementi in una tabella con il numero di righe specificato. Una volta noto questo numero (`columns`), possiamo iterare su tutti gli elementi tenendo traccia della colonna corrente nella riga corrente.

La funzione `text_tabulator()` (non mostrata qui) è leggermente più lunga ma utilizza in sostanza lo stesso approccio.

In contesti più realistici potremmo utilizzare algoritmi radicalmente diversi, sia in termini di codice che di caratteristiche prestazionali, in modo da consentire agli utenti di scegliere il compromesso più appropriato per le loro esigenze particolari. Inserire diversi algoritmi come callable - `lambda`, funzioni, metodi bound - è semplice, perché Python considera i callable come oggetti di prima classe, cioè come oggetti che possono essere passati e memorizzati in collezioni come qualsiasi altro tipo di oggetto.

Il pattern Template Method

Il pattern Template Method ci consente di definire i passi di un algoritmo, ma affidare l'esecuzione di alcuni di questi passi a opportune sottoclassi.

In questo paragrafo creeremo una classe `AbstractWordCounter` che fornisce due metodi. Il primo, `can_count(filename)`, deve restituire un boolean che indica se la classe può contare le parole presenti nel file dato (in base all'estensione del file). Il secondo metodo, `count(filename)`, deve restituire un conteggio di parole. Creeremo anche due sottoclassi: una per contare le parole di file di testo normale e l'altra per contare le parole di file HTML. Iniziamo osservando le classi in azione (con il codice tratto da `wordcount1.py`):

```
def count_words(filename):
    for wordCounter in (PlainTextWordCounter, HtmlWordCounter):
        if wordCounter.can_count(filename):
            return wordCounter.count(filename)
```

Tutti i metodi di tutte le classi sono statici. Questo significa che non si possono mantenere informazioni di stato per istanza (perché non ci sono istanze come tali) e che possiamo lavorare direttamente su oggetti di classe anziché su istanze (sarebbe facile rendere i metodi non statici e usare le istanze, se avessimo la necessità di mantenere dati di stato).

Iteriamo su due oggetti di classe per il conteggio di parole, e se uno di essi è in grado di contare le parole presenti nel file dato, eseguiamo il conteggio e restituiamo il valore. Se nessuno di essi è in grado di farlo, restituiamo (implicitamente) `None` a indicare che non è stato possibile eseguire alcun conteggio.

```
class AbstractWordCounter:
@staticmethod
def can_count(filename):
raise NotImplementedError()
@staticmethod
def count(filename):
raise NotImplementedError()
```

```
class AbstractWordCounter(
metaclass=abc.ABCMeta):
@staticmethod
@abc.abstractmethod
def can_count(filename):
pass
@staticmethod
@abc.abstractmethod
def count(filename):
pass
```

Questa classe astratta fornisce l'interfaccia per il conteggio delle parole, i cui metodi devono essere reimplementati dalle sottoclassi. La porzione di codice a sinistra, tratta da `wordcount1.py`, adotta un approccio più tradizionale, mentre quella a destra, tratta da `wordcount2.py`, utilizza un approccio più moderno che sfrutta il modulo `abc` (*abstract base class*).

```
class PlainTextWordCounter(AbstractWordCounter):

    @staticmethod
```

```
    def can_count(filename):
        return filename.lower().endswith(".txt")

    @staticmethod
    def count(filename):
        if not PlainTextWordCounter.can_count(filename):
            return 0
        regex = re.compile(r"\w+")
        total = 0
        with open(filename, encoding="utf-8") as file:
            for line in file:
                for _ in regex.finditer(line):
                    total += 1
        return total
```

Questa sottoclasse implementa l'interfaccia per il contatore di parole usando una nozione molto semplicistica di ciò che costituisce una parola, e assumendo che tutti i file `.txt` siano codificati in UTF-8 (o ASCII a 7 bit, poiché è un sottoinsieme di UTF-8).

```
class HtmlWordCounter(AbstractWordCounter):

    @staticmethod
    def can_count(filename):
        return filename.lower().endswith((".htm", ".html"))

    @staticmethod
    def count(filename):
        if not HtmlWordCounter.can_count(filename):
            return 0
        parser = HtmlWordCounter.__HtmlParser()
        with open(filename, encoding="utf-8") as file:
            parser.feed(file.read())
        return parser.count
```

Questa sottoclasse fornisce l'interfaccia del contatore di parole per file HTML. Utilizza il suo parser HTML privato (anch'esso una sottoclasse `html.parser.HTMLParser` incorporata all'interno della classe `HtmlWordCounter`, che vedremo tra breve). Avendo a disposizione tale parser HTML, tutto ciò che dobbiamo fare per contare le parole di un file HTML è creare un'istanza del parser e fornirle codice HTML da analizzare. Una volta completato il parsing, restituiamo il conteggio di parole risultante.

Per completezza esamineremo il parser incorporato `HtmlWordCounter.__HtmlParser` che svolge il conteggio effettivo. Il parser HTML della libreria standard di Python funziona come un parser SAX (*Simple API for XML*), nel senso che itera sul testo e richiama metodi particolari quando si verificano gli eventi corrispondenti (per esempio "start tag", "end tag" e così via). Perciò, per poter usare il parser dobbiamo crearne una sottoclasse e reimplementare i metodi che corrispondono agli eventi che ci interessano.

```
class __HtmlParser(html.parser.HTMLParser):

    def __init__(self):
        super().__init__()
        self.regex = re.compile(r"\w+")
        self.inText = True
        self.text = []
        self.count = 0
```

Abbiamo reso privata la sottoclasse `html.parser.HTMLParser` e vi abbiamo aggiunto quattro elementi. `self.regex` mantiene la nostra semplice definizione di “parola” (una sequenza di una o più lettere, cifre o trattini di sottolineatura). `self.inText bool` indica se il testo che incontriamo è visibile all’utente (anziché essere nascosto all’interno di un tag `<script>` o `<style>`). `self.text` contiene la porzione o le porzioni di testo che costituiscono il valore corrente, e `self.count` è il conteggio di parole.

```
def handle_starttag(self, tag, attrs):
    if tag in {"script", "style"}:
        self.inText = False
```

I nomi e la firma di questo metodo (e quelli di tutti i metodi `handle_…()`) sono determinati dalla classe base. Per default, i metodi handler non fanno nulla, perciò dobbiamo naturalmente reimplementare ogni metodo che ci interessi.

Non vogliamo contare le parole all’interno di script incorporati o fogli stile, perciò se incontriamo i corrispondenti tag di apertura disattiviamo l’accumulazione di testo.

```
def handle_endtag(self, tag):
    if tag in {"script", "style"}:
        self.inText = True else:
    for _ in self.regex.finditer(" ".join(self.text)):
        self.count += 1
    self.text = []
```

Se raggiungiamo il tag di chiusura di uno script o di un foglio stile, riattiviamo l’accumulazione di testo. In tutti gli altri casi iteriamo sul testo accumulato e ne contiamo le parole, poi reimpostiamo il testo accumulato a una lista vuota.

```
def handle_data(self, text):
    if self.inText:
        text = text.rstrip()
        if text:
            self.text.append(text)
```

Se riceviamo del testo e non siamo all’interno di uno script o di un foglio stile, lo accumuliamo.

Grazie alla potenza e alla flessibilità di Python nel supporto per classi annidate private, e al suo `html.parser.HTMLParser`, possiamo svolgere un parsing abbastanza sofisticato nascondendo tutti i dettagli agli utenti della classe `HtmlWordCounter`.

Il pattern Template Method è per certi versi simile al pattern Bridge che abbiamo visto in precedenza nel Capitolo 2.

Il pattern Visitor

Questo pattern è utilizzato per applicare una funzione a ciascun elemento di una collezione o di un oggetto aggregato. Si tratta di un impiego diverso da quello tipico del pattern Iterator descritto in precedenza in questo capitolo, in cui iteriamo su una collezione o un oggetto aggregato e richiamiamo un metodo su ciascun elemento, poiché con un "visitatore" anziché richiamare un metodo applichiamo una funzione esterna.

Python supporta direttamente questo pattern. Per esempio, `newList = map(function, oldSequence)` richiamerà la `function()` su ogni elemento nella `oldSequence` per produrre `newList`. Lo stesso si può fare utilizzando una list comprehension: `newList = [function(item) for item in oldSequence]`.

Se abbiamo la necessità di applicare una funzione a ogni elemento di una collezione o di un oggetto aggregato, possiamo iterare utilizzando un ciclo `for`: `for item in collection: function(item)`. Se gli elementi sono di tipi diversi, possiamo utilizzare istruzioni `if` e la funzione integrata `isinstance()` per distinguere tra di essi al fine di scegliere il codice appropriato da eseguire all'interno della `function()`.

Alcuni dei pattern comportamentali sono supportati direttamente nel linguaggio Python, mentre gli altri non sono semplici da implementare.

I pattern Chain of Responsibility, Mediator e Observer possono essere implementati convenzionalmente o usando coroutine, e forniscono tutti delle varianti sul tema del disaccoppiamento della comunicazione tra oggetti. Il pattern Command può essere usato per fornire lazy evaluation e funzioni di annullamento dei comandi. Poiché Python è un linguaggio interpretato (byte-code), possiamo implementare il pattern Interpreter utilizzando Python stesso e perfino isolare il codice interpretato in un processo separato. Il supporto per il pattern Iterator (e implicitamente per il pattern Visitor) è integrato in Python. Il pattern Memento è ben supportato dalla libreria standard di Python (per esempio utilizzando i moduli `pickle` o `json`). I pattern State, Strategy e Template Method non hanno un supporto diretto in Python, ma sono facili da implementare.

I design pattern forniscono modi utili per pensare, organizzare e implementare codice. Alcuni sono applicabili solo al paradigma della programmazione orientata agli oggetti, mentre altri possono essere utilizzati anche per la programmazione procedurale.

A partire dalla pubblicazione del libro originale sui design pattern, il tema è stato al centro (e continua a esserlo) di un notevole interesse di ricerca. Il miglior punto di partenza per trovare ulteriori informazioni al riguardo è il sito web dell'associazione non profit Hillside Group (`http://hillside.net`).

Nel prossimo capitolo esamineremo un paradigma di programmazione diverso - la concorrenza - per cercare di arrivare a un miglioramento delle prestazioni sfruttando l'hardware moderno multi-core. Ma prima di pensare alla concorrenza, dovremo svolgere il nostro primo caso di studio, sviluppare un pacchetto di gestione delle finestre che utilizzeremo e a cui faremo riferimento in vari modi e in diversi punti del libro.

Caso di studio: un pacchetto per le immagini

La libreria standard di Python non include alcun modulo per l'elaborazione di immagini, tuttavia è possibile creare, caricare e salvare immagini usando la classe `tk.PhotoImage` di Tkinter (l'esempio `barchart2.py` mostra come fare). Sfortunatamente Tkinter è in grado di leggere e scrivere soltanto i formati di immagini GIF, PPM e PGM, scarsamente diffusi, anche se con Tcl/Tk 8.6 è previsto anche il supporto del formato PNG, molto diffuso. Anche così, la classe `tk.PhotoImage` può essere utilizzata soltanto in un singolo thread (il thread della GUI principale), perciò non serve se si desiderano gestire più immagini in modo concorrente.

Potremmo naturalmente utilizzare una libreria per immagini esterna, come Pillow (`http://github.com/python-imaging/Pillow`), oppure un altro toolkit GUI (per tracciare dati 2D potremmo utilizzare il pacchetto esterno `matplotlib`, disponibile presso `http://matplotlib.org`). Tuttavia, abbiamo deciso di implementare un nostro pacchetto personalizzato in modo da poter presentare il presente caso di studio e da realizzare una base che ci tornerà utile più avanti per un altro caso di studio.

Vogliamo che il nostro pacchetto sia in grado di memorizzare i dati di immagine in modo efficiente e di lavorare subito con Python. A questo scopo, rappresenteremo un'immagine come un array lineare di colori. Ogni colore (cioè ogni pixel) sarà rappresentato da un intero a 32 bit senza segno i cui quattro byte rappresentano i componenti alfa (trasparenza), rosso, verde e blu; si parla in questo caso di formato ARGB (*Alpha, Red, Green, Blue*). Poiché utilizziamo un array monodimensionale, il pixel nella coordinata x, y si trova nell'elemento dell'array (y × larghezza) + x. Il meccanismo è illustrato nella Figura 3.8, dove il pixel evidenziato in un'immagine 8 × 8 si trova alle coordinate (5, 1), corrispondenti alla posizione d'indice 13 ((1 × 8) + 5) nell'array.

La libreria standard di Python fornisce il modulo `array` per array monodimensionali e specifici di un tipo di dati, perciò è ideale per i nostri scopi. Tuttavia, il modulo esterno `numpy` offre un codice altamente ottimizzato per la gestione degli array (di qualsiasi numero di dimensioni), perciò è utile sfruttare questo modulo, quando è disponibile. Progetteremo quindi il pacchetto `Image` in modo da utilizzare `numpy` quando possibile, mantenendo `array` come ripiego. Ciò significa che `Image` funzionerà in tutti i casi ma non potrà sfruttare al massimo `numpy`, perché il codice deve poter lavorare in modo intercambiabile con `array.array` e `numpy.ndarray`.

Vogliamo creare e modificare immagini arbitrarie, ma vogliamo anche essere in grado di caricare immagini esistenti e di salvare immagini create o modificate. Poiché le operazioni di caricamento e salvataggio dipendono dal formato di immagine, abbiamo progettato il pacchetto in modo da avere un modulo per gestire le immagini a livello generico e moduli separati (uno per ciascun formato di immagine) per gestire il caricamento e il salvataggio. Inoltre, faremo in modo che il pacchetto sia in grado di sfruttare automaticamente qualsiasi nuovo modulo di formato che sia aggiunto anche in un secondo tempo, dopo la distribuzione, purché soddisfi i requisiti della sua interfaccia.

Il pacchetto `Image` è costituito da quattro moduli. Il modulo `Image/__init__.py` fornisce tutta la funzionalità generica, mentre gli altri tre forniscono codice specifico per salvataggio e caricamento: si tratta di `Image/Xbm.py` per il formato XBM (`.xbm`) di bitmap monocromatiche, `Image/Xpm.py` per il formato XPM (`.xpm`) di mappe di pixel a colori, e di `Image/Png.py` per il formato PNG (`.png`). Quest'ultimo formato è assai complesso, ed esiste già un modulo Python che lo supporta, PyPNG (`http://github.com/drj11/ pypng`), perciò il nostro modulo `Png.py` non farà altro che fornire un semplice wrapper (usando il pattern Adapter descritto nel Capitolo 2) per PyPNG, ove sia disponibile.

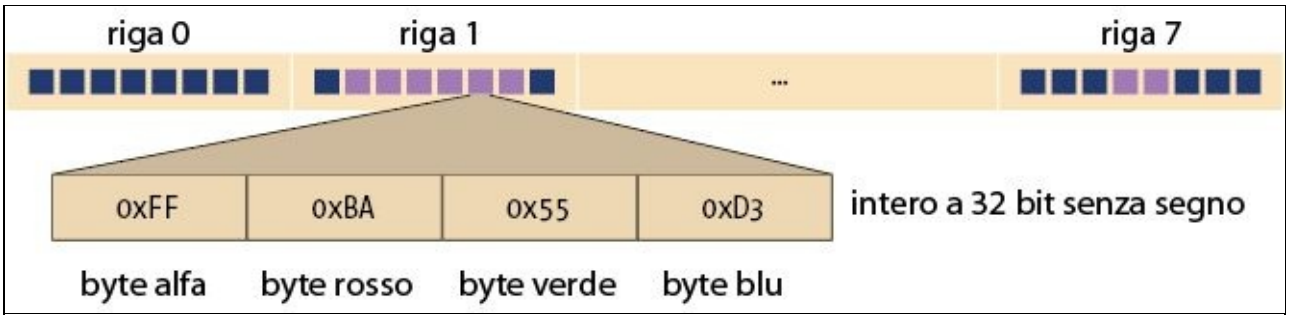


Figura 3.8 Un array di valori di colore per un'immagine 8 × 8.

Inizieremo esaminando il modulo generico (`Image/__init__.py`) e poi passeremo al modulo `Image/Xpm.py`, saltando i dettagli di basso livello. Infine, esamineremo per intero il modulo wrapper `Image/Png.py`.

Il modulo generico per le immagini

Il modulo `Image` fornisce la classe `Image` più un certo numero di funzioni e costanti per supportare l'elaborazione di immagini.

```
try:
    import numpy
except ImportError:
    numpy = None
    import array
```

Un punto fondamentale è sapere se i dati di immagine sono rappresentati utilizzando un `array.array` o un `numpy.ndarray`, perciò, dopo le normali importazioni, cerchiamo di importare `numpy`. Se l'importazione fallisce, ripieghiamo sull'importazione del modulo `array` della libreria standard per fornire le funzionalità necessarie, e creiamo una variabile `numpy` con valore `None` per i pochi punti in cui la differenza tra `array` e `numpy` è significativa.

Vogliamo che i nostri utenti siano in grado di accedere al modulo per le immagini con una semplice istruzione `import Image`. E quando fanno questo, vogliamo che tutti i moduli specifici per il caricamento e il salvataggio dei formati disponibili siano automaticamente resi disponibili. Ciò significa che l'utente dovrebbe essere in grado di creare e salvare un quadrato rosso di 64 × 64 pixel usando un codice come il seguente:

```
import Image
image = Image.Image.create(64, 64, Image.color_for_name("red"))
image.save("red_64x64.xpm")
```

Vogliamo che questo codice funzioni anche se l'utente non ha importato esplicitamente il modulo `Image/Xpm.py`. E, naturalmente, vogliamo che funzioni con ogni altro modulo specifico per un formato di immagine che si trovi nella directory `Image`, anche se è stato aggiunto dopo l'installazione iniziale del pacchetto per le immagini.

Per supportare queste funzionalità abbiamo incluso in `Image/__init__.py` del codice tenta automaticamente di caricare moduli specifici per i formati di immagine.

```
_Modules = []
for name in os.listdir(os.path.dirname(__file__)):
    if not name.startswith("_") and name.endswith(".py"):
        name = "." + os.path.splitext(name)[0]
        try:
            module = importlib.import_module(name, "Image")
            _Modules.append(module)
        except ImportError as err:
            warnings.warn("failed to load Image module: {}".format(err))
del name, module
```

Questo codice riempie la lista privata `_Modules` con qualsiasi modulo sia trovato e importato dalla directory `Image`, eccetto `__init__.py` (o qualsiasi altro modulo il cui nome inizi con un trattino basso).

Il codice opera iterando sui file presenti nella directory `Image` (se esiste nel file system). Per ciascun file `.py`, otteniamo il nome del modulo in base al nome del file. Dobbiamo prestare attenzione a far precedere il nome del modulo con il punto (`.`) poiché vogliamo importare il modulo relativo al pacchetto `Image`. Quando utilizziamo

un’importazione relativa come questa, dobbiamo fornire il nome del pacchetto come secondo argomento della funzione `importlib.import_module()`. Se l’importazione ha successo, aggiungiamo il modulo Python corrispondente alla lista di moduli; vedremo tra breve come si utilizzano.

Per evitare di appesantire il namespace `Image`, le variabili `name` e `module` vengono eliminate quando non sono più necessarie.

L’approccio a plugin utilizzato qui è facile da comprendere e da utilizzare, e funziona bene nella maggior parte dei casi. Tuttavia è affetto da una limitazione: non funziona se il pacchetto `Image` è posto all’interno di un file `.zip` (ricordiamo che Python è in grado di importare moduli che si trovano all’interno di file `.zip`: dobbiamo semplicemente inserire il file `.zip` nella lista `sys.path` e poi importarlo come se fosse un normale modulo; cfr. `http://docs.python.org/dev/library/zipimport.html`). Una soluzione per questo problema consiste nell’utilizzare la funzione `pkgutil.walk_packages()` della libreria standard (al posto di `os.listdir()`, e adattando opportunamente il codice), poiché tale funzione è in grado di lavorare sia con i pacchetti normali sia con quelli contenuti all’interno di file `.zip`; può anche cavarsela con implementazioni fornite come estensioni C e file di bytecode precompilati (`.pyc` e `.pyo`).

```
class Image:

    def __init__(self, width=None, height=None, filename=None,
            background=None, pixels=None):
        assert (width is not None and (height is not None or
                pixels is not None) or (filename is not None))
        if filename is not None:  # Da file
            self.load(filename)
        elif pixels is not None: # Da dati
            self.width = width
            self.height = len(pixels) // larghezza
            self.filename = filename
            self.meta = {}
            self.pixels = pixels
        else: # Empty
            self.width = width
            self.height = height
            self.filename = filename
            self.meta = {}
            self.pixels = create_array(width, height, background)
```

Il metodo `__init__()` della classe `Image` ha una firma piuttosto complicata, ma questo non conta, perché incoraggeremo i nostri utenti a utilizzare metodi di classe molto più semplici per creare le immagini (per esempio il metodo `Image.Image.create()` visto in precedenza in questo stesso capitolo).

```
@classmethod
def from_file(Class, filename):
    return Class(filename=filename)

@classmethod
def create(Class, width, height, background=None):
    return Class(width=width, height=height, background=background)
```

```
@classmethod
def from_data(Class, width, pixels):
    return Class(width=width, pixels=pixels)
```

Questi sono i tre metodi di classe factory per la creazione di immagini, che possono essere richiamati sulla classe stessa (per esempio con `image = Image.Image.create(200, 400)`) e funzionano correttamente anche per le sottoclassi di `Image`.

Il metodo `from_file()` crea un'immagine da un file di cui è specificato il nome. Il metodo `create()` crea un'immagine vuota con il colore di sfondo specificato (o con uno sfondo trasparente, se non è specificato alcun colore). Il metodo `from_data()` crea un'immagine con la larghezza specificata e con i pixel (cioè i colori) impostati secondo l'array monodimensionale di pixel (di tipo `array.array` o `numpy.ndarray`).

```
def create_array(width, height, background=None):
    if numpy is not None:
        if background is None:
            return numpy.zeros(width * height, dtype=numpy.uint32)
        else:
            iterable = (background for _ in range(width * height))
            return numpy.fromiter(iterable, numpy.uint32)
    else:
        typecode = "I" if array.array("I").itemsize >= 4 else "L"
        background = (background if background is not None else
                      ColorForName["transparent"])
        return array.array(typecode, [background] * width * height)
```

Questa funzione crea un array monodimensionale di interi senza segno a 32 bit (cfr. la Figura 3.8 riportata in precedenza in questo capitolo). Se `numpy` è presente e lo sfondo è trasparente, possiamo utilizzare la funzione factory `numpy.zeros()` per creare l'array con ogni intero impostato a (`0x00000000`). Qualsiasi numero con un componente alfa impostato a zero è del tutto trasparente. Se è stato specificato un colore di sfondo, creiamo un'espressione generatore che produrrà `width` × `height` valori (tutti dello stesso tipo: `background`) e passiamo questo iteratore alla funzione factory `numpy.fromiter()`.

Se `numpy` non è disponibile, dobbiamo creare un `array.array`. A differenza di `numpy`, questo modulo non ci consente di specificare la dimensione esatta degli interi che vogliamo utilizzare, perciò facciamo il meglio che possiamo. Utilizziamo lo specificatore di tipo `"I"` (intero senza segno, dimensione minima di due byte) se è effettivamente di quattro o più byte, altrimenti utilizziamo lo specificatore di tipo `"L"` (intero senza segno, dimensione minima di quattro byte). Questo allo scopo di assicurarci di utilizzare l'intero con la dimensione minima che consente di contenere quattro byte, anche su macchine a 64 bit dove un intero senza segno occuperebbe normalmente otto byte. Poi creiamo un array per contenere elementi del tipo

corrispondente allo specificatore e lo riempiamo con `width` × `height` valori per lo sfondo (esamineremo più avanti il dizionario di default `ColorForName`).

```
class Error(Exception): pass
```

Questa classe ci fornisce un tipo di eccezione `Image.Error`. Avremmo potuto utilizzare semplicemente una delle eccezioni integrate (per esempio `ValueError`), ma in questo modo sarà più facile per gli utenti del nostro pacchetto `Image` catturare eccezioni specifiche senza rischiare di confonderle con altre.

```
    def load(self, filename):
        module = Image._choose_module("can_load", filename)
    if module is not None:
        self.width = self.height = None
        self.meta = {}
        module.load(self, filename)
        self.filename = filename
    else:
        raise Error("no Image module can load files of type {}".format(
                    os.path.splitext(filename)[1]))
```

Il modulo `Image.__init__.py` non sa nulla dei formati di file per le immagini. Tuttavia, i moduli specifici hanno tale conoscenza, e sono stati caricati precedentemente quando abbiamo riempito la lista `_Modules`. I moduli specifici per le immagini potrebbero essere considerati come varianti del pattern Template Method o del pattern Strategy, entrambi descritti precedentemente in questo capitolo.

In questo caso cerchiamo di recuperare un modulo che sia in grado di caricare il file specificato. Se ne troviamo uno, inizializziamo alcune delle variabili istanza dell'immagine e indichiamo al modulo di caricare il file. Se il metodo `load()` del modulo ha successo, riempirà `self.pixels` con un array di valori di colore e imposterà opportunamente `self.width` e `self.height`; altrimenti, solleverà un'eccezione (vedremo esempi di metodi `load()` specifici per vari formati più avanti in questo stesso capitolo).

```
    @staticmethod
    def _choose_module(actionName, filename):
        bestRating = 0
        bestModule = None
        for module in _Modules:
            action = getattr(module, actionName, None)
            if action is not None:
                rating = action(filename)
                if rating > bestRating:
                    bestRating = rating
                    bestModule = module
        return bestModule
```

Questo metodo statico è utilizzato per trovare un modulo nella lista privata `_Modules` che possa svolgere l'azione (`actionName`) sul file (`filename`). Itera su tutti i moduli caricati, e per ognuno cerca di recuperare la funzione `actionName` (per esempio

`can_load()` o `can_save()`) usando la funzione integrata `getattr()`. Per ogni funzione di azione trovata, il metodo la richiama con il nome di file specificato.

La funzione di azione dovrebbe restituire un valore intero 0 se non è in grado di eseguire l'azione, 100 se è in grado di eseguirla perfettamente, o un valore intermedio se riesce a eseguirla solo in parte. Per esempio, il modulo `Image/Xbm.py` restituisce 100 per file con estensione `.xbm`, perché lo supporta pienamente, ma restituisce 0 per tutte le altre estensioni. Invece, il modulo `Image/Xpm.py` restituisce solo 80 per i file `.xpm`, perché non supporta l'intera specifica XPM (anche se funziona perfettamente su tutti i file `.xpm` sui quali è stato testato).

Al termine viene restituito il modulo con il miglior punteggio, oppure, se non viene trovato alcun modulo adatto allo scopo, viene restituito `None`.

```
    def save(self, filename=None):
        filename = filename if filename is not None else self.filename
        if not filename:
            raise Error("can't save without a filename")
        module = Image._choose_module("can_save", filename)
        if module is not None:
            module.save(self, filename)
            self.filename = filename
        else:
            raise Error("no Image module can save files of type {}".format(
                        os.path.splitext(filename)[1]))
```

Questo metodo è molto simile al metodo `load()` perché cerca di ottenere un modulo che sia in grado di salvare un file con il nome specificato (cioè che possa salvare nel formato indicato dall'estensione del file), ed esegue il salvataggio.

```
    def pixel(self, x, y):
        return self.pixels[(y * self.width) + x]
```

Il metodo `pixel()` restituisce il colore nella posizione data come valore ARGB (un intero a 32 bit senza segno).

```
    def set_pixel(self, x, y, color):
        self.pixels[(y * self.width) + x] = color
```

Il metodo `set_pixel()` imposta il pixel dato al valore ARGB specificato se le coordinate `x` e `y` rientrano nell'intervallo di validità, altrimenti solleva un'eccezione `IndexError`.

Il modulo `Image` fornisce alcuni metodi di base per il disegno, tra cui `line()`, `ellipse()` e `rectangle()`. Ci limitiamo a mostrarne uno particolarmente rappresentativo.

```
    def line(self, x0, y0, x1, y1, color):
    Dx = abs(x1 - x0)
    Dy = abs(y1 - y0)
        xInc = 1 if x0 < x1 else -1
        yInc = 1 if y0 < y1 else -1
        d = Dx - Dy
```

```
        while True:
            self.set_pixel(x0, y0, color)
            if x0 == x1 and y0 == y1:
                break
            d2 = 2 * d
            if d2 > -Dy:
                d -= Dy
                x0 += xInc
            if d2 < Dx:
                d += Dx
                y0 += yInc
```

Questo metodo utilizza l'algoritmo di Bresenham (spiegato presso `http://en.wikipedia.org/wiki/Bresenham's_line_algorithm`, e che richiede soltanto l'aritmetica degli interi) per disegnare una linea dal punto (`x0`, `y0`) al punto (`x1`, `y1`). Grazie al supporto di Unicode in Python 3, possiamo utilizzare nomi di variabili naturali per questo contesto; per esempio, `Dx` e `Dy` per rappresentare differenze nei valori delle coordinate x e y, e `d` e `d2` per i valori di errore.

```
    def scale(self, ratio):
        assert 0 < ratio < 1
        rows = round(self.height * ratio)
        columns = round(self.width * ratio)
        pixels = create_array(columns, rows)
        yStep = self.height / rows
        xStep = self.width / columns index = 0
        for row in range(rows):
            y0 = round(row * yStep)
            y1 = round(y0 + yStep)
            for column in range(columns):
                x0 = round(column * xStep)
                x1 = round(x0 + xStep)
                pixels[index] = self._mean(x0, y0, x1, y1)
                index += 1
        return self.from_data(columns, pixels)
```

Questo metodo crea e restituisce una nuova versione ridotta dell'immagine. Il rapporto dimensionale deve essere compreso nell'intervallo (0.0, 1.0), dove 0.75 produce un'immagine di larghezza e altezza pari a 3/4 dell'originale, mentre un rapporto di 0.5 produce un'immagine di dimensione pari a 1/4 dell'originale, cioè di metà altezza e metà larghezza. Ogni pixel (cioè ogni colore) nell'immagine risultante è la media dei colori nel rettangolo dell'immagine di origine che tale pixel deve rappresentare.

Le coordinate x, y dell'immagine sono interi, ma per evitare imprecisioni dobbiamo utilizzare l'aritmetica dei numeri in virgola mobile (per esempio utilizzare `/` anziché `//`) elaborando i dati dei pixel. Perciò utilizziamo la funzione integrata `round()` ogni volta che ci servono valori interi. Alla fine utilizziamo il metodo di classe factory `Image.Image.from_data()` per creare una nuova immagine basata sul numero calcolato di colonne e usando l'array di pixel che abbiamo creato e riempito di colori.

```
    def _mean(self, x0, y0, x1, y1):
        aTotal, redTotal, greenTotal, blueTotal, count = 0, 0, 0, 0, 0
        for y in range(y0, y1):
            if y >= self.height:
```

```
                break
            offset = y * self.width
            for x in range(x0, x1):
                if x >= self.width:
                    break
                a, r, g, b = self.argb_for_color(self.pixels[offset + x])
                aTotal += a
                redTotal += r
                greenTotal += g
                blueTotal += b count += 1
        a = round(aTotal / count)
        r = round(redTotal / count)
        g = round(greenTotal / count)
        b = round(blueTotal / count)
        return self.color_for_argb(a, r, g, b)
```

Questo metodo privato accumula le somme dei componenti alfa, rosso, verde e blu di tutti i pixel nel rettangolo specificato dalle coordinate `x0`, `y0`, `x1`, `y1`. Ognuna di queste somme viene poi divisa per il numero di pixel esaminati, per produrre un colore che è la media di tali pixel. Il processo è illustrato nella Figura 3.9.

```
MAX_ARGB = 0xFFFFFFFF
MAX_COMPONENT = 0xFF
```

Il minimo valore ARGB a 32 bit è `0x0` (cioè `0x00000000`, trasparente; in termini rigorosi, nero trasparente). Queste due costanti del modulo `Image` specificano il valore ARGB massimo (bianco pieno) e il massimo valore di qualsiasi componente di colore (255).

```
    @staticmethod
    def argb_for_color(color):
        if numpy is not None:
            if isinstance(color, numpy.uint32):
                color = int(color)
        if isinstance(color, str):
            color = color_for_name(color)
        elif not isinstance(color, int) or not (0 <= color <= MAX_ARGB):
            raise Error("invalid color {}".format(color))
        a = (color >> 24) & MAX_COMPONENT
        r = (color >> 16) & MAX_COMPONENT
        g = (color >> 8) & MAX_COMPONENT
        b = (color & MAX_COMPONENT)
        return a, r, g, b
```

Questo metodo statico (e funzione di modulo) restituisce quattro componenti di colore (ognuno compreso nell’intervallo da 0 a 255) per un colore dato. Il colore passato può essere un `int`, un `numpy.uint32` oppure una `str` che ne indica il nome. I singoli componenti di colore (byte) dell’`int` che rappresenta il colore sono quindi estratti (come `int`) usando operazioni di shift sui bit (`>>`) ed and sui bit (`&`).

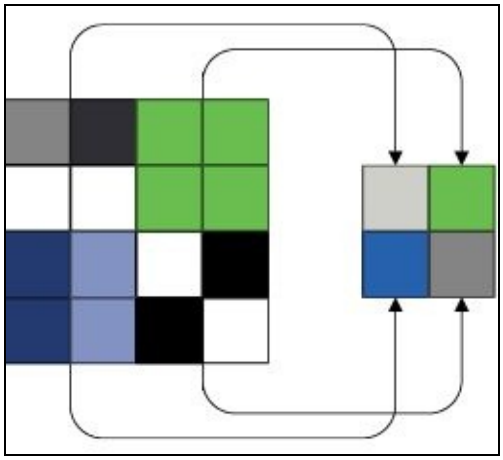


Figura 3.9 Ridimensionamento di un’immagine 4 × 4 per un fattore 0.5.

```
    @staticmethod
    def color_for_name(name):
        if name is None:
            return ColorForName["transparent"]
        if name.startswith("#"):
            name = name[1:]
            if len(name) == 3:        # aggiunge alfa pieno
                name = "F" + name     # ora ha 4 cifre hex
            if len(name) == 6:        # aggiunge alfa pieno
                name = "FF" + name    # ora ha tutte le 8 cifre hex
            if len(name) == 4:        # in origine #FFF o #FFFF
                components = []
                for h in name:
                    components.extend([h, h])
                name = "".join(components)    # ora ha tutte le 8 cifre hex
            return int(name, 16)
        return ColorForName[name.lower()]
```

Questo metodo statico (e funzione di modulo) restituisce un valore ARGB a 32 bit per un dato colore `str`. Se il colore passato è `None`, il metodo restituisce un colore trasparente. Se la stringa inizia con `#`, si assume che sia un colore in stile HTML nella forma `"#HHH"`, `"#HHHH"`, `"#HHHHHH"` o `"#HHHHHHHH"`, dove `H` è una cifra esadecimale. Se il numero di cifre fornite è sufficiente soltanto per un valore RGB, aggiungiamo come prefisso due `"F"` per fare in modo che abbia un canale alfa opaco. Altrimenti, restituiamo il colore dal dizionario `ColorForName`; in questo caso si ha sempre successo perché `ColorForName` è un dizionario di default.

```
ColorForName = collections.defaultdict(lambda: 0xFF000000, {
    "transparent": 0x00000000, "aliceblue": 0xFFF0F8FF,
    ...
    "yellow4": 0xFF8B8B00, "yellowgreen": 0xFF9ACD32})
```

`ColorForName` è un `collections.defaultdict` che restituisce un intero senza segno a 32 bit che codifica i componenti alfa, rosso, verde e blu di un colore specificato, oppure, se il nome del colore non è nel dizionario, restituisce un nero pieno (`0xFF000000`). Benché gli utenti di `Image` siano liberi di utilizzare questo dizionario, la funzione `color_for_name()` è più comoda e versatile. I nomi di colore sono tratti dal file `rgb.txt` fornito con X11, con l’aggiunta del colore trasparente.

La funzione `collections.defaultdict()` accetta una funzione factory come primo argomento, seguito da qualsiasi argomento che un normale `dict` accetti. La funzione factory è usata per produrre il valore per qualsiasi elemento che sia portato in vita quando si accede a una chiave mancante. In questo caso abbiamo usato un `lambda` che restituisce sempre lo stesso valore (nero pieno). Benché sia possibile passare argomenti parole chiave (per esempio `transparent=0x00000000`), il numero di colori è superiore al limite di Python di 255 argomenti, perciò inizializziamo il dizionario di default con un dizionario normale creato con la sintassi `{key: value}`, che non ha tale limite.

```
argb_for_color = Image.argb_for_color
rgb_for_color = Image.rgb_for_color
color_for_argb = Image.color_for_argb
color_for_rgb = Image.color_for_rgb
color_for_name = Image.color_for_name
```

Dopo la classe `Image` abbiamo creato alcune funzioni di utilità basate su alcuni dei metodi statici della classe. Questo significa, per esempio, che dopo aver eseguito l'importazione con `import Image`, l'utente può richiamare `Image.color_for_name()` o, se ha un'istanza di `Image.Image`, `image.color_for_name()`.

Con questo abbiamo completato l'esame del modulo core `Image` (in `Image/__init__.py`). Abbiamo omesso alcune costanti meno interessanti, qualche metodo di `Image.Image` (`rectangle()`, `ellipse()` e `subsample()`), la proprietà `size` (che restituisce semplicemente una coppia di larghezza e altezza) e vari metodi statici per la gestione dei colori. Il modulo è sufficiente per creare, caricare, disegnare e salvare file di immagini utilizzando i formati XBM e XPM, e se è installato il modulo PyPNG, anche con il formato PNG.

Ora esamineremo due moduli specifici per un formato di immagine su cui il modulo `Image` conta molto. Non tratteremo il modulo `Image/Xbm.py`, perché, a parte i dettagli di basso livello relativi al formato XBM, non aggiungeremmo alcunché a quanto possiamo apprendere dall'esame del modulo `Image/Xpm.py`.

Panoramica sul modulo Xpm

Ogni modulo dedicato a uno specifico formato di immagine dovrebbe fornire quattro funzioni. Due sono `can_load()` e `can_save()`, che dovrebbero restituire entrambe 0 per indicare che non sono in grado di operare, 100 per indicare che sono in grado, o un valore intermedio per indicare che possono operare solo in maniera imperfetta. Il modulo dovrebbe anche fornire funzioni `load()` e `save()`, che dovrebbero essere

richiamate soltanto con un nome di file per cui la corrispondente funzione `can_load()` o `can_save()` abbia restituito un valore diverso da zero.

```
    def can_load(filename):
        return 80 if os.path.splitext(filename)[1].lower() == ".xpm" else 0

    def can_save(filename):
        return can_load(filename)
```

Il modulo `Image/Xpm.py` implementa gran parte della specifica XPM, a parte alcune caratteristiche utilizzate raramente. Alla luce di ciò, riporta un valore di 80 (quindi non perfetto) sia per il caricamento, sia per il salvataggio.

NOTA

Un'alternativa al controllo del tipo di file in base all'estensione consiste nel leggere i primi byte, cioè il suo "numero magico". Per esempio, i file XPM iniziano con i byte `0x2F 0x2A 0x20 0x58 0x50 0x4D 0x20 0x2A 0x2F ("/* XPM */")` e i file PNG con `0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A ("·PNG····")`).

Questo significa che, se è stato aggiunto un nuovo modulo di gestione di XPM, per esempio come `Image/Xpm2.py`, purché riporti un punteggio maggiore di 80, tale modulo sarà utilizzato al posto di questo (abbiamo discusso questo precedentemente esaminando il metodo `Image._choose_module()`).

```
(_WANT_XPM, _WANT_NAME, _WANT_VALUES, _WANT_COLOR, _WANT_PIXELS, _DONE) = ("WANT_XPM",
                         "WANT_NAME", "WANT_VALUES", "WANT_COLOR", "WANT_PIXELS", "DONE")
_CODES = "".join((chr(x) for x in range(32, 127) if chr(x) not in '\\"'))
```

Il formato XPM utilizza testo normale (ASCII a 7 bit) che deve essere analizzato per estrarne i dati. Il formato è costituito da alcuni metadati (larghezza, altezza, numero di colori e così via), da una tabella di colori e dai dati che identificano ciascun pixel per riferimento alla tabella di colori. Esaminare i dettagli significherebbe allontanarci eccessivamente da Python, ci limitiamo a utilizzare un semplice parser codificato a mano, e queste costanti forniscono gli stati del parser.

```
def load(image, filename):
    colors = cpp = count = None
    state = _WANT_XPM
    palette = {}
    index = 0
    with open(filename, "rt", encoding="ascii") as file:
        for lino, line in enumerate(file, start=1):
            line = line.strip()
            ...
```

Questo è l'inizio della funzione `load()` del modulo. L'immagine passata è di tipo `Image.Image`, e all'interno della funzione (in una parte non mostrata qui), gli attributi `pixels`, `width` e `height` sono tutti impostati direttamente. L'array di pixel è creato utilizzando la funzione `Image.create_array()` in modo che il modulo `Xpm.py` non debba conoscere né preoccuparsi del fatto che l'array sia un `array.array` oppure un

`numpy.ndarray`, purché sia monodimensionale e di lunghezza pari a larghezza × altezza. Questo significa, tuttavia, che dobbiamo accedere all'array di pixel soltanto utilizzando metodi comuni a entrambi i tipi.

```
def save(image, filename):
    name = Image.sanitized_name(filename)
    palette, cpp = _palette_and_cpp(image.pixels)
    with open(filename, "w+t", encoding="ascii") as file:
        _write_header(image, file, name, cpp, len(palette))
        _write_palette(file, palette)
        _write_pixels(image, file, palette)
```

I formati XBM e XPM includono entrambi un nome, nel file effettivo, basato sul loro nome di file, ma che deve essere un identificatore valido in linguaggio C. Otteniamo questo nome utilizzando la funzione `Image.sanitized_name()`. Quasi tutto il lavoro per il salvataggio è affidato a funzioni ausiliarie private, nessuna delle quali riveste particolare interesse in questo caso, perciò non le trattiamo.

```
def sanitized_name(name):
    name = re.sub(r"\W+", "", os.path.basename(os.path.splitext(name)[0]))
    if not name or name[0].isdigit():
        name = "z" + name
    return name
```

La funzione `Image.sanitized_name()` accetta un nome di file e produce un nome, basato su di esso, che contiene soltanto lettere dell'alfabeto latino non accentate, cifre e trattini bassi, e che inizia con una lettera o un trattino basso. Nell'espressione regolare, `\W+` corrisponde a uno o più caratteri non parole (cioè caratteri che non sono validi negli identificatori in C).

È possibile aggiungere al modulo `Image` il supporto per qualsiasi altro formato creando un modulo adatto che vada nella directory `Image` e abbia le quattro funzioni richieste: `can_load()`, `can_save()`, `load()` e `save()`, dove le prime due restituiscono interi appropriati per i nomi di file forniti. Un formato di immagine molto diffuso è PNG, che tuttavia è piuttosto complesso. Fortunatamente possiamo adattare il modulo PyPNG in modo da sfruttarlo a nostro vantaggio con un minimo sforzo, come vedremo nel seguito.

Il modulo wrapper PNG

Il modulo PyPNG (`http://github.com/drj11/pypng`) fornisce un buon supporto per il formato di immagine PNG, ma è privo dell'interfaccia richiesta dal modulo `Image` per i moduli specifici di un particolare formato. Perciò, in questo paragrafo creeremo il modulo `Image/Png.py`, che utilizzerà il pattern Adapter (descritto nel Capitolo 2) per aggiungere il supporto per le immagini PNG al modulo `Image`. A differenza di quanto

abbiamo fatto nel paragrafo precedente, in cui abbiamo mostrato soltanto un piccolo campione di codice, in questo caso presenteremo tutto il codice del modulo `Image/Png.py`.

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

Iniziamo tentando di importare il modulo `png` di PyPNG. In caso di fallimento creiamo una variabile `png` di valore `None` che possiamo controllare più avanti.

```
def can_load(filename):
    return (80 if png is not None and
            os.path.splitext(filename)[1].lower() == ".png" else 0)

def can_save(filename):
    return can_load(filename)
```

Se il modulo `png` è stato importato con successo, restituiamo un punteggio di 80 (leggermente imperfetto) a indicare il supporto per il formato PNG di questo modulo. Utilizziamo 80 anziché 100 per consentire a un altro modulo di superare il nostro. Come per il formato XPM, restituiamo lo stesso punteggio per caricamento e salvataggio, anche se sarebbe del tutto lecito restituire punteggi diversi.

```
def load(image, filename):
    reader = png.Reader(filename=filename)
    image.width, image.height, pixels, _ = reader.asRGBA8()
    image.pixels = Image.create_array(image.width, image.height)
    index = 0
    for row in pixels:
        for r, g, b, a in zip(row[::4], row[1::4], row[2::4], row[3::4]):
            image.pixels[index] = Image.color_for_argb(α, r, g, b)
            index += 1
```

Iniziamo creando un `png.Reader` e fornendogli il nome di file che abbiamo ricevuto: otteniamo così che il file PNG sia caricato nell’istanza `reader`. Poi estraiamo i valori di larghezza, altezza e pixel dell’immagine, e scartiamo i metadati.

Il modulo PyPNG utilizza il formato RGBA, mentre il nostro modulo `Image` utilizza il formato ARGB, perciò dobbiamo tenere conto di questa differenza. A questo scopo estraiamo i pixel utilizzando il metodo `png.Reader.asRGBA8()`, che restituisce un array bidimensionale le cui righe contengono valori di componenti di colore. Per esempio, i pixel per un’immagine la cui prima riga inizia con un pixel rosso pieno seguito da un pixel blu pieno comporterebbero per la prima riga una lista di valori che inizia con: `0xFF, 0x00, 0x00, 0xFF, 0x00, 0x00, 0xFF, 0xFF`.

Una volta ottenuti i pixel RGBA, creiamo un nuovo array della dimensione corretta e con tutti i pixel impostati come trasparenti. Possiamo poi iterare su ciascuna riga dei componenti di colore ed estrarre ciascun tipo di componente. Per

esempio, i componenti del rosso si trova nelle posizioni di riga 0, 4, 8, 12, …, quelli del verde nelle posizioni 1, 5, 9, 13, …, quelli del blu in 2, 6, 10, 14, …, e quelli in 3, 7, 11, 15, … Possiamo quindi utilizzare la funzione integrata `zip()` per produrre tuple di 4 componenti di colore. Così la prima 4-tupla è ricavata dalla prima riga nelle posizioni d'indice (0, 1, 2, 3), la seconda 4-tupla è ricavata dalle posizioni d'indice (4, 5, 6, 7), e così via. Per ogni tupla creiamo un valore di colore ARGB e lo inseriamo nell'array di pixel monodimensionale della nostra immagine.

```
def save(image, filename):
    with open(filename, "wb") as file:
        writer = png.Writer(width=image.width, height=image.height,
                    alpha=True)
        writer.write_array(file, list(_rgba_for_pixels(image.pixels)))
```

La funzione `save()` delega la maggior parte del suo lavoro al modulo `png`. Inizia creando un `png.Writer` con alcuni metadati appropriati, e poi scrive tutti i pixel su di esso. Poiché `Image` utilizza valori ARGB e `png` utilizza valori RGBA, abbiamo usato una funzione ausiliaria privata per convertire dagli uni agli altri.

```
def _rgba_for_pixels(pixels):
    for color in pixels:
        a, r, g, b = Image.argb_for_color(color)
        for component in (r, g, b, a):
            yield component
```

Questa funzione itera sull'array fornito (`image.pixels`) e separa ciascun componente di ogni colore. Poi restituisce ciascuno di questi componenti (secondo l'ordine RGBA) al chiamante.

Il codice riportato in questo paragrafo è completo, perché tutto il lavoro pesante è svolto dal modulo `png` di PyPNG.

Il modulo `Image` fornisce un'utile interfaccia per disegnare (`set_pixel()`, `line()`, `rectangle()`, `ellipse()`) e il supporto per caricare e salvare immagini nei formati XBM, XPM e (se è installato PyPNG) PNG. Fornisce anche un metodo `subsample()` (per un rapido e grezzo ridimensionamento) e un metodo `scale()` (per un ridimensionamento accurato), oltre ad alcune funzioni di manipolazione di colori e metodi statici.

Il modulo `Image` può essere usato in contesti di elaborazione concorrente, per esempio per creare, caricare, disegnare e salvare immagini in più thread o processi, cosa che lo rende più comodo per esempio di Tkinter, che è in grado di gestire le immagini soltanto nel thread principale (GUI). Sfortunatamente, tuttavia, il ridimensionamento è piuttosto lento. Un modo per aumentare la velocità di ridimensionamento - purché si disponga di una macchina multi-core e di più immagini da scalare nello stesso tempo - è quello di usare la concorrenza, come

vedremo nel Capitolo 4. Tuttavia, l’operazione di ridimensionamento è CPU-bound, perciò i miglioramenti più significativi in cui possiamo sperare per il ricorso alla concorrenza sono proporzionali al numero di processori; per esempio, su una macchina a quattro core il massimo che potremmo ottenere sarebbe un incremento della velocità di non più di 4×. Perciò, nel Capitolo 5 vedremo come utilizzare Cython per un incremento molto più significativo della velocità.

Concorrenza ad alto livello

L’interesse nei confronti della programmazione concorrente è cresciuto rapidamente dall’inizio del nuovo millennio. L’accelerazione è stata favorita da Java, che ha reso la concorrenza molto più comune, dall’onnipresenza delle macchine multicore e dal fatto che il supporto per la programmazione concorrente è disponibile nella maggior parte dei linguaggi di programmazione moderni.

Scrivere e provvedere alla manutenzione di programmi concorrenti è più difficile (a volte molto più difficile) che scrivere e mantenere programmi non concorrenti. Inoltre, i programmi concorrenti possono a volte avere prestazioni inferiori (anche di molto) rispetto a programmi equivalenti non concorrenti. Tuttavia, quando sono scritti bene, i programmi concorrenti possono avere prestazioni tanto superiori a quelle dei cugini non concorrenti da giustificare abbondantemente il maggiore sforzo necessario.

La maggior parte dei linguaggi moderni (tra cui C++ e Java) supporta la concorrenza in modo diretto e di solito fornisce anche funzionalità aggiuntive di alto livello tramite proprie librerie standard. La concorrenza può essere implementata in diversi modi; la principale distinzione è data dall’accesso diretto (cioè utilizzando la memoria condivisa) o indiretto (cioè utilizzando la comunicazione inter-processo, o IPC) ai dati. La concorrenza basata sui thread si ha quando diversi thread di esecuzione operano all’interno dello stesso processo di sistema. Tipicamente questi thread accedono a dati condivisi tramite un accesso serializzato alla memoria condivisa; la serializzazione viene realizzata dal programmatore per mezzo di un determinato meccanismo di blocco. La concorrenza basata sui processi (multiprocessing) si ha quando più processi distinti vengono eseguiti in modo indipendente. Tipicamente, i processi concorrenti accedono ai dati condivisi utilizzando la IPC, sebbene possano utilizzare anche la memoria condivisa se il linguaggio o le sue librerie lo consentono. Un altro tipo di concorrenza è quella basata sulla “attesa concorrente” piuttosto che sull’esecuzione concorrente; è l’approccio adottato dalle implementazioni dell’I/O asincrono.

Python è dotato di un supporto di basso livello per l’I/O asincrono (i moduli `asyncore` e `asynchat`). Il supporto di alto livello è offerto dalla piattaforma esterna Twisted (`http://twistedmatrix.com`). L’inserimento nella libreria standard del supporto

per l'I/O asincrono di alto livello, comprendente i cicli di eventi, è previsto per la versione 3.4 di Python (`http://www.python.org/dev/peps/pep-3156`).

Per quanto riguarda la più tradizionale concorrenza basata sui thread e sui processi, Python supporta entrambi gli approcci. Il supporto di Python per il threading è piuttosto convenzionale, ma il supporto per il multiprocessing è di livello molto più alto rispetto a quello della maggior parte degli altri linguaggi e librerie. Inoltre, il supporto di Python per il multiprocessing utilizza le stesse astrazioni del threading per far sì che il passaggio da un approccio all'altro sia agevole, perlomeno quando non viene utilizzata la memoria condivisa.

A causa del GIL (*Global Interpreter Lock*), l'interprete Python può essere in esecuzione solamente su uno dei core del processore (questa limitazione non vale per Jython e per alcuni altri interpreti Python; nessuno degli esempi di concorrenza riportati in questo libro fa affidamento sulla presenza o sull'assenza del GIL). Il codice C può acquisire e rilasciare il GIL, quindi non ha lo stesso vincolo, e buona parte di Python (e delle sue librerie standard) sono scritte in C. Nondimeno, questo significa che una concorrenza realizzata per mezzo del threading potrebbe non produrre gli incrementi di velocità sperati.

In generale, per l'elaborazione CPU-bound, il ricorso al threading può facilmente condurre a prestazioni peggiori di quelle che si avrebbero senza concorrenza. Una soluzione consiste nello scrivere il codice in Cython (cfr. il Capitolo 5), che è essenzialmente Python con della sintassi aggiuntiva e che viene compilato in puro C. In questo modo si può arrivare a centuplicare la velocità; molto più di quanto sarebbe probabilmente ottenibile ricorrendo a un qualsiasi tipo di concorrenza, con la quale il miglioramento delle prestazioni è proporzionale al numero dei core. Se però la concorrenza è l'approccio giusto, allora per l'elaborazione CPU-bound è meglio evitare del tutto il GIL utilizzando il modulo `multiprocessing`. Se si utilizza tale modulo, invece di utilizzare thread di esecuzione separati all'interno dello stesso processo (e quindi di determinare una competizione per il GIL), si hanno processi separati, ciascuno dei quali utilizza la propria istanza indipendente dell'interprete Python, senza conflitti.

Per l'elaborazione I/O-bound (per esempio nel caso delle attività di rete), l'utilizzo della concorrenza può comportare incrementi di velocità notevoli. Spesso in questi casi la latenza della rete è un fattore talmente preponderante che il fatto che la concorrenza venga realizzata con il threading o con il multiprocessing diventa irrilevante.

Consigliamo di iniziare scrivendo dapprima un programma non concorrente, quando possibile, dato che è più semplice e rapido da scrivere di uno concorrente, e più facile da testare. Una volta ottenuto un programma non concorrente corretto, può anche darsi che esso risulti sufficientemente veloce così com'è. Se invece il programma non è veloce, può comunque essere utilizzato per un confronto con una versione concorrente sia in termini sia di risultati (cioè di correttezza) sia di prestazioni. Per quanto riguarda il tipo di concorrenza da scegliere, consigliamo il multiprocessing per i programmi CPU-bound e il multiprocessing oppure il threading per i programmi I/O-bound. In ogni caso, non conta solamente il tipo di concorrenza, ma anche il livello.

In questo libro definiamo tre livelli di concorrenza:

- **Concorrenza a basso livello**: è una concorrenza che fa uso esplicito di operazioni atomiche. Questo tipo di concorrenza è adatto per chi scrive librerie più che per gli sviluppatori di applicazioni, dato che è facile commettere errori e che eseguirne il debugging può essere estremamente difficoltoso. Python non supporta questo tipo di concorrenza, benché le implementazioni della concorrenza di Python siano tipicamente realizzate utilizzando operazioni di basso livello.
- **Concorrenza a livello intermedio**: è una concorrenza che non utilizza operazioni atomiche esplicite, ma utilizza lock espliciti. È il livello di concorrenza supportato dalla maggior parte dei linguaggi di programmazione. Python supporta la programmazione concorrente a questo livello con classi come `threading.Semaphore`, `threading.Lock` e `multiprocessing.Lock`. Questo livello di concorrenza viene comunemente utilizzato dai programmatori di applicazioni, dato che spesso è l'unico disponibile.
- **Concorrenza ad alto livello**: è una concorrenza che non prevede operazioni atomiche esplicite né lock espliciti (è possibile che lock e operazioni atomiche avvengano dietro le quinte, ma non è necessario occuparsene). Alcuni linguaggi moderni stanno iniziando a supportare la concorrenza ad alto livello. Python ha il modulo `concurrent.futures` (Python 3.2) e le classi collezione `queue.Queue` e `multiprocessing`, per supportare la concorrenza di alto livello.

Gli approcci di livello intermedio alla concorrenza sono semplici da utilizzare, ma tendono a causare molti errori. Sono particolarmente vulnerabili a problemi sottili e difficili da individuare, oltre che a spettacolari crash e inceppamenti dei programmi, che si manifestano senza alcuno schema riconoscibile.

Il problema fondamentale è la condivisione dei dati. I dati condivisi mutabili devono essere protetti mediante lock per garantire che tutti gli accessi siano serializzati (cioè che solamente un thread o processo alla volta possa accedere ai dati). Inoltre, quando più processi o thread tentano di accedere agli stessi dati condivisi, tutti tranne uno vengono bloccati (cioè rimangono inattivi). Ciò significa che, quando è attivo un lock, può accadere che l'applicazione utilizzi solamente un thread o processo (come se fosse un'applicazione non concorrente), mentre tutti gli altri rimangono in attesa. Perciò è opportuno che i lock vengano utilizzati il meno frequentemente possibile e per il tempo più breve possibile. La soluzione più semplice consiste nel non condividere affatto i dati mutabili. In questo modo non occorrono lock espliciti e la maggior parte dei problemi legati alla concorrenza semplicemente non si pone.

A volte, ovviamente, capita che più thread o processi concorrenti debbano accedere agli stessi dati, ma possiamo risolvere queste situazioni senza ricorrere a lock (espliciti). Una soluzione consiste nell'utilizzare una struttura di dati che preveda l'accesso concorrente. Il modulo `queue` mette a disposizione diverse code thread-safe, mentre per la concorrenza basata sul multiprocessing si possono utilizzare le classi `multiprocessing.JoinableQueue` e `multiprocessing.Queue`. Possiamo utilizzare queste code per indicare un'unica fonte di job per tutti i thread o i processi concorrenti e un'unica destinazione per i risultati, lasciando che dei lock si occupi la struttura di dati stessa.

Se abbiamo dati che vogliamo utilizzare in modo concorrente, ma per i quali non sarebbe adeguato utilizzare una coda che supporti la concorrenza, allora il modo migliore di procedere senza ricorrere ai lock consiste nel passare dati non mutabili (cioè numeri o stringhe) oppure dati mutabili che vengano solamente letti. Se è necessario utilizzare dati mutabili, l'approccio migliore consiste nell'effettuarne una copia profonda. La copia consente di evitare le complicazioni e i rischi che i lock comportano, al costo dell'elaborazione e della memoria necessari per la copia stessa. In alternativa, per il multiprocessing, è possibile utilizzare tipi di dati che supportino l'acceso concorrente (in particolare `multiprocessing.Value` per un singolo valore modificabile o `multiprocessing.Array` per un array di valori modificabili), sempre che essi siano creati da un `multiprocessing.Manager`, come vedremo più avanti nel capitolo.

Nei primi due paragrafi di questo capitolo esamineremo la concorrenza utilizzando due applicazioni: una CPU-bound e una I/O-bound. In entrambi i casi utilizzeremo gli strumenti di Python per la concorrenza ad alto livello, sia le collaudate code thread-safe sia il nuovo (Python 3.2) modulo `concurrent.futures`.

Il terzo paragrafo di questo capitolo illustra un caso di studio che mostra come utilizzare l’elaborazione concorrente in un’applicazione GUI (*Graphical User Interface*) mantenendo un’interfaccia grafica reattiva che indichi il grado di avanzamento e consenta l’annullamento dell’operazione.

Concorrenza CPU-bound

Nel caso di studio dedicato a `Image` del Capitolo 3 abbiamo mostrato il codice per ridimensionare un'immagine, notando che l'operazione era piuttosto lenta. Immaginiamo di voler scalare un'intera serie di immagini e di volerlo fare nel modo più veloce possibile sfruttando la presenza di più core.

Il ridimensionamento o scalatura delle immagini è un'operazione CPU-bound, quindi ci si aspetta che il multiprocessing sia in grado di produrre le prestazioni migliori, e ciò è avvalorato dai tempi mostrati nella Tabella 4.1 (i tempi sono stati rilevati su una macchina AMD64 3 GHz quad-core con poco carico durante l'elaborazione di 56 immagini per una dimensione complessiva di 316 MiB). Nel caso di studio del Capitolo 5 combineremo il multiprocessing con Cython per ottenere incrementi di velocità molto maggiori.

Tabella 4.1 Confronto tra le velocità di ridimensionamento delle immagini.

Programma	Concorrenza	Secondi	Incremento velocità
`imagescale-s.py`	*Nessuna*	784	*Riferimento*
`imagescale-c.py`	4 coroutine	781	1,00×
`imagescale-t.py`	4 thread con un pool di thread	1339	0,59×
`imagescale-q-m.py`	4 processi con una coda	206	3,81×
`imagescale-m.py`	4 processi con un pool di processi	201	3,90×

I risultati per il programma `imagescale-t.py` che utilizza quattro thread illustrano con chiarezza che l'utilizzo del threading per l'elaborazione CPU-bound produce prestazioni peggiori rispetto a quelle di programmi non concorrenti. Ciò dipende dal fatto che tutta l'elaborazione veniva eseguita da Python sullo stesso core e, oltre al ridimensionamento, Python doveva mantenere il contesto nei passaggi tra l'uno e l'altro dei quattro diversi thread, il che rappresenta un notevole carico aggiuntivo. Confrontiamo tutto ciò con le versioni multiprocessing, entrambe le quali erano in grado di distribuire il lavoro su tutti i core della macchina. La differenza tra la versione con coda multiprocessing e quella con pool di processi non è significativa; entrambe hanno mostrato incrementi di velocità dell'ordine che ci aspettavamo, cioè direttamente proporzionali al numero di core.

NOTA

Avviare nuovi processi è molto più dispendioso su Windows che sulla maggior parte degli altri sistemi operativi. Fortunatamente le code e i pool di Python utilizzano pool di processi persistenti dietro le quinte in modo da evitare di incorrere ripetutamente in questi costi di avvio dei processi.

Tutti i programmi di ridimensionamento delle immagini accettano parametri dalla riga di comando, che vengono poi esaminati con `argparse`. In tutte le versioni, i

parametri sono la dimensione cui ridurre le immagini, la scelta di utilizzare o meno la scalatura (viene utilizzata in tutti i nostri esempi cronometrati) e le directory di origine e di destinazione delle immagini. Le immagini le cui dimensioni sono inferiori a quelle indicate vengono copiate e non ridimensionate; tutte le immagini utilizzate per il cronometraggio erano da ridimensionare. Per le versioni concorrenti, è possibile anche specificare la concorrenza (cioè quanti thread o processi utilizzare); ciò esclusivamente ai fini del debugging e del cronometraggio. Per i programmi CPU-bound, normalmente utilizzeremmo un numero di thread o di processi pari al numero dei core. Per i programmi I/O-bound, si utilizza un multiplo del numero di core (2×, 3×, 4× o più) a seconda della larghezza di banda di rete.

Per completezza, riportiamo di seguito la funzione `handle_commandline()` utilizzata nei programmi concorrenti di ridimensionamento delle immagini.

```
def handle_commandline():
    parser = argparse.ArgumentParser()
    parser.add_argument("-c", "--concurrency", type=int,
                default=multiprocessing.cpu_count(),
                help="specify the concurrency (for debugging and "
                    "timing) [default: %(default)d]")
    parser.add_argument("-s", "--size", default=400, type=int,
                help="make a scaled image that fits the given dimension "
                    "[default: %(default)d]")
    parser.add_argument("-S", "--smooth", action="store_true",
                help="use smooth scaling (slow but good for text)")
    parser.add_argument("source",
                help="the directory containing the original .xpm images")
    parser.add_argument("target",
                help="the directory for the scaled .xpm images")
    args = parser.parse_args()
    source = os.path.abspath(args.source)
    target = os.path.abspath(args.target)
    if source == target:
        args.error("source and target must be different")
    if not os.path.exists(args.target):
        os.makedirs(target)
    return args.size, args.smooth, source, target, args.concurrency
```

Normalmente non daremmo agli utenti facoltà di scelta riguardo alla concorrenza, ma può essere utile farlo per il debugging, il cronometraggio e i test, quindi in questo caso l’abbiamo concessa. La funzione `multiprocessing.cpu_count()` restituisce il numero di core della macchina (per esempio 2 per una macchina con un processore dual-core, 8 per una con due processori quad-core).

Il modulo `argparse` adotta un approccio dichiarativo alla creazione di un parser della riga di comando. Una volta creato il parser, elaboriamo la riga di comando e acquisiamo i parametri. Effettuiamo alcuni controlli di sicurezza (per esempio per impedire all’utente di sovrascrivere le immagini originali con quelle ridimensionate) e creiamo la directory di destinazione nel caso non esista. La funzione `os.makedirs()` è simile alla funzione `.mkdir()` con la differenza che la prima può creare le directory intermedie e non solamente una singola sottodirectory.

Prima di passare al codice, prendete nota delle seguenti importanti regole, che valgono per qualsiasi file Python che utilizzi il modulo `multiprocessing`.

- Il file deve essere un modulo importabile. Per esempio, `my-mod.py` è un nome valido per un programma Python ma non per un modulo (dato che `import my-mod` è un errore di sintassi); `my_mod.py` o `MyMod.py` sono invece entrambi validi.
- Il file deve avere una funzione di ingresso (per esempio `main()`) e terminare con una chiamata al punto di ingresso. Per esempio: `if __>name__ == "main": main()`.
- Su Windows, il file Python e l'interprete Python (`python.exe` o `pythonw.exe`) devono trovarsi sulla stessa unità disco (per esempio `C:`).

Nei paragrafi che seguono esaminiamo le due versioni multiprocessing del programma per il ridimensionamento delle immagini, `imagescale-q-m.py` e `imagescale-m.py`. Entrambi i programmi mostrano l'avanzamento (cioè visualizzano il nome dell'immagine in corso di elaborazione) e consentono l'annullamento (per esempio, tramite la pressione della combinazione di tasti Ctrl+C).

Utilizzare le code e il multiprocessing

Il programma `imagescale-q-m.py` crea una coda di job da eseguire (per esempio, immagini da ridimensionare) e una coda di risultati.

```
Result = collections.namedtuple("Result", "copied scaled name")
Summary = collections.namedtuple("Summary", "todo copied scaled canceled")
```

La tupla `Result` viene utilizzata per memorizzare un singolo risultato. Si tratta del numero di immagini copiate e del numero di immagini ridimensionate (sempre 1 e 0 oppure 0 e 1) e del nome dell'immagine risultante. La tupla `Summary` è utilizzata per memorizzare il riepilogo di tutti i risultati.

```
def main():
    size, smooth, source, target, concurrency = handle_commandline()
    Qtrac.report("starting…")
    summary = scale(size, smooth, source, target, concurrency)
    summarize(summary, concurrency)
```

Questa funzione `main()` è la stessa per tutti i programmi di ridimensionamento delle immagini. Inizia con la lettura della riga di comando tramite la funzione personalizzata `handle_commandline()` che abbiamo discusso in precedenza. Restituisce le dimensioni cui le immagini vanno ridotte, un valore booleano che indica se debba essere utilizzato il ridimensionamento smooth, la directory di origine da cui leggere le immagini, la directory di destinazione in cui scrivere le immagini ridimensionate e

(per le versioni concorrenti) il numero di thread o di processi da utilizzare (che per default è pari al numero di core).

Il programma segnala all'utente di essersi avviato e poi esegue la funzione `scale()` che svolge tutto il lavoro. Quando la funzione `scale()` restituisce il riepilogo dei risultati, visualizziamo quest'ultimo utilizzando la funzione `summarize()`.

```
def report(message="", error=False):
    if len(message) >= 70 and not error:
        message = message[:67] + "..."
    sys.stdout.write("\r{:70}{}".format(message, "\n" if error else ""))
    sys.stdout.flush()
```

Per comodità, questa funzione si trova nel modulo `Qtrac.py`, dato che viene utilizzata da tutti gli esempi di programmi concorrenti da console di questo capitolo. La funzione sovrascrive la riga corrente della console con il messaggio dato (troncandolo a 70 caratteri se necessario) ed esegue il flush dell'output in modo che esso venga visualizzato immediatamente. Se il messaggio consiste nella segnalazione di un errore, viene stampato anche un carattere di ritorno a capo in modo che il messaggio di errore non venga sovrascritto dal messaggio successivo, e non viene effettuato alcun troncamento.

```
def scale(size, smooth, source, target, concurrency):
    canceled = False
    jobs = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()
    create_processes(size, smooth, jobs, results, concurrency)
    todo = add_jobs(source, target, jobs)
    try:
        jobs.join()
    except KeyboardInterrupt:   # Potrebbe non funzionare su Windows
        Qtrac.report("canceling…")
        canceled = True
    copied = scaled = 0
    while not results.empty():  # Sicuro perché tutti i job sono terminati
        result = results.get_nowait()
        copied += result.copied
        scaled += result.scaled
    return Summary(todo, copied, scaled, canceled)
```

Questa funzione è il cuore del programma concorrente di ridimensionamento delle immagini che fa ricorso al multiprocessing e alle code, e il suo compito è illustrato nella Figura 4.1. La funzione inizia creando una coda *joinable* di job da eseguire. Una coda joinable è una coda per la quale è possibile attendere (fino a quando non è vuota). Poi viene creata una coda non joinable di risultati. Successivamente, la funzione crea i processi per eseguire il lavoro: saranno pronti per essere utilizzati ma bloccati, dato che nella coda delle operazioni non è stato ancora inserito alcun job. Quindi, viene richiamata la funzione `add_jobs()` per riempire la coda di job.

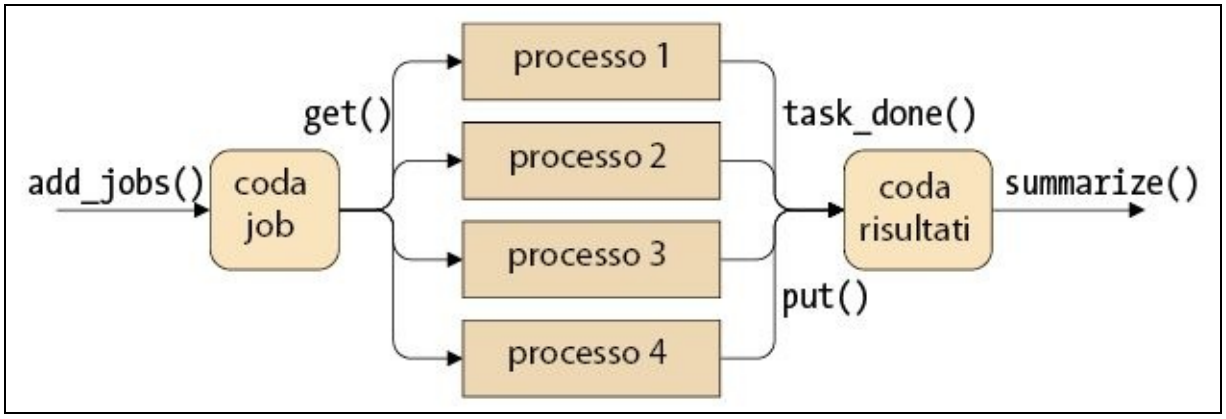


Figura 4.1 Gestire operazioni e risultati concorrenti con le code.

Una volta inseriti nella coda tutti i job, attendiamo che la coda si svuoti utilizzando il metodo `multiprocessing.JoinableQueue.join()`. Lo si utilizza all'interno di un blocco `try … except` in modo che, se l'utente annulla (per esempio premendo Ctrl+C su Unix), sia possibile gestire l'annullamento in modo adeguato.

Quando tutte le operazioni sono state eseguite (o il programma è stato interrotto), si effettua un'iterazione sulla coda dei risultati. Normalmente, l'utilizzo del metodo `empty()` su una coda concorrente non è affidabile, ma in questo caso funziona bene dato che tutti i processi che svolgono il lavoro sono terminati e la coda non viene più aggiornata. Per questo stesso motivo, per acquisire i risultati possiamo utilizzare anche il metodo `multiprocessing.Queue.get_nowait()` che non crea alcun bloccaggio, invece del consueto metodo `multiprocessing.Queue.get()`, che lo prevede.

Una volta raccolti tutti i risultati, restituiamo una tupla `Summary` contenente le informazioni dettagliate. Per una esecuzione normale, il valore `todo` sarà zero e il valore `canceled` sarà `False`; per un'esecuzione annullata, invece, probabilmente `todo` sarà diverso da zero e `canceled` sarà `True`.

Sebbene questa funzione sia denominata `scale()`, in realtà è una funzione piuttosto generica di "esecuzione di operazioni concorrenti" che fornisce job ai processi e raccoglie i risultati. Potrebbe facilmente essere adattata ad altre situazioni.

```
def create_processes(size, smooth, jobs, results, concurrency):
    for _ in range(concurrency):
        process = multiprocessing.Process(target=worker, args=(size,
                    smooth, jobs, results))
        process.daemon = True
        process.start()
```

Questa funzione crea processi multiprocessing per eseguire il lavoro. A ciascun processo viene data la stessa funzione `worker()` (dato che svolgono tutti lo stesso compito) e le informazioni dettagliate sul compito da svolgere, che comprendono la coda condivisa dei job e la coda condivisa dei risultati. Naturalmente non occorre

preoccuparsi di bloccare queste code condivise, dato che badano autonomamente alla loro sincronizzazione. Una volta creato un processo lo trasformiamo in demone: quando il processo principale termina, termina in modo corretto anche tutti i propri processi demone (mentre quelli non demone rimangono in esecuzione e, su Unix, diventano processi zombie).

Dopo avere creato ciascun processo e averlo trasformato in demone, gli indichiamo di iniziare l'esecuzione della funzione assegnatagli. Il processo si bloccherà immediatamente, dato che non abbiamo ancora inserito alcun job nella coda apposita. Ma questo non importa, dato che il lock ha luogo in un processo separato e non blocca il processo principale. Di conseguenza, tutti i processi multiprocessing vengono creati rapidamente e questa funzione si conclude. Poi, nella funzione chiamante, inseriamo dei job nella coda in modo che i processi bloccati possano lavorarvi.

```
def worker(size, smooth, jobs, results):
    while True:
        try:
            sourceImage, targetImage = jobs.get()
            try:
                result = scale_one(size, smooth, sourceImage, targetImage)
                Qtrac.report("{} {}".format("copied" if result.copied else
                        "scaled", os.path.basename(result.name)))
                results.put(result)
            except Image.Error as err: Qtrac.report(str(err), True)
        finally:
            jobs.task_done()
```

Per svolgere il lavoro concorrente è possibile creare una sottoclasse `multiprocessing.Process` (o una sottoclasse `threading.Thread`). In questo caso tuttavia abbiamo adottato un approccio lievemente più semplice e creato una funzione che viene passata come argomento `target` di `multiprocessing.Process` (è possibile fare esattamente la stessa cosa con `threading .Threads`).

Il worker esegue un ciclo infinito e in ciascuna iterazione tenta di ottenere un job da eseguire dalla coda di job condivisa. Utilizzare un ciclo infinito è sicuro, perché il processo è un demone e viene quindi terminato quando il programma termina. Il metodo `multiprocessing.Queue.get()` blocca fino a quando non è in grado di restituire un job, che in questo esempio è una tupla di due valori composta dal nome dell'immagine di origine e da quello dell'immagine di destinazione.

Una volta ricevuto un job, effettuiamo il ridimensionamento (o la copia) utilizzando la funzione `scale_one()` e riportiamo ciò che è stato fatto. Inseriamo inoltre l'oggetto `result` (di tipo `Result`) nella coda condivisa dei risultati.

Quando utilizziamo una coda joinable è essenziale che per ciascuna operazione ricevuta eseguiamo `multiprocessing.JoinableQueue.task_done()`. Ciò consente al metodo `multiprocessing.JoinableQueue.join()` di sapere quando la coda può essere sottoposta a join (cioè quando è vuota, senza più job da eseguire).

```
def add_jobs(source, target, jobs):
    for todo, name in enumerate(os.listdir(source), start=1):
        sourceImage = os.path.join(source, name)
        targetImage = os.path.join(target, name)
        jobs.put((sourceImage, targetImage))
    return todo
```

Dopo essere stati creati e avviati, tutti i processi rimangono bloccati in attesa di ricevere job dalla coda condivisa.

Per ciascuna immagine da elaborare, questa funzione crea due stringhe: `sourceImage`, che contiene il percorso completo di un'immagine di origine, e `targetImage`, con il percorso completo dell'immagine di destinazione. Ciascuna coppia di percorsi viene aggiunta come tupla di due valori alla coda di job condivisa. Alla fine, la funzione restituisce il numero totale di job da eseguire.

Non appena il primo job viene aggiunto alla coda di job, uno dei processi worker bloccati lo preleva e inizia a lavorare su di esso, e lo stesso avviene per il secondo job aggiunto, e per il terzo, fino a quando tutti i processi worker non hanno un job da svolgere. Poi, è probabile che la coda di job acquisisca ulteriori job mentre i processi worker sono all'opera; viene acquisito un nuovo job ogni volta che un processo worker termina quello che sta svolgendo.

```
def scale_one(size, smooth, sourceImage, targetImage):
    oldImage = Image.from_file(sourceImage)
    if oldImage.width <= size and oldImage.height <= size:
        oldImage.save(targetImage)
        return Result(1, 0, targetImage)
    else:
        if smooth:
            scale = min(size / oldImage.width, size / oldImage.height)
            newImage = oldImage.scale(scale)
        else:
            stride = int(math.ceil(max(oldImage.width / size,
                                       oldImage.height / size)))
            newImage = oldImage.subsample(stride)
        newImage.save(targetImage)
        return Result(0, 1, targetImage)
```

Questa funzione è quella in cui ha effettivamente luogo il ridimensionamento (o la copia). Utilizza il modulo `cyImage` (descritto nel Capitolo 5) oppure ripiega sul modulo `Image` (descritto nel Capitolo 3) se `cyImage` non è disponibile. Se l'immagine è già più piccola di quanto stabilito, viene semplicemente salvata nel percorso di destinazione e viene restituito un `Result` che indica che è stata copiata un'immagine, che non è stata ridimensionata alcuna immagine, e il nome dell'immagine di destinazione. Altrimenti, l'immagine viene ridimensionata oppure sottocampionata e l'immagine

risultante viene salvata. In questo caso, il `Result` che viene restituito indica che non è stata copiata alcuna immagine, che è stata ridimensionata un'immagine e, anche qui, il nome dell'immagine di destinazione.

```
def summarize(summary, concurrency):
    message = "copied {} scaled {} ".format(summary.copied, summary.scaled)
    difference = summary.todo - (summary.copied + summary.scaled)
    if difference:
        message += "skipped {} ".format(difference)
        message += "using {} processes".format(concurrency)
    if summary.canceled:
        message += " [canceled]" Qtrac.report(message)
    print()
```

Quando tutte le immagini sono state elaborate (cioè quando la coda di job è stata svuotata), l'oggetto `Summary` viene creato (nella funzione `scale()`) e passato a questa funzione. Tipicamente questa funzione produce un riepilogo simile al seguente, visualizzato sulla seconda riga:

```
$ ./imagescale-m.py -S /tmp/images /tmp/scaled
copied 0 scaled 56 using 4 processes
```

Su Linux, per rilevare i tempi è sufficiente anteporre `time` al comando. Su Windows non esiste un comando analogo, ma esistono altre soluzioni (si veda per esempio `http://stackoverflow.com/questions/673523/how-to-measure-execution-time-of-command-in-windows-command-line`). Tra l'altro, il rilevamento dei tempi all'interno dei programmi che utilizzano il multiprocessing non sembra funzionare. Nei nostri esperimenti abbiamo scoperto che il cronometraggio rilevava il tempo di esecuzione del processo principale ma escludeva quello dei processi worker. Notate che il modulo `time` di Python 3.3 ha diverse nuove funzioni che supportano il cronometraggio accurato.

La differenza di tre secondi tra `imagescale-q-m.py` e `imagescale- m.py` non è significativa e può capitare facilmente che si inverta tra un'esecuzione e l'altra. Quindi, in realtà, queste due versioni sono equivalenti.

Utilizzare i future e il multiprocessing

Python 3.2 ha introdotto il modulo `concurrent.futures`, che costituisce un mezzo di alto livello per realizzare la concorrenza in Python utilizzando più thread e più processi. In questo sottoparagrafo rivedremo tre funzioni del programma `imagescale-m.py` (tutto il resto è come nel programma `imagescale-q-m.py` che abbiamo visto nel sottoparagrafo precedente). Il programma `imagescale-m.py` utilizza i *future*. Stando alla documentazione, un `concurrent.futures.Future` è un oggetto che "incapsula l'esecuzione asincrona di un callable" (cfr.

http://docs.python.org/dev/library/concurrent.futures.html#future-objects). I future si creano richiamando il metodo `concurrent.futures.Executor.submit()` e possono segnalare il loro stato (interrotto, in esecuzione, completato) e il risultato o l'eccezione che hanno prodotto.

La classe `concurrent.futures.Executor` non può essere utilizzata direttamente, perché è una classe base astratta. Deve essere invece utilizzata una delle sue due sottoclassi concrete. `concurrent.futures.ProcessPoolExecutor()` produce una concorrenza basata sull'utilizzo di più processi. L'uso di un pool di processi significa che ogni `Future` utilizzato con esso può eseguire o restituire solo oggetti pickleable, tra cui le funzioni non annidate, ovviamente. Questa restrizione non vale per `concurrent.futures.ThreadPoolExecutor`, che realizza la concorrenza utilizzando più thread.

Concettualmente, utilizzare un pool di thread o di processi è più semplice che utilizzare le code, come illustrato nella Figura 4.2.

Figura 4.2 Gestione di job concorrenti e risultati con un esecutore a pool.

```
def scale(size, smooth, source, target, concurrency):
    futures = set()
    with concurrent.futures.ProcessPoolExecutor(
            max_workers=concurrency) as executor:
        for sourceImage, targetImage in get_jobs(source, target):
            future = executor.submit(scale_one, size, smooth, sourceImage,
                    targetImage)
            futures.add(future)
        summary = wait_for(futures)
        if summary.canceled:
            executor.shutdown()
        return summary
```

Questa funzione ha la stessa firma e svolge lo stesso compito della funzione analoga del programma `imagescale-q-m.py`, ma funziona in modo radicalmente differente. Iniziamo creando un insieme vuoto di future. Poi creiamo un esecutore per il pool di processi. Dietro le quinte, esso creerà una serie di processi worker. Il loro numero viene determinato euristicamente, ma in questo caso abbiamo ridefinito questo meccanismo in modo da specificare direttamente il numero, puramente per comodità di debugging e rilevamento dei tempi.

Ora che abbiamo un esecutore per il pool di processi, eseguiamo un'iterazione sulle operazioni restituite dalla funzione `get_jobs()` e le inviamo al pool. Il metodo `concurrent. futures.ProcessPoolExecutor.submit()` accetta una funzione worker e altri argomenti opzionali e restituisce un oggetto `Future`. Aggiungiamo ciascun future al

nostro set di future. Il pool inizia a lavorare non appena ha almeno un future su cui operare. Una volta che tutti i future sono stati creati, richiamiamo una funzione `wait_for()` personalizzata e passiamo a essa l'insieme di future. Questa funzione rimarrà bloccata fino a quando non saranno stati ultimati tutti i future (o fino all'annullamento da parte dell'utente). Se l'utente annulla, fermiamo manualmente l'esecutore del pool di processi.

```
def get_jobs(source, target):
    for name in os.listdir(source):
        yield os.path.join(source, name), os.path.join(target, name)
```

Questa funzione svolge lo stesso servizio della funzione `add_jobs()` del paragrafo precedente, ma non inserisce i job in una coda; è un generatore che fornisce job su richiesta.

```
def wait_for(futures):
    canceled = False
    copied = scaled = 0
    try:
        for future in concurrent.futures.as_completed(futures):
                err = future.exception()
                if err is None:
                    result = future.result()
                    copied += result.copied
                    scaled += result.scaled
                    Qtrac.report("{} {}".format("copied" if result.copied else
                            "scaled", os.path.basename(result.name)))
                elif isinstance(err, Image.Error):
                    Qtrac.report(str(err), True)
                else:
                    raise err  # Non previsto
    except KeyboardInterrupt:
        Qtrac.report("canceling…")
        canceled = True
        for future in futures:
            future.cancel()
    return Summary(len(futures), copied, scaled, canceled)
```

Una volta creati tutti i future, richiamiamo questa funzione per attendere il loro completamento. La funzione `concurrent.futures.as_completed()` attua un bloccaggio fino a quando un future non termina (o non viene annullato) e poi restituisce il future stesso. Se il worker callable eseguito dal future ha generato un'eccezione, questa viene restituita dal metodo `Future.exception()`; altrimenti, il metodo restituisce `None`. Se non si è verificata alcuna eccezione, acquisiamo il risultato del future e segnaliamo l'avanzamento all'utente. Se si è verificata un'eccezione di un tipo che possiamo ragionevolmente aspettarci (prodotta cioè dal modulo `Image`), la segnaliamo all'utente. Se invece abbiamo un'eccezione inattesa, la solleviamo, dato che può significare che il programma contiene un errore logico oppure che è stato interrotto dall'utente con Ctrl+C.

Se l'utente interrompe il programma premendo Ctrl+C, effettuiamo un'iterazione su tutti i future per interromperli. Alla fine, restituiamo un riepilogo del lavoro svolto.

L’uso di `concurrent.futures` è un metodo più pulito e solido rispetto all’utilizzo delle code, anche se entrambi gli approcci sono molto più semplici e migliori rispetto a uno che implichi l’utilizzo di lock espliciti, quando si usa il multithreading. È anche semplice passare dal multithreading al multiprocessing: è sufficiente utilizzare un `concurrent. futures.ThreadPoolExecutor` invece di un `concurrent.futures.ProcessPoolExecutor`. Quando utilizziamo il multithreading, di qualsiasi tipo esso sia, se abbiamo la necessità di accedere ai dati condivisi dobbiamo utilizzare dei tipi non modificabili o una copia profonda (per esempio, per l’accesso in sola lettura), oppure utilizzare i lock (per esempio per serializzare gli accessi in lettura e scrittura) o ancora ricorrere a un tipo thread-safe (per esempio un `queue.Queue`).

Analogamente, quando utilizziamo il multiprocessing, per accedere ai dati condivisi dobbiamo utilizzare dei tipi non modificabili o la copia profonda, e per l’accesso in lettura e scrittura occorre utilizzare dei `multiprocessing.Value` o `multiprocessing.Array` gestiti, oppure utilizzare delle `multiprocessing.Queue`. Idealmente, si dovrebbe cercare di evitare del tutto l’utilizzo di dati condivisi. Se ciò non è possibile, è bene condividere i dati solo in lettura (per esempio utilizzando tipi non modificabili o la copia profonda) oppure utilizzare code compatibili con la concorrenza, in modo che non siano necessari lock espliciti, e il nostro codice risulti più semplice in fase di comprensione e di manutenzione.

Concorrenza I/O-bound

Un’esigenza comune è quella di prelevare una serie di file o di pagine web da Internet. A causa della latenza della rete, di solito è possibile effettuare più download in contemporanea e concludere l’operazione in un tempo molto minore di quello necessario per scaricare un file alla volta.

In questo paragrafo esamineremo i programmi `whatsnew-q.py` e `whatsnew-t.py`, che programmi scaricano *feed RSS*, piccoli documenti XML contenenti riassunti di notizie riguardanti la tecnologia. I feed provengono da vari siti web e il programma li utilizza per creare una pagina HTML con i collegamenti alle diverse notizie. La Figura 4.3 mostra parte di una delle pagine HTML di “novità” così generate. La Tabella 4.2 mostra i tempi rilevati per varie versioni del programma (i tempi sono stati rilevati su una macchina AMD64 3 GHz quad-core con poco carico durante il download da quasi 200 siti web su una connessione di tipo domestico).

Gli incrementi di velocità dei programmi “novità” appaiono proporzionali al numero dei core, ma si tratta solo di una coincidenza; i core erano tutti sottoutilizzati e la maggior parte del tempo veniva speso in attesa dell’I/O di rete.

La tabella mostra anche i tempi rilevati per le versioni di un programma `gigapixel` (non presente nel libro). Questi programmi accedono al sito web `http://www.gigapan.org` e prelevano quasi 500 file in formato JSON, per un totale di 1,9 MiB, contenenti metadati relativi a immagini gigapixel. Il codice delle versioni di questo programma rispecchia quello dei programmi “novità”, anche se i programmi `gigapixel` fanno registrare incrementi di velocità molto maggiori.

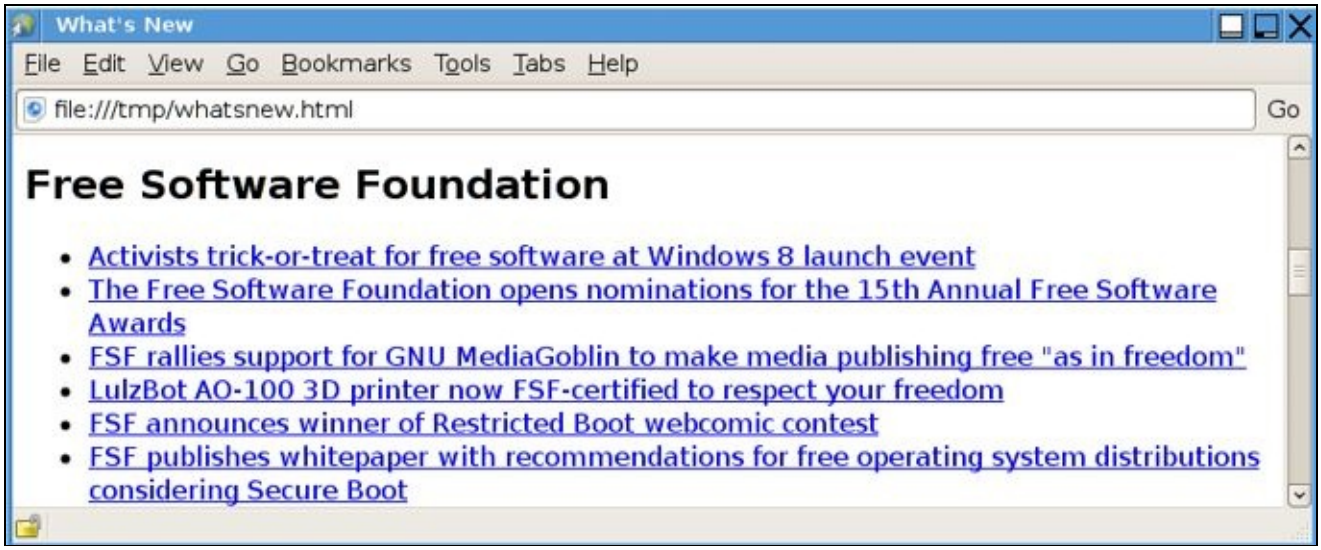


Figura 4.3 Alcuni link a notizie tecnologiche tratti da un feed RSS.

Tabella 4.2 Confronto tra le velocità di trasferimento.

Programma	Concorrenza	Secondi	Incremento velocità
`whatsnew.py`	*Nessuna*	172	*Riferimento*
`whatsnew-c.py`	16 coroutine	180	0,96×
`whatsnew-q-m.py`	16 processi con una coda	45	3,82×
`whatsnew-m.py`	16 processi con un pool di processi	50	3,44×
`whatsnew-q.py`	16 thread con una coda	50	3,44×
`whatsnew-t.py`	16 thread con un pool di thread	48	3,58×
`gigapixel.py`	*Nessuna*	238	*Riferimento*
`gigapixel-q-m.py`	16 processi con una coda	35	6,80×
`gigapixel-m.py`	16 processi con un pool di processi	42	5,67×
`gigapixel-q.py`	16 thread con una coda	37	6,43×
`gigapixel-t.py`	16 thread con un pool di thread	37	6,43×

Le migliori prestazioni sono dovute al fatto che i programi `gigapixel` accedono a un unico sito che dispone di una banda molto ampia, mentre i programmi “novità” devono accedere a molti siti differenti, con ampiezze di banda differenti.

Data questa variabilità della latenza della rete, gli incrementi di velocità possono facilmente variare; le versioni concorrenti possono mostrare miglioramenti da 2× fino a 10× o anche maggiori, a seconda dei siti cui accedono, della quantità di dati trasferiti e dell’ampiezza di banda della connessione di rete. In considerazione di ciò, le differenze tra la versioni multiprocessing e quelle multithreading non sono significative e possono facilmente capovolgersi tra un’esecuzione e l’altra.

Il punto fondamentale da ricordare, in riferimento alla Tabella 4.2 è che otterremo un download decisamente più rapido utilizzando un approccio concorrente, anche se l’incremento di velocità effettivo è variabile tra un’esecuzione e l’altra ed è influenzato anche dalle circostanze.

Utilizzare le code e il threading

Inizieremo esaminando il programma `whatsnew-q.py`, che utilizza più thread e due code thread-safe. Una delle due è una coda di job; in essa, ciascun job da svolgere è un URL. L’altra coda è la coda dei risultati, e ciascun risultato è una tupla di due valori contenente il valore `True` e un frammento di codice HTML da inserire nella pagina HTML in costruzione, oppure il valore `False` e un messaggio di errore.

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting…")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    jobs = queue.Queue()
    results = queue.Queue()
    create_threads(limit, jobs, results, concurrency)
```

```
    todo = add_jobs(filename, jobs)
    process(todo, jobs, results, concurrency)
```

La funzione `main()` dirige il lavoro. In avvio elabora la riga di comando, da cui ricava il valore limite (il numero massimo di nuovi elementi da leggere da un dato URL) e il livello di concorrenza per il debugging e il cronometraggio. Il programma segnala quindi all'utente di essersi avviato e acquisisce il nome del file con il percorso completo del file di dati che contiene gli URL e i rispettivi titoli.

Successivamente la funzione crea le due code compatibili con i thread e i thread worker. Dopo che tutti i thread worker sono stati avviati (i thread rimangono ovviamente inattivi dato che per il momento non c'è alcun lavoro da svolgere), inseriamo tutti i job da svolgere nella coda apposita. Infine, nella funzione `process()`, attendiamo che i job vengano svolti e poi mostriamo i risultati. La struttura di concorrenza del programma è illustrata nel suo complesso nella Figura 4.4.

Tra l'altro, se abbiamo molti job da inserire, o se l'inserimento dei job richiede molto tempo, potrebbe essere preferibile inserirli in un thread separato (o in un un processo separato, se stiamo utilizzando il multiprocessing).

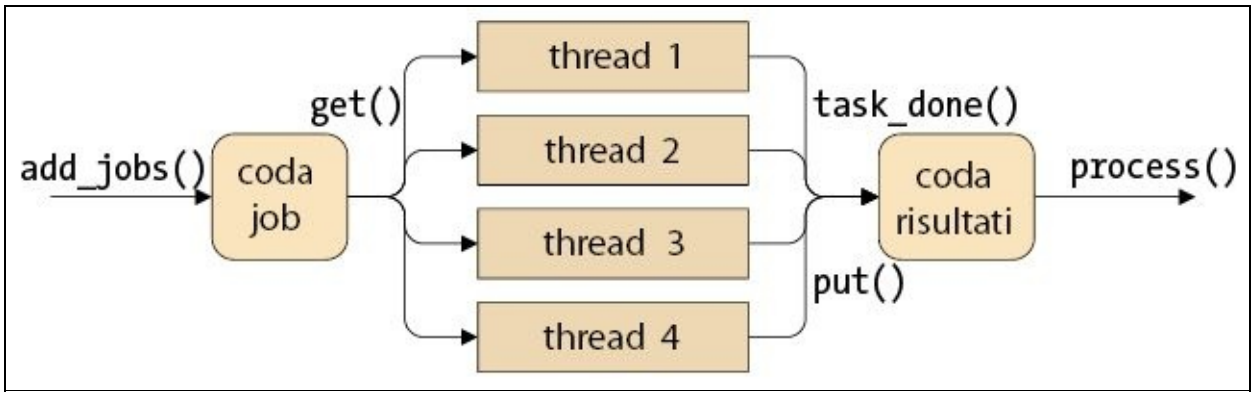


Figura 4.4 Gestione di operazioni e risultati concorrenti con le code.

```
def handle_commandline():
    parser = argparse.ArgumentParser()
    parser.add_argument("-l", "--limit", type=int, default=0,
            help="the maximum items per feed [default: unlimited]")
    parser.add_argument("-c", "--concurrency", type=int,
            default=multiprocessing.cpu_count() * 4,
            help="specify the concurrency (for debugging and "
                    "timing) [default: %(default)d]")
    args = parser.parse_args()
    return args.limit, args.concurrency
```

Dato che i programmi "novità" sono I/O-bound, diamo loro un livello di concorrenza di default multiplo del numero di core; in questo caso, 4×.

NOTA

Questo multiplo è stato scelto perché è quello che ha funzionato meglio nei nostri test. Vi consigliamo di fare delle prove, dato che le configurazioni sono diverse.

```
def create_threads(limit, jobs, results, concurrency):
    for _ in range(concurrency):
        thread = threading.Thread(target=worker, args=(limit, jobs, results))
        thread.daemon = True
        thread.start()
```

Questa funzione crea i thread worker nella quantità indicata dalla variabile `concurrency` e assegna a ciascuno di essi una funzione worker da eseguire e i parametri con cui richiamare la funzione.

Come abbiamo fatto nel paragrafo precedente con i processi, trasformiamo tutti i thread in demoni per assicurarci che vengano terminati alla fine del programma. Quando avviamo un thread esso immediatamente si blocca perché non vi sono operazioni da svolgere; rimangono però bloccati solamente i thread worker, non il thread principale del programma.

```
def worker(limit, jobs, results):
    while True:
        try:
            feed = jobs.get()
            ok, result = Feed.read(feed, limit)
            if not ok:
                Qtrac.report(result, True)
            elif result is not None:
                Qtrac.report("read {}".format(result[0][4:-6]))
                results.put(result)
        finally:
            jobs.task_done()
```

Abbiamo fatto in modo che la funzione `worker()` esegua un ciclo infinito, dato che si tratta di un demone e che quindi verrà estinto dal programma quando il programma stesso terminerà la propria esecuzione.

La funzione si blocca in attesa di ricevere un’operazione dalla coda delle operazioni. Non appena riceve un’operazione, utilizza la funzione `Feed.read()` del modulo personalizzato `Feed.py` per leggere il file indicato dall’URL. Tutti i programmi “novità” ricorrono a un modulo `Feed.py` personalizzato, che fornisce un iteratore per il file delle operazioni e un lettore per ciascun feed RSS. Se la lettura fallisce, `ok` è `False` e visualizziamo il risultato (costituito da un messaggio di errore). Altrimenti, se abbiamo un risultato (una lista di stringhe HTML), visualizziamo il primo elemento (scartando i tag HTML) e inseriamo il risultato stesso nella coda dei risultati.

Per le code che prevediamo di sottoporre a join, è essenziale che per ogni chiamata di `queue.Queue.get()` abbiamo una corrispondente chiamata di `queue.Queue.task_done()`. Ci assicuriamo che questo avvenga utilizzando un blocco `try` … `finally`. Notate che, sebbene la classe `queue.Queue` sia una coda joinable thread-safe, l’equivalente per il multiprocessing è la classe `multiprocessing.JoinableQueue`, non la classe `multiprocessing.Queue`.

```
def read(feed, limit, timeout=10):
    try:
        with urllib.request.urlopen(feed.url, None, timeout) as file:
            data = file.read()
        body = _parse(data, limit)
        if body:
            body = ["<h2>{}</h2>\n".format(escape(feed.title))] + body
            return True, body
        return True, None
    except (ValueError, urllib.error.HTTPError, urllib.error.URLError,
            etree.ParseError, socket.timeout) as err:
        return False, "Error: {}: {}".format(feed.url, err)
```

La funzione `Feed.read()` legge un dato URL (`feed`) e tenta di elaborarlo. Se l'elaborazione ha successo, la funzione restituisce `True` e una lista di frammenti HTML (un titolo e uno o più link); altrimenti restituisce `False` e `None` oppure un messaggio di errore.

```
def _parse(data, limit):
    output = []
    feed = feedparser.parse(data)  # Atom + RSS
        for entry in feed["entries"]:
            title = entry.get("title")
            link = entry.get("link")
            if title:
                if link:
                    output.append('<li><a href="{}">{}</a></li>'.format(
                        link, escape(title)))
                else:
                    output.append('<li>{}</li>'.format(escape(title)))
        if limit and len(output) == limit:
            break
    if output:
        return ["<ul>"] + output + ["</ul>"]
```

Il modulo `Feed.py` contiene due versioni della funzione privata `_parse()`. Quella mostrata qui utilizza il modulo esterno `feedparser` (`http://pypi.python.org/pypi/feedparser`), in grado di gestire news feed sia in formato Atom, sia in formato RSS. L'altra (non mostrata qui) è un ripiego utilizzato quando `feedparser` non è disponibile e può gestire solamente feed in formato RSS.

La funzione `feedparser.parse()` svolge tutto il lavoro di elaborazione del news feed. Non dobbiamo fare altro che iterare sugli elementi che essa produce ed estrarre il titolo e il link per ciascuna notizia, costruendo una lista HTML per rappresentarli.

```
def add_jobs(filename, jobs):
    for todo, feed in enumerate(Feed.iter(filename), start=1):
        jobs.put(feed)
    return todo
```

I singoli feed vengono restituiti dalla funzione `Feed.iter()` sotto forma di tuple di due valori (titolo, url), che vengono inserite nella coda delle operazioni. Al termine, viene restituito il numero totale delle operazioni da svolgere.

In questo caso avremmo potuto tranquillamente restituire `jobs.qsize()` invece di occuparci di conteggiare i vari job. Se tuttavia eseguissimo `add_jobs()` in un thread a

parte, `queue.Queue.qsize()` sarebbe inaffidabile, dato che i job verrebbero tolti dalla coda immediatamente dopo esservi state inseriti.

```
Feed = collections.namedtuple("Feed", "title url")

def iter(filename):
    name = None
    with open(filename, "rt", encoding="utf-8") as file:
        for line in file:
            line = line.rstrip()
            if not line or line.startswith("#"):
                continue
            if name is None:
                name = line
            else:
                yield Feed(name, line)
                name = None
```

Questa è la funzione `Feed.iter()` del modulo `Feed.py`. Ci aspettiamo che il file `whatsnew.dat` sia un file di testo semplice con codifica UTF-8 contenente due righe per ogni feed: il titolo (per esempio, `The Guardian - Technology`) e, sulla riga successiva, l'URL (per esempio, `http://feeds.pinboard.in/rss/u:guardiantech/`). Le righe vuote e le righe di commento (cioè quelle che iniziano con il carattere `#`) vengono ignorate.

```
def process(todo, jobs, results, concurrency):
    canceled = False
    try:
        jobs.join()  # Attende che tutto il lavoro sia svolto
    except KeyboardInterrupt:  # Potrebbe non funzionare su Windows
        Qtrac.report("canceling…")
        canceled = True
    if canceled:
        done = results.qsize()
    else:
        done, filename = output(results)
    Qtrac.report("read {}/{} feeds using {} threads{}".format(done, todo,
            concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)
```

Una volta creati tutti i thread e inseriti tutti i job, viene richiamata questa funzione, la quale a sua volta richiama `queue.Queue.join()`, che rimane in attesa fino a quando la coda non è vuota (cioè fino a quando tutti i job non sono stati effettuati) o fino all'interruzione da parte dell'utente. Se l'utente non interrompe il programma, viene richiamata la funzione `output()` per scrivere il file HTML con tutte le liste di link, quindi viene visualizzato un riepilogo. Infine, viene richiamata la funzione `open()` del modulo `webbrowser` per aprire il file HTML nel browser web predefinito dell'utente (Figura 4.3).

```
def output(results):
    done = 0
    filename = os.path.join(tempfile.gettempdir(), "whatsnew.html")
    with open(filename, "wt", encoding="utf-8") as file:
        file.write("<!doctype html>\n")
        file.write("<html><head><title>What's New</title></head>\n")
        file.write("<body><h1>What's New</h1>\n")
        while not results.empty():  # Sicuro perché tutti i job sono stati terminati
```

```
            result = results.get_nowait()
            done += 1
            for item in result:
                file.write(item)
        file.write("</body></html>\n")
    return done, filename
```

Una volta completati tutti i job, viene richiamata questa funzione con la coda dei risultati. Ciascun risultato contiene una lista di frammenti HTML (un titolo seguito da uno o più link). Questa funzione crea un nuovo file `whatsnew.html` e vi inserisce tutti i titoli e i collegamenti dei news feed. Al termine, la funzione restituisce il numero dei risultati (cioè il conteggio dei job eseguiti con successo) e il nome del file HTML scritto. Questa informazione viene utilizzata dalla funzione `process()` per stampare il riepilogo e per aprire il file HTML nel browser web dell'utente.

Utilizzare i future e il threading

Se utilizziamo Python 3.2 o una versione successiva, possiamo sfruttare il modulo `concurrent.futures` per implementare questo programma senza dover ricorrere alle code (né a lock espliciti). In questo sottoparagrafo esaminiamo il programma `whatsnew-t.py`, che fa uso di questo modulo, ma ometteremo le funzioni che sono identiche a quelle già viste nel precedente sottoparagrafo (per esempio `handle_commandline()` e le funzioni del modulo `Feed.py`).

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting…")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    futures = set()
    with concurrent.futures.ThreadPoolExecutor(
            max_workers=concurrency) as executor:
        for feed in Feed.iter(filename):
            future = executor.submit(Feed.read, feed, limit)
            futures.add(future)
        done, filename, canceled = process(futures)
        if canceled:
            executor.shutdown()
    Qtrac.report("read {}/{} feeds using {} threads{}".format(done,
            len(futures), concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)
```

Questa funzione crea un insieme di future inizialmente vuoto e poi crea un esecutore per il pool di thread, che funziona come un esecutore di un pool di processi ma utilizzando thread separati invece di processi separati. All'interno del contesto dell'esecutore, iteriamo sul file di dati e, per ciascun feed, creiamo un nuovo future (utilizzando il metodo `concurrent.futures.ThreadPoolExecutor.submit()`) che eseguirà la funzione `Feed.read()` sull'URL `feed` dato, restituendo un numero di link pari al massimo a `limit`. Infine inseriamo il future nell'insieme di future.

Una volta creati tutti i future, chiamiamo una funzione `process()` personalizzata che attende fino a quando tutti i future non si sono conclusi (o fino all'interruzione da parte dell'utente). Poi viene visualizzato un riepilogo dei risultati e, se il programma non è stato interrotto dall'utente, la pagina HTML generata viene aperta nel browser web dell'utente.

```
def process(futures):
    canceled = False
    done = 0
    filename = os.path.join(tempfile.gettempdir(), "whatsnew.html")
    with open(filename, "wt", encoding="utf-8") as file:
        file.write("<!doctype html>\n")
        file.write("<html><head><title>What's New</title></head>\n")
        file.write("<body><h1>What's New</h1>\n")
        canceled, results = wait_for(futures)
        if not canceled:
            for result in (result for ok, result in results if ok and
                    result is not None):
                done += 1
                for item in result:
                    file.write(item)
        else:
            done = sum(1 for ok, result in results if ok and result is not None)
        file.write("</body></html>\n")
    return done, filename, canceled
```

Questa funzione scrive l'inizio del file HTML e poi richiama una funzione personalizzata `wait_for()` per attendere che tutto il lavoro venga completato. Se l'utente non interrompe il programma, la funzione itera sui risultati (che sono tuple di due valori: `True`, `list` o `False`, `str` o `False`, `None`) e quando un risultato contiene una lista (costituita da un titolo seguito da uno o più link) i suoi elementi vengono scritti nel file HTML.

Se l'utente ha interrotto il programma, ci limitiamo a calcolare il numero dei feed letti con successo. In entrambi i casi, restituiamo il numero dei feed letti, il nome del file HTML e un valore che indica se il programma sia stato interrotto dall'utente.

```
def wait_for(futures):
    canceled = False
    results = []
    try:
        for future in concurrent.futures.as_completed(futures):
            err = future.exception()
            if err is None:
                ok, result = future.result()
                if not ok:
                    Qtrac.report(result, True)
                elif result is not None:
                    Qtrac.report("read {}".format(result[0][4:-6]))
                results.append((ok, result))
            else:
                raise err  # Non previsto
    except KeyboardInterrupt:
        Qtrac.report("canceling…")
        canceled = True
        for future in futures:
            future.cancel()
    return canceled, results
```

Questa funzione esegue un’iterazione sui future, rimanendo in attesa fino a quando un future non viene completato o annullato. Quando riceve un future, la funzione segnala un errore oppure una lettura effettuata con successo e in entrambi i casi aggiunge il valore booleano e il risultato (una lista di stringhe e una stringa di errore) alla lista dei risultati.

Se l’utente interrompe l’esecuzione (premendo Ctrl+C), annulliamo tutti i future. Al termine, restituiamo un valore che indica se il programma sia stato interrotto dall’utente e la lista dei risultati.

Utilizzare `concurrent.futures` è opportuno sia quando si utilizzano più thread sia quando si utilizzano più processi. In termini di prestazioni, poi, è chiaro che il multithreading, se utilizzato nelle giuste circostanze (elaborazione centrata sull’I/O piuttosto che sulla CPU) e con la dovuta attenzione, produce i miglioramenti delle prestazioni che ci potremmo attendere.

Caso di studio: un’applicazione GUI concorrente

Scrivere applicazioni GUI (*Graphical User Interface*) concorrenti può essere complesso, soprattutto se si utilizza Tkinter, lo strumento standard di Python per la creazione di applicazioni GUI. Una breve introduzione alla programmazione di applicazioni GUI con Tkinter viene proposta nel Capitolo 7; se non avete esperienza con Tkinter vi consigliamo di leggere prima questo capitolo e poi tornare qui.

Un approccio ovvio per ottenere un applicazione GUI concorrente è quello di utilizzare funzionalità multithreading, ma dal punto di vista pratico ciò può risultare in un’applicazione GUI lenta o anche bloccata quando sono attivi diversi processi; del resto le applicazioni GUI sono CPU-bound. Un approccio alternativo prevede l’utilizzo di funzionalità multiprocessing, ma anche in questo caso il risultato può essere una reattività molto scarsa.

In questo paragrafo esaminiamo l’applicazione ImageScale (presente nella directory `imagescale` degli esempi). L’applicazione è illustrata nella Figura 4.5 e presenta un approccio sofisticato che integra elaborazione concorrente con un’interfaccia GUI reattiva che registra il progresso delle operazioni e supporta gli annullamenti.

Come illustrato nella Figura 4.6, l’applicazione combina programmazione multithreading e multiprocessing. Ci sono due thread di esecuzione, il thread GUI principale e un thread worker, e quest’ultimo trasferisce il suo lavoro a un insieme di processi. Questa architettura produce un’interfaccia GUI che è sempre reattiva, poiché la maggior parte del tempo di elaborazione per il core viene condiviso dai due thread, e il thread worker (che non fa quasi nulla da solo) si occupa del resto. I processi del thread worker completano l’esecuzione con i propri core (su una macchina multicore), così da non entrare assolutamente in conflitto con l’interfaccia GUI.

Un programma da console analogo, `imagescale-m.py`, è composto da 130 righe di codice (ce ne siamo occupati in precedenza nel Capitolo 4). A titolo di confronto, l’applicazione GUI ImageScale è distribuita in 5 file (Figura 4.7), per un totale di quasi 500 righe di codice. Il codice per il ridimensionamento dell’immagine è di sole 60 righe circa; la gran parte del resto è costituita dal codice GUI.

Nei paragrafi seguenti esamineremo il codice più significativo per la programmazione GUI concorrente e alcune altre parti per avere un quadro più

completo e di più facile comprensione.

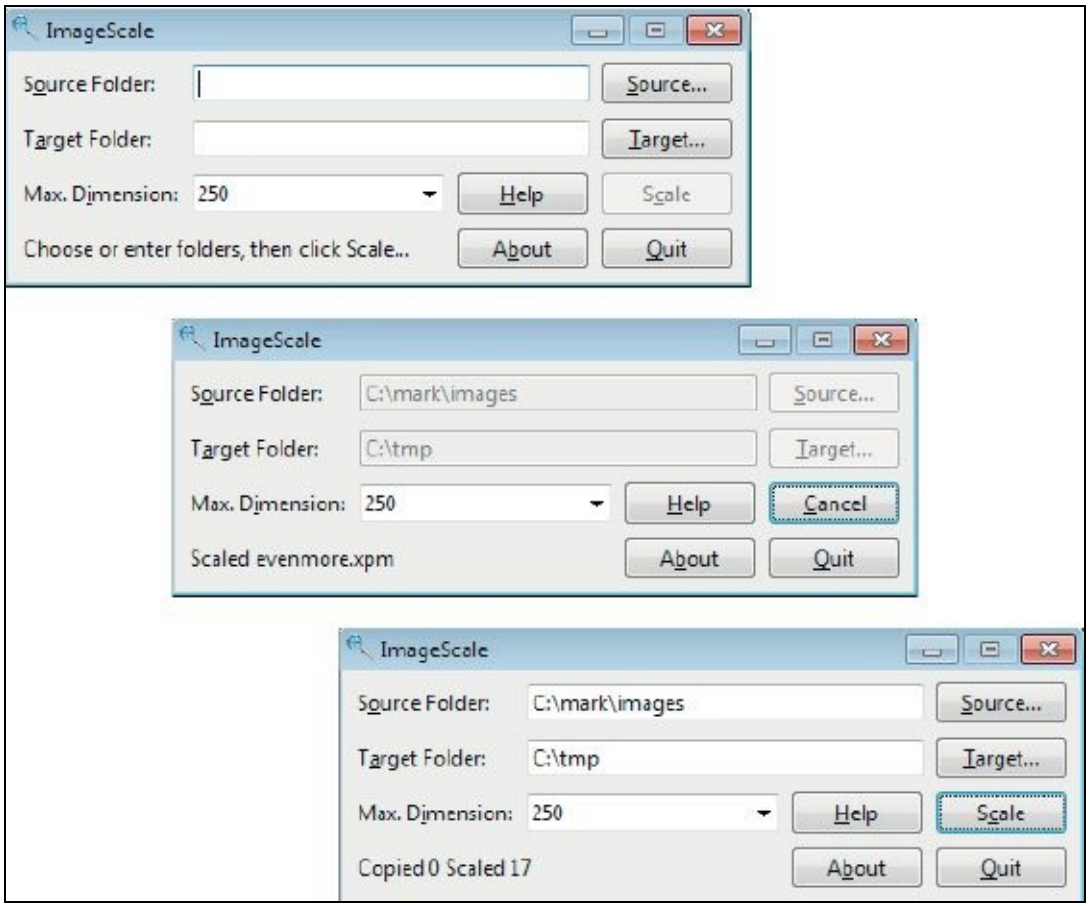


Figura 4.5 L'applicazione ImageScale prima, durante e dopo il ridimensionamento di alcune immagini.

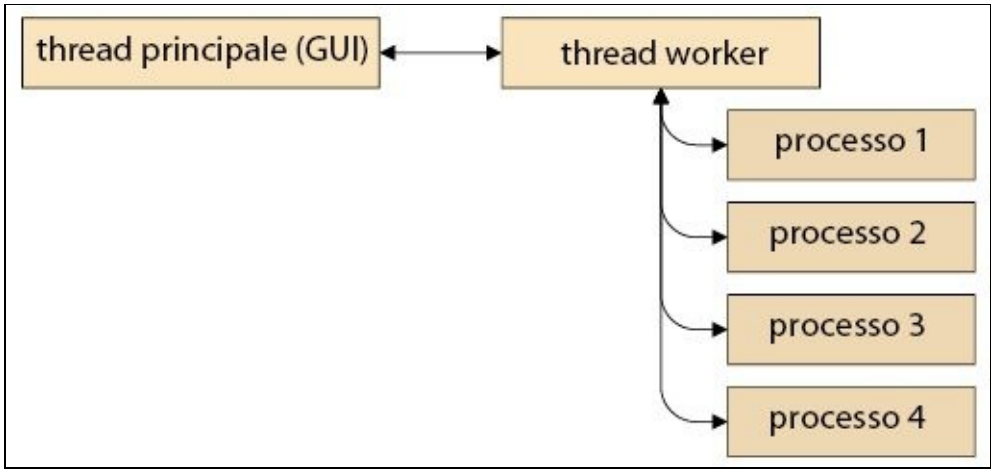


Figura 4.6 Il modello concorrente dell'applicazione ImageScale (le frecce indicano la comunicazione).

Creazione dell'applicazione GUI

In questo paragrafo esaminiamo il codice più importante per la creazione della GUI e per il supporto della concorrenza, prendendo materiale dai file `imagescale/imagescale.pyw` e `imagescale/Main.py`.

```
import tkinter as tk
import tkinter.ttk as ttk
import tkinter.filedialog as filedialog
import tkinter.messagebox as messagebox
```

Queste sono le importazioni relative alla GUI nel modulo `Main.py`. Alcuni utenti di Tkinter importano utilizzando `from tkinter import *`, ma qui preferiamo che il materiale importato mantenga i nomi GUI nel loro namespace e al tempo stesso che questi siano agevoli da utilizzare, perciò utilizziamo `tk` al posto di `tkinter`.

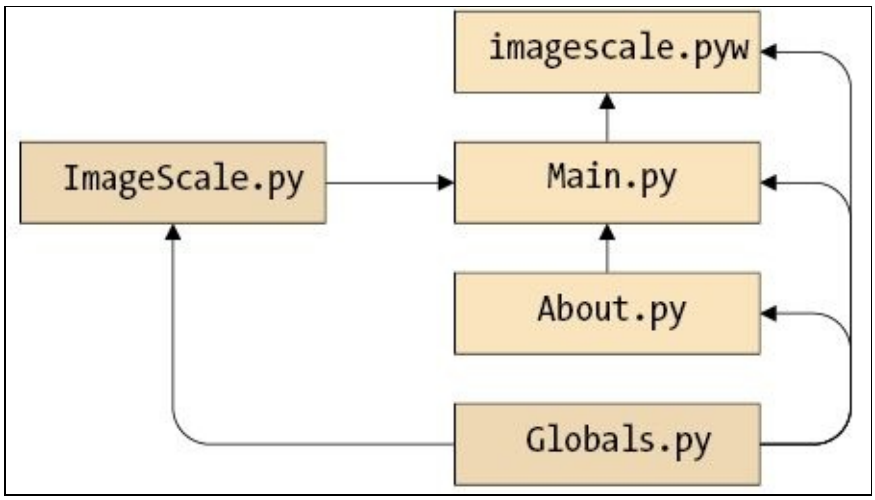


Figura 4.7 I file dell'applicazione ImageScale nel loro contesto (le frecce indicano le importazioni).

```
def main():
    application = tk.Tk()
    application.withdraw()  # non si vede finché è pronta alla visualizzazione
    window = Main.Window(application)
    application.protocol("WM_DELETE_WINDOW", window.close)
    application.deiconify()  # visualizza
    application.mainloop()
```

Questo è il punto d'ingresso dell'applicazione nel file `imagescale.pyw`. La funzione reale presenta anche del codice aggiuntivo che non è mostrato qui, relativo alle preferenze dell'utente e all'impostazione dell'icona dell'applicazione.

I punti chiave importanti sono innanzitutto che dobbiamo sempre creare un oggetto di primo livello `tkinter.Tk`, normalmente invisibile (il contenitore ultimo). Poi che dobbiamo creare un'istanza di finestra (in questo caso una sottoclasse personalizzata `tkinter.ttk.Frame`), e infine, dobbiamo avviare il ciclo di eventi di Tkinter.

Per evitare fenomeni di sfarfallamento o la comparsa di una finestra incompleta, nascondiamo l'applicazione non appena creata (in modo che l'utente non possa vederla a questo punto), e soltanto quando la finestra è stata completata la rendiamo visualizzabile.

La chiamata del protocollo `tkinter.Tk.protocol()` viene utilizzata per comunicare a Tkinter che se l'utente fa clic sul pulsante di chiusura della finestra ×, deve essere

richiamato il metodo personalizzato `Main.Window.close()` (in OS X, il pulsante di chiusura è solitamente un cerchio rosso, con un puntino nero in mezzo se non sono state salvate modifiche). Questo metodo viene esaminato più avanti in questo capitolo.

I programmi GUI hanno una struttura di elaborazione analoga ad alcuni programmi server; una volta avviati, aspettano semplicemente che gli eventi accadano, e quindi rispondono a questi. In un server, gli eventi possono essere connessioni e comunicazioni di rete, ma in un'applicazione GUI, gli eventi possono essere generati dall'utente (come accade con la pressione di un tasto o facendo clic con il mouse) o dal sistema (come lo scadere di un timer o un messaggio che riporta che la finestra è stata visualizzata; per esempio, dopo che la finestra di un'altra applicazione che era visualizzata sopra a questa è stata spostata o chiusa). Il ciclo di eventi della GUI è illustrato nella Figura 4.8. Abbiamo analizzato esempi di gestione degli eventi nel Capitolo 3.

```
PAD = "0.75m"
WORKING, CANCELED, TERMINATING, IDLE = ("WORKING", "CANCELED",
        "TERMINATING", "IDLE")

class Canceled(Exception): pass
```

Qui sono riportate alcune delle costanti che vengono importate dai moduli GUI di ImageScale utilizzando `from Globals import *`. Il `PAD` rappresenta una distanza di riempimento di 0,75 mm utilizzata per il layout dei widget (di cui non riportiamo il codice). Le altre costanti sono soltanto cifre che identificano lo stato in cui si trova l'applicazione: `WORKING`, `CANCELED`, `TERMINATING` o `IDLE`. Vedremo successivamente come viene utilizzata l'eccezione `Canceled`.

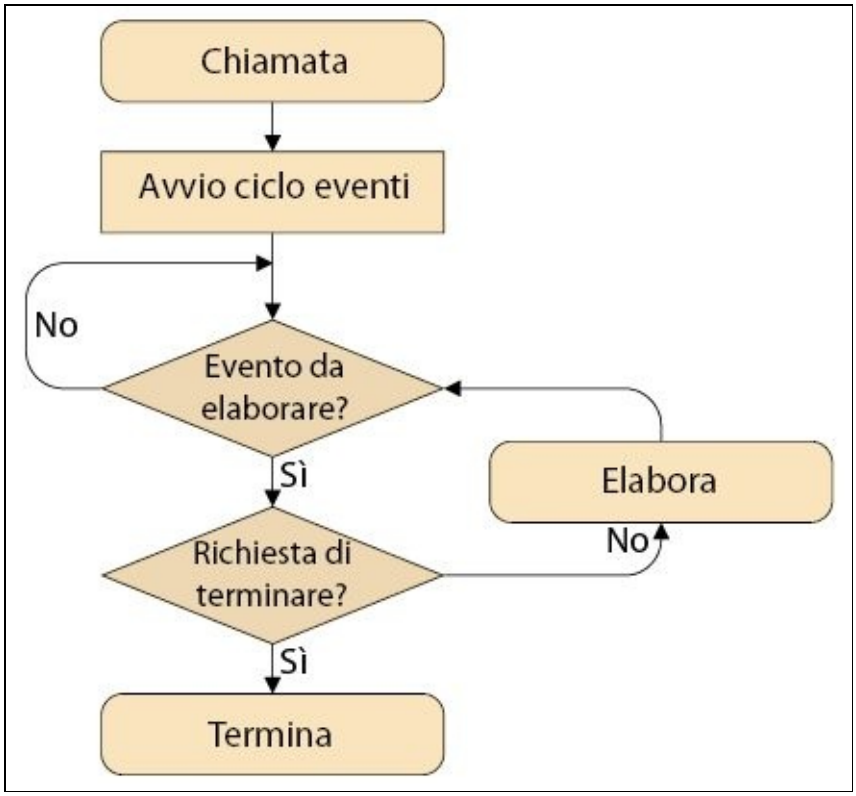


Figura 4.8 Un classico ciclo di eventi GUI.

```
class Window(ttk.Frame):

    def __init__(self, master):
        super().__init__(master, padding=PAD)
        self.create_variables()
        self.create_ui()
        self.sourceEntry.focus()
```

Quando viene creata una `Window`, deve richiamare il metodo `__init__()` della sua classe base. In questo caso creiamo anche le variabili che il programma utilizzerà e la stessa interfaccia utente. Alla fine, impostiamo l'utilizzo della tastiera per la casella di inserimento del testo che serve a indicare la directory di origine. Ciò significa che l'utente può digitare immediatamente la directory, oppure, ovviamente, può fare clic sul pulsante *Source* per attivare una finestra di dialogo per la scelta della directory, e impostarla in questo modo.

Non mostreremo il metodo `create_ui()`, né i metodi `create_widgets()`, `layout_widgets()` e `create_bindings()` da questo attivati, perché riguardano soltanto la creazione della GUI e non hanno nulla a che vedere con la programmazione concorrente (naturalmente presenteremo esempi di creazione di GUI nel Capitolo 7, quando introdurremo la programmazione con Tkinter).

```
def create_variables(self):
    self.sourceText = tk.StringVar()
    self.targetText = tk.StringVar()
    self.statusText = tk.StringVar()
    self.statusText.set("Choose or enter folders, then click Scale…")
    self.dimensionText = tk.StringVar()
    self.total = self.copied = self.scaled = 0
```

```
        self.worker = None
        self.state = multiprocessing.Manager().Value("i", IDLE)
```

Abbiamo illustrato soltanto le righe più importanti di questo metodo. Le variabili `tkinter.StringVar` contengono stringhe che sono associate ai widget dell'interfaccia utente. Le variabili `total`, `copied` e `scaled` sono utilizzate a scopo di conteggio. Il thread worker, inizialmente `None`, viene impostato come secondo thread quando l'utente avanza una richiesta di elaborazione di qualsiasi tipo.

Se l'utente annulla (ovvero, fa clic sul pulsante *Cancel*), come vedremo successivamente, viene richiamato il metodo `scale_or_cancel()`. Questo metodo imposta lo stato dell'applicazione (che può essere `WORKING`, `CANCELED`, `TERMINATING` o `IDLE`). Analogamente, se l'utente chiude l'applicazione (ovvero, fa clic sul pulsante *Quit*), viene richiamato il metodo `close()`. Naturalmente, se l'utente annulla il ridimensionamento o termina l'applicazione durante il processo di ridimensionamento, la risposta deve essere il più veloce possibile. Ciò significa modificare il testo del pulsante *Cancel* in `Canceling…` e disattivare il pulsante, e impedire che i processi del thread worker proseguano nel loro lavoro. Una volta fermati i processi, il pulsante *Scale* deve essere riattivato. Ciò significa che entrambi i thread e i processi del thread worker devono essere in grado di controllare regolarmente lo stato dell'applicazione per verificare se l'utente abbia annullato o terminato l'applicazione.

Un modo per rendere accessibile lo stato dell'applicazione sarebbe quello di utilizzare una variabile di stato e un lock. Ma ciò significherebbe che dovremmo impostare il lock prima di qualsiasi accesso alla variabile di stato e poi disattivarlo. Non è cosa difficile utilizzando un context manager, ma è facile dimenticare di impostare il lock. Fortunatamente, il modulo `multiprocessing` fornisce la classe `multiprocessing.Value`, che può contenere un valore singolo di un determinato tipo a cui si può accedere in sicurezza perché ha un suo sistema di lock (come le code thread-safe). Per creare un `Value`, dobbiamo passare un identificatore di tipo - in questo caso abbiamo utilizzato `"i"` per indicare `int` - e un valore iniziale, qui la costante `IDLE` poiché l'applicazione inizia nello stato `IDLE`.

Un punto importante è che, invece di creare direttamente un `multiprocessing.Value`, abbiamo creato un `multiprocessing.Manager` e fatto in modo che creasse un `Value` in nostra vece. Questo aspetto è essenziale per il corretto funzionamento del valore `Value` (se avessimo più di un `Value` o `Array`, dovremmo creare un'istanza di `multiprocessing.Manager`

e utilizzarla per ciascun valore, ma in questo esempio non è assolutamente necessario).

```
def create_bindings(self):
    if not TkUtil.mac():
        self.master.bind("<Alt-a>", lambda *args:
                self.targetEntry.focus())
        self.master.bind("<Alt-b>", self.about)
        self.master.bind("<Alt-c>", self.scale_or_cancel)
        ...
    self.sourceEntry.bind("<KeyRelease>", self.update_ui)
    self.targetEntry.bind("<KeyRelease>", self.update_ui)
    self.master.bind("<Return>", self.scale_or_cancel)
```

Quando creiamo un pulsante `tkinter.ttk.Button`, possiamo associarvi un comando (ovvero una funzione o metodo) che Tkinter deve eseguire quando il pulsante viene attivato. Ciò è stato fatto nel metodo `create_widgets()`, non mostrato qui. Desideriamo anche fornire supporto agli utenti che utilizzano la tastiera. Quindi, per esempio, se l'utente fa clic sul pulsante *Scale*, o - su piattaforme non OS X - preme i tasti Alt+C o Invio, viene attivato il metodo `scale_or_cancel()`.

Quando l'applicazione si avvia, il pulsante *Scala* è disabilitato perché non ci sono cartelle di origine e di destinazione. Ma una volta impostate queste cartelle (digitando o utilizzando una finestra di dialogo di impostazione della directory mediante i pulsanti *Source* e *Target*), il pulsante *Scale* deve essere attivabile. Per ottenere ciò, disponiamo del metodo `update_ui()` che abilita o disabilita i widget a seconda della situazione, e richiamiamo questo metodo ogni volta che l'utente digita nelle caselle di testo di origine o destinazione.

Il modulo `TkUtil` è incluso nel codice degli esempi del libro. Contiene diverse utilità, come `TkUtil.mac()`, che rileva se il sistema operativo è di tipo OS X, e fornisce supporto generico per le finestre modali e altre utili funzionalità.

NOTA

Tkinter - o meglio, la piattaforma sottostante Tcl/Tk 8.5 - tiene conto di alcune differenze tra piattaforme Linux, OS X e Windows. Tuttavia, molte delle differenze devono ancora essere gestite autonomamente dal programmatore, soprattutto per OS X.

```
def update_ui(self, *args):
    guiState = self.state.value
    if guiState == WORKING:
        text = "Cancel"
        underline = 0 if not TkUtil.mac() else -1
        state = "!" + tk.DISABLED
    elif guiState in {CANCELED, TERMINATING}:
        text = "Canceling…"
        underline = -1
        state = tk.DISABLED
    elif guiState == IDLE:
        text = "Scale"
        underline = 1 if not TkUtil.mac() else -1
        state = ("!" + tk.DISABLED if self.sourceText.get() and
    self.targetText.get() else tk.DISABLED)
    self.scaleButton.state((state,))
    self.scaleButton.config(text=text, underline=underline)
```

```
        state = tk.DISABLED if guiState != IDLE else "!" + tk.DISABLED
        for widget in (self.sourceEntry, self.sourceButton,
                       self.targetEntry, self.targetButton):
            widget.state((state,))
        self.master.update()  # Si assicura che la GUI esegua un refresh
```

Questo metodo viene richiamato quando avviene un cambiamento che potrebbe influire sull'interfaccia utente. Può essere richiamato direttamente o in risposta a un evento - un tasto premuto o un pulsante attivato che è collegato ad esso - e in questo caso vengono passati uno più ulteriori argomenti, che ignoriamo.

Iniziamo rilevando lo stato della GUI (`WORKING`, `CANCELED`, `TERMINATING` o `IDLE`). Invece di creare una variabile, avremmo potuto utilizzare direttamente `self.state.value` in ciascuna stringa con `if`, ma al di sotto deve esserci un lock, quindi è meglio attivarlo una volta per ridurne al minimo la durata. Non importa se lo stato cambia durante l'esecuzione del metodo, perché questo cambiamento risulterebbe comunque in una nuova chiamata del metodo.

Se l'applicazione è in funzione, desideriamo che il testo del pulsante di ridimensionamento sia `Cancel` (poiché stiamo utilizzando questo pulsante come un pulsante di avvio e interruzione), e che sia attivo. Sulla maggior parte delle piattaforme, una lettera sottolineata indica un tasto rapido (per esempio, il pulsante *Cancel* può esser attivato premendo Alt+C), ma questa funzionalità non è supportata dai sistemi OS X, quindi su queste piattaforme togliamo la sottolineatura utilizzando una posizione d'indice non valida.

Una volta noto lo stato dell'applicazione, aggiorniamo il testo e la sottolineatura del pulsante di ridimensionamento, e abilitiamo e disabilitiamo alcuni dei widget a seconda del caso. E alla fine richiamiamo il metodo `update()` per forzare Tkinter all'aggiornamento della finestra in modo che rifletta tutte le modifiche apportate.

```
    def scale_or_cancel(self, event=None):
        if self.scaleButton.instate((tk.DISABLED,)):
            return
        if self.scaleButton.cget("text") == "Cancel":
            self.state.value = CANCELED
            self.update_ui()
        else:
            self.state.value = WORKING
            self.update_ui()
            self.scale()
```

Il pulsante di ridimensionamento viene utilizzato per iniziare il ridimensionamento, oppure per annullarlo, poiché ne modifichiamo il testo a seconda dello stato dell'applicazione. Se l'utente preme Alt+C (su piattaforme diverse da OS X) o Invio, fa clic sul pulsante *Scale* o *Cancel* (ossia il pulsante di ridimensionamento), viene richiamato questo metodo.

Se il pulsante non è abilitato, non possiamo fare niente (non è ovviamente possibile fare clic su un pulsante disabilitato, ma l'utente potrebbe comunque richiamare questo metodo utilizzando una combinazione di tasti di scelta rapida come Alt+C).

Se il pulsante è abilitato e il testo è *Cancel*, modifichiamo lo stato dell'applicazione in `CANCELED` e aggiorniamo l'interfaccia utente. Nello specifico, il pulsante di ridimensionamento sarà disabilitato e il testo modificato in *Canceling*. Come vedremo, durante l'elaborazione controlliamo regolarmente se lo stato dell'applicazione è cambiato, così rileveremo subito l'annullamento e fermeremo qualsiasi elaborazione ulteriore. Quando l'annullamento è stato completato, il pulsante di ridimensionamento sarà nuovamente attivo e il testo impostato sarà *Scale*. Nella Figura 4.5 è mostrata l'applicazione prima, durante e dopo il ridimensionamento di alcune immagini. La Figura 4.9 mostra l'applicazione prima, durante e dopo l'annullamento.

Se il testo del pulsante è *Scale*, impostiamo lo stato a `WORKING`, aggiorniamo l'interfaccia utente (quindi ora il testo del pulsante è *Cancel*) e iniziamo il ridimensionamento.

```
def scale(self):
    self.total = self.copied = self.scaled = 0
    self.configure(cursor="watch")
    self.statusText.set("Scaling…")
    self.master.update()  # Si assicura che la GUI esegua il refresh
    target = self.targetText.get()
    if not os.path.exists(target):
        os.makedirs(target)
    self.worker = threading.Thread(target=ImageScale.scale, args=(
            int(self.dimensionText.get()), self.sourceText.get(),
            target, self.report_progress, self.state,
            self.when_finished))
    self.worker.daemon = True
    self.worker.start()  # ritorna immediatamente
```

Iniziamo impostando tutti i valori a zero e modificando il cursore dell'applicazione in modo che indichi uno stato "occupato". Quindi aggiorniamo l'etichetta di stato e riaggiorniamo la GUI in modo che l'utente possa vedere che il ridimensionamento è iniziato. Successivamente, creiamo la directory di destinazione, se non esiste già, e anche qualsiasi directory intermedia mancante.

Quando tutto è pronto, creiamo un nuovo thread worker (qualsiasi thread worker precedente non è più utilizzabile e può quindi essere eliminato) utilizzando la funzione `threading.Thread()`, impostando la funzione che vogliamo eseguire nel thread e gli argomenti relativi. Gli argomenti sono la dimensione massima delle immagini ridimensionate; le directory di origine e di destinazione; un comando associato (in questo caso il metodo bound `self.report_progress()`), che viene richiamato quando

viene eseguito ciascun job; il `Value` di stato dell'applicazione in modo che i processi del thread worker possano verificare se l'utente ha annullato; e un comando associato (in questo caso, il metodo bound `self.when_finished()`) da richiamare quando l'elaborazione è stata portata a termine (o è stata annullata).

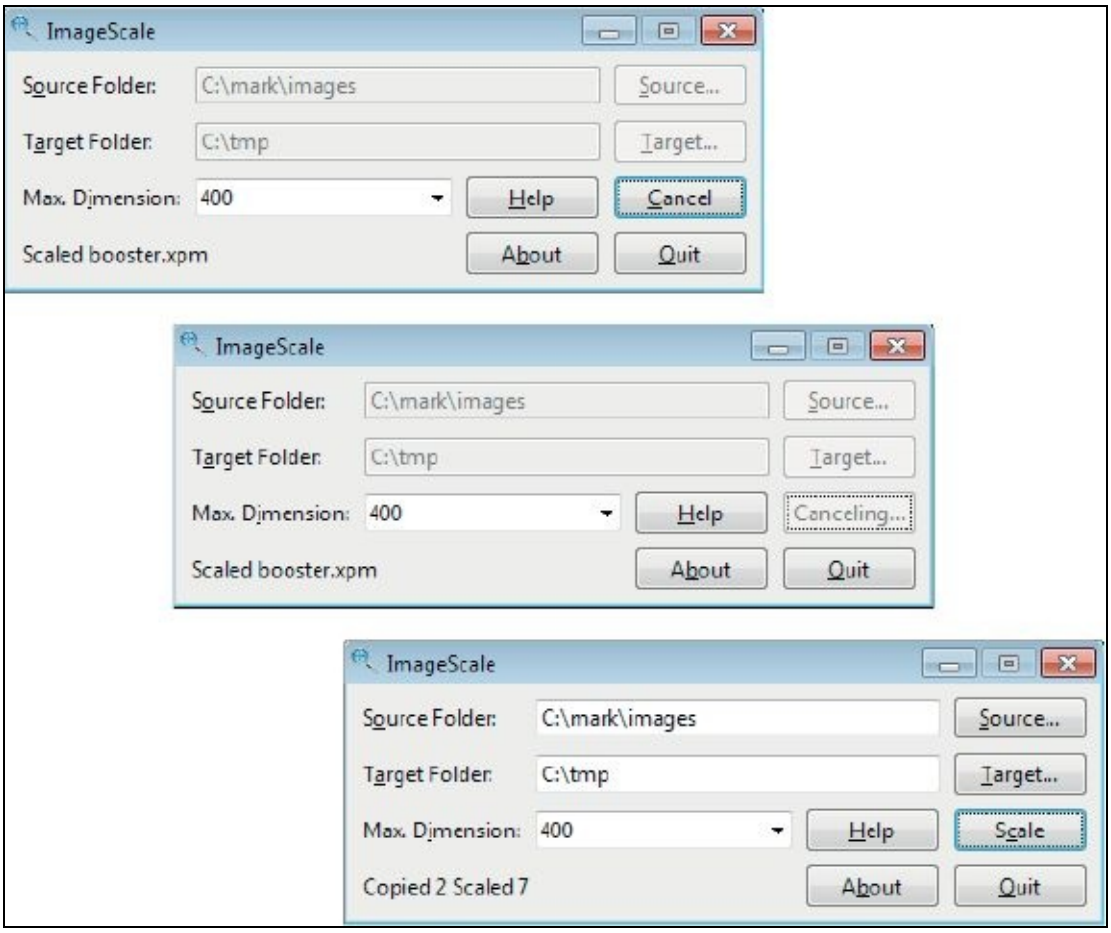


Figura 4.9 L'applicazione ImageScale prima, durante, e dopo l'annullamento.

Una volta creato il thread, lo rendiamo un demone per essere certi che venga terminato in maniera pulita se l'utente chiude l'applicazione, quindi iniziamo l'esecuzione.

Come vedremo, il thread worker non esegue quasi nessuna operazione in modo che il thread della GUI abbia il maggior tempo possibile per operare sul core condiviso. La funzione `ImageScale.scale()` delega tutte le operazioni del thread worker ai processi multipli che vengono eseguiti su altri core (in una macchina multicore), in modo che la GUI rimanga reattiva (anche se con questa architettura la GUI è comunque reattiva, anche su una macchina a core singolo, perché il thread della GUI occupa comunque la CPU per lo stesso tempo del thread worker).

Il modulo worker di ImageScale

Abbiamo mantenute separate le funzioni attivate dal thread operativo nel loro stesso modulo, `imagescale/ImageScale.py`, dal quale sono tratte le stringhe di codice di questo sottoparagafo. Non è soltanto una questione di comodità organizzativa, ma una necessità, poiché desideriamo che queste funzioni siano utilizzabili per il modulo `multiprocessing`, e ciò implica che debbano essere importabili e che qualsiasi dato di modulo sia recuperabile. I moduli che contengono widget o sottoclassi di widget GUI non possono - e certamente non devono - essere importati in questo modo, perché questa procedura può generare confusione nel sistema delle finestre.

Il modulo ha tre funzioni, la prima delle quali, `ImageScale.scale()`, è quella eseguita dal thread worker.

```
def scale(size, source, target, report_progress, state, when_finished):
    futures = set()
    with concurrent.futures.ProcessPoolExecutor(
            max_workers=multiprocessing.cpu_count()) as executor:
        for sourceImage, targetImage in get_jobs(source, target):
            future = executor.submit(scale_one, size, sourceImage,
                    targetImage, state)
            future.add_done_callback(report_progress)
            futures.add(future)
            if state.value in {CANCELED, TERMINATING}:
                executor.shutdown()
                for future in futures:
                    future.cancel()
                break
        concurrent.futures.wait(futures)  # Continua a lavorare finché non termina
    if state.value != TERMINATING:
        when_finished()
```

Questa funzione è eseguita dal thread `self.worker` creato con il metodo `Main.Window.scale()`. Utilizza un insieme di processi (ovvero è multiprocesso piuttosto che multithread) per eseguire effettivamente le operazioni. Ciò assicura che soltanto il thread operativo possa attivare questa funzione, mentre tutte le operazioni vengono delegate a processi separati.

Per ciascuna immagine originaria e finale recuperata dalla funzione `ImageScale.get_jobs()`, viene creato un future per eseguire la funzione `ImageScale.scale_one()` con la dimensione massima (`size`), l’immagine originaria e quella finale, e il valore di stato `Value` dell’applicazione.

Nei paragrafi precedenti abbiamo atteso che i future terminassero utilizzando la funzione `concurrent.futures.as_completed()`, ma in questo caso aggiungiamo una funzione di callback a ciascun future (il metodo `Main.Window.report_progress()`) e utilizziamo invece `concurrent.futures.wait()`.

Dopo che tutti i future sono stati aggiunti controlliamo se l’utente ha annullato o terminato, e se così è interrompiamo l’insieme dei processi e annulliamo tutti i future.

Per default, la funzione `concurrent.futures.Executor.shutdown()` ha effetto soltanto una volta che tutti i future sono stati terminati o annullati.

Una volta che tutti i future sono stati creati, questa funzione blocca (il thread worker, non il thread della GUI) con il comando `concurrent.futures.wait()`. Ciò significa che se l'utente annulla dopo la creazione dei future, dobbiamo controllare l'eventuale annullamento quando viene eseguito ciascun comando di future (ovvero, all'interno della funzione `ImageScale.scale_one()`).

Una volta che l'elaborazione è stata terminata o annullata, e finché non arriviamo alla fase di chiusura, richiamiamo il comando di callback `when_finished()` che è stato inserito. Una volta giunti alla conclusione del metodo `scale()`, il thread è concluso.

```
def get_jobs(source, target):
    for name in os.listdir(source):
        yield os.path.join(source, name), os.path.join(target, name)
```

Questa breve funzione generatore genera tuple di due elementi con i nomi dell'immagine di origine e di destinazione con percorsi completi.

```
Result = collections.namedtuple("Result", "name copied scaled")

def scale_one(size, sourceImage, targetImage, state):
    if state.value in {CANCELED, TERMINATING}:
        raise Canceled()
    oldImage = Image.Image.from_file(sourceImage)
    if state.value in {CANCELED, TERMINATING}:
        raise Canceled()
    if oldImage.width <= size and oldImage.height <= size:
        oldImage.save(targetImage)
        return Result(targetImage, 1, 0)
    else:
        scale = min(size / oldImage.width, size / oldImage.height)
        newImage = oldImage.scale(scale)
        if state.value in {CANCELED, TERMINATING}:
            raise Canceled()
        newImage.save(targetImage)
        return Result(targetImage, 0, 1)
```

Questa è la funzione che esegue effettivamente il ridimensionamento (o la copia); utilizza il modulo `cyImage` (trattato nel Capitolo 5) o ripiega sul modulo `Image` (trattato nel Capitolo 3). Per ciascun job viene prodotto un `Result`, oppure viene sollevata l'eccezione personalizzata `Canceled` se l'utente ha annullato o ha chiuso.

Se l'utente annulla o chiude durante un processo di caricamento, ridimensionamento o salvataggio, la funzione non si interromperà finché non è terminata l'azione corrispondente. Ciò significa che quando l'utente annulla o chiude, potrebbe dover attendere *n* immagini prima che si completi il caricamento, il ridimensionamento o il salvataggio, dove *n* è il numero dei processi dell'insieme complessivo che devono essere eseguiti prima che l'annullamento abbia effetto. Verificare l'impostazione di comandi di annullamento o terminazione prima di ogni

operazione complessa e non annullabile (caricare l'immagine originale, ridimensionare, salvare) è la cosa migliore che possiamo fare per rendere l'applicazione il più possibile reattiva.

Ogni volta che viene prodotto un risultato (o sollevata un'eccezione `Canceled`), il future associato termina. E poiché abbiamo associato un comando a ciascun future, questo comando viene attivato, e in questo caso si tratta del metodo `Main.Window.report_progress()`.

Gestione dell'avanzamento di un'operazione da parte della GUI

In questo paragrafo esaminiamo i metodi della GUI con cui viene indicato all'utente il progresso di un'operazione. Questi metodi si trovano nel file `imagescale/Main.py`.

Poiché abbiamo diversi processi che eseguono future, è possibile che due o più di essi richiamino `report_progress()` contemporaneamente. In effetti non dovrebbe succedere mai, perché il comando associato con un future viene attivato nel thread in cui l'associazione è stata creata - in questo caso, il thread worker - e poiché il thread worker è soltanto uno, in teoria il metodo non può essere richiamato più di una volta contemporaneamente. Tuttavia, si tratta di un dettaglio dell'implementazione, e come tale sarebbe riduttivo fare affidamento su di esso. Quindi, per quanto desideriamo realizzare programmi concorrenti di alto livello ed evitare caratteristiche di medio livello come i lock, in questo caso non abbiamo nessuna scelta. Perciò abbiamo creato un lock per assicurarci che il lavoro del metodo `report_progress()` sia sempre serializzato.

```
ReportLock = threading.Lock()
```

Il lock è in `Main.py`, e viene utilizzato soltanto in un unico metodo.

```
    def report_progress(self, future):
        if self.state.value in {CANCELED, TERMINATING}:
            return
        with ReportLock:  # Serializza chiamate a Window.report_progress()
            self.total += 1  # e accede a self.total, e così via.
            if future.exception() is None:
            result = future.result()
            self.copied += result.copied
            self.scaled += result.scaled
            name = os.path.basename(result.name)
            self.statusText.set("{} {}".format(
                    "Copied" if result.copied else "Scaled", name))
            self.master.update()  # Si assicura che la GUI esegua il refresh
```

Questo metodo viene richiamato ogni volta che un future termina, normalmente o con un'eccezione. Se l'utente ha annullato, lasciamo perdere, poiché l'interfaccia utente verrà comunque aggiornata dal metodo `when_finished()`. E se l'utente ha chiuso, non ha senso aggiornare l'interfaccia utente, poiché verrà chiusa con la terminazione dell'applicazione.

La maggior parte del corpo del metodo è serializzata da un lock, quindi se due o più future terminano allo stesso tempo, soltanto per uno alla volta verrà eseguita questa parte del metodo; gli altri saranno bloccati fino al termine del lock (non dobbiamo preoccuparci del `self.state Value`, perché si tratta di un valore sincronizzato). Poiché siamo nel contesto di un lock, vogliamo svolgere la minor quantità di lavoro possibile in modo da ridurre al minimo ogni blocco.

Iniziamo incrementando il numero totale di job. Se il future ha sollevato un'eccezione (per esempio, `Canceled`), non facciamo nient'altro. Diversamente, aumentiamo il numero di job di copia e ridimensionamento (di 0 e 1 o 1 e 0) e aggiorniamo l'etichetta di stato della GUI. È molto importante effettuare gli aggiornamenti della GUI nel contesto di un lock. Serve per evitare il rischio di un comportamento indefinito che potrebbe verificarsi se due o più aggiornamenti della GUI venissero eseguiti in maniera concorrente.

```
def when_finished(self): self.state.value = IDLE
    self.configure(cursor="arrow")
    self.update_ui()
    result = "Copied {} Scaled {}".format(self.copied, self.scaled)
    difference = self.total - (self.copied + self.scaled)
    if difference:  # Questa parte entra in gioco se l'utente ha annullato
        result += " Skipped {}".format(difference)
    self.statusText.set(result)
    self.master.update()  # Si assicura che la GUI esegua il refresh
```

Questo metodo viene richiamato dal thread worker dopo che ha terminato, per completare l'elaborazione o a causa dell'annullamento, ma non nel caso della terminazione. Poiché questo metodo viene richiamato soltanto quando il thread worker e i suoi processi sono terminati, non è assolutamente necessario utilizzare `ReportLock`. Il metodo imposta lo stato dell'applicazione nuovamente su `IDLE`, riabilita il normale cursore a freccia e imposta il testo dell'etichetta di stato in modo che visualizzi le operazioni effettuate e indichi se l'utente ha annullato.

Gestione della terminazione del programma GUI

Terminare un programma GUI concorrente non significa semplicemente chiuderlo. Dobbiamo innanzitutto fermare tutti i thread worker - e in particolare i processi - in

modo che possiamo terminare il programma in maniera pulita e non lasciare nessun processo fantasma che sottragga risorse (per esempio, di memoria).

La terminazione viene gestita con il metodo `close()` del modulo `imagescale/Main.py`.

```
def close(self, event=None):
    ...
    if self.worker is not None and self.worker.is_alive():
        self.state.value = TERMINATING
        self.update_ui()
        self.worker.join()  # Attende che il worker termini
    self.quit()
```

Se l'utente fa clic sul pulsante *Quit* o sul pulsante di chiusura della finestra × (o preme Alt+Q nelle piattaforme diverse da OS X), viene impiegato questo metodo. Vengono salvate alcune impostazioni dell'utente (non mostrate qui) e quindi viene effettuato un controllo per verificare se il thread operativo è ancora in funzione (ovvero, se l'utente ha chiuso mentre il ridimensionamento è in corso). Se è così, il metodo imposta lo stato dell'applicazione a `TERMINATING` e aggiorna l'interfaccia utente in modo che l'utente possa vedere che l'avanzamento viene annullato. Il cambiamento di stato viene rilevato dai processi del thread worker (poiché verificano regolarmente il `Value` di stato) e non appena questi rilevano la terminazione, cessano di operare. La chiamata di `threading.Thread.join()` blocca fino a che il thread worker e i suoi processi terminano. Se non aspettiamo, potremmo lasciare in esecuzione alcuni processi fantasma (ovvero, processi che consumano memoria senza eseguire niente di utile). Alla fine interviene `tkinter.ttk.Frame.quit()` per terminare l'applicazione.

L'applicazione ImageScale mostra come sia possibile integrare programmazione multithreading e multiprocessing per produrre un'applicazione GUI che funzioni in maniera concorrente, rimanendo comunque reattiva per l'utente. Inoltre, l'architettura dell'applicazione registra il progresso delle operazioni e l'annullamento.

Scrivere programmi concorrenti utilizzando funzionalità di alto livello come le code thread-safe e i future, ed evitando funzionalità di medio livello come i lock, è molto più facile che utilizzare caratteristiche di basso e medio livello. Tuttavia, dobbiamo procedere con attenzione per essere certi che il nostro programma concorrente abbia davvero prestazioni superiori rispetto a un programma equivalente non concorrente. Per esempio, in Python dobbiamo evitare di utilizzare funzionalità multithreading per processi CPU-bound.

Dobbiamo anche essere sicuri di non modificare accidentalmente i dati mutabili condivisi. Quindi, dobbiamo sempre trasmettere dati immodificabili (per esempio, numeri e stringhe), o dati murabili che vengano soltanto letti (ovvero, che siano stati scritti prima che iniziasse l'elaborazione dell'applicazione concorrente), oppure

effettuare la copia profonda di dati mutabili. In ogni caso, come ha illustrato il caso di studio dell'applicazione ImageScale, talvolta è proprio necessario condividere dei dati. Fortunatamente, utilizzando `multiprocessing.Value` (o `multiprocessing.Array`), siamo in grado di farlo senza dover attivare dei lock. In alternativa, possiamo creare delle classi thread-safe personalizzate. Vedremo un esempio nel Capitolo 6.

Estendere Python

Python è sufficientemente veloce per la gran parte dei programmi. E nei casi in cui non lo è, spesso possiamo raggiungere la velocità richiesta utilizzando la concorrenza, come abbiamo visto nel capitolo precedente. A volte, però, c'è davvero bisogno di maggior velocità di elaborazione.

Esistono tre modi in cui possiamo velocizzare l'esecuzione dei nostri programmi Python: possiamo usare PyPy (`http://pypy.org`), che ha un compilatore JIT (*Just in Time*) integrato; possiamo usare codice in C o in C++ per le funzioni critiche oppure possiamo compilare il codice Python (o Cython) in C utilizzando Cython (sono via via disponibili nuovi compilatori Python come Numba, `http://numba.pydata.org`, e Nuitka, `http://nuitka.net`).

Una volta installato PyPy si possono eseguire i propri programmi Python utilizzando l'interprete PyPy al posto dell'interprete standard CPython. Questo produce un notevole miglioramento sui programmi di lunga esecuzione, dato che il costo della compilazione JIT viene ammortizzato dalla durata di esecuzione ridotta, mentre potrebbe addirittura rallentare i programmi con tempi di esecuzione molto brevi.

Per usare codice in C o C++, che sia codice nostro o librerie esterne, dobbiamo renderlo disponibile al programma Python, in modo che possa beneficiare della velocità di esecuzione del C e del C++. Per chi vuole scrivere proprio codice C o C++ un approccio sensato consiste nell'utilizzo diretto dell'interfaccia Python C (`http://docs.python.org/3/extending`). Chi, invece, volesse fare uso di codice C o C++ già esistente, avrà a disposizione diversi altri approcci. La prima opzione consiste nell'utilizzo di un wrapper che racchiuda il codice C o C++ e ne produca un'interfaccia Python. I due strumenti più usati per far questo sono SWIG (`http://www.swig.org`) e SIP (`http://www.riverbankcomputing.co.uk/software/sip`). Un'altra opzione per il solo C++ consiste nell'utilizzo di `boost::python` (`http://www.boost.org/libs/python/doc/`). CFFI (*C Foreign Function Interface for Python*), nonostante sia l'ultimo arrivato, viene usato dal ben noto PyPy (`http://bitbucket.org/cffi/cffi`).

ESTENDERE PYTHON SU OS X E WINDOWS

Sebbene gli esempi di questo capitolo siano stati provati solo su Linux, dovrebbero funzionare anche su OS X e Windows (per molti programmatori `ctypes` e Cython, queste saranno le piattaforme di sviluppo principali). Tuttavia, potrebbero essere necessari piccoli aggiustamenti specifici della piattaforma utilizzata. Questo perché la maggioranza dei sistemi Linux dispone di un compilatore GCC pronto per l'uso e di librerie ed eseguibili dell'architettura corretta (32 o 64 bit) per la macchina su cui girano, mentre la situazione su sistemi OS X e Windows è generalmente molto più complicata o almeno un tantino differente.

Su OS X e Windows, in genere è necessario far corrispondere compilatore e architettura (a 32 o 64 bit) usati per compilare Python con quelli usati dalle librerie condivise esterne (i file `.dylib` e `.DLL`) o dal codice Cython. Su OS X, il compilatore potrebbe essere sempre GCC ma oggi sarà più probabilmente Clang; su Windows potrebbe essere una qualche incarnazione di GCC o un compilatore commerciale come quelli venduti da Microsoft. Inoltre, sia OS X che Windows spesso hanno librerie condivise all'interno delle cartelle delle applicazioni piuttosto che in cartelle di sistema, e i file header devono spesso essere ottenuti separatamente.

Perciò, invece di fornire un mare di informazioni specifiche per piattaforma e compilatore (che diventerebbero rapidamente obsolete) ci concentreremo sull'uso di `ctypes` e Cython, lasciando agli utilizzatori di sistemi non Linux l'onere di determinare i requisiti d'uso di tali tecnologie sulle proprie macchine.

Tutte le possibilità appena descritte meritano di essere esplorate, ma in questo capitolo ci concentreremo su due altre tecnologie: il pacchetto `ctypes` fornito come parte della libreria standard di Python (`http://docs.python.org/3/library/ctypes.html`) e Cython (`http://cython.org`). Entrambe possono essere usate per fornire a Python interfacce per codice C e C++ autoprodotto o di terze parti.

Cython può inoltre essere usato per compilare sia codice Python che Cython in C per migliorarne le prestazioni - a volte con risultati stupefacenti.

Accesso a librerie C con ctypes

Il pacchetto `ctypes` della libreria standard permette di accedere a funzioni di terze parti o autoprodotte scritte in C o C++ (e in realtà in ogni linguaggio compilato che usi la convenzione di chiamata del C) che siano state compilate all'interno di una libreria condivisa (`.so` in Linux, `.dylib` in OS X o `.DLL` in Windows). In questo paragrafo e nella prima parte del paragrafo seguente scriveremo un modulo per accedere da Python ad alcune funzioni C di una libreria condivisa. La libreria che useremo sarà `libhyphen.so`, o, su alcuni sistemi, `libhyphen.uno.so` (cfr. il riquadro "Estendere Python su OS X e Windows"). Questa libreria in genere viene installata da OpenOffice.org o da LibreOffice e contiene una funzione che, data una parola, la restituisce con trattini di separazione per ciascuna sillaba. Anche se la funzione fa quel che sembra essere un'operazione facile, la firma della funzione è abbastanza complicata (cosa che la rende ideale per illustrare `ctypes`). Infatti useremo tre funzioni: una per caricare il dizionario di sillabazione, una per effettuare la sillabazione vera e propria, l'ultima per liberare le risorse utilizzate a compito concluso.

La tipica sequenza di utilizzo di `ctypes` consiste nel caricamento della libreria in memoria, nell'ottenimento dei riferimenti alle funzioni che si vogliono usare e nella successiva chiamata delle funzioni stesse. Il modulo `Hyphenate1.py` segue questa sequenza. Per prima cosa, vediamo come viene impiegato il modulo. Ecco una sessione interattiva eseguita al prompt di Python (per esempio in IDLE):

```
>>> import os
>>> import Hyphenate1 as Hyphenate
>>>
>>> # Localizza i vostri file hyph*.dic
>>> path = "/usr/share/hyph_dic"
>>> if not os.path.exists(path): path = os.path.dirname(__file__)
>>> usHyphDic = os.path.join(path, "hyph_en_US.dic")
>>> deHyphDic = os.path.join(path, "hyph_de_DE.dic")
>>>
>>> # Crea dei wrapper in modo che non occorra specificare il dizionario
>>> hyphenate = lambda word: Hyphenate.hyphenate(word, usHyphDic)
>>> hyphenate_de = lambda word: Hyphenate.hyphenate(word, deHyphDic)
>>>
>>> # Usa i wrapper
>>> print(hyphenate("extraordinary"))
ex-traor-di-nary
>>> print(hyphenate_de("außergewöhnlich"))
außerge-wöhn-lich
```

La sola funzione che usiamo all'interno del modulo è `Hyphenate1.hyphenate()`, che fa uso del motore di sillabazione della libreria. Nel modulo ci sono alcune funzioni private che accedono ad altre funzioni della libreria. Tra l'altro, i dizionari usati per la sillabazione hanno lo stesso formato utilizzato dal programma di impaginazione open source TEX.

Tutto il codice è nel modulo `Hyphenate1.py`. Le tre funzioni che utilizziamo della libreria sono:

```
HyphenDict *hnj_hyphen_load(const char *filename);

void hnj_hyphen_free(HyphenDict *hdict);

int hnj_hyphen_hyphenate2(HyphenDict *hdict, const char *word,
    int word_size, char *hyphens, char *hyphenated_word, char ***rep,
    int **pos, int **cut);
```

Queste firme provengono dal file header `hyphen.h`. Il simbolo `*` in C e C++ rappresenta un puntatore. Un puntatore contiene l'indirizzo di un blocco di memoria; ovvero di un intervallo contiguo di celle di dimensione di un byte. Il blocco può essere composto da un byte - nel caso minimo - o avere qualsiasi altra dimensione; per esempio, 8 byte per un intero a 64 bit. Le stringhe generalmente richiedono da 1 a 4 byte per carattere (a seconda della modalità di codifica in memoria) più un certo overhead fisso.

La prima funzione, `hnj_hyphen_load()`, richiede che le venga passato un nome di file sotto forma di puntatore a un blocco di `char` (byte). Questo dev'essere un dizionario di sillabazione in formato TEX. La funzione `hnj_hyphen_load()` restituisce un puntatore a `HyphenDict struct` - un complesso oggetto aggregato (come un'istanza di classe Python). Fortunatamente non è necessario conoscere nulla del contenuto di `HyphenDict`, perché ci basta il puntatore.

In C, le funzioni che accettano delle stringhe C come parametri - ovvero puntatori a blocchi di caratteri o byte - prevedono generalmente due approcci: possono richiedere solo un puntatore, nel qual caso suppongono che l'ultimo carattere del blocco sia `0x00` (`'\0'`) (in questo caso si dice che la stringa C è "terminata da null", o "null-terminated"), oppure richiedono un puntatore e il numero di caratteri da leggere. La funzione `hnj_hyphen_load()` richiede solo un puntatore, per cui la stringa fornita come parametro dovrà essere terminata da null. Come vedremo, se passiamo una stringa a `ctypes.create_string_buffer()` otteniamo la stessa convertita in una stringa C terminata da null.

Ogni volta che carichiamo in memoria un dizionario di sillabazione, dovremo curarci di scaricarlo dopo l'uso. Se non lo facessimo, occuperemmo memoria inutilmente. La seconda funzione, `hnj_hyphen_free()`, richiede un puntatore a un `HyphenDict` e libera le risorse associate. Questa funzione non ha alcun valore di ritorno. Una volta liberato, un puntatore non dev'essere mai riutilizzato, così come non si riutilizza una variabile dopo averla cancellata con `del` in Python.

La terza funzione, `hnj_hyphen_hyphenate2()`, è quella che compie la sillabazione vera e propria. L'argomento `hdict` è un puntatore all'oggetto `HyphenDict` che è stato restituito dalla funzione `hnj_hyphen_load()` (e che non è ancora stato liberato dalla funzione `hnj_hyphen_free()`).

`word` è la parola che vogliamo sillabare, fornita come puntatore a un blocco di byte codificati in UTF-8. `word_size` è il numero di byte nel blocco.

`hyphens` è un puntatore a un blocco di byte che non intendiamo usare, ma che dobbiamo comunque passare alla funzione perché questa funzioni correttamente. `hyphenated_word` punta a un blocco di byte sufficientemente lungo da contenere la parola originale codificata UTF-8 con i trattini aggiunti (la libreria in realtà inserisce dei caratteri `=` come separatori). Inizialmente questo blocco dovrebbe contenere tutti byte `0x00`.

`rep` è un puntatore a un puntatore a un blocco di byte; non ne abbiamo necessità, ma dobbiamo fornire un puntatore valido in ogni caso. Analogamente, `pos` e `cut` sono puntatori a puntatori a `int` a cui non siamo interessati. Il valore di ritorno della funzione è un flag binario, dove 1 significa errore e 0 significa successo.

Ora che sappiamo cosa vogliamo utilizzare delle funzioni esterne, passiamo al codice del modulo `Hyphenate1.py` (come sempre, omettendo le inclusioni), cominciando col reperimento e il caricamento in memoria della libreria condivisa di sillabazione.

```
class Error(Exception): pass

_libraryName = ctypes.util.find_library("hyphen")
if _libraryName is None:
    _libraryName = ctypes.util.find_library("hyphen.uno")
if _libraryName is None:
    raise Error("cannot find hyphenation library")

_LibHyphen = ctypes.CDLL(_libraryName)
```

Cominciamo creando una classe per la gestione delle eccezioni, `Hyphenate1.Error`, in modo che gli utenti del nostro modulo possano distinguere tra eccezioni specifiche della libreria ed errori più generali come `ValueError`. La funzione `ctypes.util.find_library()` cerca una libreria condivisa. Su Linux essa aggiunge al nome fornito `lib` all'inizio e `.so` al fondo, in per cui la prima chiamata cercherà `libhyphen.so` in diverse posizioni standard del sistema operativo. Su OS X la libreria cercata sarà `hyphen.dylib` mentre su Windows sarà `hyphen.dll`. Questa libreria viene talvolta chiamata `libhyphen.uno.so`, quindi, se non la troviamo col nome originale, la cercheremo sotto questa forma; se non la troviamo nemmeno così, solleveremo un'eccezione.

Se la libreria viene trovata, viene caricata in memoria tramite la funzione `ctypes.CDLL()` e la variabile privata `_LibHyphen` viene impostata come suo riferimento. Coloro che vogliono scrivere programmi solo per Windows che accedono a interfacce specifiche di tale sistema operativo possono usare le funzioni `ctypes.OleDLL()` e `ctypes.WinDLL()` per caricare le librerie delle API di Windows.

Una volta caricata la libreria, possiamo creare dei wrapper Python per le funzioni di libreria a cui siamo interessati. Un metodo comune consiste nell'assegnare una funzione di libreria a una variabile Python e specificare il tipo degli argomenti (come lista di tipi `ctypes`) e il valore di ritorno (come singolo tipo `ctypes`) della funzione.

Se si fornisce alla funzione un numero errato di argomenti o il tipo di ritorno sbagliato, il programma andrà in crash. Il pacchetto CFFI (`http://bitbucket.org/cffi/cffi`) è molto più robusto sotto questo aspetto e funziona molto meglio assieme all'interprete PyPy (`http://pypy.org`) di quanto non faccia `ctypes`.

```
_load = _LibHyphen.hnj_hyphen_load
_load.argtypes = [ctypes.c_char_p] # const char *filename
_load.restype = ctypes.c_void_p    # HyphenDict *
```

Qui abbiamo creato una funzione privata nel modulo, `_load()`, che - quando viene richiamata - richiama a sua volta la sottostante funzione `hnj_hyphen_load()` della libreria di sillabazione. Una volta ottenuto un riferimento alla funzione di libreria, dobbiamo specificarne argomenti e tipi dei valori di ritorno. In questo caso abbiamo un solo argomento (di tipo C `const char *`), che possiamo rappresentare direttamente con `ctypes.c_char_p` ("C character pointer" - puntatore a char). La funzione restituisce un puntatore a una struct `HyphenDict`. Si potrebbe gestire la cosa creando una classe che erediti `ctypes.Structure` in modo da rappresentare il tipo. Tuttavia, dato che dobbiamo solo passare il puntatore alla struttura, senza dover accedere direttamente al suo contenuto, possiamo semplicemente dichiarare che la funzione restituisce un `ctypes.c_void_p` ("C void pointer" - puntatore C vuoto), che può puntare a qualsiasi tipo di variabile.

Queste tre righe (oltre a trovare e caricare in memoria la libreria, in primis) sono tutto quel che serve per ottenere un metodo `_load()` che carica un dizionario di sillabazione.

```
_unload = _LibHyphen.hnj_hyphen_free
_unload.argtypes = [ctypes.c_void_p] # HyphenDict *hdict
_unload.restype = None
```

Il codice riportato qui segue lo stesso schema del precedente. La funzione `hnj_hyphen_free()` richiede un solo argomento, un puntatore a una `HyphenDict struct`, ma

dovendo passare solo il puntatore, possiamo specificare come tipo di ritorno un puntatore `void` - che punterà in verità a `HyphenDict struct`. Questa funzione non ha alcun valore di ritorno; lo indichiamo specificandone il `restype` a `None` (se non specificassimo alcun `restype`, Python assumerebbe che la funzione restituisca un `int`).

```
_int_p = ctypes.POINTER(ctypes.c_int)
_char_p_p = ctypes.POINTER(ctypes.c_char_p)

_hyphenate = _LibHyphen.hnj_hyphen_hyphenate2
_hyphenate.argtypes = [
        ctypes.c_void_p,    # HyphenDict *hdict
        ctypes.c_char_p,    # const char *word
        ctypes.c_int,       # int word_size
        ctypes.c_char_p,    # char *hyphens   [non necessario]
        ctypes.c_char_p,    # char *hyphenated_word
        _char_p_p,          # char ***rep     [non necessario]
        _int_p,             # int **pos       [non necessario]
        _int_p]             # int **cut       [non necessario]
_hyphenate.restype = ctypes.c_int    # int
```

Questa è la funzione più complicata che dobbiamo utilizzare. L'argomento `hdict` è un puntatore a una `HyphenDict struct`, definito come puntatore C `void`. Quindi abbiamo la parola `word` da sillabare, passata sotto forma di puntatore a un blocco di byte, per cui utilizziamo un puntatore a char C. Questa è seguita da `word_size`, il numero di byte che compongono la parola, specificato come intero (`ctypes.c_int`). Infine abbiamo il buffer `hyphens` di cui non abbiamo bisogno, quindi la parola sillabata `hyphenated_word`, di nuovo definita come puntatore a carattere C. Non esiste alcun tipo `ctypes` per un puntatore doppio a un carattere (byte), per cui abbiamo creato un tipo personalizzato, `_char_p_p`, specificato come un puntatore a un puntatore a un carattere C. Abbiamo fatto qualcosa di analogo per i due puntatori a puntatori a interi.

In termini rigorosi non dobbiamo specificare alcun `restype`, perché il valore di ritorno della funzione è intero, ma preferiamo un approccio ridondante ed esplicito.

Abbiamo creato delle funzioni wrapper private per le funzioni della libreria di sillabazione, perché vogliamo isolare gli utenti del modulo dai dettagli di basso livello. A tal scopo, realizziamo una sola funzione pubblica, `hyphenate()`, che accetta una parola da sillabare, un dizionario e il carattere da usare come separatore. Per una maggiore efficienza, caricheremo in memoria ciascun dizionario di sillabazione una sola volta. E, ovviamente, ci assicureremo che tutti i dizionari di sillabazione vengano scaricati dalla memoria all'uscita del programma.

```
def hyphenate(word, filename, hyphen="-"):
    originalWord = word
    hdict = _get_hdict(filename)
    word = word.encode("utf-8")
    word_size = ctypes.c_int(len(word))
    word = ctypes.create_string_buffer(word)
    hyphens = ctypes.create_string_buffer(len(word) + 5)
    hyphenated_word = ctypes.create_string_buffer(len(word) * 2)
```

```
    rep = _char_p_p(ctypes.c_char_p(None))
    pos = _int_p(ctypes.c_int(0))
    cut = _int_p(ctypes.c_int(0))
    if _hyphenate(hdict, word, word_size, hyphens, hyphenated_word, rep,
            pos, cut):
        raise Error("hyphenation failed for '{}'".format(originalWord))
    return hyphenated_word.value.decode("utf-8").replace("=", hyphen)
```

La funzione inizia memorizzando un riferimento alla parola da sillabare in modo che possa essere usata in un eventuale messaggio d'errore, qualora si rendesse necessario. Quindi, otteniamo il dizionario di sillabazione: la funzione privata `_get_hdict()` restituisce un puntatore alla struct `HyphenDict` che corrisponde al nome di file fornito. Se il dizionario è già stato caricato, viene restituito il puntatore memorizzato precedentemente; altrimenti viene caricato il nuovo dizionario (una sola volta), il puntatore viene memorizzato per usi successivi e quindi restituito al chiamante.

La parola dev'essere passata alla funzione di sillabazione come blocco di byte codificati UTF-8, cosa ottenibile facilmente tramite il metodo `str.encode()`. Dobbiamo anche passare il numero di byte occupati dalla parola da sillabare: lo calcoliamo e convertiamo l'`int` dal formato Python al formato C. Non possiamo passare un oggetto grezzo `bytes` di Python a una funzione C, quindi creiamo un buffer stringa (nient'altro che un blocco di `char` C) per contenere la parola. La chiamata di `ctypes.create_string_buffer()` crea un blocco di `char` C a partire da un oggetto `bytes` della dimensione specificata. Anche se non intendiamo usare l'argomento `hyphens`, dobbiamo ugualmente prepararlo. La documentazione dice che dev'essere un puntatore a un blocco di `char` C la cui lunghezza è cinque più quella della parola (in byte). Perciò creiamo un blocco di `char` C. La parola sillabata verrà memorizzata nel blocco di `char` C che viene passato alla funzione, quindi dobbiamo crearne uno di dimensione sufficiente. La documentazione consiglia una dimensione doppia rispetto a quella della parola originale.

Non vogliamo usare gli argomenti `rep`, `pos` e `cut`, ma dobbiamo come al solito passare dei valori corretti per far lavorare la funzione. `rep` è un puntatore a un puntatore a un puntatore a un puntatore a un blocco di `char` C, per cui abbiamo creato un puntatore a un blocco vuoto (un puntatore nullo in C ovvero un puntatore che non punta a nulla) e abbiamo assegnato un puntatore a un puntatore a questo puntatore alla variabile `rep`. Per quanto rigurda gli argomenti `pos` e `cut`, abbiamo creato dei puntatori a puntatori a interi di valore pari a 0.

Una volta impostati tutti gli argomenti, richiamiamo la funzione privata `_hyphenate()` (dietro le quinte stiamo richiamando in realtà la funzione `hnj_hyphen_hyphenate2()` della

libreria di sillabazione), generando un errore se la funzione restituisce un valore diverso da zero (errore). In caso contrario, estraiamo i byte grezzi dalla parola sillabata utilizzando la proprietà `value` (che restituisce un blocco `bytes` il cui ultimo byte contiene `0x00`). Quindi decodifichiamo i byte utilizzando la codifica UTF-8 in una `str` e sostituiamo i segni di uguale (`=`) prodotti dalla libreria di sillabazione con il separatore fornito dall'utente (che per default è `-`). La stringa risultante viene quindi restituita come ritorno della funzione `hyphenate()`.

Notate come nelle funzioni C che usano `char *` e dimensione al posto di stringhe terminate da null, possiamo accedere ai byte tramite la proprietà `raw` invece che tramite `value`.

```
_hdictForFilename = {}

def _get_hdict(filename):
    if filename not in _hdictForFilename:
        hdict = _load(ctypes.create_string_buffer(
                filename.encode("utf-8")))
        if hdict is None:
            raise Error("failed to load '{}'".format(filename))
        _hdictForFilename[filename] = hdict
    hdict = _hdictForFilename.get(filename) if hdict is None:
        raise Error("failed to load '{}'".format(filename))
    return hdict
```

Questa funzione privata restituisce un puntatore a una struct `HyphenDict`, riutilizzando puntatori a dizionari già caricati in memoria. Se il nome del file non si trova già in `_hdictForFilename dict`, si tratta di un nuovo dizionario e dev'essere caricato. Dato che il nome del file viene passato sotto forma di un `const char *` (ovvero è immutabile), possiamo creare e passare direttamente un buffer stringa `ctypes`. Se la funzione `_load()` restituisce `None` il caricamento è fallito, e segnaliamo la cosa sollevando un'eccezione. In caso contrario, memorizziamo il puntatore per poterlo riutilizzare in seguito. Alla fine, sia che il dizionario sia stato caricato ora o precedentemente, cerchiamo di ottenerne il puntatore corrispondente per restituirlo al chiamante.

```
def _cleanup():
    for hyphens in _hdictForFilename.values():
        _unload(hyphens)

atexit.register(_cleanup)
```

`_hdictForFilename dict` contiene puntatori per tutti i dizionari di sillabazione caricati in memoria. Dobbiamo assicurarci di scaricarli prima che il programma termini. Creiamo una funzione privata `_cleanup()` che richiami la nostra funzione privata `_unload()` per ciascun puntatore a dizionario (e che a sua volta richiami la funzione di libreria `hnj_hyphen_free()`). Non ci preoccupiamo di ripulire la `_hdictForFilename dict` alla

fine perché `_cleanup()` viene richiamata solo alla conclusione del programma (per cui la `dict` verrà cancellata comunque). Ci assicuriamo che la chiamata a `_cleanup()` avvenga, registrandola come una funzione “in uscita” utilizzando la funzione `register()` del modulo `atexit` della libreria standard.

Abbiamo analizzato tutto il codice richiesto per creare una funzione `hyphenate()` in un modulo che fa uso di una libreria condivisa di sillabazione. L’uso di `ctypes` richiede alcuni accorgimenti (per esempio impostare i tipi corretti e inizializzare gli argomenti), ma apre il mondo del C e del C++ ai programmi Python.

Un utilizzo pratico di `ctypes` si ha quando si vuole scrivere del codice in cui la velocità è il fattore critico in C o C++, e lo si vuole inserire in una libreria condivisa, in modo che possa essere utilizzato sia da Python (via `ctypes`) che dai propri programmi C e C++. L’altro uso classico consiste nell’utilizzo di funzionalità rese disponibili mediante librerie condivise, anche se nella maggior parte delle situazioni è più facile trovare una libreria standard o un modulo di produttori esterni che richiami già la libreria in questione.

Il modulo `ctypes` offre moltissime altre funzionalità avanzate che non abbiamo spazio di trattare qui. E anche se è più complicato da usare rispetto a CFFI e Cython, può talvolta essere più comodo, dato che è già compreso all’interno di Python.

Usare Cython

La descrizione di Cython (`http://cython.org`) sul relativo sito web recita: “È un linguaggio di programmazione che rende la scrittura di estensioni C per il linguaggio Python semplice come il Python stesso”. Cython può essere utilizzato in tre modi diversi. Il primo consiste nel racchiudere codice C o C++, esattamente come con `ctypes`, anche se ovviamente l’uso di Cython sarà più semplice, specialmente per chi è già a suo agio con il C e il C++. Il secondo modo consiste nella compilazione di codice Python in codice C (ben più veloce). Questo già consentirebbe di ottenere un raddoppio delle prestazioni per il codice che usa la CPU in modo intensivo. Il terzo modo è simile al secondo, solo che invece di lasciare il codice com’è nel file `.pyx`, questo viene “Cythonizzato”, ovvero si sfruttano le estensioni del linguaggio offerte da Cython in modo che il risultato della compilazione sia codice C ancora più efficiente. In questo modo si possono raggiungere incrementi prestazionali di 100 volte o più nei processi CPU-bound.

Accesso a librerie C da Cython

In questo paragrafo creeremo il modulo `Hyphenate2`, che fornisce esattamente la stessa funzionalità del modulo `Hyphenate1.py` creato nel paragrafo precedente, solo che questa volta useremo Cython invece di `ctypes`. La versione con `ctypes` faceva uso di un solo file, `Hyphenate1.py`, mentre in Cython dobbiamo creare una cartella nella quale inserire quattro file.

Il primo file richiesto è `Hyphenate2/setup.py`. Questo minuscolo file di infrastruttura contiene un’unica istruzione che indica a Cython dove trovare la libreria di sillabazione e che cosa importare. Il secondo file è `Hyphenate2/init_.py`. Questo è un file opzionale che contiene un’unica istruzione che esporta la funzione pubblica `Hyphenate2.hyphenate()` e l’eccezione `Hyphenate2.Error`. Il terzo file è `Hyphenate2/chyphenate.pxd`; viene utilizzato per informare Cython della libreria di sillabazione e delle funzioni a cui vogliamo accedere. Il quarto file è `Hyphenate2/Hyphenate.pyx`; è un modulo Cython che useremo per implementare la funzione pubblica `hyphenate()` e le relative funzioni di supporto private. Esamineremo ciascuno di questi file uno per uno.

```
distutils.core.setup(name="Hyphenate2",
        cmdclass={"build_ext": Cython.Distutils.build_ext},
        ext_modules=[distutils.extension.Extension("Hyphenate",
                ["Hyphenate.pyx"], libraries=["hyphen"])])
```

Questo è il contenuto del file `Hyphenate2/setup.py`, escludendo le inclusioni. Fa uso del pacchetto `distutils` di Python.

NOTA

È probabilmente meglio installare il pacchetto Python `distribute` versione 0.6.28 o superiore, o ancora meglio il pacchetto `setuptools` versione 0.7 o superiore, `http://python-packaging-user-guide.readthedocs.org`. È richiesto uno strumento di installazione di pacchetti aggiornato per installare molte estensioni di terze parti, ivi comprese quelle utilizzate in questo libro.

`name` è opzionale. `cmdclass` dev'essere fornito come indicato precedentemente. La prima stringa fornita a `Extension()` è il nome che vogliamo abbia il modulo una volta compilato (per esempio `Hyphenate.so`). Questo viene seguito da un elenco di file `.pyx` che contengono il codice da compilare e, opzionalmente, da un elenco di librerie esterne C o C++. In questo esempio, ovviamente, abbiamo necessità della libreria di sillabazione.

Per comporre l'estensione, eseguite i comandi che seguono nella cartella contenente i file (per esempio `Hyphenate2`):

```
$ cd pipeg/Hyphenate2
$ python3 setup.py build_ext --inplace
running build_ext
cythoning Hyphenate.pyx to Hyphenate.c
building 'Hyphenate' extension
creating build
creating build/temp.linux-x86_64-3.3
...
```

Se avete più di un interprete Python sulla vostra macchina, dovrete specificare il percorso completo di quello che volete usare. In Python 3.1 questi comandi produrranno `Hyphenate.so`, nelle versioni successive verrà creata una libreria della versione specifica; per esempio, `Hyphenate.cpython-33m.so` nel caso di Python 3.3.

```
from Hyphenate2.Hyphenate import hyphenate, Error
```

Questo è il file `Hyphenate2/__init__.py` completo. Lo forniamo come piccola comodità per l'utente, in modo che sia possibile scrivere qualcosa `import Hyphenate2 as Hyphenate` per richiamare `Hyphenate.hyphenate()`. Altrimenti l'importazione sarebbe `import Hyphenate2.Hyphenate as Hyphenate`.

```
cdef extern from "hyphen.h":
    ctypedef struct HyphenDict:
        pass
    HyphenDict *hnj_hyphen_load(char *filename)
    void hnj_hyphen_free(HyphenDict *hdict)
    int hnj_hyphen_hyphenate2(HyphenDict *hdict, char *word,
            int word_size, char *hyphens, char *hyphenated_word,
            char ***rep, int **pos, int **cut)
```

Questo è il file `Hyphenate2/chyphenate.pxd`. Un file `.pxd` è richiesto ogni qual volta si voglia accedere a librerie esterne C o C++ da codice Cython.

La prima riga dichiara il nome del file header C o C++ che contiene le dichiarazioni delle funzioni e dei tipi a cui si vuole accedere. Il corpo, quindi, dichiara queste funzioni e i relativi tipi. Cython consente di accedere comodamente a struct C o C++ senza doverne dichiarare tutti i dettagli in precedenza. Questo è consentito solo quando si passa un puntatore a una struct senza dover accedere ai campi contenuti al suo interno; questa è la situazione più comune e si applica di certo alla libreria di sillabazione. Le dichiarazioni delle funzioni vengono essenzialmente copiate dal file header C o C++, omettendo il punto e virgola al termine di ogni riga. Cython usa questo file `.pxd` per creare un ponte di codice C tra il software compilato in Cython e la libreria esterna a cui il file `.pxd` fa riferimento.

Ora che abbiamo creato I file `setup.py`, `__init__.py` e `chyphenate.pxd`, siamo pronti per l'ultimo file: `Hyphenate.pyx`. Questo contiene il codice Cython, ovvero Python con estensioni Cython. Iniziamo con le inclusioni, poi analizzeremo ciascuna funzione singolarmente.

```
import atexit
cimport chyphenate
cimport cpython.pycapsule as pycapsule
```

Per assicurarci che i dizionari di sillabazione vengano scaricati dalla memoria all'uscita del programma, importiamo il modulo `atexit` della libreria standard. I file Cython possono importare moduli Python classici tramite `import` oltre a file Cython `.pxd` (ovvero contenitori per librerie C esterne) tramite `cimport`. Quindi, qui, importiamo `chyphenate.pxd` nel modulo `chyphenate`, e questo ci mette a disposizione il tipo `chyphenate.HyphenDict` e le tre funzioni della libreria di sillabazione.

Vogliamo creare un `dict` Python le cui chiavi sono i nomi di file dei dizionari di sillabazione e i valori i rispettivi puntatori a `chyphenate.HyphenDict`. Tuttavia, i `dict` Python non possono contenere puntatori (non sono un tipo Python). Fortunatamente Cython ci offre una soluzione: `pycapsule`. Questo modulo Cython può incapsulare un puntatore all'interno di un oggetto Python che - stavolta - può essere memorizzato in qualsiasi collezione Python. Come vedremo tra poco, `pycapsule` consente anche di estrarre il puntatore dall'oggetto Python.

```
def hyphenate(str word, str filename, str hyphen="-"):
    cdef chyphenate.HyphenDict *hdict = _get_hdict(filename)
    cdef bytes bword = word.encode("utf-8")
    cdef int word_size = len(bword)
    cdef bytes hyphens = b"\x00" * (word_size + 5)
    cdef bytes hyphenated_word = b"\x00" * (word_size * 2)
```

```
    cdef char **rep = NULL
    cdef int *pos = NULL
    cdef int *cut = NULL
    cdef int failed = chyphenate.hnj_hyphen_hyphenate2(hdict, bword, word_size, hyphens,
hyphenated_word, &rep, &pos, &cut)
    if failed:
        raise Error("hyphenation failed for '{}'".format(word))
    end = hyphenated_word.find(b"\x00")
    return hyphenated_word[:end].decode("utf-8").replace("=", hyphen)
```

Questa funzione è strutturalmente identica alla versione `ctypes` che abbiamo creato nel paragrafo precedente. La differenza più evidente è che forniamo tipi espliciti a tutti gli argomenti e a tutte le variabili. Questo non è strettamente richiesto da Cython, ma gli permette di effettuare delle ottimizzazioni per migliorare le prestazioni.

`hdict` è un puntatore a una `HyphenDict struct`, mentre `bword` contiene i byte codificati UTF-8 della parola da sillabare. `word_size int` viene creato facilmente. Per il blocco `hyphens` che non useremo dovremo comunque creare un buffer (un blocco di `char` C) grande a sufficienza - moltiplicando un byte null per la dimensione richiesta. Usiamo la stessa tecnica che abbiamo applicato per il buffer `hyphenated_word`.

Non utilizziamo gli argomenti `rep`, `pos` e `cut`, ma dobbiamo ugualmente passarli o la funzione non opererà correttamente. In tutti e tre i casi, li creiamo con la sintassi dei puntatori C (ovvero `cdef char **rep`) usando un livello in meno di reindirizzamento - un puntatore in meno, cioè un `*` in meno) di quanto sia effettivamente richiesto. Quindi, nella chiamata alla funzione, utilizzeremo l’operatore di indirizzo del C (`&`) per passare l’indirizzo, ottenendo indietro il livello di reindirizzamento mancante. Non possiamo passare un puntatore C null (`NULL`) per questi argomenti, perché la funzione li richiede non null, anche se alla fine non puntano a niente. Ricordate che in C `NULL` è un puntatore che non punta a nulla.

Una volta inizializzati opportunamente tutti gli argomenti, richiamiamo la funzione esportata dal modulo Cython `chyphenate` (per la precisione dal file `chyphenate.pxd`). Se la sillabazione ha successo, restituiamo la parola sillabata. Per far questo, dobbiamo scorrere il buffer `hyphenated_word` fino al primo carattere null e decodificare i caratteri risultanti come UTF-8 in una `str`, per sostituire il carattere di separazione della libreria (`=`) con quello specificato dall’utente (o con quello di default, che è `-`).

```
_hdictForFilename = {}

cdef chyphenate.HyphenDict *_get_hdict(
        str filename) except <chyphenate.HyphenDict*>NULL:
    cdef bytes bfilename = filename.encode("utf-8")
    cdef chyphenate.HyphenDict *hdict = NULL
    if bfilename not in _hdictForFilename:
        hdict = chyphenate.hnj_hyphen_load(bfilename)
        if hdict == NULL:
```

```
            raise Error("failed to load '{}'".format(filename))
        _hdictForFilename[bfilename] = pycapsule.PyCapsule_New(
                <void*>hdict, NULL, NULL)
    capsule = _hdictForFilename.get(bfilename)
    if not pycapsule.PyCapsule_IsValid(capsule, NULL):
        raise Error("failed to load '{}'".format(filename))
    return <chyphenate.HyphenDict*>pycapsule.PyCapsule_GetPointer(capsule,
        NULL)
```

Questa funzione privata è definita utilizzando `cdef` invece di `def`; questo significa che sarà una funzione Cython e non una funzione Python. Dopo `cdef` specifichiamo il tipo del valore di ritorno della funzione, in questo caso un puntatore a un `chyphenate.HyphenDict`. Quindi assegniamo un nome alla funzione come di consueto, seguito dagli argomenti, generalmente con i rispettivi tipi. In questo caso c'è un solo argomento stringa, il nome del file.

Dato che il tipo di ritorno è un puntatore, piuttosto che un oggetto Python (ovvero `object`), non sarebbe normalmente possibile riportare le eccezioni al chiamante. In effetti, ogni eccezione produrrebbe semplicemente la stampa di un messaggio di avvertimento, ma altrimenti l'eccezione verrebbe ignorata. Ma noi vogliamo che questa funzione sia in grado di sollevare eccezioni Python vere e proprie. Abbiamo ottenuto questo comportamento specificando un valore di ritorno che indica quando è avvenuta un'eccezione; nello specifico un puntatore null a `chyphenate.HyphenDict`.

La funzione inizia dichiarando un puntatore a `chyphenate.HyphenDict` contenente il valore null (cioè punta a nulla). Quindi verifichiamo se il nome del file contenente il dizionario di sillabazione è nella `_hdictForFilename dict`. Se non c'è, dobbiamo caricare un nuovo dizionario utilizzando la funzione `hnj_hyphen_load()` della libreria, resa disponibile attraverso il nostro modulo `chyphenate`. Se il caricamento va a buon fine, viene restituito un puntatore a `chyphenate.HyphenDict` che non sarà null. Facciamo un cast di questo puntatore in modo che sia un puntatore `void` (che può quindi puntare a qualsiasi cosa) e creiamo un nuovo `pycapsule.PyCapsule` per memorizzarlo. La sintassi `<type>` di Cython trasforma un tipo C in un diverso tipo C. Per esempio, la chiamata Cython `<int>(x)` converte un valore `x` (che dev'essere un numero o un `char` C) in un `int` C. Questa sintassi è simile a quella di Python `int(x)`, salvo che in Python `x` può essere un `int` o un `float Python` - o una `str` contenente un intero (per esempio `"123"`) - e restituisce un `int` Python.

Il secondo argomento di `pycapsule.PyCapsule_New()` è il nome da dare al puntatore incapsulato (come `char *` C), mentre il terzo è il puntatore a una funzione distruttrice. Non volendo impostare nessuno dei due, assegniamo a entrambi dei puntatori null. Quindi memorizziamo il puntatore incapsulato nel `dict` col nome del file come chiave.

Alla fine, sia che abbiamo caricato il dizionario di sillabazione nella stessa chiamata o no, cerchiamo di recuperare la capsula che contiene il relativo puntatore. Dobbiamo verificare che la capsula contenga un puntatore valido (ovvero non null) passando la capsula e il nome associato alla funzione `pycapsule.PyCapsule_IsValid()`. Al posto del nome passiamo un null, perché non abbiamo dato alcun nome alle nostre capsule. Se la capsula è valida, estraiamo il puntatore usando la funzione `pycapsule.PyCapsule_GetPointer()` - di nuovo, passando la capsula e un puntatore null al posto del nome - convertendo il puntatore da `void` a `chyphenate.HyphenDict`, che poi restituiamo.

```
def _cleanup():
    cdef chyphenate.HyphenDict *hdict = NULL
    for capsule in _hdictForFilename.values():
        if pycapsule.PyCapsule_IsValid(capsule, NULL):
            hdict = (<chyphenate.HyphenDict*>
                    pycapsule.PyCapsule_GetPointer(capsule, NULL))
            if hdict != NULL:
                chyphenate.hnj_hyphen_free(hdict)
atexit.register(_cleanup)
```

Quando il programma termina, vengono richiamate tutte le funzioni registrate tramite `atexit.register()`. In questo caso, la funzione richiama la funzione privata `_cleanup()` del nostro modulo. Tale funzione inizia dichiarando un puntatore a `chyphenate.HyphenDict` con valore null. Quindi scorre lungo i valori di `_hdictForFilename dict`, ciascuno dei quali è una capsula contenente un puntatore a un `chyphenate.HyphenDict` senza nome. Per ciascuna capsula valida che ha un puntatore non null, viene richiamata la funzione `chyphenate.hnj_hyphen_free()`.

Il wrapper Cython per la libreria condivisa di sillabazione è molto simile a quello della versione `ctypes`, tranne per il fatto che richiede una propria cartella e tre minuscoli file di supporto. Se vogliamo solo che Python acceda a librerie C e C++ già esistenti, `ctypes` da solo è sufficiente, anche se molti programmatori possono trovare Cython o CFFI più semplice da usare. Tuttavia Cython ha un’altra caratteristica: permette di scrivere codice Cython - ovvero Python con estensioni - che può essere compilato in codice C con tempi di esecuzione efficienti. Nel prossimo paragrafo ci occuperemo di questo meccanismo.

Scrivere moduli Cython per aumentare la velocità

Generalmente il codice Python viene eseguito a una velocità più che sufficiente, o la velocità del codice è limitata da fattori esterni (per esempio la latenza della rete) che nessuna ottimizzazione del codice potrà mai migliorare. Tuttavia, nei processi

che usano la CPU in modo intensivo, è possibile raggiungere la velocità del codice C compilato utilizzando la sintassi di Python e le estensioni Cython.

Prima di avviare qualsiasi tipo di ottimizzazione, è essenziale effettuare una profilazione del codice. La maggioranza dei programmi trascorre gran parte del loro tempo di esecuzione all'interno di una piccola porzione di codice, per cui non importa quanto si ottimizzi: se l'ottimizzazione non riguarda quella porzione, tutto diventa inutile. La profilazione ci consente di vedere esattamente dove si trovano i colli di bottiglia e di indirizzare gli sforzi di ottimizzazione al codice che li richiede. Inoltre rende possibile misurare gli effetti delle ottimizzazioni confrontando i profili prima e dopo.

Abbiamo visto nello studio del modulo `Image` (Capitolo 3) che la funzione `scale()` non era molto veloce. In questo paragrafo proveremo a ottimizzarla.

```
Scaling using Image.scale()...
        18875915 function calls in 21.587 seconds
    ncalls    tottime    percall    cumtime    percall    filename:lineno(function)
         1    0.000      0.000      21.587     21.587     <string>:1(<module>)
         1    1.441      1.441      21.587     21.587     __init__.py:305(scale)
    786432    7.335      0.000      19.187      0.000     __init__.py:333(_mean)
   3145728    6.945      0.000       8.860      0.000     __init__.py:370(argb_for_color)
    786432    1.185      0.000       1.185      0.000     __init__.py:399(color_for_argb)
         1    0.000      0.000       0.000      0.000     __init__.py:461(<lambda>)
         1    0.000      0.000       0.002      0.002     __init__.py:479(create_array)
         1    0.000      0.000       0.000      0.000     __init__.py:75(__init__)
```

Questo è un profilo del metodo (escluse le funzioni integrate che non possiamo ottimizzare) così come prodotto dal modulo `cProfile` della libreria standard (cfr. il programma d'esempio `benchmark_Scale.py`). Più di 21 secondi per ridimensionare una fotografia a colori di 2048×1536 (3.145.728 pixel) non è di certo indice di velocità, ed è facile capire dove viene speso tutto questo tempo: il metodo `_mean()` e i metodi statici `argb_for_color()` e `color_for_argb()`. Vogliamo un confronto reale con Cython, quindi in primo luogo abbiamo copiato il metodo `scale()` e i relativi metodi di supporto (`_mean()` e così via) nel modulo `Scale/Slow.py` trasformandoli in funzioni. Abbiamo quindi profilato il risultato.

```
Scaling using Scale.scale_slow()...
        9438727 function calls in 14.397 seconds
    ncalls    tottime    percall    cumtime    percall    filename:lineno(function)
         1    0.000      0.000      14.396     14.396     <string>:1(<module>)
         1    1.358      1.358      14.396     14.396     Slow.py:18(scale)
    786432    6.573      0.000      12.109      0.000     Slow.py:46(_mean)
   3145728    3.071      0.000       3.071      0.000     Slow.py:69(_argb_for_color)
    786432    0.671      0.000       0.671      0.000     Slow.py:77(_color_for_argb)
```

Senza il sovraccarico dell'orientazione dell'oggetto, la funzione `scale()` dimezza le chiamate (da 18 milioni a 9 milioni) ma ottiene solo un miglioramento prestazionale di una volta e mezza.

Tuttavia, avendo isolato le funzioni più pesanti, possiamo produrre una versione ottimizzata in Cython per vedere come si comporta.

Inseriamo il codice Cython nel modulo `Scale/Fast.pyx` e usiamo `cProfile` per profilarlo, scalando la stessa fotografia usata per le due prove precedenti.

```
Scaling using Scale.scale_fast()...
        4 function calls in 0.114 seconds
    ncalls    tottime    percall    cumtime    percall    filename:lineno(function)
    1         0.000      0.000      0.114      0.114      <string>:1(<module>)
    1         0.113      0.113      0.113      0.113      Scale.Fast.scale
```

Il modulo `cProfile` non può analizzare il metodo `Scale.Fast.scale()` perché non è Python: è stato compilato in C. Ma non importa, dato che produce un incremento di velocità di 189×! Ovviamente, riscalare una sola immagine può non essere rappresentativo, ma i test su una vasta gamma di immagini ha prodotto sempre miglioramenti che non sono mai scesi sotto 130× rispetto al metodo originale.

Questo impressionante miglioramento è stato raggiunto grazie a molti tipi di ottimizzazione, alcuni specifici della funzione `scale()` e delle relative funzioni ausiliarie, altri di carattere più generale. Riportiamo di seguito i più importanti contributi di Cython al miglioramento prestazionale della funzione Cython `scale()`.

- Copiare il file Python originale (per esempio `Slow.py`) in un file Cython (per esempio `Fast.pyx`) ha prodotto un incremento di 2×.
- Sostituire tutte le funzioni private Python con funzioni Cython C ha prodotto un ulteriore incremento di 3×.
- Usare la funzione `round()` della libreria C `libc` al posto della funzione integrata `round()` di Python ha prodotto un ulteriore incremento di 4× (queste due funzioni non sono sempre intercambiabili, dato che hanno comportamenti differenti, tuttavia producono gli stessi risultati quando vengono usate come nelle funzioni `scale()` e `_mean()`).
- Passare viste di memoria al posto di array produce un ulteriore incremento 3×.

Ulteriori incrementi minori sono stati ottenuti utilizzando tipi specifici per tutte le variabili, passando una `struct` al posto di un `object` Python, rendendo le funzioni più piccole di tipo inline e con ottimizzazioni convenzionali come il precalcolo degli offset.

Ora che abbiamo visto quanta differenza può fare Cython, esaminiamo il codice ottimizzato, ovvero il modulo `Fast.pyx` e in particolare le versioni Cythonizzate della funzione `scale()` e delle funzioni ausiliarie `_mean()`, `_argb_for_color()` e `_color_for_argb()`.

Il metodo `Image.scale()` originale è già stato discusso in precedenza, anche se la funzione riportata qui è la versione Cython della funzione `Scale.Slow.scale()` del modulo `Slow.py`. Esattamente la stessa considerazione può essere fatta per le funzioni `_mean()`e `_argb_for_color()`. Il codice nei metodi e nelle funzioni è praticamente identico. La sola differenza consiste nel fatto che i metodi accedono ai dati dei pixel tramite `self` e richiamano altri metodi, mentre le funzioni passano i dati dei pixel esplicitamente e richiamano altre funzioni.

Cominciamo con le inclusioni del file `Scale/Fast.pyx` e le dichiarazioni di supporto.

```
from libc.math cimport round
import numpy
cimport numpy
cimport cython
```

Cominciamo con l'importazione della funzione `round()` della libreria C `libc` per sostituire la funzione integrata di Python `round()`. Avremmo potuto ovviamente scrivere `cimport libc.math` e quindi utilizzare `libc.math.round()` per la funzione C e `round()` per la funzione Python, ove volessimo usarle entrambe. Quindi importiamo NumPy più il modulo `numpy.pxd` fornito assieme a Cython, che consente a Cython di accedere a NumPy a livello del C. Per la funzione `scale()` di Cython abbiamo deciso di richiedere NumPy come dipendenza, ovvia conseguenza della richiesta di elaborazione veloce di array. Vogliamo anche importare il modulo Cython `cython.pxd` per alcuni dei decoratori che fornisce.

```
_DTYPE = numpy.uint32
ctypedef numpy.uint32_t _DTYPE_t

cdef struct Argb:
    int alpha
    int red
    int green
    int blue

DEF MAX_COMPONENT = 0xFF
```

Le prime due righe definiscono due tipi - `_DTYPE` per Python e `_DTYPE_t` per il C - entrambi come alias di interi senza segno a 32 bit NumPy. Quindi creiamo una struct C di nome `Argb`, che contiene quattro interi indicizzati per nome (è l'equivalente C di `Argb = collections.namedtuple("Argb", "alpha red green blue")`). Creiamo anche una costante C utilizzando l'istruzione Cython `DEF`.

```
@cython.boundscheck(False)
def scale(_DTYPE_t[:] pixels, int width, int height, double ratio):
    assert 0 < ratio < 1
    cdef int rows = <int>round(height * ratio)
    cdef int columns = <int>round(width * ratio)
    cdef _DTYPE_t[:] newPixels = numpy.zeros(rows * columns, dtype=_DTYPE)
    cdef double yStep = height / rows
    cdef double xStep = width / columns
```

```
    cdef int index = 0
    cdef int row, column, y0, y1, x0, x1
    for row in range(rows):
        y0 = <int>round(row * yStep)
        y1 = <int>round(y0 + yStep) for column in range(columns):
        x0 = <int>round(column * xStep)
        x1 = <int>round(x0 + xStep)
        newPixels[index] = _mean(pixels, width, height, x0, y0, x1, y1)
        index += 1
    return columns, newPixels
```

La funzione `scale()` usa lo stesso algoritmo di `Image.scale()`, solo richiede un array monodimensionale di pixel come primo argomento, seguito dalle dimensioni dell'immagine e dal rapporto di scala. Abbiamo disattivato il controllo dei limiti, anche se ciò non ha incrementato le prestazioni in questo caso. L'array di pixel viene passato sotto forma di vista di memoria; questo meccanismo è molto più efficiente del passaggio di array di `numpy.ndarray` e non aggiunge alcun carico ulteriore a Python. Ovviamente sono possibili molte altre ottimizzazioni di programmazione grafica - per esempio assicurarsi che la memoria sia allineata a specifici blocchi di byte - ma qui ci concentriamo su Cython piuttosto che sulla grafica.

Come abbiamo anticipato precedentemente, la sintassi `<type>` serve a convertire un tipo in un altro in Cython. La creazione delle variabili è essenzialmente analoga a quella del metodo `Image.scale()` solo che qui usiamo i tipi C (`int` per i numeri interi e `double` per i numeri a virgola mobile). Possiamo ancora usare la solita sintassi Python; per esempio, i cicli `for ... in`.

```
@cython.cdivision(True)
@cython.boundscheck(False)
cdef _DTYPE_t _mean(_DTYPE_t[:] pixels, int width, int height, int x0,
        int y0, int x1, int y1):
    cdef int alphaTotal = 0
    cdef int redTotal = 0
    cdef int greenTotal = 0
    cdef int blueTotal = 0
    cdef int count = 0
    cdef int y, x, offset
    cdef Argb argb
    for y in range(y0, y1):
        if y >= height:
            break
        offset = y * width
        for x in range(x0, x1):
            if x >= width:
                break
            argb = _argb_for_color(pixels[offset + x])
            alphaTotal += argb.alpha
            redTotal += argb.red
            greenTotal += argb.green
            blueTotal += argb.blue
            count += 1
    cdef int a = <int>round(alphaTotal / count)
    cdef int r = <int>round(redTotal / count)
    cdef int g = <int>round(greenTotal / count)
    cdef int b = <int>round(blueTotal / count)
    return _color_for_argb(a, r, g, b)
```

I componenti cromatici di ciascun pixel dell'immagine riscalata sono la media dei componenti cromatici dei pixel dell'immagine originale che questi devono

rappresentare. I pixel originali vengono passati efficientemente a una vista di memoria, seguiti dalle dimensioni dell'immagine originale e dagli angoli di una regione rettangolare in cui fare la media dei componenti cromatici dei pixel.

Invece di effettuare il calcolo (y × larghezza) + x per ogni pixel, calcoliamo la prima parte (come scostamento `offset`) una sola volta per riga.

Per inciso, usando il decoratore `@cython.cdivision`, abbiamo indicato a Cython di usare l'operatore / del C piuttosto che quello di Python per rendere la funzione leggermente più veloce.

```
cdef inline Argb _argb_for_color(_DTYPE_t color):
    return Argb((color >> 24) & MAX_COMPONENT,
        (color >> 16) & MAX_COMPONENT, (color >> 8) & MAX_COMPONENT,
        (color & MAX_COMPONENT))
```

Questa funzione è posta inline, ovvero invece di aggiungere il costo computazionale di una chiamata a una funzione, il suo codice verrà inserito nel punto in cui verrebbe richiamata (all'interno della funzione `_mean()`) per renderne l'esecuzione più veloce possibile.

```
cdef inline _DTYPE_t _color_for_argb(int a, int r, int g, int b):
    return (((a & MAX_COMPONENT) << 24) | ((r & MAX_COMPONENT) << 16) |
            ((g & MAX_COMPONENT) << 8) | (b & MAX_COMPONENT))
```

Anche questa funzione è posta inline per migliorarne le prestazioni, dato che viene richiamata per ciascun pixel da scalare.

La direttiva Cython `inline` è una richiesta che viene normalmente onorata solo per piccole semplici funzioni come quelle usate qui. Notate, inoltre, che anche se l'uso di funzioni inline migliora le prestazioni in questo specifico esempio, la cosa non è vera in generale e in alcuni casi può addirittura degradarle. Questo avviene se il codice inline utilizza troppa della cache del processore. Come sempre, è bene effettuare una profilazione prima e dopo ciascuna ottimizzazione sulla macchina o sulle macchine in cui intendiamo installare il programma, per poter prendere una decisione informata sul mantenere o meno l'ottimizzazione.

Cython ha molte altre funzionalità, rispetto a quelle usate in questo esempio, e dispone di una documentazione molto esaustiva. Il principale svantaggio di Cython consiste nella necessità di un compilatore e di una suite di strumenti di supporto per ogni piattaforma su cui si vogliono rilasciare i moduli Cython. Ma una volta che si sono installati tali strumenti, Cython è in grado di velocizzare in maniera incredibile qualsiasi codice che usi la CPU intensivamente.

Caso di studio: un modulo di gestione immagini accelerato

Nel Capitolo 3 abbiamo presentato un caso di studio sul modulo Python `Image` originale. In questo paragrafo analizzeremo molto rapidamente il modulo `cyImage`, l'alternativa Cython che offre la gran parte delle funzionalità del modulo `Image` originale ma con tempi di esecuzione decisamente più rapidi.

Le due principali differenze tra `Image` e `cyImage` sono, in primo luogo che il primo importa ogni possibile modulo di gestione immagini sia disponibile per qualsiasi formato, mentre il secondo ha un set fisso di moduli specifici per ciascun formato, in secundis `cyImage` richiede NumPy, laddove `Image` userà NumPy solo se è disponibile e ricorrerà ad `array` in caso contrario.

La Tabella 5.1 confronta le prestazioni del modulo Cython `cyImage` rispetto al modulo `Image` originale nei programmi di ridimensionamento immagini. Ma perché l'uso di Cython produce solo un miglioramento di 8× (per core), quando la funzione Cython `scale()` produce un miglioramento di 130×?

Tabella 5.1 Confronto di prestazioni tra moduli Cython di ridimensionamento immagini.

Programma	Concorrenza	Cython	Secondi	Incremento prestazionale
`imagescale-s.py`	*Nessuna*	No	780	*Base di riferimento*
`imagescale-cy.py`	*Nessuna*	Sì	88	8,86×
`imagescale-m.py`	4 processi in un pool	No	206	3,79×
`imagescale.py`	4 processi in un pool	Sì	23	33,91×

In sostanza, se si utilizza Cython per la riscalatura, questa non richiede più praticamente alcun tempo di esecuzione, ma le immagini originali devono comunque essere caricate e quelle risultanti salvate. Cython non produce alcun miglioramento sostanziale nella gestione dei file, perché quella di Python (a partire da Python 3.1) è già scritta in C. Quindi abbiamo cambiato il profilo prestazionale da uno in cui il collo di bottiglia era il ridimensionamento a uno in cui è la gestione dei file - dove, purtroppo, abbiamo ben poco da ottimizzare.

Per costruire il modulo `cyImage` il primo passo consiste nel creare una cartella `cyImage` e copiarvi i moduli della cartella `Image`. Il secondo prevede di rinominare quelli che vogliamo "Cythonizzare": in questo caso `__init__.py` diventa `Image.pyx`, `Xbm.py` diventa `Xbm.pyx` e `Xpm.py` diventa `Xpm.pyx`. Dobbiamo anche creare nuovi file `__init__.py` e `setup.py`.

L’esperienza ha dimostrato che sostituendo il corpo del metodo `Image.Image.scale()` con quello della funzione `Scale.Fast.scale()` e analogamente per `Image.Image._mean()` con `Scale.Fast._mean()`, si produce un incremento prestazionale irrisorio. Il problema sembra essere che Cython velocizza le funzioni molto più di quanto non faccia per i metodi. Tenendo presente questo, abbiamo copiato il modulo `Scale.Fast.pyx` nella cartella `cyImage` rinominandolo in `_Scale.pyx`. Abbiamo quindi cancellato il metodo `Image.Image._mean()` e abbiamo modificato il metodo `Image.Image.scale()` in modo che passi tutti i propri compiti alla funzione `_Scale.scale()`. Questo ha prodotto il miglioramento di 130× che ci attendevamo, anche se - ovviamente - la velocizzazione complessiva è stata decisamente più contenuta, come abbiamo evidenziato in precedenza.

```
try:
    import cyImage as Image
except ImportError:
    import Image
```

Anche se `cyImage` non è un sostituto perfetto di `Image` (non ha alcun supporto per il formato PNG e richiede espressamente NumPy), nei casi in cui è sufficiente, si può utilizzare questo modello di importazione.

```
distutils.core.setup(name="cyImage",
        include_dirs=[numpy.get_include()],
        ext_modules=Cython.Build.cythonize("*.pyx"))
```

Questo è il corpo del file `cyImage/setup.py`, a parte le inclusioni. Indica a Cython dove trovare i file header di NumPy per compilare tutti i file `.pyx` che trova nella cartella `cyImage`.

```
from cyImage.cyImage.Image import (Error, Image, argb_for_color, rgb_for_color, color_for_argb,
color_for_rgb, color_for_name)
```

All’interno del modulo `Image` abbiamo inserito tutte le funzionalità generiche nel file `Image/__init__.py`, mentre per `cyImage` andiamo a inserirle nel file `cyImage/Image.pyx`. Abbiamo quindi creato questo file `cyImage/__init__.py` con una sola riga, che importa diversi oggetti compilati - un’eccezione, una classe e alcune funzioni - rendendole disponibili direttamente come `cyImage.Image.from_file()`, `cyImage.color_for_name()`e così via. Dato che è possibile utilizzare la clausola `as` nell’importazione, alla fine possiamo scrivere `Image.Image.from_file()`, `Image.Image.Image()` e così via.

Non stiamo ad analizzare i file `.pyx`, avendo già visto nel paragrafo precedente come trasformare codice Python in codice Cython. Ci soffermiamo invece sulle inclusioni del file `cyImage/Image.pyx` e sul nuovo metodo `cyImage.Image.scale()`.

```
import sys
from libc.math cimport round
from libc.stdlib
cimport abs import numpy
cimport numpy
cimport cython
import cyImage.cyImage.Xbm as Xbm
import cyImage.cyImage.Xpm as Xpm
import cyImage.cyImage._Scale as Scale
from cyImage.Globals import *
```

Abbiamo scelto di usare le funzioni `round()` e `abs()` del C al posto delle versioni Python. E invece di effettuare importazioni dinamiche, come succedeva per il modulo `Image`, qui importiamo direttamente i moduli specifici del formato utilizzato (ovvero `cyImage/Xbm.pyx` e `cyImage/Xpm.pyx` o, in realtà, le librerie C condivise che Cython compila al loro interno).

```
    def scale(self, double ratio):
        assert 0 < ratio < 1
        cdef int columns
        cdef _DTYPE_t[:] pixels
        columns, pixels = Scale.scale(self.pixels, self.width, self.height, ratio)
        return self.from_data(columns, pixels)
```

Questo è il codice del metodo `cyImage.Image.scale()` per intero. È brevissimo perché passa tutto il lavoro alla funzione `cyImage._Scale.scale()` (che è una copia della funzione `Scale.Fast.scale()` che abbiamo visto nel paragrafo precedente).

Usare Cython non è comodo come programmare in Python puro - per giustificare il lavoro extra, dovremmo sempre prima profilare il codice Python per trovare eventuali colli di bottiglia. Se i punti nevralgici sono la gestione dell'I/O sui file o la latenza della rete, Cython difficilmente produrrà qualche miglioramento (e potrebbe essere opportuno considerare di usare la concorrenza). Quando i problemi sono a carico della CPU, invece, Cython può produrre sostanziali miglioramenti prestazionali, per cui vale senz'altro la pena di installare il framework e configurare gli strumenti di compilazione per poterlo utilizzare.

Una volta profilato e identificato quel che si vuole ottimizzare, la cosa migliore consiste nel separare il codice lento, mettendolo in un modulo a parte e profilare nuovamente l'esecuzione, sincerandoci di aver isolato correttamente il problema. Quindi copiamo e rinominiamo il modulo da Cythonizzare (da `.py` a `.pyx`) e creiamo un file `setup.py` adatto (ed eventualmente un file `__init__.py` per comodità). Ancora una volta è tempo di profilazione, questa volta per verificare se Cython è in grado di produrre almeno il miglioramento atteso di 2×. Ora possiamo iniziare a Cythonizzare blocchi di codice ripetutamente e riprofilare: dichiarando tipi, utilizzando viste di memoria, sostituendo il corpo di metodi lenti con chiamate a funzioni Cythonizzate. Dopo ciascun ciclo di ottimizzazione, possiamo tornare indietro nel caso le modifiche

non comportino miglioramenti, mantenendo quelle vantaggiose - finché non arriviamo al livello di prestazioni desiderato o terminiamo le ottimizzazioni da provare.

Donald Knuth ha detto: “Dovremmo tralasciare le piccole ottimizzazioni, diciamo per il 97% del tempo: l’ottimizzazione prematura è la radice di tutti i mali” (“Structured Programming with go to Statements”, *ACM Journal Computing Surveys* Vol. 6, N°4, dicembre 1974, p. 268). Inoltre, nessuna ottimizzazione potrà mai correggere l’uso dell’algoritmo sbagliato. Ma se abbiamo usato l’algoritmo corretto e la profilazione ha rivelato la presenza di colli di bottiglia, `ctypes` e Cython sono due ottimi esempi di strumenti in grado di velocizzare drammaticamente il codice che usa intensamente la CPU.

Accedere a funzionalità all’interno di librerie condivise utilizzando la convenzione di chiamata del C tramite `ctypes` o Cython consente di scrivere programmi in Python (linguaggio di alto livello) che fanno uso di codice veloce e di basso livello. Inoltre, si ha sempre la possibilità di scrivere codice in C o C++ e accedervi tramite `ctypes`, Cython o addirittura direttamente attraverso l’interfaccia Python C. Se vogliamo migliorare le prestazioni dei processi che usano intensivamente la CPU, l’uso della concorrenza permette incrementi prestazionali proporzionali al numero di core impiegati.

Tuttavia, l’uso di codice veloce compilato dal C può produrre miglioramenti di 100× rispetto al codice Python puro. Cython offre il meglio del Python e del C: la comodità della forma e della sintassi di Python con la velocità del C e l’accesso alle relative librerie condivise.

Elaborazione di rete ad alto livello

La libreria standard di Python mette a disposizione un eccellente supporto per l'elaborazione di rete di ogni tipo, dal basso livello all'alto livello. Il supporto di rete a basso livello è fornito da moduli quali `socket`, `ssl`, `asyncore` e `asynchat`, quello di livello intermedio è fornito, per esempio, dal modulo `socketserver`. Il supporto di alto livello è fornito dai molti moduli che supportano vari protocolli Internet, tra cui in particolare i moduli `http` e `urllib`.

Esistono anche diversi moduli esterni che supportano l'elaborazione di rete, tra cui Pyro4 (oggetti remoti Python; `http://packages.python.org/Pyro4`), PyZMQ (binding Python per la libreria 0MQ basata sul linguaggio C; `http://zeromq.github.com/pyzmq`) e Twisted (`http://twistedmatrix.com`). Per chi è interessato solo a HTTP e HTTPS, il pacchetto esterno `requests` (`http://python-requests.org`) risulta facile da usare.

In questo capitolo esamineremo due moduli che offrono supporto per l'elaborazione di rete di alto livello: il modulo `xmlrpc` della libreria standard (*XML Remote Procedure Call*) e il modulo esterno RPyC (*Remote Python Call*; `http://rpyc.sourceforge.net`). Entrambi ci isolano da molti dettagli di livello basso e intermedio, e risultano potenti ma comodi da usare.

In questo capitolo presentiamo un server e due client, per `xmlrpc` e RPyC. I server e i client svolgono in sostanza gli stessi compito, perciò possiamo confrontare facilmente i due approcci. I server si occupano di gestire le letture di un contatore (per esempio un contatore del gas o dell'elettricità), e i client sono utilizzati dalle persone che eseguono le letture per chiedere contatori da leggere e per fornire letture o motivi per cui non è stato possibile effettuare una lettura.

La più importante differenza tra gli esempi di questo capitolo è che il server con `xmlrpc` non è concorrente, mentre quello con RPyC è concorrente. Come vedremo, queste differenze di implementazione hanno un impatto significativo sul modo in cui gestiamo i dati di cui i server hanno la responsabilità.

Per mantenere i server i più semplici possibile abbiamo separato la gestione delle letture dei contatori in un modulo a parte (`Meter.py`, non concorrente, e `MeterMT.py`, che supporta la concorrenza). Un altro vantaggio di questa separazione è che facilita il compito di capire come sostituire il modulo del contatore con un modulo

personalizzato in grado di gestire dati piuttosto diversi, e perciò rende i client e i server molto più facilmente adattabili ad altri scopi.

Scrivere applicazioni XML-RPC

Svolgere comunicazioni di rete utilizzando protocolli di basso livello significa che per ogni porzione di dati che vogliamo trasmettere dobbiamo impacchettare i dati, inviarli, spacchettarli all'altro capo e infine svolgere delle operazioni in risposta ai dati inviati. Questo processo può diventare rapidamente pesante e tendente all'errore. Una soluzione è quella di utilizzare una libreria RPC (*Remote Procedure Call*). Questo ci consente di inviare semplicemente un nome di funzione e gli argomenti (per esempio stringhe, numeri, date) lasciando il compito di impacchettare, inviare, spacchettare ed eseguire l'operazione (per esempio richiamare la funzione) alla libreria RPC. Un protocollo RPC standardizzato molto noto è XML-RPC; le librerie che implementano questo protocollo codificano i dati (nomi di funzione e relativi argomenti) in formato XML e utilizzano HTTP come meccanismo di trasporto.

La libreria standard di Python include i moduli `xmlrpc.server` e `xmlrpc.client`, che forniscono supporto per il protocollo. Il protocollo stesso è neutro rispetto al linguaggio di programmazione, perciò, anche se scriviamo un server XML-RPC in Python, tale server sarà accessibile a client XML-RPC scritti in qualsiasi linguaggio che supporti il protocollo in questione. È anche possibile scrivere client XML-RPC in Python che si connettano a server XML- RPC scritti in altri linguaggi.

Il modulo `xmlrpc` ci consente di utilizzare alcune estensioni specifiche di Python, per esempio per passare oggetti Python, ma facendo ciò ci si vincola all'utilizzo di client e server Python. L'esempio riportato nel seguito non utilizza questa caratteristica.

Un'alternativa più leggera a XML-RPC è JSON-RPC, che fornisce la stessa gamma di funzionalità ma utilizza un formato di dati molto più leggero (solitamente comporta molti byte in meno da inviare in rete). La libreria di Python include il modulo `json` per la codifica e la decodifica di dati Python in o da JSON, ma non fornisce moduli client o server JSON-RPC. Tuttavia sono disponibili numerosi moduli JSON-RPC Python (`http://en.wikipedia.org/wiki/JSON-RPC`). Un'altra alternativa, per i casi in cui abbiamo soltanto client e server Python, è quella di usare RPyC, come vedremo più avanti nel paragrafo dedicato alla scrittura di applicazioni RPyC.

Un wrapper per i dati

I dati che vogliamo far gestire a client e server sono incapsulati dal modulo `Meter.py`. Tale modulo fornisce una classe `Manager` che memorizza letture di contatore e fornisce metodi con cui i lettori possono effettuare il login, acquisire job e inviare

risultati. Questo modulo potrebbe facilmente essere sostituito da un altro per gestire dati del tutto diversi.

```
class Manager:

    SessionId = 0
    UsernameForSessionId = {}
    ReadingForMeter = {}
```

Il `SessionID` è utilizzato per fornire a ogni login effettuato con successo un ID di sessione univoco.

La classe mantiene anche due dizionari statici: uno con chiavi per ID di sessione e valori di nome utente, e l'altro con chiavi per numero del contatore e valori di lettura corrispondenti.

Nessuno di questi dati statici dev'essere thread-safe, perché il server `xmlrpc` non è concorrente. La versione `MeterMT.py` di questo modulo supporta la concorrenza, e vedremo come differisce da `Meter.py` nel paragrafo dedicato alla scrittura di applicazioni RPyC.

In un contesto più realistico, i dati probabilmente saranno memorizzati in un file DBM o in un database, che potrebbero entrambi essere utilizzati al posto del dizionario di dati per il contatore impiegato qui.

```
    def login(self, username, password):
        name = name_for_credentials(username, password)
        if name is None:
            raise Error("Invalid username or password")
        Manager.SessionId += 1
        sessionId = Manager.SessionId
        Manager.UsernameForSessionId[sessionId] = username
        return sessionId, name
```

Vogliamo che i lettori di contatori eseguano il login con nome utente e password prima di consentire loro di acquisire job o di inviare risultati.

Se il nome utente e la password sono corretti, restituiamo un ID di sessione univoco per l'utente e il nome reale dell'utente (da visualizzare per esempio nell'interfaccia utente). A ogni login effettuato con successo viene fornito un ID di sessione univoco, che viene aggiunto al dizionario `UsernameForSessionId`. Tutti gli altri metodi richiedono un ID di sessione valido.

```
_User = collections.namedtuple("User", "username sha256")

def name_for_credentials(username, password):
    sha = hashlib.sha256()
    sha.update(password.encode("utf-8"))
    user = _User(username, sha.hexdigest())
    return _Users.get(user)
```

Qesta funzione, quando viene richiamata, calcola l'hash SHA-256 della password fornita e, se nome utente e hash corrispondono a una voce nel dizionario privato `_Users` del modulo (non mostrato qui), restituisce il nome effettivo corrispondente, altrimenti restituisce `None`.

Il dizionario `_Users` contiene chiavi `_User` costituite da un nome utente (per esempio `carol`), un hash SHA-256 della password dell'utente e valori per il nome reale (per esempio "Carol Dent"). Questo significa che non vengono memorizzate le password effettive.

NOTA

L'approccio impiegato qui è comunque non sicuro; per renderlo sicuro dovremmo aggiungere a ciascuna password un testo che faccia da "sale" in modo che due password identiche non producano lo stesso valore di hash. Un'alternativa migliore è quella di usare il pacchetto esterno `passlib` (`http://code.google.com/p/passlib`).

```
def get_job(self, sessionId):
    self._username_for_sessionid(sessionId)
    while True: # Crea un contatore finto
        kind = random.choice("GE")
        meter = "{}{}".format(kind, random.randint(40000,
                99999 if kind == "G" else 999999))
        if meter not in Manager.ReadingForMeter:
            Manager.ReadingForMeter[meter] = None
            return meter
```

Una volta che i lettori di contatori hanno effettuato il login, possono richiamare questo metodo per ottenere il numero di un contatore da leggere. Il metodo inizia controllando che l'ID di sessione sia valido; se non lo è, il metodo `_username_for_sessionid()` solleverà un'eccezione `Meter.Error`.

Non abbiamo veramente un database di contatori da leggere, perciò creiamo un contatore finto ogni volta che un lettore chiede l'assegnazione di un lavoro. A questo scopo creiamo un numero di contatore (per esempio "E350718" o "G72168") e lo inseriamo nel dizionario `ReadingForMeter` con un valore di lettura `None` non appena creiamo un contatore finto che non è già nel dizionario.

```
def _username_for_sessionid(self, sessionId):
    try:
        return Manager.UsernameForSessionId[sessionId]
    except KeyError:
        raise Error("Invalid session ID")
```

Questo metodo restituisce il nome utente corrispondente all'ID di sessione dato oppure converte un generico `KeyError` per un ID di sessione non valido in un `Meter.Error` personalizzato.

Spesso è meglio utilizzare un'eccezione personalizzata anziché una integrata, perché in questo modo possiamo catturare le particolari eccezioni che ci aspettiamo

di trovare e non quelle più generiche che, ove non siano state catturate, rileverebbero errori nella logica del nostro codice.

```
    def submit_reading(self, sessionId, meter, when, reading, reason=""):
        if isinstance(when, xmlrpc.client.DateTime):
            when = datetime.datetime.strptime(when.value,
                    "%Y%m%dT%H:%M:%S")
        if (not isinstance(reading, int) or reading < 0) and not reason:
            raise Error("Invalid reading")
        if meter not in Manager.ReadingForMeter:
            raise Error("Invalid meter ID")
        username = self._username_for_sessionid(sessionId)
        reading = Reading(when, reading, reason, username)
        Manager.ReadingForMeter[meter] = reading
        return True
```

Questo metodo accetta un ID di sessione, un numero di contatore (per esempio “G72168”), la data e l’ora in cui è avvenuta la lettura, il valore di lettura (un intero positivo o -1 se non è stata ottenuta una lettura) e il motivo per cui non è stato possibile effettuare una lettura (che è una stringa non vuota nel caso di letture non effettuate con successo).

Possiamo impostare il server XML-RPC in modo da utilizzare tipi Python integrati, ma non viene fatto per default (e non l’abbiamo fatto) perché il protocollo XML-RPC è neutro rispetto al linguaggio. Questo significa che il nostro server XML-RPC potrebbe servire client scritti in qualsiasi linguaggio che supporti XML-RPC, non solo Python. Lo svantaggio di non usare tipi Python è che gli oggetti data/ora vengono passati come `xmlrpc.client.DateTime` anziché come `datetime.datetime`, perciò dobbiamo convertirli in `datetime.datetime` (un’alternativa sarebbe quella di passarli come stringhe data/ora nel formato ISO-8601).

Una volta che abbiamo preparato e controllato i dati, recuperiamo il nome utente corrispondente al lettore di contatori di cui è stato passato l’ID di sessione e lo utilizziamo per creare un oggetto `Meter.Reading`. Questo è semplicemente una tupla con nome:

```
Reading = collections.namedtuple("Reading", "when reading reason username")
```

Alla fine impostiamo la lettura del contatore. Restituiamo `True` (anziché il valore di default `None`), perché per default il modulo `xmlrpc.server` non supporta `None`, e vogliamo mantenere il nostro server neutro rispetto al linguaggio (RPyC può gestire qualsiasi valore di ritorno Python).

```
    def get_status(self, sessionId):
        username = self._username_for_sessionid(sessionId)
        count = total = 0
        for reading in Manager.ReadingForMeter.values():
            if reading is not None:
                total += 1
                if reading.username == username:
                    count += 1
        return count, total
```

Dopo che un lettore di contatori ha inviato una lettura, potrebbe voler conoscere il proprio stato, cioè quante letture ha fatto e il numero totale di letture che il server ha gestito a partire dal suo avvio. Questo metodo calcola questi numeri e li restituisce.

```
    def _dump(file=sys.stdout):
        for meter, reading in sorted(Manager.ReadingForMeter.items()):
            if reading is not None:
                print("{}={}@{}[{}]{}".format(meter, reading.reading, reading.when.isoformat()
[:16], reading.reason, reading.username), file=file)
```

Questo metodo è fornito puramente a scopo di debugging, in modo che possiamo controllare che tutte le letture di contatore svolte siano state registrate correttamente.

Le funzionalità offerte da `Meter.Manager` - un metodo `login()` e metodi per ottenere e impostare i dati - sono tipiche di una classe wrapper di dati che un server potrebbe utilizzare. Dovrebbe essere semplice sostituire questa classe con una progettata per dati completamente diversi, continuando comunque a usare gli stessi client e server descritti in questo capitolo. Occorre soltanto tenere presente che, se si utilizzano server concorrenti, occorre usare lock o classi thread-safe per qualsiasi dato condiviso, come vedremo più avanti in questo capitolo.

Scrivere server XML-RPC

Grazie al modulo `xmlrpc.server`, scrivere server XML-RPC personalizzati è molto facile. Il codice riportato in questo paragrafo è tratto da `meterserver-rpc.py`.

```
def main():
    host, port, notify = handle_commandline()
    manager, server = setup(host, port)
    print("Meter server startup at {} on {}:{}{}".format(
            datetime.datetime.now().isoformat()[:19], host, port, PATH))
    try:
        if notify:
            with open(notify, "wb") as file:
                file.write(b"\n")
        server.serve_forever()
    except KeyboardInterrupt:
        print("\rMeter server shutdown at {}".format(
                datetime.datetime.now().isoformat()[:19]))
        manager._dump()
```

Questa funzione ottiene il nome di host e il numero di porta, crea un `Meter.Manager` e un `xmlrpc.server.SimpleXMLRPCServer`, e inizia l’attività di server.

Se la variabile `notify` contiene un nome di file, il server crea il file corrispondente e vi scrive una singola riga nuova. Questo file non è utilizzato quando il server è avviato manualmente, ma come vedremo più avanti, se il server è avviato da un client GUI, questo gli passa un file di notifica. Il client GUI attende quindi finché il file è

stato creato, e a quel punto sa che il server è in esecuzione, quindi elimina il file in questione e comincia la comunicazione con il server.

Si può arrestare il server premendo Ctrl+C o inviandogli un segnale `INT` (per esempio con `kill -2 pid` su Linux), che l'interprete Python trasforma in un `KeyboardInterrupt`. Se il server viene arrestato in questa maniera, facciamo in modo che il gestore stampi le letture a scopo di ispezione (questa è la sola ragione per cui questa funzione necessita di accesso all'istanza del gestore).

```
HOST = "localhost"
PORT = 11002

def handle_commandline():
    parser = argparse.ArgumentParser(conflict_handler="resolve")
    parser.add_argument("-h", "--host", default=HOST,
            help="hostname [default %(default)s]")
    parser.add_argument("-p", "--port", default=PORT, type=int,
            help="port number [default %(default)d]")
    parser.add_argument("--notify", help="specify a notification file")
    args = parser.parse_args()
    return args.host, args.port, args.notify
```

Questa funzione è riportata qui soltanto perché utilizza `-h` (e `--host`) come opzioni per impostare il nome di host. Per default l'argomento `argparse` riserva le opzioni `-h` (e `--help`) per indicare di visualizzare la guida della riga di comando e terminare. Vogliamo invece assumere il controllo sull'uso di `-h` (lasciando stare `--help`), e lo facciamo impostando il gestore di conflitti del parser.

Sfortunatamente, quando è stato effettuato il porting di `argparse` in Python 3, è stata mantenuta la vecchia formattazione di Python 2 con `%`, anziché sostituirla con le parentesi tipiche di Python 3 come in `str.format()`. Alla luce di ciò, quando vogliamo includere valori di default nel testo di guida, dobbiamo scrivere `%(default)t`, dove `t` è il tipo del valore (`d` per intero decimale, `f` per virgola mobile, `s` per stringa).

```
def setup(host, port):
    manager = Meter.Manager()
    server = xmlrpc.server.SimpleXMLRPCServer((host, port),
            requestHandler=RequestHandler, logRequests=False)
    server.register_introspection_functions()
    for method in (manager.login, manager.get_job,
            manager.submit_reading, manager.get_status):
        server.register_function(method)
    return manager, server
```

Questa funzione è utilizzata per creare il gestore dei dati (il contatore) e il server. Il metodo `resister_introspection_functions()` rende disponibili ai client tre funzioni di introspezione: `system.listMethods()`, `system.methodHelp()` e `system.methodSignature()` (non sono utilizzate dai nostri client XML-RPC, ma potrebbero servire per il debugging di client più complessi). Ognuno dei metodi di gestione che vogliamo mettere a disposizione dei client deve essere registrato con il server, cosa che si ottiene

facilmente usando il metodo `register_function()` (cfr. il riquadro dedicato ai metodi bound e unbound nel Capitolo 2).

```
PATH = "/meter"

class RequestHandler(xmlrpc.server.SimpleXMLRPCRequestHandler):
    rpc_paths = (PATH,)
```

Il server contatore non necessita di svolgere particolari azioni di gestione delle richieste, perciò abbiamo creato il gestore di richieste più semplice possibile, che eredita `xmlrpc.server.SimpleXMLRPCRequestHandler` e ha un percorso univoco per identificare le richieste al server contatore.

Ora che abbiamo creato un server, possiamo creare i client per accedervi.

Scrivere client XML-RPC

In questo paragrafo esamineremo due diversi client: uno basato sulla console che presuppone che il server sia già in esecuzione, e un altro client GUI che utilizza un server in esecuzione o avvia il proprio server se non ne trova uno.

Un client XML-RPC basato su console

Prima di entrare nei dettagli del codice, esaminiamo una tipica sessione di console interattiva. Il server `meterserver-rpc.py` deve essere stato avviato prima che questa interazione abbia luogo.

```
$ ./meterclient-rpc.py
Username [carol]:
Password:
Welcome, Carol Dent, to Meter RPC
Reading for meter G5248: 5983
Accepted: you have read 1 out of 18 readings
Reading for meter G72168: 2980q
Invalid reading
Reading for meter G72168: 29801
Accepted: you have read 2 out of 21 readings
Reading for meter E445691:
Reason for meter E445691: Couldn’t find the meter
Accepted: you have read 3 out of 26 readings
Reading for meter E432365: 87712
Accepted: you have read 4 out of 28 readings
Reading for meter G40447: Reason for meter G40447:
$
```

L’utente Carol avvia un client contatore. Le viene chiesto di inserire il proprio nome utente o di premere Invio per accettare il valore di default (mostrato tra parentesi quadre), perciò preme Invio. Poi le viene chiesto di inserire la password, cosa che fa senza alcun riscontro visivo. Il server riconosce l’utente e le porge il benvenuto salutandola con il suo nome completo. Il client poi chiede al server un contatore da leggere e chiede a Carol di inserire una lettura. Se Carol inserisce un

numero, questo viene passato al server e normalmente viene accettato. Se Carol commette un errore (come accade per la seconda lettura), o se la lettura non è valida per qualche altro motivo, viene mostrata una notifica e la richiesta di inserire nuovamente la lettura. Ogni volta che una lettura (o una motivazione) è accettata, viene visualizzato il numero di letture effettuate da Carlo nella sessione corrente e il numero delle letture totali (incluse quelle effettuate da altre persone che utilizzano il server nello stesso tempo). Se Carol preme Invio senza inserire una lettura, le viene chiesto di specificare una motivazione per cui non è in grado di fornire una lettura. E se Carol non inserisce una lettura o una motivazione, il client termina.

```
def main():
    host, port = handle_commandline()
    username, password = login()
    if username is not None:
        try:
            manager = xmlrpc.client.ServerProxy("http://{}:{}{}".format(host, port, PATH))
            sessionId, name = manager.login(username, password)
            print("Welcome, {}, to Meter RPC".format(name))
            interact(manager, sessionId)
        except xmlrpc.client.Fault as err:
            print(err)
        except ConnectionError as err:
            print("Error: Is the meter server running? {}".format(err))
```

Questa funzione inizia determinando il nome di host del server e il numero di porta (o i rispettivi valori di default) e poi ottiene il nome utente e la password dell'utente. Crea quindi un proxy (`manager`) per l'istanza `Meter.Manager` usata dal server (abbiamo trattato il pattern Proxy nel paragrafo corrispondente del Capitolo 2).

Una volta che il gestore proxy è stato creato, utilizziamo il proxy per effettuare il login e poi per iniziare a interagire con il server. Se non c'è alcun server in esecuzione, otteniamo un'eccezione `ConnectionError` (o `socket.error` nelle versioni precedenti Python 3.3).

```
def login():
    loginName = getpass.getuser()
    username = input("Username [{}]: ".format(loginName))
    if not username:
        username = loginName
    password = getpass.getpass()
    if not password:
        return None, None
    return username, password
```

La funzione `getuser()` del modulo `getpass` restituisce il nome utente per l'utente attualmente collegato, e lo utilizziamo come nome utente di default. La funzione `getpass()` richiede una password e non visualizza quanto viene inserito. `input()` e `getpass.getpass()` restituiscono stringhe senza caratteri di nuova riga in coda.

```
def interact(manager, sessionId):
    accepted = True
    while True:
        if accepted:
            meter = manager.get_job(sessionId)
```

```
            if not meter:
                print("All jobs done")
                break
        accepted, reading, reason = get_reading(meter)
        if not accepted:
            continue
        if (not reading or reading == -1) and not reason:
            break
        accepted = submit(manager, sessionId, meter, reading, reason)
```

Se il login ha successo, questa funzione è richiamata per gestire l'interazione client-server, che consiste nell'acquisire ripetutamente un job dal server (un contatore da leggere), nell'ottenere una lettura o una motivazione dall'utente, e nell'inviare i dati al server, finché l'utente non inserisce né una lettura né una motivazione.

```
def get_reading(meter):
    reading = input("Reading for meter {}: ".format(meter))
    if reading:
        try:
            return True, int(reading), ""
        except ValueError:
            print("Invalid reading")
            return False, 0, ""
    else:
        return True, -1, input("Reason for meter {}: ".format(meter))
```

Questa funzione deve gestire tre casi: quello in cui l'utente inserisce una lettura valida (un intero), quello in cui inserisce una lettura non valida e quello in cui non inserisce alcuna lettura. Se non è inserita alcuna lettura, l'utente inserisce una motivazione o non la inserisce (nell'ultimo caso significa che ha terminato).

```
def submit(manager, sessionId, meter, reading, reason):
    try:
        now = datetime.datetime.now()
        manager.submit_reading(sessionId, meter, now, reading, reason)
        count, total = manager.get_status(sessionId)
        print("Accepted: you have read {} out of {} readings".format(count, total))
        return True
    except (xmlrpc.client.Fault, ConnectionError) as err:
        print(err)
        return False
```

Ogni volta che viene ottenuta una lettura o una motivazione, questa funzione è utilizzata per inviarla al server tramite il gestore proxy. Una volta che la lettura o la motivazione è stata inviata, la funzione chiede i dati di stato (cioè il numero di letture inviate dall'utente e il numero di letture inviate in totale da quando il server è stato avviato).

Il codice del client è più lungo di quello del server, ma molto semplice. E poiché utilizziamo XML-RPC, il client potrebbe essere scritto in qualsiasi linguaggio che supporti il protocollo. È anche possibile scrivere client che utilizzino tecnologie diverse per l'interfaccia utente, come Urwid (`http://excess.org/urwid`) per interfacce utente sulla console Unix o un toolkit GUI come Tkinter.

Un client XML-RPC con GUI

La programmazione di GUI con Tkinter è introdotta nel Capitolo 7, perciò consigliamo a chi non ha familiarità con Tkinter di leggere prima tale capitolo e tornare poi qui. In questo paragrafo ci concentreremo soltanto sugli aspetti del programma GUI `meter-rpc.pyw` che riguardano l'interazione con il server del contatore. Il programma è mostrato nella Figura 6.1.

```
class Window(ttk.Frame):

    def __init__(self, master):
        super().__init__(master, padding=PAD)
        self.serverPid = None
        self.create_variables()
        self.create_ui()
        self.statusText.set("Ready…")
        self.countsText.set("Read 0/0")
        self.master.after(100, self.login)
```

Quando viene creata la finestra principale, impostiamo `None` come PID (*Process ID*) del server e richiamiamo il metodo `login()` 100 millisecondi dopo che la finestra principale è stata costruita. Questo lascia a Tkinter il tempo per disegnare la finestra principale, e prima che l'utente possa interagire con essa, viene creata una finestra di login modale. Una finestra modale di applicazione è l'unica finestra con cui l'utente può interagire per una data applicazione. Ciò significa che, anche se l'utente può vedere la finestra principale, non può utilizzarla finché non ha effettuato il login e la finestra di login modale è stata superata.

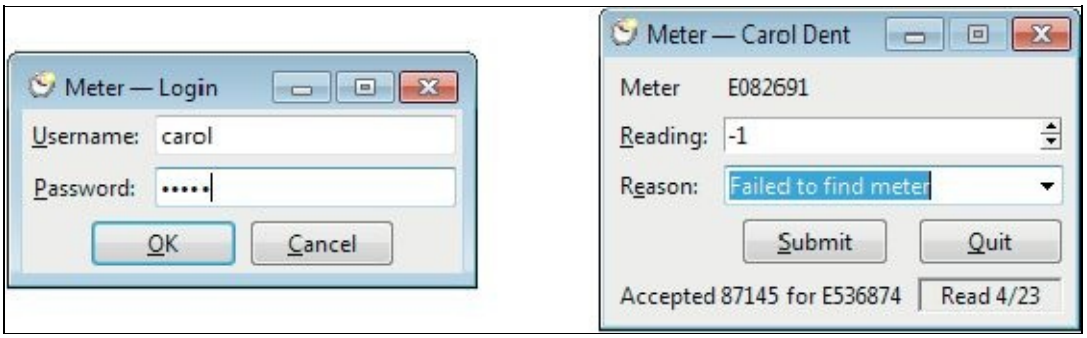


Figura 6.1 La finestra per il login e la finestra principale per l'applicazione GUI del contatore XML-RPC su Windows.

```
class Result:

    def __init__(self):
        self.username = None
        self.password = None
        self.ok = False
```

Questa classe (tratta da `MeterLogin.py`) è utilizzata per contenere i risultati dell'interazione dell'utente con la finestra di login modale. Passando alla finestra di dialogo un riferimento a un'istanza `Result`, possiamo assicurarci di poter accedere a ciò che l'utente ha inserito anche dopo che la finestra di dialogo è stata chiusa ed eliminata.

```
def login(self):
        result = MeterLogin.Result()
        dialog = MeterLogin.Window(self, result)
        if result.ok and self.connect(result.username, result.password):
                self.get_job()
        else:
            self.close()
```

Questo metodo crea un oggetto risultato e poi una finestra di login modale per l'applicazione. La chiamata di `MeterLogin.Window()` visualizza la finestra di login e blocca tutto finché non viene chiusa. Finché questa finestra è visualizzata, l'utente non può interagire con altre finestre dell'applicazione, perciò deve inserire nome utente e password e fare clic su *OK* oppure annullare facendo clic su *Cancel*.

Dopo che l'utente ha fatto clic su uno dei pulsanti, la finestra viene chiusa (ed eliminata). Se l'utente ha fatto clic su *OK* (cosa possibile soltanto se ha inserito un nome utente non vuoto e una password non vuota), viene effettuato un tentativo di connessione al server e viene ottenuto il primo job. Se l'utente ha annullato il login o la connessione è fallita, la finestra principale viene chiusa (ed eliminata) e l'applicazione termina.

```
    def connect(self, username, password):
        try:
            self.manager = xmlrpc.client.ServerProxy("http://{}:{}{}"
                    .format(HOST, PORT, PATH))
            name = self.login_to_server(username, password)
            self.master.title("Meter \u2014 {}".format(name))
            return True
        except (ConnectionError, xmlrpc.client.Fault) as err:
            self.handle_error(err)
            return False
```

Non appena l'utente ha inserito il suo nome utente e la sua password, viene richiamato questo metodo, che inizia creando un proxy per l'istanza `Meter.Manager` del server e poi tenta di effettuare il login. Dopo questa fase, il metodo modifica il titolo dell'applicazione impostando il nome dell'applicazione stessa, un *trattino em* (—, codice Unicode `U+2014`) e il nome dell'utente, quindi restituisce `True`.

Se si verifica un errore, viene visualizzata una finestra di messaggio con il testo dell'errore e viene restituito `False`.

```
    def login_to_server(self, username, password):
        try:
            self.sessionId, name = self.manager.login(username, password)
        except ConnectionError:
            self.start_server()
            self.sessionId, name = self.manager.login(username, password)
        return name
```

Se un server contatore è già in esecuzione, il tentativo di connessione iniziale avrà successo e saranno ottenuti l'ID di sessione e il nome utente. Se invece il tentativo di login fallisce a causa di un `ConnectionError`, l'applicazione assume che il server non sia in esecuzione e tenta di avviarlo, quindi tenta di effettuare il login una seconda volta.

Se anche il secondo tentativo fallisce, l'errore `ConnectionError` viene propagato al chiamante (`self.login()`), che lo cattura e lo visualizza all'utente in una finestra di messaggio, quindi l'applicazione termina.

```
SERVER = os.path.join(os.path.dirname(os.path.realpath(__file__)),"meterserver-rpc.py")
```

Questa costante imposta il nome del server con il percorso completo. Si assume che il server si trovi nella stessa directory del client GUI. Naturalmente è più comune la situazione in cui il server si trova su una macchina e il server su un'altra, tuttavia alcune applicazioni sono create in due parti separate - server e client - destinate a trovarsi sulla stessa macchina.

Lo schema di applicazione in due parti è utile quando vogliamo isolare del tutto la funzionalità di un'applicazione dalla sua interfaccia utente. Questo approccio ha lo svantaggio che occorre fornire due eseguibili anziché uno solo, e che comporta un certo sovraccarico per la rete, ma questo non dovrebbe essere apprezzabile dall'utente se client e server si trovano sulla stessa macchina. I vantaggi sono che il client e il server possono essere sviluppati in modo indipendente, e che risulta molto più facile effettuare il porting di tali applicazioni su nuove piattaforme, poiché il codice può essere scritto usando codice indipendente dalla piattaforma, e il lavoro di porting può concentrarsi quasi interamente sul client. Inoltre, in questo modo è possibile trarre vantaggio dalle nuove tecnologie per l'interfaccia utente (per esempio un nuovo toolkit GUI) semplicemente effettuando il porting del client. Un altro potenziale vantaggio è un livello di sicurezza a grana più fine; per sempio, il server può essere eseguito con permessi specifici e limitati, mentre il client viene eseguito semplicemente con i permessi dell'utente.

```
    def start_server(self):
        filename = os.path.join(tempfile.gettempdir(),
                "M{}.$$$".format(random.randint(1000, 9999)))
        self.serverPid = subprocess.Popen([sys.executable, SERVER, "--host", HOST, "--port",
str(PORT), "--notify", filename]).pid
        print("Starting the server…")
        self.wait_for_server(filename)
```

Il server viene avviato utilizzando la funzione `subprocess.Popen()`. Questo particolare utilizzo significa che il sottoprocesso (cioè il server) è avviato senza bloccaggio.

Se eseguissimo un programma normale (un sottoprocesso) che dovremmo terminare, potremmo attendere che finisca. Ma in questo caso dobbiamo avviare un server che non terminerà finché il nostro client non farà altrettanto, perciò non è possibile attendere. Inoltre, dobbiamo fornire al server una possibilità di avviarsi, poiché non possiamo continuare a tentare di effettuare login finché non lo troviamo in esecuzione. Esiste una soluzione semplice: creiamo un nome di file pseudocasuale

e avviamo il server passando tale nome come argomento di notifica. Possiamo quindi attendere che il server si avvi e creare il file di notifica per consentire al client di sapere che il server è pronto.

```
    def wait_for_server(self, filename):
        tries = 100
        while tries:
            if os.path.exists(filename):
                os.remove(filename)
                break
            time.sleep(0.1)  # Fornisce al server la possibilità di avviarsi
            tries -= 1
        else:
            self.handle_error("Failed to start the RPC Meter Server")
```

Questo metodo blocca (o congela) l’interfaccia utente per un massimo di 10 secondi (100 tentativi × 0.1 secondi), anche se in pratica l’attesa è quasi sempre di una frazione di secondo. Non appena il server crea il file di notifica, il client elimina il file e riprende l’elaborazione di eventi; in questo caso, il tentativo di effettuare il login dell’utente usando le credenziali fornite, e poi la visualizzazione della finestra principale pronta per l’inserimento di letture del contatore. Se il server fallisce l’avvio, il ciclo `while` termina senza un `break` e viene eseguita la sua clausola `else`.

Il polling non è l’approccio ideale, soprattutto in un’applicazione GUI, ma poiché vogliamo ottenere una soluzione indipendente dalla piattaforma e l’applicazione non può lavorare senza che il server sia disponibile, in questo caso rappresenta l’approccio più ragionevole che possiamo adottare.

```
    def get_job(self):
        try:
            meter = self.manager.get_job(self.sessionId)
            if not meter:
                messagebox.showinfo("Meter \u2014 Finished",
                    "All jobs done", parent=self)
                self.close()
            self.meter.set(meter)
            self.readingSpinbox.focus()
        except (xmlrpc.client.Fault, ConnectionError) as err:
            self.handle_error(err)
```

Una volta che il login al server è stato effettuato con successo (con il server avviato dall’applicazione, se necessario, durante questo processo), viene richiamato questo metodo per ottenere il primo job da svolgere. La variabile `self.meter` è di tipo `tkinter.StringVar` ed è associata all’etichetta che visualizza il numero del contatore.

```
    def submit(self, event=None):
        if self.submitButton.instate((tk.DISABLED,)):
            return
        meter = self.meter.get()
        reading = self.reading.get()
        reading = int(reading) if reading else -1
        reason = self.reason.get()
        if reading > -1 or (reading == -1 and reason and reason != "Read"):
            try:
                self.manager.submit_reading(self.sessionId, meter, datetime.datetime.now(),
reading, reason) self.after_submit(meter, reading, reason)
```

```
        except (xmlrpc.client.Fault, ConnectionError) as err:
            self.handle_error(err)
```

Questo metodo è richiamato ogni volta che l'utente fa clic sul pulsante *Submit* - cosa che l'applicazione consente soltanto se la lettura è diversa da zero o la motivazione non è vuota. Il contatore, la lettura (come `int`) e la motivazione sono tutti ottenuti dai widget dell'interfaccia utente e poi inviati al server tramite il gestore proxy. Se la lettura inviata è accettata, viene richiamato il metodo `after_submit()`; altrimenti, l'errore è trasmesso al metodo `handle_error()`.

```
def after_submit(self, meter, reading, reason):
    count, total = self.manager.get_status(self.sessionId)
    self.statusText.set("Accepted {} for {}".format(
            reading if reading != -1 else reason, meter))
    self.countsText.set("Read {}/{}".format(count, total))
    self.reading.set(-1)
    self.reason.set("")
    self.get_job()
```

Questo metodo chiede al gestore proxy lo stato corrente e aggiorna le etichette per lo stato e i conteggi, inoltre effettua il reset di lettura e motivazione e chiede al gestore un altro job.

```
def handle_error(self, err):
    if isinstance(err, xmlrpc.client.Fault):
        err = err.faultString
    messagebox.showinfo("Meter \u2014 Error",
            "{}\nIs the server still running?\n"
            "Try Quitting and restarting.".format(err), parent=self)
```

Se si verifica un errore, viene richiamato questo metodo, che visualizza l'errore in una finestra di messaggio modale con un singolo pulsante *OK*.

```
def close(self, event=None):
    if self.serverPid is not None:
        print("Stopping the server…")
        os.kill(self.serverPid, signal.SIGINT)
        self.serverPid = None
    self.quit()
```

Quando l'utente chiude l'applicazione, controlliamo se quest'ultima ha avviato direttamente il server contatore o ne ha usato uno già in esecuzione. Nel primo caso l'applicazione termina il server inviandogli un interrupt (che Python trasformerà in un'eccezione `KeyboardInterrupt`).

La funzione `os.kill()` invia un segnale (una delle costanti del modulo `signal`) al programma con il dato ID di processo. Questa funzione è compatibile solo con Unix per Python 3.1, ma funziona anche su Windows a partire da Python 3.2.

Il client di console, `meterclient-rpc.py`, contiene circa 100 righe di codice. Il client GUI, `meter-rpc.pyw`, contiene circa 250 righe (più altre 100 per la finestra di login `MeterLogin.py`). Entrambi sono facili da usare e altamente portabili, e grazie al supporto

dei temi di Tkinter, il client GUI ha un look and feel nativo sia su OS X che su Windows.

Scrivere applicazioni RPyC

Se stiamo scrivendo server Python e client Python, anziché utilizzare un protocollo verboso come XML-RPC, possiamo sfruttare un protocollo specifico di Python. Esistono molti pacchetti che offrono RPC da Python a Python, ma nel seguito abbiamo scelto di utilizzare RPyC (`http://rpyc.sourceforge.net`). Questo modulo offre due modalità di utilizzo: quella “classica” e la più recente “basata sul servizio”. Noi utilizzeremo quella più recente basata sul servizio.

Per default i server RPyC sono concorrenti, perciò non possiamo utilizzare il wrapper di dati non concorrente (`Meter.py`) descritto nel paragrafo precedente. Utilizzeremo invece un nuovo modulo `MeterMT.py` che introduce due nuove classi, `ThreadSafeDict` e `_MeterDict`, e ha una classe `Manager` modificata che utilizza questi dizionari anziché i `dict` standard.

Un wrapper per i dati thread-safe

Il modulo `MeterMT` contiene una classe `Manager` che supporta la concorrenza e due dizionari thread-safe. Iniziamo a esaminare i dati e i metodi della classe `Manager` con particolare riguardo alle differenze rispetto alla classe `Meter.Manager` descritta nel paragrafo precedente.

```
class Manager:

    SessionId = 0
    SessionIdLock = threading.Lock()
    UsernameForSessionId = ThreadSafeDict()
    ReadingForMeter = _MeterDict()
```

Per supportare la concorrenza, la classe `MeterMT.Manager` deve utilizzare dei lock per serializzare l’accesso ai suoi dati statici. Per gli ID di sessione utilizziamo un lock direttamente, me per i due dizionari utilizziamo dizionari thread-safe personalizzati che vedremo tra breve.

```
    def login(self, username, password):
        name = name_for_credentials(username, password)
        if name is None:
            raise Error("Invalid username or password")
        with Manager.SessionIdLock:
            Manager.SessionId += 1
            sessionId = Manager.SessionId
        Manager.UsernameForSessionId[sessionId] = username
        return sessionId, name
```

Questo metodo differisce dall’originale visto in precedenza soltanto perché incrementiamo e assegniamo l’ID di sessione statico nel contesto di un lock. Senza il lock, sarebbe possibile che, per esempio, il thread A incrementasse l’ID di sessione, e

poi il thread B facesse lo stesso, e poi i thread A e B leggerebbero entrambi lo stesso valore doppiamente incrementato, anziché ottenere un ID di sessione univoco.

```
    def get_status(self, sessionId):
        username = self._username_for_sessionid(sessionId)
        return Manager.ReadingForMeter.status(username)
```

Questo metodo ora trasferisce quasi tutto il suo lavoro a un metodo `_MeterDict.status()` personalizzato, che vedremo più avanti.

```
    def get_job(self, sessionId):
        self._username_for_sessionid(sessionId)
        while True:  # Create fake meter
            kind = random.choice("GE")
            meter = "{}{}".format(kind, random.randint(40000,
                    99999 if kind == "G" else 999999))
            if Manager.ReadingForMeter.insert_if_missing(meter):
                return meter
```

Questo metodo differisce dalla versione precedente per le ultime due righe. Vogliamo controllare se il contatore finto è presente nel dizionario, e se non lo è, vogliamo inserirlo con un valore `None` per la lettura iniziale. In questo modo ci assicuriamo che non possa essere riutilizzato. Nel caso precedente abbiamo effettuato il controllo e l'inserimento con due istruzioni separate, ma non possiamo fare ciò in un contesto concorrente, perché è possibile che uno o più altri thread siano eseguiti tra un'istruzione e l'altra. Perciò, ora trasferiamo il lavoro a un metodo personalizzato `_MeterDict.insert_if_missing()` che indica se l'inserimento è stato effettuato.

```
    def submit_reading(self, sessionId, meter, when, reading, reason=""):
        if (not isinstance(reading, int) or reading < 0) and not reason:
            raise Error("Invalid reading")
        if meter not in Manager.ReadingForMeter:
            raise Error("Invalid meter ID")
        username = self._username_for_sessionid(sessionId)
        reading = Reading(when, reading, reason, username)
        Manager.ReadingForMeter[meter] = reading
```

Questa versione è molto simile a quella di XML-RPC, l'unica differenza è che ora non dobbiamo convertire il valore di data/ora `when` e non dobbiamo restituire `True`, poiché un valore di ritorno implicito `None` è perfettamente accettabile per RPyC.

Un semplice dizionario thread-safe

Se utilizziamo CPython (la versione standard di Python implementata in C), in teoria il GIL (*Global Interpreter Lock*) fa apparire i `dict` thread-safe, perché l'interprete Python può eseguire un solo thread per volta (indipendentemente dal numero di core), e quindi le singole chiamate di metodo sono eseguite come azioni atomiche. Tuttavia, questo non aiuta quando abbiamo la necessità di richiamare due o più metodi `dict` in un'unica azione atomica. E in ogni caso, non dobbiamo affidarci a

questo dettaglio di implementazione; dopo tutto, altre implementazioni di Python (per esempio Jython e IronPython) non hanno un GIL, perciò non si può assumere che i loro metodi `dict` siano eseguiti in modo atomico.

Se vogliamo un dizionario veramente thread-safe, dobbiamo usarne uno esterno o crearne uno personalizzato. Creare un dizionario non è difficile, perché prendiamo un `dict` esistente e forniamo l'accesso attraverso i nostri metodi thread-safe. Nel seguito esamineremo `ThreadSafeDict`, un dizionario thread-safe che mette a disposizione un sottoinsieme dell'interfaccia di `dict` sufficiente per fornire dizionari contatore.

```
class ThreadSafeDict:

    def __init__(self, *args, **kwargs):
        self._dict = dict(*args, **kwargs)
        self._lock = threading.Lock()
```

`ThreadSafeDict` riunisce insieme un `dict` e un `threading.Lock`. Non abbiamo voluto ereditare `dict`, perché vogliamo mediare tutti gli accessi a `self._dict` in modo che siano sempre serializzati (cioè vogliamo fare in modo che un unico thread per volta possa accedere a `self._dict`).

```
    def copy(self):
        with self._lock:
            return self.__class__(**self._dict)
```

I lock di Python supportano il protocollo context manager, perciò per effettuare un lock basta utilizzare un'istruzione `with`, fidando nel fatto che il lock sarà rilasciato quando non servirà più, anche in caso di eccezioni.

L'istruzione `with self._lock` attiva un blocco se qualsiasi altro thread detiene il lock e continua nel corpo del blocco soltanto dopo che il lock è stato acquisito, cioè quando nessun altro thread lo detiene. Ecco perché è importante fare il meno possibile nel modo più rapido possibile, nel contesto di un lock. In questo caso particolare l'operazione è costosa, ma non c'è una soluzione migliore.

Se una classe implementa un metodo `copy()`, tale metodo dovrebbe restituire una copia dell'istanza su cui è richiamato. Non possiamo restituire `self._dict.copy()`, poiché produce un `dict` normale. Avremmo potuto restituire `ThreadSafeDict(**self._dict)`, ma così avremmo ottenuto sempre un `ThreadSafeDict`, anche da un'istanza di sottoclasse (a meno che la sottoclasse reimplementasse il metodo `copy()`). Il codice che abbiamo utilizzato in questo esempio funziona sia per `ThreadSafeDict` sia per le sottoclassi (cfr. il riquadro dedicato allo spacchettamento di sequenze e mappe nel Capitolo 1).

```
    def get(self, key, default=None):
        with self._lock:
            return self._dict.get(key, default)
```

Questo metodo fornisce un’implementazione thread-safe del metodo `dict.get()`.

```
def __getitem__(self, key):
    with self._lock:
        return self._dict[key]
```

Questo metodo speciale fornisce supporto per l’accesso a valori di dizionario mediante la chiave, cioè `value = d[key]`.

```
def __setitem__(self, key, value):
    with self._lock:
        self._dict[key] = value
```

Questo metodo speciale fornisce supporto per inserire elementi nel dizionario o per modificare il valore di un elemento esistente usando la sintassi `d[key] = value`.

```
def __delitem__(self, key):
    with self._lock:
        del self._dict[key]
```

Questo è il metodo speciale che supporta l’istruzione `del`, cioè `del d[key]`.

```
def __contains__(self, key):
    with self._lock:
        return key in self._dict
```

Questo metodo speciale restituisce `True` se il dizionario contiene un elemento con la chiave data, altrimenti restituisce `False`. È utilizzato attraverso la parola chiave `in`; per esempio, `if k in d: ...`

```
def __len__(self):
    with self._lock:
        return len(self._dict)
```

Questo metodo speciale restituisce il numero di elementi di un dizionario. Supporta la funzione integrata `len()`; per esempio, `count = len(d)`.

Il `ThreadSafeDict` non fornisce i metodi `clear()`, `fromkeys()`, `items()`, `keys()`, `pop()`, `popitem()`, `setdefault()`, `update()`, e `values()` disponibili in `dict`. La maggior parte di questi metodi dovrebbe essere semplice da implementare, tuttavia, i metodi che restituiscono viste (come `items()`, `keys()` e `values()`) richiedono particolare attenzione. L’approccio più semplice e sicuro è quello di non implementarli affatto. Un’alternativa è quella di fare in modo che restituiscano una copia dei loro dati in forma di lista (per esempio, `keys()` potrebbe essere implementato con un corpo costituito da `with self._lock: return list(self._dict.keys())`). Per dizionari grandi questo potrebbe comportare l’uso di una grande quantità di memoria, e naturalmente un tale metodo impedirebbe ad altri thread di accedere al dizionario durante la sua esecuzione.

Un altro approccio per creare un dizionario thread-safe sarebbe quello di creare un dizionario normale all'interno di un unico thread. Se prestiamo attenzione al fatto di scrivere in questo dizionario soltanto dal thread in cui è stato creato (o se utilizziamo un lock e scriviamo nel dizionario soltanto dai thread che detengono tale lock), potremmo poi fornire viste di sola lettura (cioè thread-safe) di tale dizionario ad altri thread utilizzando la classe `types.MappingProxyType` introdotta in Python 3.3.

La sottoclasse per il dizionario del contatore

Anziché utilizzare un normale `ThreadSafeDict` per il dizionario con le letture del contatore (numero del contatore, valori di lettura), abbiamo creato una sottoclasse privata `_MeterDict` che aggiunge due nuovi metodi.

```
class _MeterDict(ThreadSafeDict):

    def insert_if_missing(self, key, value=None):
        with self._lock:
            if key not in self._dict:
                self._dict[key] = value
                return True
        return False
```

Questo metodo inserisce la chiave e il valore dati nel dizionario e restituisce `True`, oppure, se la chiave (cioè il numero del contatore finto) è già contenuta nel dizionario, non compie alcuna azione e restituisce `False`. Così ci assicuriamo che ogni richiesta di job riguardi un contatore nuovo e unico.

Il codice eseguito dal metodo `insert_if_missing()` è in sostanza:

```
if meter not in ReadingForMeter:  # SBAGLIATO!
    ReadingForMeter[key] = None
```

`ReadingForMeter` è un'istanza di `_MeterDict` e quindi eredita tutte le funzionalità della classe `ThreadSafeDict`. Anche se il metodo `ReadingForMeter.__contains__()` (per `in`) e il metodo `ReadingForMeter.__setitem__()` (per `[]`) sono entrambi thread-safe, il codice mostrato qui non è thread-safe. Il motivo è che un altro thread potrebbe accedere al dizionario `ReadingForMeter` dopo l'istruzione `if` ma prima dell'assegnazione. La soluzione consiste nell'eseguire entrambe le operazioni nel contesto dello stesso lock, esattamente ciò che fa il metodo `insert_if_missing()`.

```
    def status(self, username):
        count = total = 0
        with self._lock:
            for reading in self._dict.values():
                if reading is not None:
                    total += 1
                    if reading.username == username:
                        count += 1
        return count, total
```

Questo è un metodo potenzialmente costoso, poiché itera su tutti i valori del dizionario sottostante nel contesto di un lock. Un'alternativa sarebbe quella di avere un'unica istruzione all'interno del contesto - `values = self._dict.values()` - e di svolgere l'iterazione dopo (cioè al di fuori del contesto del lock). Dipende dalle circostanze se sia più rapido copiare gli elementi all'interno di un lock e poi elaborare le copie senza un lock, oppure elaborare gli elementi all'interno di un lock. L'unico modo per saperlo con sicurezza è quello di testare entrambi gli approcci in contesti realistici.

Scrivere server RPyC

Abbiamo visto in precedenza che è facile creare un server XML-RPC utilizzando il modulo `xmlrpc.server`. È altrettanto facile - benché diverso - creare un server RPyC.

```
import datetime
import threading
import rpyc
import sys
import MeterMT

PORT = 11003

Manager = MeterMT.Manager()
```

Questo è l'inizio di `meterserver-rpyc.py`. Importiamo un paio di moduli della libreria standard, poi il modulo `rpyc`, e poi il nostro modulo thread-safe `MeterMT`. Abbiamo impostato un numero di porta fisso, anche se potrebbe facilmente essere cambiato utilizzando un'opzione della riga di comando e il modulo `argparse`, come abbiamo fatto per la versione XML-RPC. Inoltre abbiamo creato una singola istanza di un `MeterMT.Manager`, che sarà condivisa dai thread del server RPyC.

```
if __name__ == "__main__":
    import rpyc.utils.server
    print("Meter server startup at {}".format(
            datetime.datetime.now().isoformat()[:19]))
    server = rpyc.utils.server.ThreadedServer(MeterService, port=PORT)
    thread = threading.Thread(target=server.start)
    thread.start()
    try:
        if len(sys.argv) > 1:  # Notifica se è stato richiamato da un client GUI
            with open(sys.argv[1], "wb") as file:
                    file.write(b"\n")
        thread.join()
    except KeyboardInterrupt:
        pass
    server.close()
    print("\rMeter server shutdown at {}".format(
            datetime.datetime.now().isoformat()[:19]))
    MeterMT.Manager._dump()
```

Questa è la parte finale del programma server. Importiamo il modulo server RPyC e annunciamo l'avvio. Poi creiamo un'istanza di un server threaded e passiamo una

classe `MeterService`. Il server creerà istanze di questa classe ove necessario. Torneremo su questa classe tra breve.

Una volta che il server è stato creato, potremmo semplicemente scrivere `server.start()` e terminare. Così il server sarebbe avviato e lasciato in esecuzione "per sempre". Vogliamo invece che l'utente possa interrompere il server premendo Ctrl+C (o con un segnale `INT`) e che il server stampi le letture del contatore quando viene arrestato.

Per raggiungere questo scopo, avviamo il server in un proprio thread, da cui esso creerà un pool di thread per gestire le connessioni in arrivo, e poi attueremo un blocco in attesa che il thread del server termini (mediante `thread.join()`). Se il server è interrotto, catturiamo e ignoriamo l'eccezione e chiudiamo il server. La chiamata di `close()` attua un blocco finché ogni thread server termina la connessione corrente. Poi annunciamo la chiusura del server e stampiamo le letture del contatore che gli sono state inviate.

Se il server è avviato da un client GUI, ci aspettiamo che il client passi un nome di file di notifica come unico argomento. Se è presente un argomento di notifica, creiamo il file e scriviamo in esso una nuova riga per notificare al client che il server è in esecuzione e pronto.

Quando fa uso di un modo di servizio, un server RPyC utilizza una sottoclasse `rpyc.Service` che può poi impiegare come factory per produrre istanze del servizio (le factory sono state trattate nel Capitolo 1). Abbiamo creato la classe `MeterService` come wrapper per l'istanza `MeterMT.Manager` creata all'inizio del programma.

```
class MeterService(rpyc.Service):

    def on_connect(self):
        pass

    def on_disconnect(self):
        pass
```

Ogni volta che viene effettuata una connessione a un servizio, viene richiamato il metodo `on_connect()` di quel servizio. Similmente, quando una connessione termina, viene richiamato il metodo `on_disconnect()`. Non dobbiamo fare nulla in entrambi i casi, perciò abbiamo creato metodi che "non fanno nulla". È del tutto legittimo evitare di implementare questi metodi se non sono necessari; li abbiamo inclusi qui solo per mostrare le loro firme.

```
    exposed_login = Manager.login
    exposed_get_status = Manager.get_status
    exposed_get_job = Manager.get_job
```

Un servizio può esporre metodi (o classi e altri oggetti) ai client. Qualunque classe o metodo il cui nome inizia con `exposed_` è disponibile per l'accesso da parte dei client, e nel caso dei metodi i client possono richiamarli con o senza tale prefisso. Per esempio, un client contatore RPyC potrebbe richiamare `exposed_login()` oppure `login()`.

Per i metodi `exposed_login()`, `exposed_get_status()` ed `exposed_get_job()` non facciamo altro che impostarli ai metodi corrispondenti nell'istanza di gestione del contatore del programma.

```
    def exposed_submit_reading(self, sessionId, meter, when, reading,
            reason=""):
        when = datetime.datetime.strptime(str(when)[:19],
                "%Y-%m-%d %H:%M:%S")
        Manager.submit_reading(sessionId, meter, when, reading, reason)
```

Per questo metodo abbiamo fornito un wrapper sul metodo di gestione del contatore. Il motivo è che la variabile `when` è passata come `datetime.datetime` wrapped da RPyC `netref`, anziché come `datetime.datetime` puro. Nella maggior parte dei casi non farebbe differenza, ma in questo caso vogliamo memorizzare gli effettivi `datetime.datetime` nel dizionario del contatore, anziché dei riferimenti a `datetime.datetime` remoti (cioè lato client), perciò convertiamo la data/ora wrapped in una stringa di data/ora ISO 8601 e ne effettuiamo il parsing in un `datetime.datetime` lato server, che poi passiamo al metodo `MeterMT.Manager.submit_reading()`.

Il codice riportato in questo paragrafo è il server contatore RPyC completo, e sarebbe più cordo di qualche riga se rimuovessimo i metodi `on_connect()` e `on_disconnect()`.

Scrivere client RPyC

Per creare client RPyC si lavora in modo molto simile a quello che abbiamo visto per i client XML-RPC, perciò nel seguito ci limiteremo a evidenziare le differenze tra i due tipi.

Un client RPyC di console

Esattamente come il client XML-RPC, anche il client RPyC richiede che il server sia avviato e arrestato separatamente, e funziona soltanto quando un server è in esecuzione.

Il codice per il programma `meterclient-rpyc.py` è quasi identico a quello per il client `meterclient-rpc.py` che abbiamo visto in precedenza in questo capitolo. Soltanto le

funzioni `main()` e `submit()` sono diverse.

```
def main():
    username, password = login()
    if username is not None:
        try:
            service = rpyc.connect(HOST, PORT)
            manager = service.root
            sessionId, name = manager.login(username, password)
            print("Welcome, {}, to Meter RPYC".format(name))
            interact(manager, sessionId)
        except ConnectionError as err:
            print("Error: Is the meter server running? {}".format(err))
```

La prima differenza è che abbiamo usato un nome di host e un numero di porta inseriti esplicitamente nel codice. Naturalmente avremmo potuto facilmente rendere questi dati configurabili, come abbiamo fatto per il client XML-RPC. La seconda differenza è che, invece di creare un gestore proxy e poi connettersi, iniziamo con la connessione a un server che fornisce servizi. In questo caso il server fornisce un unico servizio (`MeterService`), e possiamo usare un proxy di gestione del contatore. Tutto l'altro codice - per il login del gestore del contatore, per l'invio di letture e per ottenere lo stato - è identico a prima, con una sola eccezione: la funzione `submit()` cattura eccezioni diverse da quelle catturate dal client XML-RPC.

Sincronizzare nomi di host e numeri di posta può essere complicato, soprattutto se un conflitto ci obbliga a utilizzare un numero di porta diverso da quello che utilizziamo normalmente. Questo problema può essere evitato usando un server di registro. Dobbiamo eseguire il server `registry_server.py` fornito con RPyC nella nostra rete. I server RPyC cercano automaticamente tale server quando si avviano, e se lo trovano, registrano con esso i loro servizi. A questo punto i client, invece di utilizzare `rpyc.connect(host, port)`, possono utilizzare `rpyc.connect_by_service(service)`; per esempio, `rpyc.connect_by_service("Meter")`.

Un client GUI RPyC

Il client GUI RPyC `meter-rpyc.pyw` è mostrato nella Figura 6.2. In realtà i client GUI RPyC e XML-RPyC sono visivamente indistinguibili quando sono eseguiti sulla stessa piattaforma.

Per creare un client GUI RPyC che utilizza Tkinter e utilizzerà automaticamente un server contatore esistente, o ne avvierà uno se necessario, si può utilizzare quasi lo stesso codice che abbiamo utilizzato per il client GUI XML-RPC. In effetti le differenze si riducono a un paio di metodi modificati, un'importazione, alcune costanti e qualche eccezione diverse nelle clausole `except`.

```
    def connect(self, username, password):
        try:
            self.service = rpyc.connect(HOST, PORT)
        except ConnectionError:
            filename = os.path.join(tempfile.gettempdir(),
                    "M{}.$$$".format(random.randint(1000, 9999)))
            self.serverPid = subprocess.Popen([sys.executable, SERVER,
                    filename]).pid
            self.wait_for_server(filename)
            try:
                self.service = rpyc.connect(HOST, PORT)
            except ConnectionError:
                self.handle_error("Failed to start the RPYC Meter server")
                return False
        self.manager = self.service.root
        return self.login_to_server(username, password)
```

Dopo che è stata utilizzata la finestra di login per ottenere il nome utente e la password, viene richiamato questo metodo per effettuare la connessione al server e il login dell'utente con il gestore del contatore.

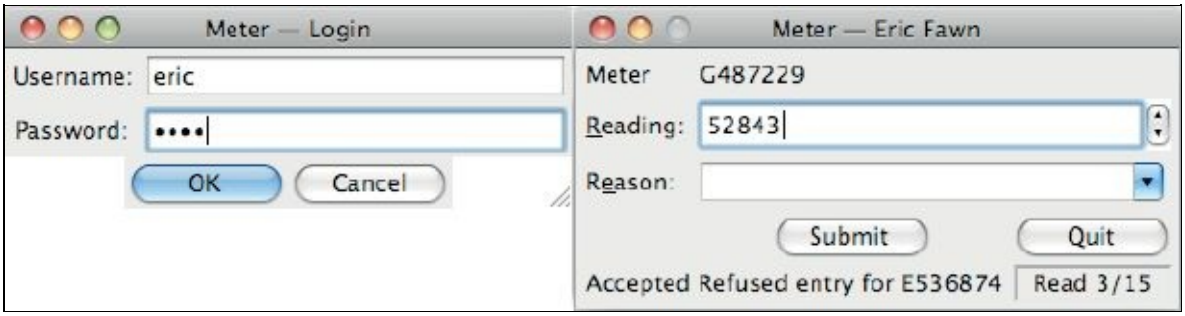


Figura 6.2 La finestra di login e la finestra principale dell'applicazione GUI RPyC del contatore su OS X.

Se il tentativo di connessione fallisce, assumiamo che il server non sia in esecuzione e cerchiamo di avviarlo, passandogli il nome di un file di notifica. Il server viene iniziato senza bloccaggio (cioè in modo asincrono), ma dobbiamo attendere che sia in esecuzione prima di tentare di connetterci. Il metodo `wait_for_server()` è quasi identico a quello che abbiamo visto precedentemente in questo capitolo, a parte il fatto che questa versione solleva un'eccezione `ConnectionError` anziché richiamare `handle_error()`. Se la connessione viene stabilita, acquisiamo un gestore proxy e cerchiamo di eseguire il login dell'utente sul server contatore.

```
    def login_to_server(self, username, password):
        try:
            self.sessionId, name = self.manager.login(username, password)
            self.master.title("Meter \u2014 {}".format(name))
            return True
        except rpyc.core.vinegar.GenericException as err:
            self.handle_error(err)
            return False
```

Se le credenziali dell'utente sono accettabili, impostiamo l'ID di sessione e inseriamo il nome dell'utente nella barra del titolo dell'applicazione. Se il login

fallisce, restituiamo `False`, facendo terminare l'applicazione (e anche il server, se è stato avviato dall'applicazione GUI).

Nessuno degli esempi riportati in questo capitolo fa ricorso alla cifratura, perciò uno spione potrebbe potenzialmente osservare il traffico di rete client-server. Questo potrebbe non comportare alcun problema per applicazioni che non trasferiscono dati privati, o che eseguono client e server sulla stessa macchina, o in cui client e server si trovano al sicuro dietro un firewall, o in cui si utilizzano connessioni di rete cifrate. Ma se è richiesta la cifratura, la si può ottenere. Nel caso di XML-RPC, un approccio è quello di utilizzare il pacchetto esterno PyCrypto (`http://www.dlitz.net/software/pycrypto`) per cifrare tutti i dati inviati in rete. Un altro approccio consiste nell'utilizzare il TLS (*Transport Layer Security*, i "socket sicuri"), che è supportato dal modulo `ssl` di Python. Nel caso di RPyC è molto più facile ottenere una buona sicurezza, grazie al supporto integrato. RPyC può usare SSL con chiavi e certificati, oppure il più semplice approccio con tunneling SSH (*Secure Shell*).

Python offre un eccellente supporto di rete che copre ogni livello dal più basso al più alto. La libreria standard include moduli per tutti i protocolli di alto livello più comuni, tra cui FTP per il trasferimento di file, POP3, IMAP4 e SMTP per l'email; HTTP e HTTPS per il traffico web e naturalmente TCP/IP e altri protocolli socket di basso livello. Il modulo di livello intermedio `socketserver` può essere usato come base per creare server, ma non manca il supporto per server di livello più alto, per esempio, il modulo `smtpd` per creare server di posta, il modulo `http.server` per server web e il modulo `xmlrpc.server` che abbiamo visto utilizzato in questo capitolo per server XML-RPC.

Sono anche disponibili molti moduli di rete esterni, particolarmente per framework web che supportano la WSGI (*Web Server Gateway Interface*) di Pyhon, cfr. `http://www.python.org/dev/peps/pep-3333`. Per ulteriori informazioni sui framework web esterni Python cfr. `http://wiki.python.org/moin/WebFrameworks`, e per approfondimenti sui server web cfr. `http://wiki.python.org/moin/WebServers`.

Interfacce con Tkinter

Le applicazioni GUI (*Graphical User Interface*) ben progettate sono in grado di offrire agli utenti interfacce accattivanti, intuitive e di facile utilizzo. Inoltre, più l'applicazione è sofisticata, maggiore è il vantaggio che essa può trarre da una GUI personalizzata, soprattutto se quest'ultima comprende widget personalizzati specifici dell'applicazione stessa (nel descrivere un oggetto GUI, i programmatori GUI Windows utilizzano spesso i termini "controllo", "contenitore" o "modulo". In questo libro utilizziamo il termine generico *widget*, derivante dalla programmazione GUI Unix). A titolo di paragone, le applicazioni web possono causare parecchia confusione, con menu e barre degli strumenti del browser in aggiunta ai widget dell'applicazione web. E fino a quando il canvas HTML5 non sarà ampiamente disponibile, le applicazioni web hanno un mezzo assai limitato per la presentazione di widget personalizzati. Per di più, in termini di prestazioni le applicazioni web non possono competere con le applicazioni native.

Gli utenti di smartphone sono sempre più in grado di interagire con le proprie app mediante il controllo vocale, ma per PC desktop, portatili e tablet, le scelte possibili sono ancora principalmente tra le applicazioni GUI tradizionali controllate tramite mouse e tastiera o voce, e applicazioni a comando tattile. Al momento della stesura di questo libro, pressoché tutti i dispositivi con comandi tattili utilizzano librerie proprietarie e richiedono l'utilizzo di linguaggi e strumenti specifici. Fortunatamente, la libreria esterna open source Kivy (`http://kivy.org`) è progettata per fornire a Python il supporto per lo sviluppo di applicazioni tattili indipendenti dalla piattaforma, atto a risolvere questo problema. Ovviamente, ciò non cambia il fatto che la maggior parte delle interfacce tattili sia progettata per macchine con potenza di elaborazione limitata e schermi piccoli e che tali interfacce consentano all'utente di vedere un'unica applicazione alla volta.

Gli utenti desktop e avanzati desiderano trarre il massimo vantaggio dai propri schermi di grandi dimensioni e dai processori sempre più potenti, e il modo migliore di ottenere ciò è ancora con le applicazioni GUI tradizionali. Inoltre, il controllo vocale, così come fornito, per esempio, dalle moderne versioni di Windows, è progettato per funzionare con le applicazioni GUI esistenti. I programmi Python a riga di comando possono essere utilizzati tra varie piattaforme e lo sono anche i programmi GUI Python, a condizione di utilizzare un toolkit GUI appropriato.

Esistono svariati toolkit tra cui scegliere. Di seguito è riportata una breve panoramica dei quattro toolkit principali disponibili in Python 3 e che funzionano come minimo in Linux, OS X e Windows, con aspetto nativo.

- **PyGtk e PyGObject**: PyGtk (`http://www.pygtk.org`) è stabile e valido. Tuttavia, lo sviluppo è cessato nel 2011 a favore di una tecnologia successiva denominata PyGObject (`http://live.gnome.org/PyGObject`). Sfortunatamente, al momento della stesura del libro, PyGObject non può essere considerato indipendente dalla piattaforma, poiché tutta l'attività di sviluppo sembra essere confinata a sistemi basati su Unix.
- **PyQt4 e PySide**: PyQt4 (`http://www.riverbankcomputing.co.uk`) fornisce binding Python per il framework di sviluppo dell'applicazione GUI Qt 4 (`http://qt-project.org`). PySide (`http://www.pyside.org`) è un progetto più recente altamente compatibile con PyQt4 che possiede una licenza più aperta. PyQt4 è probabilmente il toolkit GUI Python indipendente dalla piattaforma più stabile e maturo disponibile (nota: l'autore di questo libro è stato per un periodo il direttore della documentazione di Qt e ha scritto un volume sulla programmazione PyQt4 intitolato *Rapid GUI Programming with Python and Qt*, cfr. la bibliografia a fine volume). È previsto che PyQt e PySide abbiano entrambi una versione che supporti Qt 5.
- **Tkinter**: fornisce binding al toolkit GUI Tcl/Tk (`http://www.tcl.tk`). Python 3 viene fornito normalmente con Tcl/Tk 8.5, anche se quest'ultimo sarà probabilmente sostituito da Tcl/Tk 8.6 con la versione Python 3.4 o successiva. Diversamente dagli altri toolkit citati qui, Tkinter è assai elementare, senza alcun supporto integrato per barre degli strumenti, finestre a dock o barre di stato (sebbene tutte queste caratteristiche possano essere comunque create). Inoltre, mentre gli altri toolkit funzionano automaticamente con molte funzioni specifiche di piattaforma - come la barra di menu universale di OS X - Tkinter (almeno con Tcl/Tk 8.5) richiede ai programmatori di considerare le molteplici differenze di piattaforma. Le virtù principali di Tkinter riguardano il fatto che è fornito con Python come standard,e che è un pacchetto molto contenuto in confronto agli altri toolkit.
- wxPython: wxPython (`http://www.wxpython.org`) fornisce binding al toolkit wxWidgets (`http://www.wxwidgets.org`). Sebbene wxPython sia in auge da parecchi anni, ne è stata intrapresa una riscrittura significativa per il porting in Python 3.

Fatta eccezione per PyGObject, i toolkit elencati in precedenza forniscono tutto il necessario per creare applicazioni GUI indipendenti dalla piattaforma con Python. Se ci occupiamo soltanto di un'unica piattaforma specifica, certamente sono disponibili binding Python per librerie GUI specifiche della piattaforma (cfr. `http://wiki.python.org/moin/GuiProgramming`), oppure possiamo utilizzare un interprete Python specifico della piattaforma, quale Jython o IronPython. Se desideriamo creare grafica 3D, possiamo farlo solitamente all'interno di uno dei toolkit GUI. In alternativa, possiamo utilizzare PyGame (`http://www.pygame.org`), oppure, se le nostre esigenze sono più semplici, possiamo utilizzare direttamente uno dei binding OpenGL Python, come vedremo nel prossimo capitolo.

Poiché Tkinter è fornito come standard, possiamo creare applicazioni GUI che possiamo implementare facilmente (se necessario, anche mettendo in bundle Python e Tcl/Tk con l'applicazione stessa; cfr. per esempio `http://cx-freeze.sourceforge.net`). Tali applicazioni sono più attraenti e facili da utilizzare rispetto ai programmi a riga di comando e sono spesso più accettabili per gli utenti, in particolare su OS X e Windows.

Questo capitolo illustra tre applicazioni di esempio: una piccola applicazione del tipo "hello world", un piccolo convertitore di valute e il più sostanzioso gioco Gravitate. Quest'ultimo può essere considerato come una variante del gioco TileFall/Same dove le mattonelle gravitano al centro per riempire lo spazio vuoto invece di cadere e spostarsi a sinistra. L'applicazione Gravitate illustra come creare un'applicazione Tkinter a stile finestra principale con alcuni dei moderni elementi accessori quali menu, finestre di dialogo e barra di stato. Analizzeremo alcune finestre di dialogo di Gravitate e l'infrastruttura a finestra principale di Gravitate.

Introduzione a Tkinter

La programmazione GUI non è più complessa di qualsiasi altro tipo di programmazione speciale e ha il pregio potenziale di produrre applicazioni dall'aspetto professionale che le persone amano utilizzare.

Osservate, però, che l'argomento della programmazione GUI è talmente ricco che non possiamo esplorarlo in profondità in un singolo capitolo; come minimo sarebbe necessario un intero libro per farlo. Quello che possiamo fare, in ogni caso, è esaminare alcuni degli aspetti chiave della scrittura di programmi GUI e, in particolare, come colmare alcune delle lacune all'interno delle strutture di Tkinter. Per prima cosa inizieremo dal classico programmino "hello world", in questo caso `hello.pyw`, mostrato in esecuzione nella Figura 7.1.

```
import tkinter as tk
import tkinter.ttk as ttk
class Window(ttk.Frame):

    def __init__(self, master=None):
        super().__init__(master)  # Crea self.master
        helloLabel = ttk.Label(self, text="Hello Tkinter!")
        quitButton = ttk.Button(self, text="Quit", command=self.quit)
        helloLabel.pack() quitButton.pack() self.pack()
```

Figura 7.1 L'applicazione Hello con finestra di dialogo in Linux, OS X e Windows.

```
window = Window()  # Crea implicitamente l'oggetto tk.Tk
window.master.title("Hello")
window.master.mainloop()
```

Questo è l'intero codice dell'applicazione `hello.pyw`. Molti programmatori Tkinter importano tutti i nomi Tkinter (per esempio `from tkinter import *`), ma noi preferiamo utilizzare i namespace (nonostante quelli abbreviati, `tk` e `ttk`) in modo da mantenere la chiarezza sulla provenienza (per inciso, il modulo `ttk` è un wrapper che racchiude l'estensione Tcl/Tk di "Tile" Ttk). Avremmo potuto semplicemente eseguire prima l'importazione e utilizzare un `tkinter.Frame` invece di `tkinter.ttk.Frame` e così via, tuttavia le versioni di `tkinter.ttk` forniscono il supporto per i temi, perciò l'utilizzo di questi è preferibile, soprattutto su OS X e Windows.

La maggior parte dei normali widget `tkinter` possiede anch'essa versioni `tkinter.ttk` con tema. I widget normali e con tema non sempre hanno le medesime interfacce ed esistono alcuni contesti nei quali è possibile utilizzare solo un widget normale, perciò

è importante consultare la documentazione (raccomandiamo la documentazione disponibile presso `http://>www.tcl.tk` a coloro che sono in grado di comprendere il codice Tcl/Tk; in caso contrario, raccomandiamo `http://www.tkdocs.com`, che illustra esempi in Python e in alcuni altri linguaggi, e anche `http://infohost.nmt.edu/tcc/help/pubs/tkinter/web`, che fornisce un'utile esercitazione/riferimento di Tkinter). Esistono anche diversi widget con tema `tkinter.ttk` per i quali non vi sono equivalenti normali; per esempio, `tkinter.ttk.Combobox`, `tkinter.ttk.Notebook` e `tkinter.ttk.Treeview`.

Lo stile di programmazione GUI che adottiamo in questo volume è quello di creare una classe per finestra, normalmente nel proprio specifico modulo. Per una finestra di livello superiore (ossia, la finestra principale di un'applicazione), solitamente si eredita da `tkinter.Toplevel` o `tkinter.ttk.Frame`, come abbiamo fatto qui. Tkinter mantiene una gerarchia di proprietà widget genitori e figli (a volte chiamati master e slave). Non dobbiamo preoccuparci di ciò, fintanto che richiamiamo la funzione integrata `super()` nel metodo `__init__()` di qualsiasi classe che creiamo che eredita un widget.

La creazione della maggior parte delle applicazioni GUI segue un pattern standard: creare una o più classi di finestre, una delle quali è la finestra principale dell'applicazione. Per ciascuna classe di finestra, creare le variabili di finestra (non ve n'è alcuna in `hello.pyw`), creare i widget, disporre i widget e specificare i metodi da richiamare in risposta agli eventi (per esempio clic del mouse, pressione di tasti, timeout). In questo caso, associamo l'utente che fa clic su `quitButton` al metodo ereditato `tkinter.ttk.Frame.quit()` che chiuderà la finestra, e poiché questa è l'unica finestra di primo livello dell'applicazione, l'applicazione sarà semplicemente terminata. Una volta pronte tutte le classi finestra, la fase finale sarà la creazione di un oggetto applicazione (in questo esempio eseguita in modo implicito) e l'avvio del ciclo di eventi GUI. Il ciclo di evento è stato illustrato già nel Capitolo 4.

Ovviamente, la maggior parte delle applicazioni GUI è molto più lunga e complicata di `hello.pyw`. Tuttavia, le classi di finestra seguono lo stesso pattern descritto qui, solo che generalmente creano un numero ben maggiore di widget e associano molti più eventi.

In molti toolkit GUI moderni è comune utilizzare per i widget i layout invece di dimensioni e posizioni codificate esplicitamente. Ciò rende possibile l'espansione o la riduzione automatica dei widget per ottenere una disposizione più pulita dei contenuti (per esempio un'etichetta o il testo di un pulsante), anche se questi variano,

mantenendo nel contempo la loro posizione correlata a tutti gli altri widget. I layout consentono inoltre ai programmatori di evitare l'esecuzione di calcoli tediosi.

Tkinter fornisce tre gestori di layout: *place* (posizioni codificate esplicitamente; utilizzato raramente), *pack* (consente di posizionare i widget attorno a una cavità centrale nozionale) e *grid* (consente di organizzare i widget in una griglia di righe e colonne; è il più diffuso). In questo esempio abbiamo utilizzato il gestore di layout *pack* per affiancare l'etichetta e il pulsante uno dopo l'altro, quindi anche per l'intera finestra. Il gestore *pack* va bene per finestre molto semplici come questa, tuttavia la griglia è la più facile da utilizzare, come vedremo negli esempi successivi.

Le applicazioni GUI rientrano in due categorie principali: con stile a finestra di dialogo e con stile a finestra principale. Le prime sono le finestre prive di menu o barre degli strumenti, che vengono invece controllate mediante pulsanti, caselle combinate ed elementi simili. L'utilizzo dello stile a finestra di dialogo è ideale per applicazioni che richiedono soltanto un'interfaccia utente semplice, come i piccoli programmi di utilità, i lettori multimediali e alcuni giochi. Le applicazioni con stile a finestra principale solitamente possiedono menu e barre degli strumenti al di sopra dell'area centrale e una barra di stato nell'area inferiore. Possono avere anche finestre agganciabili. Le finestre principali sono ideali per applicazioni più complesse e spesso hanno opzioni di menu o pulsanti di barra degli strumenti che determinano la comparsa di finestre di dialogo. Osserveremo entrambi i tipi di applicazione, iniziando da quelle con stile a finestra di dialogo, poiché quasi tutto ciò che apprenderemo in merito è pertinente anche alle finestre di dialogo utilizzate dalle applicazioni con stile a finestra principale.

Creazione di finestre di dialogo con Tkinter

Le finestre di dialogo dispongono di quattro modalità possibili e svariati livelli di intelligenza. Di seguito è riportato un breve riepilogo delle modalità, dopodiché illustreremo il concetto di intelligenza.

- **Modale globale**: la finestra modale globale è quella che blocca l'intera interfaccia utente - incluse tutte le altre applicazioni - e consente le interazioni esclusivamente con se stessa. Gli utenti non possono commutare le applicazioni né eseguire alcunché, tranne interagire con la finestra. I due casi di utilizzo comune sono le finestre di dialogo per l'accesso, poiché un bug potrebbe determinare l'instabilità dell'intero sistema.
- **Modale di applicazione**: le finestre modali di applicazione impediscono agli utenti di interagire con qualsiasi altra finestra dell'applicazione. Ma gli utenti possono comunque eseguire la commutazione contestuale su altre applicazioni. Le finestre modali sono più facili da programmare delle finestre non modali, poiché l'utente non può modificare lo stato dell'applicazione a insaputa del programmatore. In ogni caso, alcuni utenti trovano ciò scomodo.
- **Modale di finestra**: le finestre modali di finestra sono molto simili alle finestre modali di applicazione, salvo che invece di impedire l'interazione con qualsiasi altra finestra dell'applicazione, impediscono l'interazione con qualsiasi altra finestra di applicazione nella stessa gerarchia di finestre. Ciò è utile, per esempio, se l'utente apre due finestre documento di livello superiore, poiché non vogliamo che il fatto che utilizzino una finestra di dialogo in una di tali finestre impedisca l'interazione con l'altra finestra.
- **Non modale**: le finestre di dialogo non modali non bloccano l'interazione con qualsiasi altra finestra nella propria applicazione o in qualsiasi altra applicazione. Si tratta di finestre potenzialmente molto più difficoltose da creare per i programmatori, rispetto alle finestre di dialogo modali. Questo perché una finestra di dialogo non modale deve essere in grado di far fronte all'utente che interagisce con le finestre di altre applicazioni, modificando magari lo stato da cui dipende la finestra di dialogo non modale stessa.

Nella terminologia Tcl/Tk si dice che le finestre modali globali possiedono un *global grab* (traducibile in "presa globale"). Le finestre modali di applicazione e finestra (chiamate semplicemente "finestre" modali) possiedono invece una *presa locale*. In Tkinter su sistema OS X, alcune finestre modali appaiono come schede.

Una finestra di dialogo normale generalmente presenta all'utente alcuni widget e fornisce ciò che ha immesso l'utente all'applicazione. Tali finestre di dialogo non hanno alcuna conoscenza specifica dell'applicazione sottostante. Un esempio tipico è una finestra di dialogo di login che accetta semplicemente un nome utente e una password, e successivamente trasferisce all'applicazione tali dati (abbiamo visto un esempio di utilizzo di tale finestra di dialogo nel Capitolo 6. Il codice si trova in `MeterLogin.py`.).

Una finestra di dialogo intelligente è una che incarna un certo livello di conoscenza dell'applicazione e alla quale possono persino essere passati riferimenti a variabili o strutture dati dell'applicazione stessa, in modo che possa agire direttamente sui dati.

Le finestre di dialogo modali possono essere normali o intelligenti o a metà tra le due. Una finestra modale abbastanza intelligente è generalmente una che comprende quanto basta dell'applicazione per fornire la validazione, e non solo per l'elemento di dati che si presenta per la modifica, bensì per combinazioni di elementi di dati. Per esempio, una finestra di dialogo ragionevolmente intelligente per l'immissione di una data di inizio e di fine non accetta una data finale che sia antecedente a quella di inizio.

Le finestre di dialogo non modali sono quasi sempre intelligenti. Generalmente si presentano in due tipologie: *applica/chiudi*) e *live* (in diretta). Le finestre di dialogo del tipo applica/chiudi consentono agli utenti di interagire con i widget, quindi occorre fare clic su un pulsante *Applica* per vedere i risultati nella finestra principale dell'applicazione. Le finestre di dialogo *live* applicano le modifiche mentre l'utente interagisce con i widget della finestra di dialogo; sono assai comuni su OS X. Le finestre di dialogo non modali più intelligenti offrono un pulsante *Annulla/Ripeti* o *Default* (per ripristinare i widget ai valori predefiniti dell'applicazione) e magari un pulsante *Ripristina* (per ripristinare i widget ai valori che detenevano al momento della prima chiamata della finestra di dialogo). Le finestre di dialogo non modali possono essere ordinarie se forniscono semplicemente informazioni, come una finestra di dialogo della Guida. Queste generalmente hanno solo un pulsante *Chiudi*.

Le finestre di dialogo non modali sono particolarmente utili quando si modificano colori, tipi di carattere, formati o template, poiché consentono di vedere gli effetti di ciascuna variazione e di apportare un'altra modifica e un'altra ancora. Utilizzare una finestra di dialogo modale in tali situazioni significa che dobbiamo aprire la finestra di dialogo, apportare le modifiche, accettare la finestra di dialogo, quindi ripetere questo ciclo per ogni variazione finché i risultati non sono soddisfacenti.

La finestra principale di un’applicazione a finestra di dialogo è essenzialmente una finestra di dialogo non modale. Le applicazioni a finestra principale hanno solitamente sia finestre di dialogo modali sia non modali che appaiono in risposta alla scelta da parte dell’utente di particolari opzioni di menu o al clic del mouse su particolari pulsanti della barra degli strumenti.

Creazione di un’applicazione a finestra di dialogo

Nel seguito esamineremo un’applicazione a finestra di dialogo molto semplice, eppure assai utile, che esegue conversioni di valuta. Il codice sorgente si trova nella directory `currency` e l’applicazione è illustrata nella Figura 7.2.

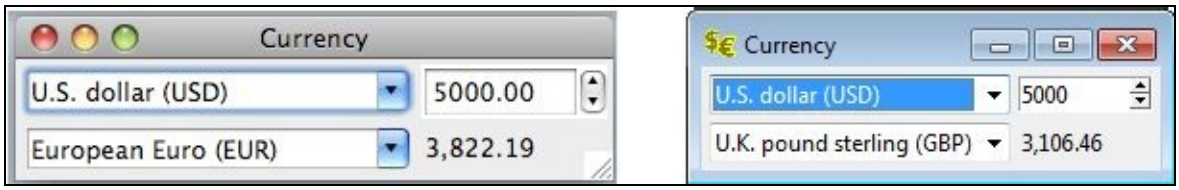


Figura 7.2 L’applicazione Currency a finestra di dialogo su OS X e Windows.

L’applicazione ha due caselle combinate che elencano i nomi delle valute (e i relativi identificatori), una casella con pulsanti di selezione per l’immissione di un importo e un’etichetta che indica il valore dell’importo convertito dalla valuta superiore a quella inferiore.

Il codice dell’applicazione è distribuito su tre file Python: `currency.pyw`, ossia il programma che eseguiamo; `Main.py`, che fornisce la classe `Main.Window`; e `Rates.py`, che fornisce la funzione `Rates.get()` discussa nel Capitolo 1. In aggiunta, sono presenti due icone, `currency/images/ icon_16x16.gif` e `currency/images/icon_32x32.gif`, che forniscono le icone per l’applicazione su Linux e su Windows.

Le applicazioni GUI Python possono utilizzare l’estensione standard `.py`, ma su OS X e Windows l’estensione `.pyw` viene spesso associata a un diverso interprete Python (per esempio `pythonw.exe` invece di `python.exe`). Questo interprete consente di eseguire l’applicazione senza avviare una finestra di console, perciò è molto apprezzata dagli utenti. Per i programmatori, però, è meglio eseguire applicazioni GUI Python dall’interno di una console mediante l’interprete Python standard, poiché ciò consente che sia visibile qualsiasi output `sys.stdout` e `sys.stderr` come aiuto per il debugging.

La funzione main() dell’applicazione Currency

La cosa migliore, soprattutto per programmi estesi, è avere un modulo “eseguibile” molto piccolo e per tutto il resto del codice far sì che questo si trovi in file di modulo `.py` distinti (a prescindere dalla loro dimensione). Su macchine moderne veloci potrebbe sembrare che ciò non faccia alcuna differenza la prima volta che si esegue il programma, ma in tale prima esecuzione tutti i file di modulo `.py` (salvo quello “eseguibile”) vengono compilati in file `.pyc`. La seconda e le volte successive che il programma viene eseguito, Python utilizzerà i file `.pyc` (salvo laddove un file `.py` sia stato modificato), perciò i tempi di avvio saranno più veloci rispetto alla prima volta.

Il file eseguibile dell’applicazione `currency.pyw` contiene una piccola funzione `main()`.

```
def main():
    application = tk.Tk()
    application.title("Currency")
    TkUtil.set_application_icons(application, os.path.join(
            os.path.dirname(os.path.realpath(__file__)), "images"))
    Main.Window(application)
    application.mainloop()
```

La funzione inizia creando l’oggetto applicazione Tkinter. Si tratta in realtà di una finestra di livello superiore normalmente invisibile che funge da widget genitore (o *master*, o *root*). Nell’applicazione `hello.pyw` abbiamo fatto creare ciò a Tkinter implicitamente, ma normalmente è meglio crearlo noi stessi in modo da poter applicare impostazioni a livello di tutta l’applicazione. Qui, per esempio, abbiamo impostato il titolo dell’applicazione in “Currency”.

Gli esempi del libro sono forniti con il modulo `TkUtil`, che comprende alcune comode funzioni integrate per supportare la programmazione Tkinter, più alcuni moduli che illustreremo quando li incontreremo. Qui utilizziamo la funzione `TkUtil.set_application_icons()`.

Con il titolo e le icone impostati (sebbene le icone vengano ignorate su OS X), creiamo un’istanza della finestra principale dell’applicazione, passandole l’oggetto applicazione come genitore (o master), quindi avviamo il ciclo di eventi GUI. L’applicazione si chiuderà quando il ciclo di eventi terminerà; per esempio, se richiamiamo `tkinter.Tk.quit()`.

```
def set_application_icons(application, path):
    icon32 = tk.PhotoImage(file=os.path.join(path, "icon_32x32.gif"))
    icon16 = tk.PhotoImage(file=os.path.join(path, "icon_16x16.gif"))
    application.tk.call("wm", "iconphoto", application, "-default", icon32, icon16)
```

Per completezza, riportiamo la funzione `TkUtil.set_application_icons()`. La classe `tk.PhotoImage` può caricare un’immagine pixmap in formato PGM, PPM o GIF (si prevede l’aggiunta del supporto per il formato PNG in Tcl/Tk 8.6). Avendo creato le due immagini, richiamiamo la funzione `tkinter.Tk.tk.call()` e in effetti le inviamo un

comando Tcl/Tk. Si dovrebbe evitare di arrivare a un livello così basso se possibile, ma a volte è necessario laddove Tkinter non leghi la funzionalità che ci occorre.

La classe Main.Window dell'applicazione Currency

La finestra principale dell'applicazione Currency segue il pattern descritto in precedenza e tale pattern è chiaramente visibile nelle chiamate effettuate nel metodo `__init__()` della classe. Tutto il codice di questo paragrafo è tratto da `currency/Main.py`.

```
class Window(ttk.Frame):

def __init__(self, master=None):
    super().__init__(master, padding=2)
    self.create_variables()
    self.create_widgets()
    self.create_layout()
    self.create_bindings()
    self.currencyFromCombobox.focus()
    self.after(10, self.get_rates)
```

È essenziale richiamare la funzione `super()` integrata quando inizializziamo una classe che eredita un widget. Qui, non solo passiamo il master (ossia, l'oggetto applicazione `tk.Tk` della funzione `main()` dell'applicazione), ma anche un valore di riempimento di 2 pixel. Questo riempimento fornisce un margine tra il bordo interno della finestra dell'applicazione e i widget disposti al suo interno.

Successivamente, creiamo le variabili e i widget della finestra (ossia, dell'applicazione) e disponiamo i widget. Quindi, creiamo i binding di evento, dopodiché forniamo il focus tastiera alla casella combinata superiore pronta per la variazione della valuta iniziale da parte dell'utente. Infine, richiamiamo il metodo `after()` di Tkinter, che assume un tempo in millisecondi e un callable che richiamerà dopo che sono trascorsi tali millisecondi.

Poiché scarichiamo i tassi di cambio da Internet, possono essere necessari alcuni secondi prima che arrivino. Tuttavia, vogliamo assicurare che l'applicazione sia visibile direttamente (altrimenti l'utente potrebbe pensare che non si è avviata e provare a riavviarla nuovamente). Così, differiamo l'ottenimento dei tassi di cambio finché l'applicazione non abbia avuto il tempo necessario per visualizzarsi.

```
    def create_variables(self):
        self.currencyFrom = tk.StringVar()
        self.currencyTo = tk.StringVar()
        self.amount = tk.StringVar()
        self.rates = {}
```

Le `tkinter.StringVar` sono variabili che detengono stringhe e che possono essere associate ai widget. Così, quando viene modificata una stringa di `StringVar`, tale modifica viene rispecchiata automaticamente in qualsiasi widget associato, e

viceversa. Avremmo potuto rendere `self.amount` una `tkinter.IntVar`, ma poiché Tcl/Tk opera quasi interamente in termini di stringhe internamente, è spesso più comodo utilizzare stringhe quando si lavora con esso, persino per i numeri. Il `rates` è un `dict` con chiavi dei nomi valute e valori dei tassi di conversione.

```
Spinbox = ttk.Spinbox if hasattr(ttk, "Spinbox") else tk.Spinbox
```

Il widget `tkinter.ttk.Spinbox` non è stato aggiunto a Tkinter di Python 3 ma si spera arrivi con Python 3.4. Questa porzione di codice ci consente di sfruttarlo se disponibile, con il difetto di una casella con pulsanti di selezione senza tema. Le loro interfacce non sono le stesse, perciò occorre prestare attenzione a utilizzare solo quelle funzioni che siano comuni per entrambi.

```
    def create_widgets(self):
        self.currencyFromCombobox = ttk.Combobox(self,
                textvariable=self.currencyFrom)
        self.currencyToCombobox = ttk.Combobox(self,
                textvariable=self.currencyTo)
        self.amountSpinbox = Spinbox(self, textvariable=self.amount,
                from_=1.0, to=10e6, validate="all", format="%0.2f", width=8)
        self.amountSpinbox.config(validatecommand=(
                self.amountSpinbox.register(self.validate), "%P"))
        self.resultLabel = ttk.Label(self)
```

Ogni widget deve essere creato con un genitore (o master), salvo per l’oggetto `tk.Tk`, che solitamente è la finestra o il frame all’interno del quale verrà disposto il widget. Qui, creiamo due caselle combinate e associamo ciascuna di esse al proprio `StringVar`.

Creiamo inoltre una casella con pulsanti di selezione, anch’essa associata a un `StringVar`, con un minimo e un massimo impostati. Il valore di larghezza `width` della casella con pulsanti di selezione è in caratteri; il formato `format` utilizza `%` nel vecchio stile di Python 2 (equivalente a un formato stringa `str.format()` di `"{:0.2f}"`); e l’argomento `validate` indica di validare ogni volta che varia il valore della casella con pulsanti di selezione, sia con immissione dei numeri da parte dell’utente sia con l’utilizzo dei pulsanti di selezione. Una volta creata la casella con pulsanti di selezione, registriamo un callable di validazione, che sarà richiamato con un argomento che corrisponde al formato fornito (`"%P"`); questa è una stringa di formato Tcl/Tk e non Python. Tra l’altro, il valore della casella con pulsanti di selezione è impostato automaticamente al valore minimo (`from_`), in questo caso 1.0, se non è impostato esplicitamente alcun altro valore.

Infine, creiamo l’etichetta che visualizzerà l’importo calcolato, senza assegnare alcun testo iniziale.

```
    def validate(self, number):
        return TkUtil.validate_spinbox_float(self.amountSpinbox, number)
```

Questo è il callable di validazione che abbiamo registrato con la casella con pulsanti di selezione. In questo contesto, il formato `"%P"` Tcl/Tk significa il testo della casella con pulsanti di selezione. Perciò, ogni volta che il valore della casella con pulsanti di selezione viene modificato, questo metodo viene richiamato con il testo della casella con pulsanti di selezione. L'effettiva validazione viene affidata a una comoda funzione generica nel modulo `TkUtil`.

```
def validate_spinbox_float(spinbox, number=None):
    if number is None:
        number = spinbox.get()
    if number == "":
        return True
    try:
        x = float(number)
        if float(spinbox.cget("from")) <= x <= float(spinbox.cget("to")):
            return True
    except ValueError:
        pass
    return False
```

Si prevede che a questa funzione vengano passati una casella con pulsanti di selezione e un valore numerico (come stringa o `None`). Se non viene passato alcun valore, la funzione ottiene il testo stesso della casella con pulsanti di selezione. Restituisce `True` (ossia, "valido") per una casella con pulsanti di selezione vuota, per consentire all'utente di eliminare il valore della casella con pulsanti di selezione e iniziare a digitare un nuovo numero da zero. Altrimenti, cerca di convertire il testo in un numero a virgola mobile e controlla che rientri nell'intervallo della casella con pulsanti di selezione.

Tutti i widget Tkinter hanno un metodo `config()` che accetta uno o più argomenti `key=value` per impostare gli attributi dei widget e un metodo `cget()` che accetta un argomento `key` e restituisce il valore di attributo associato. Hanno anche un metodo `configure()` che è semplicemente un alias per `config()`.

```
    def create_layout(self):
        padWE = dict(sticky=(tk.W, tk.E), padx="0.5m", pady="0.5m")
        self.currencyFromCombobox.grid(row=0, column=0, **padWE)
        self.amountSpinbox.grid(row=0, column=1, **padWE)
        self.currencyToCombobox.grid(row=1, column=0, **padWE)
        self.resultLabel.grid(row=1, column=1, **padWE)
        self.grid(row=0, column=0, sticky=(tk.N, tk.S, tk.E, tk.W))
        self.columnconfigure(0, weight=2)
        self.columnconfigure(1, weight=1)
        self.master.columnconfigure(0, weight=1)
        self.master.rowconfigure(0, weight=1)
        self.master.minsize(150, 40)
```

Questo metodo crea il layout mostrato nella Figura 7.3. Ciascun widget è posto in una specifica posizione di griglia e reso "calamitato" nelle direzioni Ovest ed Est, nel senso che sarà esteso o ristretto orizzontalmente quando la finestra è ridimensionata, ma non cambierà in altezza. Viene inoltre inserito un riempimento di 0,5 mm

(millimetri) attorno ai widget nella direzione x e y, in modo tale che ciascun widget sia circondato da uno spazio vuoto di 0,5 mm (cfr. il riquadro sullo spacchettamento nel Capitolo 1).

(0, 0) currencyFromCombobox	(0, 1) amountSpinbox
(1, 0) currencyToCombobox	(1, 1) resultLabel

Figura 7.3 Il layout della finestra principale dell'applicazione Currency.

Una volta disposti i widget, viene disposta la finestra stessa in una griglia che consiste di una singola cella che si estenderà o restringerà in tutte le direzioni (Nord, Sud, Est, Ovest). Quindi, sono configurate le colonne con i pesi: questi sono i fattori di estensione. In questo caso, quindi, se la finestra viene espansa in senso orizzontale, per ogni pixel di larghezza supplementare fornito alla casella con pulsanti di selezione ed etichetta, le caselle combinate otterranno due pixel di larghezza supplementari. Vengono inoltre forniti pesi non zero alla stessa cella singola della finestra; ciò rende il contenuto della finestra ridimensionabile. Infine, viene fornita alla finestra una dimensione minima sensata; altrimenti, l'utente sarebbe in grado di ridurla pressoché a zero.

```
    def create_bindings(self):
        self.currencyFromCombobox.bind("<<ComboboxSelected>>",
                self.calculate)
        self.currencyToCombobox.bind("<<ComboboxSelected>>",
                self.calculate)
    self.amountSpinbox.bind("<Return>", self.calculate)
    self.master.bind("<Escape>", lambda event: self.quit())
```

Questo metodo viene utilizzato per associare gli eventi alle azioni. Qui siamo interessati a due tipi di evento: "eventi virtuali", ossia eventi personalizzati che sono prodotti da alcuni widget ed "eventi reali", che rappresentano cose che accadono nell'interfaccia utente, come la pressione di un tasto o il ridimensionamento della finestra. Gli eventi virtuali vengono identificati fornendone il nome in doppie parentesi angolari e gli eventi reali fornendone il nome in parentesi angolari singole.

Ogni volta che varia il valore selezionato della casella combinata, aggiunge l'evento virtuale `<<ComboboxSelected>>` alla coda di eventi del ciclo. Per entrambe le caselle combinate, abbiamo scelto di legare questo eventi a un metodo `self.calculate()` che ricalcolerà la conversione di valuta. Per la casella con pulsanti di selezione, imponiamo soltanto un ricalcolo se l'utente preme Invio o Return. Se invece l'utente preme Esc, usciamo dall'applicazione richiamando il metodo ereditato `tkinter.ttk.Frame.quit()`.

```
    def calculate(self, event=None):
        fromCurrency = self.currencyFrom.get()
```

```
        toCurrency = self.currencyTo.get()
        amount = self.amount.get()
        if fromCurrency and toCurrency and amount:
            amount = ((self.rates[fromCurrency] / self.rates[toCurrency]) *
                    float(amount))
            self.resultLabel.config(text="{:,.2f}".format(amount))
```

Questo metodo ottiene le due valute da utilizzare e l'importo da convertire, quindi esegue la conversione. Alla fine, imposta il testo dell'etichetta del risultato all'importo convertito, utilizzando virgole come separatore delle migliaia e mostrando due cifre dopo il punto decimale.

```
    def get_rates(self):
        try:
            self.rates = Rates.get()
            self.populate_comboboxes()
        except urllib.error.URLError as err:
            messagebox.showerror("Currency \u2014 Error", str(err),
                    parent=self)
            self.quit()
```

Questo metodo viene richiamato mediante un timer per fornire alla finestra la possibilità di ridisegnarsi. Ottiene un dizionario di tassi (chiavi di nomi di valuta, valori di fattori di conversione) e compila di conseguenza le caselle combinate. Se non è possibile ottenere i tassi, appare un messaggio di errore e dopo la chiusura di tale messaggio da parte dell'utente (per esempio facendo clic su *OK*), l'applicazione viene chiusa.

La funzione `tkinter.messagebox.showerror()` accetta un titolo finestra, un testo di messaggio e opzionalmente un genitore (se è fornito, la casella del messaggio vi si centrerà sopra). Poiché i file Python 3 utilizzano la codifica UTF-8, avremmo potuto utilizzare un carattere di trattino em (—), ma abbiamo preferito utilizzare invece l'escape Unicode per maggiore chiarezza in stampa.

```
    def populate_comboboxes(self):
        currencies = sorted(self.rates.keys())
        for combobox in (self.currencyFromCombobox,
                self.currencyToCombobox):
            combobox.state(("readonly",))
            combobox.config(values=currencies)
        TkUtil.set_combobox_item(self.currencyFromCombobox, "USD", True)
        TkUtil.set_combobox_item(self.currencyToCombobox, "GBP", True)
        self.calculate()
```

Questo metodo compila le caselle combinate con i nomi delle valute in ordine alfabetico. Le caselle combinate sono impostate in modo che siano di sola lettura. Tentiamo quindi di impostare la valuta della casella con pulsanti di selezione su dollari U.S. e quella inferiore su sterline inglesi. Infine, richiamiamo `self.calculate()` per impostare un valore di conversione iniziale.

Ogni widget con tema Tkinter possiede un metodo `state()` per impostare uno o più stati e un metodo `instate()` per controllare se il widget si trova in uno stato particolare.

Gli stati utilizzati più comunemente sono `"disabled"`, `"readonly"` e `"selected"`.

```
def set_combobox_item(combobox, text, fuzzy=False):
    for index, value in enumerate(combobox.cget("values")):
        if (fuzzy and text in value) or (value == text):
            combobox.current(index)
            return
    combobox.current(0 if len(combobox.cget("values")) else -1)
```

Questa funzione generica si trova nel modulo `TkUtil`. Tenta di impostare il valore della casella combinata dato alla voce che ha il testo fornito - o alla voce che contiene il testo dato se `fuzzy` è `True`.

Questa semplice ma utile applicazione per le valute ha circa 200 righe di codice (esclusi i moduli di libreria standard o il modulo `TkUtil` del libro). È assai comune per le piccole utilità GUI necessitare di molto più codice rispetto alle equivalenti a riga di comando, ma la disparità diminuisce rapidamente con applicazioni più complesse e sofisticate.

Creazione di finestre di dialogo di applicazione

La creazione di applicazioni indipendenti a finestra è assai semplice e comoda per piccoli programmi di utilità, lettori multimediali e per alcuni giochi. Ma per applicazioni più complesse, è solito avere una finestra principale e finestre di dialogo di supporto. In questo paragrafo, vedremo come creare una finestra di dialogo modale e una non modale.

Per quanto concerne widget, layout e associazioni non vi è alcuna differenza tra finestre modali e non modali. Tuttavia, mentre le finestre di dialogo modali assegnano generalmente ciò che immette l’utente a delle variabili, quelle non modali richiamano metodi di applicazione o modificano i dati dell’applicazione in risposta alle interazioni dell’utente. Inoltre, le finestre di dialogo modali comportano un blocco quando vengono richiamate, mentre non lo fanno quelle non modali, differenza di cui il nostro codice deve tenere conto.

Creazione di finestre di dialogo modali

In questo paragrafo esamineremo la finestra di dialogo delle preferenze dell’applicazione Gravitate. Il codice della finestra di dialogo si trova in `gravitate/Preferences.py` e la finestra di dialogo è illustrata nella Figura 7.4.

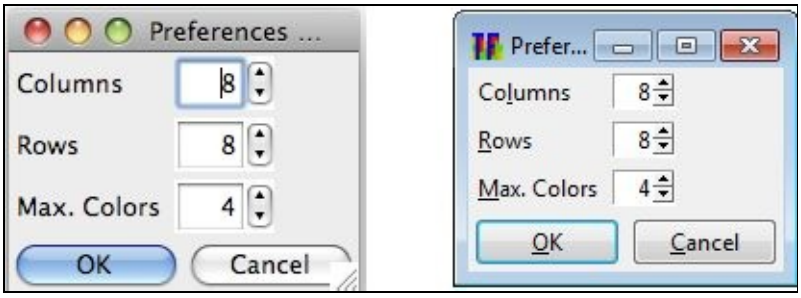


Figura 7.4 La finestra di dialogo modale per le preferenze dell'applicazione Gravitate su OS X e Windows.

Su Linux e Windows, quando l'utente fa clic sull'opzione di menu *File* e seleziona *Preferences* in Gravitate, viene richiamato il metodo `Main.Window.preferences()` e ciò determina la comparsa della finestra di dialogo modale delle preferenze. Su OS X l'utente deve fare clic sull'opzione *Preferences* del menu dell'applicazione, secondo le convenzioni del sistema (purtroppo, però, dobbiamo pensare noi a gestire entrambi i casi, come vedremo più avanti in questo capitolo).

```
def preferences(self):
    Preferences.Window(self, self.board)
    self.master.focus()
```

Questo è il metodo di finestra principale che richiama la finestra di dialogo delle preferenze. Si tratta di una finestra di dialogo intelligente perciò, invece di passarle qualche valore e poi, quando l'utente fa clic su *OK*, aggiornare lo stato dell'applicazione, le passiamo direttamente un oggetto di applicazione, in questo caso `self.board` di tipo `Board`, una sottoclasse `tkinter.Canvas` per mostrare la grafica 2D.

Il metodo crea una nuova finestra di dialogo di preferenze. Questa chiamata determina la visualizzazione della finestra di dialogo e il blocco (poiché la finestra di dialogo è modale) finché l'utente non fa clic su *OK* o *Cancel*. Qui non dobbiamo eseguire alcuna ulteriore operazione, poiché la finestra di dialogo stessa è abbastanza intelligente da aggiornare l'oggetto `Board` se l'utente fa clic su *OK*. Una volta chiusa a finestra di dialogo, tutto ciò che facciamo è accertarci che la finestra principale abbia il focus di tastiera.

Tkinter integra il modulo `tkinter.simpledialog` che fornisce un paio di classi base per la creazione di finestre di dialogo personalizzate, e alcune comode funzioni pronte all'uso per far comparire finestre di dialogo atte a ottenere singoli valori da parte dell'utente, quale `tkinter.simpledialog.askfloat()`. Le finestre di dialogo pronte all'uso forniscono alcuni agganci integrati per agevolarne al massimo l'ereditarietà e per personalizzarle con i nostri widget. In ogni caso, al momento della stesura di questo volume, non hanno subito da tempo alcun aggiornamento e non utilizzano i widget con tema. Per tale ragione, gli esempi del libro prevedono l'uso del modulo

`TkUtil/Dialog.py` che fornisce una classe base per finestre di dialogo personalizzate con tema e che funziona in modo analogo a `tkinter.simpledialog.Dialog`, e che fornisce inoltre alcune comode funzioni, quale `TkUtil.Dialog.get_float()`.

Tutte le finestre di dialogo del libro adottano il modulo `TkUtil` invece di `tkinter.simpledialog`, così da poter sfruttare i widget con tema che conferiscono a Tkinter un aspetto nativo su OS X e Windows.

```
class Window(TkUtil.Dialog.Dialog):

def __init__(self, master, board):
self.board = board
super().__init__(master, "Preferences \u2014 {}".format(APPNAME),
TkUtil.Dialog.OK_BUTTON|TkUtil.Dialog.CANCEL_BUTTON)
```

La finestra di dialogo assume un genitore (`master`) e un'istanza `Board`. Questa istanza sarà utilizzata per fornire i valori iniziali per i widget della finestra di dialogo, e se l'utente fa clic su *OK*, saranno forniti al widget i valori che l'utente ha impostato prima che la finestra di dialogo si distrugga. La costante `APPNAME` (non mostrata qui) detiene la stringa `"Gravitate"`.

Le classi che ereditano `TkUtil.Dialog.Dialog` devono fornire un metodo `body()` che crei i widget della finestra di dialogo ma non i relativi pulsanti: questo lo fa la classe di base. Devono inoltre fornire un metodo `apply()` che verrà richiamato solo se l'utente accetta la finestra di dialogo (ossia, se fa clic su *OK* o *Sì*, in funzione di quale pulsante di "accettazione" è stato specificato). È inoltre possibile creare un metodo `initialize()` e un metodo `validate()`, che però non sono necessari per questo esempio.

```
    def body(self, master):
        self.create_variables()
        self.create_widgets(master)
        self.create_layout()
        self.create_bindings()
    return self.frame, self.columnsSpinbox
```

Questo metodo deve creare le variabili della finestra di dialogo, disporre i widget della finestra di dialogo e fornire i legami di vento (con l'esclusione dei pulsanti e relative associazioni). Deve restituire il widget che contiene tutti i widget che abbiamo creato (generalmente un frame) oppure quel widget più il widget al quale deve essere fornito il focus di tastiera iniziale. Qui restituiamo il frame in cui dobbiamo mettere tutti i nostri widget e la prima casella con pulsanti di selezione come widget con focus di tastiera iniziale.

```
    def create_variables(self):
        self.columns = tk.StringVar()
        self.columns.set(self.board.columns)
        self.rows = tk.StringVar()
        self.rows.set(self.board.rows)
        self.maxColors = tk.StringVar()
        self.maxColors.set(self.board.maxColors)
```

Questa finestra di dialogo è molto semplice, poiché utilizza solo etichette e caselle con pulsanti di selezione. Per ogni casella con pulsanti di selezione, creiamo un `tkinter.StringVar` da associare a essa e inizializziamo il valore di `StringVar` con il valore corrispondente nell'istanza passata in `Board`. Potrebbe sembrare più naturale utilizzare i `tkinter.IntVar`, tuttavia Tcl/Tk utilizza internamente solo le stringhe, perciò gli `StringVar` sono spesso una scelta migliore.

```
    def create_widgets(self, master):
        self.frame = ttk.Frame(master)
        self.columnsLabel = TkUtil.Label(self.frame, text="Columns",
                underline=2)
        self.columnsSpinbox = Spinbox(self.frame,
                textvariable=self.columns, from_=Board.MIN_COLUMNS,
                to=Board.MAX_COLUMNS, width=3, justify=tk.RIGHT,
                validate="all")
        self.columnsSpinbox.config(validatecommand=(
        self.columnsSpinbox.register(self.validate_int),
                "columnsSpinbox", "%P"))
        ...
```

Questo metodo viene utilizzato per creare i widget. Iniziamo creando un frame esterno che possiamo restituire come widget genitore per tutti gli altri widget che creiamo. Il genitore del frame deve essere quello fornito dalla finestra di dialogo; tutti gli altri widget che creiamo devono avere il frame (o un figlio del frame) come genitore.

Abbiamo mostrato il codice solo per i widget di colonne, poiché il codice delle righe e dei colori massimi è strutturalmente identico. In ciascun caso, creiamo un'etichetta e una casella con pulsanti di selezione e a ognuna di queste ultime è associata al proprio `StringVar` corrispondente. L'attributo `width` è il numero di larghezza in caratteri della casella con pulsanti di selezione.

Per inciso, onde evitare l'inconveniente di dover scrivere, per esempio, `underline=-1 if TkUtil.mac() else 0` quando si creano le etichette, abbiamo utilizzato `TkUtil.Label` invece di `tkinter.ttk.Label`.

```
class Label(ttk.Label):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        if mac():
            self.config(underline=-1)
```

Questa piccola classe ci consente di impostare la lettera sottolineata per indicare la scelta rapida da tastiera e di non preoccuparci del fatto che il codice sia eseguito o meno su OS X, poiché in tal caso la sottolineatura verrebbe disabilitata impostando il valore `-1`. Anche il modulo `TkUtil/__init__.py` presenta le classi `Button`, `Checkbutton` e `Radiobutton`, tutte con lo stesso metodo `__init__()` come quello mostrato qui.

```
    def validate_int(self, spinboxName, number):
        return
                TkUtil.validate_spinbox_int(getattr(self, spinboxName), number)
```

Abbiamo discusso come validare le caselle con pulsanti di selezione e la funzione `TkUtil.validate_spinbox_float()` più indietro in questo capitolo. L'unica differenza tra il metodo `validate_int()` utilizzato qui e il metodo `validate()` utilizzato in precedenza (a parte il nome e il fatto che qui eseguiamo la convalida di interi) è che qui classifichiamo mediante la casella con pulsanti di selezione per convalidare, mentre nell'esempio precedente abbiamo utilizzato una casella con pulsanti di selezione specifica.

Alla funzione di convalida registrata sono state fornite due stringhe: la prima, il nome della casella con pulsanti di selezione relativa e la seconda, una stringa di formato Tcl/Tk. Queste vengono passate alla funzione di convalida quando ha luogo l'operazione di convalida e Tcl/Tk ne esegue l'analisi. Nel caso del nome della casella con pulsanti di selezione, Tcl/Tk non esegue alcuna operazione, ma sostituisce `"%P"` con il valore stringa della casella con pulsanti di selezione. La funzione `TkUtil.validate_spinbox_int()` richiede un widget di casella con pulsanti di selezione e un valore stringa come argomenti. Perciò, qui utilizziamo la funzione integrata `getattr()`, passandole la finestra di dialogo (`self`) e il nome dell'attributo che desideriamo (`spinboxName`), e ottenendo un riferimento per il relativo widget della casella con pulsanti di selezione.

```
    def create_layout(self):
        padW = dict(sticky=tk.W, padx=PAD, pady=PAD)
        padWE = dict(sticky=(tk.W, tk.E), padx=PAD, pady=PAD)
        self.columnsLabel.grid(row=0, column=0, **padW)
        self.columnsSpinbox.grid(row=0, column=1, **padWE)
        self.rowsLabel.grid(row=1, column=0, **padW)
        self.rowsSpinbox.grid(row=1, column=1, **padWE)
        self.maxColorsLabel.grid(row=2, column=0, **padW)
        self.maxColorsSpinbox.grid(row=2, column=1, **padWE)
```

Questo metodo crea il layout illustrato nella Figura 7.5. È molto semplice, perché dispone sulla griglia tutte le etichette allineandole a sinistra (`sticky=tk.W`, ossia, West, cioè Ovest) e tutte le caselle con pulsanti di selezione per riempire tutto lo spazio orizzontale disponibile, mentre aggiunge uno spazio di riempimento per tutti i widget pari a 0,75 mm (la costante `PAD`, non mostrata qui). Cfr. il riquadro sullo spacchettamento nel Capitolo 1.

(0, 0) columnsLabel	(0, 1) columnsSpinbox
(1, 0) rowsLabel	(1, 1) rowsSpinbox
(2, 0) maxColorsLabel	(2, 1) maxColorsSpinbox

Figura 7.5 Layout del corpo della finestra di dialogo delle preferenze nell'applicazione Gravitate.

```
    def create_bindings(self):
        if not TkUtil.mac():
            self.bind("<Alt-l>", lambda *args:
            self.columnsSpinbox.focus()) self.bind("<Alt-r>", lambda *args:
            self.rowsSpinbox.focus()) self.bind("<Alt-m>",
                    lambda *args: self.maxColorsSpinbox.focus())
```

Per piattaforme non OS X, è opportuno fornire agli utenti la capacità di navigare tra le caselle con pulsanti di selezione e fare clic sui pulsanti mediante scelte rapida da tastiera. Per esempio, se l'utente preme Alt+R, la casella con pulsanti di selezione delle righe otterrà il focus di tastiera. Non dobbiamo farlo per i pulsanti, poiché se ne prende cura la classe di base.

```
    def apply(self):
        columns = int(self.columns.get())
        rows = int(self.rows.get())
        maxColors = int(self.maxColors.get())
        newGame = (columns != self.board.columns or
                   rows != self.board.rows or
                   maxColors != self.board.maxColors)
        if newGame:
            self.board.columns = columns
            self.board.rows = rows
            self.board.maxColors = maxColors
            self.board.new_game()
```

Questo metodo viene richiamato solo se l'utente fa clic sul pulsante di "accettazione" (*OK* o *Sì*) della finestra di dialogo. Recuperiamo i valori di `StringVar` e li convertiamo in `int` (che dovrebbe sempre avere esito positivo). Quindi li assegniamo agli attributi corrispondenti all'istanza `Board`. Se qualsiasi valore è variato, iniziamo un nuovo gioco, così da tenere conto della variazione.

In applicazioni grandi e complesse potrebbe essere necessario procedere a lungo con la navigazione - fare clic su opzioni di menu e richiamare finestre di dialogo - prima di raggiungere la finestra di dialogo da provare. Per rendere la prova più facile, è spesso utile aggiungere una dichiarazione `if __name__== "__main__":` alla fine di un modulo che contiene una classe di finestre, e inserire codice che richiami la finestra di dialogo a scopo di test. Di seguito è riportato il codice all'interno di tale dichiarazione per il modulo `gravitate/Preferences.py`.

```
    def close(event):
        application.quit()
    application = tk.Tk()
    scoreText = tk.StringVar()
    board = Board.Board(application, print, scoreText)
    window = Window(application, board)
    application.bind("<Escape>", close)
    board.bind("<Escape>", close)
    application.mainloop()
    print(board.columns, board.rows, board.maxColors)
```

Iniziamo creando una funzione che chiuda l'applicazione. Quindi creiamo l'oggetto `tk.Tk` normalmente nascosto (ma visibile in questo caso) che funga da

genitore definitivo dell'applicazione. Associamo il tasto Esc alla funzione `close()` in modo tale che l'utente possa chiudere facilmente la finestra.

L'istanza `Board` viene passata normalmente alla finestra di dialogo dalla finestra principale richiamante, ma poiché qui eseguiamo la finestra di dialogo indipendente, dobbiamo creare noi tale istanza.

Successivamente, creiamo la finestra di dialogo, che attuerà un blocco finché l'utente non fa clic su *OK* o *Cancel*. Ovviamente, la finestra di dialogo appare solo una volta che il ciclo di eventi è iniziato. E una volta che la finestra di dialogo si è chiusa, stampiamo i valori degli attributi `Board` che la finestra di dialogo può cambiare. Se l'utente ha fatto clic su *OK*, questi devono riflettere eventuali variazioni che l'utente ha fatto nella finestra di dialogo; altrimenti, devono avere tutti i loro valori originali.

Creazione di finestre di dialogo non modali

In questo paragrafo esamineremo la finestra di dialogo non modale della Guida dell'applicazione Gravitate, mostrata nella Figura 7.6.

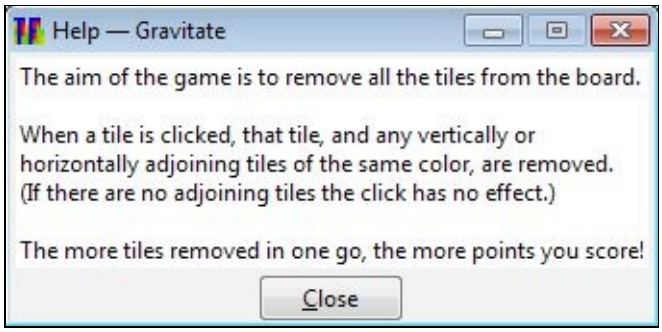


Figura 7.6 La finestra di dialogo non modale della Guida dell'applicazione Gravitate su Windows

Come abbiamo detto in precedenza, per i widget, i layout e le associazioni di evento, non vi è alcuna differenza tra le finestre di dialogo modali e quelle non modali. Ciò che distingue i due tipi è che quando è visualizzata una finestra di dialogo non modale questa non si blocca (mentre lo fa una modale), così il chiamante (per esempio la finestra principale) continua a eseguire il proprio ciclo di eventi ed è possibile interagire con esso. Inizieremo osservando il codice che richiama la finestra di dialogo, quindi il codice della finestra di dialogo.

```
def help(self, event=None):
    if self.helpDialog is None:
        self.helpDialog = Help.Window(self)
    else:
        self.helpDialog.deiconify()
```

Questo è il metodo `Main.Window.help()`. `Main.Window` mantiene una variabile di istanza (`self.helpDialog`) impostata su `None` nel metodo `__init__()` (non mostrato qui). La prima volta che l'utente richiama la finestra della Guida, la finestra di dialogo viene creata e passata alla finestra principale come genitore. L'atto della creazione del widget ne determina la comparsa sulla finestra principale, ma poiché la finestra di dialogo è non modale, il ciclo di eventi della finestra principale riprende e l'utente può interagire sia con la finestra di dialogo sia con la finestra principale.

La seconda e le volte successive che l'utente richiama la finestra di dialogo della Guida, abbiamo già un riferimento a essa, così la mostriamo semplicemente di nuovo (usando `tkinter.Toplevel.deiconify()`). Ciò funziona perché quando l'utente chiude la finestra di dialogo, invece di distruggersi, la finestra di dialogo semplicemente si nasconde. Creare, visualizzare e nascondere una finestra di dialogo la prima volta che la si utilizza, e poi mostrarla e nasconderla negli utilizzi successivi, è molto più veloce che crearla, visualizzarla e poi distruggerla ogni volta che è necessario. Inoltre, nascondere una finestra di dialogo ne preserva lo stato tra i vari utilizzi.

```
class Window(tk.Toplevel):

    def __init__(self, master):
        super().__init__(master)
        self.withdraw()
        self.title("Help \u2014 {}".format(APPNAME))
        self.create_ui()
        self.reposition()
        self.resizable(False, False)
        self.deiconify()
        if self.winfo_viewable():
            self.transient(master)
        self.wait_visibility()
```

Le finestre di dialogo non modali ereditano solitamente `tkinter.ttk.Frame` o `tkinter.Toplevel`, così come abbiamo fatto qui. La finestra di dialogo assume un genitore (`master`). La chiamata di `tkinter.Toplevel.withdraw()` nasconde immediatamente la finestra (persino prima che l'utente la veda) per assicurare che non vi sia alcun sfarfallio mentre la finestra viene creata.

In seguito, impostiamo il titolo della finestra a "Guida - Gravitate", quindi creiamo i widget della finestra di dialogo. Poiché il testo della guida è troppo breve, abbiamo impostato la finestra di dialogo in modo che non sia ridimensionabile e abbiamo lasciato a Tkinter il compito di renderla della corretta dimensione per mostrare il testo della guida e il pulsante di chiusura. Se avessimo parecchio testo della guida, avremmo potuto utilizzare una sottoclasse `tkinter.Text` con barre di scorrimento e rendere la finestra di dialogo ridimensionabile.

Una volta che tutti i widget sono stati creati e disposti, richiamiamo `tkinter.Toplevel. deiconify()` per visualizzare la finestra. Se la finestra è visualizzabile (ossia, mostrata dal gestore delle finestre del sistema) - così come dovrebbe essere - notifichiamo a Tkinter che questa finestra è transitoria in relazione al proprio genitore. Questa notifica fornisce un suggerimento al sistema delle finestre che la finestra transitoria potrebbe presto scomparire per aiutare a ottimizzarne il ridisegno di ciò che viene rivelato quando è nascosta o distrutta.

La chiamata di `tkinter.Toplevel.wait_visibility()` alla fine attua un blocco (per un tempo troppo breve perché l'utente se ne accorga) finché la finestra è visibile. Per impostazione predefinita, le finestre `tkinter.Toplevel` sono non modali, ma se aggiungiamo due istruzioni supplementari dopo l'ultima, possiamo rendere la finestra modale. Tali istruzioni sono `self.grab_set()` e `self.wait_window(self)`. La prima istruzione limita il focus dell'applicazione ("grab" nella terminologia Tk/Tcl) a questa finestra, rendendola così modale. La seconda istruzione attua un blocco finché la finestra non si chiude. Non abbiamo visto né l'uno né l'altro di questi comandi quando abbiamo illustrato le finestre di dialogo modali, perché è un pattern standard a creare una finestra modale ereditando `tkinter.simpledialog.Dialog` (oppure in questo libro, `TkUtil.Dialog`), entrambi i quali hanno queste due dichiarazioni.

L'utente può ora interagire con questa finestra, con la finestra principale dell'applicazione e con qualsiasi altra finestra modale dell'applicazione che risulti visibile.

```
def create_ui(self):
    self.helpLabel = ttk.Label(self, text=_TEXT, background="white")
    self.closeButton = TkUtil.Button(self, text="Close", underline=0)
    self.helpLabel.pack(anchor=tk.N, expand=True, fill=tk.BOTH,
            padx=PAD, pady=PAD) self.closeButton.pack(anchor=tk.S)
    self.protocol("WM_DELETE_WINDOW", self.close)
    if not TkUtil.mac():
        self.bind("<Alt-c>", self.close)
    self.bind("<Escape>", self.close)
    self.bind("<Expose>", self.reposition)
```

L'interfaccia utente della finestra è così semplice che l'abbiamo creata tutta in quest'unico metodo. Anzitutto, creiamo l'etichetta per visualizzare il testo della guida (nella costante `_TEXT`, non mostrata qui), quindi il pulsante *Close*. Abbiamo utilizzato un `TkUtil.Button` (derivato da `tkinter.ttk.Button`) in modo tale che la sottolineatura sia correttamente ignorata su OS X (in precedenza abbiamo visto una sottoclasse `TkUtil.Label` pressoché identica a `TkUtil.Button`).

Con soli due widget, ha senso utilizzare il gestore di layout più semplice, perciò qui posizioniamo l'etichetta nella parte superiore della finestra, la impostiamo in

modo che possa aumentare in entrambe le direzioni, e applichiamo il pulsante nella parte inferiore.

Se non ci troviamo su OS X, aggiungiamo un tasto rapido Alt+C per il pulsante *Close* e, su tutte le piattaforme, associamo il tasto Esc alla chiusura della finestra.

Poiché la finestra di dialogo non modale viene nascosta e mostrata invece che distrutta e ricreata, è possibile che l'utente visualizzi la finestra della Guida, quindi la chiuda (ossia, la nasconda), quindi passi nella finestra principale, quindi visualizzi nuovamente la finestra della Guida. È perfettamente ragionevole lasciare la finestra della Guida ovunque sia stata visualizzata prima (oppure, ovunque l'utente l'abbia spostata per ultimo). Tuttavia, poiché il testo della Guida è così breve, sembra che abbia più senso riposizionarlo ogni volta che viene mostrato. Ciò si ottiene associando l'evento `<Expose>` (che si verifica ogni volta che una finestra deve ridisegnarsi) a un metodo `reposition()` personalizzato.

```
def reposition(self, event=None):
    if self.master is not None:
        self.geometry("+{}+{}".format(self.master.winfo_rootx() + 50,
            self.master.winfo_rooty() + 50))
```

Questo metodo sposta la finestra sulla stessa posizione del proprio master (ossia, la finestra principale), ma con un offset di 50 pixel a destra e verso il basso.

In teoria non dovremmo richiamare questo metodo esplicitamente nel metodo `__init__()`, ma così facendo, garantiamo che la finestra sia posizionata correttamente prima di essere mostrata. Ciò evita che la finestra salti improvvisamente in posizione dopo la sua prima apparizione, poiché quando viene mostrata, si trova già nel posto giusto.

```
def close(self, event=None):
    self.withdraw()
```

Se la finestra di dialogo viene chiusa - dall'utente premendo Esc o Alt+C, facendo clic sul pulsante *Close* o sul pulsante di chiusura × - questo metodo viene richiamato. Il metodo nasconde semplicemente la finestra invece di distruggerla. La finestra può essere visualizzata di nuovo richiamando `tkinter.Toplevel.deiconify()` su di essa.

Creazione di applicazioni a finestra principale con Tkinter

In questo paragrafo vediamo come creare gli aspetti più generali dell'applicazione con stile a finestre di Gravitate. L'applicazione è mostrata nella Figura 7.7 e il gioco è descritto nel riquadro dedicato a Gravitate.

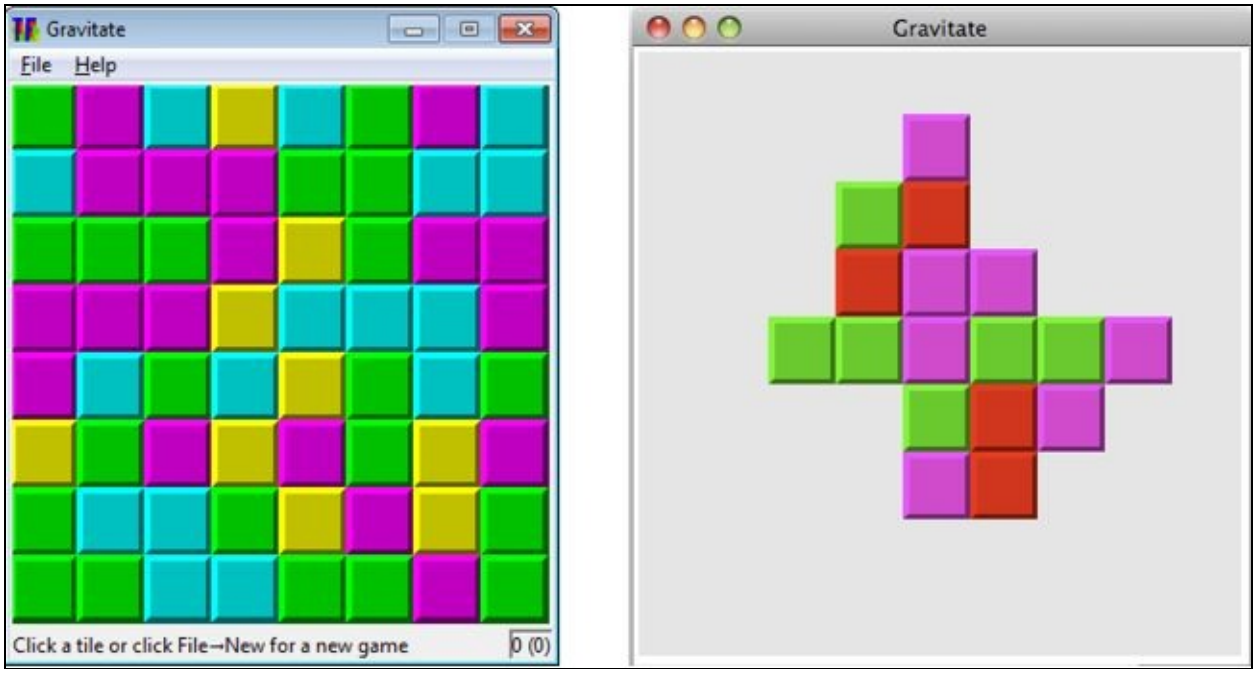


Figura 7.7 L'applicazione Gravitate su Windows e OS X.

L'interfaccia utente presenta alcuni degli elementi standard che gli utenti si aspetterebbero, come una barra del menu, un widget centrale, una barra di stato e finestre di dialogo. Tkinter fornisce il supporto per i menu, ma dobbiamo creare da noi il widget centrale e la barra di stato. Dovrebbe essere assai semplice adattare il codice dell'applicazione Gravitate per creare altre applicazioni a finestra principale, modificando il widget nell'area centrale e i menu e la barra di stato, mantenendo però la stessa struttura complessiva.

Gravitate consiste di sette file Python e di nove immagini di icona. L'eseguibile dell'applicazione è `gravitate/gravitate.pyw` e la finestra principale si trova in `gravitate/Main.py`. Questi sono supportati da tre finestre di dialogo: `gravitate/About.py`, che non tratteremo; `gravitate/Help.py`, che abbiamo illustrato nel paragrafo precedente; e `gravitate/Preferences.py`, anch'esso trattato in precedenza in questo capitolo. L'area centrale della finestra principale è occupata da un `Board` di `gravitate/Board.py`, una sottoclasse di `tkinter.Canvas` (che non abbiamo spazio per illustrare qui).

```
def main():
    application = tk.Tk()
```

```
application.withdraw()
application.title(APPNAME)
application.option_add("*tearOff", False)
TkUtil.set_application_icons(application, os.path.join(
        os.path.dirname(os.path.realpath(__file__)), "images"))
window = Main.Window(application)
application.protocol("WM_DELETE_WINDOW", window.close)
application.deiconify()
application.mainloop()
```

GRAVITATE

Lo scopo del gioco è rimuovere tutte le mattonelle dalla tavola. Quando si fa clic su una mattonella, questa e qualsiasi altra mattonella adiacente orizzontalmente o verticalmente dello stesso colore vengono rimosse (se non vi sono mattonelle adiacenti, il clic non ha alcun effetto). Più mattonelle si rimuovono in un'unica mossa, più punti si ottengono.

La logica di Gravitate è simile a quella del gioco Tile Fall o SameGame. La differenza chiave tra Gravitate e gli altri due è che, quando le mattonelle vengono rimosse in Tile Fall o SameGame, cadono e si spostano a sinistra per riempire eventuali spazi, mentre con Gravitate le mattonelle "gravitano" verso il centro della tavola.

Gli esempi del libro comprendono tre versioni di Gravitate. La prima versione si trova nella directory `gravitate` ed è descritta qui. La seconda versione si trova nella directory `gravitate2`: ha la stessa logica di gioco della prima versione, e in aggiunta presenta una barra degli strumenti occultabile/visualizzabile e una finestra di dialogo delle preferenze più sofisticata, che fornisce opzioni aggiuntive, quale la scelta della forma delle mattonelle e di un fattore di zoom per ingrandire o ridurre la visualizzazione delle stesse. Inoltre, `gravitate2` memorizza i punteggi più alti tra le sessioni e può essere giocato sia mediante la tastiera sia con il mouse, esplorato con i tasti freccia e rimosso premendo la barra spaziatrice. La terza versione è tridimensionale ed è illustrata nel Capitolo 8. Esiste anche una versione online presso `http://www.qtrac.eu/gravitate.html`.

Questa è la funzione `main()` del file `gravitate/gravitate.pyw`. Crea l'oggetto di primo livello `tkinter.Tk` normalmente nascosto, quindi nasconde immediatamente l'applicazione per evitare sfarfallio mentre viene creata la finestra principale. Per default Tkinter ha un menu di tipo *tear-off* (sorta di pop-up liberamente spostabili, un ritorno al passato verso l'antica GUI Motif); abbiamo disattivato la caratteristica, poiché nessuna GUI moderna ne fa uso. In seguito, abbiamo impostato le icone dell'applicazione usando una funzione che abbiamo illustrato precedentemente. Creiamo quindi la finestra principale dell'applicazione, e istruiamo Tkinter in modo che, se viene fatto clic sul pulsante di chiusura dell'applicazione ×, richiami il metodo `Main.Window.close()`. Infine, mostriamo l'applicazione (o meglio, aggiungiamo un evento al ciclo di eventi per programmarne la visualizzazione), quindi avviamo il ciclo di eventi. A questo punto apparirà l'applicazione in questione.

Creazione di una finestra principale

Le finestre principali di Tkinter sono in linea di principio, non diverse dalle finestre di dialogo. Nella pratica, tuttavia, hanno una barra dei menu e una barra di

stato, spesso delle barre degli strumenti e talvolta delle finestre agganciabili. Solitamente, inoltre, hanno un unico widget centrale - un editor di testi, una tabella (per esempio nel caso di un foglio elettronico) o un grafico (per esempio per un gioco o una simulazione o una visualizzazione. Nel caso di Gravitate abbiamo una barra dei menu, un widget grafico centrale e una barra di stato.

```
class Window(ttk.Frame):

    def __init__(self, master):
        super().__init__(master, padding=PAD)
        self.create_variables()
        self.create_images()
        self.create_ui()
```

La classe `Main.Window` di Gravitate deriva da `tkinter.ttk.Frame` e affida la maggior parte del lavoro alla classe base e a tre metodi ausiliari.

```
    def create_variables(self):
        self.images = {}
        self.statusText = tk.StringVar()
        self.scoreText = tk.StringVar()
        self.helpDialog = None
```

I messaggi di testo per la barra di stato sono memorizzati in `self.statusText`, e il testo del punteggio permanente (e del più alto) in `self.scoreText`. La finestra di dialogo della Guida è inizialmente impostata a `None` (cfr. il precedente paragrafo di questo capitolo dedicato a questo argomento).

È molto comune che le applicazioni GUI visualizzino delle icone accanto alle opzioni di menu, e comunque le icone sono essenziali per i pulsanti delle barre degli strumenti. Per il gioco Gravitate abbiamo inserito tutte le immagini delle icone nella sottodirectory `gravitate/images` e abbiamo definito un insieme di costanti per i loro nomi (per esempio la costante `NEW` è impostata alla stringa `"New"`). Quando la `Main.Window` viene creata, richiama un metodo personalizzato `create_images()` per caricare tutte le immagini necessarie come valori del dizionario `self.images`. È essenziale mantenere i riferimenti alle immagini caricate da Tkinter, altrimenti saranno affidate alla garbage collection (e scompariranno).

```
    def create_images(self):
        imagePath = os.path.join(os.path.dirname(
                os.path.realpath(__file__)), "images")
        for name in (NEW, CLOSE, PREFERENCES, HELP, ABOUT):
            self.images[name] = tk.PhotoImage(
                    file=os.path.join(imagePath, name + "_16x16.gif"))
```

Abbiamo scelto di usare nei menu immagini di 16 × 16 pixel, perciò per ogni costante di azione (`NEW`, `CLOSE` e così via), carichiamo le immagini appropriate.

La costante integrata `__file__` contiene il nome di file con il percorso. Utilizziamo `os.path.realpath()` per ottenere il percorso assoluto ed eliminare componenti "`..`" e link simbolici, poi estraiamo soltanto la porzione con la directory (scartando il nome di file) e la uniamo con `"images"` per ottenere il percorso alla sottodirectory `images` dell'applicazione.

```
    def create_ui(self):
        self.create_board()
        self.create_menubar()
        self.create_statusbar()
        self.create_bindings()
        self.master.resizable(False, False)
```

Grazie al nostro approccio al refactoring, questo metodo affida il lavoro ad altri metodi ausiliari. E poi, quando l'interfaccia utente è completa, imposta la finestra in modo che non sia ridimensionabile. Dopo tutto, non ha senso che l'utente attui un ridimensionamento quando le mattonelle hanno dimensione fissa (anche l'applicazione Gravitate 2 non consente il ridimensionamento, ma consente agli utenti di cambiare la dimensione delle mattonelle e ridimensiona di conseguenza la finestra).

```
def create_board(self):
self.board = Board.Board(self.master, self.set_status_text, self.scoreText)
self.board.update_score()
self.board.pack(fill=tk.BOTH, expand=True)
```

Questo metodo crea un'istanza `Board` (una sottoclasse di `tkinter.Canvas`) e passa il metodo `self.set_status_text()` in modo che possa visualizzare messaggi transitori nella barra di stato della finestra principale, e il `self.scoreText` in modo che possa aggiornare il punteggio.

Una volta creata la tavola, richiamiamo il suo metodo `update_score()` per visualizzare "0 (0)" nell'indicatore di punteggio permanente. Inoltre inseriamo la tavola nella finestra principale, indicando di espanderla in entrambe le direzioni.

```
    def create_bindings(self):
        modifier = TkUtil.key_modifier()
        self.master.bind("<{}-n>".format(modifier), self.board.new_game)
        self.master.bind("<{}-q>".format(modifier), self.close)
        self.master.bind("<F1>", self.help)
```

Qui forniamo tre tasti di scelta rapida: Ctrl+N (o .N) per avviare una nuova partita, Ctrl+Q (o .Q) per uscire e F1 per aprire (o visualizzare, se è nascosta) la finestra della Guida. Il metodo `TkUtil.key_modifier()` restituisce il nome del modificatore appropriato per il tasto di scelta rapida in base alla piattaforma (`"Control"` o `"Command"`).

Creazione di menu

Tkinter visualizza i menu al di sotto della barra del titolo su Linux e Windows, secondo lo stile tradizionale. Ma su OS X, Tkinter integra i menu nel singolo menu di OS X nella parte superiore dello schermo. Tuttavia, come vedremo, dobbiamo aiutare Tkinter a gestire tale integrazione.

Menu e sottomenu sono istanze di `tkinter.Menu`. Un solo menu deve essere creato come barra dei menu della finestra di primo livello (per esempio la barra dei menu della finestra principale), e tutti gli altri come suoi figli.

```
def create_menubar(self):
    self.menubar = tk.Menu(self.master)
    self.master.config(menu=self.menubar)
    self.create_file_menu()
    self.create_help_menu()
```

Qui creiamo un nuovo menu vuoto come figlio della finestra e impostiamo l'attributo `menu` della finestra (cioè la barra dei menu) a questo menu (`self.menubar`). Poi aggiungiamo sottomenu alla barra dei menu; in qeusto caso soltanto due, che esamineremo entrambi nel seguito.

Creazione di un menu per i file

La maggior parte delle applicazioni a finestra principale ha un menu per i file, con opzioni per creare un nuovo documento, aprire un documento esistente, salvare il documento corrente e uscire dall'applicazione. Tuttavia, per i giochi molte di queste opzioni non servono, perciò l'applicazione Gravitate ne prevede soltanto due, come si vede nella Figura 7.8.

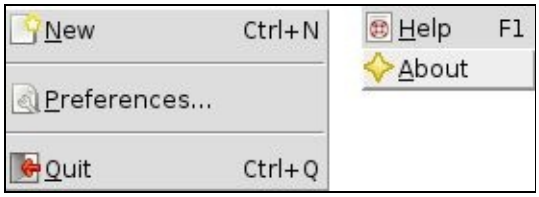


Figura 7.8 I menu dell'applicazione Gravitate su Linux.

```
def create_file_menu(self):
    modifier = TkUtil.menu_modifier()
    fileMenu = tk.Menu(self.menubar, name="apple")
    fileMenu.add_command(label=NEW, underline=0,
            command=self.board.new_game, compound=tk.LEFT,
            image=self.images[NEW], accelerator=modifier + "+N")
    if TkUtil.mac():
        self.master.createcommand("exit", self.close)
        self.master.createcommand("::tk::mac::ShowPreferences",
                self.preferences)
    else:
        fileMenu.add_separator()
        fileMenu.add_command(label=PREFERENCES + ELLIPSIS, underline=0,
        command=self.preferences, image=self.images[PREFERENCES],
        compound=tk.LEFT)
    fileMenu.add_separator()
    fileMenu.add_command(label="Quit", underline=0,
            command=self.close, compound=tk.LEFT,
```

```
                image=self.images[CLOSE],
                accelerator=modifier + "+Q")
        self.menubar.add_cascade(label="File", underline=0,
                menu=fileMenu)
```

Questo metodo viene utilizzato per creare i widget. Le costanti sono scritte interamente in maiuscolo, e a meno che non sia indicato altrimenti, contengono stringhe con lo stesso nome; per esempio `NEW` è una costante per la stringa `"New"`.

Il metodo inizia ottenendo il modificatore da utilizzare per i tasti di scelta rapida (Cmd su OS X, Ctrl su Linux e Windows), poi crea il menu dei file come figlio della barra dei menu della finestra. Il nome assegnato a questo menu (`"apple"`) indica a Tkinter che su OS X questo menu dovrà essere integrato con il menu dell'applicazione, mentre sarà ignorato su altre piattaforme.

Le opzioni di menu vengono aggiunte utilizzando i metodi `tkinter.Menu.add_command()`, `tkinter.Menu.add_checkbutton()` e `tkinter.Menu.add_radiobutton()`, anche se per Gravitate utilizziamo in realtà soltanto il primo. Per aggiungere i separatori si utilizza `tkinter.add_separator()`. L'attributo `underline` è ignorato su OS X, e su Windows le sottolineature sono visibili soltanto se impostate come tali, o se l'utente tiene premuto il tasto Alt. Per ciascuna opzione di menu specifichiamo il testo di etichetta, il carattere da sottolineare, il comando da eseguire quando viene richiamata l'opzione di menu e l'icona del menu (attributo `image`). L'attributo `compound` indica come gestire icone e testo: `tk.LEFT` mostra entrambi, con l'icona a sinistra. Abbiamo anche impostato un tasto di scelta rapida; per esempio, per attivare ll'opzione *New* del menu *File* lutente può premere Ctrl+N su Linux e Windows, oppure .N su OS X.

Su OS X le opzioni di menu *Preferences* e *Quit* dell'applicazione corrente sono visualizzate nel menu dell'applicazione (che si trova alla destra del menu mela, prima del menu file dell'applicazione). Per l'integrazione con OS X utilizziamo il metodo `tkinter.Tk.create-command()` per associare `::tk::mac::ShowPreferences` ed `exit` ai metodi corrispondenti di Gravitate. Per altre piattaforme aggiungiamo *Preferences* e *Quit* come normali opzioni di menu.

Una volta che il menu dei file è stato interamente riempito, lo aggiungiamo come menu 'a cascata' (cioè come sottomenu) della barra dei menu.

```
def menu_modifier():
    return "Command" if mac() else "Ctrl"
```

Questa funzione di `TkUtil/__init__.py` è utilizzata per il testo nei menu. La parola `"Command"` è gestita in modo particolare su OS X e appare come il simbolo punto (.).

Creazione di un menu per la Guida

Il menu della guida dell'applicazione ha soltanto due opzioni: *Help* e *About*. Tuttavia, nel caso di OS X entrambe queste opzioni sono gestite in modo diverso da Linux e Windows, perciò il codice deve tenere conto delle differenze.

```
def create_help_menu(self):
    helpMenu = tk.Menu(self.menubar, name="help")
    if TkUtil.mac():
        self.master.createcommand("tkAboutDialog", self.about)
        self.master.createcommand("::tk::mac::ShowHelp", self.help)
    else:
        helpMenu.add_command(label=HELP, underline=0,
                command=self.help, image=self.images[HELP],
                compound=tk.LEFT, accelerator="F1")
        helpMenu.add_command(label=ABOUT, underline=0,
                command=self.about, image=self.images[ABOUT],
                compound=tk.LEFT)
    self.menubar.add_cascade(label=HELP, underline=0,
            menu=helpMenu)
```

Iniziamo creando il menu della Guida con il nome `"help"`. Tale nome è ignorato su Linux e su Windows ma garantisce che, su OS X, il menu si integri in maniera corretta con la guida di sistema. Se lavoriamo su un sistema OS X, utilizziamo il metodo `tkinter.Tk.createcommand()` per associare ill Tcl/Tk `tkAboutDialog` e i comandi `::tk::mac::ShowHelp` con i metodi appropriati di Gravitate. Su altre piattaforme creiamo le opzioni di menu per la guida e le informazioni sul programma in maniera normale.

Creazione di una barra di stato con indicatori

L'applicazione Gravitate è dotata di una tipica barra di stato che visualizza messaggi di testo transitori sulla sinistra e un indicatore di stato permanente sulla destra. La Figura 7.7 mostra l'indicatore di stato e (nella schermata a sinistra) un messaggio transitorio.

```
def create_statusbar(self):
    statusBar = ttk.Frame(self.master)
    statusLabel = ttk.Label(statusBar, textvariable=self.statusText)
    statusLabel.grid(column=0, row=0, sticky=(tk.W, tk.E))
    scoreLabel = ttk.Label(statusBar, textvariable=self.scoreText,
            relief=tk.SUNKEN)
    scoreLabel.grid(column=1, row=0)
    statusBar.columnconfigure(0, weight=1)
    statusBar.pack(side=tk.BOTTOM, fill=tk.X)
    self.set_status_text("Click a tile or click File→New for a new game")
```

Per creare una barra di stato, iniziamo creando un frame. Poi aggiungiamo un'etichetta, che associamo con il `self.statusText` (di tipo `StringVar`). Ora possiamo impostare il testo di stato mediante `self.statusText`, anche se nella pratica utilizzeremo un metodo. Aggiungiamo inoltre un indicatore di stato permanente, un'etichetta che visualizza il punteggio (e il punteggio massimo) e che è associata a `self.scoreText`.

Disponiamo le due etichette a griglia nel frame della barra di stato e facciamo in modo che `statusLabel` (per i messaggi transitori) occupi tutto lo spazio disponibile in larghezza. Disponiamo il frame della barra di stato nella parte inferiore della finestra principale e lo allunghiamo in orizzontale fino a occupare l'intera ampiezza della finestra. Al termine, impostiamo un messaggio transitorio iniziale utilizzando un metodo personalizzato `set_status_text()`.

```
def set_status_text(self, text):
    self.statusText.set(text)
    self.master.after(SHOW_TIME, lambda: self.statusText.set(""))
```

Questo metodo imposta il testo di `self.statusText` al testo desiderato (che potrebbe essere vuoto) e poi cancella il testo dopo `SHOW_TIME` millisecondi (5 secondi in questo esempio).

Anche se avremmo potuto inserire soltanto delle etichette nella barra di stato, non vi è motivo per non poter aggiungere altri widget come caselle combinate, di selezione o pulsanti, per esempio.

In questo capitolo siamo riusciti a presentare soltanto alcuni impieghi di base di Tkinter. Da quando Python ha adottato Tcl/Tk 8.5 (la prima versione a usare i temi), Tkinter è diventato molto più interessante, poiché ora supporta il look and feel nativo su OS X e su Windows. Tkinter contiene alcuni widget molto potenti e flessibili, in particolare `tkinter.Text`, che può essere utilizzato per impostare e presentare testo elegantemente formattato, e `tkinter.Canvas` per la grafica 2D (l'abbiamo utilizzato per Gravitate e Gravitate 2). Altri tre widget molto utili sono `tkinter.ttk.Treeview` per visualizzare tabelle o alberi di elementi, `tkinter.ttk.Notebook` per mostrare schede (usato nella finestra delle preferenze di Gravitate 2) e `tkinter.ttk.Panedwindow` per suddivisioni di finestre.

Anche se Tkinter non fornisce alcune delle funzionalità di alto livello che altri toolkit GUI offrono, come abbiamo visto consente di creare facilmente barre di stato con messaggi transitori e indicatori di stato permanenti. I menu di Tkinter sono più sofisticati di quanto ci sia servito in questo capitolo, supportano sottomenu, sotto-sottomenu e così via, oltre a opzioni di menu con caselle attivabili (nello stile di caselle di controllo e pulsanti di opzione). Inoltre è piuttosto semplice creare menu contestuali.

Se pensiamo alle caratteristiche moderne che potremmo apprezzare, ci vengono subito in mente le barre degli strumenti. Sono facili da creare, ma occorre un po' di attenzione per fare in modo che possano essere nascoste e visualizzate, e si dispongano automaticamente in modo da tenere conto delle dimensioni delle finestre.

Un’altra funzionalità moderna di cui molte applicazioni potrebbero giovarsi è quella delle finestre agganciabili. È possibile creare finestre agganciabili che possano essere nascoste o mostrate, trascinate da un’area di aggancio a un’altra e perfino spostate liberamente sullo schermo.

Tra gli esempi di questo libro sono incluse due applicazioni non trattate in questo capitolo per mancanza di spazio: `texteditor` e `texteditor2`; la seconda è mostrata nella Figura 7.9. Entrambe queste applicazioni illustrano come implementare barre degli strumenti che possono essere visualizzate e nascoste e sono in grado di disporsi automaticamente, sottomenu, opzioni di menu attivabili o disattivabili, menu contestuali e un elenco dei file recenti. Entrambe dispongono anche di una finestra di dialogo estesa, visibile nella Figura 7.10. Inoltre, mostrano come utilizzare il widget `tkinter.Text` e come interagire con l’area degli Appunti. In più, `texteditor2` illustra come implementare finestre agganciabili (anche se quelle spostabili liberamente sullo schermo non funzionano correttamente su OS X).

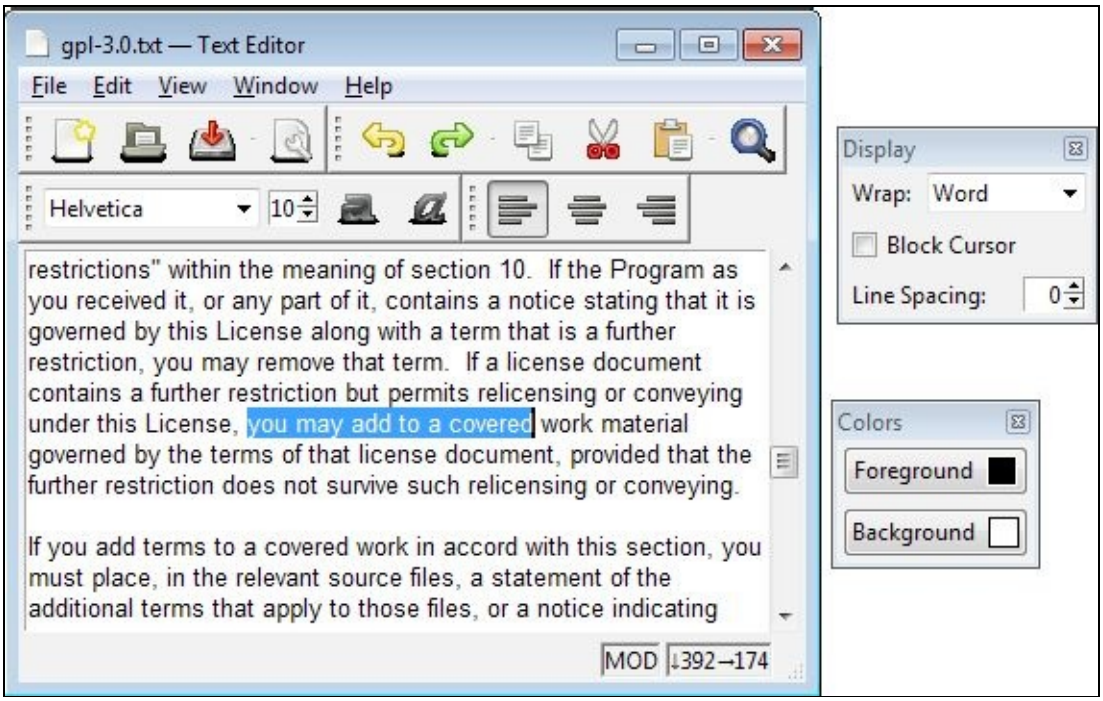


Figura 7.9 L’applicazione Text Editor 2 su Windows.

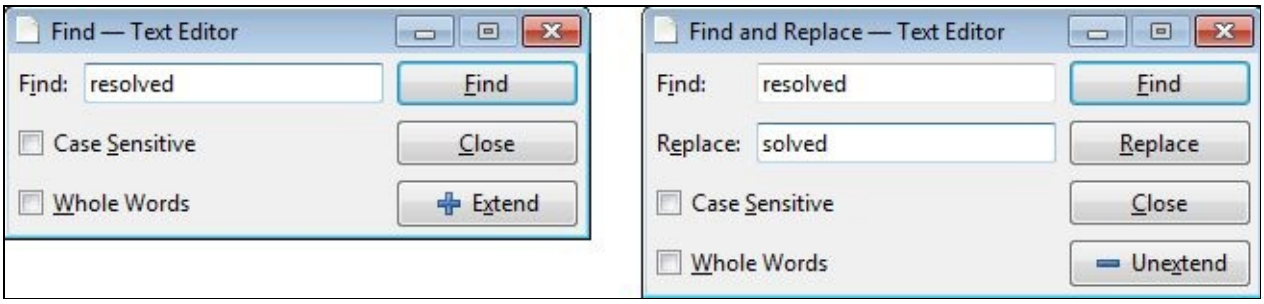


Figura 7.10 La finestra di dialogo estesa dell’applicazione Text Editor su Windows.

Tkinter richiede più lavoro rispetto ad altri toolkit GUI, tuttavia non impone molte limitazioni, e se creiamo la necessaria infrastruttura (per esempio per barre degli strumenti e finestre agganciabili), con sufficiente attenzione, possiamo riutilizzarla in tutte le nostre applicazioni. Tkinter è molto stabile e fornito direttamente con Python, perciò è l’ideale per creare applicazioni GUI facili da distribuire.

Grafica 3D OpenGL

Molte applicazioni moderne, come gli strumenti di progettazione, visualizzazione di dati, giochi e naturalmente gli screensaver, utilizzano grafica 3D. Tutti i toolkit GUI di Python citati nel capitolo precedente forniscono supporto per la grafica 3D, direttamente o tramite add-on. Il supporto 3D è quasi sempre fornito nella forma di un'interfaccia alle librerie OpenGL di sistema.

Esistono anche molti pacchetti di grafica 3D Python che forniscono interfacce di alto livello per semplificare la programmazione OpenGL, citiamo Python Computer Graphics Kit (`http://cgkit.sourceforge.net`), OpenCASCADE (`http://github.com/tenko/occmodel`) e VPython (`http://www.vpython.org`).

È anche possibile accedere a OpenGL in modo più diretto. I due pacchetti principali che forniscono questa funzionalità sono `PyOpenGL` (`http://pyopengl.sourceforge.net`) e `pyglet` (`http://www.pyglet.org`). Entrambi fanno da wrapper per le librerie OpenGL, il che rende molto semplice la traduzione di esempi in C (il linguaggio nativo di OpenGL, utilizzato anche nei libri su OpenGL) in Python. Entrambi i pacchetti sono utilizzabili per creare programmi 3D indipendenti, nel caso di `PyOpenGL` utilizzando il suo wrapper per la libreria GUI OpenGL `GLUT`, e nel caso di `pyglet` attraverso il suo stesso supporto per gestione di eventi e finestre.

Nel caso di programmi 3D indipendenti è probabilmente meglio utilizzare un toolkit GUI esistente insieme con `PyOpenGL` (che è in grado di interoperare con Tkinter, PyQt, PySide e wxPython, tra gli altri), oppure, ove sia sufficiente una GUI più semplice, `pyglet`.

Esistono molte versioni di OpenGL, e due modi abbastanza diversi di utilizzarlo. L'approccio tradizionale (o "modalità diretta") funziona sempre e per tutte le versioni: comporta la chiamata di funzioni OpenGL che sono eseguite immediatamente. Un approccio più moderno, disponibile a partire dalla versione 2.0, consiste nell'impostare la scena utilizzando l'approccio tradizionale e poi scrivere programmi OpenGL nel linguaggio OpenGL Shading Language (una particolare versione del linguaggio C). Tali programmi vengono poi inviati (come testo normale) alla GPU, che li compila e li esegue. Questo approccio può produrre programmi

molto più rapidi ed è molto più versatile rispetto all'approccio tradizionale, ma non è ancora ampiamente supportato.

In questo capitolo esamineremo un programma `PyOpenGL` e due programmi `pyglet`, che illustrano molti aspetti fondamentali della programmazione 3D OpenGL. Utilizzeremo l'approccio tradizionale, perché è molto più facile comprendere come creare grafica 3D attraverso chiamate di funzioni che dover apprendere il linguaggio OpenGL Shading Language, e in ogni caso, questo libro si occupa principalmente di programmazione in Python. La lettura di questo capitolo presuppone la conoscenza della programmazione OpenGL, perciò la maggior parte delle chiamate qui riportate non è spiegata nei dettagli. I lettori che non hanno familiarità con OpenGL potrebbero trovare utile come base di partenza consultare la *OpenGL SuperBible* citata nella bibliografia.

Un punto importante da evidenziare è una convenzione di nomenclatura di OpenGL. Molti nomi di funzioni terminano con un numero seguito da una o più lettere. Il numero indica il numero di argomenti e le lettere ne indicano il tipo. Per esempio, la funzione `glColor3f()` è utilizzata per impostare il colore corrente usando tre argomenti a virgola mobile - rosso, verde e blu, ciascuno nell'intervallo da 0.0 a 1.0 - mentre la funzione `glColor4ub()` è usata per impostare il colore usando quattro argomenti di tipo byte senza segno - rosso, verde, blu, alfa (trasparenza), ciascuno nell'intervallo da 0 a 255. Naturalmente in Python possiamo utilizzare normalmente numeri di qualsiasi tipo e lasciare che le conversioni necessarie siano effettuate automaticamente.

Le scene tridimensionali sono solitamente proiettate su superfici bidimensionali (per esempio lo schermo del computer) in due modi: ortograficamente o con prospettiva. La proiezione ortografica preserva le dimensioni degli oggetti ed è solitamente preferita per gli strumenti CAD. Le proiezioni prospettiche mostrano oggetti più grandi da vicino e più piccoli in lontananza; questo può produrre effetti più realistici, particolarmente nel caso dei paesaggi. Entrambi i tipi di proiezioni sono utilizzati nei giochi. Nella prima parte di questo capitolo creeremo una scena che fa uso della prospettiva, mentre nella seconda parte creeremo una scena che utilizza una proiezione ortografica.

Una scena con prospettiva

In questo paragrafo creeremo i programmi per il disegno di cilindri mostrati nella Figura 8.1. Entrambi i programmi mostrano tre assi colorati e un cilindro vuoto illuminato. La versione `PyOpenGL` (mostrata a sinistra nella figura) è quella più pura in termini di aderenza alle interfacce OpenGL, mentre la versione `pyglet` (mostrata a destra nella figura) è leggermente più facile da programmare e un poco più efficiente.

Gran parte del codice è identica in entrambi i programmi, e alcuni dei metodi differiscono soltanto per i loro nomi. Tenendo conto di ciò, esamineremo prima la versione `PyOpenGL` in forma completa, e poi ci limiteremo a trattare le differenze della versione `pyglet`. Nel prosieguo del capitolo vedremo poi molto altro codice `pyglet`.

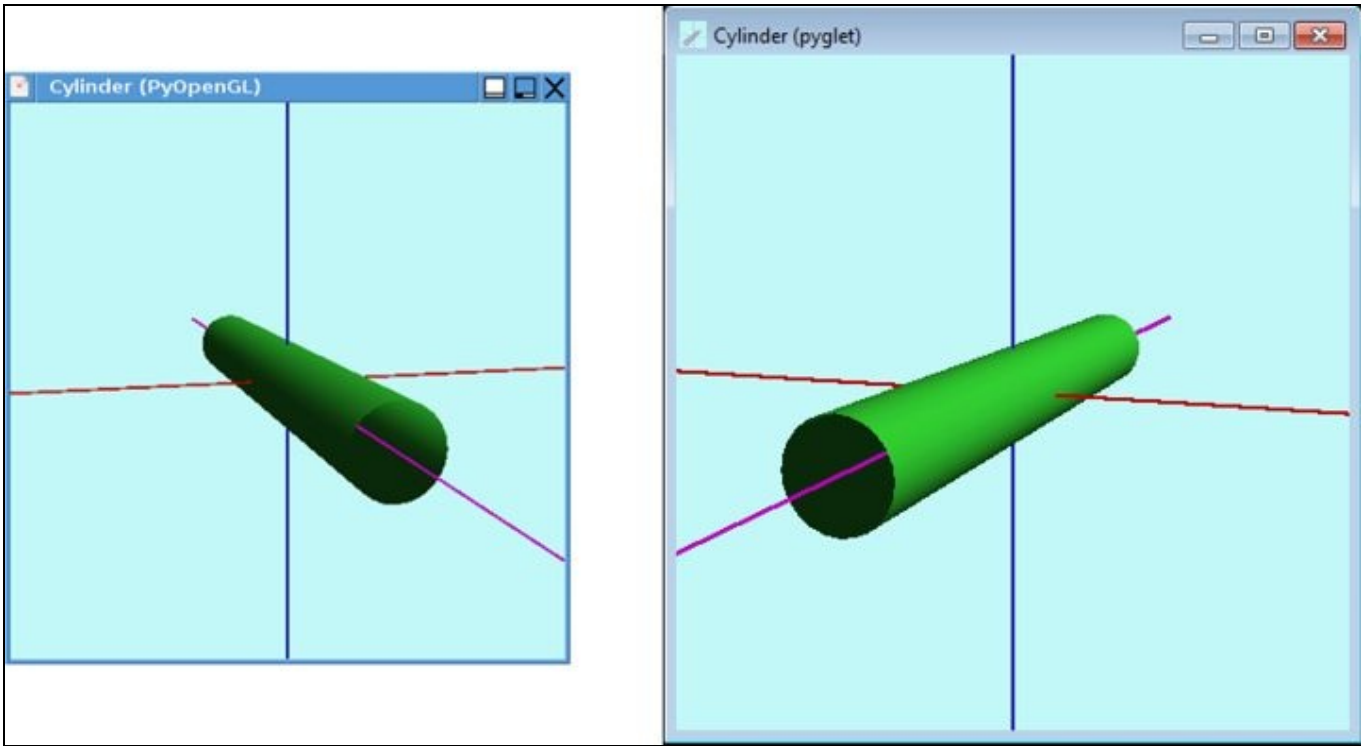


Figura 8.1 I programmi per il disegno di cilindri su Linux e Windows.

Creare un cilindro con PyOpenGL

Il programma `cylinder1.pyw` crea una semplice scena che l'utente può ruotare in modo indipendente rispetto agli assi x e y. E quando la finestra contenente la scena viene ridimensionata, la scena viene scalata di conseguenza.

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
```

Il programma utilizza la libreria OpenGL `GL` (libreria core), `GLU` (libreria di utilità) e `GLUT` (toolkit per le finestre). Normalmente è meglio evitare di effettuare le importazioni con la sintassi `from module import *`, ma nel caso di `PyOpenGL` appare ragionevole, perché tutti i nomi importati iniziano con un prefisso `gl`, `glu`, `glut` o `GL`, quindi possono essere facilmente individuati e difficilmente causeranno conflitti.

```
SIZE = 400
ANGLE_INCREMENT = 5

def main():
    glutInit(sys.argv)
    glutInitWindowSize(SIZE, SIZE)
    window = glutCreateWindow(b"Cylinder (PyOpenGL)")
    glutInitDisplayString(b"double=1 rgb=1 samples=4 depth=16")
    scene = Scene(window)
    glutDisplayFunc(scene.display)
    glutReshapeFunc(scene.reshape)
    glutKeyboardFunc(scene.keyboard)
    glutSpecialFunc(scene.special) glutMainLoop()
```

La libreria `GLUT` fornisce la gestione di eventi e le finestre di primo livello che normalmente sono fornite da un toolkit GUI. Per utilizzare questa libreria dobbiamo richiamare inizialmente `glutInit()` e passare gli argomenti della riga di comando del programma; verranno applicati e rimossi tutti quelli riconosciuti. Poi, opzionalmente, possiamo impostare una dimensione iniziale della finestra (come facciamo in questo caso). Poi creiamo una finestra e le forniamo un titolo iniziale. La chiamata di `glutInitDisplayString()` è utilizzata per impostare alcuni dei parametri di contesto OpenGL, in questo caso per attivare il doppio buffering, per utilizzare il modello di colore RGBA (*Red, Green, Blue, Alpha*), per attivare il supporto dell'antialiasing e per impostare un buffer con 16 bit di precisione (cfr. la documentazione di `PyOpenGL` per un elenco di tutte le opzioni con i relativi significati).

Le interfacce OpenGL utilizzano stringhe a 8 bit (normalmente in codifica ASCII). Un modo per passare tali stringhe è quello di utilizzare il metodo `str.encode()`, che restituisce un `bytes` codificato con la codifica specificata, per esempio `"title".encode("ascii")`, che restituisce `b'title'`. Tuttavia, nel nostro caso abbiamo usato direttamente i letterali `bytes`.

La scena è una classe `Scene` personalizzata che utilizzeremo per il rendering della grafica OpenGL nella finestra. Una volta creata la scena, registriamo alcuni dei suoi metodi come funzioni callback `GLUT`, cioè come funzioni che OpenGL richiamerà in risposta a particolari eventi. Registriamo il metodo `Scene.display()`, che sarà richiamato a ogni visualizzazione della finestra (cioè la prima volta e ogni volta che viene resa visibile, se era coperta). Registriamo anche il metodo `Scene.reshape()`, che è richiamato ogni volta che la finestra viene ridimensionata; il metodo `Scene.keyboard()` che è

richiamato quando l'utente preme un tasto (esclusi certi tasti) e il metodo `Scene.special()`, che è richiamato quando l'utente preme un tasto non gestito dalla funzione di tastiera registrata.

Dopo aver creato la finestra e registrato le funzioni callback, iniziamo il ciclo di eventi `GLUT`, che sarà eseguito fino al termine del programma.

```
class Scene:

    def __init__(self, window):
        self.window = window
        self.xAngle = 0
        self.yAngle = 0
        self._initialize_gl()
```

Iniziamo la classe `Scene` mantenendo un riferimento alla finestra OpenGL e impostando a zero gli angoli degli assi x e y. Rimandiamo tutte le operazioni di inizializzazione specifiche per OpenGL a una funzione separata che richiameremo alla fine.

```
    def _initialize_gl(self):
        glClearColor(195/255, 248/255, 248/255, 1)
        glEnable(GL_DEPTH_TEST) glEnable(GL_POINT_SMOOTH)
        glHint(GL_POINT_SMOOTH_HINT, GL_NICEST)
        glEnable(GL_LINE_SMOOTH)
        glHint(GL_LINE_SMOOTH_HINT,GL_NICEST)
        glEnable(GL_COLOR_MATERIAL)
        glEnable(GL_LIGHTING)
        glEnable(GL_LIGHT0)
        glLightfv(GL_LIGHT0, GL_POSITION, vector(0.5, 0.5, 1, 0))
        glLightfv(GL_LIGHT0, GL_SPECULAR, vector(0.5, 0.5, 1, 1))
        glLightfv(GL_LIGHT0, GL_DIFFUSE, vector(1, 1, 1, 1))
        glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 50)
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, vector(1, 1, 1, 1))
        glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)
```

Questo metodo è richiamato soltanto una volta per impostare il contesto OpenGL. Iniziamo impostando il colore "clear" (quello di sfondo) a una tonalità blu chiaro. Poi abilitiamo varie caratteristiche di OpenGL, di cui la più importante è la creazione di una luce. Si deve alla presenza di questa luce il fatto che il cilindro non appare di colore uniforme. Inoltre facciamo in modo che il colore di base del cilindro (non illuminato) si basi su chiamate di funzioni `glColor…()`; per esempio, avendo abilitato l'opzione `GL_COLOR_MATERIAL`, l'impostazione del colore corrente al rosso con `glColor3ub(255, 0, 0)` avrà effetto anche sul colore del materiale (in questo caso sul colore del cilindro).

```
def vector(*args):
    return (GLfloat * len(args))(*args)
```

Questa funzione ausiliaria è usata per creare un array OpenGL di valori a virgola mobile (ognuno di tipo `GLfloat`).

```
    def display(self):
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
```

```
        glMatrixMode(GL_MODELVIEW)
        glPushMatrix() glTranslatef(0, 0, -600)
        glRotatef(self.xAngle, 1, 0, 0)
        glRotatef(self.yAngle, 0, 1, 0)
        self._draw_axes()
        self._draw_cylinder()
        glPopMatrix()
```

Questo metodo è richiamato quando la finestra della scena viene visualizzata per la prima volta e ogni volta che la scena viene resa nuovamente visibile (per esempio dopo che è stata spostata o chiusa una finestra che la copriva). Sposta la scena all'indietro (lungo l'asse z) in modo che la vediamo di fronte, e la ruota sugli assi x e y in base all'interazione dell'utente (inizialmente sono rotazioni di zero gradi). Una volta che la scena è stata traslata e ruotata, disegniamo gli assi e poi il cilindro stesso.

```
def _draw_axes(self):
glBegin(GL_LINES)
glColor3f(1, 0, 0)         # asse x
glVertex3f(-1000, 0, 0)
glVertex3f(1000, 0, 0)
glColor3f(0, 0, 1)         # asse y
glVertex3f(0, -1000, 0)
glVertex3f(0, 1000, 0)
glColor3f(1, 0, 1)         # asse z
glVertex3f(0, 0, -1000)
glVertex3f(0, 0, 1000)
glEnd()
```

Un *vertice* in OpenGL è un punto nello spazio tridimensionale. Ogni asse è disegnato nello stesso modo: impostiamo il suo colore e poi specifichiamo il vertice iniziale e quello finale. Le funzioni `glColor3f()` e `glVertex3f()` richiedono ciascuna tre argomenti a virgola mobile, ma noi abbiamo usato degli `int` lasciando che Python si occupasse delle conversioni necessarie.

```
    def _draw_cylinder(self):
        glPushMatrix()
        try:
            glTranslatef(0, 0, -200)
            cylinder = gluNewQuadric()
            gluQuadricNormals(cylinder, GLU_SMOOTH)
            glColor3ub(48, 200, 48)
            gluCylinder(cylinder, 25, 25, 400, 24, 24)
        finally:
            gluDeleteQuadric(cylinder)
        glPopMatrix()
```

La libreria di utilità `GLU` offre il supporto integrato per la creazione di alcune forme 3D di base, tra cui i cilindri. Iniziamo spostando il nostro punto di partenza un po' più indietro lungo l'asse z, poi creiamo una *quadrica*, cioè un oggetto che può essere utilizzato per il rendering di varie forme 3D. Impostiamo il colore usando tre byte senza segno (valori di rosso, verde e blu nell'intervallo da 0 a 255). La chiamata di `gluCylinder()` accetta la quadrica generica, il raggio del cilindro a ciascuna estremità (in questo caso sono uguali), l'altezza del cilindro e due fattori di granularità (valori più elevati producono risultati migliori che richiedono più tempo di elaborazione). E alla

fine cancelliamo esplicitamente la quadrica, anziché affidarci alla garbage collection di Python per ridurre al minimo l'uso di risorse.

```
    def reshape(self, width, height):
        width = width if width else 1
        height = height if height else 1
        aspectRatio = width / height
        glViewport(0, 0, width, height)
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(35.0, aspectRatio, 1.0, 1000.0)
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
```

Questo metodo è richiamato ogni volta che la finestra della scena viene ridimensionata. Quasi tutto il lavoro è affidato alla funzione `gluPerspective()`. E in effetti il codice riportato qui dovrebbe servire come punto di partenza per qualsiasi scena che utilizzi una proiezione prospettica.

```
    def keyboard(self, key, x, y):
        if key == b"\x1B":  # Escape
            glutDestroyWindow(self.window)
```

Se l'utente preme un tasto (che non sia un tasto funzione, un tasto freccia, PagSu, PagGiù, Home, Fine o Ins), viene richiamato questo metodo (registrato con `glutKeyboardFunc()`). Qui controlliamo se è stato premuto il tasto Esc, e in questo caso eliminiamo la finestra, e poiché non vi sono altre finestre, così termina il programma.

```
    def special(self, key, x, y):
        if key == GLUT_KEY_UP:
            self.xAngle -= ANGLE_INCREMENT
        elif key == GLUT_KEY_DOWN:
            self.xAngle += ANGLE_INCREMENT
        elif key == GLUT_KEY_LEFT:
            self.yAngle -= ANGLE_INCREMENT
        elif key == GLUT_KEY_RIGHT:
            self.yAngle += ANGLE_INCREMENT
        glutPostRedisplay()
```

Questo metodo è stato registrato con la funzione `glutSpecialFunc()` ed è richiamato ogni volta che l'utente preme un tasto funzione, un tasto freccia, PagSu, PagGiù, Home, Fine o Ins. Qui vediamo solo come rispondere ai tasti freccia. Se viene premuto un tasto freccia, incrementiamo o decrementiamo l'angolo dell'asse x o dell'asse y e indichiamo al toolkit `GLUT` di ridisegnare la finestra. Il risultato è la chiamata del callable registrato con `glutDisplayFunc()`, in questo esempio il metodo `Scene.display()`.

Abbiamo visto il codice completo per il programma `PyOpenGL cylinder1.pyw`. Chi ha familiarità con OpenGL o con il linguaggio C dovrebbe trovarsi subito a proprio agio, poiché le chiamate OpenGL sono quasi tutte simili.

Creare un cilindro con pyglet

Strutturalmente, la versione `pyglet` (`cylinder2.pyw`) è molto simile alla versione `PyOpenGL`. La differenza fondamentale è che `pyglet` provvede da sé alla gestione di eventi e alla creazione di finestre, perciò non abbiamo la necessità di utilizzare chiamate `GLUT`.

```
def main():
    caption = "Cylinder (pyglet)"
    width = height = SIZE
    resizable = True
    try:
        config = Config(sample_buffers=1, samples=4, depth_size=16,
                double_buffer=True)
        window = Window(width, height, caption=caption, config=config,
                resizable=resizable)
    except pyglet.window.NoSuchConfigException:
        window = Window(width, height, caption=caption,
                resizable=resizable)
    path = os.path.realpath(os.path.dirname(__file__))
    icon16 = pyglet.image.load(os.path.join(path, "cylinder_16x16.png"))
    icon32 = pyglet.image.load(os.path.join(path, "cylinder_32x32.png"))
    window.set_icon(icon16, icon32)
    pyglet.app.run()
```

Anziché passare la configurazione del contesto OpenGL come stringa di `bytes`, `pyglet` supporta l'uso di un oggetto `pyglet.gl.Config` per specificare i nostri requisiti. Qui iniziamo creando la nostra configurazione preferita e poi la nostra `Window` personalizzata (una sottoclasse di `pyglet.window.Window`) basata su tale configurazione; in caso di fallimento, ripieghiamo sulla creazione della finestra con una configurazione di default.

Una bella caratteristica di `pyglet` è che supporta l'impostazione dell'icona dell'applicazione, che appare generalmente in un angolo della barra del titolo e nelle icone per il passaggio da un'applicazione all'altra. Una volta che la finestra è stata creata e le icone configurate, avviamo il ciclo di eventi `pyglet`.

```
class Window(pyglet.window.Window):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.set_minimum_size(200, 200)
        self.xAngle = 0 self.yAngle = 0
        self._initialize_gl()
        self._z_axis_list = pyglet.graphics.vertex_list(2,
                ("v3i", (0, 0, -1000, 0, 0, 1000)),
                ("c3B", (255, 0, 255) * 2))  # un solo colore per vertice
```

Questo metodo è simile all'equivalente metodo `Scene` che abbiamo esaminato precedentemente. Una differenza è che in questo caso abbiamo impostato una dimensione minima per la finestra. Come vedremo tra breve, `pyglet` è in grado di disegnare linee in tre modi diversi; il terzo prevede di disegnare un elenco

preesistente di coppie vertice-colore, e in questo caso utilizziamo una lista di questo tipo. La funzione che crea la lista accetta il numero di coppie vertice-colore seguito da una sequenza di coppie in tale numero. Ogni coppia è costituita da un formato stringa e una sequenza. In questo caso il formato stringa della prima coppia significa “vertici specificati da tre coordinate intere”, perciò in questo caso sono dati due vertici. La seconda coppia indica “colori specificati da tre byte senza segno”, e qui sono forniti due colori (identici), uno per ciascun vertice.

Non mostriamo i metodi `_initialize_gl()`, `on_draw()`, `on_resize()` e `_draw_cylinder()`. Il metodo `_initialize_gl()` è molto simile a quello usato in `cylinder1.pyw`. Inoltre, il corpo del metodo `on_draw()` richiamato automaticamente da `pyglet` per visualizzare sottoclassi `pyglet.window.Window` è identico al corpo del metodo `Scene.display()` del programma `cylinder1.pyw`. Similmente, il metodo `on_resize()` richiamato per gestire il ridimensionamento ha lo stesso corpo del metodo `Scene.reshape()` del precedente programma. I metodi `_draw_cylinder()` di entrambi i programmi (`Scene._draw_cylinder()` e `Window._draw_cylinder()`) sono identici.

```
    def _draw_axes(self):
        glBegin(GL_LINES)                       # asse x (stile tradizionale)
        glColor3f(1, 0, 0)
        glVertex3f(-1000, 0, 0)
        glVertex3f(1000, 0, 0)
        glEnd()
        pyglet.graphics.draw(2, GL_LINES,       # asse y (stile pyglet "live")
                ("v3i", (0, -1000, 0, 0, 1000, 0)),
                ("c3B", (0, 0, 255) * 2))
        self._z_axis_list.draw(GL_LINES)        # asse z (stile pyglet efficiente)
```

Abbiamo disegnato ciascun asse usando una tecnica diversa per illustrare alcune delle opzioni disponibili. L’asse x è tracciato usando chiamate di funzioni OpenGL tradizionali esattamente nello stesso modo per la versione `PyOpenGL` del programma. L’asse y è tracciato indicando a `pyglet` di disegnare linee comprese tra due punti (potrebbe essere un numero qualsiasi di punti, naturalmente) per cui forniamo i corrispondenti vertici e colori. Soprattutto quando si hanno numerose linee, questo approccio dovrebbe essere un po’ più efficiente rispetto a quello tradizionale. L’asse z è tracciato nel modo più efficiente possibile: prendiamo una lista preesistente di coppie vertice-colore memorizzate come `pyglet.graphics.vertex_list` e le indichiamo di disegnare se stessa come un insieme di linee tra vertici.

```
    def on_text_motion(self, motion):  # Ruota rispetto all'asse x o y
        if motion == pyglet.window.key.MOTION_UP:
            self.xAngle -= ANGLE_INCREMENT
        elif motion == pyglet.window.key.MOTION_DOWN:
            self.xAngle += ANGLE_INCREMENT
        elif motion == pyglet.window.key.MOTION_LEFT:
            self.yAngle -= ANGLE_INCREMENT
        elif motion == pyglet.window.key.MOTION_RIGHT:
            self.yAngle += ANGLE_INCREMENT
```

Se l’utente preme un tasto freccia, viene richiamato questo metodo (purché sia definito). Qui svolgiamo lo stesso lavoro fatto nel metodo `special()` del precedente esempio, solo che utilizziamo costanti specifiche di `pyglet` anziché costanti `GLUT` per i tasti.

Non abbiamo fornito un metodo `on_key_press()` (che sarebbe richiamato per altri tasti premuti) perché l’implementazione di default di `pyglet` chiude la finestra (e quindi termina il programma) se si preme Esc, e questo è proprio il comportamento che desideriamo.

I due programmi di disegno di cilindri sono lunghi entrambi circa 140 righe. Tuttavia, se utilizziamo `pyglet.graphics.vertex_list` e altre estensioni di `pyglet`, guadagniamo sia in comodità, soprattutto per la gestione di eventi e finestre, sia in efficienza.

Un gioco in proiezione ortografica

Nel Capitolo 7 abbiamo mostrato il codice per un gioco 2D denominato Gravitate, omettendo però il codice per il disegno delle mattonelle. In effetti ogni piastrella era prodotta disegnando un quadrato circondato da quattro trapezoidi isosceli posizionati sopra, sotto, a sinistra e a destra. I trapezoidi sopra e a sinistra erano disegnati in una tonalità più chiara del colore del quadrato, quelli sotto e a destra in una tonalità più scura; tutto ciò consentiva di ottenere un aspetto tridimensionale (cfr. la Figura 7.7 e il riquadro dedicato al gioco Gravitate nel Capitolo 7).

In questo paragrafo esamineremo gran parte del codice per il gioco Gravitate 3D mostrato nella Figura 8.2. Questo programma utilizza al posto delle mattonelle delle sfere che vengono disposte con degli spazi tra di loro, in modo che l'utente possa vedere all'interno della struttura tridimensionale quando ruota la scena rispetto agli assi x e y. Ci concentreremo sul codice per la GUI e la grafica 3D, tralasciando alcuni dei dettagli di basso livello che implementano la logica del gioco. Il codice sorgente completo si trova in `gravitate3d.pyw`.

La funzione `main()` del programma (non mostrata qui) è quasi identica a quella di `cylinder2.pyw`, le uniche differenze sono il nome della didascalia e i nomi delle immagini utilizzate per le icone.

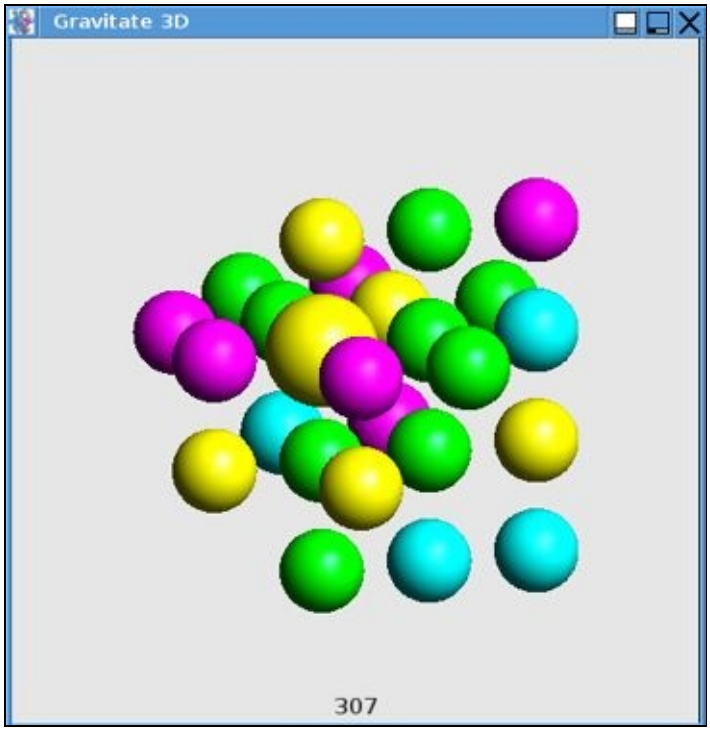


Figura 8.2 Il programma Gravitate 3D su Linux.

```
BOARD_SIZE = 4  # Deve essere > 1.
ANGLE_INCREMENT = 5
RADIUS_FACTOR = 10
DELAY = 0.5 # secondi
```

```
MIN_COLORS = 4
MAX_COLORS = min(len(COLORS), MIN_COLORS)
```

Queste sono alcune delle costanti utilizzate dal programma. `BOARD_SIZE` è il numero di sfere in ciascun asse; quando è impostato a 4, produce una griglia 4 × 4 × 4 di 64 sfere. `ANGLE_INCREMENT` è impostato a 5 per indicare che, quando l'utente preme un tasto freccia, la scena viene ruotata a passi di 5°. `DELAY` è il tempo di attesa tra la cancellazione della sfera (e di tutte le sfere adiacenti con lo stesso colore, e di tutte le sfere ad esse adiacenti con lo stesso colore) che l'utente ha selezionato e su cui ha fatto clic, e lo spostamento di sfere verso il centro per colmare ogni spazio. `COLORS` (non mostrata qui) è una lista di 3-tuple di interi (ognuno compreso nell'intervallo da 0 a 255), ognuna delle quali rappresenta un colore.

Quando l'utente fa clic su una sfera non selezionata, la seleziona (mentre quella precedentemente selezionata viene deselezionata), e la sfera viene evidenziata disegnandola con un raggio che è di un fattore `RADIUS_FACTOR` più grande di quello usato normalmente. Quando l'utente fa clic su una sfera selezionata, tale sfera, con tutte quelle dello stesso colore a essa adiacenti, e quelle adiacenti a queste ultime, e così via, vengono cancellate - purché vengano cancellate almeno due sfere. Altrimenti, la sfera viene semplicemente deselezionata.

```
class Window(pyglet.window.Window):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.set_minimum_size(200, 200)
        self.xAngle = 10
        self.yAngle = -15
        self.minColors = MIN_COLORS
        self.maxColors = MAX_COLORS
        self.delay = DELAY
        self.board_size = BOARD_SIZE
        self._initialize_gl()
        self.label = pyglet.text.Label("", bold=True, font_size=11,
                anchor_x="center")
        self._new_game()
```

Questo metodo `__init__()` contiene più istruzioni dei metodi equivalenti per il programma del cilindro, perché dobbiamo impostare i colori, il ritardo e la dimensione dell'area. Inoltre avviamo il programma con una certa rotazione iniziale, in modo che l'utente possa vedere subito che si tratta di un gioco tridimensionale.

Una funzionalità particolarmente utile offerta da `pyglet` è rappresentata dalle etichette di testo. Qui creiamo un'etichetta vuota centrata nella parte inferiore della scena, che utilizzeremo per visualizzare messaggi e il punteggio corrente.

La chiamata al metodo personalizzato `_initialize_gl()` (non mostrato qui, ma simile a quello che abbiamo visto precedentemente) imposta lo sfondo e la luce. Una volta

che è tutto a posto in termini di logica del programma e OpenGL, iniziamo una nuova partita.

```
def _new_game(self):
    self.score = 0
    self.gameOver = False
    self.selected = None
    self.selecting = False
    self.label.text = ("Click to Select • Click again to Delete • "
            "Arrows to Rotate")
    random.shuffle(COLORS)
    colors = COLORS[:self.maxColors]
    self.board = []
    for x in range(self.board_size):
        self.board.append([])
        for y in range(self.board_size):
            self.board[x].append([])
            for z in range(self.board_size):
                color = random.choice(colors)
                self.board[x][y].append(SceneObject(color))
```

Questo metodo crea un’area di gioco in cui ogni sfera è colorata con un colore casuale scelto dalla lista `COLORS` e dove sono in uso al più `self.maxColors` colori. L’area di gioco è rappresentata da una lista di liste di liste di oggetti `SceneObject`. Ognuno di questi oggetti ha un colore (il colore della sfera passato al costruttore) e un colore di selezione (generato automaticamente e utilizzato per la selezione, come vedremo più avanti).

Poiché abbiamo modificato il testo dell’etichetta, `pyglet` ridisegnerà la scena (richiamando il nostro metodo `on_draw()`)e la nuova partita sarà visibile e in attesa di interazione dell’utente.

Disegno dell’area di gioco

Quando una scena è visualizzata per la prima volta, o portata alla luce nel momento in cui una finestra che la copriva viene chiusa o spostata, `pyglet` richiama il metodo `on_draw()`. E quando una scena è ridimensionata (cioè quando viene ridimensionata la finestra), `pyglet` richiama il metodo `on_resize()`.

```
def on_resize(self, width, height):
    size = min(self.width, self.height) / 2
    height = height if height else 1
    width = width if width else 1
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    if width <= height:
        glOrtho(-size, size, -size * height / width,
                size * height / width, -size, size)
    else:
        glOrtho(-size * width / height,size * width / height,
                -size, size, -size, size)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
```

Per Gravitate 3D abbiamo utilizzato una proiezione ortografica. Il codice mostrato qui dovrebbe funzionare così com'è per qualsiasi scena ortografica (perciò, se utilizzassimo `PyOpenGL`, chiameremmo questo `reshape()` e lo registreremmo con la funzione `glutReshapeFunc()`).

```
def on_draw(self):
    diameter = min(self.width, self.height) / (self.board_size * 1.5)
    radius = diameter / 2
    offset = radius - ((diameter * self.board_size) / 2)
    radius = max(RADIUS_FACTOR, radius - RADIUS_FACTOR)
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_MODELVIEW)
    glPushMatrix()
    glRotatef(self.xAngle, 1, 0, 0)
    glRotatef(self.yAngle, 0, 1, 0)
    with Selecting(self.selecting):
            self._draw_spheres(offset, radius, diameter)
    glPopMatrix()
    if self.label.text:
        self.label.y = (-self.height // 2) + 10
        self.label.draw()
```

Questo è l'equivalente in `pyglet` di un metodo `PyOpenGL display()` registrato con la funzione `glutDisplayFunc()`. Vogliamo che l'area di gioco occupi il massimo spazio possibile nella finestra, mantenendo la possibilità di ruotarla senza causare il ritaglio di alcuna sfera. Abbiamo anche bisogno di calcolare un offset per assicurarci che l'area di gioco sia centrata correttamente.

Una volta esauriti i preliminari, ruotiamo la scena (per esempio se l'utente ha premuto un tasto freccia) e poi disegniamo le sfere nel contesto di un context manager `Selecting` personalizzato. Tale context manager garantisce che determinate impostazioni siano attivate o disattivate in base al fatto che la scena sia disegnata per essere vista dall'utente o sia disegnata in modo nascosto, allo scopo di individuare su quale sfera ha fatto clic l'utente (la selezione è discussa nel paragrafo seguente).

Se l'etichetta contiene del testo, ci assicuriamo che la sua posizione y corrisponda alla parte inferiore della finestra - poiché la finestra potrebbe essere stata ridimensionata - e poi le indichiamo di ridisegnarsi (cioè di disegnare il testo che contiene).

```
def _draw_spheres(self, offset, radius, diameter):
    try:
        sphere = gluNewQuadric()
        gluQuadricNormals(sphere, GLU_SMOOTH)
        for x, y, z in itertools.product(range(self.board_size),
                repeat=3):
            sceneObject = self.board[x][y][z]
            if self.selecting:
                color = sceneObject.selectColor
            else:
                color = sceneObject.color
            if color is not None:
                self._draw_sphere(sphere, x, y, z, offset, radius,
                        diameter, color)
        finally:
            gluDeleteQuadric(sphere)
```

Le quadriche possono essere usate per disegnare sfere allo stesso modo dei cilindri, anche se in questo caso dobbiamo disegnare molte sfere (fino a 64) anziché un cilindro soltanto. Possiamo comunque usare la stessa quadrica per disegnare ogni sfera.

Anziché scrivere `for x in range(self.board.size): for y in range(self.board. size): for z in range(self.board.size):` per produrre una tripletta x, y, z per ogni sfera nella lista di liste di liste, abbiamo ottenuto lo stesso risultato utilizzando un singolo ciclo `for` insieme alla funzione `itertools.product()`.

Per ciascuna tripletta recuperiamo l'oggetto corrispondente (i cui colori saranno `None` se l'oggetto è stato cancellato) e impostiamo il colore a quello di selezione, se stiamo disegnando per vedere su quale sfera è stato fatto clic, oppure il colore della sfera, se stiamo disegnando per visualizzare la sfera all'utente. Se il colore non è `None` disegniamo quella particolare sfera.

```
def _draw_sphere(self, sphere, x, y, z, offset, radius, diameter, color):
        if self.selected == (x, y, z):
            radius += RADIUS_FACTOR
        glPushMatrix()
        x = offset + (x * diameter)
        y = offset + (y * diameter)
        z = offset + (z * diameter)
        glTranslatef(x, y, z)
        glColor3ub(*color)
        gluSphere(sphere, radius, 24, 24)
        glPopMatrix()
```

Questo metodo è usato per disegnare ciascun offset di sfera nella posizione corretta all'interno della griglia di sfere tridimensionale. Se la sfera è selezionata, la disegniamo con un raggio maggiorato. Gli ultimi due argomenti per `gluSphere()` sono due fattori di granularità (per cui valori più elevati producono risultati migliori, che sono anche più costosi in termini di elaborazione).

Selezione di oggetti

Selezionare un oggetto in uno spazio a tre dimensioni che è visualizzato su una superficie a due dimensioni non è facile! Negli anni sono state sviluppate varie tecniche, ma quella che abbiamo usato in questo caso per il gioco Gravitate 3D sembra la più affidabile, ed è anche la più diffusa.

Questa tecnica funziona come segue. Quando l'utente fa clic sulla scena, questa viene ridisegnata in un buffer esterno allo schermo e non visibile all'utente, in cui ogni oggetto è disegnato con un colore univoco. Il colore del pixel che si trova nella posizione del clic viene poi letto dal buffer e usato per identificare l'oggetto univoco

associato con esso. Perché tutto ciò funzioni, dobbiamo disegnare la scena senza antialiasing, né illuminazione, né texture, in modo che ciascun oggetto sia disegnato nel proprio colore univoco senza che vi siano ulteriori elaborazioni di colore.

Iniziamo esaminando lo `SceneObject` da cui è rappresentata ogni sfera, poi passeremo a esaminare il context manager `Selecting`.

```
class SceneObject:

    __SelectColor = 0

    def __init__(self, color):
        self.color = color
        SceneObject.__SelectColor += 1
        self.selectColor = SceneObject.__SelectColor
```

Assegniamo a ciascun oggetto sulla scena il suo colore di visualizzazione (`self.color`), che non deve necessariamente essere univoco, e anche un colore di selezione univoco. La variabile statica privata `__SelectColor` è un intero che viene incrementato per ogni nuovo oggetto, ed è utilizzato per assegnare a ciascun oggetto un colore di selezione univoco.

```
    @property
    def selectColor(self):
        return self.__selectColor
```

Questa proprietà restituisce il colore di selezione dell'oggetto, che può essere `None` (per esempio nel caso di un oggetto cancellato) o una tupla di 3 elementi costituiti da numeri interi corrispondenti a componenti di colore (ognuno compreso nell'intervallo da 0 a 255).

```
    @selectColor.setter
    def selectColor(self, value):
        if value is None or isinstance(value, tuple):
            self.__selectColor = value
        else:
            parts = []
            for _ in range(3):
                value, y = divmod(value, 256)
                parts.append(y)
            self.__selectColor = tuple(parts)
```

Questo metodo per impostare il colore di selezione accetta il valore dato se è `None` o una `tuple`; altrimenti calcola una tupla di colore univoca in base al valore intero univoco fornito. Il primo oggetto riceve un valore 1, perciò il suo colore è `(1, 0, 0)`. Il secondo riceve il valore 2, perciò il suo colore è `(2, 0, 0)`, e così via fino al 255-esimo, il cui colore è `(255, 0, 0)`. Il 256-esimo colore è `(0, 1, 0)`, il 257-esimo è `(1, 1, 0)` e il 258-esimo è `(2,1, 0)`, e così via. Questo sistema è in grado di gestire oltre sedici milioni di oggetti unici, un numero che dovrebbe essere sufficiente per la maggior parte delle situazioni.

```
SELECTING_ENUMS = (GL_ALPHA_TEST, GL_DEPTH_TEST, GL_DITHER,
        GL_LIGHT0, GL_LIGHTING, GL_MULTISAMPLE, GL_TEXTURE_1D,
        GL_TEXTURE_2D, GL_TEXTURE_3D)
```

Dobbiamo attivare o disattivare antialiasing, illuminazione, texture e qualsiasi altra cosa che possa modificare il color e di un oggetto a seconda del fatto che stiamo disegnando l'oggetto in questione per visualizzarlo all'utente oppure fuori schermo, allo scopo di individuare su quale oggetto l'utente ha fatto clic. Questi sono gli enum OpenGL che hanno impatto sul colore dell'oggetto nel programma Gravitate 3D.

```
class Selecting:

        def __init__(self, selecting):
            self.selecting = selecting
```

Il context manager `Selecting` ricorda se le sfere che sono state disegnate nel suo contesto sono destinate alla visualizzazione oppure al rilevamento dell'oggetto su cui l'utente ha fatto clic, cioè alla selezione.

```
    def __enter__(self):
        if self.selecting:
            for enum in SELECTING_ENUMS:
                glDisable(enum)
            glShadeModel(GL_FLAT)
```

Quando entriamo nel context manager, se il disegno è effettuato a scopo di selezione, disabilitiamo tutti gli aspetti dello stato di OpenGL che possono modificare il colore e passiamo a un modello di *flat shading* (sfaccettato).

```
    def __exit__(self, exc_type, exc_value, traceback):
        if self.selecting:
            for enum in SELECTING_ENUMS:
                glEnable(enum)
            glShadeModel(GL_SMOOTH)
```

Quando usciamo dal context manager, nel caso in cui stiamo disegnando a scopo di selezione, riabilitiamo tutti gli aspetti dello stato di OpenGL che possono modificare il colore e torniamo a un modello di *smooth shading* (per pixel).

È facile vedere come funziona il meccanismo di selezione apportando due modifiche al codice sorgente. Per prima cosa cambiamo il `+= 1` nel metodo `SceneObject.__init__()` in `+= 500`. In secondo luogo, contrassegniamo come commento l'istruzione `self.selecting = False` nel metodo `Window.on_mouse_press()` (che esamineremo più avanti). Ora eseguiamo il programma e facciamo clic su una sfera qualsiasi: la sfera sarà ridisegnata per visualizzare la scena di selezione, che normalmente sarebbe fuori schermo, ma per tutti gli altri aspetti opera come di consueto.

Gestione dell'interazione dell'utente

Il gioco Gravitate 3D è controllato principalmente con il mouse, tuttavia si utilizzano anche i tasti freccia per consentire di ruotare l'area di gioco, e altri tasti sono utilizzati per avviare una nuova partita e per uscire.

```
def on_mouse_press(self, x, y, button, modifiers):
    if self.gameOver:
        self._new_game()
        return
    self.selecting = True
    self.on_draw()
    self.selecting = False
    selectColor = (GLubyte * 3)()
    glReadPixels(x, y, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, selectColor)
    selectColor = tuple([component for component in selectColor])
    self._clicked(selectColor)
```

Questo metodo è richiamato da `pyglet` ogni volta che l'utente fa clic con un pulsante del mouse (purché il metodo sia stato definito). Se la partita è terminata, consideriamo questo clic come il segnale di iniziarne una nuova. Altrimenti, assumiamo che l'utente stia facendo clic su una sfera; impostiamo `selecting` a `True` e ridisegniamo la scena (questo accade fuori dallo schermo, perciò l'utente non se ne accorge) e poi riportiamo `selecting` su `False`.

La funzione `glReadPixels()` è uitilizzata per leggere i colori di uno o più pixel; in questo caso ce ne serviamo per leggere il pixel fuori schermo corrispondente alla posizione in cui l'utente ha fatto clic e recuperare il suo valore RGB come tre byte senza segno (ognuno compreso nell'intervallo da 0 a 255). Poi inseriamo questi byte in una tupla di 3 interi in modo da poterla confrontare con il colore di selezione univoco di ogni sfera.

Notate che la nostra chiamata di `glReadPixels()` presuppone un sistema di coordinate in cui l'origine di y è in basso a sinistra (in `pyglet`). Se il sistema di coordinate ha un'origine di y in alto a sinistra, servono altre due istruzioni: `viewport = (GLint * 4)(); glGetIntegerv(GL_VIEWPORT, viewport)`, e la `y` nella chiamata di `glReadPixels()` deve essere sostituita con `viewport[3] - y`.

```
def _clicked(self, selectColor):
    for x, y, z in itertools.product(range(self.board_size), repeat=3):
        if selectColor == self.board[x][y][z].selectColor:
            if (x, y, z) == self.selected:
                self._delete()  # Il secondo clic cancella
            else:
                self.selected = (x, y, z)
            return
```

Richiamiamo questo metodo ogni volta che l'utente fa clic, eccetto quando il clic comporta l'avvio di una nuova partita. Utilizziamo la funzione `itertools.product()` per produrre ogni tripletta x, y, z per l'area di gioco e confrontiamo il colore di selezione dell'oggetto che si trova nella posizione corrispondente a ciascuna tripletta con il

colore del pixel su cui è stato fatto clic: se troviamo corrispondenza, abbiamo identificato univocamente l'oggetto della scena su cui l'utente ha fatto clic. Se questo oggetto è già selezionato, allora l'utente ha fatto clic su di esso per una seconda volta, perciò cerchiamo di cancellare l'oggetto in questione e le sfere adiacenti con lo stesso colore. Altrimenti, l'utente ha fatto clic sull'oggetto per selezionarlo, perciò lo selezioniamo (deselezionando qualsiasi oggetto selezionato precedentemente).

```
def _delete(self):
    x, y, z = self.selected
    self.selected = None
    color = self.board[x][y][z].color
    if not self._is_legal(x, y, z, color):
        return
    self._delete_adjoining(x, y, z, color)
    self.label.text = "{:,}".format(self.score)
    pyglet.clock.schedule_once(self._close_up, self.delay)
```

Questo metodo è utilizzato per cancellare la sfera su cui l'utente ha fatto clic e le sfere adiacenti con lo stesso colore (e le sfere adiacenti a queste ultime con lo stesso colore). Iniziamo deselezionando la sfera selezionata, poi controlliamo se la cancellazione è lecita (cioè se esiste almeno una sfera adiacente adatta). In caso positivo, eseguiamo la cancellazione utilizzando il metodo `_delete_adjoining()` e i suoi metodi ausiliari (non mostrati qui). Poi aggiorniamo l'etichetta per mostrare il nuovo punteggio incrementato e programmiamo una chiamata del metodo `self._close_up()` (non mostrato qui) dopo mezzo secondo. In questo modo l'utente può vedere quali sfere sono state cancellate prima che tutti gli spazi siano riempiti da sfere che gravitano verso il centro (un'alternativa più sofisticata sarebbe quella di animare la chiusura spostando le sfere di uno o pochi pixel per volta fino a raggiungere le nuove posizioni).

```
def on_key_press(self, symbol, modifiers):
    if (symbol == pyglet.window.key.ESCAPE or
        ((modifiers & pyglet.window.key.MOD_CTRL or
          modifiers & pyglet.window.key.MOD_COMMAND) and
          symbol == pyglet.window.key.Q)):
        pyglet.app.exit()
    elif ((modifiers & pyglet.window.key.MOD_CTRL or
           modifiers & pyglet.window.key.MOD_COMMAND) and
           symbol == pyglet.window.key.N):
        self._new_game()
    elif (symbol in {pyglet.window.key.DELETE, pyglet.window.key.SPACE,
                     pyglet.window.key.BACKSPACE} and
          self.selected is not None):
        self._delete()
```

L'utente può terminare il programma facendo clic sul pulsante di chiusura ×, ma gli consentiamo anche di farlo premendo Esc o Ctrl+Q (o .Q). L'utente può anche avviare una nuova partita quando l'attuale è finita, con un semplice clic, oppure in qualsiasi momento premendo Ctrl+N (o .N). Gli consentiamo anche di cancellare la sfera selezionata (e le sfere adiacenti con lo stesso colore) facendo clic su di essa una

seconda volta oppure premendo i tasti Canc, Barra spaziatrice o RitCanc (Backspace).

L’oggetto `Window` ha un metodo `on_text_motion()` che gestisce i tasti freccia e ruota la scena rispetto agli assi x o y. Tale metodo non è riportato qui, perché è identico a quello che abbiamo visto precedentemente in questo stesso capitolo.

Si completa così la nostra trattazione del gioco Gravitate 3D. I soli metodi che abbiamo omesso sono quelli relativi ai dettagli della logica del gioco, in particolare i metodi che gestiscono la cancellazione di sfere adiacenti (impostando a `None` i loro colori di visualizzazione e di selezione) e quelli che provvedono a spostare le sfere verso il centro.

Creare scene 3D da programma può essere piuttosto complicato, soprattutto perché le interfacce OpenGL tradizionali sono interamente procedurali (cioè basate su funzioni) e non orientate agli oggetti. Nondimeno, grazie a `PyOpenGL` e `pyglet`, è abbastanza semplice effettuare il porting di codice C OpenGL direttamente in Python e utilizzare le interfacce OpenGL complete. Inoltre, `pyglet` in particolare offre un comodo supporto per la gestione di eventi e la creazione di finestre, mentre `PyOpenGL` fornisce l’integrazione con molti toolkit GUI, tra cui Tkinter.

In questo libro sono state spiegate molte tecniche interessanti e presentate diverse librerie utili. E nel fare ciò sono state fornite idee e ispirazioni per produrre programmi migliori con Python 3 (`http://www.python.org`).

La popolarità di Python è in continuo aumento, come anche il suo utilizzo in diversi campi applicativi e in tutti i continenti. Python è una scelta ideale come primo linguaggio, grazie al fatto che supporta la programmazione procedurale, orientata agli oggetti e funzionale, e alla sua sintassi chiara, leggera e coerente. Nondimeno, Python è un linguaggio eccezionale anche per uso professionale (come ha dimostrato Google, per esempio, in molti anni di utilizzo). Questo si deve in particolare al supporto per lo sviluppo rapido e per la produzione di codice di facile manutenzione, oltre alla facilità di accesso a potenti funzionalità scritte in C o in altri linguaggi compilati che supportano le convenzioni di chiamata del C.

Non esistono vicoli ciechi nella programmazione in Python: c'è sempre qualcosa da imparare e ci si può sempre spingere più avanti. Perciò, benché Python possa comodamente soddisfare le esigenze dei programmatori principianti, non manca delle funzioni avanzate e della profondità intellettuale in grado di soddisfare anche gli esperti più esigenti. Non solo è un linguaggio aperto nel senso della licenza e della disponibilità del codice sorgente, è aperto anche per quanto riguarda l'introspezione, fino al bytecode, se vogliamo. E naturalmente Python stesso è aperto a coloro che vogliono portare il proprio contributo (`http://docs.python.org/devguide`).

Esistono probabilmente diverse migliaia di linguaggi per computer in circolazione - anche se sono poche decine quelli ampiamente utilizzati - e oggi Python è certamente tra i più popolari. Personalmente, in qualità di informatico che ha utilizzato molti linguaggi diversi in tanti anni, ho spesso provato frustrazione per i linguaggi che i miei datori di lavoro mi chiedevano di usare. E sospetto che molti altri informatici come me pensino continuamente a creare un linguaggio migliore, uno che non abbia nessuno dei problemi e degli inconvenienti di quelli che ho sempre utilizzato, e che comprenda tutte le migliori caratteristiche dei linguaggi che ho imparato. Negli anni sono arrivato a rendermi conto che ogni volta che ho sognato un linguaggio ideale, alla fine si è rivelato essere Python, sia pure con alcune caratteristiche extra quali le costanti, la definizione opzionale dei tipi e il controllo di

accesso (attributi privati). Perciò ora non sogno più di creare il mio linguaggio di programmazione ideale: lo uso.

Grazie a Guido van Rossum e a tutti gli altri che hanno contribuito a Python, oggi e in passato, per aver consegnato al mondo un linguaggio di programmazione e un ecosistema incredibilmente potente e utile, che funziona benissimo ovunque, e che è un piacere utilizzare.

- *C++ Concurrency in Action: Practical Multithreading*

Anthony Williams (Manning Publications, Co., 2012, ISBN: 978-1-933988-77-1)

Questo libro avanzato tratta la concorrenza in C++ ma è utile anche perché descrive molti dei problemi e dei pericoli che possono affliggere i programmi concorrenti (indipendentemente dal linguaggio) e spiega come evitarli.

- *Clean Code: A Handbook of Agile Software Craftsmanship*

Robert C. Martin (Prentice Hall, 2009, ISBN: 978-0-13-235088-4)

Questo libro tratta molti degli aspetti “tattici” della programmazione: buone pratiche per l’assegnazione di nomi, la progettazione di funzioni, il refactoring e simili. Presenta molte idee che dovrebbero aiutare qualsiasi programmatore a migliorare il proprio stile di scrittura del codice e a rendere i propri programmi più semplici dal punto di vista della manutenzione (gli esempi sono scritti in Java).

- *Code Complete: A Practical Handbook of Software Construction, Second Edition*

Steve McConnell (Microsoft Press, 2004, ISBN: 978-0-7356-1967-8)

Questo libro mostra come realizzare software solido, andando oltre le specifiche del linguaggio per descrivere idee, principi e pratiche. È ricco di esempi che stimoleranno qualsiasi programmatore a riflettere in maniera più approfondito sul suo modo di programmare.

- *Design Patterns: Elements of Reusable Object-Oriented Software*

Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (Addison- Wesley, 1995, ISBN: 978-0-201-63361-0)

Uno dei più influenti libri di programmazione dei tempi moderni. I design pattern sono affascinanti e di grande utilità pratica nel lavoro quotidiano di programmazione (gli esempi sono per lo più scritti in C++).

- *Domain-Driven Design: Tackling Complexity in the Heart of Software*

Eric Evans (Addison-Wesley, 2004, ISBN: 978-0-321-12521-7)

Questo è un libro molto interessante sulla progettazione di software, particolarmente utile per progetti grandi che prevedono più persone. Si concentra in particolare sulla creazione e il raffinamento di domain model che rappresentano ciò che il sistema è progettato per fare, e sulla creazione di un linguaggio pervasivo attraverso il quale tutti coloro che sono coinvolti nel sistema, e non solo i tecnici del software, possano comunicare le loro idee.

- *Don't Make Me Think!: A Common Sense Approach to Web Usability, Second Edition*

Steve Krug (New Riders, 2006, ISBN: 978-0-321-34475-5)

Un breve e pratico libro sull'usabilità del Web basato su studi ed esperienze dell'autore. Applicando i concetti presentati nel libro, facili da comprendere, si può migliorare qualsiasi sito web di qualsiasi dimensione.

- *GUI Bloopers 2.0: Common User Interface Design Don'ts and Dos*

Jeff Johnson (Morgan Kaufmann, 2008, ISBN: 978-0-12-370643-0)

Non fatevi fuorviare dal titolo un po' colloquiale: si tratta di un libro molto serio che qualsiasi programmatore di GUI dovrebbe leggere. Non sarete d'accordo con ogni singolo suggerimento, ma vi farà riflettere con più attenzione e profondità sulla progettazione di interfacce utente.

- *Java Concurrency in Practice*

Brian Goetz, et. al. (Addison-Wesley, 2006, ISBN: 978-0-321-34960-6)

Questo libro fornisce un'eccellente trattazione della concorrenza in Java, con molti suggerimenti sulla programmazione concorrente che sono applicabili a qualsiasi linguaggio.

- *The Little Manual of API Design*

Jasmin Blanchette (Trolltech/Nokia, 2008)

Un manuale molto breve (disponibile gratuitamente presso `http://www21.in.tum.de/~blanchet/api-design.pdf`) che fornisce idee e spunti per la progettazione di API e trae la maggior parte degli esempi dal toolkit Qt.

- *Mastering Regular Expressions, Third Edition*

Jeffrey E.F. Friedl (O'Reilly Media, 2006, ISBN: 978-0-596-52812-6)

È considerato il testo di riferimento sulle espressioni regolari. È scritto in modo accessibile, con molti esempi concreti e spiegati nei dettagli.

- *OpenGL SuperBible: Comprehensive Tutorial and Reference, Fourth Edition*

Richard S. Wright, Jr., Benjamin Lipchak e Nicholas Haemel (Addison-Wesley, 2007, ISBN: 978-0-321-49882-3)

Fornisce una buona introduzione alla grafica 3D con OpenGL, adatta per programmatori privi di esperienza nella grafica 3D. Gli esempi sono scritti in C++, ma le API OpenGL sono fedelmente riprodotte in `pyglet` e altri binding OpenGL Python, perciò possono essere usati senza un eccessivo lavoro di adattamento.

- *Programming in Python 3: A Complete Introduction to the Python Language, Second Edition*

Mark Summerfield (Addison-Wesley, 2010, ISBN: 978-0-321-68056-3)

Questo libro insegna la programmazione in Python 3 a persone che sanno programmare in qualsiasi altro linguaggio di programmazione procedurale o orientato agli oggetti (incluso Python 2, naturalmente).

- *Python Cookbook, Third Edition*

David Beazley e Brian K. Jones (O'Reilly Media, 2013, ISBN: 978-1-4493-4037-7)

Questo libro è ricco di idee interessanti e pratiche che coprono tutti gli aspetti della programmazione in Python 3.

- *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming*

Mark Summerfield (Prentice Hall, 2008, ISBN: 978-0-13-235418-9)

Questo libro insegna la programmazione GUI usando Python 2 e il toolkit Qt 4. Presso il sito web dell'autore sono disponibili le versioni in Python 3 degli esempi, e quasi tutto il testo vale anche per PySide oltre che per PyQt.

- *Security in Computing, Fourth Edition*

Charles P. Pfleeger e Shari Lawrence Pfleeger (Prentice Hall, 2007, ISBN: 978-0-13-239077-4)

Questo libro fornisce una trattazione interessante, utile e pratica di un'ampia gamma di aspetti legati alla sicurezza informatica, spiegando come si possono portare degli attacchi e come proteggersi contro di essi.

- *Tcl and the Tk Toolkit, Second Edition*

John K. Ousterhout e Ken Jones (Addison-Wesley, 2010, ISBN: 978-0-321-33633-0)

Questo è considerato il testo di riferimento su Tcl/Tk 8.5. Tcl è un linguaggio non convenzionale, quasi privo di sintassi, ma il libro è utile per imparare come leggere la documentazione di Tcl/Tk, cosa che spesso è necessaria quando si scrivono applicazioni Python/Tkinter.