Il tutorial di Python

traduzione a cura di Riccardo Polignieri

16 dic 2020

Indice

Nota per la traduzione italiana.

Questa è una traduzione del Tutorial ufficiale della documentazione di Python, mantenuta in sincrono con il testo della repository GitHub.

Questa è la traduzione della **versione «di sviluppo» del Tutorial**, corrispondente a Python 3.10 e al «master branch» della repository, ed è aggiornata al 16 dicembre 2020.

La repository GitHub di questo progetto si trova qui: potete essere informati sugli aggiornamenti mettendo una «star» alla repository. Ogni collaborazione è gradita: se intendete partecipare, leggete prima il file CONTRIBUTING.rst.

Gli esempi *non* sono tradotti, perché il codice dovrebbe sempre essere scritto in Inglese. Tuttavia i commenti all'interno del codice sono tradotti e così pure le stringhe, quando il loro contenuto è utile alla comprensione del tutorial.

(traduzione a cura di Riccardo Polignieri)

Introduzione

Python è un linguaggio di programmazione molto produttivo e semplice da imparare. È dotato di efficienti strutture-dati di alto livello e propone un approccio alla programmazione a oggetti semplice ma efficace. La sua sintassi elegante e la tipizzazione dinamica, oltre al fatto di essere un linguaggio interpretato, ne fanno uno strumento ideale per lo *scripting* e lo sviluppo rapido di applicazioni, in molti campi e su molte piattaforme.

L'interprete di Python e la sua vasta libreria standard sono disponibili sul sito web https://www.python.org/, sotto forma sia di codice sorgente sia di eseguibile binario, per tutte le piattaforme più diffuse, e possono ssere liberamente redistribuiti. Il sito, inoltre, ospita direttamente o indirizza verso molti altri moduli Python, programmi e strumenti sviluppati da terze parti e documentazione aggiuntiva.

L'interprete Python si può estendere facilmente con nuove funzioni e tipi di dati implementati in C o C++ (o altri linguaggi interfacciabili con C). Python è inoltre adatto come linguaggio integrato per estendere e personalizzare altre applicazioni.

Questo tutorial introduce il lettore in modo informale ai concetti e alle funzioni di base del linguaggio e del suo ambiente. Può essere utile avere sotto mano un interprete Python per provare direttamente il codice, ma tutti gli esempi mostrano anche il loro output: è quindi possibile leggere il tutorial anche a sé stante.

Per una descrizione delle funzionalità e dei moduli, si veda la documentazione della libreria standard. La guida di riferimento del linguaggio approfondisce in modo formale la struttura di Python. Per scrivere estensioni in C o C++ si può consultare la guida su come estendere e incorporare Python e il manuale delle API C di Python. Ci sono poi molti libri dedicati all'approfondimento di Python.

Questo tutorial non vuole essere una descrizione esauriente che copre ogni singola funzionalità, o anche solo quelle comuni. Piuttosto, è un'introduzione agli aspetti più notevoli di Python e può dare un'idea dello stile e del «gusto» del linguaggio. Dopo averlo letto, sarete in grado di leggere e scrivere moduli e programmi in Python, e sarete pronti ad approfondire i diversi moduli contenuti nella libreria standard.

Vale anche la pena di dare un'occhiata al glossario.

CAPITOLO 1

Per stuzzicare l'appetito.

Se lavorate molto al computer, prima o poi vi troverete davanti a un compito che vi piacerebbe automatizzare. Per esempio, fare un trova-e-sostituisci su molti file contemporaneamente, o rinominare e riordinare un gruppo di foto secondo una regola complicata. Magari vi piacerebbe creare un piccolo database personale, o un'applicazione grafica per uno scopo specifico, o un semplice gioco.

Se siete un programmatore professionista, forse dovete lavorare con diverse librerie C, C++ o Java, ma il consueto ciclo di scrivere, compilare, testare, ricompilare vi sembra troppo lento. Forse state scrivendo una *suite* di test per una libreria di questo tipo e pensate che scrivere il codice dei test sia un compito noioso. O forse avete scritto un programma che potrebbe aver bisogno di un linguaggio interno per le estensioni, ma non avete voglia di progettarne e implementarne uno da zero per la vostra applicazione.

Allora Python è proprio il linguaggio che fa per voi.

Per alcuni di questi compiti potrebbe bastare uno script della shell di Unix o un *batch file* di Windows: gli script però vanno bene per spostare i file e modificare i dati testuali, non sono adatti alle applicazioni grafiche o ai giochi. Potreste scrivere un programma in C, C++ o Java, ma questo richiederebbe molto tempo di sviluppo anche solo per arrivare a una prima bozza. Python è più semplice, è disponibile su Windows, Mac OS X e Unix, e vi aiuterà a finire il lavoro più in fretta.

Python è semplice da usare, ma è un linguaggio di programmazione serio che offre molta più struttura e supporto per programmi di grandi dimensioni, rispetto a uno script della shell o un batch file. D'altra parte, Python ha anche molta più gestione delle eccezioni rispetto a C; essendo poi un linguaggio *particolarmente* «di alto livello», include tipi di dati di alto livello e flessibili, come le sue liste e dizionari. Grazie alle sue strutture-dati più astratte, Python si può usare in campi applicativi molto più vasti rispetto ad Awk o anche a Perl, pur mantenendo la stessa semplicità d'uso di questi linguaggi.

Python vi consente di dividere il vostro programma in moduli che possono essere riutilizzati in altri programmi. Include già una vasta collezione di moduli standard, che potete usare come base per il vostro lavoro, o come esempi per imparare la programmazione in Python. Questi moduli, tra l'altro, offrono soluzioni per l'input/output dei file, le chiamate di sistema, i socket e perfino interfacce per *toolkit* grafici come Tk.

Python è un linguaggio interpretato, cosa che fa risparmiare molto tempo durante lo sviluppo, perché non c'è bisogno di compilare e collegare nulla. L'interprete può essere usato in modalità interattiva ed è quindi facile sperimentare con le funzionalità del linguaggio, scrivere programmi usa-e-getta, testare i costrutti durante lo sviluppo *bottom-up*. Può essere anche una pratica calcolatrice da tenere sottomano.

Python vi consente di scrivere codice compatto e leggibile. I programmi scritti in Python sono in genere molto più corti degli equivalenti in C, C++ o Java, per diverse ragioni:

- i tipi di dato di alto livello vi permettono di codificare operazioni complesse in una singola istruzione;

- il raggruppamento delle istruzioni avviene rientrando il codice, invece di racchiuderlo tra parentesi;
- non c'è bisogno di dichiarare le variabili.

Python è *estensibile*: se conoscete il C, è facile aggiungere all'interprete una nuova funzione predefinita o un modulo, sia per aumentare la velocità di esecuzione in punti critici del codice, sia per collegare un programma Python a librerie disponibili solo in forma binaria (per esempio, librerie grafiche di terze parti). Una volta che siete diventati esperti, potete collegare l'interprete Python all'interno di un programma scritto in C e usarlo come un'estensione, o un linguaggio interno di quel programma.

A proposito, il nome del linguaggio deriva dallo show della BBC «Monty Python's Flying Circus» e non ha niente a che vedere con i rettili. Ogni riferimento agli sketch dei Monty Python nella documentazione è non solo permesso ma anzi incoraggiato.

Adesso che siete incuriositi da Python, avrete voglia di esaminarlo più nel dettaglio. Siccome il miglior modo di imparare un linguaggio è usarlo, vi invitiamo a sperimentare con l'interprete man mano che leggete il tutorial.

Dedichiamo il prossimo capitolo a spiegare il meccanismo di funzionamento dell'interprete. Si tratta di informazioni di servizio, ma sono importanti per consentirvi di provare gli esempi che verranno presentati più in là.

I capitoli successivi descrivono e dimostrano diverse funzionalità di Python e del suo ambiente, a cominciare da semplici espressioni, istruzioni e tipi di dati, proseguendo poi con le funzioni e i moduli, fino ad accennare agli argomenti più avanzati come le eccezioni e la creazione di classi personalizzate.

CAPITOLO 2

Usare l'interprete Python.

2.1 Invocare l'interprete

L'interprete Python di solito è installato in `/usr/local/bin/python3.10` sui computer dove è disponibile. Aggiungere `/usr/local/bin` alla *path* della shell Unix vi permette di invocarlo con il comando[1]

```
python3.10
```

Dal momento che la scelta della directory dell'interprete è un'opzione di installazione, sono possibili altre configurazioni. Chiedete a un esperto di Python o all'amministratore del sistema. Per esempio, un'alternativa popolare è `/usr/local/python`.

Su Windows, se avete installato Python dal Microsoft Store, sarà disponibile il comando `python3.10`. Se avete installato il *launcher* py.exe potete usare il comando `py`. Il paragrafo Excursus: Impostare le variabili d'ambiente descrive altri modi per avviare Python.

Inserire il carattere terminatore del file (`Control-D` in Unix, `Control-Z` in Windows) nel *prompt* principale costringe l'interprete a uscire con *exit status* `0`. Se non funziona, potete uscire dall'interprete con questo comando: `quit()`.

Sui sistemi che supportano la libreria GNU Readline, l'interprete consente di fare *editing* interattivo, autocompletamento e storico degli inserimenti. Il modo più rapido per controllare se l'editing della riga di comando è supportato, è inserire `Control-P` appena il prompt primario di Python è disponibile. Se sentite un beep, l'editing è supportato; l'Appendice *Editing e history substitution nell'input interattivo* presenta un'introduzione a questi comandi. Se invece non succede nulla, o se risponde semplicemente `^P`, l'editing non è disponibile: potete solo usare il *backspace* per cancellare i caratteri.

L'interprete funziona più o meno come la shell di Unix: quando è invocato con lo *standard input* collegato al terminale, legge ed esegue interattivamente i comandi; quando è invocato con il nome di un file come parametro, o con lo standard input collegato a un file, legge ed esegue il file come *script*.

Un altro modo di avviare l'interprete è `python -c command [arg] ...`, che esegue l'istruzione (o le istruzioni) in *command*, analogamente all'opzione `-c` della shell. Siccome le istruzioni Python di solito comprendono spazi o altri caratteri speciali della shell, è consigliabile racchiudere *command* tra apici singoli.

Alcuni moduli Python possono essere eseguiti come script. Si possono invocare con `python -m module [arg] ...`, che esegue il file *module* come se ne aveste scritto il nome per intero nella riga di comando.

[1] In ambiente Unix, l'eseguibile dell'interprete Python 3.x *non* è installato col nome `python`, così da non entrare in conflitto con l'eseguibile di Python 2.x, anch'esso contemporaneamente presente.

Quando si esegue uno script, talvolta è utile poter entrare in modalità interattiva al termine dell'esecuzione del file. Per fare questo, passate l'opzione `-i` prima del nome dello script.

Tutte le opzioni della riga di comando sono descritte nel capitolo Riga di comando e ambiente della documentazione.

2.1.1 Passare dei parametri

Quando specificati, il nome dello script ed eventuali successivi parametri sono convertiti in una lista di stringhe e assegnate alla variabile `argv` del modulo `sys`. Potete accedere a questa lista eseguendo `import sys`. La lista contiene sempre almeno un elemento; se non passate nessuno script né altri parametri, `sys.argv[0]` è una stringa vuota. Quando invece di uno script passate `'-'`, per indicare lo standard input, allora `sys.argv[0]` è impostato a `'-'`. Quando usate `-c` *command*, allora `sys.argv[0]` è `'-c'`. Quando usate `-m` *module*, `sys.argv[0]` è il nome completo del modulo eseguito. Le opzioni eventualmente passate dopo `-c` *command* oppure `-m` *module* non sono processate dall'interprete Python ma sono comunque disponibili in `sys.argv` e possono quindi essere gestite dal modulo o dal comando.

2.1.2 Modalità interattiva

Quando i comandi sono letti da un terminale, l'interprete è in *modalità interattiva*. In questa condizione, l'interprete resta in attesa del comando successivo presentando il *prompt primario*, di solito tre segni «maggiore-di» (>>>). Per le linee di continuazione viene usato il *prompt secondario*, in genere tre punti (...). L'interprete stampa un messaggio di benvenuto che riporta il numero di versione e l'indicazione del copyright, prima di presentare il prompt:

```
$ python3.10
Python 3.10 (default, June 4 2019, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Le linee di continuazione sono necessarie per i costrutti multi-linea. Per esempio, osservate questa istruzione `if`:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

Per ulteriori informazioni sulla modalità interattiva, si veda *Modalità interattiva*.

2.2 L'interprete e il suo ambiente

2.2.1 *Encoding* del codice

I file di codice Python sono processati con l'encoding UTF-8 di default. In questo encoding, i caratteri della gran parte dei linguaggi umani possono essere usati contemporaneamente nelle stringhe di testo, negli identificatori e nei commenti. Tuttavia la libreria standard usa esclusivamente caratteri ASCII per gli identificatori, una convenzione che il codice interessato alla compatibilità dovrebbe rispettare. Per visualizzare correttamente i caratteri, il vostro editor deve saper riconoscere l'encoding UTF-8 e deve usare un font che supporta tutti i caratteri usati nel file.

Per dichiarare un encoding diverso da quello di default, occorre aggiungere una riga speciale di commento esattamente *all'inizio* del file. La sintassi è questa:

```
# -*- coding: encoding -*-
```

dove *encoding* è uno dei vari `codecs` supportati da Python.

Per esempio, per dichiarare che occorre usare l'encoding Windows-1252 per leggere il file, la prima riga del codice dovrebbe essere:

```
# -*- coding: cp1252 -*-
```

L'eccezione alla regola è quando lo script inizia invece con una *shebang UNIX*. In questo caso, la dichiarazione di encoding deve essere la seconda riga del file. Per esempio:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

CAPITOLO 3

Un'introduzione informale a Python

Negli esempi che seguono, l'input e l'output si distinguono dalla presenza o assenza del *prompt* (>>> e ...). Per riprodurre gli esempi, quando appare il prompt dovete inserire tutto ciò che segue nella riga; le righe che non hanno il prompt sono l'output dell'interprete. Si noti che quando una riga riporta solo il prompt secondario senza nient'altro, significa che dovete inserire una riga vuota: questo serve a terminare un'istruzione multi-linea.

Molti esempi nella documentazione, anche quelli da inserire nell'interprete interattivo, includono dei commenti. I commenti in Python iniziano con il «cancelletto» # e comprendono tutto il resto della riga. Un commento può partire all'inizio della riga o dopo uno spazio, ma non può essere inserito all'interno di una stringa: un cancelletto dentro una stringa è solo un cancelletto. Siccome i commenti servono a spiegare il codice, ma non sono processati da Python, non dovete copiarli quando provate gli esempi.

Ecco alcuni esempi:

```
# questo è il primo commento
spam = 1  # e questo è il secondo
          # ... ed ecco il terzo!
text = "# Questo non è un commento perché è dentro una stringa."
```

3.1 Usare Python come una calcolatrice

Proviamo qualche semplice istruzione Python. Avviate l'interprete e aspettate il prompt primario >>> (non dovrebbe volerci molto).

3.1.1 Numeri

L'interprete funziona come una semplice calcolatrice: potete inserire un'espressione e lui restituirà il suo valore. La sintassi delle espressioni è banale: gli operatori +, −, * e / funzionano come negli altri linguaggi (Pascal o C, per esempio); le parentesi (()) si possono usare per i raggruppamenti. Per esempio:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
```

(continua...)

(...segue)

```
>>> 8 / 5  # dividere produce sempre un numero con la virgola
1.6
```

I numeri interi (per es. `2`, `4`, `20`) hanno il tipo `int`, quelli frazionari (es. `5.0`, `1.6`) hanno il tipo `float`. Vedremo altri tipi numerici più avanti in questo tutorial.

La divisione (`/`) restituisce sempre un numero *float*. Per arrotondare e ottenere un intero, scartando la parte frazionaria, potete usare l'operatore `//`. Per ottenere il resto usate `%`:

```
>>> 17 / 3  # la divisione normale restituisce un float
5.666666666666667
>>>
>>> 17 // 3  # l'arrotondamento scarta la parte frazionaria
5
>>> 17 % 3  # l'operatore % restituisce il resto della divisione
2
>>> 5 * 3 + 2  # risultato * divisore + resto
17
```

Con Python è possibile usare l'operatore `**` per calcolare le potenze[1]:

```
>>> 5 ** 2  # 5 al quadrato
25
>>> 2 ** 7  # 2 alla settima
128
```

Il segno di «uguale» (=) viene usato per assegnare un valore a una variabile. Nessun risultato viene mostrato prima del successivo prompt interattivo:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Cercare di usare una variabile non «definita» (che non ha un valore assegnato), produce un errore:

```
>>> n  # cerco di accedere a una variabile non definita
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

I numeri «con la virgola» (float) sono pienamente supportati; le operazioni che coinvolgono operandi di tipo misto convertono automaticamente gli interi in float:

```
>>> 4 * 3.75 - 1
14.0
```

In modalità interattiva, l'ultima espressione restituita è assegnata alla variabile `_`. Ciò vuol dire che, quando usate Python come una calcolatrice, è più semplice riportare i risultati, per esempio:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

[1] Dal momento che `**` ha una priorità più alta di `-`, `-3**2` sarà interpretato come `-(3**2)` ovvero `-9`. Per evitare questo e ottenere invece `9`, potete usare `(-3)**2`.

Questa variabile dovrebbe essere considerata di sola lettura. Non cercate di assegnare esplicitamente un valore a _: avreste creato una variabile locale con lo stesso nome, che maschera la quella predefinita, con il suo comportamento speciale.

Oltre a `int` e `float`, Python supporta altri tipi numerici, come `Decimal` e `Fraction`. Python ha anche il supporto per i numeri complessi e usa il suffisso `j` o `J` per la parte immaginaria (e.g. `3+5j`).

3.1.2 Stringhe

Oltre ai numeri, Python può manipolare le stringhe, che si possono esprimere in molti modi. Potete delimitarle con apici singoli (`'...'`) o doppi (`"..."`): funzionano allo stesso modo[2]. Usate \ (*backslash*) per fare *escaping* degli apici:

```
>>> 'spam eggs'  # apici singoli
'spam eggs'
>>> 'doesn\'t'  # usate \' per inserire un apice singolo nella stringa...
"doesn't"
>>> "doesn't"  # ...o usate apici doppi per delimitarla
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

Nell'output dell'interprete interattivo, le stringhe sono chiuse tra apici e i caratteri speciali sono resi con il *backslash* di *escape*. A volte l'output può sembrare diverso dall'input, perché gli apici possono cambiare, ma le due versioni sono equivalenti. La stringa è chiusa nei doppi apici se contiene un apice singolo e nessun apice doppio; altrimenti è delimitata da apici singoli. La funzione `print()` produce un output più leggibile perché omette gli apici iniziali e finali, e «stampa» anche i caratteri speciali:

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n significa "a-capo"
>>> s  # senza print(), \n viene incluso nell'output
'First line.\nSecond line.'
>>> print(s)  # con print(), \n produce una nuova riga
First line.
Second line.
```

Se non volete che il carattere dopo un *backslash* \ sia interpretato come un carattere speciale, potete usare le *raw strings* con il prefisso `r` prima dell'apice iniziale:

```
>>> print('C:\some\name')  # qui \n vuol dire "a-capo"!
C:\some
ame
>>> print(r'C:\some\name')  # si noti la r iniziale
C:\some\name
```

Le stringhe possono occupare più di una riga. Un modo per ottenere questo è usare gli apici tripli: `"""..."""` o `'''...'''`. Gli «a-capo» sono inclusi automaticamente nelle stringhe, ma è possibile evitarlo aggiungendo un *backslash* \ alla fine della riga. Questo esempio:

```
print("""\
Usage: thingy [OPTIONS]
```

(continua...)

[2] A differenza di altri linguaggi, i caratteri speciali come `\n` hanno lo stesso significato con apici singoli (`'...'`) o doppi (`"..."`). L'unica differenza tra i due è che all'interno di apici singoli non c'è bisogno di fare *escaping* di `"` (ma occorre farlo per `\'`) e viceversa.

(...segue)

```
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

produce questo output (si noti che lo «a-capo» iniziale non è incluso):

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Potete concatenare («incollare insieme») le stringhe con l'operatore + e ripeterle con il *:

```
>>> # 3 volte 'un', seguito da 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Due o più stringhe (racchiuse tra apici) una accanto all'altra sono automaticamente concatenate.

```
>>> 'Py' 'thon'
'Python'
```

Questo torna utile quando volete spezzare una stringa lunga:

```
>>> text = ('Mettete diverse stringhe tra parentesi '
...         'per unirle insieme.')
>>> text
'Mettete diverse stringhe tra parentesi per unirle insieme.'
```

Questo però funziona solo con le stringhe «pure», non con le variabili o le espressioni:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # non potete concatenare una variabile e una stringa
  File "<stdin>", line 1
    prefix 'thon'
           ^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^
SyntaxError: invalid syntax
```

Per concatenare le variabili, o una variabile con una stringa, usate l'operatore +:

```
>>> prefix + 'thon'
'Python'
```

Le stringhe possono essere *indicizzate* (indirizzate): il primo carattere ha indice 0. Non esiste un tipo di dato separato per rappresentare un carattere; un carattere è semplicemente una stringa di lunghezza uno:

```
>>> word = 'Python'
>>> word[0]  # il carattere in posizione 0
'P'
>>> word[5]  # il carattere in posizione 5
'n'
```

Gli indici possono anche essere negativi, contando a partire da destra:

```
>>> word[-1]  # l'ultimo carattere
'n'
>>> word[-2]  # il penultimo carattere
```

(continua...)

(...segue)

```
'o'
>>> word[-6]
'P'
```

Si noti che, siccome -0 è lo stesso di 0, gli indici negativi partono da -1.

Oltre agli indici, è anche consentito *sezionare* (*slicing*). Se gli indici restituiscono un singolo carattere, le sezioni vi permettono di estrarre sotto-stringhe:

```
>>> word[0:2]  # i caratteri dalla posizione 0 inclusa a 2 esclusa
'Py'
>>> word[2:5]  # i caratteri dalla posizione 2 inclusa a 5 esclusa
'tho'
```

Si noti che l'inizio è sempre incluso, la fine è esclusa. Questo fa sì che `s[:i] + s[i:]` sia sempre uguale a `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Gli indici delle sezioni hanno dei pratici valori di default: se si omette il primo indice, vuol dire «0»; se si omette il secondo, vuol dire «la lunghezza della stringa».

```
>>> word[:2]   # i caratteri dall'inizio alla posizione 2 esclusa
'Py'
>>> word[4:]   # i caratteri dalla posizione 4 inclusa alla fine
'on'
>>> word[-2:]  # i caratteri dalla penultima posizione inclusa alla fine
'on'
```

Un trucco per ricordare come funzionano le sezioni è pensare che gli indici puntino tra un carattere e l'altro, con lo spazio a sinistra del primo carattere che vale 0. Allora, lo spazio a destra dell'ultimo carattere di una stringa di lunghezza *n* avrà indice *n*. Per esempio:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

I numeri della prima riga sono le posizioni degli indici 0…6 della stringa; la seconda riga riporta i corrispondenti indici negativi. La sezione da *i* a *j* è composta da tutti i caratteri che stanno tra gli spazi numerati da *i* a *j*.

Per gli indici non-negativi, la lunghezza di una sezione è la differenza tra gli indici, se entrambi non escono dai limiti della stringa. Per esempio, la lunghezza di `word[1:3]` è 2.

Se usate un indice troppo grande, otterrete un errore:

```
>>> word[42]  # la stringa ha solo 6 caratteri
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Tuttavia, gli indici che escono dai limiti sono comunque consentiti, quando li usiamo per estrarre una sezione:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Le stringhe in Python non possono essere modificate: sono immutabili. Di conseguenza, assegnare alla posizione di un indice produce un errore:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Se vi serve una nuova stringa, dovete crearla:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

La funzione predefinita `len()` restituisce la lunghezza di una stringa:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Vedi anche:

Sequenze di testo - str Le stringhe sono esempi del tipo di dati *sequenza*, e supportano le comuni operazioni possibili con le sequenze.

Metodi per le stringhe Le stringhe hanno un gran numero di metodi per manipolazioni di base e ricerca.

Stringhe formattate Le stringhe possono includere delle espressioni al loro interno.

Sintassi di format Informazioni sulla formattazione delle stringhe con `str.format()`.

Formattazione in stile printf Il vecchio modo di formattare, con l'operatore `%` a destra della stringa.

3.1.3 Liste

Python ha alcuni tipi di dati *composti*, che servono a raggruppare insieme altri dati. Il più versatile di questo è la *lista*, che si può scrivere come un elenco di valori (elementi) separati da virgola e racchiusi tra parentesi quadre. Le liste possono contenere valori di tipo diverso, anche se di solito tutti gli elementi hanno lo stesso tipo.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Come le stringhe e tutti gli altri tipi di sequenza, le liste possono essere indicizzate e sezionate:

```
>>> squares[0]  # l'indice restituisce l'elemento
1
>>> squares[-1]
25
>>> squares[-3:]  # la sezione restituisce una nuova lista
[9, 16, 25]
```

Tutte le operazioni di sezionamento restituiscono una nuova lista che contiene gli elementi richiesti. Ciò significa che la sezione dell'esempio seguente restituisce una copia per indirizzo della lista:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Le liste supportano anche operazioni come il concatenamento:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

A differenza delle stringhe che sono immutabili le liste sono un tipo mutabile, per cui è possibile cambiare il loro contenuto:

```
>>> cubes = [1, 8, 27, 65, 125]  # c'è qualcosa di sbagliato
>>> 4 ** 3  # 4 al cubo fa 64, non 65!
64
>>> cubes[3] = 64  # rimpiazza il valore sbagliato
>>> cubes
[1, 8, 27, 64, 125]
```

Potete anche aggiungere nuovi elementi alla fine della lista, con il metodo `append()` (parleremo meglio dei metodi più tardi):

```
>>> cubes.append(216)  # aggiunge il cubo di 6
>>> cubes.append(7 ** 3)  # e il cubo di 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

È possibile inoltre assegnare a una sezione, cosa che può anche cambiare la dimensione della lista o svuotarla del tutto:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # rimpiazza alcuni valori
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # adesso li rimuove
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # svuota la lista rimpiazzando tutti gli elementi con una lista vuota
>>> letters[:] = []
>>> letters
[]
```

La funzione predefinita `len()` si applica anche alle liste:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

È possibile *annidare* le liste, ovvero creare liste dentro altre liste. Per esempio:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 I primi passi verso la programmazione

Certamente possiamo usare Python per compiti più complessi che sommare due più due. Per esempio, possiamo scrivere i primi numeri della serie di Fibonacci in questo modo:

```
>>> # serie di Fibonacci:
... # la somma di due elementi è l'elemento seguente
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

Questo esempio introduce diversi aspetti nuovi.

- La prima riga contiene un *assegnamento multiplo*: le variabili `a` e `b` ottengono simultaneamente i valori 0 e 1. Nell'ultima riga il trucco si ripete, mostrando così che le espressioni nella parte destra sono tutte valutate *prima* che l'assegnamento abbia luogo. Le espressioni della parte destra sono valutate nell'ordine, da sinistra a destra.
- Un ciclo `while` viene eseguito fin quando la condizione (in questo caso, `a < 10`) resta verificata. In Python, come in C, tutti gli interi tranne lo zero sono «veri». Lo zero è «falso». La condizione può anche riguardare una stringa o una lista, o in effetti qualsiasi sequenza. Tutto ciò che ha lunghezza non-nulla è «vero»; le sequenza vuote sono «false». Il test usato in questo esempio è una semplice comparazione. Gli operatori standard per la comparazione sono gli stessi di C: < (minore di), > (maggiore di), == (uguale a), <= (minore o uguale a), >= (maggiore o uguale a) e `!=` (diverso da).
- Il *corpo* del ciclo è *rientrato*: il rientro è il modo di Python per raggruppare le istruzioni. In modalità interattiva, dovete inserire una tabulazione o degli spazi per ciascuna riga rientrata. In realtà, preparerete le istruzioni più complicate in un editor da programmatore: tutti gli editor validi hanno la funzione di rientro automatico. Quando inserite un'istruzione composta in modalità interattiva, dovete concluderla con una riga bianca per indicare che è terminata, dal momento che il parser non può indovinare quando avete inserito l'ultima riga. Si noti che ciascuna riga all'interno di un blocco deve essere rientrata della stessa misura.
- La funzione `print()` scrive il valore del parametro o dei parametri che le passate. È diverso da scrivere semplicemente l'espressione da calcolare (come avete fatto prima nell'esempio della calcolatrice), in quanto `print()` può gestire più parametri, numeri con la virgola e stringhe. Le stringhe sono stampate senza apici; tra ciascun parametro viene inserito uno spazio, per permettervi di formattare l'output in modo elegante, così:

  ```
  >>> i = 256*256
  >>> print('Il valore di i è', i)
  Il valore di i è 65536
  ```

  Potete usare il parametro *keyword* «end» per evitare l'inserimento di una riga vuota dopo ciascun output, o per terminare l'output con una stringa diversa:

  ```
  >>> a, b = 0, 1
  >>> while a < 1000:
  ...     print(a, end=',')
  ...     a, b = b, a+b
  ...
  0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
  ```

CAPITOLO 4

Altri strumenti per il controllo di flusso

Oltre a `while` di cui abbiamo già parlato, Python utilizza le consuete istruzioni per il controllo del flusso, comuni a molti linguggi, con qualche peculiarità.

4.1 Istruzione `if`

Forse l'istruzione più famosa è la `if`. Per esempio:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Possono esserci nessuna, una o più sezioni `elif` e la sezione `else` è opzionale. La parola riservata "`elif`" è una scorciatoia per «else if», e permette di evitare troppi livelli annidati. Una sequenza `if ... elif ... elif ...` sostituisce le istruzioni `switch` o `case` tipiche di altri linguaggi.

4.2 Istruzione `for`

Se siete abituati a Pascal o a C, troverete l'istruzione `for` in Python leggermente diversa. Invece di iterare solo su una progressione aritmetica, come in Pascal, o dare la possibilità di definire sia il passo dell'iterazione sia la condizione d'arresto, come in C, il `for` di Python itera sugli elementi di una qualsiasi sequenza (una lista, una stringa...), nell'ordine in cui appaiono nella sequenza. Per esempio, ma senza alcun sottinteso omicida:

```
>>> # Misura la lunghezza di alcune stringhe:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Il codice che *modifica* una collezione mentre itera sulla stessa può essere complicato da scrivere correttamente. Di solito è più semplice iterare su una *copia* della collezione, o crearne una nuova:

```
# Una collezione di esempio
users = {'Hans': 'active', 'Eleonore': 'inactive', 'Keitaro': 'active'}

# Strategia: iterare su una copia
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategia: creare una nuova collezione
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3 La funzione `range()`

Se dovete iterare su una sequenza di numeri, la funzione predefinita `range()` è molto comoda. Produce una progressione aritmetica:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Il punto di arresto indicato non fa parte della sequenza generata: `range(10)` produce dieci valori, che sono anche gli indici corretti per una sequenza di lunghezza 10. Potete far partire l'intervallo da un numero diverso o specificare un incremento, anche negativo. A volte l'incremento è chiamato «il passo»:

```
range(5, 10)
   5, 6, 7, 8, 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

Per iterare sugli indici di una sequenza, potete combinare le funzioni `range()` e `len()` come segue:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In casi del genere, tuttavia, vi conviene usare la funzione `enumerate()`: si veda per questo *Tecniche di iterazione*.

Se cercate semplicemente di «stampare» un intervallo, succede una cosa strana:

```
>>> print(range(10))
range(0, 10)
```

L'oggetto restituito da `range()` si comporta in modo simile a una lista, ma in effetti non lo è. In realtà è un oggetto che restituisce l'elemento successivo della sequenza desiderata, quando vi iterate sopra, ma non *crea* davvero la lista, per risparmiare spazio.

Chiamiamo iterabile un oggetto di questo tipo: ovvero, un oggetto adatto a essere usato da funzioni e costrutti che si aspettano qualcosa da cui ottenere via via elementi successivi, finché ce ne sono. Abbiamo visto che l'istruzione `for` è un costrutto di questo tipo; invece, un esempio di funzione che accetta un iterabile come argomento è `sum()`:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Vedremo più in là altri esempi di funzioni che restituiscono degli iterabili, o che accettano iterabili come argomento. Infine, se siete curiosi di sapere come si può ottenere una lista da un `range()`, ecco la risposta:

```
>>> list(range(4))
[0, 1, 2, 3]
```

Nel capitolo *Strutture dati* approfondiremo ancora la funzione `list()`.

4.4 Le istruzioni `break` e `continue`, e la clausola `else` nei cicli

L'istruzione `break` come in C, «salta fuori» dal ciclo `for` o `while` più interno in cui è inserita.

Le istruzioni di iterazione possono avere una clausola `else`: questa viene eseguita quando il ciclo termina perché l'iterabile si è esaurito (in un `for`), o perché la condizione è divenuta «falsa» (in un `while`); non viene però eseguita quando il ciclo termina a causa di una istruzione `break`. Per esempio, il ciclo seguente ricerca i numeri primi:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'è uguale a', x, '*', n//x)
...             break
...     else:
...         # il ciclo è finito senza trovare un fattore primo
...         print(n, 'è un numero primo')
...
2 è un numero primo
3 è un numero primo
```

(continua...)

(...segue)

```
4 è uguale a 2 * 2
5 è un numero primo
6 è uguale a 2 * 3
7 è un numero primo
8 è uguale a 2 * 4
9 è uguale a 3 * 3
```

(Sì, questo codice è giusto. Fate attenzione: la clausola `else` appartiene al ciclo `for`, *non* all'istruzione `if`.)

Quando viene usata in un ciclo, la clausola `else` è più simile alla `else` di un'istruzione `try`, piuttosto che a quella di un `if`. La `else` di un'istruzione `try` viene eseguita quando non sono rilevate eccezioni, e allo stesso modo la `else` di un ciclo viene eseguita quando non ci sono `break`. Approfondiremo l'istruzione `try` e le eccezioni nel capitolo *Gestire le eccezioni*.

L'istruzione `continue`, anch'essa un prestito dal C, prosegue con la successiva iterazione del ciclo:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Trovato un numero pari", num)
...         continue
...     print("Trovato un numero dispari", num)
Trovato un numero pari 2
Trovato un numero dispari 3
Trovato un numero pari 4
Trovato un numero dispari 5
Trovato un numero pari 6
Trovato un numero dispari 7
Trovato un numero pari 8
Trovato un numero dispari 9
```

4.5 L'istruzione `pass`

L'istruzione `pass` non fa nulla. Può essere usata quando sintatticamente è richiesta un'istruzione, ma il programma in sé non ha bisogno di fare nulla. Per esempio:

```
>>> while True:
...     pass  # Blocca in attesa dell'interruzione da tastiera (Ctrl+C)
...
```

Si usa di solito per creare una classe elementare:

```
>>> class MyEmptyClass:
...     pass
...
```

Un altro modo di usare `pass` è come segnaposto per una funzione o una condizione, quando state scrivendo codice nuovo e volete ragionare in termini più astratti. Il `pass` verrà ignorato silenziosamente:

```
>>> def initlog(*args):
...     pass   # Ricordati di implementare questa funzione!
...
```

4.6 Definire le funzioni

Possiamo creare una funzione che scrive i numeri di Fibonacci fino a un limite determinato:

```
>>> def fib(n):    # scrive la serie di Fibonacci fino a n
...     """Scrive la serie di Fibonacci fino a n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Adesso chiamate la funzione appena definita:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

La parola chiave `def` introduce la *definizione* di una funzione. Deve essere seguita dal nome della funzione e da una lista di parametri *formali* tra parentesi. Le istruzioni che compongono il corpo della funzione iniziano nella riga successiva, e devono essere rientrate.

Opzionalmente, la prima istruzione della funzione può essere una stringa non assegnata: questa è la *docstring*, ovvero la stringa di documentazione della funzione. Potete trovare altre informazioni nella sezione *Stringhe di documentazione*. Esistono strumenti che usano le docstring per generare automaticamente la documentazione online o stampata, o per consentire all'utente di accedervi interattivamente. Includere la documentazione nel vostro codice è una buona pratica e dovrebbe diventare un'abitudine.

L'esecuzione di una funzione produce una nuova tabella dei simboli usati per le variabili locali alla funzione. Più precisamente, tutti gli *assegnamenti* fatti all'interno della funzione conservano il valore in una tabella dei simboli locale; invece, i *riferimenti* alle variabili per prima cosa cercano il nome nella tabella locale, quindi nella tabella locale delle eventuali funzioni «superiori» in cui la nostra può essere inclusa, quindi nella tabella dei simboli globali, infine nella tabella dei nomi predefiniti. Di conseguenza è possibile *riferirsi* a una variabile globale o di una funzione superiore, ma non è possibile *assegnarle* un valore (a meno di non ricorrere all'istruzione `global` per le variabili globali, o a `nonlocal` per quelle delle funzioni superiori).

I parametri *reali* (gli argomenti[1]) di una funzione sono introdotti nella tabella dei simboli locali nel momento in cui la funzione è chiamata. Quindi, gli argomenti sono «passati per valore» (dove però il «valore» è sempre un *riferimento* all'oggetto, non il valore dell'oggetto).[2] Quando una funzione chiama un'altra funzione, una nuova tabella di simboli è creata per quella chiamata.

La *definizione* della funzione associa il nome della funzione con l'oggetto-funzione nella tabella dei simboli corrente. L'interprete riconosce l'oggetto a cui punta il nome come un oggetto-funzione definito dall'utente. Anche altri nomi possono puntare al medesimo oggetto-funzione e possono essere usati per accedere alla funzione:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Se avete esperienza con altri linguaggi, potreste obiettare che `fib` non è una funzione ma una procedura, dal momento che non restituisce un valore. Tuttavia in Python anche le funzioni senza un'istruzione `return` esplicita *restituiscono* in effetti un valore, per quanto piuttosto insignificante. Questo valore si chiama `None` (è un nome predefinito). L'interprete di solito evita di emettere direttamente `None` in output, quando è l'unica cosa che dovrebbe scrivere. Se volete davvero vedere il `None`, potete usare la funzione `print()`:

[1] ndT: in questa traduzione italiana cerchiamo di mantenere una coerente, se pure acrobatica, distinzione tra *parametri* (quelli formali, che appaiono nella *definizione* della funzione) e *argomenti* (i parametri reali, che appaiono nella *chiamata* della funzione). Il testo originale è talvolta meno preciso.

[2] In effetti, una descrizione più accurata sarebbe *passati per riferimento all'oggetto*, dal momento che, se viene passato un oggetto mutabile, il codice chiamante vedrà tutte le modifiche fatte dal codice chiamato (come l'inserimento di elementi in una lista).

```
>>> fib(0)
>>> print(fib(0))
None
```

Non è difficile scrivere una funzione che *restituisce* una lista di numeri di Fibonacci, invece di scriverla:

```
>>> def fib2(n):  # restituisce i numeri di Fibonacci fino a n
...     """Restituisce una lista con i numeri Fibonacci fino a n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # vedi sotto
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # chiama la funzione
>>> f100                # scrive il risultato
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Questo esempio, come di consueto, introduce alcuni concetti nuovi:

- L'istruzione `return` esce dall'esecuzione della funzione restituendo un valore. Se `return` non seguito da alcuna espressione, allora restituisce `None`. Anche uscire dalla funzione senza un `return` restituisce `None`.
- L'istruzione `result.append(a)` chiama un *metodo* dell'oggetto-lista `result`. Un metodo è una funzione che «appartiene» all'oggetto e si può chiamare con la sintassi `obj.methodname` dove `obj` è l'oggetto (che potrebbe essere il risultato di un'espressione) e `methodname` è il nome del metodo che è stato definito nel tipo dell'oggetto. Tipi diversi definiscono metodi diversi. Metodi di tipi diversi possono avere lo stesso nome, senza che ciò produca ambiguità. Potete definire i vostri tipi e i vostri metodi, usando le *classi*: vedi *Classi*. Il metodo `append()` mostrato nell'esempio è definito per gli oggetti-lista: aggiunge un nuovo elemento in coda alla lista. In questo esempio è equivalente a `result = result + [a]`, ma più efficiente.

4.7 Altre cose sulla definizione delle funzioni

È possibile definire le funzioni con un numero variabile di parametri. Ci sono tre modi per fare questo, che si possono combinare tra loro.

4.7.1 Parametri con valori di default

Il modo più utile è specificare un valore di default per uno o più parametri. In questo modo è possibile chiamare la funzione con meno argomenti di quelli che la definizione prescriverebbe. Per esempio:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Questa funzione può essere chiamata in diversi modi:

- passando solo l'argomento necessario: `ask_ok('Do you really want to quit?')`

- passando anche uno degli argomenti opzionali: `ask_ok('OK to overwrite the file?', 2)`
- o passando tutti gli argomenti: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Questo esempio introduce anche la parola-chiave `in`, che testa se na sequenza contiene un certo valore oppure no.

I valori di default sono valutati al momento della definizione della funzione, nella tabella dei simboli che ospita la definizione. Quindi questo

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

restituirà `5`.

Attenzione: I valori di default sono valutati una volta sola. Questo fa differenza quando il default è un oggetto *mutabile* come una lista, un dizionario o un'istanza di molte altre classi. Per esempio, questa funzione accumula gli argomenti che le vengono passati in chiamate successive:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

Questo produrrà

```
[1]
[1, 2]
[1, 2, 3]
```

Se non volete che i valori di default siano condivisi tra chiamate successive, potete scrivere la funzione in questo modo:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2 Parametri *keyword*

Le funzioni possono essere chiamate anche passando argomenti keyword nella forma `kwarg=value`. Per esempio, questa funzione

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

prevede un parametro obbligatorio (`voltage`) e tre opzionali (`state`, `action` e `type`). Questa funzione può essere chiamata in molti modi diversi:

```
parrot(1000)                                          # 1 arg. posizionale
parrot(voltage=1000)                                  # 1 arg. keyword
parrot(voltage=1000000, action='VOOOOOM')             # 2 arg. keyword
parrot(action='VOOOOOM', voltage=1000000)             # 2 arg. keyword
parrot('a million', 'bereft of life', 'jump')         # 3 arg. posizionali
parrot('a thousand', state='pushing up the daisies')  # 1 posizionale, 1 keyword
```

Ma tutte queste chiamate invece non sono valide:

```
parrot()                     # manca un argomento richiesto
parrot(voltage=5.0, 'dead')  # argomento non-keyword dopo un keyword
parrot(110, voltage=220)     # doppio valore per lo stesso argomento
parrot(actor='John Cleese')  # argomento keyword sconosciuto
```

Nella chiamata di funzione, gli argomenti keyword devono seguire quelli posizionali. Ciascun argomento keyword passato deve corrispondere a uno accettato dalla funzione (`actor` non è un argomento valido per la funzione `parrot`), anche se l'ordine non è importante. Questo vale anche per gli argomenti non opzionali (`parrot(voltage=1000)` è una chiamata valida). Nessun argomento può ricevere un valore più di una volta. Ecco un esempio che non funziona perché viola questa restrizione:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for keyword argument 'a'
```

Quando compare un parametro finale nella forma `**name`, questo può ricevere un dizionario (vedi Tipi di mapping - dizionari) che contiene tutti gli argomenti keyword che non corrispondono a un parametro formale. Questo può essere unito a un parametro nella forma `*name` (che descriviamo nella prossima sezione), che riceve una *tupla* con tutti gli argomenti posizionali che eccedono quelli indicati nella lista dei parametri. `*name` deve essere elencato prima di `**name`. Per esempio, se definiamo una funzione in questo modo:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Potrebbe essere chiamata così:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

e naturalmente restituirebbe questo:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Si noti che l'ordine in cui sono scritti gli argomenti corrisponde sempre a quello in cui li abbiamo inseriti nella chiamata di funzione.

4.7.3 Parametri speciali

Gli argomenti possono essere passati a una funzione Python per *posizione*, oppure esplicitamente in modo *keyword*. Per ragioni di leggibilità e performance, è una buona idea regolamentare i modi in cui si possono passare gli argomenti, così che basti solo un'occhiata alla definizione della funzione per capire se i vari elementi sono passati per posizione, per *keyword* o in entrambi i modi.

Una definizione di funzione potrebbe essere così:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        posizionali o keyword   |
        |                                - solo keyword
         -- solo posizionali
```

dove / e * sono opzionali. Se vengono usati, questi simboli distinguono il tipo di parametro a seconda di come l'argomento può essere passato alla funzione: solo posizionale, posizione o keyword, solo keyword. Gli argomenti keyword sono detti anche «passati per nome».

Parametri posizionali o keyword

Se / e * non compaiono nella definizione della funzione, allora gli argomenti possono essere passati per posizione o per nome (keyword).

Parametri solo posizionali

Volendo specificare più in dettaglio, è possibile marcare certi parametri come *solo posizionali*. Per i parametri solo posizionali, l'ordine in cui sono elencati deve essere rispettato e non possono essere passati per nome. I parametri solo posizionali sono messi prima del segno /, che è usato per separarli logicamente dagli altri parametri. Se non c'è il segno / nella definizione della funzione, allora non ci sono parametri solo posizionali.

I parametri che vengono dopo il / possono essere *posizionali o keyword*, oppure *solo keyword*.

Parametri solo keyword

Per marcare i parametri come «solo keyword», indicando quindi che gli argomenti corrispondenti possono essere passati solo per nome, mettete un segno * nella lista dei parametri, subito prima del primo parametro «solo keyword».

Esempi

Si considerino queste definizioni di funzione, facendo attenzione ai segni / e *:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
```

(continua...)

(...segue)

```
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

La prima, `standard_arg`, ha la forma più comune e non pone alcuna restrizione al modo di chiamare la funzione. Gli argomenti possono essere passati indifferentemente per posizione o per nome:

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

La seconda funzione, `pos_only_arg`, può solo passare gli argomenti per posizione, come prescrive il segno `/` nella sua definizione:

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got an unexpected keyword argument 'arg'
```

La terza, `kwd_only_args`, permette solo di passare gli argomenti per nome, avendo il segno `*` nella definizione:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

L'ultima utilizza tutte e tre le convenzioni per la chiamata, nella stessa definizione:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got an unexpected keyword argument 'pos_only'
```

Infine, si consideri questa definizione di funzione, che presenta un potenziale conflitto tra il parametro posizionale `name` e un `**kwds` che potrebbe a sua volta contenere `name` tra le sue chiavi:

```
def foo(name, **kwds):
    return 'name' in kwds
```

Non c'è modo di chiamare la funzione e farle restituire `True`: infatti la chiave `'name'` sarà sempre collegata al primo argomento, mai a `**kwds`. Per esempio:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
```

Tuttavia, se usiamo il segno / per specificare i parametri solo posizionali, allora diventa possibile usare `name` come parametro posizionale e allo stesso tempo mettere `'name'` tra gli argomenti keyword:

```
def foo(name, /, **kwds):
    return 'name' in kwds
>>> foo(1, **{'name': 2})
True
```

In altre parole, i nomi dei parametri posizionali possono essere usati in `**kwds` senza pericolo di ambiguità.

Ricapitolando

Scegliere che tipo di parametri impiegare nella definizione di una funzione dipende dalla necessità:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

Qualche indicazione:

- Usate i parametri solo posizionali se volete che il nome dei parametri non sia disponibile per l'utente. Questo è utile quando i nomi non hanno un significato particolare, o se volete che l'ordine dei parametri sia obbligato, o se avete bisogno anche di qualche parametro keyword oltre a quelli posizionali.
- Usate i parametri solo keyword quando i nomi hanno un significato e la definizione della funzione è più chiara esplicitando i nomi, o se volete impedire che l'utente possa affidarsi all'ordine degli argomenti passati.
- Dal punto di vista dell'interfaccia, usate i parametri solo posizionali per prevenire che un cambiamento futuro nel nome del parametro modifichi la API della funzione.

4.7.4 Liste di parametri arbitrari

Infine, il metodo usato meno frequentemente consiste nello specificare che una funzione può essere chiamata passando un numero arbitrario di argomenti. Questi valori verranno conservati in una *tupla*. Prima dei parametri variabili, è possibile inserire degli altri parametri normali.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Di solito questi parametri «variadici» vengono per ultimi nella lista della definizione, perché catturano tutti i restanti argomenti che vengono passati alla funzione. Tutti i parametri formali che vengono dopo `*args` non possono che essere «solo keyword», ovvero argomenti che possono essere passati solo per nome.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.7.5 Spacchettare le liste di argomenti

Il caso opposto si verifica quando i valori da passare sono già contenuti in una lista o in una tupla, e devono essere «spacchettati» perché la chiamata di funzione richiede argomenti posizionali separati. Per esempio, la funzione predefinita `range()` prevede un parametro *start* e uno *stop*. Se non sono disponibili separatamente, potete scrivere la chiamata di funzione con l'operatore `*`, che spacchetta gli argomenti di una lista o una tupla:

```
>>> list(range(3, 6))    # chiamata normale con argomenti separati
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))  # chiamata con argomenti spacchettati da una lista
[3, 4, 5]
```

Analogamente, i dizionari possono essere spacchettati con l'operatore `**` per passare argomenti keyword:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin
↪' demised !
```

4.7.6 Funzioni lambda

È possibile creare delle piccole funzioni anonime con la parola-chiave `lambda`. Questa funzione restituisce la somma dei suoi due argomenti: `lambda a, b: a+b`. Le funzioni lambda possono essere usate dovunque si può usare una normale funzione. Dal punto di vista sintattico, sono limitate a una singola espressione. Dal punto di vista semantico, sono solo una scorciatoia al posto di una normale definizione di funzione. Come le funzioni interne ad altre funzioni, anche le lambda possono accedere a variabili definite nella funzione soprastante:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Questo esempio utilizza una lambda per restituire una funzione. Un altro possibile utilizzo è quando si vuole passare una piccola funzione come argomento di un'altra funzione:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.7 Stringhe di documentazione

Ci sono alcune convenzioni sul contenuto e la formattazione di una stringa di documentazione.

La prima riga dovrebbe essere un sintetico riepilogo dello scopo dell'oggetto documentato. Per brevità, non dovrebbe dichiarare esplicitamente il nome dell'oggetto o il suo tipo, dal momento che queste informazioni si possono ottenere in altro modo (a meno che il nome non sia un verbo che descrive l'azione della funzione - *questo naturalmente è più facile in Inglese, ndT*). La riga dovrebbe iniziare con la lettera maiuscola e finire con un punto.

Se la stringa ha più di una riga, la seconda dovrebbe essere vuota, in modo da separare visivamente il sommario dal resto della documentazione. Le righe successive dovrebbero contenere uno o più paragrafi che descrivono come si deve usare l'oggetto, i suoi *side-effect*, etc.

Il parser di Python non elimina lo spazio dei rientri da una stringa multi-riga: di conseguenza i *tool* che processano la documentazione dovranno compiere questa operazione, se lo desiderano. Per questo occorre utilizzare una convenzione: la prima riga non vuota *dopo* la riga iniziale determina lo spazio di rientro per tutto il resto della stringa. (Non possiamo usare la prima riga, perché di solito inizia con gli apici e quindi la stringa in sé non ha nessun rientro apparente.) Lo spazio «equivalente» a questo rientro deve essere quindi eliminato da tutte le righe della stringa. Non dovrebbero esserci righe con un rientro minore di questo, ma se ci sono allora tutto lo spazio iniziale dovrebbe essere tolto. Lo spazio «equivalente» dovrebbe essere calcolato dopo la conversione delle eventuali tabulazioni in spazi (di solito otto).

Ecco un esempio di docstring multi-riga:

```
>>> def my_function():
...     """Non fa nulla, ma lo documenta.
...
...     Davvero, non fa proprio nulla.
...     """
...     pass
...
>>> print(my_function.__doc__)
Non fa nulla, ma lo documenta.

    Davvero, non fa proprio nulla.
```

4.7.8 Annotazione di funzioni

Le annotazioni sono del tutto facoltative: si tratta di metadati informativi sui tipi utilizzati dalle funzioni (si vedano la **PEP 3107** e la **PEP 484** per ulteriori informazioni).

Le annotazioni sono conservate nell'attributo `__annotations__` della funzione, che è un dizionario, e non hanno effetto su nessun'altra parte della funzione. Le annotazioni dei parametri si indicano con un «due punti» dopo il nome del parametro, seguito da un'espressione che restituisce il valore dell'annotazione. Le annotazioni per i valori di ritorno si indicano con un `->` seguito da un'espressione, collocati tra la fine della lista dei parametri e il «due punti» che termina l'istruzione `def`. Nell'esempio che segue sono annotati un parametro posizionale, un parametro keyword e il valore di ritorno:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.8 Intermezzo: stile per il codice

Prima di iniziare a scrivere codice Python più lungo e complesso, è arrivato il momento di affrontare il tema dello «stile» del codice. Molti linguaggi possono essere scritti (o più precisamente, *formattati*) usando stili diversi; alcuni più leggibili di altri. È sempre una buona idea facilitare la lettura del vostro codice per gli altri, e per questo adottare uno stile chiaro aiuta moltissimo.

Nel mondo Python, la **PEP 8** si è affermata come la guida di stile usata in molti progetti: promuove uno stile molto leggibile e scorrevole all'occhio. Tutti i programmatori Python dovrebbero leggerla prima o poi; sintetizziamo qui i punti più importanti per voi:

- I rientri si fanno con 4 spazi, non con le tabulazioni.

  4 spazi sono un buon compromesso tra rientri più stretti (che permettono più livelli di annidamento) e più larghi (che sono più facili da leggere). Le tabulazioni fanno solo confusione ed è meglio non usarle.

- Le righe non devono superare i 79 caratteri.

  Questo è per aiutare gli utenti con schermi piccoli e rende possibile affiancare due file di codice su quelli più grandi.

- Lasciate una riga vuota per separare le funzioni e le classi, e anche i blocchi di codice più grandi all'interno delle funzioni.

- Quando possibile, mettete i commenti su una riga separata.

- Usate le docstring.

- Mettete uno spazio prima e dopo gli operatori e dopo la virgola, ma non accanto alle parentesi: `a = f(1, 2) + g(3, 4)`.

- Adottate dei nomi consistenti per le vostre classi e le funzioni; la convenzione è usare `UpperCamelCase` per le classi e `lowercase_with_underscores` per le funzioni e i metodi. Il nome del primo parametro di un metodo è sempre `self` (si veda *Introduzione alle classi* per ulteriori informazioni su classi e metodi).

- Non usate encoding esotici se il vostro codice deve essere usato in un contesto internazionale. UTF-8 (il default per Python), o anche il semplice ASCII, sono preferibili in ogni caso.

- Analogamente, non usate caratteri non-ASCII per gli identificatori se vi è anche la più remota possibilità che delle persone di nazionalità diversa leggeranno e lavoreranno sul codice.

CAPITOLO 5

Strutture dati

Questo capitolo descrive più in dettaglio argomenti già visti e aggiunge anche alcune cose nuove.

5.1 Un approfondimento sulle liste

Il tipo di dato «lista» ha diverse funzionalità ulteriori. Ecco un elenco di tutti i metodi disponibili per gli oggetti-lista:

`list.`**`append`**(*x*)
Aggiunge un elemento alla fine della lista. Equivale a `a[len(a):] = [x]`.

`list.`**`extend`**(*iterable*)
Estende una lista aggiungendovi tutti gli elementi di un oggetto iterabile. Equivale a `a[len(a):] = iterable`.

`list.`**`insert`**(*i*, *x*)
Inserisce un elemento alla posizione data. Il primo parametro è l'indice dell'elemento *prima del quale* sarà inserito il nostro, quindi `a.insert(0, x)` inserisce all'inizio della lista e `a.insert(len(a), x)` equivale a `a.append(x)`.

`list.`**`remove`**(*x*)
Rimuove il primo elemento della lista che ha valore *x*. Emette un `ValueError` se non esiste un elemento con questo valore.

`list.`**`pop`**([*i*])
Rimuove e *restituisce* l'elemento alla posizione specificata. Se non viene specificato un indice, `a.pop()` rimuove e restituisce l'ultimo elemento della lista. (Le parentesi quadre intorno alla *i* nell'elenco dei parametri non significano che dovreste usare quelle parentesi quando chiamate il metodo, ma indicano invece che il parametro è *opzionale*. Vedrete molto spesso questa notazione nella documentazione della libreria standard di Python.)

`list.`**`clear`**()
Rimuove tutti gli elementi della lista. Equivale a `del a[:]`.

`list.`**`index`**(*x*[, *start*[, *end*]])
Restituisce l'indice (partendo da zero) del primo elemento con valore *x*. Emette un `ValueError` se non esiste un elemento con quel valore.

I parametri opzionali *start* e *end* limitano la ricerca all'interno di una determinata sotto-lista, e sono interpretati come nella notazione per il sezionamento. L'indice restituito è però relativo all'intera lista, non alla sequenza che inizia con *start*.

`list.`**`count`**`(`*x*`)`
Restituisce il numero di volte che *x* appare nella lista.

`list.`**`sort`**`(*, `*key=None*`, `*reverse=False*`)`
Ordina sul posto gli elementi della lista. I parametri possono essere usati per aggiungere criteri all'ordinamento: si veda la funzione `sorted()` per il loro uso.

`list.`**`reverse`**`()`
Capovolge sul posto gli elementi della lista.

`list.`**`copy`**`()`
Restituisce una copia per indirizzo (*shallow copy*) della lista. Equivale a `a[:]`.

Un esempio che utilizza molti metodi delle liste:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Il prossimo "banana", dalla posizione 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Avrete notato che i metodi come `insert`, `remove` o `sort`, che modificano soltanto la lista, non hanno valore di ritorno – ovvero, restituiscono il `None` di default.[1] Questo è un principio di design che vale per tutte le strutture-dati mutabili in Python.

Un'altra cosa da osservare è che non tutti i dati possono essere ordinati o confrontati. Per esempio, `[None, 'hello', 10]` non può essere ordinato perché gli interi non possono essere confrontati con le stringhe e *None* non si può confrontare con altri tipi di dato. Inoltre, ci sono alcuni tipi che non hanno un ordinamento predefinito: per esempio, `3+4j < 5+7j` non è una comparazione valida.

5.1.1 Usare le liste come pile

È molto facile, grazie ai metodi che abbiamo visto, usare le liste come una pila (*stack*) ovvero come strutture in cui l'ultimo elemento aggiunto è il primo restituito (*last-in, first-out*). Per aggiungere un elemento in cima allo stack, usate `append()`. Per estrarre un elemento dalla cima dello stack, usate `pop()` senza un indice esplicito. Per esempio:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
```

(continua...)

[1] Altri linguaggi preferiscono restituire l'oggetto mutato, cosa che consente il concatenamento dei metodi, per esempio `d->insert("a")->remove("b")->sort();`.

(...segue)

```
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Usare le liste come code

È anche possibile usare le liste come code (*queue*), dove il primo elemento aggiunto è il primo restituito (*first-in, first-out*). Tuttavia le liste non sono strutture efficienti per questo scopo. Gli `append` e i `pop` alla fine della lista sono veloci, ma gli `insert` e i `pop` *all'inizio* sono lenti (perché tutti gli altri elementi devono slittare di una posizione).

Per implementare una coda, usate invece `collections.deque`, che è pensata appositamente per avere `append` e `pop` veloci da entrambi i lati. Per esempio:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arriva
>>> queue.append("Graham")          # Graham arriva
>>> queue.popleft()                 # Il primo ad arrivare parte
'Eric'
>>> queue.popleft()                 # Adesso parte il secondo arrivato
'John'
>>> queue                           # Il resto, in ordine di arrivo
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 List comprehension

Una *list comprehension* è un modo conciso di creare una lista. Accade di frequente di dover creare una lista dove ciascun elemento è il risultato di un'operazione condotta sugli elementi di un'altra lista o iterabile; oppure, di dover estrarre gli elementi che soddisfano una certa condizione.

Per esempio, vogliamo creare una lista di numeri quadrati, come questa:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Si noti che in questo modo creiamo e sovrascriviamo più volte una variabile `x` che resta in vita anche dopo che il ciclo è terminato. Possiamo eliminare questo *side effect* creando la lista in questo modo:

```
squares = list(map(lambda x: x**2, range(10)))
```

o, in modo equivalente:

```
squares = [x**2 for x in range(10)]
```

che è più sintetico e leggibile.

Una *list comprehension* è racchiusa tra parentesi quadre; contiene un'espressione, seguita da una clausola `for`, seguita da zero o più clausole `for` o `if`. Il risultato è una nuova lista costruita valutando l'espressione nel contesto delle clausole `for` e `if` che la seguono. Per esempio, questa *list comprehension* produce una combinazione degli elementi di due liste, se non sono uguali:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

è equivalente a:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Si noti che l'ordine del `for` e dello `if` è lo stesso in entrambe le soluzioni.

Se l'espressione è una tupla (come `(x, y)` nell'esempio precedente) deve essere messa tra parentesi.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # crea una nuova lista con i valori raddoppiati
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # fitra la lista togliendo i valori negativi
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # applica una funzione a tutti gli elementi
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # chiama un metodo su ciascun elemento
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # crea una lista di tiple del tipo (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # le tuple devono essere tra parentesi, o viene emesso un errore
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # "appiattisce" una lista con due 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Le *list comprehension* possono contenere espressioni complesse e funzioni dentro funzioni:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4 List comprehension annidate

L'espressione iniziale di una *list comprehension* può essere qualsiasi cosa, anche un'altra *list comprehension*.

Per esempio, questa è una matrice 3x4, implementata come una lista di tre liste di lunghezza 4:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

La seguente *list comprehension* annidata traspone righe e colonne:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Come abbiamo visto nel paragrafo precedente, la *list comprehension* annidata è valutata nel contesto del `for` che la segue; il nostro esempio equivale quindi a:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

che a sua volta è la stessa cosa di:

```
>>> transposed = []
>>> for i in range(4):
...     # le 3 righe seguenti equivalgono alla list comp. annidata
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Nella pratica di tutti i giorni, è preferibile usare le funzioni predefinite alle istruzioni di controllo di flusso troppo complicate. La funzione `zip()` è molto adatta al nostro specifico scenario:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Si veda *Spacchettare le liste di argomenti* per l'uso dell'asterisco in questa chiamata di funzione.

5.2 L'istruzione `del`

L'istruzione `del` consente di rimuovere un elemento da una lista, data la sua posizione anziché il valore. È differente dal metodo `pop()`, che restituisce il valore dell'elemento rimosso. L'istruzione `del` può anche essere usata per rimuovere una sezione della lista, o svuotare l'intera lista (come abbiamo già fatto assegnando una lista vuota alla sezione). Per esempio:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
```

(continua...)

(...segue)

```
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` può anche eliminare una variabile:

```
>>> del a
```

Adesso riferirsi ad `a` produce un errore, almeno finché non le viene assegnato un nuovo valore. Vedremo in seguito altri possibili usi di `del`.

5.3 Tuple e sequenze

Abbiamo visto che le liste e le stringhe hanno molte proprietà in comune, come le operazioni di indicizzazione e sezionamento. In effetti sono due esempi del tipo di dato *sequenza* (si veda Sequenze - liste, tuple, range). Dal momento che Python è un linguaggio in evoluzione, altri tipi di sequenza potrebbero essere aggiunti in futuro. Un altro tipo di sequenza predefinita è la *tupla*.

Una tupla è una serie di valori separati da virgola, per esempio:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Le tuple possono essere annidate:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Le tuple sono immutabili:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # ma possono contenere oggetti mutabili:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Come si può vedere, le tuple in output sono sempre scritte con le parentesi, in modo che le tuple annidate siano leggibili facilmente. Possono essere scritte in input con o senza parentesi, anche se molto spesso le parentesi sono comunque necessarie (se la tupla fa parte di un'espressione più grande). Non è possibile assegnare a un elemento della tupla: tuttavia è possibile creare tuple che contengono oggetti mutabili, come una lista.

Anche se le tuple possono sembrare simili alle liste, sono usate in contesti diversi e per scopi diversi. Le tuple sono immutabili e di solito ospitano una collezione di elementi eterogenei, a cui si può accedere tramite «spacchettamento» (vedi oltre) o indici, o anche attributi, nel caso di una `namedtuples`. Le liste sono mutabili e di solito ospitano elementi omogenei, a cui si accede iterando sulla lista.

Le tuple che hanno zero o un elemento pongono un problema di costruzione: la sintassi prevede due piccole stranezze per risolvere questi casi. Le tuple vuote si creano con una coppia di parentesi, senza nulla dentro. Le tuple con un solo elemento hanno una virgola finale (non è sufficiente mettere il valore tra parentesi per creare una tupla). Non è bello da vedere, ma funziona. Per esempio:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- notare la virgola finale
>>> len(empty)
```

(continua...)

(...segue)

```
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

L'assegnazione `t = 12345, 54321, 'hello!'` è un esempio di *impacchettamento* di tupla: i valori``12345``, `54321` e `'hello!'` sono impacchettati insieme nella tupla. L'inverso è anche possibile:

```
>>> x, y, z = t
```

Questo si chiama, prevedibilmente, *spacchettamento* di sequenza, e funziona con tutti i tipi di sequenza, a destra del segno di uguaglianza. Lo spacchettamento richiede che il numero delle variabili sul lato sinistro sia uguale al numero di elementi della sequenza sul lato destro. Si noti che l'assegnamento multiplo è in realtà una combinazione delle due operazioni di impacchettamento e spacchettamento.

5.4 Set

Python ha un tipo di dato per i *set*. Un set è una collezione non ordinata senza elementi duplicati. Tra gli utilizzi più frequenti vi sono i test di appartenenza e l'eliminazione dei duplicati. I set supportano anche le operazioni matematiche di unione, intersezione, differenza e differenza simmetrica.

Per creare un set si può usare la funzione `set()` o le parentesi graffe. Si noti che per creare un set vuoto occorre usare `set()`, non `{}`: questo infatti crea un *dizionario* vuoto, come vedremo nella prossima sezione.

Ecco una breve dimostrazione:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # i duplicati sono stati rimossi
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # test di appartenza veloce
True
>>> 'crabgrass' in basket
False

>>> # Dimostra le operazioni sui set con i caratteri di due parole
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # caratteri unici in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # in a ma non in b
{'r', 'd', 'b'}
>>> a | b                              # in a o b o entrambi
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # sia in a sia in b
{'a', 'c'}
>>> a ^ b                              # in a o b, ma non in entrambi
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Analogamente alle *list comprehensions*, esistono le *set comprehensions*:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5 Dizionari

Un altro utile tipo predefinito in Python è il *dizionario* (si veda Tipi di mapping - dizionari). I dizionari sono anche chiamati «array associativi» o «memorie associative» in altri linguaggi. A differenza delle sequenze che sono indicizzate con intervalli numerici, i dizionari sono indicizzati con *chiavi*; le chiavi possono essere di qualsiasi tipo immutabile: stringhe e numeri sono sempre adatti come chiavi. Le tuple possono essere usate come chiavi, se contengono solo stringhe, numeri o altre tuple; se una tupla contiene qualsiasi altro oggetto mutabile, direttamente o indirettamente, allora non può fungere da chiave per un dizionario. Non potete usare le liste come chiavi, dal momento che queste possono essere modificate sul posto con l'assegnamento a un indice, il sezionamento o metodi come `append()` e `extend()`.

Conviene pensare a un dizionario come a una collezione di coppie *chiave: valore*, con il requisito che le chiavi devono essere univoche all'interno del dizionario. Una coppia di parentesi graffe crea un dizionario vuoto: `{}`. Per inizializzare il dizionario è possibile inserire nelle parentesi delle coppie *chiave: valore*; questo è anche il modo in cui i dizionari sono scritti in output.

Le operazioni principali con i dizionari sono: conservare un valore accoppiandolo a una chiave; ed estrarre il valore data la chiave. È inoltre possibile cancellare una coppia *chiave: valore* con `del`. Se si accoppia un valore a una chiave già in uso, il vecchio valore viene sovrascritto. Estrarre un valore con una chiave inesistente produce un errore.

Usare `list(d)` su un dizionario restituisce una lista di tutte le chiavi usate nel dizionario, in ordine di inserimento (se le preferite ordinate, potete invece usare `sorted(d)`). Per sapere se una chiave è presente in un dizionario, usate la parola-chiave `in`.

Ecco un esempio di utilizzo di un dizionario:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

La funzione `dict()` costruisce un dizionario da una sequenza di coppie *chiave, valore*:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Inoltre, è possibile usare le *dict comprehension* per creare dizionari da espressioni arbitrarie che restituiscono coppie *chiave: valore*:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Quando le chiavi sono delle stringhe, è più semplice passare a `dict()` degli argomenti keyword:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6 Tecniche di iterazione

Quando occorre iterare su un dizionario, le chiavi e i valori corrispondenti si possono estrarre contemporaneamente con il metodo `items()`:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

Quando si itera su una sequenza, l'indice e il valore corrispondente si possono estrarre contemporaneamente con la funzione `enumerate()`:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Per iterare su due o più sequenze contemporaneamente, queste possono essere accoppiate con la funzione `zip()`:

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Per iterare su una sequenza in ordine inverso, si scrive l'iterazione in avanti e su questa si chiama poi la funzione `reversed()`:

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Per iterare su una sequenza in modo ordinato, usate la funzione `sorted()` che restituisce una nuova lista ordinata, lasciando inalterato l'originale:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

Usate la funzione `set()` su una sequenza per eliminare i duplicati. Combinare `sorted()` con `set()` è un modo idiomatico per iterare sugli elementi unici di una sequenza in ordine alfabetico:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

Talvolta si cerca di modificare la lista mentre ci si sta iterando sopra; è spesso più semplice creare invece una nuova lista:

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7 Un approfondimento sulle condizioni

Le condizioni usate nelle istruzioni `while` e `if` possono contenere qualsiasi operatore, non solo di confronto.

Gli operatori di confronto `in` e `not in` controllano se un valore esiste o meno in una sequenza. Gli operatori `is` e `is not` ci dicono se due oggetti sono effettivamente lo stesso oggetto: questo importa solo per gli oggetti mutabili come le liste. Tutti gli operatori di confronto hanno la stessa priorità, che è più bassa di quella di tutti gli altri operatori numerici.

I confronti possono essere collegati. Per esempio, `a < b == c` testa se `a` è minore di `b` e inoltre se `b` è uguale a `c`.

I confronti possono essere combinati usando gli operatori booleani `and` e `or`; il risultato di un confronto, o di qualsiasi altra espressione booleana, si può negare con `not`. Questi operatori hanno una priorità più bassa degli operatori di confronto; tra di loro, `not` ha la priorità più alta e `or` la più bassa, così che `A and not B or C` equivale a `(A and (not B)) or C`. Come sempre, si possono usare le parentesi per esprimere la priorità desiderata.

Gli operatori booleani `and` e `or` sono detti «operatori corto-circuito»: i loro argomenti sono valutati da sinistra a destra, ma la valutazione si ferma non appena l'esito è chiaro. Per esempio, se `A` e `C` sono «veri» ma `B` è «falso», allora `A and B and C` si ferma prima di valutare l'espressione `C`. Quando vengono usati per restituire un valore, e non come booleani, gli operatori corto-circuito restituiscono l'ultimo argomento valutato.

È possibile assegnare a una variabile il risultato di un confronto o di un'altra espressione booleana. Per esempio,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Si noti che in Python, a differenza di C, un assegnamento dentro un'espressione può essere fatto solo esplicitamente con il walrus operator `:=`. Questo evita una serie di problemi comuni che si incontrano programmando in C: scrivere per sbaglio = in un'espressione, quando si intende ==.

5.8 Confronto di sequenze e altri tipi

In genere è possibile confrontare un oggetto-sequenza con una sequenza dello stesso tipo. Il confronto è fatto in ordine *lessicografico*: prima sono confrontati i primi due elementi tra loro; se sono diversi questo determina l'esito del confronto; se sono uguali, si confrontano i secondi elementi e così via, fino a quando una delle due sequenze termina. Se due elementi da confrontare sono essi stessi delle sequenze, viene effettuato un confronto lessicografico tra questi, ricorsivamente. Se tutti gli elementi sono uguali fra loro, le sequenze sono considerate uguali. Se una sequenza è una sotto-sequenza iniziale di un'altra, è la sequenza più breve a risultare la minore nel confronto. L'ordine lessicografico per le stringhe usa i *code point* Unicode per confrontare i singoli caratteri. Ecco alcuni esempi di confronto tra sequenze dello stesso tipo:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Si noti che confrontare oggetti di tipo diverso con < o > è possibile, purché gli oggetti abbiano un metodo di confronto adeguato. Per esempio, i diversi tipi numerici sono confrontati in base al loro valore, quindi 0 è uguale a 0.0 e così via. In assenza di un metodo di confronto, l'interprete non fornisce un ordinamento arbitrario, ma emette invece un'eccezione `TypeError`.

CAPITOLO 6

Moduli

Se chiudete l'interprete di Python e poi vi rientrate, non ritroverete le definizioni che avevate impostato (funzioni e variabili). Di conseguenza, se volete scrivere un programma più lungo, vi conviene usare un editor di testo per preparare le istruzioni per l'interprete, e invocarlo poi con il file risultante come input. In questo modo avete creato uno *script*. Quando poi il vostro programma diventa più lungo, potreste volerlo dividere in diversi file più maneggevoli. Potreste anche voler usare in diversi programmi le stesse funzioni utili che avete scritto, senza bisogno di copiarle tutte le volte.

A questo scopo, in Python potete mettere le definizioni in un file e usarle poi in uno script o nella sessione interattiva dell'interprete. Un file di questo tipo è un *modulo*; le definizioni di un modulo possono essere *importate* in altri moduli o nel modulo *principale* (ovvero, l'insieme delle variabili a cui avete accesso da uno script quando è eseguito, o dalla modalità interattiva).

Un modulo è un file che contiene definizioni e istruzioni Python. Il nome del file è quello del modulo più il suffisso `.py`. Dentro il modulo, il nome è disponibile come valore della variabile globale `__name__` (una stringa). Per esempio, usate il vostro editor preferito per creare un file dal nome `fibo.py` nella directory corrente, che contiene questo:

```
# modulo per i numeri di Finonacci

def fib(n):    # scrive i numeri di Fibonacci fino a n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # restituisce i numeri di Fibonacci fino a n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Adesso entrate nell'interprete interattivo dei comandi e importate questo modulo così:

```
>>> import fibo
```

Questa istruzione non inserisce i nomi delle funzioni definite in `fibo` direttamente nella tabella dei simboli corrente; invece, vi inserisce il nome del modulo `fibo`. Usando il nome del modulo potete accedere alle funzioni che contiene:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Se intendete usare spesso una funzione, potete assegnarle un nome locale:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Approfondimenti sui moduli

Un modulo può contenere istruzioni eseguibili oltre a definizioni di funzioni. Queste istruzioni devono essere intese come un modo di inizializzare il modulo. Sono eseguite solo la *prima volta* che il nome del modulo è incontrato in una istruzione `import`.[1] (Sono anche eseguite se il modulo è eseguito come script.)

Ciascun modulo ha una sua tabella dei simboli, che vale come tabella globale per tutte le funzioni che vi sono definite. Quindi l'autore del modulo può usare delle variabili globali senza preoccuparsi di conflitti accidentali con le variabili globali dell'utente del modulo. D'altra parte, se siete sicuri di quello che fate, potete accedere alle variabili globali del modulo con la stessa notazione che usate per riferirvi alle sue funzioni, ovvero `modname.itemname`.

I moduli possono importare altri moduli. È consuetudine, ma non obbligatorio, mettere tutte le istruzioni `import` all'inizio del modulo (o dello script). I nomi dei moduli importati sono inseriti nella tabella dei simboli globale del modulo importatore.

Esiste una variante dell'istruzione `import` che consente di importare direttamente i nomi contenuti in un modulo nella tabella dei simboli del modulo importatore. Per esempio:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

In questo modo però il nome del modulo, i cui nomi interni sono importati, non è importato esso stesso (quindi, in questo esempio, `fibo` non è definito).

C'è poi una variante che consente di importare *tutti* i nomi definiti in un modulo:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

In questo modo vengono importati tutti i nomi del modulo, eccetto quelli che iniziano con un *underscore* (_). In genere i programmatori Python non usano questa tecnica, dal momento che introduce un numero sconosciuto di nomi nell'interprete, eventualmente sovrascrivendo nomi che erano già stati definiti.

Si noti che in generale importare `*` da un modulo o da un package è considerato cattiva pratica, perché spesso rende il codice più difficile da leggere. Tuttavia va bene usare questa tecnica nelle sessioni interattive, per risparmiare battute nei nomi da inserire.

Se il nome del modulo è seguito dalla parola-chiave `as`, allora il nome che segue `as` è collegato direttamente al modulo importato.

[1] In effetti anche le definizioni di funzione sono delle «istruzioni» che vengono eseguite; l'esecuzione della definizione di una funzione a livello del modulo inserisce il nome della funzione nella tabella dei simboli globale.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Questo modo di importare il modulo è del tutto equivalente a `import fibo`, con l'unica differenza che adesso il modulo sarà disponibile con il nome `fib`.

Si può anche usare in combinazione con la parola-chiave `from`, con effetti analoghi:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Nota: Per ragioni di efficienza, ogni modulo è importato solo una volta nella sessione dell'interprete. Di conseguenza, se nel frattempo modificate il vostro modulo, dovete riavviare l'interprete. In alternativa, se si tratta di un modulo che state testando interattivamente, potete usare la funzione `importlib.reload()`, ovvero scrivere `import importlib; importlib.reload(modulename)`.

6.1.1 Eseguire moduli come script

Quando eseguite un modulo Python con

```
python fibo.py <arguments>
```

il codice del modulo verrà eseguito, proprio come se lo aveste importato, ma la variabile `__name__` sarà impostata a `"__main__"`. Ciò vuol dire che, se inserite alla fine del modulo questa clausola:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

potete rendere questo file utilizzabile sia come script sia come modulo importabile, perché il codice incluso nella clausola, che parsa la riga di comando, verrà eseguito solo quando il modulo è eseguito come il file «main»:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

Se il modulo è importato, il codice non verrà eseguito:

```
>>> import fibo
>>>
```

Questa tecnica è usata spesso, sia per fornire una comoda interfaccia utente per un modulo, sia per eseguire dei test (facendo in modo che, quando si esegue il modulo come script, si esegua una suite di test).

6.1.2 Il percorso di ricerca dei moduli

Quando importiamo un modulo di nome `spam`, l'interprete per prima cosa cerca tra i moduli predefiniti se ne esiste uno con quel nome. Se non lo trova, cerca un file `spam.py` in una lista di directory contenuta nella variabile `sys.path`. Questa, a sua volta, viene inizializzata con le seguenti *path*:

- La directory che contiene lo script importatore (o la directory corrente se questo non è specificato).
- La variabile d'ambiente `PYTHONPATH` (se impostata, contiene una lista di directory, con la stessa sintassi della variabile `PATH`).
- Un default che dipende dall'installazione di Python.

Nota: Nei file system che supportano i symlink, la directory che contiene lo script importatore è calcolata dopo aver seguito i symlink. In altre parole, la directory che contiene il symlink *non* è aggiunta al percorso di ricerca dei moduli.

Dopo che è stata inizializzata, è possibile modificare `sys.path` dall'interno di un programma Python. La directory che contiene lo script in esecuzione è collocata all'inizio del lista dei percorsi da cercare, davanti alla directory della libreria standard. Ciò vuol dire che i moduli locali, se hanno lo stesso nome, verranno importati al posto di quelli della libreria standard. In genere questo è un errore, a meno che non sia fatto intenzionalmente. Si veda la sezione *Moduli della libreria standard* per maggiori informazioni.

6.1.3 File «compilati»

Per velocizzare il caricamento dei moduli, Python conserva nella directory di cache `__pycache__` una versione compilata di ciascun modulo, con il nome `module.version.pyc`, dove *version* specifica il formato del file compilato: di solito è il numero di versione di Python. Per esempio, in CPyhton 3.3 la versione compilata del modulo `spam.py` si chiamerebbe `__pycache__/spam.cpython-33.pyc`. Questa convenzione permette la coesistenza di moduli compilati da diverse versioni di Python.

Python confronta la data di ultima modifica del modulo con la sua versione compilata, e ricompila all'occorrenza. Questo processo è completamente automatico. Inoltre, i moduli compilati sono indipendenti dalla piattaforma, così che lo stesso modulo possa essere condiviso su sistemi diversi, con diverse architetture.

Python non controlla la cache in due circostanze. In primo luogo, quando un modulo è caricato direttamente dalla riga di comando, Python ricompila sempre il modulo senza conservarlo nella cache. In secondo luogo, non controlla la cache se non c'è anche il modulo originale. Per ottenere una distribuzione senza sorgenti (solo compilata), oltre a togliere il modulo originale, il modulo compilato deve essere collocato nella directory dei file originali.

Alcuni consigli per gli esperti:

- Potete usare le opzioni `-O` o `-OO` della riga di comando Python per ridurre le dimensioni del modulo compilato. La `-O` rimuove le istruzioni *assert*, mentre `-OO` rimuove sia gli *assert* sia le docstring. Dal momento che alcuni programmi potrebbero averne bisogno, usate queste opzioni solo se sapete che cosa state facendo. I moduli «ottimizzati» hanno un contrassegno `opt-` e di solito sono più piccoli. Future versioni di Python potrebbero cambiare gli effetti dell'ottimizzazione.
- Un programma non è più veloce se usa i file `.pyc` invece dei normali `.py`. L'unica differenza è la velocità di *caricamento* del modulo.
- Il modulo `compileall` della libreria standard può compilare tutti i moduli di una directory.
- Si veda la **PEP 3147** per ulteriori dettagli su questi procedimenti, incluso un diagramma di flusso dei vari passaggi.

6.2 Moduli della libreria standard

Python è distribuito con una libreria standard di moduli, documentata in un una sezione separata, la Guida di Riferimento della Libreria Standard. Alcuni moduli sono pre-caricati nell'interprete: questi forniscono delle operazioni che non fanno parte del linguaggio, ma sono comunque predefinite, sia per ragioni di efficienza, sia per dare accesso alle primitive del sistema operativo sottostante. La composizione di questi moduli dipende dalla configurazione, che a sua volta dipende dalla piattaforma. Per esempio, `winreg` è solo disponibile su Windows. Un modulo meritevole di attenzione particolare è `sys`, sempre disponibile. Le variabili `sys.ps1` e `sys.ps2` definiscono le stringhe usate per il prompt primario e secondario:

```
>>> import sys
>>> sys.ps1
'>>> '
```

(continua...)

(...segue)

```
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

Queste variabili sono disponibili solo se l'interprete è in modalità interattiva.

La variabile `sys.path` è una lista di stringhe che determina il percorso di ricerca dei moduli da importare. È inizializzata con delle path contenute nella variabile d'ambiente `PYTHONPATH`, oppure da default predefiniti se questa non è impostata. Potete modificare `sys.path` con le normali tecniche di manipolazione delle liste:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 La funzione `dir()`

La funzione predefinita `dir()` ci dice quali nomi sono definiti in un modulo. Restituisce una lista ordinata di stringhe:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

Senza argomenti, `dir()` elenca i nomi disponibili attualmente:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Si noti che nell'elenco compaiono tutti i tipi di nomi: variabili, moduli, funzioni e così via.

`dir()` non elenca però i nomi delle funzioni e delle variabili predefinite. Se volete un lista di questi, sono definiti nel modulo `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 'zip']
```

6.4 Package

I package sono un modo di strutturare il *namespace* di un modulo Python usando la «notazione col punto». Per esempio, il nome `A.B` indica un sotto-modulo `B` all'interno di un package `A`. Proprio come i moduli permettono a diversi autori di non doversi preoccupare dei nomi *di variabile* usati in altri moduli, così i package permettono agli autori di package con molti moduli, come NumPy o Pillow, di non doversi preoccupare dei nomi *dei moduli* usati da altri.

Immaginate di voler costruire una collezione di moduli (un package) per la gestione di suoni e file sonori. Ci sono diversi formati di file sonori (di solito sono riconoscibili dalle estensioni, per esempio `.wav`, `.aiff`, `.au`): quindi avrete bisogno di creare e mantenere una raccolta crescente di moduli per la conversione tra i vari formati. Ci sono poi molte diverse operazioni che si possono fare sui suoni (mixare, aggiungere eco, equalizzare, creare un effetto stereo artificiale): quindi dovrete scrivere una serie interminabile di moduli che implementano queste operazioni. Ecco una possibile struttura per il vostro package (espressa in forma di gerarchia del file system):

```
sound/                          package top-level
      __init__.py               inizializzazione del package
      formats/                  sotto-package per le conversioni
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  sotto-package per gli effetti
```

(continua...)

(...segue)

```
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  sotto-package per i filtri
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Quando importate il package, Python cerca nei percorsi della `sys.path` la directory del package.

I file `__init__.py` sono necessari perché Python consideri effettivamente come un package la directory che contiene i moduli. Questo è per evitare che directory con un nome comune, per esempio `string`, possano nascondere inavvertitamente dei nomi di moduli che vengono dopo nell'ordine dei percorsi di ricerca. Nel caso più semplice, `__init__.py` può essere lasciato vuoto, ma è anche possibile fargli eseguire del codice di inizializzazione o impostare la variabile `__all__`, come vedremo tra poco.

Gli utenti del package possono importare dei singoli moduli al suo interno, per esempio:

```
import sound.effects.echo
```

Questo carica il modulo `sound.effects.echo`. Occorre riferirsi a questo con il nome completo.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Un modo alternativo per importare il modulo è questo:

```
from sound.effects import echo
```

Anche in questo modo carichiamo il modulo `echo`, ma lo rendiamo disponibile senza il prefisso del nome del package. Può essere quindi usato così:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Un altro modo ancora è importare direttamente la funzione o la variabile richiesta:

```
from sound.effects.echo import echofilter
```

Ancora una volta, questo carica il modulo `echo`, rendendo però direttamente disponibile la sua funzione `echofilter()`:

```
echofilter(input, output, delay=0.7, atten=4)
```

Notate che quando si usa la modalità `from package import item`, allora *item* può essere sia il nome di un modulo (o sotto-package) del package, sia qualche altro nome definito nel package, come una funzione, una classe o una variabile. L'istruzione `import` per prima cosa controlla se *item* è definito nel package; se no, assume che si tratti di un modulo e cerca di caricarlo. Se l'operazione fallisce, viene emessa un'eccezione `ImportError`.

Al contrario, quando usate la sintassi `import item.subitem.subsubitem`, ogni elemento eccetto l'ultimo *deve* essere un package; l'ultimo può essere un package o un modulo, ma *non* può essere una classe o una funzione o una variabile definita nell'elemento precedente.

6.4.1 Importare * da un package

Che cosa succede quando scriviamo `from sound.effects import *`? Idealmente, ci si potrebbe aspettare che questa istruzione provochi una scansione nel file system, trovi i moduli presenti nel package e li importi tutti in un colpo solo. Questo però potrebbe richiedere molto tempo e importare un sotto-modulo potrebbe causare *side-effect* indesiderati, che dovrebbero verificarsi solo quando il modulo è importato direttamente.

L'unica soluzione è che l'autore del package fornisca un indice esplicito del suo contenuto. L'istruzione `import` segue questa convenzione: se il modulo `__init__.py` di un package definisce una lista col nome `__all__`, allora considera questa come l'indice dei moduli che dovrebbero essere importati da un `from package import *`. È compito dell'autore aggiornare la lista quando rilascia una nuova versione del package. Un autore potrebbe anche non fornire la lista, se decide che non può essere utile importare «*» dal suo package. Per esempio, il file `sound/effects/__init__.py` potrebbe contenere questo codice:

```
__all__ = ["echo", "surround", "reverse"]
```

In questo modo, `from sound.effects import *` importerebbe i tre moduli indicati del package `sound`.

Se `__all__` non è definito, allora l'istruzione `from sound.effects import *` *non* importa comunque tutti i moduli del package `sound.effects` nel *namespace* corrente. Si limita a garantire che il package `sound.effects` sia stato effettivamente importato eventualmente eseguendo il codice trovato nel file `__init__.py`) e quindi importa tutti i nomi definiti nel package: questo comprende tutti i nomi definiti (e i moduli esplicitamente importati) nel `__init__.py`. Include anche tutti i moduli del package che sono stati esplicitamente importati in precedenza. Si consideri questo codice:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In questo esempio, i moduli `echo` e `surround` sono importati nel *namespace* corrente perché sono definiti nel package `sound.effects` al momento di eseguire l'istruzione `from...import` (funziona allo stesso modo quando la variabile `__all__` è definita).

Anche se alcuni moduli sono progettati per esportare solo alcuni nomi, secondo certi criteri, quando importate con `import *`, questa è comunque considerata una cattiva pratica nel codice «di produzione».

Ricordate che non c'è niente di male a importare `from package import specific_submodule`. In effetti questo è il modo raccomandato, a meno che il modulo importatore non stia anche importando un altro modulo con lo stesso nome da un altro package.

6.4.2 Riferimenti intra-package

Quando i package contengono a loro volta dei sub-package (come nel caso del nostro esempio `sound`), potete usare gli import *assoluti* per riferirvi a moduli di package «cugini». Per esempio, se il modulo `sound.filters.vocoder` ha bisogno di usare il modulo `echo` nel package `sound.effects`, può importarlo con `from sound.effects import echo`.

Potete anche usare gli import *relativi*, negli import del tipo `from module import name`. Gli import relativi usano una notazione con punti iniziali per indicare il package corrente e genitore interessati dall'import. Dal modulo `surround`, per esempio, potreste importare:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Si noti che gli import relativi si basano sul nome del modulo importatore. Siccome il nome del modulo principale è sempre `"__main__"`, i moduli intesi per essere usati come script (come il modulo principale di un'applicazione Python) devono sempre usare gli import assoluti.

6.4.3 Package in directory multiple

I package hanno un attributo speciale `__path__`. Questa variabile è una lista, inizializzata con il nome della directory dove risiede il file `__init__.py` del package, prima che il codice di questo sia eseguito. Potete modificare il contenuto della variabile: così facendo modificate i percorsi di ricerca dei moduli e dei sub-package del package, per tutte le successive importazioni.

Anche se è una funzionalità raramente necessaria, può essere usata per estendere l'insieme dei moduli disponibili in un package.

CAPITOLO 7

Input e Output

Ci sono diversi modi per presentare l'output di un programma: i dati possono essere «stampati» in un formato leggibile per l'utente, o conservati in un file per uso futuro. In questo capitolo prenderemo in esame alcune possibilità.

7.1 Formattazione gradevole dell'output

Finora abbiamo visto due modi di scrivere un valore: le espressioni e la funzione `print()`. (Un terzo modo è quello di usare il metodo `write()` degli oggetti-file: lo standard output può essere raggiunto con `sys.stdout`. Si veda la documentazione della libreria standard per maggiori informazioni.)

Spesso si desidera avere un maggior controllo sulla formattazione dell'output, piuttosto di limitarsi a scrivere valori separati da spazi. Ci sono diversi modi per formattare l'output.

- Usare le *formattazioni di stringa*, mettendo una `f` o `F` prima degli apici iniziali: in questo modo è possibile includere nella stringa un'espressione tra parentesi graffe, che può fare riferimento a variabili o valori.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- Il metodo delle stringhe `str.format()` richiede più lavoro manuale. Potete usare le parentesi graffe per marcare il posto dove una variabile sarà sostituita e potete specificare delle indicazioni dettagliate di formattazione, ma dovete anche indicare quali informazioni formattare.

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

- Infine, potete gestire la stringa «manualmente», usando gli operatori di concatenamento e sezionamento per creare qualunque layout vi venga in mente. Il tipo di dato stringa ha alcuni metodi utili in questo senso, che «mettono in colonna» il testo allineandolo alla spaziatura voluta.

Quando non vi serve un output raffinato, ma vi basta dare un'occhiata ad alcune variabili a scopo di debug, potete convertire qualsiasi valore a una stringa con le funzioni `repr()` o `str()`.

La funzione `str()` restituisce una rappresentazione leggibile del valore, mentre `repr()` genera una rappresentazione che può essere consumata dall'interprete, o che forza un `SyntaxError` se non esiste una sintassi equivalente. Se un oggetto non ha una rappresentazione leggibile, `str()` restituisce lo stesso valore di `repr()`. Per molti valori, come i numeri o costrutti come le liste e i dizionari, entrambe le funzioni producono la stessa rappresentazione. Le stringhe, d'altra parte, hanno due rappresentazioni diverse.

Alcuni esempi:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # repr() aggiunge apici e backslash:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # possiamo passare a repr() qualsiasi oggetto come argomento:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

Il modulo `string` contiene una classe `Template` che presenta ancora un altro metodo per integrare valori dentro una stringa, usando dei segnaposto come `$x` e rimpiazzandoli con valori da un dizionario; offre però meno controllo sulla formattazione.

7.1.1 Stringhe formattate

Le stringhe formattate, chiamate anche *f-string*, hanno il prefisso `f` o `F` e consentono di inserire delle espressioni Python nella stringa, racchiudendole dentro parentesi graffe.

L'espressione può essere seguita da una sintassi che specifica la formattazione da applicare: questo permette un maggiore controllo su come il valore verrà formattato. Nell'esempio che segue arrotondiamo pi greco a tre cifre decimali:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

Per espandere un «campo» a un numero minimo di caratteri, basta mettere un numero intero dopo il `':'`. Questo è utile per creare incolonnamenti:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

Altri modificatori servono a convertire il valore prima di formattarlo. `'!a'` converte in `ascii()`, `'!s'` applica la funzione `str()`, e `'!r'` applica `repr()`:

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

Informazioni complete su come specificare la formattazione si trovano nella guida di riferimento nella sezione Linguaggio di specifica della formattazione.

7.1.2 Il metodo format() delle stringhe

L'uso più semplice del metodo `str.format()` è qualcosa del genere:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

Le parentesi graffe e i caratteri che contengono (i «campi da formattare») vengono sostituiti dai valori passati al metodo `str.format()`. All'interno delle parentesi, è possibile usare un numero per riferirsi alla posizione degli argomenti passati a `str.format()`.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

Se a `str.format()` vengono passati degli argomenti keyword, è possibile usare il nome dell'argomento per riferirsi al rispettivo valore:

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

Argomenti posizionali e keyword possono essere usati insieme:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                                   other='Georg'))
The story of Bill, Manfred, and Georg.
```

Quando avete una stringa da formattare molto lunga e volete dividerla, può far comodo riferirsi alle variabili da formattare per nome, non per posizione. Ciò può essere fatto semplicemente passando un dizionario e usando la notazione con le parentesi quadre `'[]'` per accedere alle sue chiavi:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Un'alternativa è passare la tabella come argomento keyword, con la notazione "**".

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Questo metodo è particolarmente utile in combinazione con la funzione predefinita `vars()`, che restituisce un dizionario che contiene tutte le variabili locali.

Per esempio, questo produce delle colonne bene allineate che mostrano i numeri interi, i loro quadrati e cubi:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
```

(continua...)

(...segue)

```
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Per una discussione completa della formattazione con `str.format()`, si veda Sintassi della formattazione delle stringhe.

7.1.3 Formattazione manuale delle stringhe

Ecco lo stesso esempio dei quadrati e dei cubi, formattato manualmente:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # notare l'uso di 'end' nella riga precedente
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Si noti che il singolo spazio aggiunto tra le colonne è dovuto al modo in cui funziona `print()`, che aggiunge sempre uno spazio tra i suoi argomenti.

Il metodo `str.rjust()` giustifica a destra una stringa rispetto a un campo di determinata lunghezza, aggiungendo gli spazi necessari a sinistra. Esistono metodi analoghi `str.ljust()` e `str.center()`. Questi metodi non producono output, si limitano a restituire una nuova stringa. Se la stringa da giustificare è troppo lunga rispetto al campo, non la troncano ma si limitano a restituirla inalterata: questo scompaginerà il vostro output, ma è senz'altro meglio dell'alternativa, ovvero alterare il dato. (Se davvero preferite troncare, potete fare un sezionamento, per esempio `x.ljust(n)[:n]`.)

Un altro metodo, `str.zfill()`, completa una stringa numerica con degli «0» a sinistra. Inoltre capisce quando trova il segno positivo o negativo:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4 Vecchio metodo di formattazione

L'operatore `%` (modulo) può anche essere usato per la formattazione delle stringhe. Data la sintassi `'stringa' % valori`, le occorrenze di `%` in `'stringa'` sono rimpiazzate da zero o più elementi di `valori`. Questa operazione viene chiamata comunemente «interpolazione di stringa». Per esempio:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

Per ulteriori informazioni, si veda la sezione Formattazione di stringa in stile printf.

7.2 Leggere e scrivere files

La funzione `open()` restituisce un oggetto-file e si usa in genere con due argomenti: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
```

Il primo parametro è una stringa che indica il nome del file. Il secondo è una stringa che descrive il modo in cui il file verrà usato. Il *modo* può essere `'r'` quando il file verrà solo letto, `'w'` per le operazioni di sola scrittura (un eventuale file pre-esistente verrà cancellato), e `'a'` che aggiunge alla fine del file tutti i dati che vengono scritti. `'r+'` consente sia la lettura sia la scrittura. Passare un *modo* è opzionale: se l'argomento è omesso, il file è aperto in modalità `'r'` di default.

In genere i file sono aperti in modalità testuale (*text mode*), il che significa leggere e scrivere delle *stringhe* di testo con un encoding specificato. Se l'encoding non è indicato, il default dipende dalla piattaforma (si veda la documentazione della funzione `open()`). Se si aggiunge una `'b'` all'argomento *mode*, il file è aperto in modalità binaria (*binary mode*): i dati sono letti e scritti in forma di *bytes*. Tutti i file che non contengono testo dovrebbero essere aperti con questa modalità.

In modalità testuale, Python, in lettura, converte a `\n` gli «a-capo» caratteristici della piattaforma (`\n` su Unix, `\r\n` su Windows). In scrittura, tutti gli `\n` sono ri-convertiti secondo la convenzione della piattaforma. Queste modifiche dietro le quinte vanno bene per i file di testo, ma corrompono i dati binari di un file `JPEG` o `EXE`. Occorre prestare attenzione ad aprire questi file solo in modalità binaria.

È buona pratica usare l'istruzione `with` quando si deve gestire un oggetto-file. In questo modo il vantaggio è che il file verrà sempre chiuso al termine delle operazioni, anche se nel frattempo dovesse essere emessa un'eccezione. Usare `with` è anche più sintetico del corrispondente blocco `try-finally`:

```
>>> with open('workfile') as f:
...     read_data = f.read()

>>> # In effetti il file è stato chiuso automaticamente:
>>> f.closed
True
```

Se non usate `with`, allora dovreste chiamare `f.close()` per chiudere il file e liberare immediatamente le risorse di sistema collegate.

Avvertimento: Chiamare `f.write()` senza usare `with` o chiamare `f.close()` **potrebbe** comportare che gli argomenti di `f.write()` non siano scritti completamente nel file su disco, anche se il programma dovesse terminare senza problemi.

Una volta chiuso il file, sia con un'istruzione `with` sia chiamando `f.close()`, ogni tentativo di usarlo di nuovo fallirà automaticamente:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1 Metodi degli oggetti-file

In ciascuno degli esempi seguenti assumiamo che un oggetto-file `f` sia stato appena creato.

Per leggere il contenuto di un file, chiamate `f.read(size)`, che legge una determinata quantità di dati e li restituisce in forma di stringa (in modalità testuale) o di oggetti byte (in modalità binaria). *Size* è un parametro numerico opzionale. Se *size* è omesso, o è negativo, l'intero contenuto del file verrà letto e restituito: può essere un problema se il file occupa il doppio della memoria disponibile. Altrimenti, al massimo un numero *size* di caratteri (in modalità testuale) o di byte (in modalità binaria) verranno letti e restituiti. Se è stata raggiunta la fine del file, `f.read()` restituisce una stringa vuota (`''`).

```
>>> f.read()
'Questo è tutto il file.\n'
>>> f.read()
''
```

`f.readline()` legge una singola riga del file. Lascia il carattere di «a-capo» finale (`\n`) nella stringa restituita, omettendolo solo alla fine se il file non termina con una nuova riga. In questo modo il valore di ritorno non è ambiguo: se `f.readline()` restituisce una stringa vuota, vuol dire che è stata raggiunta la fine del file; invece, una riga vuota nel file è restituita come `'\n'`, ovvero una stringa che contiene solo il carattere di «a-capo».

```
>>> f.readline()
'Questa è la prima riga del file.\n'
>>> f.readline()
'Seconda riga del file.\n'
>>> f.readline()
''
```

Per leggere le righe di un file, è possibile iterare sull'oggetto-file. Questo metodo è efficiente per il consumo di memoria, veloce e porta a scrivere codice più semplice:

```
>>> for line in f:
...     print(line, end='')
...
Questa è la prima riga del file.
Seconda riga del file.
```

Se volete mettere tutte le righe di un file in una lista, potete usare `list(f)` o `f.readlines()`.

`f.write(string)` scrive il contenuto di una *stringa* in un file e restituisce il numero dei caratteri che sono stati scritti:

```
>>> f.write('This is a test\n')
15
```

Altri tipi di oggetti devono essere convertiti prima di scriverli, o in una stringa (in modalità testuale) o in bytes (in modalità binaria):

```
>>> value = ('the answer', 42)
>>> s = str(value)  # converte la tupla in una stringa
>>> f.write(s)
18
```

`f.tell()` restituisce un numero intero che rappresenta la posizione corrente nell'oggetto-file, come numero di byte a partire dall'inizio del file, se questo è aperto in modalità binaria; se è aperto in modalità testuale, il numero non indica tuttavia il numero di caratteri.

Per cambiare la posizione nell'oggetto-file, usate `f.seek(offset, whence)`. La nuova posizione è calcolata aggiungendo *offset* a un punto di riferimento indicato dall'argomento *whence*. Passando 0 a *whence*, la misura viene fatta dall'inizio del file; 1 indica la posizione attuale; 2 usa la fine del file come punto di riferimento. Se l'argomento *whence* viene omesso, il suo default è 0, ovvero l'inizio del file è preso come riferimento:

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # vai al sesto byte del file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # vai al terzultimo byte prima della fine
13
>>> f.read(1)
b'd'
```

In modalità testuale (per i file aperti senza una `b` passata all'argomento *mode*) è permesso di riferirsi solo all'inizio del file, con la sola eccezione di un `seek(0, 2)` che si riferisce esattamente alla fine del file; inoltre gli unici *offset* validi sono quelli restituiti da `f.tell()`, oppure 0. Tutti gli altri possibili *offset* producono risultati non definiti.

Gli oggetti-file dispongono di altri metodi di uso meno frequente, come `isatty()` o `truncate()`; rimandiamo alla documentazione della libreria standard per informazioni complete su questi oggetti.

7.2.2 Persistenza di dati strutturati con `json`

Le stringhe si possono leggere e scrivere facilmente nei file. I numeri richiedono un piccolo sforzo aggiuntivo, dal momento che il metodo `read()` restituisce solo una stringa, che quindi deve poi essere passata per la conversione a funzioni come `int()`, che riceve stringhe come `'123'` e restituisce il corrispondente valore numerico 123. Tuttavia, quando volete «salvare» strutture-dati più complesse come liste annidate e dizionari, diventa complicato fare a mano il *parsing* e la serializzazione.

Invece di costringervi a scrivere e correggere continuamente del codice per persistere dati complessi nei file, Python vi consente di usare un formato di interscambio popolare, chiamato JSON (JavaScript Object Notation). Il modulo `json` della libreria standard converte gerarchie di dati Python nelle loro rappresentazioni in formato stringa: questo processo si chiama serializzazione (*serializing*). Ricostruire i dati a partire dalla loro rappresentazione si chiama deserializzazione (*deserializing*). Nell'intervallo tra i due processi, la stringa che rappresenta l'oggetto può essere salvata in un file o altro tipo di struttura, o inviata a un computer remoto tramite una connessione di rete.

Nota: Il formato JSON è molto usato dalle applicazioni moderne per lo scambio dei dati. Molti programmatori lo conoscono già, e questo lo rende una buona scelta per l'interoperabilità.

Dato un oggetto `x`, potete ricavarne la rappresentazione JSON con una sola riga di codice:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Una variante della funzione `dumps()`, chiamata `dump()`, serializza l'oggetto e lo scrive in un file di testo. Quindi, se `f` è un file aperto in modalità di scrittura, potete fare questo:

```
json.dump(x, f)
```

Per ricostruire l'oggetto, se `f` è un file aperto in modalità di lettura, basta fare:

```
x = json.load(f)
```

Questa tecnica di serializzazione è semplice e riesce a gestire liste e dizionari; tuttavia, serializzare istanze di classi arbitrarie in JSON richiede qualche sforzo ulteriore. Si veda la documentazione del modulo `json` per ulteriori spiegazioni.

Vedi anche:

il modulo `pickle`

Al contrario di *JSON*, il protocollo di *pickle* permette la serializzazione di oggetti Python complessi. Di conseguenza, è specifico di Python e non può essere usato per comunicare con applicazioni scritte in altri linguaggi. Inoltre è intrinsecamente non sicuro: deserializzare un *pickle* che proviene da una fonte non affidabile può provocare l'esecuzione di codice arbitrario, se i dati sono stati confezionati da un attaccante abile.

CAPITOLO 8

Errori ed eccezioni

Fino ad ora non abbiamo parlato in modo specifico dei messaggi di errore ma, se avete provato gli esempi, sicuramente ne avrete visto qualcuno. Ci sono almeno due tipi di errore: gli *errori di sintassi* e le *eccezioni*.

8.1 Errori di sintassi

Gli errori di sintassi, noti anche come errori di parsing, sono forse l'inciampo più comune quando state ancora imparando Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

Il parser riporta la riga sbagliata e mostra una piccola «freccia» che indica il primo punto in cui l'errore è stato rilevato. L'errore è causato (o almeno rilevato) dall'elemento che *precede* la freccia: nell'esempio qui sopra, l'errore è rilevato nella funzione `print()`, perché mancano i «due punti» (`':'`) prima. Anche il nome del file e la riga sono riportati, in modo da sapere dove guardare, se l'input proviene da uno script.

8.2 Eccezioni

Anche quando un'istruzione o un'espressione sono corretti dal punto di vista sintattico, possono provocare un errore quando sono *eseguiti*. Gli errori rilevati durante l'esecuzione si chiamano eccezioni e non sono sempre fatali: imparerete presto come gestirle nel vostro programma Python. Molte eccezioni, comunque, non sono gestite dal programma e restituiscono messaggi di errore come questi:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

(continua...)

(...segue)

```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

L'ultima riga del messaggio d'errore ci dice che cosa è successo. Gli oggetti-eccezioni possono avere diversi tipi, e la prima parte del messaggio riporta il tipo: negli esempi qui sopra, `ZeroDivisionError`, `NameError` e `TypeError`. La stringa mostrata come tipo è il nome dell'eccezione predefinita incontrata. Questo succede per tutte le eccezioni predefinite, ma potrebbe essere diverso per le eccezioni definite dall'utente (anche se è comunque una convenzione utile). I nomi delle eccezioni standard sono identificatori predefiniti, ma non parole-chiave riservate.

Il resto della riga fornisce dettagli che dipendono dal tipo dell'eccezione e da che cosa l'ha causata.

Tutto ciò che precede il messaggio d'errore mostra il *contesto* in cui è avvenuta l'eccezione, nella forma di un *traceback* dello stack. In generale, il traceback elenca le righe di codice coinvolte nel problema; tuttavia non visualizza le righe lette dallo standard input.

La sezione della documentazione Eccezioni predefinite elenca tutte le eccezioni predefinite e il loro significato.

8.3 Gestire le eccezioni

I programmi possono gestire delle eccezioni specifiche. Nell'esempio che segue, chiediamo un input all'utente, fin quando non inserisce un numero valido; in ogni caso l'utente può interrompere il programma (con `Control-C` o in qualunque modo consentito dal sistema operativo). Si noti che un'interruzione generata dall'utente provoca un'eccezione `KeyboardInterrupt`:

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

L'istruzione `try` funziona in questo modo:

- Per prima cosa, viene eseguito il blocco *try*, ovvero le istruzioni tra il `try` e lo `except`.
- Se nessuna eccezione viene incontrata, il blocco *except* non viene eseguito e l'esecuzione dell'istruzione `try` termina così.
- Se durante l'esecuzione del blocco *try* viene incontrata un'eccezione, le eventuali istruzioni rimanenti del blocco vengono saltate. Quindi, se il tipo dell'eccezione coincide con quella nominata dopo la parola-chiave `except`, allora viene eseguito il blocco *except*. Quindi l'esecuzione prosegue normalmente con ciò che segue l'istruzione `try`.
- Se viene incontrata un'eccezione che non corrisponde a quella prevista nel blocco *except*, allora l'eccezione è passata ad eventuali altre istruzioni `try` annidate di livello superiore; se nessun gestore viene trovato, l'eccezione è *non gestita*: a questo punto l'esecuzione del programma si arresta con il messaggio di errore visto sopra.

L'istruzione `try` può avere più di una clausola *except*, per specificare gestori per diverse eccezioni: non più di un gestore per volta può essere eseguito. Il gestore affronta solo l'eccezione che si è verificata nella clausola *try* corrispondente, non quelle che eventualmente si verificano in altri gestori della stessa istruzione `try`. Una clausola *except* può gestire più eccezioni, specificandole come una tupla (con parentesi obbligatorie), per esempio:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Un'eccezione specificata in una clausola `except` è compatibile con l'eccezione che si verifica se sono istanze della stessa classe, o se quest'ultima è una sotto-classe della prima (ma non il contrario: se l'eccezione specificata è una sotto-classe di quella che si verifica, non sono compatibili). Per esempio, il codice che segue produrrà nell'ordine B, C, D:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Si noti che, mettendo le clausole *except* in ordine inverso (con `except B` al primo posto), l'output prodotto sarebbe B, B, B: viene eseguita la prima clausola *except* in grado di gestire l'eccezione.

È possibile omettere il nome dell'eccezione nell'ultima clausola *except*, in modo che serva come risorsa estrema. Questa strategia va però usata con cautela, dal momento che è facile mascherare in questo modo un errore di programmazione. È anche possibile scrivere un messaggio di errore e quindi ri-emettere l'eccezione, in modo che il codice chiamante possa eventualmente gestirla:

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

L'istruzione `try ... except` prevede una clausola opzionale *else* che, se presente, deve venire dopo tutte le clausole *except*. Vi si può inserire del codice che deve essere eseguito solo se la clausola *try* non emette alcuna eccezione. Per esempio:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Usare `else` è preferibile a inserire del codice in più nel `try`, perché in questo modo si evita di intercettare accidentalmente delle eccezioni emesse dal codice che non si intendeva proteggere nella clausola *try*.

Quando si verifica un'eccezione, questa può avere un valore associato, detto anche *argomento* dell'eccezione. La presenza e il tipo di questo argomento dipende dall'eccezione.

La clausola *except* può specificare una variabile dopo il nome dell'eccezione. La variabile è legata all'istanza dell'eccezione, e i suoi argomenti sono conservati in `instance.args`. Per comodità, l'istanza dell'eccezione definisce un metodo `__str__()` tale per cui gli argomenti possono essere scritti direttamente, senza doversi riferire a `.args`. È possibile anche istanziare l'eccezione prima di emetterla, in modo da aggiungere gli attributi desiderati:

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # l'istanza dell'eccezione
...     print(inst.args)     # gli argomenti conservati in .args
...     print(inst)          # __str__ scrive direttamente gli argomenti
...                          # ma può essere sovrascritto nelle sottoclassi
...     x, y = inst.args     # spacchettiamo gli argomenti
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Se un'eccezione ha degli argomenti, questi sono scritti nell'ultima parte («detail») del messaggio di errore causato dall'eccezione non gestita.

Un gestore può intercettare non solo le eccezioni che accadono direttamente nel blocco *try*, ma anche quelle emesse da funzioni chiamate (anche indirettamente) dal codice del *try*. Per esempio:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4 Emettere eccezioni

L'istruzione `raise` permette di forzare l'emissione di una specifica eccezione. Per esempio:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

L'unico argomento di `raise` è il nome dell'eccezione da emettere. Questa deve essere o un'istanza o una classe-eccezione (ovvero, una classe che deriva da `Exception`). Se viene passata una classe, questa sarà implicitamente istanziata chiamando il costruttore senza argomenti:

```
raise ValueError  # scorciatoia per 'raise ValueError()'
```

Se avete bisogno di rilevare soltanto un'eccezione, ma non intendete davvero gestirla, potete usare una forma più semplice di `raise` che permette di rilanciare l'eccezione:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5 Concatenamento di eccezioni

L'istruzione `raise` accetta un'opzione `from` che consente di concatenare due eccezioni. Per esempio:

```
# exc deve essere l'istanza di una eccezione, o None
raise RuntimeError from exc
```

Questo è utile per trasformare un'eccezione in un'altra. Per esempio:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

Il concatenamento delle eccezioni avviene automaticamente quando un'eccezione viene emessa da dentro una clausola `except` oppure `finally`. L'idioma `from None` disabilita il concatenamento:

```
>>> try:
...     open('database.sqlite')
... except IOError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

Per ulteriori informazioni sul meccanismo del concatenamento, si veda la sezione sulle Eccezioni predefinite.

8.6 Eccezioni personalizzate

Un programma può creare le sue eccezioni interne, scrivendo una nuova classe-eccezione (si veda la sezione *Classi* per ulteriori informazioni sulle classi in Python). Le eccezioni dovrebbero tipicamente derivare dalla classe `Exception`, direttamente o indirettamente.

Le classi delle eccezioni possono fare tutto ciò che farebbe una classe normale, ma di solito si preferisce mantenerle semplici, spesso fornendole solo di qualche attributo che aiuta a capire il problema quando viene intercettato dai gestori dell'eccezione. Quando si scrive un modulo che può incontrare diversi casi di errore, una pratica comune è scrivere una classe-madre per le eccezioni di quel modulo, e delle sotto-classi che descrivono eccezioni specifiche per le diverse condizioni di errore:

```
class Error(Exception):
    """Classe-madre per le eccezioni di questo modulo."""
    pass

class InputError(Error):
    """Eccezione emessa in caso di errore nell'input.

    Attributi:
        expression -- espressione di input che ha generato l'errore
        message -- spiegazione dell'errore
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Emessa quando un'operazione provoca una transizione di stato
    non permessa.

    Attributi:
        previous -- stato iniziale della transizione
        next -- stato finale che si cercava di ottenere
        message -- motivo per cui la transizione non è ammessa
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

In genere si fa in modo che le eccezioni personalizzate abbiano nomi che finiscono in «Error», analogamente ai nomi delle eccezioni standard.

Molti moduli della libreria standard definiscono eccezioni proprie, per segnalare errori che possono verificarsi nelle funzioni che contengono. Per altre informazioni sulle classi, si veda la sezione *Classi*.

8.7 Definire azioni di chiusura

L'istruzione `try` prevede un'altra clausola opzionale che permette di definire azioni di chiusura e pulizia che devono essere eseguite in qualsiasi circostanza. Per esempio:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
```

(continua...)

(...segue)

```
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Se è presente una clausola `finally`, questa verrà eseguita come ultima cosa, prima che il keyword:*try* sia completato. Il blocco `finally` viene eseguito in ogni caso, indipendentemente dal fatto che il codice nel `try` emetta un'eccezione o no. Approfondiamo nel dettaglio alcuni casi complessi:

- Se si incontra un'eccezione durante l'esecuzione del blocco `try`, l'eccezione potrebbe essere gestita da un blocco `except`. Se l'eccezione non è gestita, allora viene rilanciata dopo l'esecuzione del blocco `finally`.
- L'eccezione potrebbe accadere durante l'esecuzione di una clausola `except` o `else`. Anche in questo caso l'eccezione è rilanciata dopo l'esecuzione del blocco `finally`.
- Se il codice del blocco `try` raggiunge un'istruzione `break continue` o `return`, allora la clausola `finally` sarà eseguita immediatamente prima di queste istruzioni.
- Se entrambi i blocchi `try` e `finally` comprendono un'istruzione `return`, allora il valore restituito sarà quello del `finally`, non quello del `try`.

Per esempio:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

Un esempio più complesso:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("divisione per zero!")
...     else:
...         print("il risultato è", result)
...     finally:
...         print("eseguo la clausola finally")
...
>>> divide(2, 1)
il risultato è 2.0
eseguo la clausola finally
>>> divide(2, 0)
divisione per zero!
eseguo la clausola finally
>>> divide("2", "1")
eseguo la clausola finally
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Come si può vedere, il blocco `finally` è eseguito in ogni caso. Il `TypeError` emesso quando si cerca di dividere due stringhe non è gestito dalla clausola `except` e quindi viene rilanciato, una volta che il `finally` è stato eseguito.

In uno scenario concreto, la clausola `finally` è utile per rilasciare le risorse esterne (come una connessione a un file o a un database), indipendentemente dal fatto che l'utilizzo sia andato a buon fine.

8.8 Azioni di chiusura predefinite

Alcuni oggetti definiscono delle operazioni di chiusura e pulizia, quando non sono più necessari, indipendentemente dal fatto che l'utilizzo dell'oggetto sia andato a buon fine oppure no. Si consideri il seguente esempio, che cerca di aprire un file e scriverne il contenuto sullo schermo:

```
for line in open("myfile.txt"):
    print(line, end="")
```

Il problema qui è che lasciamo il file aperto per un tempo indeterminato, dopo che questa parte del codice è stata eseguita. Questo non è grave per un semplice script, ma diventa un problema per le applicazioni più grandi. L'istruzione `with` consente di usare oggetti come i file in modo tale da assicurarsi sempre le opportune operazioni di chiusura e pulizia.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

Dopo che l'istruzione è stata eseguita, il file *f* viene sempre chiuso, anche nel caso in cui, processandolo, si dovesse incontrare una condizione di errore. Se un oggetto definisce, come i file, delle operazioni di chiusura predefinite, questo viene indicato nella sua documentazione.

CAPITOLO 9

Classi

Le classi servono a unire insieme dati e funzionalità. Creare una classe equivale a creare un nuovo *tipo*, a partire dal quale possono essere create nuove *istanze* (oggetti). Ciascuna istanza può avere degli attributi suoi propri, che ne *mantengono* lo stato. Le istanze possono anche avere dei metodi (definiti nella classe) che ne *modificano* lo stato.

A differenza di altri linguaggi di programmazione, il meccanismo delle classi in Python aggiunge pochissima sintassi e semantica: è un misto delle classi che si possono trovare in C++ e in Modula-3. Le classi Python forniscono tutti gli strumenti standard della programmazione a oggetti (OOP): il meccanismo dell'ereditarietà consente di avere più di una classe-base, una sotto-classe può sovrascrivere i metodi della sua classe-madre, e un metodo può invocare il metodo della classe-madre con lo stesso nome. Gli oggetti possono contenere quanti dati si desidera. Proprio come i moduli, le classi partecipano della natura dinamica di Python: sono create a *runtime* e possono essere modificate ulteriormente dopo la loro creazione.

Usando la terminologia di C++, di solito i membri di una classe (inclusi i dati) sono *pubblici* (a eccezione di quanto spiegato in *Variabili private*) e tutte le funzioni interne sono *virtuali*. Come in Modula-3, non ci sono scorciatoie per referenziare i membri di un oggetto dall'interno dei suoi metodi: un metodo è chiamato con un primo argomento esplicito che rappresenta l'oggetto stesso, che è fornito implicitamente dalla chiamata. Le classi sono oggetti esse stesse, come in Smalltalk: questo permette la semantica per l'importazione e la rinomina. A differenza di C++ e Modula-3, i tipi predefiniti possono essere usati come classi-base che l'utente può estendere. Inoltre, come in C++, la maggior parte degli operatori predefiniti che godono di sintassi speciale (gli operatori aritmetici, l'indicizzazione, etc.) possono essere ridefiniti per le istanze delle classi.

(Dal momento che non esiste una terminologia universale per le classi, utilizziamo occasionalmente il lessico di C++ e Smalltalk. Sarebbe preferibile usare il lessico di Modula-3 che, rispetto a C++, è più simile a Python per quel che riguarda gli oggetti: ma sospettiamo che pochi lettori abbiano familiarità con questo linguaggio.)

9.1 A proposito di nomi e oggetti

Gli oggetti hanno una identità univoca e più di un nome (in più di uno *scope*) può essere collegato allo stesso oggetto: è ciò che in altri linguaggi si chiama *aliasing*. Questa caratteristica, a prima vista, non è sempre apprezzata in Python e può tranquillamente essere ignorata quando si gestiscono oggetti di tipo immutabile (numeri, stringhe, tuple). Tuttavia, lo *aliasing* può avere effetti imprevisti sulla semantica del codice Python che maneggia oggetti mutabili come liste, dizionari e la maggior parte degli altri tipi. Di solito questi effetti sono sfruttati positivamente dal programma, dal momento che lo *aliasing* è in qualche modo simile ai puntatori. Per esempio, passare un oggetto è più economico, dal momento che l'implementazione sottostante deve solo passare un puntatore; e se una

funzione modifica un oggetto passato come argomento, il codice chiamante vedrà le modifiche: questo elimina la necessità di un doppio meccanismo di passaggio degli argomenti, come avviene in Pascal.

9.2 *Scope* e *namespace* in Python

Prima di affrontare le classi, dobbiamo parlare delle regole degli *scope* in Python.[1] Le classi manipolano i *namespace* in modo sottile ed è necessario sapere come funzionano *scope* e *namespace* per capire che cosa succede davvero. D'altra parte, la comprensione di questi argomenti è utile in qualsiasi applicazione avanzata di Python.

Iniziamo con alcune definizioni.

Un *namespace* è una mappatura che collega nomi a oggetti. Di solito i *namespace* sono attualmente implementati come dizionari (per ragioni di performance), ma dall'esterno questo non si vede e potrebbe cambiare in futuro. Esempi di *namespace* sono l'insieme dei nomi predefiniti (che comprende funzioni come `abs()` e i nomi delle eccezioni predefinite); i nomi globali di un modulo; i nomi locali a una chiamata di funzione. In un certo senso, l'insieme degli attributi di un oggetto è anch'esso un *namespace*. La cosa fondamentale da capire sui *namespace* è che non c'è assolutamente nessuna relazione tra i nomi di diversi *namespace*. Per esempio, due moduli diversi potrebbero definire entrambi una funzione `maximize` senza nessuna confusione: gli utenti dei due moduli devono riferirsi a questa funzione premettendo il nome del modulo.

A proposito: usiamo il termine *attributo* per qualsiasi nome che segue un punto: per esempio, nell'espressione `z.real`, `real` è un attributo dell'oggetto `z`. Tecnicamente, un riferimento a un nome in un modulo è un riferimento a un *attributo* di quel modulo: nell'espressione `modname.funcname`, `modname` è un oggetto-modulo e `funcname` è un suo attributo. In questo caso c'è una corrispondenza ovvia tra gli attributi del modulo e i nomi globali definiti nel modulo: condividono lo stesso *namespace*![2]

Gli attributi possono essere di sola lettura o scrivibili: a questi ultimi è possibile assegnare dei valori. Gli attributi dei moduli sono scrivibili: potete scrivere `modname.the_answer = 42`. Gli attributi scrivibili sono anche eliminabili con l'istruzione `del`. Per esempio, `del modname.the_answer` rimuoverà l'attributo `the_answer` dall'oggetto `modname`.

I *namespace* sono creati in momenti diversi e hanno cicli di vita diversi. Il *namespace* che contiene i nomi predefiniti è creato dall'interprete Python all'avvio e non viene mai distrutto. Il *namespace* globale di un modulo è creato al momento della lettura delle definizioni del modulo; anche questi di solito durano fin quando l'interprete è in esecuzione. Le istruzioni eseguite all'invocazione dell'interprete, sia quelle che sono lette da un file *script* sia quelle eseguite in modalità interattiva, sono considerate parte di un modulo chiamato `__main__`, e quindi hanno un loro *namespace* globale. (I nomi predefiniti, in effetti, vivono anch'essi in un modulo: questo si chiama `builtins`.)

Il *namespace* locale di una funzione viene creato al momento di chiamare la funzione e distrutto quando la funzione restituisce il suo risultato o emette un'eccezione che non viene gestita all'interno della funzione. (In realtà, «dimenticato» è un termine più appropriato per descrivere quello che accade.) Naturalmente le invocazioni ricorsive hanno ciascuna il proprio *namespace*.

Uno *scope* è una regione di codice Python, all'interno del programma, dove il *namespace* è direttamente accessibile. Con «direttamente accessibile» intendiamo che un riferimento *non qualificato* (senza ricorrere alla notazione col punto) a un nome riesce effettivamente a raggiungere quel nome nel *namespace*.

Anche se gli *scope* sono determinati in modo statico, sono usati in modo dinamico. In qualsiasi momento durante l'esecuzione del programma esistono tre o quattro *scope* annidati, i cui *namespace* sono direttamente accessibili:

- lo *scope* più interno, dove un nome è cercato per prima cosa, contiene i nomi locali;
- gli *scope* di ogni eventuale funzione di ordine superiore, che sono ricercati dal più prossimo al più lontano, contengono nomi non-locali ma anche non-globali;

[1] ndT: in questa traduzione rifiutiamo con decisione la consueta, orribile restituzione di *scope* (area in cui una variabile è visibile: dal Greco *skopein*, osservare) con l'Italiano «scopo» (fine, proposito: dal Latino *scopus*, bersaglio). Lasciamo inalterato *scope* e, per contiguità, non traduciamo neppure *namespace* (che di solito è reso in modo più accettabile con «spazio dei nomi»).

[2] Tranne che per una cosa. Gli oggetti-modulo hanno un attributo di sola lettura nascosto, che si chiama `__dict__`: è il dizionario usato per implementare il *namespace* del modulo. Il nome `__dict__` è un attributo del modulo, ma non un suo nome globale. Naturalmente questa è un'eccezione nell'implementazione astratta dei *namespace* e dovrebbe essere usata solo da strumenti come i *debugger* post-mortem.

- il penultimo *scope* più lontano contiene i nomi globali del modulo corrente;
- lo *scope* più generale (dove il nome è cercato per ultimo) è il *namespace* che contiene i nomi predefiniti.

Se un nome è dichiarato *global*, allora tutti i riferimenti a questo puntano direttamente allo *scope* intermedio che contiene i nomi globali del moduli. Per ri-collegare variabili che si trovano fuori dallo *scope* più interno, potete usare l'istruzione `nonlocal`. Se dichiarata *nonlocal*, una variabile è di sola lettura: tentare di scrivere in questa variabile non farà altro che creare un *nuova* variabile nello *scope* locale, lasciando immutata la variabile esterna con il medesimo nome.

In genere lo *scope* locale «vede» i nomi locali al codice della funzione corrente. Al di fuori di una funzione, lo *scope* locale vede lo stesso *namespace* dello *scope* globale: ovvero, il *namespace* del modulo.

È importante capire che gli *scope* sono determinati «dal testo del codice». Lo *scope* globale di una funzione definita in un modulo è il *namespace* di quel modulo: non importa da dove è chiamata la funzione, o con quale alias. Ma d'altro canto, la *ricerca* di un nome avviene dinamicamente, a *runtime*. È anche vero che l'architettura del linguaggio evolve verso la risoluzione statica dei nomi, a *compile time*, quindi non dovreste fare affidamento sulla risoluzione dinamica dei nomi (e in effetti, le variabili locali sono già determinate in modo statico).

Una peculiarità di Python è che, in assenza di istruzioni `global` o `nonlocal`, gli assegnamenti alle variabili sono sempre indirizzati allo *scope* più interno. Gli assegnamenti non copiano i dati, collegano semplicemente i nomi agli oggetti. Lo stesso vale per le eliminazioni: l'istruzione `del x` rimuove il collegamento di `x` dal *namespace* ricercato dallo *scope* locale. In effetti, tutte le operazioni che introducono nomi nuovi utilizzano lo *scope* locale: in particolare, le istruzioni `import` e le definizioni di funzione collegano il nome del modulo o della funzione allo *scope* locale.

L'istruzione `global` può essere usata per indicare che una particolare variabile vive nello *scope* globale e dovrebbe essere ri-collegata lì; l'istruzione `nonlocal` indica che una particolare variabile vive nel *namespace* di ordine superiore e dovrebbe essere ri-collegata lì.

9.2.1 Esempi di *scope* e *namespace*

Questo esempio dimostra come riferirsi ai diversi *scope* e *namespace* e come `global` e `nonlocal` influiscono sul collegamento delle variabili.

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("Dopo un'assegnazione locale:", spam)
    do_nonlocal()
    print("Dopo un'assegnazione 'nonlocal':", spam)
    do_global()
    print("Dopo un'assegnazione 'global':", spam)

scope_test()
print("Nello scope globale:", spam)
```

L'output di questo esempio è:

```
Dopo un'assegnazione locale: test spam
Dopo un'assegnazione 'nonlocal': nonlocal spam
```

(continua...)

(...segue)

```
Dopo un'assegnazione 'global': nonlocal spam
Nello scope globale: global spam
```

Si noti che l'assegnazione *locale* (che è il comportamento di default) non cambia il collegamento della variabile *spam* della funzione *scope_test*. D'altra parte l'assegnamento `nonlocal` cambia il collegamento dello *spam* di *test_spam*, e l'assegnamento `global` cambia il collegamento dello *spam* del modulo.

Si noti inoltre che non esisteva un collegamento per la variabile *spam* prima dell'assegnamento `global`.

9.3 Introduzione alle classi

Le classi introducono qualche nuovo aspetto nella sintassi, tre tipi di oggetto nuovi e della nuova semantica.

9.3.1 Sintassi della definizione di una classe

Questa è la forma più semplice di definizione di una classe:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

La definizione delle classi, come quella delle funzioni (l'istruzione `def`) deve essere *eseguita* prima di avere qualsiasi effetto. (Si potrebbe anche collocare la definizione in un ramo di un'istruzione `if`, o all'interno di una funzione.)

In pratica, le istruzioni all'interno di una definizione di classe sono in genere definizioni di funzione: ma sono permesse anche altre istruzioni, e talvolta sono anzi utili (ne riparleremo in seguito). Le definizioni di funzione all'interno della classe di solito hanno una particolare lista di parametri, dovuta alle convenzioni di chiamata per i metodi (di nuovo, ne riparleremo).

Quando il flusso di esecuzione del codice entra nella definizione della classe, un nuovo *namespace* viene creato e usato come *scope* locale: ovvero, tutte le successive assegnazioni di variabili locali finiscono in questo nuovo *namespace*. In particolare, le definizioni di funzione collegano qui il nome della funzione.

Quando si esce dalla definizione della classe nel modo normale (perché il flusso di esecuzione abbandona la classe), viene creato un *oggetto-classe*. Questo oggetto è in sostanza un «contenitore» per il contenuto del *namespace* creato dalla definizione della classe: diremo di più sugli oggetti-classe nella prossima sezione. Lo *scope* locale originario (quello che era attivo subito prima di entrare nella definizione della classe) viene ripristinato e l'oggetto-classe viene collegato in questo *namespace* al nome fornito nell'intestazione della definizione della classe (nel nostro esempio, `ClassName`).

9.3.2 Gli oggetti-classe

Gli oggetti-classe supportano due tipi di operazione: il riferimento agli attributi e l'istanziamento.

Il riferimento agli attributi utilizza la normale sintassi che si usa per queste operazioni in Python: `obj.name`. Un nome di attributo è valido se era nel *namespace* della classe al momento della creazione dell'oggetto-classe. Quindi, se una definizione di classe è fatta così,

```
class MyClass:
    """Un semplice esempio di classe."""
    i = 12345

```

(continua...)

(...segue)

```
    def f(self):
        return 'hello world'
```

allora `MyClass.i` e `MyClass.f` sono riferimenti validi agli attributi, e restituiscono un intero e un oggetto-funzione rispettivamente. Gli attributi della classe possono anche essere assegnati, ovvero potete cambiare il valore di `MyClass.i` con un assegnamento. Anche `__doc__` è un attributo valido, e restituisce la docstring della classe (`"Un semplice esempio di classe."`)

Lo *istanziamento* usa invece la notazione di chiamata di funzione. Fate finta che la classe sia una funzione senza parametri che restituisce una nuova istanza della classe. Per esempio, con riferimento alla classe dell'esempio precedente,

```
x = MyClass()
```

crea una nuova *istanza* della classe e assegna questo oggetto alla variabile locale `x`.

L'operazione di istanziamento («invocare» un oggetto-classe) crea un oggetto vuoto. Molto spesso le classi preferiscono creare istanze predisposte con uno specifico stato iniziale. Per questo è possibile definire nella classe un metodo speciale chiamato `__init__()`, così:

```
def __init__(self):
    self.data = []
```

Se la classe definisce un metodo `__init__()` allora l'operazione di istanziamento lo invoca automaticamente per l'istanza appena creata. Quindi nel nostro esempio, una nuova istanza già inizializzata può essere ottenuta con:

```
x = MyClass()
```

Naturalmente il metodo `__init__()` può essere reso più flessibile dotandolo di parametri. In questo caso gli argomenti passati all'istanziamento della classe sono trasferiti al metodo `__init__()`. Per esempio:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Oggetti-istanza

Che cosa possiamo fare con gli oggetti-istanza? L'unica operazione possibile con questi oggetti è il riferimento agli attributi. Ci sono due tipi di nomi di attributo validi: i *dati* («campi») e i *metodi*.

I *dati* corrispondono alle «variabili di istanza» di Smalltalk e ai «data members» di C++. Gli attributi-dati non devono essere dichiarati: proprio come le variabili locali, iniziano a esistere nel momento in cui sono assegnati per la prima volta. Per esempio, se `x` è una istanza della classe `MyClass` che abbiamo definito sopra, questo codice scriverà il valore «16» senza lasciar traccia:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

L'altro tipo di attributo dell'istanza è il *metodo*. Un metodo è una funzione che «appartiene» all'oggetto-istanza. (In Python, il termine «metodo» non si usa solo in relazione alle istanze delle classi: anche altri tipi di oggetti possono avere dei metodi. Per esempio, gli oggetti-lista hanno metodi come *append*, *insert*, *remove*, *sort* e così

via. In ogni caso, nel resto di questo capitolo, useremo «metodo» solo per riferirci ai metodi degli oggetti-istanza di una classe, a meno che non sia specificato diversamente.)

I nomi validi per i metodi di un'istanza dipendono dalla sua classe. Per definizione, tutti gli attributi della classe che corrispondono a degli oggetti-funzione sono metodi della sua istanza. Quindi nel nostro esempio `x.f` è un riferimento valido al metodo, dal momento che `MyClass.f` è una funzione; ma `x.i` non lo è, perché `MyClass.i` non è una funzione. Tuttavia `x.f` *non è* la stessa cosa di `MyClass.f`: il primo è un oggetto-metodo, il secondo è un oggetto-funzione.

9.3.4 Oggetti-metodo

Di solito un metodo viene invocato non appena è stato collegato:

```
x.f()
```

Nell'esempio di `MyClass`, questa chiamata restituirà la stringa `'hello world'`. Tuttavia non è necessario invocare il metodo immediatamente: `x.f` è un oggetto-metodo che può essere «conservato» e chiamato più tardi. Per esempio,

```
xf = x.f
while True:
    print(xf())
```

continuerà a scrivere `hello world` fino alla fine del mondo.

Che cosa succede di preciso quando un metodo è invocato? Avrete notato che l'invocazione `x.f()` è stata fatta senza passare argomenti, anche se la definizione di `f()` specifica in effetti un parametro. Che cosa succede a questo? Certamente Python dovrebbe emettere un'eccezione se una funzione che richiede un argomento è invocata senza passarlo, anche se poi l'argomento non dovesse essere usato nella funzione stessa...

In realtà probabilmente avrete indovinato la risposta: la peculiarità dei metodi è che l'oggetto-istanza è passato automaticamente come primo argomento della funzione. Nel nostro esempio, la chiamata `x.f()` è esattamente equivalente a `MyClass.f(x)`. In generale, invocare un metodo con una lista di *n* argomenti è equivalente a chiamare la corrispondente funzione con una lista di argomenti identica, ma che inserisce al primo posto l'oggetto-istanza.

Se non riuscite a comprendere esattamente come funziona, può essere utile dare un'occhiata all'implementazione. Quando referenziamo un attributo (che non sia un dato) di una classe, il nome viene cercato nell'istanza della classe. Se il nome corrisponde a un attributo che è un oggetto-funzione, allora un oggetto-metodo viene creato mettendo insieme (puntatori a) l'oggetto-istanza e l'oggetto-funzione appena trovato, per formare un nuovo oggetto astratto: l'oggetto-metodo, appunto. Quando l'oggetto-metodo è invocato con una lista di argomenti, una nuova lista viene creata unendola all'oggetto-istanza: l'oggetto-funzione viene chiamato con questa nuova lista di argomenti.

9.3.5 Variabili di classe e di istanza

In generale, le variabili di istanza sono per i dati che devono restare unici per ciascuna istanza; le variabili di classe sono per attributi e metodi condivisi tra tutte le istanze della classe:

```
class Dog:

    kind = 'canine'         # variabile di classe condivisa tra le istanze

    def __init__(self, name):
        self.name = name    # variabile di istanza unica per ciascuna istanza

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # condivisa tra tutti i cani
```

(continua...)

(...segue)

```
'canine'
>>> e.kind                  # condivisa tra tutti i cani
'canine'
>>> d.name                  # unica per d
'Fido'
>>> e.name                  # unica per e
'Buddy'
```

Come abbiamo visto in *A proposito di nomi e oggetti*, i dati condivisi possono avere comportamenti sorprendenti quando sono oggetti mutabili come liste e dizionari. Nell'esempio che segue, la lista *tricks* non dovrebbe essere utilizzata come una variabile di classe, perché una singola lista verrebbe condivisa tra tutte le istanze di *Dog*:

```
class Dog:

    tricks = []             # uso sbagliato di una variabile di classe

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

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # inaspettata condivisione tra tutte le istanze
['roll over', 'play dead']
```

Il design corretto per la classe prevede l'uso di una variabile *di istanza*, invece:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # crea una lista vuota per ciascuna istanza

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 Osservazioni varie

Se lo stesso nome è usato per un attributo di classe e uno di istanza, allora il meccanismo di ricerca dà priorità all'attributo di istanza:

```
>>> class Warehouse:
        purpose = 'storage'
        region = 'west'

>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

Gli attributi che sono dati possono essere referenziati anche dai metodi, oltre che dai normali «clienti» (utilizzatori) di un oggetto. In altre parole, le classi non sono adatte a implementare tipi di dati astratti. In effetti, non è in alcun modo possibile in Python garantire la protezione di un dato: questo può essere solo basato su convenzioni. (D'altro canto, la *implementazione* di Python, scritta in C, può nascondere completamente i dettagli di implementazione e controllare l'accesso a un oggetto, se necessario: si può sfruttare questo aspetto scrivendo delle estensioni di Python in C.)

I «clienti» dovrebbero fare attenzione a usare gli attributi-dati: potrebbero scompaginare delle invarianti utilizzate dai metodi, sovrascrivendole con i loro attributi-dati. Si noti che i clienti possono aggiungere dati per conto proprio a un oggetto-istanza, senza compromettere il funzionamento dei metodi, fintanto che non ci sono conflitti tra i nomi. Ancora una volta, l'uso di una convenzione per i nomi può risparmiare molti grattacapi.

Non ci sono particolari scorciatoie per referenziare dati (o metodi) dall'interno dei metodi. Riteniamo che in questo modo il codice sia più leggibile: non c'è la possibilità di confondere variabili locali e variabili di istanza quando si scorre con l'occhio il codice di un metodo.

Di solito il primo parametro di un metodo viene chiamato `self`. Si tratta solo di una convenzione: il nome di per sé non ha alcun significato speciale per Python. Notate tuttavia che non seguire questa convenzione rende il vostro codice meno leggibile per gli altri programmatori Python; è anche probabile che gli strumenti di introspezione del codice si basino sul rispetto di questa convenzione.

Ogni oggetto-funzione che sia un attributo di classe definisce un oggetto-metodo per l'istanza di quella classe. Non è necessario che il codice della funzione sia fisicamente contenuto all'interno della definizione della classe: si può anche assegnare una variabile locale a un oggetto-funzione esterno. Per esempio:

```
# Una funzione definita all'esterno della classe
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Adesso `f`, `g` e `h` sono tutti attributi della classe `C`, che si riferiscono a oggetti-funzione: di conseguenza sono anche tutti metodi delle istanze della classe; `h` sarà esattamente equivalente a `g`. Si noti però che questa pratica in genere confonde solo le idee a chi deve leggere il codice.

I metodi possono invocare altri metodi usando gli attributi-metodo del loro parametro `self`:

```
class Bag:
    def __init__(self):
```

(continua...)

(...segue)

```
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

I metodi possono accedere ai nomi globali nello stesso modo delle funzioni ordinarie. Lo *scope* globale associato a un metodo è il modulo che contiene la definizione. (Una classe non è mai utilizzata come *scope* globale.) Anche se di rado esiste una ragione valida per accedere a un *dato* globale dall'interno di un metodo, ci sono comunque motivi validi per usare lo *scope* globale: per cominciare, le funzioni e i moduli importati nello *scope* globale possono essere usate dai metodi, così come le funzioni e le classi ivi definite. In genere la classe che contiene il metodo è essa stessa definita nello *scope* globale, e nella prossima sezione vedremo dei buoni motivi per cui un metodo potrebbe voler accedere al nome della sua stessa classe.

Ogni valore è un oggetto e di conseguenza ha una *classe*, che è chiamata il suo *tipo*. Il nome della classe/tipo è conservato in `object.__class__`.

9.5 Ereditarietà

Naturalmente una classe non sarebbe degna di questo nome se non supportasse l'ereditarietà. La sintassi per definire una sotto-classe è questa:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Il nome della classe-madre `BaseClassName` deve essere definito in uno *scope* che contiene la definizione della sotto-classe. Al posto del nome della classe-madre è anche consentito inserire un'espressione arbitraria. Questo è utile, per esempio, quando la classe-madre è definita in un altro modulo:

```
class DerivedClassName(modname.BaseClassName):
```

L'esecuzione della definizione di una sotto-classe è simile a quella della classe-madre. Al momento della sua costruzione, l'oggetto-classe ricorda la sua classe-madre. In questo modo può risolvere i riferimenti agli attributi: se viene richiesto un attributo che non si trova nella classe, la ricerca procede nella classe-madre. Il meccanismo si applica ricorsivamente, se la classe-madre a sua volta deriva da qualche altra classe.

Non vi è nulla di speciale nell'istanziare una sotto-classe: `DerivedClassName()` crea una nuova istanza della classe. I riferimenti ai metodi sono risolti così: si cerca il corrispondente attributo di classe, scendendo lungo la catena delle classi-madri se necessario; il riferimento al metodo è valido se il nome trovato corrisponde a un oggetto-funzione.

Le sotto-classi possono sovrascrivere i metodi delle loro classi-madri. Dal momento che i metodi non hanno privilegi speciali quando chiamano altri metodi dello stesso oggetto, un metodo in una classe-madre che chiama un altro metodo definito nella stessa classe potrebbe finire per chiamare in realtà un metodo sovrascritto in una sotto-classe. (Per i programmatori C++: tutti i metodi in Python sono `virtual`.)

Un metodo di una sotto-classe potrebbe voler *estendere* invece che semplicemente rimpiazzare il metodo della classe-madre con lo stesso nome. Per chiamare il metodo della classe-madre, semplicemente basta chiamare `BaseClassName.methodname(self, arguments)`. Talvolta questa tecnica può servire anche al codice «cliente». (Si noti però che questo funziona solo se la classe-madre è accessibile nello *scope* globale come `BaseClassName`.)

Python ha due funzioni predefinite che si occupano di ereditarietà:

- `isinstance()` controlla il tipo di un'istanza: `isinstance(obj, int)` restituirà `True` se `obj.__class__` è un `int` o qualcosa derivato da `int`.
- `issubclass()` controlla l'ereditarietà di una classe: `issubclass(bool, int)` restituisce `True`, dal momento che la classe `bool` è una sotto-classe di `int`. Al contrario, `issubclass(float, int)` è `False` perché `float` non è una sotto-classe di `int`.

9.5.1 Ereditarietà multipla

Python supporta anche una forma di ereditarietà multipla. Una classe con più di una classe-madre si può scrivere così:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Nei casi più semplici e per la maggior parte dei casi d'uso, potete assumere che la ricerca di un attributo ereditato proceda da sinistra a destra, con una «ricerca in profondità», e senza cercare una seconda volta nella stessa classe quando le gerarchie si sovrappongono. Quindi, se un attributo non viene trovato in `DerivedClassName`, lo si cerca in `Base1`, quindi ricorsivamente nelle classi-madre di `Base1`, quindi in `Base2` e così via.

In realtà le cose sono leggermente più complicate; il meccanismo di ricerca dei metodi cambia dinamicamente per supportare chiamate cooperative alla funzione `super()`. Questo approccio è noto come «call-next-method» in alcuni linguaggi dotati di ereditarietà multipla e offre più possibilità rispetto al *super* dei linguaggi con ereditarietà semplice.

La ricerca con ordinamento dinamico si rende necessaria perché tutte le ereditarietà multiple finiscono per avere una o più gerarchie «a rombo»: ovvero, almeno una delle classi-madre può essere raggiunta in più di un modo a partire dalla sotto-classe. Per esempio, in Python tutte le classi ereditano da `object`, quindi tutti gli schemi di ereditarietà multipla hanno necessariamente più di un percorso per arrivare a `object`. Per evitare di cercare più di una volta nelle classi-madre, l'algoritmo dinamico traccia un percorso di ricerca lineare tale da preservare il principio «da sinistra a destra», da raggiungere ciascuna classe-madre una sola volta, e da essere monotonico (ovvero, è possibile creare una sotto-classe senza influenzare il percorso di ricerca già esistente per la gerarchia superiore). Queste proprietà, nel loro insieme, rendono possibile la progettazione di classi affidabili ed estensibili in un contesto di ereditarietà multipla. Per ulteriori dettagli, si veda https://www.python.org/download/releases/2.3/mro/.

9.6 Variabili private

In Python non esiste il concetto di istanza «privata» di una variabile, che è accessibile solo dall'interno del suo oggetto. Tuttavia esiste una convenzione, adottata quasi ovunque nel codice scritto in Python: un nome che inizia con il «trattino basso» (per esempio `_spam`) dovrebbe essere trattato come una componente non-pubblica della API (che sia una funzione, un metodo o un dato). Dovrebbe essere considerato come un dettaglio di implementazione, suscettibile di essere modificato in futuro senza preavviso.

Esiste almeno uno scenario reale in cui è desiderabile disporre di una variabile privata: quando si vogliono evitare conflitti con nomi definiti dalle sotto-classi. Python fornisce un supporto limitato per questa necessità, attraverso il *name mangling*. Tutti i nomi che hanno almeno due «trattini bassi» iniziali e non più di un «trattino basso» finale (come `__spam`) sono rimpiazzati con `_classname__spam`, dove `classname` è il nome della classe corrente senza trattini bassi iniziali. Questa manipolazione avviene per tutti gli identificatori di questo tipo, indipendentemente dalla loro posizione, purché siano definiti all'interno della classe.

La manipolazione dei nomi permette alle sotto-classi di sovrascrivere un metodo senza comprometterne l'invocazione da un'altra classe. Per esempio:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # una copia privata del metodo update()

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # sovrascrive update() con una nuova *signature*
        # ma non rompe il funzionamento della chiamata in __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Questo esempio funzionerebbe anche se `MappingSubclass` volesse introdurre un suo `__update`, dal momento che sarebbe rimpiazzato con `_Mapping__update` nella classe-madre e con `_MappingSubclass__update` nella sotto-classe.

Si noti che il meccanismo del *mangling* vuole essere soprattutto un modo per evitare conflitti di nomi: è comunque sempre possibile accedere o modificare una variabile «privata». Questo può essere anzi utile in talune circostanze, per esempio in un debugger.

Si noti inoltre che il codice passato alle funzioni `exec()` o `eval()` non considera la classe che invoca il metodo come la classe «corrente» e quindi non usa quel nome per il *mangling*; è un effetto simile a quello dell'istruzione `global`, che infatti, anch'essa, vale solo nel codice che è stato compilato insieme (nel senso di *byte-compiled*). Lo stesso vale per `getattr()`, `setattr()` e `delattr()`, e anche quando si utilizza direttamente `__dict__`.

9.7 Note varie

Può essere utile talvolta disporre di una struttura-dati simile al «record» di Pascal o a «struct» in C, impacchettando insieme alcuni dati referenziati con variabili. Si può usare una classe vuota:

```
class Employee:
    pass

john = Employee()  # Crea una scheda di impiegato vuota

# Riempie i campi della scheda
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Se una sezione di codice Python si aspetta di ricevere uno specifico tipo di dato astratto, le si può passare invece una classe che emula i metodi di quel tipo di dato. Per esempio, se avete una funzione che formatta dei dati provenienti da un file di testo, potete definire una classe con dei metodi `read()` e `readline()` che invece prelevano i dati da una stringa-buffer, e passare questa alla funzione come argomento.

Gli oggetti-metodi di istanza hanno a loro volta degli attributi: `m.__self__` è l'oggetto-istanza che possiede il metodo `m()`, e `m.__func__` è l'oggetto-funzione corrispondente al metodo.

9.8 Iteratori

Avrete probabilmente già notato che è possibile iterare su molti oggetti contenitori con l'istruzione `for`:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

Questo modo di accesso è chiaro, sintetico, efficiente. L'uso degli iteratori è onnipresente in Python. Dietro le quinte, l'istruzione `for` chiama la funzione `iter()` dell'oggetto contenitore. La funzione restituisce un oggetto iteratore, che a sua volta definisce il metodo `__next__()`, che accede agli elementi del contenitore, uno alla volta. Quando gli elementi sono finiti, `__next__()` emette un'eccezione `StopIteration`, che comunica all'istruzione `for` di terminare. Potete chiamare direttamente il metodo `__next__()` usando la funzione predefinita `next()`. Questo esempio spiega come funziona il meccanismo:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Conoscendo il meccanismo che governa il comportamento degli iteratori, è facile aggiungere questa funzionalità alle vostre classi. Occorre definire un metodo `__iter__()` che restituisce un oggetto a sua volta dotato di un metodo `__next__()`. Se la classe definisce già `__next__()`, allora `__iter__()` può limitarsi a restituire `self`:

```
class Reverse:
    """Un iteratore che cicla all'indietro su una sequenza."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
```

(continua...)

(...segue)

```
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.9 Generatori

Un generatore è uno strumento semplice e potente per creare iteratori. I generatori sono definiti come normali funzioni che però utilizzano l'istruzione `yield` quando vogliono restituire dei dati. Ogni volta che la funzione `next()` viene chiamata su un generatore, questo riprende l'esecuzione da dove l'aveva interrotta (ricorda tutti i valori in sospeso e qual è stata l'ultima istruzione eseguita). Ecco un esempio che mostra come creare un generatore può essere molto semplice:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Tutto ciò che può essere fatto con un generatore può anche essere fatto con un iteratore in una classe, come visto nel paragrafo precedente. I generatori però sono più compatti grazie al fatto che i metodi `__iter__()` e `__next__()` vengono creati automaticamente.

Un altro vantaggio importante è che le variabili locali e lo stato dell'esecuzione vengono salvati tra una chiamata e l'altra. In questo modo scrivere la funzione è più facile e molto più chiaro, rispetto a dover usare variabili di istanza come `self.index` e `self.data`.

Oltre alla creazione automatica dei metodi e alla persistenza dello stato del programma, un generatore emette automaticamente un'eccezione `StopIteration` quando termina. Combinate insieme, queste caratteristiche permettono di creare iteratori con la stessa facilità con cui si scrive una normale funzione.

9.10 Espressioni-generatore

Alcuni semplici generatori possono essere scritti in modo sintetico come delle espressioni, usando una sintassi simile a quella delle *list comprehension*, ma con le parentesi tonde invece delle parentesi quadre. Queste espressioni sono adatte alle situazioni in cui il generatore è consumato immediatamente da una funzione di ordine superiore. Le espressioni-generatore sono più compatte, ma meno versatili rispetto a un normale generatore; tendono a consumare meno memoria dell'equivalente *list comprehension*.

Esempi:

```
>>> sum(i*i for i in range(10))                 # somma di quadrati
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # prodotto scalare
```

(continua...)

(...segue)

```
260

>>> unique_words = set(word for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

CAPITOLO 10

Una breve visita alla libreria standard

10.1 Interfacce al sistema operativo

Il modulo `os` contiene moltissime funzioni per interagire con il sistema operativo:

```
>>> import os
>>> os.getcwd()      # restituisce la directory corrente
'C:\\Python310'
>>> os.chdir('/server/accesslogs')   # cambia la directory corrente
>>> os.system('mkdir today')   # esegue "mkdir" nella shell di sistema
0
```

È importante usare `import os` e non `from os import *`, in modo che la funzione `os.open()` non mascheri la funzione predefinita `open()` che lavora in modo molto differente.

È conveniente ricorrere alle funzioni predefinite `dir()` e `help()` per ricevere aiuto interattivo quando si lavora con moduli di grandi dimensioni come `os`:

```
>>> import os
>>> dir(os)
<restituisce una lista di tutte le funzioni del modulo>
>>> help(os)
<restituisce una documentazione completa costruita in base alle docstring>
```

Per il lavoro di tutti i giorni con file e directory, `shutil` fornisce un'interfaccia di livello più alto, che è più semplice da usare:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2 Caratteri jolly per i file

Il modulo `glob` comprende una funzione per cercare file con i caratteri jolly:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Argomenti della riga di comando

Molti script di hanno bisogno di processare gli argomenti della riga di comando. Questi argomenti vengono conservati nell'attributo *argv* del modulo `sys`, sotto forma di una lista. Per esempio, l'output che segue risulta dall'aver invocato `python demo.py one two three` al prompt della shell:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

Il modulo `argparse` mette a disposizione un meccanismo più sofisticato per gestire gli argomenti della riga di comando. Lo script che segue estrae uno o più nomi di file e un numero opzionale di righe da visualizzare:

```
import argparse

parser = argparse.ArgumentParser(prog = 'top',
    description = 'Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

Quando viene invocato con `python top.py --lines=5 alpha.txt beta.txt`, lo script imposta `args.lines` a 5 e `args.filenames` a `['alpha.txt', 'beta.txt']`.

10.4 Re-dirigere lo standard error e terminare il programma

Il modulo `sys` ha degli attributi per *stdin*, *stdout* e *stderr*. Quest'ultimo è utile per emettere avvisi e messaggi d'errore e renderli visibili anche quando lo standard output è stato re-diretto:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

Il modo più diretto per terminare un programma è usare `sys.exit()`.

10.5 Ricerca di pattern nelle stringhe

Il modulo `re` fornisce strumenti per il trattamento delle stringhe con le *regular expression*. Per ricerche e manipolazioni sofisticate, le regular expression costituiscono una soluzione compatta ed efficiente:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Tuttavia, per ricerche e sostituzioni semplici, è preferibile usare i metodi delle stringhe, che sono più semplici da leggere e correggere:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Matematica

Il modulo `math` dà accesso alla sottostante libreria C, che raccoglie funzioni per il calcolo in virgola mobile:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Il modulo `random` consente di effettuare selezioni casuali:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # campionamento senza rimpiazzamento
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # un float casuale
0.17970987693706186
>>> random.randrange(6)    # in intero casuale compreso in range(6)
4
```

Il modulo `statistics` produce misure statistiche di base (media, mediana, varianza etc.) su dati numerici:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

Il progetto SciPy offre molti altri moduli per il calcolo numerico.

10.7 Accesso a internet

Esistono diversi moduli per accedere a internet e gestire i protocolli internet. Due dei più semplici sono `urllib.request` per raccogliere dati da una URL e `smtplib` per spedire email:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8')  # converte i dati binari in testo
...         if 'EST' in line or 'EDT' in line:  # cerca EST
...             print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
```

(continua...)

(...segue)

```
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Si noti che l'ultimo esempio richiede che un server mail sia funzionante su localhost.)

10.8 Date e orari

Il modulo `datetime` contiene delle classi per manipolazioni semplici e complesse di date e orari. Anche se i calcoli con le date sono supportati, il modulo si concentra soprattutto sull'estrazione dei componenti per scopi di manipolazione e formattazione. Sono anche previsti oggetti sensibili alle *timezone*.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # le date supportano l'aritmetica del calendario
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Compressione dei dati

I moduli `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` e `tarfile` offrono il supporto per i comuni formati di archiviazione e compressione dei dati.

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Misurazione di performance

Alcuni utenti di Python sono molto interessati a conoscere la differenza tra vari approcci allo stesso problema, in termini di performance. Python mette a disposizione uno strumento di misura che risponde immediatamente a queste domande.

Per esempio, si può provare a usare lo spacchettamento di tupla, invece del tradizionale approccio di scambiare le variabili. Il modulo `timeit` ci fa rapidamente vedere che in effetti esiste un leggero vantaggio di performance:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

Mentre `timeit` ha un livello di granularità più fine, i moduli `profile` e `pstats` forniscono strumenti per identificare, all'interno di sezioni di codice più ampie, le parti che provocano rallentamenti.

10.11 Controllo di qualità

Una strada per scrivere codice di alta qualità è quella di scrivere dei test per ciascuna funzione, man mano che viene sviluppata, e di eseguire i test con una certa frequenza durante il processo di sviluppo.

Il modulo `doctest` è uno strumento per scansionare un modulo e validare i test che sono contenuti nelle sue docstring. Creare un test è questione di un semplice copia-e-incolla, nella docstring, dell'invocazione e del risultato atteso. In questo modo si migliora la documentazione, fornendo un esempio di utilizzo per l'utente, e si permette a *doctest* di garantire che il codice resti fedele a quanto documentato:

```
def average(values):
    """Restituisce la media aritmetica di una lista di numeri.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # valida automaticamente i test inclusi
```

Il modulo `unittest` non è di immediato utilizzo come `doctest`, ma permette di mantenere una raccolta più completa di test in file separati:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()  # invocare dalla riga di comando esegue tutti i test
```

10.12 Le batterie sono incluse

La filosofia di Python è che «le batterie sono incluse». Ne è prova l'inclusione nella libreria standard di grandi package che forniscono strumenti più sofisticati e robusti. Per esempio:

- Con i moduli `xmlrpc.client` e `xmlrpc.server`, realizzare invocazioni di procedure remote diventa quasi banale. Nonostante il nome, non è necessario conoscere o manipolare XML per usarli.
- Il package `email` è una libreria per manipolare i messaggi email, che include MIME e altri documenti basati sulla **RFC 2822**. A differenza di `smtplib` e `poplib`, che ricevono e spediscono messaggi, questo package fornisce un set di strumenti completo per costruire e decodificare strutture complesse, allegati inclusi, e per implementare gli encoding di internet e i protocolli degli *header*.
- Il package `json` supporta il *parsing* di questo popolare formato d'interscambio. Il modulo `csv` fornisce strumenti per la lettura e scrittura di file in formato CSV, molto diffuso per i database e i fogli di calcolo. La gestione di XML è garantita dai package `xml.etree.ElementTree`, `xml.dom` e `xml.sax`. Complessivamente, questi moduli e package semplificano molto lo scambio di informazioni tra le applicazioni Python e il mondo esterno.
- Il modulo `sqlite3` permette l'accesso ai database SQLite, mettendo a disposizione uno strumento di persistenza accessibile con una sintassi SQL leggermente modificata.
- L'internazionalizzazione è garantita da un gran numero di moduli come `gettext`, `locale` e il package `codecs`.

CAPITOLO 11

Una breve visita alla libreria standard - 2

Questa seconda parte del tour presenta strumenti più avanzati che supportano le esigenze dei programmi professionali. Di rado ce n'è bisogno per i piccoli script.

11.1 Formattazione dell'output

Il modulo `reprlib` presenta una versione della funzione predefinita `repr()`, adattata per visualizzare in forma abbreviata le collezioni molto grandi o con molti livelli di annidamento:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

`pprint` permette un controllo più raffinato sulla scrittura di oggetti predefiniti o creati dall'utente, in modo che l'output resti leggibile dall'interprete. Quando il risultato è più lungo di una riga, il *pretty printer* aggiunge interruzioni di riga e rientri per rendere più chiara la struttura dei dati:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

`textwrap` formatta i paragrafi di testo in modo che rispettino una determinata larghezza dello schermo:

```
>>> import textwrap
>>> doc = """Il metodo wrap() è come fill(), tranne che restituisce una
... lista di stringhe, invece di una sola stringa lunga con gli a-capo
... che separano le righe processate."""
...
>>> print(textwrap.fill(doc, width=40))
Il metodo wrap() è come fill(), tranne
```

(continua...)

(...segue)

```
che restituisce una lista di stringhe,
invece di una sola stringa lunga con gli
a-capo che separano le righe processate.
```

Il modulo `locale` accede al database dei formati «culturali» specifici per i dati. Per esempio, l'attributo *grouping* della funzione localizzata *format* permette di formattare i numeri con i separatori di gruppo corretti:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()  # un mapping delle convenzioni applicabili
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                       conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Template

Il modulo `string` include una versatile classe `Template`, con una sintassi semplice, adatta a essere modificata dagli utenti finali. In questo modo gli utenti possono personalizzare il programma senza doverne alterare il codice.

Il formato utilizza dei nomi segnaposto composti da `$` con un identificatore Python valido (ovvero, caratteri alfanumerici e trattini bassi). Se il segnaposto è inserito tra parentesi, è possibile aggiungere dei caratteri immediatamente dopo, senza spazio in mezzo. Un `$$` è lo *escape* che produce un singolo `$`:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

Il metodo `substitute()` emette un `KeyError` quando il segnaposto non è «alimentato» da un dizionario o un argomento *keyword*. Per le applicazioni di tipo «stampa unione», i dati forniti potrebbero essere incompleti e quindi potrebbe essere più appropriato il metodo `safe_substitute()` che lascia semplicemente il segnaposto, quando il dato manca:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Una sotto-classe di *Template* può specificare un delimitatore arbitrario. Per esempio, un tool per la rinomina automatica di una collezione di foto potrebbe decidere di usare il simbolo di percentuale per segnaposti come la data corrente, un numero progressivo, un formato di file e così via:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
```

(continua...)

(...segue)

```
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Un altro scenario in cui i template sono utili è per separare la logica del programma dai dettagli di ciascun formato di output. In questo modo è possibile per esempio sostituire template personalizzati per file XML, output di solo testo o report in HTML.

11.3 Formati per campi di dati binari

Il modulo `struct` ha le funzioni `pack()` e `unpack()` che consentono di lavorare con record di dati binari di lunghezza variabile. Questo esempio mostra come iterare sullo *header* di un file ZIP, senza usare il modulo `zipfile`. I codici `"H"` e `"I"` indicano un numero di due e quattro byte senza segno. Il segno `"<"` indica che si tratta di numeri di larghezza standard e ordinamento *little-endian*:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                      # mostra i primi 3 headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # salta allo header successivo
```

11.4 Multi-threading

Il *threading* è una tecnica per separare *task* che non dipendono da un'esecuzione sequenziale. Si possono usare i thread per migliorare la reattività delle applicazioni che devono ricevere input dall'utente mentre svolgono altri compiti in background. Un caso d'uso simile è la necessità di compiere operazioni di input/output in parallelo con dei calcoli in un altro thread.

Questo esempio mostra come l'interfaccia di alto livello del modulo `threading` permette di eseguire compiti in background mentre il programma principale continua a essere attivo:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
```

(continua...)

(...segue)

```
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # aspetta che i compiti in background finiscano
print('Main program waited until background was done.')
```

La sfida principale dei programmi multi-threading è di coordinare i thread che devono condividere dati o altre risorse. Per questo, il modulo *threading* mette a disposizione diverse primitive di sincronizzazione come lock, eventi, condizioni e semafori.

Anche con questi strumenti raffinati, piccoli errori di design possono causare problemi difficili da riprodurre. Di conseguenza, l'approccio più usato consiste nel concentrare tutte le operazioni di accesso alla risorsa in un solo thread, e quindi usare il modulo `queue` per fare arrivare a quel thread le richieste degli altri. Usare oggetti `Queue` per le comunicazioni tra thread porta a scrivere applicazioni più semplici da progettare, più leggibili e affidabili.

11.5 Logging

Il modulo `logging` offre un sistema di logging completo e flessibile. Nella forma più semplice, un messaggio di log è inviato a un file o a `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Questo produce l'output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

I messaggi di informazione e debug sono soppressi di default, quando l'output è inviato allo *standard error*. Altre opzioni per l'output comprendono l'invio di messaggi tramite email, *datagram*, socket, o a un server HTTP. Un filtro può scegliere la modalità di invio in base alla priorità del messaggio: `DEBUG`, `INFO`, `WARNING`, `ERROR`, e `CRITICAL`.

Il sistema di logging è configurabile direttamente da Python, o può essere inizializzato da un file di configurazione modificabile dall'utente, per personalizzare il sistema senza alterare il codice dell'applicazione.

11.6 Weak References

Python gestisce automaticamente la memoria, facendo *reference counting* per gli oggetti e usando il garbage collection per eliminarli. La memoria viene liberata poco dopo che l'ultimo riferimento all'oggetto è stato cancellato.

Questo approccio funziona bene nella maggior parte dei casi, ma talvolta si rende necessario tracciare un oggetto per tutto il tempo in cui è usato da qualcun altro. Purtroppo, questo tracciamento comporta la creazione di un riferimento, cosa che rende l'oggetto permanente. Il modulo `weakref` permette invece di tracciare oggetti senza per questo dover creare riferimenti. Quando non c'è più bisogno dell'oggetto, questo viene automaticamente rimosso dal registro delle *weak references* e un callback viene invocato per l'oggetto *weakref*. Questo meccanismo viene usato, per esempio, per conservare in cache gli oggetti costosi da creare:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # crea un riferimento
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # non crea un riferimento
>>> d['primary']                # raggiunge l'oggetto se è ancora in vita
10
>>> del a                       # elimina il riferimento
>>> gc.collect()                # aziona subito il garbage collector
0
>>> d['primary']                # la chiave è stata rimossa automaticamente
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # la chiave è stata rimossa automaticamente
  File "C:/python310/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Strumenti per lavorare con le liste

Il tipo predefinito «lista» può soddisfare le esigenze di molte strutture-dati. Tuttavia occasionalmente c'è bisogno di un'implementazione alternativa con altri vantaggi e svantaggi in termini di performance.

Il modulo `array` ha una classe `array()` simile a una lista che conserva i dati in modo più compatto, ma solo se sono di un medesimo tipo. L'esempio che segue mostra un array i cui elementi sono conservati come numeri binari di due byte senza segno (codice `"H"`), invece dei consueti 16 byte che sarebbero impiegati da una normale lista Python:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

Il modulo `collections` ha un oggetto `deque()` simile a una lista, che permette *append* e *pop* rapidi a entrambi gli estremi, ma accessi più lenti al centro. Questi oggetti vanno bene per implementare code e ricerche in ampiezza nei grafi:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
```

(continua...)

(...segue)

```
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

Oltre a implementazioni alternative per le liste, la libreria standard contiene anche altri strumenti, come il modulo `bisect` che può manipolare le liste ordinate:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

Il modulo `heapq` implementa *heap* a partire da normali liste. Il valore più basso è sempre mantenuto all'inizio. Ciò è utile per le applicazioni che hanno bisogno di accedere spesso all'elemento più piccolo, senza dover ordinare tutta la lista per trovarlo:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # ordina la lista come heap
>>> heappush(data, -5)                 # aggiunge un valore
>>> [heappop(data) for i in range(3)]  # produce i tre valori più piccoli
[-5, 0, 1]
```

11.8 Aritmetica decimale in virgola mobile

Il modulo `decimal` offre un tipo `Decimal` per operare con i numeri decimali in virgola mobile. In confronto con il tipo predefinito `float` che implementa un numero *binario* in virgola mobile, questa classe è conveniente per

- le applicazioni finanziarie, o quando è richiesta una rappresentazione decimale esatta,
- avere più controllo sulla precisione,
- avere più controllo sull'arrotondamento per esigenze legali o normative,
- mantenere le cifre decimali significative,
- le applicazioni dove il risultato deve essere uguale al calcolo fatto a mano.

Per esempio, calcolare il 5% di tasse su 70 centesimi di costo telefonico fornisce un risultato diverso in virgola mobile decimale o binaria. La differenza diventa importante se il risultato è arrotondato al centesimo più vicino:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

I risultati in `Decimal` mantengono gli zero finali, conservando quattro decimali significativi da una moltiplicazione tra numeri con due decimali significativi. Il modulo *decimal* riproduce il risultato dei calcoli fatti a mano

ed evita i problemi che nascono quando una quantità binaria in virgola mobile non può rappresentare esattamente una quantità decimale.

Usare una rappresentazione esatta permette a `Decimal` di calcolare i resti precisamente e di effettuare test di uguaglianza che fallirebbero con la rappresentazione binaria in virgola mobile:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

Il modulo `decimal` permette di svolgere calcoli con tutta la precisione richiesta:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

CAPITOLO 12

Virtual environment e package

12.1 Introduzione

I programmi Python usano spesso moduli e package che non sono compresi nella libreria standard. Inoltre le applicazioni talvolta hanno bisogno di una specifica versione di una libreria, perché è necessario che un certo baco sia stato risolto, oppure perché fanno uso di una vecchia versione dell'interfaccia della libreria.

Ciò vuol dire che non è possibile che una singola installazione di Python possa venire incontro alle esigenze di ogni possibile applicazione. Se il programma A richiede la versione 1.0 di un certo modulo, ma il programma B ha bisogno della 2.0, queste necessità sono in conflitto e installare una delle due versioni non permetterà all'altro programma di funzionare correttamente.

La soluzione è creare un virtual environment, ovvero una directory auto-sufficiente che contiene una installazione di Python, per una particolare versione di Python, oltre a un certo numero di pacchetti aggiuntivi.

Programmi diversi possono usare virtual environment diversi. Per risolvere il problema di richieste in conflitto dell'esempio precedente, il programma A può avere il suo environment con la versione 1.0 installate, mentre il programma B avrà un altro virtual environment con la versione 2.0. Se in seguito B richiede un aggiornamento della libreria alla versione 3.0, ciò non avrà conseguenze sull'environment di A.

12.2 Creare un virtual environment

`venv` è il modulo usato per creare e gestire virtual environment. `venv` installa in genere la versione più recente di Python che avete disponibile. Se avete installato più versioni di Python nel vostro sistema, potete selezionarne una in particolare invocando `python3` o qualsiasi versione desiderate.

Per creare un virtual environment, decidete in quale directory volete collocarlo e avviate il modulo `venv` come uno script, passando il percorso della directory scelta:

```
python3 -m venv tutorial-env
```

Questo crea la directory `tutorial-env` se non esiste; inoltre crea al suo interno le directory che contengono una copia dell'interprete Python, la libreria standard e diversi file di corredo.

Un luogo comune per conservare i virtual environment è `.venv`: questo nome mantiene la directory nascosta nella shell in modo che non sia d'impiccio, e al contempo dice chiaramente a che cosa serve la directory. Inoltre evita conflitti con i file `.env` di definizione delle variabili d'ambiente, che qualche tool supporta.

Creato il virtual environment, non resta che attivarlo.

Su Windows invocate:

```
tutorial-env\Scripts\activate.bat
```

Su Unix o MacOS:

```
source tutorial-env/bin/activate
```

(Lo script è scritto per la bash shell. Se usate **csh** o **fish**, usate invece gli script alternativi `activate.csh` o `activate.fish`.)

Attivare il virtual environment cambia il prompt della shell per mostrare il nome dell'environment in uso; modifica inoltre l'ambiente di lavoro in modo che invocare `python` restituisca quella particolare versione e installazione dell'interprete. Per esempio:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May  6 2016, 10:59:36)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

12.3 Gestire i pacchetti con Pip

Potete installare, aggiornare, rimuovere package usando un programma chiamato **pip**. Per default `pip` installerà pacchetti pubblicati sul Python Package Index. Potete cercare nel PyPI con il vostro browser o usando le funzioni di ricerca di Pip:

```
(tutorial-env) $ pip search astronomy
skyfield               - Elegant astronomy for Python
gary                   - Galactic astronomy and gravitational dynamics.
novas                  - The United States Naval Observatory NOVAS astronomy␣
↪library
astroobs               - Provides astronomy ephemeris to plan telescope␣
↪observations
PyAstronomy            - A collection of astronomy related tools for Python.
...
```

`pip` offre un numero di comandi interni: «search», «install», «uninstall», «freeze», etc. (Si veda la guida a Installare moduli Python per la documentazione completa di `pip`.)

Per installare l'ultima versione disponibile di un package, basta specificare il suo nome:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

Potete anche installare una versione specifica, indicando il nome seguito da == e il numero di versione:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
```

(continua...)

(...segue)

```
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Se eseguite due volte questo comando, `pip` vi informerà che la versione richiesta è già presente e non farà nient'altro. Potete indicare un altro numero di versione per ottenere quella, oppure eseguire `pip install --upgrade` per aggiornare il pacchetto all'ultima versione:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`pip uninstall`, seguito dal nome di uno o più pacchetti, li rimuoverà dal virtual environment.

`pip show` visualizza informazioni su un particolare pacchetto:

```
(tutorial-env) $ pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`pip list` elenca tutti i pacchetti installati in un virtual environment:

```
(tutorial-env) $ pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`pip freeze` produce una lista simile di pacchetti installati, ma usa un formato che può essere letto da `pip install`. Una convenzione molto usata è di collocare questa lista in un file `requirements.txt`:

```
(tutorial-env) $ pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

Il file `requirements.txt` può essere incluso nel controllo di versione e distribuito come parte dell'applicazione. Gli utenti possono poi usarlo per installare tutti i pacchetti necessari con `install -r`:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
```

(continua...)

(...segue)

```
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` ha molte altre opzioni. Consultate la guida a Installare moduli Python per la documentazione completa di `pip`. Se avete scritto un package Python e volete pubblicarlo sul Python Package Index, leggete la guida a Distribuire moduli Python.

CAPITOLO 13

E adesso?

Dopo aver letto questo tutorial il vostro interesse per Python sarà probabilmente cresciuto: siete ansiosi di usare Python in qualche progetto reale. Come dovreste procedere adesso, per saperne di più?

Questo tutorial fa parte della documentazione ufficiale di Python. Altre sezioni della documentazione sono:

- La libreria standard: sfogliate questo manuale, che fornisce una documentazione completa (anche se sintetica) per tipi, funzioni e moduli della libreria standard. La distribuzione di Python contiene *molte cose*. Ci sono moduli per leggere le caselle di posta Unix, ricevere documenti via HTTP, generare numeri casuali, leggere le opzioni della riga di comando, scrivere programmi grafici, comprimere dati e molto altro ancora. Un'occhiata veloce a questa documentazione dovrebbe bastare per farsi un'idea di che cosa è disponibile.
- Installare i moduli Python spiega come installare moduli aggiuntivi scritti da altri programmatori Python.
- La guida di riferimento del linguaggio è una spiegazione dettagliata della sintassi e della semantica di Python: è una lettura non facile, ma molto utile come guida del linguaggio in sé.

Altro materiale su Python:

- https://www.python.org: il sito web di Python. Contiene codice, documentazione e indicazioni per altri contenuti su Python in giro per il web. Questo sito ha dei *mirror* in vari posti nel mondo: Europa, Giappone, Australia etc. Il *mirror* potrebbe essere più veloce del sito principale, dipende dalla vostra posizione geografica.
- https://docs.python.org: accesso rapido alla documentazione di Python.
- https://pypi.org: il Python Package Index, un tempo chiamato anche «Cheese Shop»[1], è un indice di moduli creati da programmatori Python e disponibili per il download. Una volta che siete pronti a rilasciare il vostro codice, potete registrarvi qui, in modo che altri possano trovare il vostro lavoro.
- https://code.activestate.com/recipes/langs/python/: il «Python Cookbook» è una notevole raccolta di esempi, moduli più complessi e script utili. I contributi più interessanti sono raccolti nel libro omonimo (O'Reilly & Associates, ISBN 0-596-00797-3).
- http://www.pyvideo.org raccoglie video su Python, da convegni e incontri di *user-group*.
- https://scipy.org: il progetto «Scientific Python» comprende moduli per calcoli e manipolazioni performanti su array, oltre a moltissimi moduli di algebra lineare, trasformata di Fourier, risolutori non-lineari, distribuzioni di numeri casuali, analisi statistica e altro ancora.

[1] «Cheese Shop» è uno sketch dei Monty Python: un cliente entra in un negozio di formaggi ma qualunque formaggio desidera, il commesso risponde che ne è sprovvisto.

Per domande su Python e per segnalare problemi, potete postare nel newsgroup *comp.lang.python*, o alla mailing list python-list@python.org: sono collegati tra loro, quindi un messaggio spedito all'uno sarà automaticamente inoltrato all'altra. Ci sono centinaia di post ogni giorno, con domande (e riposte), suggerimenti per nuove funzionalità, annunci di nuovi progetti. Gli archivi sono disponibili su https://mail.python.org/pipermail/.

Prima di inviare un messaggio, assicuratevi di aver letto la lista di domande frequenti (FAQ). Qui trovano risposta le domande poste più spesso e potrebbe trovarsi anche la soluzione al vostro problema.

CAPITOLO 14

Editing e history substitution nell'input interattivo

Alcune versioni dell'interprete Python supportano l'editing dell'input corrente e la *history substitution*, in modo analogo alle shell Korn e GNU Bash. Questa possibilità è implementata con la libreria GNU Readline, che supporta diversi stili di editing e che ha una sua documentazione, che pertanto non ripetiamo in questa sede.

14.1 Tab completion e history editing

Il completamento delle variabili e dei nomi dei moduli è abilitato automaticamente all'avvio dell'interprete, così che il tasto `Tab` invoca la funzione di *completion* cercando tra i nomi delle istruzioni, le variabili locali e i nomi dei moduli disponibili. Le espressioni con il punto, come `string.a`, sono valutate fino al punto finale; quindi vengono suggeriti completamenti tratti dagli attributi dell'oggetto risultante. Si noti che così facendo è possibile che sia eseguito del codice dell'applicazione, se l'espressione comprende un oggetto con un metodo `__getattr__()` definito. La configurazione di default salva inoltre la storia dei comandi in un file `.python_history` nella vostra directory home. La storia sarà nuovamente disponibile nella prossima sessione dell'interprete.

14.2 Alternative all'interprete interattivo

Queste funzionalità costituiscono un grande passo avanti rispetto alle prime versioni dell'interprete. Tuttavia, alcune cose restano irrisolte: sarebbe utile presentare un rientro adeguato per le linee di continuazione, dal momento che l'interprete riconosce se il prossimo *token* richiede un rientro. Il meccanismo di completamento potrebbe utilizzare la tabella dei simboli dell'interprete. Sarebbe anche utile avere un comando per controllare o suggerire il bilanciamento delle parentesi, degli apici etc.

Un'alternativa potenziata per l'interprete interattivo, disponibile da parecchi anni, è IPython che supporta la *tab completion*, l'esplorazione degli oggetti e la gestione avanzata della storia dei comandi. Può anche essere personalizzato in molti aspetti e incorporato in altre applicazioni. Un ambiente di sviluppo simile è bpython.

CAPITOLO 15

Aritmetica in virgola mobile: problemi e limiti

I computer rappresentano i numeri in virgola mobile come frazioni binarie. Per esempio, la frazione decimale[1]

```
0.125
```

vale 1/10 + 2/100 + 5/1000, e allo stesso modo la frazione binaria

```
0.001
```

vale 0/2 + 0/4 + 1/8. Queste due frazioni hanno lo stesso valore: l'unica differenza è che una è espressa come frazione in base 10, l'altra come frazione in base 2.

Purtroppo molte frazioni decimali non possono essere rappresentate in modo esatto come frazioni binarie. Una conseguenza di ciò è che, in generale, i numeri con la virgola decimali che inserite possono essere rappresentati nel computer solo in modo approssimato come numeri binari con la virgola.

Il problema è più facile da capire in base 10. Si consideri la frazione 1/3. Si può approssimare in base 10:

```
0.3
```

o meglio,

```
0.33
```

o meglio,

```
0.333
```

e così via. Non importa quante cifre decimali si vogliono scrivere, il risultato non sarà mai esattamente 1/3, ma un'approssimazione sempre migliore di 1/3.

Allo stesso modo, non importa quante cifre si scrivono, ma il valore decimale 0.1 non può essere rappresentato esattamente in base 2. Il valore 1/10 in base 2 è la frazione periodica

```
0.0001100110011001100110011001100110011001100110011...
```

Se arrestiamo lo sviluppo a un numero qualsiasi di cifre, otteniamo un'approssimazione. Sui computer moderni i numeri con la virgola sono rappresentati con una frazione binaria dove il numeratore usa i primi 53 bit,

[1] ndT: i numeri «con la virgola» in Inglese (e in Python, e in qualsiasi linguaggio di programmazione) si scrivono naturalmente «con il punto». *Virgola mobile* in Inglese è *floating point*.

partendo dal bit più significativo, e il denominatore è una potenza di 2. Nel caso di 1/10, la frazione binaria è `3602879701896397 / 2 ** 55`: vicina, ma non esattamente uguale al vero valore di 1/10.

Molti utenti non si accorgono di questa approssimazione perché le cifre visualizzate non sono abbastanza. Python scrive un'approssimazione decimale del vero valore che internamente è rappresentato come un'approssimazione binaria. Sulla maggior parte dei computer, se Python dovesse scrivere il valore decimale esatto dell'approssimazione binaria interna di 0.1, dovrebbe farci vedere

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Sono molte più cifre di quelle di cui la gente ha bisogno. Python preferisce mantenere un numero gestibile di cifre decimali, visualizzando un valore approssimato:

```
>>> 1 / 10
0.1
```

Bisogna però ricordare che, anche se il risultato visualizzato assomiglia al valore esatto di 1/10, il valore *memorizzato* è quello della frazione binaria più prossima.

Un aspetto interessante è che ci sono molti numeri decimali che condividono la rappresentazione della stessa frazione binaria più prossima. Per esempio, i numeri `0.1` e `0.10000000000000001` e `0.1000000000000000055511151231257827021181583404541015625` sono tutti approssimati da `3602879701896397 / 2 ** 55`. Dal momento che tutti questi valori decimali condividono la stessa approssimazione, ciascuno di essi potrebbe essere visualizzato preservando l'invariante `eval(repr(x)) == x`.

Molti anni fa, il prompt di Python e la funzione predefinita `repr()` sceglievano tra le possibili rappresentazioni quella con 17 cifre significative, `0.10000000000000001`. A partire da Python 3.1, sulla maggior parte delle piattaforme Python è in grado di scegliere quella più breve e visualizza semplicemente `0.1`.

Si noti che questo comportamento è dovuto alla natura intrinseca dell'aritmetica binaria in virgola mobile: non è un baco di Python e non è neppure un baco nel vostro codice. Avreste lo stesso risultato con tutti i linguaggi che si appoggiano all'aritmetica in virgola mobile del vostro hardware (anche se un linguaggio potrebbe non *visualizzare* la differenza di default, oppure non in tutte le modalità di output).

Per ottenere un output più semplice, potete usare la formattazione delle stringhe per limitare il numero delle cifre significative:

```
>>> format(math.pi, '.12g')  # produce 12 cifre significative
'3.14159265359'

>>> format(math.pi, '.2f')   # produce 2 cifre significative dopo la virgola
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

È importante capire che, in un certo senso, si tratta di un'illusione: state semplicemente arrotondando la *visualizzazione* del vero valore conservato dal computer.

Un'illusione può portare a un'altra illusione. Per esempio, siccome 0.1 non è esattamente 1/10, sommare tre volte 0.1 potrebbe non dare 0.3:

```
>>> .1 + .1 + .1 == .3
False
```

Inoltre, siccome 0.1 non può avvicinarsi ulteriormente al valore esatto di 1/10 e 0.3 non può avvicinarsi di più a 3/10, arrotondare preventivamente con la funzione `round()` non è una soluzione:

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

Anche se i numeri non possono avvicinarsi di più al loro valore reale, la funzione `round()` può essere utile comunque per arrotondare *dopo*, in modo da rendere confrontabili i risultati approssimati:

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

L'aritmetica binaria in virgola mobile presenta molte sorprese come questa. Spieghiamo nel dettaglio il problema di «0.1» nella sezione successiva. Si veda The Perils of Floating Point per un elenco più completo di altri inciampi frequenti.

Come si usa concludere, «non ci sono risposte facili». Tuttavia non bisogna neppure avere troppa paura della virgola! Gli errori nelle operazioni decimali in Python sono ereditati dall'architettura in virgola mobile sottostante, e sulle macchine moderne questi sono dell'ordine di una parte su 2**53 per ciascuna operazione. È più che adeguato nella maggior parte dei casi, ma dovete tener presente che non si tratta di aritmetica decimale e che ciascuna nuova operazione può accumulare un nuovo errore di arrotondamento.

Anche se esistono dei casi estremi, nella vita di tutti i giorni l'aritmetica in virgola mobile si comporta come ci si aspetta, se si arrotonda semplicemente il risultato finale al numero di decimali che si desidera. Di solito basta la funzione `str()`; per un controllo più fine si può usare il metodo `str.format()` e la sua sintassi di formattazione.

Per gli scenari dove è richiesta una rappresentazione decimale esatta, potete usare il modulo `decimal`, che implementa l'aritmetica decimale adatta per la contabilità e i programmi che fanno calcoli di alta precisione.

Una forma alternativa di aritmetica esatta è quella del modulo `fractions`, che implementa l'aritmetica dei numeri razionali (così che numeri come 1/3 possano essere espressi in modo esatto).

Se fate un uso massiccio di operazioni in virgola mobile potreste voler considerare il pacchetto NumPy e i molti altri package di interesse matematico e statistico compresi nel progetto SciPy.

Python fornisce degli strumenti utili per le rare occasioni in cui davvero volete conoscere il valore esatto di un *float*. Il metodo `float.as_integer_ratio()` esprime il valore del numero sotto forma di frazione:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Siccome il rapporto è un valore esatto, può essere usato per ricreare il valore originario senza perdita di precisione:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

Il metodo `float.hex()` esprime il numero in notazione esadecimale (base 16), restituendo il valore esatto conservato nel computer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Anche questa rappresentazione esadecimale è precisa e può essere usata per ricostruire il numero originale:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Dal momento che questa rappresentazione è esatta, può essere usata per trasportare il valore in modo affidabile tra diverse versioni di Python (su diverse piattaforme) e per scambiare dati con altri linguaggi che supportano lo stesso formato (come Java e C99).

Un altro strumento utile è la funzione `math.fsum()`, che aiuta ad alleviare il problema della perdita di precisione durante la somma. Questa funzione tiene traccia dei «decimali perduti» man mano che i valori sono aggiunti al totale. Questo può fare la differenza nella precisione complessiva, evitando che gli errori si accumulino al punto di influenzare il risultato finale:

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

15.1 Errore di rappresentazione

Questa sezione spiega in dettaglio l'esempio di «0.1» visto sopra e mostra come eseguire un'analisi di casi del genere. Si assume che il lettore abbia una conoscenza di base della rappresentazione binaria in virgola mobile.

Con «errore di rappresentazione» si intende il fatto che alcune frazioni decimali (la maggior parte, in effetti) non possono essere rappresentate in modo esatto come frazioni binarie (in base 2). Questo è il motivo di fondo per cui Python (o Perl, C, C++, Java, Fortran e molti altri) talvolta non visualizzano esattamente il numero decimale che uno si aspetta.

Perché succede? 1/10 non può essere rappresentato come una frazione binaria. Quasi tutti i computer oggi (novembre 2000) usano l'aritmetica in virgola mobile IEEE-754 e in quasi tutte le piattaforme un *float* di Python è implementato come un numero «in doppia precisione» IEEE-754. Questi numeri hanno una precisione di 53 bit, quindi il computer in ingresso cerca di convertire 0.1 alla frazione più vicina che riesce a ottenere nella forma *J*/2***N* dove *J* è un intero che contiene esattamente 53 bit. Quindi, scrivendo

```
1 / 10 ~= J / (2**N)
```

come

```
J ~= 2**N / 10
```

e ricordando che *J* ha esattamente 53 bit (ovvero è `>= 2**52` ma `< 2**53`), il miglior valore per *N* è 56:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

Ovvero, 56 è l'unico valore di *N* che permette a *J* di avere esattamente 53 bit. Il miglior valore di *J* è di conseguenza il quoziente arrotondato:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Dal momento che il resto è maggiore della metà di 10, la migliore approssimazione si ottiene arrotondando verso l'alto:

```
>>> q+1
7205759403792794
```

Quindi la migliore approssimazione possibile di 1/10 come numero in doppia precisione IEEE-754 è:

```
7205759403792794 / 2 ** 56
```

Dividere numeratore e denominatore per due riduce la frazione a:

```
3602879701896397 / 2 ** 55
```

Si noti che, avendo arrotondato verso l'alto, questo numero è leggermente più grande di 1/10; se avessimo arrotondato verso il basso, sarebbe più piccolo. Comunque in nessun caso potrebbe essere *esattamente* 1/10.

Il computer quindi non «vede» mai 1/10: vede piuttosto la frazione esatta che abbiamo ricavato qui sopra, ovvero la migliore approssimazione IEEE-754 che può ottenere:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Se moltiplichiamo la frazione per 10**55, possiamo vedere il valore che si sviluppa per 55 cifre decimali:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

Questo vuol dire che il numero esatto conservato internamente è uguale al valore decimale 0.1000000000000000055511151231257827021181583404541015625. Invece di visualizzare il valore decimale per intero, molti linguaggi (incluse le vecchie versioni di Python) lo arrotondano a 17 cifre significative:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

I moduli `fractions` e `decimal` facilitano questi calcoli:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

CAPITOLO 16

Appendice

16.1 Modalità interattiva

16.1.1 Gestione degli errori

Quando si verifica un errore, l'interprete emette un messaggio di errore e uno *stack trace*. In modalità interattiva, ritorna quindi al prompt primario; se l'input arriva da un file, esce con *exit status* non-zero dopo aver visualizzato lo *stack trace*. (Le eccezioni gestite da una clausola `except` all'interno di un'istruzione `try` non sono «errori» in questo contesto.) Alcuni errori sono irrimediabilmente fatali e comportano l'uscita con *exit-status* non-zero: per esempio inconsistenze interne, o alcune situazioni di esaurimento della memoria disponibile. Tutti i messaggi di errore sono scritti nello *standard error*; l'output normale delle istruzioni eseguite è scritto nello *standard output*.

Inserire il carattere di interruzione (di solito `Control-C` o `Delete`) al prompt primario o secondario cancella l'input e ritorna al prompt primario.[1] Inserire un'interruzione mentre un'istruzione è in esecuzione emette l'eccezione `KeyboardInterrupt`, che può essere gestita da un'istruzione `try`.

16.1.2 Script Python eseguibili

Sui sistemi Unix/BSD, gli script Python possono essere resi direttamente eseguibili, come gli script della shell, con la riga

```
#!/usr/bin/env python3.5
```

all'inizio dello script (si assume che l'interprete sia nella `PATH` di sistema dell'utente) e dando al file modalità eseguibile. I caratteri `#!` devono essere esattamente all'inizio del file. Su alcune piattaforme, questa prima riga deve terminare con un «a-capo» in stile Unix (`'\n'`) e non Windows (`'\r\n'`). Si noti che il cancelletto `'#'` viene usato in Python per iniziare un commento.

Si può dare al file dello script modalità eseguibile con il comando **`chmod`**.

```
$ chmod +x myscript.py
```

Su Windows non esiste la nozione di «modalità eseguibile». L'installazione di Python associa automaticamente le estensioni dei file `.py` con `python.exe`, in modo che fare doppio clic sul file lo esegue come script. L'estensione può anche essere `.pyw`: in questo caso la finestra della console che appare normalmente non viene mostrata.

[1] Un problema della libreria GNU Readline potrebbe impedirlo.

16.1.3 Il file di avvio interattivo

Quando usate Python interattivamente, può far comodo che alcuni comandi standard siano eseguiti automaticamente ogni volta che l'interprete viene avviato. Questo si può fare creando una variabile d'ambiente `PYTHONSTARTUP` che contiene il nome di un file con i vostri comandi di avvio. È simile a un file `.profile` per le shell di Unix.

Questo file viene preso in considerazione solo per le sessioni interattive, non quando Python legge l'input da uno script, e non quando `/dev/tty` è indicato esplicitamente come sorgente per le istruzioni (altrimenti il terminale si comporta come una normale sessione interattiva). Il file è eseguito nello stesso *namespace* dei comandi interattivi, quindi gli oggetti che definisce possono essere importati come nomi non qualificati nella sessione dell'interprete. In questo file potete anche cambiare i prompt `sys.ps1` e `sys.ps2`.

Se volete leggere un file di avvio aggiuntivo nella directory corrente, potete farlo nel file di avvio principale con del codice come `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. Se volete usare il file di avvio in uno script, dovete farlo in modo esplicito nello script:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4 Personalizzare l'installazione

Python mette a disposizione due strumenti che vi consentono di personalizzarlo: i moduli `sitecustomize` e `usercustomize`. Per vederli in azione, dovete per prima cosa ricavare la collocazione della vostra directory *site-packages*. Avviate Python ed eseguite questo codice:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.5/site-packages'
```

Adesso potete creare un file `usercustomize.py` in questa directory e collocarvi qualsiasi istruzione. Questo avrà effetto su qualsiasi invocazione di Python, a meno che non venga passata l'opzione `-s` per disabilitarne l'importazione automatica.

Il modulo `sitecustomize` funziona allo stesso modo, ma viene creato di solito da un amministratore del computer nella directory *site-packages* globale, ed è importato *prima* di `usercustomize`. Si veda la documentazione del modulo `site` per ulteriori informazioni.

Indice

Simboli

A

B

C

D

F

H

J

M

N

O

P

R

S

V