
Python Tutorial

Release 3.13.1

Guido van Rossum and the Python development team

gennaio 21, 2025

Python Software Foundation
Email: docs@python.org

1	Stuzzichiamo il tuo appetito	3
2	Uso dell'interprete di Python	5
2.1	Invocazione dell'interprete	5
2.1.1	Passare Argomenti	6
2.1.2	Modalità Interattiva	6
2.2	L'interprete e il suo ambiente	6
2.2.1	Codifica del codice sorgente	6
3	Un'introduzione informale a Python	9
3.1	Usare Python come calcolatrice	9
3.1.1	Numeri	9
3.1.2	Testo	11
3.1.3	Liste	15
3.2	Primi passi di programmazione	16
4	Altri Strumenti di Controllo del Flusso	19
4.1	Istruzioni <code>if</code>	19
4.2	Istruzioni <code>for</code>	19
4.3	La Funzione <code>range()</code>	20
4.4	<code>break</code> and <code>continue</code> Statements	21
4.5	<code>else</code> Clauses on Loops	22
4.6	Istruzioni <code>pass</code>	22
4.7	Istruzioni <code>match</code>	23
4.8	Definizione di Funzioni	26
4.9	Ulteriori informazioni sulla Definizione di Funzioni	27
4.9.1	Valori predefiniti degli argomenti	27
4.9.2	Argomenti chiave-valore	28
4.9.3	Parametri speciali	30
4.9.4	Liste di Argomenti Arbitrarie	32
4.9.5	Spacchettamento delle Liste di Argomenti	33
4.9.6	Espressioni Lambda	33
4.9.7	Stringhe di Documentazione	33
4.9.8	Annotationi delle Funzioni	34
4.10	Intermezzo: Stile di Codifica	34
5	Strutture Dati	37
5.1	Più sulle Liste	37
5.1.1	Usare le Liste come Pile	38
5.1.2	Usare le Liste come Code	39
5.1.3	Comprensioni di Lista	39

5.1.4	Comprensioni di Lista Nidificate	41
5.2	L'istruzione <code>del</code>	41
5.3	Tuple e Sequenze	42
5.4	Insiemi	43
5.5	Dizionari	44
5.6	Tecniche di Looping	45
5.7	Maggiori informazioni sulle Condizioni	46
5.8	Confronto tra Sequenze e Altri Tipi	47
6	Moduli	49
6.1	Più sui Moduli	50
6.1.1	Esecuzione di moduli come script	51
6.1.2	Il percorso di ricerca del modulo	51
6.1.3	File Python «compilati»	52
6.2	Moduli Standard	52
6.3	La funzione <code>dir()</code>	53
6.4	Pacchetti	54
6.4.1	Importare * Da un Pacchetto	56
6.4.2	Riferimenti Intra-pacchetto	57
6.4.3	Pacchetti in Directory Multiple	57
7	Input e Output	59
7.1	Formattazione Avanzata dell'Output	59
7.1.1	Stringhe Formattate Letterali	60
7.1.2	Il Metodo String <code>format()</code>	61
7.1.3	Formattazione Manuale delle Stringhe	62
7.1.4	Vecchia formattazione delle stringhe	63
7.2	Lettura e Scrittura di File	63
7.2.1	Metodi degli Oggetti File	64
7.2.2	Salvare dati strutturati con <code>json</code>	65
8	Errori ed Eccezioni	67
8.1	Errori di Sintassi	67
8.2	Eccezioni	67
8.3	Gestione delle Eccezioni	68
8.4	Sollevere Eccezioni	70
8.5	Collegamento delle Eccezioni	71
8.6	Eccezioni Definite dall'Utente	72
8.7	Definizione di Azioni di Pulizia	72
8.8	Azioni di Pulizia Predefinite	74
8.9	Sollevere e Gestire Eccezioni Multiple e Non Correlate	74
8.10	Arricchire le Eccezioni con Note	76
9	Classi	79
9.1	Una parola su nomi e oggetti	79
9.2	Visibilità e spazi dei nomi in Python	80
9.2.1	Esempio di visibilità e spazi dei nomi	81
9.3	Una prima occhiata alle classi	82
9.3.1	Sintassi della definizione di classe	82
9.3.2	Oggetti della Classe	82
9.3.3	Oggetti Istanza	83
9.3.4	Oggetti Metodo	83
9.3.5	Variabili di Classe e Istanza	84
9.4	Osservazioni Varie	85
9.5	Ereditarietà	86
9.5.1	Ereditarietà Multipla	87
9.6	Variabili Private	88
9.7	Varie ed Eventuali	89
9.8	Iteratori	89

9.9	Generatori	90
9.10	Espressioni di Generatore	91
10	Breve Panoramica della Libreria Standard	93
10.1	Interfaccia del Sistema Operativo	93
10.2	Metacaratteri nei Nomfile	93
10.3	Argomenti dalla Linea di Comando	94
10.4	Reindirizzamento dell'Uscita degli Errori e Terminazione del Programma	94
10.5	Pattern Matching di Stringhe	94
10.6	Matematica	95
10.7	Accesso a Internet	95
10.8	Date e Orari	96
10.9	Compressione dei Dati	96
10.10	Misurazione delle Prestazioni	96
10.11	Controllo della Qualità	97
10.12	Batterie Incluse	97
11	Giro Breve della Libreria Standard — Parte II	99
11.1	Formattazione dell'Output	99
11.2	Templating	100
11.3	Lavorare con Layout Binari dei Dati	101
11.4	Multi-threading	101
11.5	Logging	102
11.6	Riferimenti Deboli	103
11.7	Strumenti per Lavorare con Liste	103
11.8	Decimal Floating-Point Arithmetic	104
12	Ambienti Virtuali e Pacchetti	107
12.1	Introduzione	107
12.2	Creazione di Ambienti Virtuali	107
12.3	Gestione dei Pacchetti con pip	108
13	E adesso?	111
14	Editing dell'input interattivo e sostituzione della history	113
14.1	Completamento con il tasto Tab e modifica della history	113
14.2	Alternative all'interprete interattivo	113
15	Floating-Point Arithmetic: Issues and Limitations	115
15.1	Errore di Rappresentazione	118
16	Appendice	121
16.1	Modalità Interattiva	121
16.1.1	Gestione degli Errori	121
16.1.2	Script Eseguibili Python	121
16.1.3	Il File di Avvio Interattivo	122
16.1.4	I Moduli di Personalizzazione	122
A	Glossary	125
B	About this documentation	143
B.1	Contributors to the Python documentation	143
C	Storia e licenza	145
C.1	Storia del software	145
C.2	Termini e condizioni di accesso o di utilizzo di Python	146
C.2.1	PSF ACCORDO DI LICENZA PER PYTHON 3.13.1	146
C.2.2	CONTRATTO DI LICENZA DI BEOPEN.COM PER PYTHON 2.0	147
C.2.3	CNRI CONTRATTO DI LICENZA PER PYTHON 1.6.1	147
C.2.4	CWI CONTRATTO DI LICENZA PER PYTHON DA 0.9.0 A 1.2	148

C.2.5	ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.1 DOCUMENTATION	149
C.3	Licenze e riconoscimenti per il software incorporato	149
C.3.1	Mersenne Twister	149
C.3.2	Socket	150
C.3.3	Servizi di socket asincrone	151
C.3.4	Gestione dei cookie	151
C.3.5	Tracciabilità dell'esecuzione	152
C.3.6	Funzioni UUencode e UUdecode	152
C.3.7	Chiamate di procedura remota XML	153
C.3.8	test_epoll	153
C.3.9	Select kqueue	154
C.3.10	SipHash24	154
C.3.11	strtod e dtoa	155
C.3.12	7.4 OpenSSL	155
C.3.13	expat	158
C.3.14	libffi	159
C.3.15	zlib	159
C.3.16	cfuhash	160
C.3.17	libmpdec	160
C.3.18	W3C C14N test suite	161
C.3.19	mimalloc	162
C.3.20	asyncio	162
C.3.21	Global Unbounded Sequences (GUS)	162
D	Copyright	165
	Indice	167

Python è un linguaggio di programmazione potente e facile da imparare. Dispone di strutture di dati efficienti e di alto livello e di un approccio semplice ma efficace alla programmazione orientata agli oggetti. La sintassi elegante e la tipizzazione dinamica di Python, insieme alla sua natura interpretata, ne fanno un linguaggio ideale per lo scripting e il rapido sviluppo di applicazioni in molte aree per la maggior parte delle piattaforme.

L'interprete Python e l'ampia libreria standard sono liberamente disponibili in forma di codici sorgenti o binari per tutte le principali piattaforme dal sito web di Python, <https://www.python.org/>, e possono essere distribuiti gratuitamente. Lo stesso sito contiene anche distribuzioni e link a molti moduli, programmi e strumenti Python di terze parti gratuiti e documentazione aggiuntiva.

L'interprete Python è facilmente ampliabile con nuove funzioni e tipi di dati implementati in C o C++ (o altri linguaggi richiamabili da C). Python è adatto anche come linguaggio di estensione per applicazioni personalizzabili.

Questo tutorial introduce il lettore in modo informale ai concetti di base e alle caratteristiche del linguaggio e del sistema Python. Aiuta ad avere un interprete Python a portata di mano per un'esperienza pratica, ma tutti gli esempi sono autonomi, così il tutorial può essere letto anche off-line.

Per una descrizione degli oggetti e dei moduli standard, si veda `library-index`. `reference-index` fornisce una definizione più formale del linguaggio. Per scrivere le estensioni in C o C++, si legga `extending-index` e `c-api-index`. Ci sono anche diversi libri che coprono Python in profondità.

Questo tutorial non vuole essere completo e coprire ogni singola caratteristica, come pure ogni caratteristica usata comunemente. Invece, introduce molte delle caratteristiche più importanti di Python, e vi darà una buona idea sullo stile del linguaggio. Dopo averlo letto, sarete in grado di leggere e scrivere moduli e programmi Python, e sarete pronti per saperne di più sui vari moduli della libreria Python, i quali sono descritti in `library-index`.

Anche il *[Glossary](#)* vale la pena di essere sfogliato.

Stuzzichiamo il tuo appetito

Se si fa molto lavoro sui computer, alla fine si scopre che c'è qualche compito che si desidera automatizzare. Ad esempio, è possibile eseguire un “ricerca e sostituisci” su un gran numero di file di testo, oppure rinominare e riorganizzare un gruppo di file di foto in modo complesso. Magari vi piacerebbe scrivere un piccolo database personalizzato, o un'applicazione GUI specializzata, o un semplice gioco.

Se sei uno sviluppatore di software professionista, potresti dover lavorare con diverse librerie C/C++/Java, ma il solito ciclo di scrittura/compilazione/test/ricompilazione è troppo lento. Forse stai scrivendo una suite di test per una libreria di questo tipo e credi che la scrittura del codice di test sia un compito noioso. O forse avete scritto un programma che potrebbe utilizzare una libreria che usa un proprio linguaggio di programmazione, e non volete progettare e implementare un linguaggio completamente nuovo per la vostra applicazione.

Python è il linguaggio che fa per te.

Si potrebbe scrivere uno script di shell Unix o file batch di Windows per alcune di queste attività, tuttavia se è vero che gli script di shell sono i migliori per spostarsi tra i file o modificare i file di testo, non sono adatti per applicazioni GUI o giochi. Si potrebbe scrivere un programma C/C++/Java, ma può essere necessario molto tempo di sviluppo per ottenere anche solo una prima bozza di programma. Python è più semplice da usare, disponibile su Windows, Mac OS X e Unix, e vi aiuterà a svolgere il lavoro più rapidamente.

Python è semplice da usare, ma è un vero e proprio linguaggio di programmazione, che offre una struttura e un supporto per programmi di grandi dimensioni molto più di quanto possano offrire script di shell o file batch. D'altra parte, Python offre anche molto più controllo degli errori di C e, essendo un linguaggio *di altissimo livello*, ha tipi di dati di alto livello integrati, come array flessibili e dizionari. A causa dei suoi tipi di dati più generali, Python è applicabile ad un dominio molto più grande di Awk o anche di Perl, ma molte cose sono almeno altrettanto facili in Python come in quei linguaggi.

Python permette di suddividere il programma in moduli che possono essere riutilizzati in altri programmi Python. Viene fornito con una vasta collezione di moduli standard che potete usare come base dei vostri programmi — o come esempi per iniziare ad imparare a programmare in Python. Alcuni di questi moduli forniscono cose come I/O di file, chiamate di sistema, socket e persino strumenti per lo sviluppo di interfacce grafiche utente come Tk.

Python è un linguaggio interpretato, che può farvi risparmiare molto tempo durante lo sviluppo del programma perché non è necessaria alcuna compilazione e *linking*. L'interprete può essere usato in modo interattivo, il che rende facile sperimentare con le caratteristiche del linguaggio, scrivere programmi usa e getta o testare le funzioni durante lo sviluppo del programma dal basso verso l'alto. È anche una pratica calcolatrice da tavolo.

Python consente di scrivere programmi in modo compatto e leggibile. I programmi scritti in Python sono in genere molto più brevi dei programmi equivalenti in C, C++ o Java, per diversi motivi:

- i tipi di dati di alto livello consentono di esprimere operazioni complesse in un'unica dichiarazione;

- il raggruppamento delle istruzioni è fatto usando l'indentazione invece che per parentesi iniziali e finali;
- non sono necessarie dichiarazioni di variabili o argomenti.

Python è *estensibile*: se si conosce la programmazione in C è facile aggiungere una nuova funzione o un modulo all'interprete, sia per eseguire operazioni critiche alla massima velocità, sia per collegare programmi Python a librerie che possono essere disponibili solo in forma di eseguibili binari (come una libreria grafica fornita da un fornitore). Una volta che ci si è fatto il callo, è possibile anche collegare l'interprete Python ad un'applicazione scritta in C e usarlo come un'estensione o come linguaggio per impartire comandi a quell'applicazione.

A proposito, la lingua prende il nome dallo show della BBC «Monty Python's Flying Circus» e non ha nulla a che fare con i rettili. Fare riferimento agli schetch Monty Python nella documentazione non solo è permesso, ma è incoraggiato!

Ora che siete tutti entusiasti di Python, vorrete esaminarlo più in dettaglio. Poiché il modo migliore per imparare un linguaggio è usarlo, il tutorial vi invita a giocare con l'interprete Python mentre lo leggete.

Nel capitolo successivo vengono spiegate le modalità di utilizzo dell'interprete. Si tratta di informazioni piuttosto banali, ma essenziali per provare gli esempi mostrati in seguito.

Il resto del tutorial introduce varie caratteristiche del linguaggio e del sistema Python attraverso esempi, iniziando con semplici espressioni, comandi e tipi di dati, attraverso funzioni e moduli, e infine toccando concetti avanzati come eccezioni e classi definite dall'utente.

Uso dell'interprete di Python

2.1 Invocazione dell'interprete

L'interprete Python è solitamente installato come `/usr/local/bin/python3.13` su quelle macchine dove è disponibile; aggiungere `/usr/local/bin` al path di ricerca della vostra shell Unix rende possibile avviarlo digitando il comando:

```
python3.13
```

alla shell.¹ Poiché la scelta della cartella in cui è situato l'interprete è un'opzione di installazione, è possibile trovarlo anche in altri percorsi; verificate con il vostro guru Python o con l'amministratore di sistema. (Ad esempio, `/usr/local/python` è un'alternativa popolare.)

Sulle macchine Windows dove hai installato Python dallo Microsoft Store, il comando `python3.13` sarà disponibile. Se hai installato il `py.exe` launcher, puoi utilizzare il comando `py`. Consulta `setting-envvars` per altri modi di avviare Python.

Digitando un carattere di fine file (`Control-D` su Unix, `Control-Z` su Windows) nel prompt, l'interprete esce con stato zero. Se non funziona, è possibile uscire dall'interprete digitando il seguente comando: `quit()`.

Le funzioni di editing di linea dell'interprete includono l'editing interattivo, la sostituzione della *history* e il completamento del codice su sistemi che supportano la libreria [GNU Readline](#). Forse il controllo più veloce per vedere se la modifica da riga di comando è supportata è digitando `Control-P` al primo prompt di Python che si ottiene. Se emette un segnale acustico, si ha una modifica da riga di comando; vedere [Appendix Editing dell'input interattivo e sostituzione della history](#) per un'introduzione alle combinazioni di tasti. Se non vi sembra accada nulla, o se vedete la stringa `^P`, la modifica a riga di comando non è disponibile; potrai usare solo il *backspace* per rimuovere i caratteri dalla riga corrente.

L'interprete opera in maniera simile alla shell Unix: quando viene chiamato con lo *standard input* collegato ad un dispositivo tty, legge ed esegue i comandi in modo interattivo; quando viene chiamato con un nome di file come argomento o con un file come *standard input*, legge ed esegue un *script* da quel file.

Un secondo modo di iniziare l'interprete è `python -c command [arg] ...`, che esegue le istruzioni in *command*, analogamente all'opzione `-c`. Poiché le istruzioni di Python spesso contengono spazi o altri caratteri speciali per la shell, di solito si consiglia di racchiudere *command*, nella sua interezza, con apici singoli.

¹ Su Unix, l'interprete Python 3.x non è installato di default con l'eseguibile chiamato `python`, in modo che non sia in conflitto con un eseguibile Python 2.x installato contemporaneamente.

Alcuni moduli Python sono utili anche come script. Questi possono essere invocati usando `python -m module [arg] ...`, che esegue il file sorgente del *modulo* come se tu avessi scritto il suo nome completo sulla riga di comando.

Quando si utilizza un file di script, a volte è utile essere in grado di eseguire lo script ed entrare in modalità interattiva in seguito. Questo può essere fatto passando `-i` prima dello script.

Tutte le opzioni della riga di comando sono descritte in [using-on-general](#).

2.1.1 Passare Argomenti

Una volta resi noti all'interprete, il nome dello script e gli argomenti aggiuntivi vengono poi trasformati in una lista di stringhe e assegnati alla variabile `argv` nel modulo `sys`. È possibile accedere a questa lista eseguendo `import sys`. La lunghezza della lista è almeno una; quando non viene dato alcuno script e nessun argomento, `sys.argv[0]` è una stringa vuota. Quando il nome dello script è dato come `'-'` (che significa ingresso standard), `sys.argv[0]` è settato a `'-'`. Quando è usato il comando `-c`, `sys.argv[0]` viene settato a `'-c'`. Quando viene usato il modulo `-m`, `sys.argv[0]` viene impostato con il nome completo del modulo. Tutte le opzioni trovate dopo i comandi `-c` o i moduli `-m` non vengono consumati dall'interprete di Python che processa le opzioni, ma vengono lasciate in `sys.argv` in modo che vengano poi gestite manualmente dal comando o dal modulo.

2.1.2 Modalità Interattiva

Quando i comandi vengono letti da una tty, l'interprete è detto essere in *modo interattivo*. In questa modalità richiede il comando successivo con il *prompt*, di solito tre simboli di *maggior di* (`>>>`); per le linee di continuazione viene emesso un *prompt* secondario, che per impostazione predefinita consiste di tre punti (`. . .`). L'interprete stampa un messaggio di benvenuto indicando il numero di versione e un avviso di copyright prima di stampare il *prompt*:

```
$ python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Le linee di continuazione sono necessarie quando si inserisce un costrutto su più linee. Come esempio, date un'occhiata a questa dichiarazione `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, vedere [Modalità Interattiva](#).

2.2 L'interprete e il suo ambiente

2.2.1 Codifica del codice sorgente

Per impostazione predefinita, i file sorgente di Python sono trattati come codificati in UTF-8. In questa codifica, i caratteri della maggior parte delle lingue del mondo possono essere usati simultaneamente in stringhe, variabili e commenti — la libreria standard tuttavia usa solo caratteri ASCII per le variabili, una convenzione che ogni codice portatile dovrebbe seguire. Per visualizzare correttamente tutti questi caratteri, l'editor deve riconoscere che il file è UTF-8, e deve utilizzare un font che supporti tutti i caratteri del file.

Per dichiarare una codifica diversa da quella predefinita, è necessario aggiungere un commento speciale come *prima* riga del file. La sintassi è la seguente:

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

dove *codifica* è uno dei `codecs` validi supportati da Python.

Per esempio, per dichiarare che deve essere usata la codifica Windows-1252, la prima riga del file del codice sorgente dovrebbe essere:

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

Un'eccezione alla regola della *prima linea* è quando il codice sorgente inizia con una linea UNIX “*shebang*” *line*. In questo caso, la dichiarazione della codifica dovrebbe essere aggiunta come seconda riga del file. Per esempio:

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

Note

Un'introduzione informale a Python

Nei seguenti esempi, input e output si distinguono per la presenza o meno del prompt (`>>>` e `...`): per ripetere l'esempio, è necessario digitare tutto dopo il prompt, quando questo è presente; le righe che non iniziano con un prompt vengono emesse dall'interprete. Si noti che un prompt secondario su una linea da solo in un esempio significa che è necessario digitare una riga vuota; questo viene utilizzato per terminare un comando a più righe.

Molti degli esempi di questo manuale, anche quelli inseriti al prompt interattivo, includono commenti. I commenti in Python iniziano con il carattere *hash*, #, e si estendono fino alla fine della riga. Un commento può apparire all'inizio di una riga o dopo uno spazio bianco o codice, ma non all'interno di una stringa. Un carattere hash all'interno di una stringa letterale è solo un carattere. Poiché i commenti servono solo a chiarire il codice e non sono interpretati da Python, possono essere omessi quando si scrivono gli esempi.

Alcuni esempi:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1 Usare Python come calcolatrice

Proviamo con alcuni semplici comandi di Python. Avviare l'interprete e attendere il prompt primario, `>>>`. (Non dovrebbe volerci molto.)

3.1.1 Numeri

L'interprete agisce come una semplice calcolatrice: è possibile digitare un'espressione per scrivere il valore. La sintassi dell'espressione è semplice: gli operatori `+`, `-`, `*` e `/` funzionano come nella maggior parte degli altri linguaggi (per esempio, Pascal o C); le parentesi `()` possono essere utilizzate per il raggruppamento. Per esempio:

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

(continues on next page)

(continua dalla pagina precedente)

```
>>> 8 / 5 # division always returns a floating-point number
1.6
```

I numeri interi (es. 2, 4, 20) hanno tipo `int`, quelli in virgola mobile (es. 5.0, 1.6) hanno tipo `float`. Torneremo a parlare ancora dei tipi numerici più avanti nel tutorial.

La divisione (`/`) restituisce sempre un `float`. Per fare *floor division* e ottenere un risultato intero (scartando qualsiasi cifra decimale) si può usare l'operatore `//`; per calcolare il resto si può usare `%`:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3 # floor division discards the fractional part
5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # floored quotient * divisor + remainder
17
```

Con Python, è possibile utilizzare l'operatore `**` per calcolare le potenze¹:

```
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

Il segno uguale (`=`) è usato per assegnare un valore ad una variabile. Successivamente, nessun risultato viene visualizzato prima della successiva richiesta:

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

Se una variabile non è stata "definita" (a cui è stato assegnato un valore), usarla vi darà un errore:

```
>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

C'è pieno supporto per i numeri in virgola mobile; gli operatori con operandi di tipo misto convertono l'intero in un numero in virgola mobile:

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

In modalità interattiva, l'ultima espressione stampata viene assegnata alla variabile `_`. Questo significa che quando si utilizza Python come calcolatrice da tavolo, è un po' più facile continuare i calcoli, ad esempio:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
```

(continues on next page)

¹ Poiché `**` ha una precedenza superiore a `-`, `-3**2` sarà interpretato come `-(3**2)` e quindi risulterà in `-9`. Per evitare questo e ottenere `9`, è possibile utilizzare `(-3)**2`.

(continua dalla pagina precedente)

```
>>> round(_, 2)
113.06
```

Questa variabile deve essere trattata dall'utente in sola lettura. Non va assegnato esplicitamente un valore ad essa — si dovrebbe invece creare una variabile locale indipendente con lo stesso nome, mascherando così la variabile *built-in* e il suo comportamento magico.

Oltre a `int` e `float`, Python supporta altri tipi di numeri, come `Decimal` e `Fraction`. Python ha anche il supporto incorporato per complex numbers, e usa il suffisso `j` o `J` per indicare la parte immaginaria (es. `3+5j`).

3.1.2 Testo

Python può manipolare testo (rappresentato dal tipo `str`, le cosiddette «stringhe») così come numeri. Questo include caratteri «!», parole «rabbit», nomi «Paris», frasi «Got your back.», ecc. «Yay! :)». Possono essere racchiusi tra virgolette singole (`'...'`) o virgolette doppie (`"..."`) con lo stesso risultato².

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!" # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975' # digits and numerals enclosed in quotes are also strings
'1975'
```

racchiudere una virgoletta dello stesso tipo, dobbiamo farne l'«escape», precedendola con `\`. In alternativa, possiamo usare l'altro tipo di virgolette:

```
>>> 'doesn\'t' # use \' to escape the single quote...
"doesn't"
>>> "doesn't" # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> 'Isn\'t," they said.'
'Isn\'t," they said.'
```

Nella shell di Python, la definizione di una stringa e la stringa di output possono apparire differenti. La funzione `print()` produce un output più leggibile, omettendo le virgolette di chiusura e stampando i caratteri con escape e speciali:

```
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s) # with print(), special characters are interpreted, so \n produces
↪new line
First line.
Second line.
```

Se non si desidera che i caratteri preceduti da `\` siano interpretati come caratteri speciali, è possibile utilizzare le cosiddette *raw strings* aggiungendo un `r` prima del primo apice:

```
>>> print('C:\some\name') # here \n means newline!
C:\some
```

(continues on next page)

² A differenza di altri linguaggi, i caratteri speciali come `\n` hanno lo stesso significato sia con apici singoli (`'...'`) che doppi (`"..."`). L'unica differenza tra i due è che all'interno delle stringhe ad apici singoli non c'è bisogno di aggiungere il carattere di escaping prima del `"` (ma bisogna inserirlo prima di un `\`) e viceversa.

(continua dalla pagina precedente)

```
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

C'è un aspetto sottile nelle stringhe raw: una stringa raw non può terminare con un numero dispari di caratteri \; vedi la voce FAQ per ulteriori informazioni e soluzioni.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End of lines are automatically included in the string, but it's possible to prevent this by adding a `\` at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...     -h                        Display this usage message
...     -H hostname              Hostname to connect to
... """)
Usage: thingy [OPTIONS]
    -h                        Display this usage message
    -H hostname              Hostname to connect to
>>>
```

Le stringhe possono essere concatenate (incollate insieme) con l'operatore `+` e ripetute con `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Due o più *literal di tipo stringa* (cioè quelli racchiusi tra virgolette) posizionati uno accanto all'altro sono automaticamente concatenati.

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

Questa funzione è particolarmente utile quando si vogliono separare stringhe lunghe:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

Questo funziona solo con due literal, ma non con variabili o *expression*:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
File "<stdin>", line 1
    prefix 'thon'
    ^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
File "<stdin>", line 1
    ('un' * 3) 'ium'
    ^^^^^
SyntaxError: invalid syntax
```

Se volete concatenare variabili o una variabile e un letterale, usate `+`:

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

Le stringhe possono essere *indicizzate* (sottoscritte), e il primo carattere ha indice 0. Non esiste un tipo carattere specifico; un carattere è semplicemente una stringa di dimensione uno:

```
>>> word = 'Python'
>>> word[0]    # character in position 0
'P'
>>> word[5]    # character in position 5
'n'
```

Gli indici possono anche essere numeri negativi, per iniziare a contare da destra:

```
>>> word[-1]   # last character
'n'
>>> word[-2]   # second-last character
'o'
>>> word[-6]
'P'
```

N.B. Poiché -0 è uguale a 0, gli indici negativi partono da -1.

Oltre all'indicizzazione, è supportato anche lo *slicing*. Mentre l'indicizzazione è usata per ottenere i singoli caratteri, lo *slicing* permette di ottenere una sottostringa:

```
>>> word[0:2]   # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]   # characters from position 2 (included) to 5 (excluded)
'tho'
```

Nello slice gli indici hanno degli utili valori predefiniti; un primo indice omissso è zero, un secondo indice omissso è uguale alla dimensione della stringa che si sta tagliando:

```
>>> word[:2]    # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]    # characters from position 4 (included) to the end
'on'
>>> word[-2:]   # characters from the second-last (included) to the end
'on'
```

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

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

Un modo per ricordare come funzionano le fette è quello di pensare agli indici come al punto *tra* caratteri, con il bordo sinistro del primo carattere numerato 0. Poi il bordo destro dell'ultimo carattere di una stringa di caratteri *n* ha l'indice *n*, per esempio:

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

La prima riga di numeri dà la posizione degli indici 0...6 nella stringa; la seconda riga dà i corrispondenti indici negativi. La porzione da *i* a *j* è costituita da tutti i caratteri tra i bordi contrassegnati rispettivamente con *i* e *j*.

Per gli indici non negativi, la lunghezza di una fetta è la differenza degli indici, se entrambi sono entro i limiti. Per esempio, la lunghezza di `word[1:3]` è 2.

Il tentativo di utilizzare un indice troppo grande comporta un errore:

```
>>> word[42] # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Tuttavia, gli indici delle sottostringhe fuori *range* (che superano la lunghezza della stringa) sono gestiti a modo quando vengono utilizzati per l'affettamento:

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

Le stringhe Python non possono essere modificate — sono *immutable*. Pertanto, l'assegnazione di una posizione indicizzata nella stringa comporta 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 avete bisogno di una stringa diversa, dovrete crearne una nuova:

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

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

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

Vedi anche

textseq

Le stringhe sono esempi di *tipi di sequenze*, e supportano le operazioni comuni supportate da questi tipi di dato.

string-methods

Un gran numero di metodi per le trasformazioni di base e la ricerca sono supportati dalle stringhe.

f-strings

Stringhe con *expression* incorporate.

formatstrings

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

old-string-formatting

Il vecchio metodo di formattazione, quello che prevede un template a sinistra dell'operatore % è descritto più dettagliatamente qui.

3.1.3 Liste

Python conosce un certo numero di tipi di dati *composti*, usati per raggruppare altri valori. La più versatile è la *lista*, che può essere scritta come una lista di valori separati da virgola (elementi) tra parentesi quadre. Gli elenchi possono contenere elementi di tipo diverso, ma di solito gli elementi hanno tutti lo stesso tipo.

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

Come le stringhe (e tutti gli altri tipi built-in *sequence*), le liste possono essere indicizzate e tagliate:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:] # slicing returns a new list
[9, 16, 25]
```

Gli elenchi supportano anche operazioni come la concatenazione:

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

A differenza delle stringhe, che sono *immutable*, le liste sono di tipo *mutable*, cioè è possibile modificarne il contenuto:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

Potete anche aggiungere nuovi elementi alla fine della lista, usando il *metodo* `list.append()` (vedremo di più riguardo i metodi più avanti):

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

L'assegnazione semplice in Python non copia mai i dati. Quando assegna una lista a una variabile, la variabile si riferisce alla *lista esistente*. Qualsiasi modifica apportata alla lista tramite una variabile sarà visibile attraverso tutte le altre variabili che si riferiscono ad essa.:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba) # they reference the same object
True
>>> rgba.append("Alpha")
>>> rgb
["Red", "Green", "Blue", "Alpha"]
```

Tutte le operazioni di taglio restituiscono una nuova lista contenente gli elementi richiesti. Ciò significa che la seguente sezione restituisce una nuova shallow copy della list:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
["Red", "Green", "Blue", "Alpha"]
>>> rgba
["Red", "Green", "Blue", "Alph"]
```

L'assegnazione a uno slice è anche possibile, per cambiare anche la dimensione della lista o cancellarne gli elementi completamente:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

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

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

E' possibile nidificare liste (creare liste contenenti altre liste), ad 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 Primi passi di programmazione

Naturalmente, possiamo usare Python per compiti più complicati che sommare insieme due più due. Per esempio, possiamo scrivere una prima sottosequenza della [Successione di Fibonacci](#) come segue:

```
>>> # Fibonacci series:
>>> # the sum of two elements defines the next
>>> a, b = 0, 1
>>> while a < 10:
...     print(a)
```

(continues on next page)

(continua dalla pagina precedente)

```

...     a, b = b, a+b
...
0
1
1
2
3
5
8

```

Questo esempio introduce diverse nuove funzionalità.

- La prima riga contiene una *assegnazione multipla*: le variabili `a` e `b` ottengono simultaneamente i nuovi valori 0 e 1. Sull'ultima riga viene usata di nuovo, dimostrando che le espressioni sul lato destro sono tutte valutate prima di una qualsiasi delle assegnazioni. Le espressioni del lato destro sono valutate da sinistra a destra.
- Il ciclo `while` viene eseguito finché la condizione (qui: `a < 10`) rimane vera. In Python, come in C, qualsiasi valore intero diverso da zero è vero; zero è falso. La condizione può anche essere una stringa o un valore di lista, infatti qualsiasi sequenza; qualsiasi cosa con una lunghezza diversa da zero è vera, le sequenze vuote sono false. Il test utilizzato nell'esempio è un semplice confronto. Gli operatori di confronto standard sono scritti come in C: `<` (minore di), `>` (maggiore di), `==` (uguale a), `<=` (minore o uguale a), `>=` (maggiore o uguale a) e `!=` (diverso da).
- Il corpo del ciclo è *indentato*: l'indentazione è il modo di Python di raggruppare le istruzioni. Al prompt interattivo, è necessario digitare un tab o spazio/i per ogni riga rientrata. In pratica si prepara un input più complicato per Python con un editor di testo; tutti gli editor di testo seri hanno una funzione di auto-indentazione. Quando un'istruzione composta è inserita nella shell interattiva, deve essere seguita da una riga vuota per indicare il completamento (poiché l'analizzatore non può indovinare quando si è digitata l'ultima riga). Si noti che ogni riga all'interno di un blocco di base deve essere rientrata della stessa quantità.
- The `print()` function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating-point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```

>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536

```

L'argomento `end` può essere usato per evitare la nuova linea dopo l'output, o 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,

```

Note

Altri Strumenti di Controllo del Flusso

Oltre all'istruzione `while` appena introdotta, Python ne utilizza altre che incontreremo in questo capitolo.

4.1 Istruzioni `if`

Forse il tipo di istruzione più conosciuto è l'istruzione `if`. Ad 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
```

Possiamo avere zero o più parti `elif`, e la parte `else` è facoltativa. La parola chiave “`elif`” è l'abbreviazione di “`else if`”, ed è utile per evitare un'eccessiva indentazione. Una sequenza `if ... elif ... elif ...` è un sostituto per le istruzioni `switch` o `case` trovate in altri linguaggi.

Se stai confrontando lo stesso valore con diverse costanti, o controllando tipi o attributi specifici, potresti trovare utile l'istruzione `match`. Per maggiori dettagli, consulta [Istruzioni `match`](#).

4.2 Istruzioni `for`

L'istruzione `for` in Python differisce leggermente da ciò a cui potresti essere abitualmente abituato in C o Pascal. Piuttosto che iterare sempre su una progressione aritmetica di numeri (come in Pascal), o dare all'utente la possibilità di definire sia il passo di iterazione che la condizione di arresto (come in C), l'istruzione `for` di Python itera sugli elementi di qualsiasi sequenza (una lista o una stringa), nell'ordine in cui appaiono nella sequenza. Ad esempio:

```
>>> # Measure some strings:
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Il codice che modifica una collezione durante l'iterazione sulla stessa può essere difficile da scrivere correttamente. Invece, è generalmente più semplice iterare su una copia della collezione o crearne una nuova:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '???': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3 La Funzione `range()`

Se hai bisogno di iterare su una sequenza di numeri, la funzione integrata `range()` è utile. Genera progressioni aritmetiche:

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

Il valore finale fornito non fa parte della sequenza generata; `range(10)` genera 10 valori, gli indici legali per gli elementi di una sequenza di lunghezza 10. È possibile far partire l'intervallo da un altro numero, o specificare un incremento diverso (anche negativo; talvolta questo è chiamato “passo”):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

Per iterare sugli indici di una sequenza, puoi combinare `range()` e `len()` nel seguente modo:

```
>>> 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
```

Nella maggior parte di questi casi, però, è conveniente utilizzare la funzione `enumerate()`, vedi *Tecniche di Looping*.

Attenzione: succede una cosa strana se provi a stampare un range:

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

In molti casi, l'oggetto restituito da `range()` si comporta come se fosse una lista, ma in realtà non lo è. È un oggetto che restituisce gli elementi successivi della sequenza desiderata quando ci si itera sopra, ma non crea effettivamente la lista, risparmiando così spazio.

Diciamo che un tale oggetto è un *iterable*, cioè adatto per funzioni e costrutti che si aspettano qualcosa da cui possono ottenere elementi fino a quando il contenitore è vuoto. Abbiamo visto che l'istruzione `for` è un tale costrutto, mentre un esempio di una funzione che prende come argomento un oggetto iterabile è `sum()`:

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

In seguito vedremo altre funzioni che restituiscono oggetti iterabili e prendono oggetti iterabili come argomenti. Nel capitolo *Strutture Dati*, discuteremo in modo più dettagliato riguardo a `list()`.

4.4 break and continue Statements

The `break` statement breaks out of the innermost enclosing `for` or `while` loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The `continue` statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
```

(continues on next page)

(continua dalla pagina precedente)

```
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5 `else` Clauses on Loops

In a `for` or `while` loop the `break` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes.

In a `for` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no `break` occurred.

In un ciclo `while`, viene eseguita dopo che la condizione del ciclo diventa falsa.

In either kind of loop, the `else` clause is **not** executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

Questo è esemplificato nel seguente ciclo `for`, che cerca numeri primi:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...         else:
...             # loop fell through without finding a factor
...             print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, **not** the `if` statement.)

One way to think of the `else` clause is to imagine it paired with the `if` inside the loop. As the loop executes, it will run a sequence like `if/if/if/else`. The `if` is inside the loop, encountered a number of times. If the condition is ever true, a `break` will happen. If the condition is never true, the `else` clause outside the loop will execute.

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see [Gestione delle Eccezioni](#).

4.6 Istruzioni `pass`

L'istruzione `pass` non fa nulla. Può essere utilizzata quando è richiesta un'istruzione sintatticamente, ma il programma non richiede alcuna azione. Per esempio:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
... 
```

Questo è comunemente usato per creare classi minime:

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

Un altro posto in cui `pass` può essere usato è come segnaposto per il corpo di una funzione o condizionale quando si sta lavorando su nuovo codice, permettendo di continuare a pensare a un livello più astratto. L'istruzione `pass` viene ignorata silenziosamente:

```
>>> def initlog(*args):
...     pass # Remember to implement this!
... 
```

4.7 Istruzioni `match`

Un'istruzione `match` prende un'espressione e confronta il suo valore con schemi successivi forniti come uno o più blocchi case. Questo è superficialmente simile a un'istruzione `switch` in C, Java o JavaScript (e molti altri linguaggi), ma è più simile al pattern matching in linguaggi come Rust o Haskell. Solo il primo schema che corrisponde viene eseguito e può anche estrarre componenti (elementi di sequenze o attributi di oggetti) dal valore in variabili.

La forma più semplice confronta un valore con uno o più letterali:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Nota l'ultimo blocco: il «nome della variabile» `_` funge da *wildcard* e non fallisce mai. Se nessun case corrisponde, nessuno dei rami viene eseguito.

Puoi combinare diversi letterali in un singolo schema usando `|` («or»):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Gli schemi possono sembrare assegnazioni di decomposizione e possono essere utilizzati per associare variabili:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Studialo attentamente! Il primo schema ha due letterali, e può essere pensato come un'estensione dello schema letterale mostrato sopra. Ma i successivi due schemi combinano un letterale e una variabile, e la variabile *associa* un valore

dal soggetto (`point`). Il quarto schema cattura due valori, che lo rende concettualmente simile all'assegnazione con *unpacking* `(x, y) = point`.

Se stai utilizzando classi per strutturare i tuoi dati, puoi usare il nome della classe seguito da un elenco di argomenti che assomiglia a un costruttore, ma con la capacità di catturare attributi in variabili:

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

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

Puoi utilizzare parametri posizionali con alcune classi integrate che forniscono un ordinamento per i loro attributi (ad esempio le dataclass). Puoi anche definire una posizione specifica per gli attributi negli schemi impostando l'attributo speciale `__match_args__` nelle tue classi. Se è impostato su («x», «y»), gli schemi seguenti sono tutti equivalenti (e tutti associano l'attributo `y` alla variabile `var`):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

Un modo consigliato per leggere gli schemi è considerarli come una forma estesa di quello che metteresti a sinistra di un'assegnazione, per capire quali variabili verrebbero impostate su cosa. Solo i nomi autonomi (come `var` qui sopra) vengono assegnati da un'istruzione di `match`. I nomi puntati (come `foo.bar`), i nomi degli attributi (il `x=` e `y=` qui sopra) o i nomi delle classi (riconosciuti dalle «(...)» accanto a loro come `Point` qui sopra) non vengono mai assegnati.

Gli schemi possono essere arbitrariamente nidificati. Ad esempio, se abbiamo una breve lista di Punti, con `__match_args__` aggiunto, potremmo abbinarla così:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
```

(continues on next page)

(continua dalla pagina precedente)

```
case _:
    print("Something else")
```

Possiamo aggiungere una clausola `if` a uno schema, nota come «guardia». Se la guardia è falsa, `match` passa a provare il blocco case successivo. Nota che la cattura del valore avviene prima che la guardia venga valutata:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

Altre caratteristiche chiave di questa istruzione:

- Come le assegnazioni di decomposizione, gli schemi di tuple e liste hanno esattamente lo stesso significato e corrispondono effettivamente a sequenze arbitrarie. Un'importante eccezione è che non corrispondano a iteratori o stringhe.
- Le sequenze di schemi supportano l'unpacking esteso: `[x, y, *rest]` e `(x, y, *rest)` funzionano in modo simile alle assegnazioni di decomposizione. Il nome dopo `*` può anche essere `_`, quindi `(x, y, *)` corrisponde a una sequenza di almeno due elementi senza la necessità di associare gli elementi rimanenti.
- Schemi di mapping: `{"bandwidth": b, "latency": l}` cattura i valori "bandwidth" e "latency" da un dizionario. A differenza degli schemi di sequenza, le chiavi aggiuntive vengono ignorate. È supportato anche un unpacking come `**rest`. (Ma `**_` sarebbe ridondante, quindi non è possibile utilizzarlo.)
- I sotto-schemi possono essere catturati utilizzando la parola chiave `as`:

```
case (Point(x1, y1), Point(x2, y2) as p2): ...
```

acquisirà il secondo elemento dell'input come `p2` (a patto che l'input sia una sequenza di due punti)

- La maggior parte dei letterali sono confrontati per uguaglianza, tuttavia i *singleton* `True`, `False` e `None` sono confrontati per identità.
- Gli schemi possono utilizzare costanti nominate. Queste devono essere nomi puntati per impedire che vengano interpretati come variabili di cattura:

```
from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :)")
```

Per una spiegazione più dettagliata e ulteriori esempi, puoi consultare [PEP 636](#) che è scritta in un formato simile a un tutorial.

4.8 Definizione di Funzioni

Possiamo creare una funzione che scrive la serie di Fibonacci fino a un limite arbitrario:

```
>>> def fib(n):      # write Fibonacci series less than n
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
>>> 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 una *definizione* di funzione. Deve essere seguita dal nome della funzione e dall'elenco tra parentesi dei parametri formali. Le istruzioni che formano il corpo della funzione iniziano alla riga successiva, e devono essere indentate.

La prima istruzione del corpo della funzione può opzionalmente essere una stringa letterale; questa stringa letterale è la stringa di documentazione della funzione, o *docstring*. (Maggiori informazioni sulle docstring possono essere trovate nella sezione [Stringhe di Documentazione](#).) Ci sono strumenti che utilizzano le docstring per produrre automaticamente documentazione online o stampata, o per consentire all'utente di navigare interattivamente attraverso il codice; è una buona pratica includere le docstring nel codice che scrivi, quindi prendilo come abitudine.

L'*esecuzione* di una funzione introduce una nuova tabella dei simboli utilizzata per le variabili locali della funzione. Più precisamente, tutte le assegnazioni di variabili in una funzione memorizzano il valore nella tabella dei simboli locali; mentre i riferimenti alle variabili cercano prima nella tabella dei simboli locali, poi nelle tabelle dei simboli locali delle funzioni circostanti, poi nella tabella dei simboli globali, e infine nella tabella dei nomi incorporati. Pertanto, le variabili globali e le variabili delle funzioni circostanti non possono essere direttamente assegnate un valore all'interno di una funzione (a meno che, per le variabili globali, siano nominate in un'istruzione `global`, o, per le variabili delle funzioni circostanti, siano nominate in un'istruzione `nonlocal`), anche se possono essere referenziate.

I parametri effettivi (argomenti) di una chiamata di funzione vengono introdotti nella tabella dei simboli locali della funzione chiamata quando viene chiamata; quindi, gli argomenti sono passati utilizzando il *passaggio per valore* (dove il *valore* è sempre un *riferimento* all'oggetto, non il valore dell'oggetto).¹ Quando una funzione chiama un'altra funzione, o si chiama ricorsivamente, viene creata una nuova tabella dei simboli locali per quella chiamata.

Una definizione di funzione associa il nome della funzione all'oggetto funzione nella tabella dei simboli corrente. L'interprete riconosce l'oggetto puntato da quel nome come una funzione definita dall'utente. Altri nomi possono anche puntare a quello stesso oggetto funzione e possono anche essere utilizzati 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
```

Provenendo da altri linguaggi, potresti obiettare che `fib` non è una funzione ma una procedura poiché non restituisce un valore. Infatti, anche le funzioni senza un'istruzione `return` restituiscono un valore, sebbene piuttosto noioso. Questo valore si chiama `None` (è un nome riservato del linguaggio). Scrivere il valore `None` è normalmente soppresso dall'interprete se sarebbe l'unico valore scritto. Puoi vederlo se lo desideri davvero utilizzando `print()`:

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

¹ In realtà, *chiamata per riferimento all'oggetto* sarebbe una descrizione migliore, poiché se viene passato un oggetto mutabile, il chiamante vedrà qualsiasi modifica che il chiamato fa ad esso (elementi inseriti in una lista).

È semplice scrivere una funzione che restituisce una lista dei numeri della serie di Fibonacci, invece di stamparla:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a) # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100) # call it
>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Questo esempio, come al solito, dimostra alcune nuove funzionalità di Python:

- L'istruzione `return` restituisce un valore da una funzione. `return` senza un argomento espressione restituisce `None`. Anche uscire dalla fine di una funzione restituisce `None`.
- L'istruzione `result.append(a)` chiama un *metodo* dell'oggetto lista `result`. Un metodo è una funzione che “appartiene” a un oggetto e si chiama `obj.methodname`, dove `obj` è un qualche oggetto (questo può essere un'espressione), e `methodname` è il nome di un metodo definito dal tipo dell'oggetto. Diversi tipi definiscono metodi diversi. Metodi di tipi diversi possono avere lo stesso nome senza causare ambiguità. (È possibile definire i propri tipi di oggetto e metodi, utilizzando *classi*, vedi [Classi](#)) Il metodo `append()` mostrato nell'esempio è definito per gli oggetti lista; aggiunge un nuovo elemento alla fine della lista. In questo esempio è equivalente a `result = result + [a]`, ma più efficiente.

4.9 Ulteriori informazioni sulla Definizione di Funzioni

È anche possibile definire funzioni con un numero variabile di argomenti. Esistono tre forme, che possono essere combinate.

4.9.1 Valori predefiniti degli argomenti

La forma più utile è specificare un valore predefinito per uno o più argomenti. Questo crea una funzione che può essere chiamata con meno argomenti di quelli che è definita per consentire. Ad esempio:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply 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:

- dando solo l'argomento obbligatorio: `ask_ok('Vuoi davvero uscire?')`
- dando uno degli argomenti opzionali: `ask_ok('OK sovrascrivere il file?', 2)`
- o addirittura dando tutti gli argomenti: `ask_ok('OK sovrascrivere il file?', 2, 'Dai su, solo sì o no!')`

Questo esempio introduce anche la parola chiave `in`. Questo testa se una sequenza contiene o meno un certo valore.

I valori predefiniti vengono valutati nel punto di definizione della funzione nello *scope* di definizione, in modo che

```
i = 5

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

i = 6
f()
```

stamperà 5.

Avvertenza importante: Il valore predefinito viene valutato solo una volta. Questo fa la differenza quando il valore predefinito è un oggetto mutabile come una lista, un dizionario, o istanze della maggior parte delle classi. Ad esempio, la seguente funzione accumula gli argomenti passati ad essa nelle chiamate successive:

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

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

Questo stamperà

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

Se non vuoi che il valore predefinito sia condiviso tra le chiamate successive, puoi scrivere la funzione in questo modo:

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

4.9.2 Argomenti chiave-valore

Le funzioni possono anche essere chiamate utilizzando *argomenti chiave-valore* della forma `kwarg=value`. Ad esempio, la seguente funzione:

```
def parrot(voltage, state='a stiff', action='vroom', 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, "!")
```

accetta un argomento obbligatorio (`voltage`) e tre argomenti opzionali (`state`, `action`, e `type`). Questa funzione può essere chiamata in uno dei seguenti modi:

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

ma tutte le chiamate seguenti non sarebbero valide:

```
parrot()                # required argument missing
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220)   # duplicate value for the same argument
parrot(actor='John Cleese') # unknown keyword argument
```

In una chiamata di funzione, gli argomenti chiave-valore devono seguire gli argomenti posizionali. Tutti gli argomenti chiave passati devono corrispondere a uno degli argomenti accettati dalla funzione (ad esempio, `actor` non è un argomento valido per la funzione `parrot`), e il loro ordine non è importante. Questo include anche gli argomenti non opzionali (ad esempio, `parrot(voltage=1000)` è valido). Nessun argomento può ricevere un valore più di una volta. Ecco un esempio che fallisce a causa di 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 argument 'a'
```

Quando è presente un parametro formale finale della forma `**name`, riceve un dizionario (vedi `typesmapping`) contenente tutti gli argomenti chiave tranne quelli corrispondenti a un parametro formale. Questo può essere combinato con un parametro formale della forma `*name` (descritto nella prossima sottosezione) che riceve una *tupla* contenente gli argomenti posizionali oltre all'elenco dei parametri formali. (`*name` deve occorrere prima di `**name`.) Ad 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 chiamato in questo modo:

```
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 stamperà:

```
-- 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
```

Nota che l'ordine in cui gli argomenti chiave-valore vengono stampati è garantito corrispondere all'ordine in cui sono stati forniti nella chiamata della funzione.

4.9.3 Parametri speciali

Per impostazione predefinita, gli argomenti possono essere passati a una funzione Python sia per posizione che esplicitamente per chiave. Per leggibilità e prestazioni, ha senso limitare il modo in cui gli argomenti possono essere passati in modo che uno sviluppatore debba solo guardare la definizione della funzione per determinare se gli elementi sono passati per posizione, per posizione o per chiave, o per chiave.

Ecco come potrebbe essere la definizione di una funzione:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    -----
    |           |           |
    |           | Positional or keyword |
    |           |           |
    |           |           | - Keyword only
    -- Positional only
```

dove / e * sono opzionali. Se utilizzati, questi simboli indicano il tipo di parametro in base a come gli argomenti possono essere passati alla funzione: solo per posizione, per posizione o per chiave, e solo per chiave. I parametri chiave-valore sono anche chiamati parametri nominali.

Argomenti Posizionali o per Chiave

Se / e * non sono presenti nella definizione della funzione, gli argomenti possono essere passati a una funzione per posizione o per chiave.

Parametri Solo per Posizione

Guardando questo in modo più dettagliato, è possibile contrassegnare certi parametri come *solo per posizione*. Se lo sono, l'ordine dei parametri è importante, e i parametri non possono essere passati per chiave. I parametri solo per posizione sono posizionati prima di un / (barra obliqua). Il / è utilizzato per separare logicamente i parametri posizionali dal resto dei parametri. Se non c'è un / nella definizione della funzione, non ci sono parametri posizionali.

I parametri che seguono il / possono essere *posizionali o per chiave* o *solo per chiave*.

Argomenti Solo per Chiave

Per contrassegnare i parametri come *solo per chiave*, indicando che i parametri devono essere passati per argomento chiave, posiziona un * nell'elenco degli argomenti subito prima del primo parametro *solo per chiave*.

Esempi di Funzioni

Considera le seguenti definizioni di funzioni di esempio prestando particolare attenzione ai marcatori / e *:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

Nella prima definizione di funzione, `standard_arg`, la forma più familiare, non pone restrizioni sulla convenzione di chiamata e gli argomenti possono essere passati per posizione o per chiave:

```
>>> standard_arg(2)
2
```

(continues on next page)

(continua dalla pagina precedente)

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

La seconda funzione `pos_only_arg` è limitata a utilizzare solo parametri posizionali poiché c'è un `/` nella definizione della funzione:

```
>>> 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 some positional-only arguments passed as keyword_
↳arguments: 'arg'
```

The third function `kwd_only_arg` only allows keyword arguments as indicated by a `*` in the function definition:

```
>>> 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
```

E l'ultimo utilizza tutte e tre le convenzioni di chiamata nella stessa definizione di funzione:

```
>>> 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 some positional-only arguments passed as keyword_
↳arguments: 'pos_only'
```

Infine, considera questa definizione di funzione che ha un potenziale conflitto tra l'argomento posizionale `name` e `**kws` che ha `name` come chiave:

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

Non esiste una chiamata possibile che lo farà restituire `True` poiché la chiave `'name'` si legherà sempre al primo parametro. Ad esempio:

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

(continues on next page)

(continua dalla pagina precedente)

```
TypeError: foo() got multiple values for argument 'name'
>>>
```

Ma utilizzando / (argomenti solo per posizione), è possibile poiché consente `name` come argomento posizionale e `'name'` come chiave negli argomenti chiave-valore:

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

In altre parole, i nomi dei parametri solo per posizione possono essere utilizzati in `**kwds` senza ambiguità.

Riepilogo

Il caso d'uso determinerà quali parametri utilizzare nella definizione della funzione:

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

Come guida:

- Usa solo argomenti per posizione se vuoi che il nome dei parametri non sia disponibile per l'utente. Questo è utile quando i nomi dei parametri non hanno un vero significato, se vuoi imporre l'ordine degli argomenti quando la funzione viene chiamata o se devi prendere alcuni parametri posizionali e argomenti chiave arbitrari.
- Usa solo argomenti per chiave quando i nomi hanno significato e la definizione della funzione è più comprensibile essendo esplicita con i nomi o vuoi impedire agli utenti di fare affidamento sulla posizione dell'argomento passato.
- Per un'API, usa solo per posizione per evitare rotture delle modifiche API se il nome del parametro viene modificato in futuro.

4.9.4 Liste di Argomenti Arbitrarie

Infine, l'opzione meno utilizzata è specificare che una funzione può essere chiamata con un numero arbitrario di argomenti. Questi argomenti saranno racchiusi in una tupla (vedi *Tuple e Sequenze*). Prima del numero variabile di argomenti, possono verificarsi zero o più argomenti normali.

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

Normalmente, questi argomenti *variadici* saranno gli ultimi nell'elenco dei parametri formali, perché raccolgono tutti gli argomenti di input rimanenti che vengono passati alla funzione. Qualsiasi parametro formale che si verifica dopo il parametro `*args` sono argomenti "solo per chiave", il che significa che possono essere utilizzati solo come argomenti chiave piuttosto che argomenti posizionali.

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

4.9.5 Spacchettamento delle Liste di Argomenti

La situazione inversa si verifica quando gli argomenti sono già in una lista o tupla ma devono essere spacchettati per una chiamata di funzione che richiede argomenti posizionali separati. Ad esempio, la funzione built-in `range()` si aspetta argomenti *start* e *stop* separati. Se non sono disponibili separatamente, scrivi la chiamata di funzione con l'operatore `*` per spacchettare gli argomenti da una lista o tupla:

```
>>> list(range(3, 6))           # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))         # call with arguments unpacked from a list
[3, 4, 5]
```

Nello stesso modo, i dizionari possono fornire argomenti chiave-valore con l'operatore `**`:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
...     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 !
```

4.9.6 Espressioni Lambda

Piccole funzioni anonime possono essere create con la parola chiave `lambda`. Questa funzione restituisce la somma dei suoi due argomenti: `lambda a, b: a+b`. Le funzioni `lambda` possono essere utilizzate ovunque siano richiesti oggetti funzione. Sono sintatticamente limitate a una singola espressione. Semanticamente, sono solo zucchero sintattico per una normale definizione di funzione. Come le definizioni di funzione nidificate, le funzioni `lambda` possono fare riferimento a variabili dallo scope contenente:

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

Nell'esempio precedente viene utilizzata un'espressione `lambda` per restituire una funzione. Un altro uso è passare una piccola funzione come argomento:

```
>>> 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')]
```

4.9.7 Stringhe di Documentazione

Ecco alcune convenzioni sul contenuto e la formattazione delle stringhe di documentazione.

La prima riga dovrebbe essere sempre un breve e conciso riassunto dello scopo dell'oggetto. Per brevità, non dovrebbe esplicitamente indicare il nome o il tipo dell'oggetto, poiché questi sono disponibili in altri modi (tranne se il nome è un verbo che descrive l'operazione di una funzione). Questa riga dovrebbe iniziare con una lettera maiuscola e terminare con un punto.

Se ci sono più righe nella stringa di documentazione, la seconda riga dovrebbe essere vuota, separando visivamente il riassunto dal resto della descrizione. Le righe seguenti dovrebbero essere uno o più paragrafi che descrivono le convenzioni di chiamata dell'oggetto, i suoi effetti collaterali, ecc.

Il parser di Python non rimuove l'indentazione dalle stringhe letterali multilinea in Python, quindi gli strumenti che elaborano la documentazione devono rimuovere l'indentazione se desiderato. Questo viene fatto utilizzando la seguente convenzione. La prima riga non vuota *dopo* la prima riga della stringa determina la quantità di indentazione per l'intera stringa di documentazione. (Non possiamo usare la prima riga poiché è generalmente adiacente alle virgolette di apertura della stringa quindi la sua indentazione non è evidente nella stringa letterale.) Lo spazio bianco «equivalente» a questa indentazione viene quindi rimosso dall'inizio di tutte le righe della stringa. Le righe che sono indentate di meno non dovrebbero verificarsi, ma se si verificano tutto il loro spazio bianco iniziale dovrebbe essere rimosso. L'equivalenza dello spazio bianco dovrebbe essere testata dopo l'espansione delle tabulazioni (a 8 spazi, normalmente).

Di seguito è riportato un esempio di una stringa di documentazione multilinea:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.9.8 Annotazioni delle Funzioni

Le annotazioni delle funzioni sono informazioni di metadati completamente opzionali sui tipi utilizzati dalle funzioni definite dall'utente (vedi [PEP 3107](#) e [PEP 484](#) per ulteriori informazioni).

Le *annotazioni* sono memorizzate nell'attributo `__annotations__` della funzione come un dizionario e non hanno alcun effetto su qualsiasi altra parte della funzione. Le annotazioni dei parametri sono definite da due punti dopo il nome del parametro, seguito da un'espressione che valuta il valore dell'annotazione. Le annotazioni di ritorno sono definite da una freccia `->`, seguita da un'espressione, tra l'elenco dei parametri e i due punti che indicano la fine della dichiarazione `def`. L'esempio seguente ha un argomento obbligatorio, un argomento opzionale, e il valore di ritorno annotato:

```
>>> 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'
```

4.10 Intermezzo: Stile di Codifica

Ora che stai per scrivere pezzi di Python più lunghi e complessi, è un buon momento per parlare dello *stile di codifica*. La maggior parte dei linguaggi può essere scritta (o più concisamente, *formattata*) in stili diversi; alcuni sono più leggibili di altri. Rendere facile per gli altri leggere il tuo codice è sempre una buona idea, e adottare uno stile di codifica piacevole aiuta enormemente.

Per Python, la **PEP 8** è la guida di stile a cui aderiscono la maggior parte dei progetti; promuove uno stile di codifica molto leggibile e piacevole alla vista. Ogni sviluppatore Python dovrebbe leggerla prima o poi; ecco i punti più importanti estratti per te:

- Usa l'indentazione a 4 spazi, e niente tabulazioni.
4 spazi sono un buon compromesso tra una piccola indentazione (permette una maggiore profondità di nidificazione) e una grande indentazione (più facile da leggere). Le tabulazioni introducono confusione, ed è preferibile non usarle.
- Manda a capo le righe in modo che non superino i 79 caratteri.
Questo aiuta gli utenti con schermi piccoli e rende possibile avere diversi file di codice affiancati su schermi più grandi.
- Usa righe vuote per separare funzioni e classi, e blocchi di codice più grandi all'interno delle funzioni.
- Quando possibile, metti i commenti su una riga a parte.
- Usa le stringhe di documentazione.
- Usa spazi attorno agli operatori e dopo le virgole, ma non direttamente all'interno dei costrutti tra parentesi: `a = f(1, 2) + g(3, 4)`.
- Nomina le tue classi e funzioni in modo coerente; la convenzione è utilizzare `UpperCamelCase` per le classi e `lowercase_with_underscores` per le funzioni e i metodi. Usa sempre `self` come nome per il primo argomento del metodo (vedi *Una prima occhiata alle classi* per ulteriori informazioni su classi e metodi).
- Non utilizzare codifiche complesse se il tuo codice è destinato ad essere utilizzato in ambienti internazionali. Il predefinito di Python, UTF-8, o anche il semplice ASCII funzionano meglio in ogni caso.
- Allo stesso modo, non utilizzare caratteri non-ASCII negli identificatori se c'è solo la minima possibilità che persone che parlano una lingua diversa leggano o mantengano il codice.

Questo capitolo descrive alcune cose che hai già imparato in maggiore dettaglio, e aggiunge anche alcune nuove cose.

5.1 Più sulle Liste

Il tipo di dato lista ha alcuni metodi aggiuntivi. Ecco tutti i metodi degli oggetti lista:

`list.append(x)`

Add an item to the end of the list. Similar to `a[len(a):] = [x]`.

`list.extend(iterable)`

Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.

`list.insert(i, x)`

Inserisce un elemento in una posizione specificata. Il primo argomento è l'indice dell'elemento prima del quale inserire, quindi `a.insert(0, x)` inserisce all'inizio della lista, e `a.insert(len(a), x)` è equivalente a `a.append(x)`.

`list.remove(x)`

Rimuove il primo elemento dalla lista il cui valore è uguale a `x`. Solleva una `ValueError` se non c'è un tale elemento.

`list.pop([i])`

Rimuove l'elemento nella posizione specificata nella lista e lo restituisce. Se non viene specificato un indice, `a.pop()` rimuove e restituisce l'ultimo elemento della lista. Solleva un `IndexError` se la lista è vuota o l'indice è fuori dal range della lista.

`list.clear()`

Remove all items from the list. Similar to `del a[:]`.

`list.index(x[, start[, end]])`

Restituisce l'indice zero-based del primo elemento nella lista il cui valore è uguale a `x`. Solleva una `ValueError` se non c'è un tale elemento.

Gli argomenti opzionali `start` e `end` sono interpretati come nella notazione delle slice e sono usati per limitare la ricerca a una particolare sottosequenza della lista. L'indice restituito è calcolato rispetto all'inizio della sequenza completa piuttosto che all'argomento `start`.

`list.count(x)`

Restituisce il numero di volte che *x* appare nella lista.

`list.sort(*, key=None, reverse=False)`

Ordina gli elementi della lista in loco (gli argomenti possono essere usati per la personalizzazione dell'ordinamento, vedere `sorted()` per la loro spiegazione).

`list.reverse()`

Inverte gli elementi della lista in loco.

`list.copy()`

Return a shallow copy of the list. Similar to `a[:]`.

Un esempio che utilizza la maggior parte dei metodi della lista:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting at position 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'
```

Avrai notato che metodi come `insert`, `remove` o `sort` che modificano solo la lista non stampano alcun valore di ritorno – ritornano il default `None`.¹ Questo è un principio di design per tutte le strutture dati mutabili in Python.

Un'altra cosa che potresti notare è che non tutti i dati possono essere ordinati o confrontati. Per esempio, `[None, 'hello', 10]` non può essere ordinata perché gli interi non possono essere confrontati con le stringhe e `None` non può essere confrontato con altri tipi. Inoltre, ci sono alcuni tipi che non hanno una relazione di ordinamento definita. Ad esempio, `3+4j < 5+7j` non è un confronto valido.

5.1.1 Usare le Liste come Pile

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved («last-in, first-out»). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

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

(continues on next page)

¹ Altri linguaggi possono restituire l'oggetto mutato, il che consente di concatenare i metodi, come `d->insert("a")->remove("b")->sort();`.

(continua dalla pagina precedente)

```
>>> 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 una lista come una coda, dove il primo elemento aggiunto è il primo elemento recuperato («first-in, first-out»); tuttavia, le liste non sono efficienti per questo scopo. Mentre gli `append` e i `pop` dalla fine della lista sono veloci, fare inserzioni o `pop` dall'inizio di una lista è lento (perché tutti gli altri elementi devono essere spostati di uno).

Per implementare una coda, usa `collections.deque` che è stata progettata 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 arrives
>>> queue.append("Graham")         # Graham arrives
>>> queue.popleft()                # The first to arrive now leaves
'Eric'
>>> queue.popleft()                # The second to arrive now leaves
'John'
>>> queue                          # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Comprensioni di Lista

Le comprensioni di lista forniscono un modo conciso per creare liste. Le applicazioni comuni sono creare nuove liste dove ogni elemento è il risultato di alcune operazioni applicate a ciascun membro di un'altra sequenza o iterabile, o creare una sottosequenza di quegli elementi che soddisfano una certa condizione.

Per esempio, supponiamo di voler creare una lista di quadrati, come:

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

Nota che questo crea (o sovrascrive) una variabile chiamata `x` che esiste ancora dopo che il ciclo è completo. Possiamo calcolare la lista dei quadrati senza effetti collaterali usando:

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

oppure, equivalentemente:

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

il che è più conciso e leggibile.

Una comprensione di lista consiste di parentesi contenenti un'espressione seguita da una clausola `for`, quindi zero o più clausole `for` o `if`. Il risultato sarà una nuova lista risultante dalla valutazione dell'espressione nel contesto delle clausole `for` e `if` che la seguono. Per esempio, questa listcomp combina gli 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)]
```

e equivale 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)]
```

Nota come l'ordine delle istruzioni `for` e `if` sia lo stesso in entrambi questi snippet.

Se l'espressione è una tupla (per es. il `(x, y)` nell'esempio precedente), deve essere racchiusa tra parentesi.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
File "<stdin>", line 1
    [x, x**2 for x in range(6)]
    ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # flatten a list using a listcomp with two '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 comprensioni di lista possono contenere espressioni complesse e funzioni nidificate:

```
>>> 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 Comprensioni di Lista Nidificate

L'espressione iniziale in una comprensione di lista può essere qualsiasi espressione arbitraria, inclusa un'altra comprensione di lista.

Considera il seguente esempio di una matrice 3x4 implementata come una lista di 3 liste di lunghezza 4:

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

La seguente comprensione di lista trasporrà 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 nella sezione precedente, la comprensione di lista interna è valutata nel contesto del `for` che la segue, quindi questo esempio è equivalente 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 lo stesso di:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     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]]
```

Nel mondo reale, dovresti preferire le funzioni built-in a strutture di controllo complesse. La funzione `zip()` farebbe un ottimo lavoro per questo caso d'uso:

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

Vedi *Spacchettamento delle Liste di Argomenti* per dettagli sull'asterisco in questa riga.

5.2 L'istruzione `del`

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

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

(continues on next page)

(continua dalla pagina precedente)

```
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` può anche essere usato per cancellare variabili intere:

```
>>> del a
```

Fare riferimento al nome `a` d'ora in poi è un errore (almeno finché un altro valore non viene assegnato ad esso). Troveremo altri usi per `del` più avanti.

5.3 Tuple e Sequenze

Abbiamo visto che liste e stringhe hanno molte proprietà comuni, come le operazioni di indicizzazione e slicing. Sono due esempi di tipi di dati *sequenza* (vedi `typeseq`). Poiché Python è un linguaggio in evoluzione, possono essere aggiunti altri tipi di dati sequenza. Esiste anche un altro tipo di dati sequenza standard: la *tupla*.

Una tupla consiste in un numero di valori separati da virgole, ad esempio:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Come vedi, in uscita le tuple sono sempre racchiuse tra parentesi, in modo che le tuple annidate siano interpretate correttamente; possono essere inserite con o senza parentesi, anche se spesso le parentesi sono comunque necessarie (se la tupla fa parte di un'espressione più grande). Tuttavia, non è possibile assegnare ai singoli elementi di una tupla, mentre è possibile creare tuple che contengono oggetti mutabili, come le liste.

Sebbene le tuple possano sembrare simili alle liste, vengono spesso utilizzate in situazioni diverse e per scopi diversi. Le tuple sono *immutable*, e di solito contengono una sequenza eterogenea di elementi che sono accessibili tramite unpacking (vedi più avanti in questa sezione) o indicizzazione (o anche per attributo nel caso di `namedtuples`). Le liste sono *mutable*, e i loro elementi sono di solito omogenei e sono accessibili iterando sulla lista.

Un problema speciale è la costruzione di tuple contenenti 0 o 1 elemento: la sintassi presenta alcune peculiarità per accogliere questi casi. Le tuple vuote sono costruite da una coppia di parentesi vuota; una tupla con un solo elemento è costruita seguendo un valore con una virgola (non è sufficiente racchiudere un singolo valore tra parentesi). Brutto, ma efficace. Ad esempio:

```
>>> empty = ()
>>> singleton = 'hello', # <-- note trailing comma
```

(continues on next page)

(continua dalla pagina precedente)

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

L'istruzione `t = 12345, 54321, 'hello!'` è un esempio di *tuple packing*: i valori 12345, 54321 e 'hello!' sono racchiusi insieme in una tupla. L'operazione inversa è anche possibile:

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

Questo è chiamato, appropriatamente, *sequence unpacking* e funziona per qualsiasi sequenza sul lato destro. Lo *sequence unpacking* richiede che ci siano tante variabili sul lato sinistro del segno di uguale quante sono gli elementi nella sequenza. Nota che l'assegnazione multipla è davvero solo una combinazione di *tuple packing* e *sequence unpacking*.

5.4 Insiemi

Python include anche un tipo di dato per gli *insiemi*. Un insieme è una collezione non ordinata senza elementi duplicati. Gli usi principali includono il controllo dell'appartenenza e l'eliminazione degli elementi duplicati. Gli oggetti *set* supportano anche operazioni matematiche come unione, intersezione, differenza e differenza simmetrica.

Le parentesi graffe o la funzione `set()` possono essere utilizzate per creare insiemi. Nota: per creare un insieme vuoto devi utilizzare `set()`, non `{}`; quest'ultimo crea un dizionario vuoto, una struttura dati che discuteremo nella sezione successiva.

Ecco una breve dimostrazione:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)           # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket      # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                               # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                           # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                           # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                           # letters in both a and b
{'a', 'c'}
>>> a ^ b                           # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Analogamente alle *list comprehensions*, sono supportate anche le *set comprehensions*:

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

5.5 Dizionari

Another useful data type built into Python is the *dictionary* (see typesmapping). Dictionaries are sometimes found in other languages as «associative memories» or «associative arrays». Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

È meglio pensare a un dizionario come a un insieme di coppie *chiave: valore*, con il requisito che le chiavi siano uniche (all'interno di un dizionario). Una coppia di parentesi crea un dizionario vuoto: `{}`. Inserendo una lista di coppie chiave:valore separate da virgole dentro le parentesi si aggiungono le coppie iniziali al dizionario; questo è anche il modo in cui i dizionari vengono scritti in uscita.

Le principali operazioni su un dizionario sono memorizzare un valore con una chiave e estrarre il valore data la chiave. È anche possibile eliminare una coppia chiave:valore con `del`. Se memorizzi usando una chiave già in uso, il vecchio valore associato a quella chiave viene dimenticato. È un errore estrarre un valore usando una chiave inesistente.

Eseguendo `list(d)` su un dizionario si ottiene una lista di tutte le chiavi utilizzate nel dizionario, in ordine di inserimento (se la vuoi ordinata, basta usare `sorted(d)`). Per controllare se una singola chiave è nel dizionario, usa la parola chiave `in`.

Ecco un piccolo esempio usando 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
```

Il costruttore `dict()` costruisce dizionari direttamente da sequenze di coppie chiave-valore:

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

Inoltre, le dict comprehensions possono essere utilizzate per creare dizionari da espressioni di chiave e valore arbitrarie:

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

Quando le chiavi sono semplici stringhe, a volte è più facile specificare le coppie utilizzando gli argomenti per parola chiave:

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

5.6 Tecniche di Looping

Quando si scorre un dizionario, è possibile recuperare contemporaneamente la chiave e il valore corrispondente usando 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 scorre una sequenza, è possibile recuperare contemporaneamente l'indice di posizione e il valore corrispondente usando la funzione `enumerate()`.

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

Per scorrere due o più sequenze contemporaneamente, le voci possono essere abbinate 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 scorrere una sequenza al contrario, specifica prima la sequenza in direzione avanti e poi chiama la funzione `reversed()`.

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

Per scorrere una sequenza in ordine ordinato, usa la funzione `sorted()` che restituisce una nuova lista ordinata lasciando invariata la sorgente.

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

Usare `set()` su una sequenza elimina gli elementi duplicati. L'uso di `sorted()` in combinazione con `set()` su una sequenza è un modo idiomatico di scorrere gli elementi unici della sequenza in ordine ordinato.

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

A volte è allettante modificare una lista mentre la stai scorrendo; tuttavia, è spesso più semplice e sicuro creare una nuova lista invece.

```
>>> 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 Maggiori informazioni sulle Condizioni

Le condizioni utilizzate nelle istruzioni `while` e `if` possono contenere qualsiasi operatore, non solo confronti.

Gli operatori di confronto `in` e `not in` sono test di appartenenza che determinano se un valore è (o non è) in un contenitore. Gli operatori `is` e `is not` confrontano se due oggetti sono davvero lo stesso oggetto. Tutti gli operatori di confronto hanno la stessa priorità, che è inferiore a quella di tutti gli operatori numerici.

I confronti possono essere concatenati. Ad esempio, `a < b == c` verifica 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`, e il risultato di un confronto (o di qualsiasi altra espressione booleana) può essere negato con `not`. Questi hanno una priorità inferiore rispetto agli operatori di confronto; tra di loro, `not` ha la priorità più alta e `or` la più bassa, in modo che `A and not B or C` sia equivalente a `(A and (not B)) or C`. Come sempre, le parentesi possono essere utilizzate per esprimere la composizione desiderata.

Gli operatori booleani `and` e `or` sono i cosiddetti operatori *short-circuit*: i loro argomenti vengono valutati da sinistra a destra, e la valutazione si arresta non appena l'esito è determinato. Ad esempio, se `A` e `C` sono veri ma `B` è falso, `A and B and C` non valuta l'espressione `C`. Quando vengono utilizzati come un valore generale e non come booleani, il valore restituito di un operatore short-circuit è l'ultimo argomento valutato.

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

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

Nota che in Python, a differenza del C, l'assegnazione dentro espressioni deve essere fatta esplicitamente con l'operatore detto walrus operator `:`. Questo evita una comune classe di problemi incontrati nei programmi C: digitare `=` in un'espressione quando si intendeva `==`.

5.8 Confronto tra Sequenze e Altri Tipi

Gli oggetti sequenza possono essere tipicamente confrontati con altri oggetti dello stesso tipo di sequenza. Il confronto usa l'ordinamento *lessicografico*: prima vengono confrontati i primi due elementi, e se differiscono questo determina il risultato del confronto; se sono uguali, vengono confrontati i due successivi, e così via, fino a quando una delle due sequenze è esaurita. Se gli elementi da confrontare sono essi stessi sequenze dello stesso tipo, il confronto lessicografico viene effettuato ricorsivamente. Se tutti gli elementi di due sequenze sono uguali tra di loro, le sequenze sono considerate uguali. Se una sequenza è una sottosequenza iniziale dell'altra, la sequenza più corta è considerata la minore. L'ordinamento lessicografico per le stringhe utilizza il numero del punto di codice Unicode per ordinare i singoli caratteri. Alcuni esempi di confronti 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)
```

Nota che confrontare oggetti di tipi diversi con `<` o `>` è legale a condizione che gli oggetti abbiano metodi di confronto appropriati. Ad esempio, i tipi numerici misti sono confrontati in base al loro valore numerico, quindi `0` è uguale a `0.0`, ecc. Altrimenti, invece di fornire un ordinamento arbitrario, l'interprete alzerà un'eccezione `TypeError`.

Se chiudi l'interprete Python e lo riapri, le definizioni che hai fatto (funzioni e variabili) vengono perse. Pertanto, se vuoi scrivere un programma un po' più lungo, è meglio usare un editor di testo per preparare l'input per l'interprete e eseguirlo con quel file come input. Questo è conosciuto come creare uno *script*. Man mano che il tuo programma diventa più lungo, potresti volerlo dividere in diversi file per una manutenzione più facile. Potresti anche voler usare una funzione utile che hai scritto in diversi programmi senza copiarne la definizione in ciascun programma.

Per supportare questo, Python ha un modo per mettere definizioni in un file e usarle in uno script o in un'istanza interattiva dell'interprete. Tale file è chiamato un *modulo*; le definizioni di un modulo possono essere *importate* in altri moduli o nel *modulo principale* (la collezione di variabili a cui hai accesso in uno script eseguito al livello superiore e in modalità calcolatrice).

Un modulo è un file contenente definizioni e istruzioni Python. Il nome del file è il nome del modulo con il suffisso `.py` aggiunto. All'interno di un modulo, il nome del modulo (come stringa) è disponibile come valore della variabile globale `__name__`. Ad esempio, usa il tuo editor di testo preferito per creare un file chiamato `fibonacci.py` nella directory corrente con il seguente contenuto:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Ora entra nell'interprete Python e importa questo modulo con il seguente comando:

```
>>> import fibonacci
```

Questo non aggiunge i nomi delle funzioni definite in `fib` direttamente all'attuale *namespace* (vedi *Visibilità e spazi dei nomi in Python* per maggiori dettagli); aggiunge solo il nome del modulo `fib` lì. Utilizzando il nome del modulo puoi accedere alle funzioni:

```
>>> 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 intendi usare una funzione spesso puoi assegnarla a 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 Più sui Moduli

Un modulo può contenere istruzioni eseguibili così come definizioni di funzioni. Queste istruzioni sono intese per inizializzare il modulo. Sono eseguite solo la *prima* volta che il nome del modulo viene incontrato in un'istruzione d'importazione.¹ (Sono anche eseguite se il file è eseguito come script.)

Ogni modulo ha il proprio namespace privato, che è utilizzato come il namespace globale da tutte le funzioni definite nel modulo. Pertanto, l'autore di un modulo può usare variabili globali nel modulo senza preoccuparsi di collisioni accidentali con le variabili globali di un utente. D'altra parte, se sai quello che stai facendo puoi toccare le variabili globali di un modulo con la stessa notazione utilizzata per riferirsi alle sue funzioni, `modname.itemname`.

I moduli possono importare altri moduli. È consuetudine, ma non obbligatorio, posizionare tutte le istruzioni `import` all'inizio di un modulo (o script, se è per questo). I nomi dei moduli importati, se posizionati a livello superiore di un modulo (fuori da qualsiasi funzione o classe), sono aggiunti al namespace globale del modulo.

Esiste una variante dell'istruzione `import` che importa nomi da un modulo direttamente nel namespace del modulo importante. Ad esempio:

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

Questo non introduce il nome del modulo da cui provengono le importazioni nel namespace locale (quindi nell'esempio, `fibo` non è definito).

Esiste anche una variante per importare tutti i nomi che un modulo definisce:

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

Questo importa tutti i nomi eccetto quelli che iniziano con un underscore (`_`). Nella maggior parte dei casi i programmatori Python non utilizzano questa funzione poiché introduce un insieme sconosciuto di nomi nell'interprete, nascondendo possibilmente alcune cose che hai già definito.

Nota che in generale la pratica di importare `*` da un modulo o pacchetto è disapprovata, poiché spesso causa codice poco leggibile. Tuttavia, è accettabile usarla per risparmiare battitura nelle sessioni interattive.

Se il nome del modulo è seguito da `as`, allora il nome che segue `as` è legato direttamente al modulo importato.

¹ In realtà anche le definizioni di funzione sono "istruzioni" che vengono "eseguite"; l'esecuzione di una definizione di funzione a livello di modulo aggiunge il nome della funzione allo spazio dei nomi globale del modulo.

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

Questo importa effettivamente il modulo nello stesso modo in cui lo farebbe `import fibo`, con l'unica differenza che sarà disponibile come `fib`.

Può anche essere utilizzato quando si utilizza `from` con effetti simili:

```
>>> 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 viene importato una sola volta per sessione dell'interprete. Pertanto, se cambi i tuoi moduli, devi riavviare l'interprete – o, se è solo un modulo che vuoi testare interattivamente, usa `importlib.reload()`, es. `import importlib; importlib.reload(modulename)`.

6.1.1 Esecuzione di moduli come script

Quando esegui un modulo Python con

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

il codice nel modulo sarà eseguito, proprio come se lo avessi importato, ma con `__name__` impostato su `"__main__"`. Ciò significa che aggiungendo questo codice alla fine del tuo modulo:

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

puoi rendere il file utilizzabile sia come script che come modulo importabile, poiché il codice che analizza la linea di comando viene eseguito solo se il modulo è eseguito come file «principale»:

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

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

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

Questo è spesso usato sia per fornire un'interfaccia utente comoda a un modulo, sia per scopi di test (eseguire il modulo come script esegue una suite di test).

6.1.2 Il percorso di ricerca del modulo

Quando un modulo chiamato `spam` è importato, l'interprete cerca prima un modulo built-in con quel nome. Questi nomi di moduli sono elencati in `sys.builtin_module_names`. Se non viene trovato, allora cerca un file chiamato `spam.py` in una lista di directory data dalla variabile `sys.path`. `sys.path` è inizializzata da queste posizioni:

- La directory contenente lo script di input (o la directory corrente quando non è specificato alcun file).
- `PYTHONPATH` (una lista di nomi di directory, con la stessa sintassi della variabile di shell `PATH`).
- Il valore predefinito dipendente dall'installazione (per convenzione include una directory `site-packages`, gestita dal modulo `site`).

Maggiori dettagli sono disponibili in `sys-path-init`.

i Nota

Su file system che supportano i collegamenti simbolici, la directory contenente lo script di input viene calcolata dopo che il collegamento simbolico è stato seguito. In altre parole la directory contenente il collegamento simbolico **non** è aggiunta al percorso di ricerca del modulo.

Dopo l'inizializzazione, i programmi Python possono modificare `sys.path`. La directory contenente lo script in esecuzione è posizionata all'inizio del percorso di ricerca, davanti al percorso della libreria standard. Ciò significa che gli script in quella directory saranno caricati invece dei moduli con lo stesso nome nella directory della libreria. Questo è un errore a meno che la sostituzione non sia intenzionale. Vedi la sezione [Moduli Standard](#) per maggiori informazioni.

6.1.3 File Python «compilati»

Per velocizzare il caricamento dei moduli, Python memorizza nella cache la versione compilata di ciascun modulo nella directory `__pycache__` sotto il nome `module.version.pyc`, dove la versione codifica il formato del file compilato; contiene generalmente il numero di versione di Python. Ad esempio, nella versione CPython 3.3, la versione compilata di `spam.py` verrebbe memorizzata nella cache come `__pycache__/spam.cpython-33.pyc`. Questa convenzione di denominazione permette ai moduli compilati di versioni diverse e rilasci diversi di Python di coesistere.

Python controlla la data di modifica della sorgente rispetto alla versione compilata per vedere se è obsoleta e deve essere ricompilata. Questo è un processo completamente automatico. Inoltre, i moduli compilati sono indipendenti dalla piattaforma, quindi la stessa libreria può essere condivisa tra sistemi con diverse architetture.

Python non controlla la cache in due circostanze. Primo, ricompila sempre e non memorizza il risultato per il modulo caricato direttamente dalla linea di comando. Secondo, non controlla la cache se non esiste il modulo sorgente. Per supportare una distribuzione senza sorgente (solo compilata), il modulo compilato deve essere nella directory sorgente e non deve esserci un modulo sorgente.

Alcuni consigli per esperti:

- Puoi usare le opzioni `-O` o `-OO` sul comando Python per ridurre la dimensione di un modulo compilato. L'opzione `-O` rimuove le istruzioni `assert`, l'opzione `-OO` rimuove sia le istruzioni `assert` che le stringhe `__doc__`. Poiché alcuni programmi possono fare affidamento su queste, dovresti usare questa opzione solo se sai quello che stai facendo. I moduli «ottimizzati» hanno un tag `opt-` e sono solitamente più piccoli. I futuri rilasci possono cambiare gli effetti dell'ottimizzazione.
- Un programma non gira più velocemente quando è letto da un file `.pyc` rispetto a quando è letto da un file `.py`; l'unica cosa che è più veloce nei file `.pyc` è la velocità con cui vengono caricati.
- Il modulo `compileall` può creare file `.pyc` per tutti i moduli in una directory.
- Ci sono maggiori dettagli su questo processo, incluso un diagramma di flusso delle decisioni, in [PEP 3147](#).

6.2 Moduli Standard

Python viene fornito con una libreria di moduli standard, descritta in un documento separato, il Python Library Reference («Library Reference» da qui in avanti). Alcuni moduli sono incorporati nell'interprete; questi forniscono accesso a operazioni che non fanno parte del core del linguaggio ma sono comunque incorporate, sia per efficienza che per fornire accesso a primitive del sistema operativo come le chiamate di sistema. Il set di tali moduli è un'opzione di configurazione che dipende anche dalla piattaforma sottostante. Ad esempio, il modulo `winreg` è fornito solo sui sistemi Windows. Un particolare modulo merita attenzione: `sys`, che è incorporato in ogni interprete Python. Le variabili `sys.ps1` e `sys.ps2` definiscono le stringhe usate come prompt primario e secondario:

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

(continues on next page)

(continua dalla pagina precedente)

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

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

La variabile `sys.path` è una lista di stringhe che determina il percorso di ricerca dell'interprete per i moduli. È inizializzata a un percorso predefinito preso dalla variabile di ambiente `PYTHONPATH`, o da un predefinito incorporato se `PYTHONPATH` non è impostato. Puoi modificarla usando le operazioni standard delle liste:

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

6.3 La funzione `dir()`

La funzione built-in `dir()` è usata per scoprire quali nomi definisce 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 che hai definito attualmente:

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

Nota che elenca tutti i tipi di nomi: variabili, moduli, funzioni, ecc.

`dir()` non elenca i nomi delle funzioni e variabili built-in. Se vuoi una lista di questi, sono definiti nel modulo standard `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 Pacchetti

I pacchetti sono un modo di strutturare il namespace dei moduli di Python usando «nomi di moduli puntati». Ad esempio, il nome del modulo `A.B` designa un sottomodulo chiamato `B` in un pacchetto chiamato `A`. Proprio come l'uso dei moduli salva gli autori di diversi moduli dal preoccuparsi dei nomi delle variabili globali degli altri, l'uso dei nomi di moduli puntati salva gli autori di pacchetti multi-modulo come `NumPy` o `Pillow` dal preoccuparsi dei nomi dei moduli degli altri.

Supponiamo che tu voglia progettare una collezione di moduli (un «pacchetto») per la gestione uniforme dei file audio e dei dati audio. Ci sono molti formati di file audio diversi (di solito riconosciuti dalla loro estensione, ad esempio: `.wav`, `.aiff`, `.au`), quindi potresti aver bisogno di creare e mantenere una collezione crescente di moduli per la conversione tra i vari formati di file. Ci sono anche molte operazioni diverse che potresti voler eseguire sui dati audio (come il mixaggio, l'aggiunta di eco, l'applicazione di una funzione equalizzatrice, la creazione di un effetto stereo artificiale), quindi, in aggiunta, scriverai una serie interminabile di moduli per eseguire queste operazioni. Ecco una possibile struttura per il tuo pacchetto (espressa in termini di filesystem gerarchico):

<code>sound/</code>	Top-level package
<code>__init__.py</code>	Initialize the sound package
<code>formats/</code>	Subpackage for file format conversions
<code>__init__.py</code>	
<code>wavread.py</code>	
<code>wavwrite.py</code>	

(continues on next page)

(continua dalla pagina precedente)

```

aiffread.py
aiffwrite.py
auread.py
auwrite.py
...
effects/                Subpackage for sound effects
__init__.py
echo.py
surround.py
reverse.py
...
filters/                Subpackage for filters
__init__.py
equalizer.py
vocoder.py
karaoke.py
...

```

Quando importa il pacchetto, Python cerca nelle directory su `sys.path` cercando la sottodirectory del pacchetto.

I file `__init__.py` sono richiesti per fare in modo che Python tratti le directory contenenti il file come pacchetti (a meno che non si utilizzi un *namespace package*, una funzionalità relativamente avanzata). Questo impedisce alle directory con un nome comune, come `string`, di nascondere inavvertitamente moduli validi che si trovano più avanti nel percorso di ricerca del modulo. Nel caso più semplice, `__init__.py` può essere solo un file vuoto, ma può anche eseguire codice di inizializzazione per il pacchetto o impostare la variabile `__all__`, descritta più avanti.

Gli utenti del pacchetto possono importare moduli individuali dal pacchetto, ad esempio:

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

Questo carica il sottomodulo `sound.effects.echo`. Deve essere referenziato con il suo nome completo.:

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

Un modo alternativo di importare il sottomodulo è:

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

Questo carica anche il sottomodulo `echo`, e lo rende disponibile senza il prefisso del pacchetto, quindi può essere usato come segue:

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

Un'ulteriore variazione è importare direttamente la funzione o la variabile desiderata:

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

Ancora una volta, questo carica il sottomodulo `echo`, ma rende direttamente disponibile la sua funzione `echofilter()`:

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

Nota che quando usi `from package import item`, l'elemento può essere un sottomodulo (o sottopacchetto) del pacchetto, o qualche altro nome definito nel pacchetto, come una funzione, classe o variabile. L'istruzione `import` testa prima se l'elemento è definito nel pacchetto; se non lo è, assume che sia un modulo e tenta di caricarlo. Se non riesce a trovarlo, viene sollevata un'eccezione `ImportError`.

Al contrario, quando usi una sintassi come `import item.subitem.subsubitem`, ciascun elemento tranne l'ultimo deve essere un pacchetto; l'ultimo elemento può essere un modulo o un pacchetto ma non può essere una classe, funzione o variabile definita nell'elemento precedente.

6.4.1 Importare * Da un Pacchetto

Ora cosa succede quando l'utente scrive `from sound.effects import *`? Idealmente, si spera che questo vada in qualche modo nel filesystem, trovi quali sottomoduli sono presenti nel pacchetto e li importi tutti. Questo potrebbe richiedere molto tempo e l'importazione di sottomoduli potrebbe avere effetti collaterali indesiderati che dovrebbero accadere solo quando il sottomodulo viene importato esplicitamente.

L'unica soluzione è che l'autore del pacchetto fornisca un indice esplicito del pacchetto. L'istruzione `import` utilizza la seguente convenzione: se il codice di `__init__.py` di un pacchetto definisce una lista denominata `__all__`, essa è considerata la lista dei nomi dei moduli che dovrebbero essere importati quando si incontra `from package import *`. È compito dell'autore del pacchetto mantenere aggiornata questa lista quando viene rilasciata una nuova versione del pacchetto. Gli autori dei pacchetti possono anche decidere di non supportarla, se non vedono un'utilità per l'importazione `*` dal loro pacchetto. Ad esempio, il file `sound/effects/__init__.py` potrebbe contenere il seguente codice:

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

Questo significherebbe che `from sound.effects import *` importerebbe i tre sottomoduli denominati del pacchetto `sound.effects`.

Tieni presente che i sottomoduli potrebbero essere ombreggiati da nomi definiti localmente. Ad esempio, se aggiungi una funzione `reverse` al file `sound/effects/__init__.py`, `from sound.effects import *` importerebbe solo i due sottomoduli `echo` e `surround`, ma *non* il sottomodulo `reverse`, perché è oscurato dalla funzione `reverse` definita localmente.:

```
__all__ = [
    "echo",      # refers to the 'echo.py' file
    "surround",  # refers to the 'surround.py' file
    "reverse",   # !!! refers to the 'reverse' function now !!!
]

def reverse(msg: str): # <-- this name shadows the 'reverse.py' submodule
    return msg[::-1]  #      in the case of a 'from sound.effects import *'
```

Se `__all__` non è definito, l'istruzione `from sound.effects import *` *non* importa tutti i sottomoduli del pacchetto `sound.effects` nello spazio dei nomi corrente; si limita ad assicurare che il pacchetto `sound.effects` sia stato importato (eventualmente eseguendo qualsiasi codice di inizializzazione in `__init__.py`) e quindi importa solo i nomi definiti nel pacchetto. Questo include qualsiasi nome definito (e sottomoduli esplicitamente caricati) nel file `__init__.py`. Include anche eventuali sottomoduli del pacchetto che sono stati esplicitamente caricati da precedenti istruzioni di `import`. Considera questo codice:

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

In questo esempio, i moduli `echo` e `surround` sono importati nello spazio dei nomi corrente perché sono definiti nel pacchetto `sound.effects` quando viene eseguita l'istruzione `from...import`. (Questo funziona anche quando è definito `__all__`.)

Sebbene certi moduli siano progettati per esportare solo nomi che seguono certi schemi quando usi `import *`, è comunque considerata una cattiva pratica nel codice di produzione.

Ricorda, non c'è nulla di sbagliato nell'usare `from package import specific_submodule`! Infatti, questa è la notazione raccomandata a meno che il modulo importante non debba usare sottomoduli con lo stesso nome da pacchetti differenti.

6.4.2 Riferimenti Intra-pacchetto

Quando i pacchetti sono strutturati in sottopacchetti (come il pacchetto `sound` nell'esempio), puoi usare importazioni assolute per riferirti a sottomoduli di pacchetti fratelli. Ad esempio, se il modulo `sound.filters.vocoder` deve utilizzare il modulo `echo` nel pacchetto `sound.effects`, può usare `from sound.effects import echo`.

Puoi anche scrivere importazioni relative, con la forma dell'istruzione di importazione `from module import name`. Queste importazioni utilizzano punti di testa per indicare i pacchetti corrente e genitore coinvolti nell'importazione relativa. Dal modulo `surround`, ad esempio, potresti usare:

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

Nota che le importazioni relative si basano sul nome del modulo corrente. Poiché il nome del modulo principale è sempre `"__main__"`, i moduli destinati all'uso come modulo principale di un'applicazione Python devono sempre usare importazioni assolute.

6.4.3 Pacchetti in Directory Multiple

Packages support one more special attribute, `__path__`. This is initialized to be a *sequence* of strings containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

Sebbene questa funzione non sia spesso necessaria, può essere utilizzata per estendere il set di moduli trovati in un pacchetto.

Note

Ci sono diversi modi per presentare l'output di un programma; i dati possono essere stampati in una forma leggibile dall'uomo o scritti in un file per un uso futuro. Questo capitolo discuterà alcune delle possibilità.

7.1 Formattazione Avanzata dell'Output

Finora abbiamo incontrato due modi di scrivere valori: *espressioni* e la funzione `print()`. (Un terzo modo è utilizzare il metodo `write()` degli oggetti file; il file di output standard può essere referenziato come `sys.stdout`. Vedi la Libreria di Riferimento per ulteriori informazioni a riguardo.)

Spesso desidererai avere più controllo sulla formattazione del tuo output rispetto al semplice stampare valori separati da spazi. Ci sono diversi modi per formattare l'output.

- Per usare *formatted string literals*, inizia una stringa con `f` o `F` prima del segno di apertura della virgoletta o della tripla virgoletta. Dentro questa stringa, puoi scrivere un'espressione Python tra i caratteri `{` e `}`, che possono riferirsi a variabili o valori letterali.

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

- Il metodo `str.format()` delle stringhe richiede più sforzo manuale. Utilizzerai comunque `{` e `}` per indicare dove una variabile sarà sostituita e potrai fornire direttive di formattazione dettagliate, ma dovrai anche fornire le informazioni da formattare. Nel blocco di codice seguente ci sono due esempi di come formattare variabili:

```
>>> yes_votes = 42_572_654
>>> total_votes = 85_705_149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes  {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

Nota come i `yes_votes` sono riempiti con spazi e con un segno negativo solo per i numeri negativi. L'esempio stampa anche `percentage` moltiplicato per 100, con 2 cifre decimali e seguito da un segno di percentuale (vedi `formatspec` per i dettagli).

- Infine, puoi gestire tutte le stringhe da solo usando operazioni di slicing e concatenazione per creare qualsiasi layout tu possa immaginare. Il tipo stringa ha alcuni metodi che eseguono operazioni utili per riempire le stringhe a una determinata larghezza di colonna.

Quando non hai bisogno di un output sofisticato ma vuoi solo una rapida visualizzazione di alcune variabili per scopi di debug, puoi convertire qualsiasi valore in una stringa con le funzioni `repr()` o `str()`.

La funzione `str()` è pensata per restituire rappresentazioni di valori che siano abbastanza leggibili, mentre `repr()` è pensata per generare rappresentazioni che possono essere lette dall'interprete (o genererà una `SyntaxError` se non esiste una sintassi equivalente). Per oggetti che non hanno una particolare rappresentazione per il consumo umano, `str()` restituirà lo stesso valore di `repr()`. Molti valori, come i numeri o strutture come liste e dizionari, hanno la stessa rappresentazione usando entrambe le funzioni. Le stringhe, in particolare, hanno due rappresentazioni distinte.

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...
>>> # The repr() of a string adds string quotes and backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
>>> repr((x, y, ('spam', 'eggs'))
"'(32.5, 40000, ('spam', 'eggs'))"
```

Il modulo `string` contiene una classe `Template` che offre un altro modo per sostituire valori nelle stringhe, utilizzando segnaposti come `$x` e sostituendoli con valori da un dizionario, ma offre un controllo molto minore sulla formattazione.

7.1.1 Stringhe Formattate Letterali

Formatted string literals (anche dette f-strings per brevità) ti permettono di includere il valore di espressioni Python all'interno di una stringa, prefissando la stringa con `f` o `F` e scrivendo espressioni come `{expression}`.

Uno specificatore di formato opzionale può seguire l'espressione. Questo permette un maggiore controllo su come il valore viene formattato. Il seguente esempio arrotonda π 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.
```

Passare un intero dopo `:` farà sì che quel campo abbia una larghezza minima di un certo numero di caratteri. Questo è utile per allineare le colonne.

```
>>> 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
```

Possono essere usati altri modificatori per convertire il valore prima che venga formattato. `'!a'` applica `ascii()`, `'!s'` applica `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'.
```

Lo specificatore `=` può essere usato per espandere un'espressione al testo dell'espressione, un segno di uguale e poi la rappresentazione dell'espressione valutata:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

Vedi [self-documenting expressions](#) per maggiori informazioni sullo specificatore `=`. Per un riferimento su queste specifiche di formato, consulta la guida di riferimento per [formatspec](#).

7.1.2 Il Metodo String `format()`

L'uso di base del metodo `str.format()` sembra così:

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

Le parentesi e i caratteri al loro interno (chiamati campi di formato) sono sostituiti dagli oggetti passati nel metodo `str.format()`. Un numero nelle parentesi può essere usato per riferirsi alla posizione dell'oggetto passato nel metodo `str.format()`.

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

Se vengono utilizzati argomenti keyword nel metodo `str.format()`, i loro valori vengono referenziati usando il nome dell'argomento.

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

Gli argomenti posizionali e keyword possono essere combinati arbitrariamente:

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

Se hai una stringa di formato molto lunga che non vuoi dividere, sarebbe utile poter riferire le variabili da formattare per nome invece che per posizione. Questo può essere fatto semplicemente passando il dizionario e usando le parentesi quadre `'[]'` per accedere alle 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
```

Questo può essere fatto anche passando il dizionario `table` come argomenti keyword usando 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 è particolarmente utile in combinazione con la funzione built-in `vars()`, che restituisce un dizionario contenente tutte le variabili locali:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k + '}'; ' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

Ad esempio, le seguenti righe producono un insieme di colonne ordinatamente allineate che danno interi e loro quadrati e cubi:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
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 panoramica completa della formattazione delle stringhe con `str.format()`, vedi `formatstrings`.

7.1.3 Formattazione Manuale delle Stringhe

Ecco la stessa tabella di quadrati e cubi, formattata manualmente:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     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
```

(Nota che l'unico spazio tra ogni colonna è stato aggiunto dal modo in cui funziona `print()`: aggiunge sempre spazi tra i suoi argomenti.)

Il metodo `str.rjust()` degli oggetti stringa giustifica a destra una stringa in un campo di una data larghezza riempiendola con spazi sulla sinistra. Esistono metodi simili `str.ljust()` e `str.center()`. Questi metodi non scrivono nulla, restituiranno solo una nuova stringa. Se la stringa di input è troppo lunga, non viene troncata, ma restituita invariata; questo comprometterà l'allineamento delle colonne ma di solito è meglio dell'alternativa, che

sarebbe mentire su un valore. (Se desideri davvero la troncatura, puoi sempre aggiungere un'operazione di slicing, come in `x.ljust(n)[:n]`.)

C'è un altro metodo, `str.zfill()`, che riempie una stringa numerica a sinistra con zeri. Comprende i segni più e meno:

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

7.1.4 Vecchia formattazione delle stringhe

The `%` operator (modulo) can also be used for string formatting. Given `format % values` (where *format* is a string), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. This operation is commonly known as string interpolation. For example:

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

Maggiori informazioni possono essere trovate nella sezione `old-string-formatting`.

7.2 Lettura e Scrittura di File

`open()` restituisce un *file object*, ed è più comunemente usato con due argomenti posizionali e uno keyword: `open(filename, mode, encoding=None)`

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

Il primo argomento è una stringa che contiene il nome del file. Il secondo argomento è un'altra stringa che contiene alcuni caratteri che descrivono il modo in cui il file verrà utilizzato. *mode* può essere `'r'` quando il file verrà solo letto, `'w'` per solo scrivere (un file esistente con lo stesso nome verrà cancellato), e `'a'` apre il file per l'aggiunta dati; qualsiasi dato scritto nel file viene automaticamente aggiunto alla fine. `'r+'` apre il file per sia lettura che scrittura. L'argomento *mode* è opzionale; `'r'` sarà assunto se omissso.

Normalmente, i file vengono aperti in *text mode*, ciò significa che leggi e scrivi stringhe dal e nel file, che sono codificate in una specifica *encoding*. Se *encoding* non è specificato, il valore predefinito dipende dal sistema (vedi `open()`). Poiché UTF-8 è lo standard de facto moderno, `encoding="utf-8"` è raccomandato a meno che non si sappia che è necessario utilizzare una diversa codifica. Aggiungendo una `'b'` al *mode*, si apre il file in *binary mode*. I dati in modalità binaria sono letti e scritti come oggetti `bytes`. Non puoi specificare *encoding* quando apri file in modalità binaria.

In modalità testo, il valore predefinito durante la lettura è convertire i terminatori di riga specifici della piattaforma (`\n` su Unix, `\r\n` su Windows) in solo `\n`. Quando si scrive in modalità testo, il valore predefinito è convertire le occorrenze di `\n` nei terminatori specifici della piattaforma. Questa modifica dietro le quinte ai dati del file va bene per i file di testo, ma corromperà i dati binari come quelli nei file JPEG o EXE. Fai molta attenzione a usare modalità binaria quando leggi e scrivi tali file.

È buona pratica usare la keyword `with` quando si lavora con oggetti file. Il vantaggio è che il file viene chiuso correttamente dopo che la sua suite finisce, anche se viene generata un'eccezione a un certo punto. Usare `with` è anche molto più breve che scrivere blocchi equivalenti `try-finally`:

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
```

(continues on next page)

(continua dalla pagina precedente)

```
>>> f.closed
True
```

Se non stai usando la keyword `with`, allora dovresti chiamare `f.close()` per chiudere il file e liberare immediatamente qualsiasi risorsa di sistema utilizzata da esso.

⚠ Avvertimento

Chiamare `f.write()` senza usare la keyword `with` o chiamare `f.close()` **potrebbe** risultare nel fatto che gli argomenti di `f.write()` non vengano completamente scritti sul disco, anche se il programma termina con successo.

Dopo che un oggetto file è stato chiuso, sia tramite un'istruzione `with`, sia chiamando `f.close()`, i tentativi di utilizzare l'oggetto file falliranno 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

Gli altri esempi in questa sezione assumeranno che un oggetto file chiamato `f` sia già stato creato.

Per leggere i contenuti di un file, chiama `f.read(size)`, che legge una certa quantità di dati e li restituisce come una stringa (in modalità testo) o un oggetto bytes (in modalità binaria). `size` è un argomento numerico opzionale. Quando `size` è omesso o negativo, verranno letti e restituiti tutti i contenuti del file; è un tuo problema se il file è due volte più grande della memoria del tuo computer. Altrimenti, al massimo `size` caratteri (in modalità testo) o `size` byte (in modalità binaria) vengono letti e restituiti. Se la fine del file è stata raggiunta, `f.read()` restituirà una stringa vuota (`''`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` legge una singola riga dal file; un carattere di newline (`\n`) viene lasciato alla fine della stringa e viene omesso solo sull'ultima riga del file se il file non termina con un newline. Questo rende il valore di ritorno non ambiguo; se `f.readline()` restituisce una stringa vuota, la fine del file è stata raggiunta, mentre una linea vuota è rappresentata da `'\n'`, una stringa contenente solo un newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

Per leggere le righe da un file, puoi iterare sull'oggetto file. Questo è efficiente in termini di memoria, veloce e porta a un codice semplice:

```
>>> for line in f:
...     print(line, end='')
...
```

(continues on next page)

(continua dalla pagina precedente)

```
This is the first line of the file.
Second line of the file
```

Se vuoi leggere tutte le righe di un file in una lista puoi anche usare `list(f)` o `f.readlines()`.

`f.write(string)` scrive i contenuti di *string* nel file, restituendo il numero di caratteri scritti.

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

Altri tipi di oggetti devono essere convertiti – o in una stringa (in modalità testo) o in un oggetto bytes (in modalità binaria) – prima di scriverli:

```
>>> value = ('the answer', 42)
>>> s = str(value) # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` restituisce un intero che indica la posizione corrente dell'oggetto file nel file, rappresentata come numero di byte dall'inizio del file quando in modalità binaria e un numero opaco quando in modalità testo.

Per cambiare la posizione dell'oggetto file, usa `f.seek(offset, whence)`. La posizione è calcolata sommando *offset* a un punto di riferimento; il punto di riferimento è selezionato dall'argomento *whence*. Un valore *whence* di 0 misura dall'inizio del file, 1 usa la posizione corrente del file, e 2 usa la fine del file come punto di riferimento. *whence* può essere omesso e predefinito a 0, utilizzando l'inizio del file come punto di riferimento.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5) # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

Nei file di testo (quelli aperti senza una `b` nella stringa di modalità), sono consentiti solo i `seek` relativi all'inizio del file (con l'eccezione del `seek` alla fine del file con `seek(0, 2)`) e gli unici valori *offset* validi sono quelli restituiti da `f.tell()`, o zero. Qualsiasi altro valore *offset* produce un comportamento non definito.

Gli oggetti file hanno alcuni metodi aggiuntivi, come `isatty()` e `truncate()` che sono meno frequentemente usati; consulta la Libreria di Riferimento per una guida completa agli oggetti file.

7.2.2 Salvare dati strutturati con json

Le stringhe possono essere facilmente scritte su e lette da un file. I numeri richiedono un po' più di sforzo, poiché il metodo `read()` restituisce solo stringhe, che dovranno essere passate a una funzione come `int()`, che prende una stringa come `'123'` e restituisce il suo valore numerico 123. Quando vuoi salvare tipi di dati più complessi come liste annidate e dizionari, il parsing e la serializzazione manuale diventano complicati.

Piuttosto che avere utenti che scrivono e fanno debug costantemente di codice per salvare tipi di dati complicati nei file, Python ti permette di usare il popolare formato di interscambio dati chiamato **JSON (JavaScript Object Notation)**. Il modulo standard chiamato `json` può prendere gerarchie di dati Python e convertirle in rappresentazioni di stringhe; questo processo è chiamato *serializzazione*. Ricostruire i dati dalla rappresentazione di stringa è chiamato *deserializzazione*. Tra la serializzazione e la deserializzazione, la stringa che rappresenta l'oggetto può essere stata memorizzata in un file o dato, o inviata su una connessione di rete a una macchina lontana.

i Nota

Il formato JSON è comunemente usato dalle applicazioni moderne per permettere lo scambio di dati. Molti programmatori sono già familiari con esso, il che lo rende una buona scelta per l'interoperabilità.

Se hai un oggetto `x`, puoi visualizzare la sua rappresentazione JSON della stringa con una semplice riga di codice:

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

Un'altra variante della funzione `dumps()`, chiamata `dump()`, semplicemente serializza l'oggetto in un oggetto *text file*. Quindi se `f` è un oggetto *text file* aperto per la scrittura, possiamo fare questo:

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

Per decodificare l'oggetto di nuovo, se `f` è un oggetto *binary file* o *text file* che è stato aperto per la lettura:

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

i Nota

I file JSON devono essere codificati in UTF-8. Usa `encoding="utf-8"` quando apri il file JSON come *text file* sia per la lettura che per la scrittura.

Questa semplice tecnica di serializzazione può gestire liste e dizionari, ma serializzare istanze di classi arbitrari in JSON richiede un po' di sforzo extra. Il riferimento per il modulo `json` contiene una spiegazione di questo.

➡ Vedi anche

`pickle` - il modulo `pickle`

Contrariamente a *JSON*, *pickle* è un protocollo che permette la serializzazione di oggetti Python arbitrariamente complessi. Come tale, è specifico per Python e non può essere usato

Finora i messaggi di errore sono stati solo menzionati, ma se hai provato gli esempi probabilmente ne hai visti alcuni. Ci sono (almeno) due tipi distinti di errori: *errori di sintassi* ed *eccezioni*.

8.1 Errori di Sintassi

Gli errori di sintassi, noti anche come errori di parsing, sono forse il tipo più comune di reclamo che si ottiene mentre si sta ancora imparando Python:

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

The parser repeats the offending line and displays little arrows pointing at the token in the line where the error was detected. The error may be caused by the absence of a token *before* the indicated token. In the example, the error is detected at the function `print()`, since a colon (`:`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

8.2 Eccezioni

Anche se un'istruzione o un'espressione è sintatticamente corretta, può causare un errore quando si tenta di eseguirla. Gli errori rilevati durante l'esecuzione sono chiamati *eccezioni* e non sono incondizionatamente fatali: imparerai presto come gestirli nei programmi Python. La maggior parte delle eccezioni non viene gestita dai programmi, tuttavia, e si traduce in messaggi di errore come mostrato qui:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    10 * (1/0)
        ~~~
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
```

(continues on next page)

(continua dalla pagina precedente)

```

File "<stdin>", line 1, in <module>
    4 + spam*3
      ^^^^
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    '2' + 2
    ~~~~^~~
TypeError: can only concatenate str (not "int") to str

```

L'ultima linea del messaggio di errore indica cosa è successo. Le eccezioni hanno tipi diversi, e il tipo viene stampato come parte del messaggio: i tipi nell'esempio sono `ZeroDivisionError`, `NameError` e `TypeError`. La stringa stampata come tipo di eccezione è il nome dell'eccezione built-in che si è verificata. Questo è vero per tutte le eccezioni built-in, ma non deve essere vero per le eccezioni definite dall'utente (anche se è una convenzione utile). I nomi delle eccezioni standard sono identificatori built-in (non parole chiave riservate).

La restante parte della linea fornisce dettagli basati sul tipo di eccezione e su cosa l'ha causata.

La parte precedente del messaggio di errore mostra il contesto in cui si è verificata l'eccezione, sotto forma di traccia dello stack. In generale, contiene una traccia dello stack che elenca le linee di origine; tuttavia, non visualizzerà linee lette dall'input standard.

`builtin-exceptions` elenca le eccezioni built-in e i loro significati.

8.3 Gestione delle Eccezioni

È possibile scrivere programmi che gestiscono eccezioni selezionate. Guarda il seguente esempio, che chiede all'utente un input finché non viene inserito un intero valido, ma consente all'utente di interrompere il programma (usando `Control-C` o cosa supporta il sistema operativo); nota che un'interruzione generata dall'utente è segnalata sollevando l'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 come segue.

- Per prima cosa, viene eseguita la *clausola try* (l'istruzione o le istruzioni tra le parole chiave `try` e `except`).
- Se non si verifica alcuna eccezione, la *clausola except* viene saltata e l'esecuzione dell'istruzione `try` è terminata.
- Se si verifica un'eccezione durante l'esecuzione della *clausola try*, il resto della *clausola* viene saltato. Poi, se il suo tipo corrisponde all'eccezione nominata dopo la parola chiave `except`, viene eseguita la *clausola except*, e quindi l'esecuzione continua dopo il blocco `try/except`.
- Se si verifica un'eccezione che non corrisponde all'eccezione nominata nella *clausola except*, viene passata alle istruzioni `try` esterne; se non viene trovato alcun gestore, è un'eccezione non gestita e l'esecuzione si interrompe con un messaggio di errore.

Un'istruzione `try` può avere più di una *clausola except*, per specificare gestori per diverse eccezioni. Al massimo verrà eseguito un gestore. I gestori gestiscono solo le eccezioni che si verificano nella corrispondente *clausola try*, non in altri gestori della stessa istruzione `try`. Una *clausola except* può nominare più eccezioni come una tupla tra parentesi, per esempio:

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

Una classe in una clausola `except` corrisponde a eccezioni che sono istanze della classe stessa o di una delle sue classi derivate (ma non viceversa — una *clausola except* che elenca una classe derivata non corrisponde a istanze delle sue classi base). Ad esempio, il seguente codice stamperà B, C, D in quell'ordine:

```
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")
```

Nota che se le *clausole except* fossero invertite (con `except B` prima), avrebbero stampato B, B, B — la prima *clausola except* che corrisponde viene attivata.

Quando si verifica un'eccezione, essa può avere valori associati, noti anche come *argomenti* dell'eccezione. La presenza e il tipo degli argomenti dipendono dal tipo di eccezione.

La *clausola except* può specificare una variabile dopo il nome dell'eccezione. La variabile è legata all'istanza dell'eccezione che tipicamente ha un attributo `args` che memorizza gli argomenti. Per comodità, i tipi di eccezioni built-in definiscono `__str__()` per stampare tutti gli argomenti senza accedere esplicitamente a `.args`.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception type
...     print(inst.args)    # arguments stored in .args
...     print(inst)         # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args    # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

L'output di `__str__()` dell'eccezione viene stampato come l'ultima parte (“dettaglio”) del messaggio per le eccezioni non gestite.

`BaseException` è la classe base comune di tutte le eccezioni. Uno dei suoi sottoclassi, `Exception`, è la classe base di tutte le eccezioni non fatali. Le eccezioni che non sono sottoclassi di `Exception` non sono tipicamente gestite, perché vengono utilizzate per indicare che il programma dovrebbe terminare. Queste includono `SystemExit` che

è sollevata da `sys.exit()` e `KeyboardInterrupt` che è sollevata quando un utente desidera interrompere il programma.

`Exception` può essere utilizzata come un jolly che cattura (quasi) tutto. Tuttavia, è buona pratica essere il più specifici possibile con i tipi di eccezioni che intendiamo gestire, e permettere a qualsiasi eccezione inattesa di propagarsi.

Il pattern più comune per la gestione di `Exception` è stamparla o registrarla e poi rilanciarla (consentendo a un chiamante di gestire l'eccezione altrettanto):

```
import sys

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

L'istruzione `try ... except` ha una *clausola else* opzionale, che, quando presente, deve seguire tutte le *clausole except*. È utile per il codice che deve essere eseguito se la *clausola try* non solleva un'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()
```

L'uso della *clausola else* è migliore che aggiungere codice ulteriore alla *clausola try* perché evita di catturare accidentalmente un'eccezione che non è stata sollevata dal codice protetto dall'istruzione `try ... except`.

I gestori delle eccezioni non gestiscono solo le eccezioni che si verificano immediatamente nella *clausola try*, ma anche quelle che si verificano all'interno delle funzioni che vengono chiamate (anche indirettamente) nella *clausola 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 Sollevare Eccezioni

L'istruzione `raise` permette al programmatore di forzare il verificarsi di una specifica eccezione. Per esempio:

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

L'unico argomento per `raise` indica l'eccezione da sollevare. Questo deve essere o un'istanza di eccezione o una classe di eccezione (una classe che deriva da `BaseException`, come `Exception` o una delle sue sottoclassi). Se viene passata una classe di eccezione, verrà istanziata implicitamente chiamando il suo costruttore senza argomenti:

```
raise ValueError # shorthand for 'raise ValueError()'
```

Se è necessario determinare se un'eccezione è stata sollevata ma non si intende gestirla, una forma più semplice dell'istruzione `raise` consente 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>
    raise NameError('HiThere')
NameError: HiThere
```

8.5 Collegamento delle Eccezioni

Se si verifica un'eccezione non gestita all'interno di una sezione `except`, avrà l'eccezione che viene gestita allegata e inclusa nel messaggio di errore:

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    open("database.sqlite")
    ~~~~^~~~~~
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

Per indicare che un'eccezione è una diretta conseguenza di un'altra, l'istruzione `raise` permette una clausola opzionale `from`:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

Questo può essere utile quando si stanno trasformando le eccezioni. 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>
    func()
    ~~~^^
  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>
    raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

Consente anche di disabilitare il collegamento automatico delle eccezioni utilizzando l'idioma `from None`:

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

Per ulteriori informazioni sulla meccanica del collegamento, vedi `bltin-exceptions`.

8.6 Eccezioni Definite dall'Utente

I programmi possono nominare le proprie eccezioni creando una nuova classe di eccezione (vedi [Classi](#) per ulteriori informazioni sulle classi Python). Le eccezioni devono essere tipicamente derivate dalla classe `Exception`, direttamente o indirettamente.

Si possono definire classi di eccezione che fanno qualsiasi cosa possa fare qualsiasi altra classe, ma di solito vengono mantenute semplici, spesso offrendo solo un numero di attributi che consentono di estrarre informazioni sull'errore dai gestori per l'eccezione.

La maggior parte delle eccezioni è definita con nomi che terminano in «Error», simili alla denominazione delle eccezioni standard.

Molti moduli standard definiscono le proprie eccezioni per segnalare errori che possono verificarsi nelle funzioni che definiscono.

8.7 Definizione di Azioni di Pulizia

L'istruzione `try` ha un'altra clausola opzionale che è intesa a definire azioni di pulizia che devono essere eseguite in tutte le circostanze. Per esempio:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

Se è presente una clausola `finally`, la clausola `finally` verrà eseguita come ultimo compito prima che l'istruzione `try` completi. La clausola `finally` viene eseguita sia che l'istruzione `try` produca o meno un'eccezione. I seguenti punti discutono casi più complessi quando si verifica un'eccezione:

- Se si verifica un'eccezione durante l'esecuzione della clausola `try`, l'eccezione può essere gestita da una clausola `except`. Se l'eccezione non è gestita da una clausola `except`, l'eccezione viene rilanciata dopo che la clausola `finally` è stata eseguita.
- Un'eccezione potrebbe verificarsi durante l'esecuzione di una clausola `except` o `else`. Anche in questo caso, l'eccezione viene rilanciata dopo che la clausola `finally` è stata eseguita.
- Se la clausola `finally` esegue un'istruzione `break`, `continue` o `return`, le eccezioni non vengono rilanciate.
- Se l'istruzione `try` raggiunge un'istruzione `break`, `continue` o `return`, la clausola `finally` verrà eseguita appena prima dell'esecuzione dell'istruzione `break`, `continue` o `return`.
- Se una clausola `finally` include un'istruzione `return`, il valore restituito sarà quello dell'istruzione `return` della clausola `finally`, non il valore dell'istruzione `return` della clausola `try`.

Per esempio:

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

Un esempio più complicato:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
```

(continues on next page)

(continua dalla pagina precedente)

```

executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    divide("2", "1")
    ~~~~~^~~~~~
  File "<stdin>", line 3, in divide
    result = x / y
            ~^^~
TypeError: unsupported operand type(s) for /: 'str' and 'str'

```

Come puoi vedere, la clausola `finally` viene eseguita in ogni caso. L'eccezione `TypeError` sollevata dividendo due stringhe non è gestita dalla clausola `except` e quindi rilanciata dopo che la clausola `finally` è stata eseguita.

Nelle applicazioni del mondo reale, la clausola `finally` è utile per rilasciare risorse esterne (come file o connessioni di rete), indipendentemente dal successo dell'uso delle risorse.

8.8 Azioni di Pulizia Predefinite

Alcuni oggetti definiscono azioni di pulizia standard da intraprendere quando l'oggetto non è più necessario, indipendentemente dal fatto che l'operazione che usa l'oggetto sia riuscita o fallita. Guarda il seguente esempio, che tenta di aprire un file e stampare il suo contenuto sullo schermo.

```

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

```

Il problema con questo codice è che lascia il file aperto per un tempo indeterminato dopo che questa parte del codice ha terminato l'esecuzione. Questo non è un problema negli script semplici, ma può essere un problema per applicazioni più grandi. L'istruzione `with` consente agli oggetti come i file di essere usati in modo tale da garantire che siano sempre puliti rapidamente e correttamente.

```

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

```

Dopo che l'istruzione è stata eseguita, il file `f` è sempre chiuso, anche se si è verificato un problema durante l'elaborazione delle linee. Gli oggetti che, come i file, forniscono azioni di pulizia predefinite indicheranno ciò nella loro documentazione.

8.9 Sollevare e Gestire Eccezioni Multiple e Non Correlate

Ci sono situazioni in cui è necessario segnalare diverse eccezioni che si sono verificate. Questo è spesso il caso nei framework di concorrenza, quando diversi compiti possono aver fallito in parallelo, ma ci sono anche altri casi d'uso in cui è desiderabile continuare l'esecuzione e raccogliere più errori piuttosto che sollevare la prima eccezione.

L'eccezione built-in `ExceptionGroup` avvolge una lista di istanze di eccezione in modo tale che possano essere sollevate insieme. È un'eccezione a sé stante, quindi può essere catturata come qualsiasi altra eccezione.

```

>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> f()
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
|       f()
|       ~^^

```

(continues on next page)

(continua dalla pagina precedente)

```

|   File "<stdin>", line 3, in f
|       raise ExceptionGroup('there were problems', excs)
| ExceptionGroup: there were problems (2 sub-exceptions)
+-+----- 1 -----
|   OSError: error 1
+----- 2 -----
|   SystemError: error 2
+-----
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>

```

Usando `except*` invece di `except`, possiamo gestire selettivamente solo le eccezioni nel gruppo che corrispondono a un certo tipo. Nell'esempio seguente, che mostra un gruppo di eccezioni annidato, ogni clausola `except*` estrae dal gruppo le eccezioni di un certo tipo lasciando che tutte le altre eccezioni si propaghino ad altre clausole e, infine, vengano rilanciate.

```

>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 2, in <module>
|       f()
|       ~^^
|   File "<stdin>", line 2, in f
|       raise ExceptionGroup(
|         ...<12 lines>...
|     )
| ExceptionGroup: group1 (1 sub-exception)
+-+----- 1 -----

```

(continues on next page)

(continua dalla pagina precedente)

```

| ExceptionGroup: group2 (1 sub-exception)
+-+----- 1 -----
| RecursionError: 4
+-----
>>>

```

Nota che le eccezioni annidate in un gruppo di eccezioni devono essere istanze, non tipi. Questo perché in pratica le eccezioni sarebbero tipicamente quelle che sono state già sollevate e catturate dal programma, seguendo il seguente pattern:

```

>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...     raise ExceptionGroup("Test Failures", excs)
...

```

8.10 Arricchire le Eccezioni con Note

Quando un'eccezione viene creata per essere sollevata, viene solitamente inizializzata con informazioni che descrivono l'errore verificatosi. Ci sono casi in cui è utile aggiungere informazioni dopo che l'eccezione è stata catturata. Per questo scopo, le eccezioni hanno un metodo `add_note(note)` che accetta una stringa e la aggiunge alla lista di note dell'eccezione. Il rendering standard del traceback include tutte le note, nell'ordine in cui sono state aggiunte, dopo l'eccezione.

```

>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information
>>>

```

Per esempio, quando si raccolgono eccezioni in un gruppo di eccezioni, potremmo voler aggiungere informazioni di contesto per gli errori individuali. Nel seguente esempio, ogni eccezione nel gruppo ha una nota che indica quando si è verificato l'errore.

```

>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:

```

(continues on next page)

(continua dalla pagina precedente)

```

...     e.add_note(f'Happened in Iteration {i+1}')
...     excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
|       raise ExceptionGroup('We have some problems', excs)
| ExceptionGroup: We have some problems (3 sub-exceptions)
+----- 1 -----
| Traceback (most recent call last):
|   File "<stdin>", line 3, in <module>
|       f()
|       ~^^
|   File "<stdin>", line 2, in f
|       raise OSError('operation failed')
| OSError: operation failed
| Happened in Iteration 1
+----- 2 -----
| Traceback (most recent call last):
|   File "<stdin>", line 3, in <module>
|       f()
|       ~^^
|   File "<stdin>", line 2, in f
|       raise OSError('operation failed')
| OSError: operation failed
| Happened in Iteration 2
+----- 3 -----
| Traceback (most recent call last):
|   File "<stdin>", line 3, in <module>
|       f()
|       ~^^
|   File "<stdin>", line 2, in f
|       raise OSError('operation failed')
| OSError: operation failed
| Happened in Iteration 3
+-----
>>>

```


Le classi forniscono un mezzo per raggruppare dati e funzionalità insieme. Creare una nuova classe crea un nuovo *tipo* di oggetto, consentendo la creazione di nuove *istanze* di quel tipo. Ogni istanza di classe può avere attributi ad essa associati per mantenere il suo stato. Le istanze di classe possono anche avere metodi (definiti dalla sua classe) per modificarne lo stato.

Rispetto ad altri linguaggi di programmazione, il meccanismo delle classi di Python aggiunge classi con un minimo di nuova sintassi e semantica. È un misto dei meccanismi di classe trovati in C++ e Modula-3. Le classi Python forniscono tutte le funzionalità standard della programmazione orientata agli oggetti: il meccanismo di ereditarietà delle classi consente più classi di base, una classe derivata può sovrascrivere qualsiasi metodo della sua classe base o classi, e un metodo può chiamare il metodo di una classe di base con lo stesso nome. Gli oggetti possono contenere quantità e tipi di dati arbitrari. Come è vero per i moduli, le classi partecipano alla natura dinamica di Python: sono create a tempo di esecuzione e possono essere modificate ulteriormente dopo la creazione.

Nella terminologia di C++, normalmente i membri della classe (compresi i membri dei dati) sono *pubblici* (tranne vedere sotto *Variabili Private*), e tutte le funzioni membro sono *virtuali*. Come in Modula-3, non ci sono abbreviazioni per fare riferimento ai membri dell'oggetto dai suoi metodi: la funzione del metodo è dichiarata con un primo argomento esplicito che rappresenta l'oggetto, che è fornito implicitamente dalla chiamata. Come in Smalltalk, le classi stesse sono oggetti. Questo fornisce la semantica per l'importazione e il ridenominare. A differenza di C++ e Modula-3, i tipi integrati possono essere utilizzati come classi di base per l'estensione da parte dell'utente. Inoltre, come in C++, la maggior parte degli operatori integrati con una sintassi speciale (operatori aritmetici, indicizzazione ecc.) possono essere ridefiniti per le istanze della classe.

(Mancando una terminologia universalmente accettata per parlare di classi, farò un uso occasionale dei termini Smalltalk e C++. Utilizzerò i termini di Modula-3, poiché la sua semantica orientata agli oggetti è più vicina a quella di Python rispetto a C++, ma mi aspetto che pochi lettori ne abbiano sentito parlare.)

9.1 Una parola su nomi e oggetti

Gli oggetti hanno individualità e più nomi (in più ambiti) possono essere vincolati allo stesso oggetto. Questo è noto come *aliasing* in altri linguaggi. Questo di solito non è apprezzato a una prima occhiata a Python e può essere tranquillamente ignorato quando si tratta di tipi di base immutabili (numeri, stringhe, tuple). Tuttavia, l'*aliasing* ha un effetto possibilmente sorprendente sulla semantica del codice Python che coinvolge oggetti mutabili come liste, dizionari e la maggior parte degli altri tipi. Questo è di solito usato a vantaggio del programma, poiché gli alias si comportano come puntatori in alcuni aspetti. Ad esempio, passare un oggetto è economico poiché viene passato solo un puntatore dall'implementazione; e se una funzione modifica un oggetto passato come argomento, il chiamante vedrà il cambiamento — questo elimina la necessità di due diversi meccanismi di passaggio degli argomenti come in Pascal.

9.2 Visibilità e spazi dei nomi in Python

Prima di introdurre le classi, devo prima dirti qualcosa sulle regole di visibilità di Python. Le definizioni di classe fanno alcuni trucchi interessanti con gli spazi dei nomi e devi sapere come funzionano le visibilità e gli spazi dei nomi per capire appieno cosa sta succedendo. Per inciso, la conoscenza su questo argomento è utile per qualsiasi programmatore Python avanzato.

Iniziamo con alcune definizioni.

Uno *spazio dei nomi* è una mappatura da nomi a oggetti. La maggior parte degli spazi dei nomi è attualmente implementata come dizionari Python, ma normalmente non è percepibile in alcun modo (tranne per le prestazioni) e potrebbe cambiare in futuro. Esempi di spazi dei nomi sono: l'insieme di nomi integrati (contenente funzioni come `abs()` e nomi di eccezioni integrate); i nomi globali in un modulo; e i nomi locali in una chiamata di funzione. In un certo senso, l'insieme di attributi di un oggetto forma anche uno spazio dei nomi. La cosa importante da sapere sugli spazi dei nomi è che non c'è assolutamente alcuna relazione tra nomi in spazi dei nomi diversi; ad esempio, due moduli diversi possono entrambi definire una funzione `massimizza` senza confusione — gli utenti dei moduli devono prefissarlo con il nome del modulo.

A proposito, uso la parola *attributo* per qualsiasi nome che segue un punto — ad esempio, nell'espressione `z.reale`, `reale` è un attributo dell'oggetto `z`. Più precisamente, i riferimenti ai nomi nei moduli sono riferimenti agli attributi: nell'espressione `nomemodulo.nomefunzione`, `nomemodulo` è un oggetto modulo e `nomefunzione` è un suo attributo. In questo caso c'è una corrispondenza diretta tra gli attributi del modulo e i nomi globali definiti nel modulo: condividono lo stesso spazio dei nomi!¹

Gli attributi possono essere di sola lettura o scrivibili. In quest'ultimo caso, l'assegnazione agli attributi è possibile. Gli attributi del modulo sono scrivibili: puoi scrivere `nomemodulo.larisposta = 42`. Gli attributi scrivibili possono anche essere eliminati con l'istruzione `del`. Ad esempio, `del nomemodulo.larisposta` rimuoverà l'attributo `larisposta` dall'oggetto denominato da `nomemodulo`.

Gli spazi dei nomi sono creati in momenti diversi e hanno durate diverse. Lo spazio dei nomi che contiene i nomi integrati è creato quando l'interprete Python si avvia e non viene mai eliminato. Lo spazio dei nomi globale per un modulo è creato quando la definizione del modulo viene letta; normalmente, gli spazi dei nomi dei moduli durano anche fino a quando l'interprete non si arresta. Le istruzioni eseguite dall'invocazione di livello superiore dell'interprete, lette da un file di script o interattivamente, sono considerate parte di un modulo chiamato `__main__`, quindi hanno il loro proprio spazio dei nomi globale. (I nomi integrati in realtà vivono anche in un modulo; questo si chiama `builtins`.)

Lo spazio dei nomi locale per una funzione è creato quando la funzione viene chiamata e eliminato quando la funzione restituisce o genera un'eccezione che non viene gestita all'interno della funzione. (In realtà, dimenticare sarebbe un modo migliore per descrivere ciò che accade effettivamente.) Naturalmente, le invocazioni ricorsive hanno ciascuna il proprio spazio dei nomi locale.

Uno *scope* è una regione testuale di un programma Python in cui uno spazio dei nomi è direttamente accessibile. «Direttamente accessibile» qui significa che un riferimento non qualificato a un nome tenta di trovare il nome nello spazio dei nomi.

Anche se gli scope sono determinati staticamente, vengono utilizzati dinamicamente. In qualsiasi momento durante l'esecuzione, ci sono 3 o 4 scope annidati i cui spazi dei nomi sono direttamente accessibili:

- lo scope più interno, che viene cercato per primo, contiene i nomi locali
- gli scope di eventuali funzioni contenenti, che vengono cercati a partire dallo scope contenente più vicino, contengono nomi non locali, ma anche non globali
- il penultimo scope contiene i nomi globali del modulo corrente
- lo scope più esterno (cercato per ultimo) è lo spazio dei nomi che contiene i nomi integrati

Se un nome è dichiarato globale, allora tutti i riferimenti e le assegnazioni vanno direttamente allo scope penultimo contenente i nomi globali del modulo. Per rilegare le variabili trovate al di fuori dello scope più interno, può essere utilizzata l'istruzione `nonlocal`; se non dichiarate `nonlocal`, quelle variabili sono di sola lettura (un tentativo di

¹ Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module's namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

scrivere su una variabile del genere creerà semplicemente una *nuova* variabile locale nello scope più interno, lasciando invariata la variabile esterna con lo stesso nome).

Di solito, lo scope locale fa riferimento ai nomi locali della funzione (testualmente) corrente. Al di fuori delle funzioni, lo scope locale fa riferimento allo stesso spazio dei nomi dello scope globale: lo spazio dei nomi del modulo. Le definizioni di classe pongono un altro spazio dei nomi nello scope locale.

È importante rendersi conto che gli scope sono determinati testualmente: lo scope globale di una funzione definita in un modulo è lo spazio dei nomi del modulo, non importa da dove o con quale alias la funzione viene chiamata. D'altra parte, la ricerca effettiva dei nomi viene fatta dinamicamente, a tempo di esecuzione — tuttavia, la definizione del linguaggio sta evolvendo verso la risoluzione statica dei nomi, a tempo di «compilazione», quindi non fare affidamento sulla risoluzione dinamica dei nomi! (Infatti, le variabili locali sono già determinate staticamente.)

Una particolarità speciale di Python è che – se non è in vigore alcuna istruzione `global` o `nonlocal` – le assegnazioni ai nomi vanno sempre nello scope più interno. Le assegnazioni non copiano i dati — vincolano solo i nomi agli oggetti. Lo stesso vale per le eliminazioni: l'istruzione `del x` rimuove il vincolo di `x` dallo spazio dei nomi riferito dallo scope locale. In realtà, tutte le operazioni che introducono nuovi nomi utilizzano lo scope locale: in particolare, le istruzioni `import` e le definizioni di funzioni vincolano il nome del modulo o della funzione nello scope locale.

L'istruzione `global` può essere utilizzata per indicare che particolari variabili vivono nello scope globale e dovrebbero essere rilegate lì; l'istruzione `nonlocal` indica che particolari variabili vivono in uno scope circoscritto e dovrebbero essere rilegate lì.

9.2.1 Esempio di visibilità e spazi dei nomi

Questo è un esempio che dimostra come fare riferimento ai diversi scope e spazi dei nomi e come `global` e `nonlocal` influenzano il vincolo 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("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

L'output del codice di esempio è:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Nota come l'assegnazione *locale* (che è predefinita) non abbia cambiato il vincolo di `spam` di `scope_test`. L'assegnazione `nonlocal` ha cambiato il vincolo di `spam` di `scope_test` e l'assegnazione `global` ha cambiato il vincolo a livello di modulo.

Si può anche vedere che non c'era alcun vincolo precedente per *spam* prima dell'assegnazione `global`.

9.3 Una prima occhiata alle classi

Le classi introducono un po' di nuova sintassi, tre nuovi tipi di oggetti e alcune nuove semantiche.

9.3.1 Sintassi della definizione di classe

La forma più semplice di definizione di classe appare così:

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

Le definizioni di classe, come le definizioni di funzione (istruzioni `def`), devono essere eseguite prima di avere effetto. (Potresti concepirle come collocare una definizione di classe in un ramo di un'istruzione `if` o all'interno di una funzione.)

In pratica, le istruzioni all'interno di una definizione di classe saranno solitamente definizioni di funzioni, ma sono ammesse e talvolta utili — torneremo su questo più tardi. Le definizioni di funzioni all'interno di una classe hanno normalmente una forma peculiare di elenco degli argomenti, dettata dalle convenzioni di chiamata per i metodi — anche questo è spiegato più tardi.

Quando una definizione di classe viene inserita, viene creato un nuovo spazio dei nomi e utilizzato come scope locale — quindi, tutte le assegnazioni alle variabili locali vanno in questo nuovo spazio dei nomi. In particolare, qui le definizioni di funzioni vincolano il nome della nuova funzione.

Quando una definizione di classe viene lasciata normalmente (tramite la `fine`), viene creato un *oggetto classe*. Questo è essenzialmente un wrapper attorno ai contenuti dello spazio dei nomi creato dalla definizione di classe; impareremo di più sugli oggetti classe nella sezione successiva. Lo scope locale originale (quello in vigore appena prima che la definizione di classe fosse inserita) viene ripristinato e l'oggetto classe è vincolato qui al nome della classe dato dall'istestazione della definizione di classe (`NomeClasse` nell'esempio).

9.3.2 Oggetti della Classe

Gli oggetti della classe supportano due tipi di operazioni: riferimenti agli attributi e istanziazione.

I *riferimenti agli attributi* utilizzano la sintassi standard utilizzata per tutti i riferimenti agli attributi in Python: `oggetto.nome`. I nomi degli attributi validi sono tutti i nomi che erano nello spazio dei nomi della classe quando l'oggetto classe è stato creato. Quindi, se la definizione della classe apparisse così:

```
class MyClass:
    """A simple example class"""
    i = 12345

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

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

L'*istanziamento* della classe utilizza la notazione funzionale. Basta immaginare che l'oggetto classe sia una funzione senza parametri che restituisce una nuova istanza della classe. Per esempio (assumendo la classe sopra menzionata):

```
x = MyClass()
```

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

L'operazione di istanziazione («chiamare» un oggetto classe) crea un oggetto vuoto. Molte classi preferiscono creare oggetti con istanze personalizzate per uno stato iniziale specifico. Pertanto, una classe può definire un metodo speciale chiamato `__init__()`, come questo:

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

Quando una classe definisce un metodo `__init__()`, l'istanziamento della classe invoca automaticamente `__init__()` per la nuova istanza della classe creata. Quindi, in questo esempio, una nuova istanza inizializzata può essere ottenuta con:

```
x = MyClass()
```

Ovviamente, il metodo `__init__()` può avere argomenti per una maggiore flessibilità. In tal caso, gli argomenti forniti all'operatore di istanziazione della classe vengono passati a `__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

Ora, cosa possiamo fare con gli oggetti istanza? Le uniche operazioni comprese dagli oggetti istanza sono i riferimenti agli attributi. Ci sono due tipi di nomi di attributi validi: attributi dati e metodi.

Gli *attributi dati* corrispondono alle «variabili di istanza» in Smalltalk, e ai «membri dati» in C++. Gli attributi dati non devono essere dichiarati; come le variabili locali, nascono nel momento in cui vengono assegnati per la prima volta. Per esempio, se `x` è l'istanza di `MyClass` creata sopra, il seguente pezzo di codice stamperà il valore 16, senza lasciare traccia:

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

The other kind of instance attribute reference is a *method*. A method is a function that «belongs to» an object.

I nomi dei metodi validi di un oggetto istanza dipendono dalla sua classe. Per definizione, tutti gli attributi di una classe che sono oggetti funzione definiscono i metodi corrispondenti delle sue istanze. Quindi nel nostro esempio, `x.f` è un riferimento a un metodo valido, poiché `MyClass.f` è una funzione, ma `x.i` no, poiché `MyClass.i` non lo è. Tuttavia, `x.f` non è la stessa cosa di `MyClass.f` — è un *oggetto metodo*, non un oggetto funzione.

9.3.4 Oggetti Metodo

Di solito, un metodo viene chiamato subito dopo essere stato associato:

```
x.f()
```

Nell'esempio `MyClass`, questo restituirà la stringa `'hello world'`. Tuttavia, non è necessario chiamare un metodo subito: `x.f` è un oggetto metodo e può essere memorizzato e chiamato in un secondo momento. Per esempio:

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

continuerà a stampare `hello world` fino alla fine dei tempi.

Cosa succede esattamente quando un metodo viene chiamato? Potreste aver notato che `x.f()` è stato chiamato senza argomento sopra, anche se la definizione della funzione per `f()` specifica un argomento. Che fine ha fatto l'argomento? Sicuramente Python solleva un'eccezione quando una funzione che richiede un argomento viene chiamata senza alcuno — anche se l'argomento non è effettivamente usato...

In realtà, potreste aver indovinato la risposta: la caratteristica speciale dei metodi è che l'istanza dell'oggetto viene passata come primo argomento della funzione. Nel nostro esempio, la chiamata `x.f()` è esattamente equivalente a `MyClass.f(x)`. In generale, chiamare un metodo con una lista di n argomenti è equivalente a chiamare la funzione corrispondente con una lista di argomenti creata inserendo l'istanza del metodo prima del primo argomento.

In generale, i metodi funzionano come segue. Quando viene referenziato un attributo non dato di un'istanza, viene cercata la classe dell'istanza. Se il nome denota un attributo di classe valido che è un oggetto funzione, i riferimenti all'istanza dell'oggetto e all'oggetto funzione vengono impacchettati in un oggetto metodo. Quando l'oggetto metodo viene chiamato con una lista di argomenti, viene costruita una nuova lista di argomenti dall'istanza dell'oggetto e la lista degli argomenti, e l'oggetto funzione viene chiamato con questa nuova lista di argomenti.

9.3.5 Variabili di Classe e Istanza

Generalmente parlando, le variabili di istanza sono per i dati unici a ciascuna istanza e le variabili di classe sono per gli attributi e i metodi condivisi da tutte le istanze della classe:

```
class Dog:

    kind = 'canine'          # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                # shared by all dogs
'canine'
>>> e.kind                # shared by all dogs
'canine'
>>> d.name                # unique to d
'Fido'
>>> e.name                # unique to e
'Buddy'
```

Come discusso in *Una parola su nomi e oggetti*, i dati condivisi possono avere effetti potenzialmente sorprendenti coinvolgendo oggetti *mutable* come liste e dizionari. Per esempio, la lista *tricks* nel seguente codice non dovrebbe essere usata come variabile di classe perché solo una singola lista verrebbe condivisa da tutte le istanze di *Dog*:

```
class Dog:

    tricks = []             # mistaken use of a class variable

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

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

(continues on next page)

(continua dalla pagina precedente)

```
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks          # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Il design corretto della classe dovrebbe usare una variabile di istanza invece:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    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 di attributo appare sia in un'istanza che in una classe, la ricerca degli attributi dà priorità all'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 dati possono essere referenziati dai metodi così come dagli utenti ordinari («clienti») di un oggetto. In altre parole, le classi non sono utilizzabili per implementare tipi di dati astratti puri. Infatti, nulla in Python permette di imporre l'incapsulamento dei dati — è tutto basato su convenzioni. (D'altra parte, l'implementazione di Python, scritta in C, può nascondere completamente i dettagli di implementazione e controllare l'accesso a un oggetto se necessario; questo può essere utilizzato dalle estensioni a Python scritte in C.)

I clienti dovrebbero usare gli attributi dati con cautela — i clienti potrebbero infrangere gli invarianti mantenuti dai metodi alterando i loro attributi dati. Si noti che i clienti possono aggiungere attributi dati propri a un oggetto istanza senza influenzare la validità dei metodi, purché vengano evitati conflitti di nomi — ancora una volta, una convenzione di nomenclatura può risparmiare molti mal di testa.

Non c'è alcuna scorciatoia per riferirsi agli attributi dati (o ad altri metodi!) dall'interno dei metodi. Trovo che questo in realtà aumenti la leggibilità dei metodi: non c'è possibilità di confondere variabili locali e variabili di istanza quando si scorre un metodo.

Spesso, il primo argomento di un metodo è chiamato `self`. Questa è niente più che una convenzione: il nome `self` non ha assolutamente alcun significato speciale per Python. Tieni presente, tuttavia, che non seguire la convenzione potrebbe rendere il tuo codice meno leggibile per altri programmatori Python, e non è nemmeno impossibile che venga scritto un programma *class browser* che si basa su tale convenzione.

Qualsiasi oggetto funzione che è un attributo di classe definisce un metodo per le istanze di quella classe. Non è necessario che la definizione della funzione sia testualmente inclusa nella definizione della classe: è anche possibile assegnare un oggetto funzione a una variabile locale nella classe. Per esempio:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Ora `f`, `g` e `h` sono tutti attributi della classe `C` che si riferiscono a oggetti funzione, e di conseguenza sono tutti metodi delle istanze della classe `C` — `h` essendo esattamente equivalente a `g`. Si noti che questa pratica di solito solo serve a confondere il lettore di un programma.

I metodi possono chiamare altri metodi utilizzando gli attributi di metodo dell'argomento `self`:

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

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

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

I metodi possono referenziare nomi globali allo stesso modo delle funzioni ordinarie. Lo scope globale associato a un metodo è il modulo contenente la sua definizione. (Una classe non è mai utilizzata come uno scope globale.) Sebbene raramente si incontri una buona ragione per usare dati globali in un metodo, ci sono molti usi legittimi dello scope globale: ad esempio, le funzioni e i moduli importati nello scope globale possono essere utilizzati dai metodi, così come le funzioni e le classi definite in esso. Di solito, la classe contenente il metodo è definita anch'essa in tale scope globale, e nella sezione successiva troveremo alcune buone ragioni per cui un metodo potrebbe voler referenziare la propria classe.

Ogni valore è un oggetto e quindi ha una *classe* (chiamata anche il suo *tipo*). È memorizzato come `object.__class__`.

9.5 Ereditarietà

Naturalmente, una caratteristica del linguaggio non sarebbe degna del nome «classe» senza supportare l'ereditarietà. La sintassi per la definizione di una classe derivata è simile:

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

(continues on next page)

(continua dalla pagina precedente)

```
.
<statement-N>
```

Il nome `BaseClassName` deve essere definito in uno spazio dei nomi accessibile dallo scope contenente la definizione della classe derivata. Al posto di un nome di classe base, sono permesse anche altre espressioni arbitrarie. Questo può essere utile, per esempio, quando la classe base è definita in un altro modulo:

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

L'esecuzione di una definizione di classe derivata procede nello stesso modo di una classe base. Quando viene costruito l'oggetto classe, viene ricordata la classe base. Questo viene utilizzato per risolvere i riferimenti agli attributi: se un attributo richiesto non è trovato nella classe, la ricerca procede a cercarlo nella classe base. Questa regola viene applicata ricorsivamente se la classe base stessa è derivata da un'altra classe.

Non c'è nulla di speciale nell'istanza di classi derivate: `DerivedClassName()` crea una nuova istanza della classe. I riferimenti ai metodi vengono risolti come segue: viene cercato l'attributo di classe corrispondente, scendendo lungo la catena delle classi base se necessario, e il riferimento al metodo è valido se questo produce un oggetto funzione.

Le classi derivate possono ignorare i metodi delle loro classi base. Poiché i metodi non hanno privilegi speciali quando chiamano altri metodi dello stesso oggetto, un metodo di una classe base che chiama un altro metodo definito nella stessa classe base può finire per chiamare un metodo di una classe derivata che lo sostituisce. (Per programmatori C++: tutti i metodi in Python sono effettivamente `virtual`.)

Un metodo sovrascritto in una classe derivata può in realtà voler estendere piuttosto che semplicemente sostituire il metodo della classe base con lo stesso nome. C'è un modo semplice per chiamare direttamente il metodo della classe base: basta chiamare `BaseClassName.methodname(self, arguments)`. Questo è occasionalmente utile anche per i clienti. (Si noti che questo funziona solo se la classe base è accessibile come `BaseClassName` nello scope globale.)

Python ha due funzioni built-in che funzionano con l'ereditarietà:

- Usa `isinstance()` per controllare il tipo di un'istanza: `isinstance(obj, int)` sarà `True` solo se `obj.__class__` è `int` o una classe derivata da `int`.
- Usa `issubclass()` per controllare l'ereditarietà delle classi: `issubclass(bool, int)` è `True` poiché `bool` è una sottoclasse di `int`. Tuttavia, `issubclass(float, int)` è `False` poiché `float` non è una sottoclasse di `int`.

9.5.1 Ereditarietà Multipla

Python supporta anche una forma di ereditarietà multipla. Una definizione di classe con più classi base è simile:

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

Per la maggior parte degli scopi, nei casi più semplici, si può pensare alla ricerca di attributi ereditati da una classe padre come alla profondità prima, da sinistra a destra, senza cercare due volte nella stessa classe dove c'è una sovrapposizione nella gerarchia. Pertanto, se un attributo non viene trovato in `DerivedClassName`, viene cercato in `Base1`, quindi (ricorsivamente) nelle classi base di `Base1`, e se non è stato trovato lì, viene cercato in `Base2`, e così via.

Infatti, è leggermente più complesso di così; l'ordine di risoluzione del metodo cambia dinamicamente per supportare le chiamate cooperative a `super()`. Questo approccio è noto in altri linguaggi con ereditarietà multipla come `call-next-method` ed è più potente della chiamata `super` trovata nei linguaggi a ereditarietà singola.

L'ordinamento dinamico è necessario perché tutti i casi di ereditarietà multipla mostrano una o più relazioni di diamante (dove almeno una delle classi padre può essere accessibile attraverso percorsi multipli dalla classe più

bassa). Per esempio, tutte le classi ereditano da `object`, quindi qualsiasi caso di ereditarietà multipla fornisce più di un percorso per raggiungere `object`. Per evitare che le classi base vengano accessibili più di una volta, l'algoritmo dinamico linearizza l'ordine di ricerca in modo che preserva l'ordunquezione da sinistra a destra specificata in ogni classe, che chiama ogni genitore solo una volta, e che è monotònico (significando che una classe può essere sottoclasse senza influenzare l'ordine di precedenza dei suoi genitori). Insieme, queste proprietà rendono possibile progettare classi affidabili ed estensibili con ereditarietà multipla. Per maggiori dettagli, vedi `python_2.3_mro`.

9.6 Variabili Private

Non esistono variabili di istanza «private» che non possono essere accessibili se non all'interno di un oggetto in Python. Tuttavia, c'è una convenzione seguita dalla maggior parte del codice Python: un nome prefisso con un trattino basso (es. `_spam`) dovrebbe essere trattato come una parte non pubblica dell'API (sia che sia una funzione, un metodo o un membro dato). Dovrebbe essere considerato un dettaglio di implementazione e soggetto a cambiamento senza preavviso.

Poiché esiste un caso d'uso valido per i membri privati della classe (ovvero per evitare conflitti di nomi con nomi definiti dalle sottoclassi), c'è un supporto limitato per tale meccanismo, chiamato *offuscamento dei nomi*. Qualsiasi identificatore della forma `__spam` (almeno due trattini bassi iniziali, al massimo uno finale) è testualmente sostituito con `_classname__spam`, dove `classname` è il nome attuale della classe con i trattini bassi iniziali rimossi. Questo offuscamento viene eseguito senza riguardo alla posizione sintattica dell'identificatore, purché si trovi all'interno della definizione di una classe.

Vedi anche

The private name mangling specifications for details and special cases.

L'offuscamento dei nomi è utile per permettere alle sottoclassi di sovrascrivere i metodi senza rompere le chiamate ai metodi interni alla 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    # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

L'esempio sopra funzionerebbe anche se `MappingSubclass` dovesse introdurre un identificatore `__update` poiché viene sostituito con `_Mapping__update` nella classe `Mapping` e `_MappingSubclass__update` nella classe `MappingSubclass` rispettivamente.

Si noti che le regole di offuscamento sono progettate soprattutto per evitare incidenti; è ancora possibile accedere o modificare una variabile considerata privata. Questo può essere utile in circostanze speciali, come nel debugger.

Si noti che il codice passato a `exec()` o `eval()` non considera la classe nome dell'invocante per essere la classe attuale; questo è simile all'effetto della dichiarazione `global`, l'effetto del quale è analogamente limitato al codice che

è compilato insieme. La stessa restrizione si applica a `getattr()`, `setattr()` e `delattr()`, così come quando si fa riferimento direttamente a `__dict__`.

9.7 Varie ed Eventuali

A volte è utile avere un tipo di dato simile al «record» del Pascal o al «struct» del C, raggruppando insieme alcuni elementi dati con nome. L'approccio idiomatico è utilizzare `dataclasses` per questo scopo:

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

Un pezzo di codice Python che si aspetta un particolare tipo di dato astratto può spesso essere passato a una classe che emula i metodi di quel tipo di dato. Per esempio, se hai una funzione che formatta alcuni dati da un oggetto file, puoi definire una classe con i metodi `read()` e `readline()` che ottengono i dati da un buffer di stringhe, e passarla come argomento.

Gli oggetti metodo d'istanza hanno anch'essi attributi: `m.__self__` è l'oggetto istanza con il metodo `m()`, e `m.__func__` è l'oggetto funzione corrispondente al metodo.

9.8 Iteratori

Ormai avrai probabilmente notato che la maggior parte degli oggetti contenitori può essere iterata utilizzando un'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 stile di accesso è chiaro, conciso e conveniente. L'uso degli iteratori permea e unifica Python. Dietro le quinte, l'istruzione `for` chiama la funzione `iter()` sull'oggetto contenitore. La funzione restituisce un oggetto iteratore che definisce il metodo `__next__()` che accede agli elementi del contenitore uno alla volta. Quando non ci sono più elementi, `__next__()` solleva un'eccezione `StopIteration` che indica al ciclo `for` di terminare. Puoi chiamare il metodo `__next__()` utilizzando la funzione built-in `next()`; questo esempio mostra come funziona tutto:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
```

(continues on next page)

(continua dalla pagina precedente)

```
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Avendo visto la meccanica dietro il protocollo dell'iteratore, è facile aggiungere il comportamento di iteratore alle tue classi. Definisci un metodo `__iter__()` che restituisce un oggetto con un metodo `__next__()`. Se la classe definisce `__next__()`, allora `__iter__()` può semplicemente restituire `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    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>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.9 Generatori

I *generatori* sono uno strumento semplice e potente per creare iteratori. Sono scritti come funzioni regolari ma utilizzano l'istruzione `yield` ogni volta che vogliono restituire dati. Ogni volta che si chiama `next()` su di esso, il generatore riprende da dove si era interrotto (ricorda tutti i valori dei dati e quale istruzione è stata eseguita per ultima). Un esempio mostra che i generatori possono essere creati con estrema facilità:

```
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
```

(continues on next page)

(continua dalla pagina precedente)

```
f
l
o
g
```

Tutto ciò che può essere fatto con i generatori può essere fatto anche con gli iteratori basati su classi come descritto nella sezione precedente. Ciò che rende i generatori così compatti è che i metodi `__iter__()` e `__next__()` sono creati automaticamente.

Un'altra caratteristica chiave è che le variabili locali e lo stato di esecuzione sono salvati automaticamente tra le chiamate. Questo rende la funzione più facile da scrivere e molto più chiara rispetto a un approccio che utilizza variabili di istanza come `self.index` e `self.data`.

Oltre alla creazione automatica dei metodi e al salvataggio dello stato del programma, quando i generatori terminano, sollevano automaticamente `StopIteration`. In combinazione, queste caratteristiche rendono facile creare iteratori con lo stesso sforzo di scrivere una funzione regolare.

9.10 Espressioni di Generatore

Alcuni generatori semplici possono essere codificati in modo conciso come espressioni utilizzando una sintassi simile alle comprensioni di lista ma con parentesi tonde anziché quadre. Queste espressioni sono progettate per situazioni in cui il generatore viene utilizzato immediatamente da una funzione racchiudente. Le espressioni di generatore sono più compatte ma meno versatili rispetto alle definizioni complete di generatore e tendono a essere più efficienti in termini di memoria rispetto alle comprensioni di lista equivalenti.

Esempi:

```
>>> sum(i*i for i in range(10))           # sum of squares
285

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

Breve Panoramica della Libreria Standard

10.1 Interfaccia del Sistema Operativo

Il modulo `os` fornisce dozzine di funzioni per interagire con il sistema operativo:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Assicurati di usare lo stile `import os` invece di `from os import *`. Questo eviterà che `os.open()` sovrasti la funzione built-in `open()` che opera in modo molto diverso.

Le funzioni built-in `dir()` e `help()` sono utili come aiuti interattivi per lavorare con moduli grandi come `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Per attività quotidiane di gestione di file e directory, il modulo `shutil` fornisce un'interfaccia di livello superiore che è più facile da usare:

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

10.2 Metacaratteri nei Nomfile

Il modulo `glob` fornisce una funzione per creare liste di file a partire da ricerche con metacaratteri nei nomi delle directory:

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

10.3 Argomenti dalla Linea di Comando

Gli script di utilità comuni spesso necessitano di elaborare gli argomenti dalla linea di comando. Questi argomenti sono memorizzati nell'attributo *argv* del modulo *sys* come una lista. Per esempio, prendiamo il seguente file `demo.py`:

```
# File demo.py
import sys
print(sys.argv)
```

Qui c'è l'output eseguendo `python demo.py one two three` dalla linea di comando:

```
['demo.py', 'one', 'two', 'three']
```

Il modulo *argparse* fornisce un meccanismo più sofisticato per elaborare gli argomenti dalla linea di comando. Il seguente script 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 eseguito dalla linea di comando con `python top.py --lines=5 alpha.txt beta.txt`, lo script assegna `args.lines` a 5 e `args.filenames` a `['alpha.txt', 'beta.txt']`.

10.4 Reindirizzamento dell'Uscita degli Errori e Terminazione del Programma

Il modulo *sys* ha anche attributi per *stdin*, *stdout* e *stderr*. Quest'ultimo è utile per emettere avvisi e messaggi di errore per renderli visibili anche quando *stdout* è stato reindirizzato:

```
>>> 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 uno script è usare `sys.exit()`.

10.5 Pattern Matching di Stringhe

Il modulo *re* fornisce strumenti per le espressioni regolari per l'elaborazione avanzata delle stringhe. Per il matching e la manipolazione complessi, le espressioni regolari offrono soluzioni concise e ottimizzate:

```
>>> import re
>>> re.findall(r'\b[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'
```

Quando sono necessarie solo semplici capacità, i metodi delle stringhe sono preferiti perché sono più facili da leggere e fare debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Matematica

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Il modulo `random` fornisce strumenti per effettuare selezioni casuali:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()                  # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)              # random integer chosen from range(6)
4
```

Il modulo `statistics` calcola proprietà statistiche di base (la media, la mediana, la varianza, ecc.) di 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 <<https://scipy.org>> ha molti altri moduli per i calcoli numerici.

10.7 Accesso a Internet

Ci sono molti moduli per accedere a internet e processare i protocolli internet. Due dei più semplici sono `urllib.request` per recuperare dati da URL e `smtplib` per inviare mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()          # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())      # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
```

(continues on next page)

(continua dalla pagina precedente)

```
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Nota che il secondo esempio ha bisogno di un mailserver in esecuzione su localhost.)

10.8 Date e Orari

Il modulo `datetime` fornisce classi per manipolare date e orari in modi sia semplici che complessi. Mentre l'aritmetica su date e orari è supportata, l'obiettivo dell'implementazione è l'estrazione efficiente dei membri per la formattazione e la manipolazione dell'output. Il modulo supporta anche oggetti che sono consapevoli del fuso orario.

```
>>> # 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.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Compressione dei Dati

I formati comuni di archiviazione e compressione dei dati sono direttamente supportati dai moduli inclusi: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` e `tarfile`.

```
>>> 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 delle Prestazioni

Alcuni utenti di Python sviluppano un profondo interesse nel conoscere le prestazioni relative di diversi approcci allo stesso problema. Python fornisce uno strumento di misurazione che risponde a queste domande immediatamente.

Per esempio, può essere allettante usare la funzione di impacchettamento e spaccettamento delle tuple invece dell'approccio tradizionale per scambiare argomenti. Il modulo `timeit` dimostra rapidamente un modesto vantaggio in termini di prestazioni:

```
>>> 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
```

In contrasto con la fine granularità di `timeit`, i moduli `profile` e `pstats` forniscono strumenti per identificare le sezioni critiche in termini di tempo in blocchi di codice più grandi.

10.11 Controllo della Qualità

Un approccio per sviluppare software di alta qualità è scrivere test per ogni funzione mentre viene sviluppata e eseguire quei test frequentemente durante il processo di sviluppo.

Il modulo `doctest` fornisce uno strumento per scansionare un modulo e validare i test incorporati nei docstring di un programma. La costruzione dei test è semplice come tagliare e incollare una tipica chiamata con i suoi risultati nel docstring. Questo migliora la documentazione fornendo all'utente un esempio e permette al modulo `doctest` di assicurarsi che il codice rimanga conforme alla documentazione:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()    # automatically validate the embedded tests
```

Il modulo `unittest` non è così semplice come il modulo `doctest`, ma permette di mantenere un insieme più completo di test in un file separato:

```
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()    # Calling from the command line invokes all tests
```

10.12 Batterie Incluse

Python ha una filosofia di «batterie incluse». Questo è reso evidente dalle capacità sofisticate e robuste dei suoi pacchetti più grandi. Per esempio:

- I moduli `xmlrpc.client` e `xmlrpc.server` rendono l'implementazione delle chiamate di procedure remote un compito quasi banale. Nonostante i nomi dei moduli, non è necessaria una conoscenza o una gestione diretta dell'XML.
- Il pacchetto `email` è una libreria per la gestione dei messaggi email, inclusi i messaggi basati su MIME e altri documenti basati su [RFC 2822](#). A differenza di `smtplib` e `poplib` che effettivamente inviano e ricevono messaggi, il pacchetto `email` ha un set completo di strumenti per costruire o decodificare strutture di messaggi complessi (inclusi allegati) e per implementare protocolli di codifica internet e di intestazioni.
- Il pacchetto `json` fornisce un supporto robusto per il parsing di questo popolare formato di interscambio dati. Il modulo `csv` supporta la lettura e la scrittura diretta di file nel formato Comma-Separated Value, comunemente supportato da database e fogli di calcolo. L'elaborazione XML è supportata dai pacchetti `xml.etree.ElementTree`, `xml.dom` e `xml.sax`. Insieme, questi moduli e pacchetti semplificano notevolmente l'interscambio di dati tra applicazioni Python e altri strumenti.
- Il modulo `sqlite3` è un wrapper per la libreria del database SQLite, fornendo un database persistente che può essere aggiornato e accessibile utilizzando una sintassi SQL leggermente non standard.
- L'internazionalizzazione è supportata da numerosi moduli tra cui `gettext`, `locale` e il pacchetto `codecs`.

Giro Breve della Libreria Standard — Parte II

Questo secondo giro copre moduli più avanzati che supportano esigenze di programmazione professionali. Questi moduli raramente si trovano in piccoli script.

11.1 Formattazione dell'Output

Il modulo `reprlib` fornisce una versione di `repr()` personalizzata per visualizzazioni abbreviate di contenitori grandi o profondamente annidati:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

Il modulo `pprint` offre un controllo più sofisticato sulla stampa di oggetti sia integrati che definiti dall'utente in modo leggibile dall'interprete. Quando il risultato è più lungo di una riga, il «pretty printer» aggiunge interruzioni di riga e indentazione per rivelare più chiaramente 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']]]
```

Il modulo `textwrap` formatta paragrafi di testo per adattarsi a una larghezza di schermo specificata:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
```

(continues on next page)

(continua dalla pagina precedente)

```
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

Il modulo `locale` accede a un database di formati di dati specifici della cultura. L'attributo di raggruppamento della funzione di formattazione del locale fornisce un modo diretto di formattare i numeri con separatori di gruppo:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()           # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%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 Templating

Il modulo `string` include una versatile classe `Template` con una sintassi semplificata adatta per l'editing da parte degli utenti finali. Questo permette agli utenti di personalizzare le loro applicazioni senza dover alterare l'applicazione.

Il formato utilizza nomi di segnaposto formati da `$` con identificatori Python validi (caratteri alfanumerici e trattini bassi). Circondando il segnaposto con parentesi graffe, è possibile farlo seguire da altre lettere alfanumeriche senza spazi intermedi. Scrivere `$$` crea un singolo `$` escapato:

```
>>> 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()` solleva un'eccezione `KeyError` quando un segnaposto non è fornito in un dizionario o come parametro keyword. Per applicazioni stile mail-merge, i dati forniti dall'utente possono essere incompleti e il metodo `safe_substitute()` potrebbe essere più appropriato — lascerà i segnaposto invariati se i dati sono mancanti:

```
>>> 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.'
```

Le sottoclassi di `Template` possono specificare un delimitatore personalizzato. Ad esempio, un'utilità per il rinominamento dei batch di un browser di foto può scegliere di usare segni percentuali per i segnaposto come la data corrente, il numero di sequenza delle immagini o il formato del file:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
... 
```

(continues on next page)

(continua dalla pagina precedente)

```
>>> 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)
>>> 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'altra applicazione per il templating è separare la logica del programma dai dettagli di più formati di output. Questo rende possibile sostituire template personalizzati per file XML, rapporti in testo semplice e rapporti HTML web.

11.3 Lavorare con Layout Binari dei Dati

Il modulo `struct` fornisce le funzioni `pack()` e `unpack()` per lavorare con formati di record binari a lunghezza variabile. Il seguente esempio mostra come scorrere le informazioni dell'intestazione in un file ZIP senza usare il modulo `zipfile`. I codici `pack` "H" e "I" rappresentano numeri senza segno a due e quattro byte rispettivamente. Il "<" indica che hanno dimensioni standard e sono in byte order little-endian:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):
    # show the first 3 file 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    # skip to the next header
```

11.4 Multi-threading

Il threading è una tecnica per disaccoppiare i compiti che non sono sequenzialmente dipendenti. I thread possono essere utilizzati per migliorare la reattività delle applicazioni che accettano l'input dell'utente mentre altri compiti vengono eseguiti in background. Un caso d'uso correlato è eseguire I/O in parallelo con calcoli in un altro thread.

Il seguente codice mostra come il modulo ad alto livello `threading` può eseguire compiti in background mentre il programma principale continua a funzionare:

```
import threading, zipfile

class AsyncZip(threading.Thread):
```

(continues on next page)

(continua dalla pagina precedente)

```

def __init__(self, infile, outfile):
    threading.Thread.__init__(self)
    self.infile = infile
    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()    # Wait for the background task to finish
print('Main program waited until background was done.')

```

La principale sfida delle applicazioni multi-threaded è coordinare i thread che condividono dati o altre risorse. A tal fine, il modulo `threading` fornisce numerosi primitivi di sincronizzazione tra cui lock, eventi, variabili di condizione e semafori.

Sebbene questi strumenti siano potenti, piccoli errori di progettazione possono provocare problemi difficili da riprodurre. Quindi, l'approccio preferito per il coordinamento dei compiti è concentrare tutto l'accesso a una risorsa in un singolo thread e poi usare il modulo `queue` per alimentare quel thread con richieste provenienti da altri thread. Le applicazioni che usano oggetti `Queue` per la comunicazione e il coordinamento tra thread sono più facili da progettare, più leggibili e più affidabili.

11.5 Logging

Il modulo `logging` offre un sistema di logging completo e flessibile. Nel suo uso più semplice, i messaggi di log vengono inviati 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 il seguente output:

```

WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down

```

Per impostazione predefinita, i messaggi informativi e di debug sono soppressi e l'output è inviato all'errore standard. Altre opzioni di output includono l'instradamento dei messaggi tramite email, datagrammi, socket o a un server HTTP. Filtri nuovi possono selezionare l'instradamento diverso basato sulla priorità del messaggio: `DEBUG`, `INFO`, `WARNING`, `ERROR`, e `CRITICAL`.

Il sistema di logging può essere configurato direttamente da Python o può essere caricato da un file di configurazione modificabile dall'utente per un logging personalizzato senza alterare l'applicazione.

11.6 Riferimenti Deboli

Python gestisce automaticamente la memoria (conteggio dei riferimenti per la maggior parte degli oggetti e *garbage collection* per eliminare i cicli). La memoria viene liberata poco dopo che l'ultimo riferimento ad essa è stato eliminato.

Questo approccio funziona bene per la maggior parte delle applicazioni ma occasionalmente è necessario tracciare gli oggetti solo finché vengono utilizzati da qualcos'altro. Sfortunatamente, solo il tracciamento crea un riferimento che li rende permanenti. Il modulo `weakref` fornisce strumenti per tracciare gli oggetti senza creare un riferimento. Quando l'oggetto non è più necessario, viene automaticamente rimosso da una tabella `weakref` e viene attivato un callback per gli oggetti `weakref`. Applicazioni tipiche includono il caching di 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)                                # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a                          # does not create a reference
>>> d['primary']                              # fetch the object if it is still alive
10
>>> del a                                    # remove the one reference
>>> gc.collect()                             # run garbage collection right away
0
>>> d['primary']                             # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                             # entry was automatically removed
  File "C:/python313/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Strumenti per Lavorare con Liste

Molte necessità di strutture dati possono essere soddisfatte con il tipo di lista integrato. Tuttavia, a volte c'è bisogno di implementazioni alternative con diversi compromessi di prestazioni.

The `array` module provides an `array` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The `collections` module provides a `deque` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
```

(continues on next page)

(continua dalla pagina precedente)

```
>>> 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 alle implementazioni alternative di liste, la libreria offre anche altri strumenti come il modulo `bisect` con funzioni per manipolare 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` fornisce funzioni per implementare heap basati su liste regolari. L'ingresso a valore più basso viene sempre mantenuto in posizione zero. Questo è utile per applicazioni che accedono ripetutamente all'elemento più piccolo ma non vogliono eseguire un ordinamento completo della lista:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # rearrange the list into heap order
>>> heappush(data, -5) # add a new entry
>>> [heappop(data) for i in range(3)] # fetch the three smallest entries
[-5, 0, 1]
```

11.8 Decimal Floating-Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating-point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for

- applicazioni finanziarie e altri usi che richiedono una rappresentazione decimale esatta,
- controllo sulla precisione,
- controllo sull'arrotondamento per soddisfare requisiti legali o normativi,
- tracciamento delle cifre decimali significative, oppure
- applicazioni in cui l'utente si aspetta che i risultati corrispondano ai calcoli fatti a mano.

Ad esempio, calcolare una tassa del 5% su una tariffa telefonica di 70 centesimi dà risultati diversi in decimale a virgola mobile e binario a virgola mobile. La differenza diventa significativa se i risultati vengono arrotondati 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
```

Il risultato `Decimal` mantiene uno zero finale, inferendo automaticamente una significatività a quattro posti dai fattori moltiplicativi con significatività a due posti. Decimale riproduce la matematica fatta a mano ed evita problemi che possono sorgere quando la virgola mobile binaria non può rappresentare esattamente quantità decimali.

La rappresentazione esatta permette alla classe `Decimal` di eseguire calcoli modulo e test di uguaglianza che sono inadatti per la virgola mobile binaria:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

Il modulo `decimal` fornisce aritmetica con precisione quanto richiesta:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857')
```


12.1 Introduzione

Le applicazioni Python spesso utilizzano pacchetti e moduli che non fanno parte della libreria standard. Le applicazioni a volte necessitano una versione specifica di una libreria, perché l'applicazione potrebbe richiedere che un bug particolare sia stato risolto o l'applicazione potrebbe essere stata scritta utilizzando una versione obsoleta dell'interfaccia della libreria.

Questo significa che potrebbe non essere possibile per una singola installazione di Python soddisfare i requisiti di ogni applicazione. Se l'applicazione A necessita della versione 1.0 di un particolare modulo ma l'applicazione B necessita della versione 2.0, allora i requisiti sono in conflitto e l'installazione della versione 1.0 o 2.0 lascerà una delle applicazioni impossibilitata a funzionare.

La soluzione a questo problema è creare un *virtual environment*, un albero di directory auto-contenuto che contiene un'installazione di Python per una particolare versione di Python, più un numero di pacchetti aggiuntivi.

Differenti applicazioni possono quindi utilizzare differenti ambienti virtuali. Per risolvere l'esempio precedente di requisiti in conflitto, l'applicazione A può avere il suo proprio ambiente virtuale con la versione 1.0 installata mentre l'applicazione B ha un altro ambiente virtuale con la versione 2.0. Se l'applicazione B richiede che una libreria venga aggiornata alla versione 3.0, questo non influenzerà l'ambiente dell'applicazione A.

12.2 Creazione di Ambienti Virtuali

Il modulo utilizzato per creare e gestire ambienti virtuali si chiama `venv`. `venv` installerà la versione di Python dalla quale il comando è stato eseguito (come riportato dall'opzione `--version`). Per esempio, eseguendo il comando con `python3.12` verrà installata la versione 3.12.

Per creare un ambiente virtuale, decidi una directory dove vuoi posizionarlo ed esegui il modulo `venv` come script con il percorso della directory:

```
python -m venv tutorial-env
```

Questo creerà la directory `tutorial-env` se non esiste, e creerà anche directory all'interno contenenti una copia dell'interprete Python e vari file di supporto.

Una posizione comune per una directory di ambiente virtuale è `.venv`. Questo nome mantiene la directory generalmente nascosta nella tua shell e quindi fuori mano, dando anche un nome che spiega il motivo dell'esistenza della directory. Previene anche conflitti con file di definizione di variabili d'ambiente `.env` che alcuni strumenti supportano.

Una volta creato un ambiente virtuale, puoi attivarlo.

Su Windows, esegui:

```
tutorial-env\Scripts\activate
```

Su Unix o MacOS, esegui:

```
source tutorial-env/bin/activate
```

(Questo script è scritto per la shell `bash`. Se utilizzi le shell `csh` o `fish`, ci sono script alternativi `activate.csh` e `activate.fish` che dovresti utilizzare.)

Attivare l'ambiente virtuale cambierà il prompt della tua shell per mostrare quale ambiente virtuale stai utilizzando, e modificherà l'ambiente in modo che eseguire `python` ti fornirà quella versione e installazione particolare di Python. 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']
>>>
```

Per disattivare un ambiente virtuale, scrivi:

```
deactivate
```

nel terminale.

12.3 Gestione dei Pacchetti con pip

Puoi installare, aggiornare e rimuovere pacchetti utilizzando un programma chiamato **pip**. Per impostazione predefinita `pip` installerà pacchetti dal [Python Package Index](#). Puoi sfogliare il Python Package Index andando sul sito web con il tuo browser.

`pip` ha un certo numero di sottocomandi: «install», «uninstall», «freeze», etc. (Consulta la guida [installing-index](#) per la documentazione completa su `pip`.)

Puoi installare l'ultima versione di un pacchetto specificando il nome del pacchetto:

```
(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
```

Puoi anche installare una versione specifica di un pacchetto fornendo il nome del pacchetto seguito da `==` e dal 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
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Se esegui nuovamente questo comando, `pip` noterà che la versione richiesta è già installata e non farà nulla. Puoi fornire un numero di versione diverso per ottenere quella versione, oppure puoi eseguire `python -m 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
```

`python -m pip uninstall` seguito da uno o più nomi di pacchetti rimuoverà i pacchetti dall'ambiente virtuale.

`python -m pip show` mostrerà informazioni su un particolare pacchetto:

```
(tutorial-env) $ python -m 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:
```

`python -m pip list` mostrerà tutti i pacchetti installati nell'ambiente virtuale:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` produrrà una lista simile dei pacchetti installati, ma l'output utilizza il formato che `python -m pip install` si aspetta. Una convenzione comune è mettere questa lista in un file `requirements.txt`:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

Il `requirements.txt` può poi essere committato nel controllo di versione e distribuito come parte di un'applicazione. Gli utenti possono poi 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))
...
Installing collected packages: novas, numpy, requests
```

(continues on next page)

(continua dalla pagina precedente)

```
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. Consulta la guida [installing-index](#) per la documentazione completa su `pip`. Quando hai scritto un pacchetto e vuoi renderlo disponibile su Python Package Index, consulta la [guida dell'utente al packaging di Python](#).

E adesso?

Probabilmente la lettura di questo tutorial ha rafforzato il tuo interesse nell'usare Python — dovresti essere desideroso di applicare Python alla risoluzione dei tuoi problemi reali. Dove dovresti andare per imparare di più?

Questo tutorial fa parte della documentazione di Python. Altri documenti disponibili sono:

- `library-index`:

Dovresti sfogliare questo manuale, che fornisce materiale di riferimento completo (anche se conciso) sui tipi, le funzioni e i moduli nella libreria standard. La distribuzione standard di Python include *molto* codice aggiuntivo. Ci sono moduli per leggere caselle di posta Unix, recuperare documenti via HTTP, generare numeri casuali, analizzare opzioni da riga di comando, comprimere dati e molte altre attività. Sfogliare il Reference della libreria ti darà un'idea di ciò che è disponibile.

- `installing-index` spiega come installare moduli aggiuntivi scritti da altri utenti Python.
- `reference-index`: Una spiegazione dettagliata della sintassi e della semantica di Python. È una lettura impegnativa, ma è utile come guida completa al linguaggio stesso.

Ulteriori risorse Python:

- <https://www.python.org>: Il principale sito web di Python. Contiene codice, documentazione e collegamenti a pagine relative a Python in tutto il web.
- <https://docs.python.org>: Accesso rapido alla documentazione di Python.
- <https://pypi.org>: L'Indice dei pacchetti Python, precedentemente anche soprannominato il Cheese Shop¹, è un indice di moduli Python creati dagli utenti che sono disponibili per il download. Una volta che inizi a rilasciare codice, puoi registrarlo qui in modo che altri possano trovarlo.
- <https://code.activestate.com/recipes/langs/python/>: Il Python Cookbook è una vasta raccolta di esempi di codice, moduli più grandi e script utili. Contributi particolarmente notevoli sono raccolti in un libro intitolato Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)
- <https://pyvideo.org> raccoglie collegamenti a video relativi a Python da conferenze e riunioni di gruppi di utenti.
- <https://scipy.org>: Il progetto Scientific Python include moduli per calcoli e manipolazioni di array veloci e una serie di pacchetti per attività come l'algebra lineare, le trasformate di Fourier, i risolutori non lineari, le distribuzioni di numeri casuali, l'analisi statistica e simili.

¹ «Cheese Shop» è uno sketch dei Monty Python: un cliente entra in un negozio di formaggi, ma qualunque formaggio chieda, il commesso dice che è finito.

Per domande e segnalazioni di problemi relativi a Python, puoi inviare un messaggio al newsgroup *comp.lang.python*, o inviarli alla mailing list all'indirizzo python-list@python.org. Il newsgroup e la mailing list sono dei **gateway**, quindi i messaggi inviati a uno saranno automaticamente inoltrati all'altro. Ci sono centinaia di messaggi al giorno, che pongono (e rispondono) domande, suggeriscono nuove funzionalità e annunciano nuovi moduli. Gli archivi della mailing list sono disponibili su <https://mail.python.org/pipermail/>.

Prima di inviare un messaggio, assicurati di controllare l'elenco delle Domande più frequenti (chiamato anche FAQ). La FAQ risponde a molte delle domande che vengono poste di nuovo e di nuovo, e potrebbe già contenere la soluzione per il tuo problema.

Editing dell'input interattivo e sostituzione della history

Alcune versioni dell'interprete Python supportano la modifica della riga di input corrente e la sostituzione della history, simili alle funzionalità presenti nella shell Korn e nella shell GNU Bash. Questo è implementato utilizzando la libreria [GNU Readline](#), che supporta vari stili di modifica. Questa libreria ha la sua documentazione che non duplicheremo qui.

14.1 Completamento con il tasto Tab e modifica della history

Il completamento dei nomi delle variabili e dei moduli è automaticamente abilitato all'avvio dell'interprete in modo che il tasto `Tab` invochi la funzione di completamento; essa esamina i nomi delle istruzioni Python, le variabili locali correnti e i nomi dei moduli disponibili. Per le espressioni puntate come `string.a`, valuterà l'espressione fino all'ultimo `'.'` e quindi suggerirà completamenti dagli attributi dell'oggetto risultante. Nota che questo potrebbe eseguire codice definito dall'applicazione se un oggetto con un metodo `__getattr__()` fa parte dell'espressione. La configurazione predefinita salva anche la tua history in un file chiamato `.python_history` nella tua directory utente. La history sarà disponibile nuovamente durante la prossima sessione dell'interprete interattivo.

14.2 Alternative all'interprete interattivo

Questa funzionalità è un enorme passo avanti rispetto alle versioni precedenti dell'interprete; tuttavia, alcuni desideri sono rimasti: sarebbe bello se l'indentazione corretta venisse suggerita sulle linee di continuazione (il parser sa se è richiesto un token di indentazione successivo). Il meccanismo di completamento potrebbe utilizzare la tabella dei simboli dell'interprete. Un comando per controllare (o addirittura suggerire) parentesi, virgolette, ecc. corrispondenti, sarebbe anche molto utile.

Un'alternativa all'interprete interattivo migliorato che esiste da molto tempo è [IPython](#), che offre il completamento con il tasto `Tab`, l'esplorazione degli oggetti e una gestione avanzata della history. Può anche essere personalizzato a fondo e incorporato in altre applicazioni. Un altro ambiente interattivo migliorato simile è [bpython](#).

Floating-Point Arithmetic: Issues and Limitations

I numeri in virgola mobile sono rappresentati nell'hardware del computer come frazioni in base 2 (binari). Ad esempio, la frazione **decimale** 0.625 ha valore $6/10 + 2/100 + 5/1000$, e allo stesso modo la frazione **binaria** 0.101 ha valore $1/2 + 0/4 + 1/8$. Queste due frazioni hanno valori identici, l'unica vera differenza è che la prima è scritta in notazione frazionaria in base 10, e la seconda in base 2.

Sfortunatamente, la maggior parte delle frazioni decimali non può essere rappresentata esattamente come frazioni binarie. Una conseguenza è che, in generale, i numeri in virgola mobile decimali che inserisci sono solo approssimati dai numeri in virgola mobile binari effettivamente memorizzati nella macchina.

Il problema è più facile da comprendere inizialmente in base 10. Considera la frazione $1/3$. Puoi approssimarla come una frazione in base 10:

```
0.3
```

o, meglio,

```
0.33
```

o, meglio,

```
0.333
```

e così via. Non importa quanti cifre sei disposto a scrivere, il risultato non sarà mai esattamente $1/3$, ma sarà una approssimazione sempre migliore di $1/3$.

Allo stesso modo, non importa quante cifre in base 2 sei disposto a usare, il valore decimale 0.1 non può essere rappresentato esattamente come una frazione in base 2. In base 2, $1/10$ è la frazione che si ripete all'infinito:

```
0.0001100110011001100110011001100110011001100110011001100110011...
```

Fermati a un qualsiasi numero finito di bit, e otterrai un'approssimazione. Nella maggior parte delle macchine odierne, i float sono approssimati usando una frazione binaria con il numeratore che utilizza i primi 53 bit a partire dal bit più significativo e con il denominatore come potenza di due. Nel caso di $1/10$, la frazione binaria è $3602879701896397 / 2^{55}$ che è vicino ma non esattamente uguale al valore vero di $1/10$.

Molti utenti non sono a conoscenza dell'approssimazione a causa del modo in cui i valori sono visualizzati. Python stampa solo un'approssimazione decimale al valore decimale vero dell'approssimazione binaria memorizzata nella macchina. Nella maggior parte delle macchine, se Python dovesse stampare il valore decimale vero dell'approssimazione binaria memorizzata per 0.1 , dovrebbe visualizzare:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Questo è un numero di cifre maggiore di quelle che la maggior parte delle persone trova utile, quindi Python mantiene il numero di cifre gestibile visualizzando invece un valore arrotondato:

```
>>> 1 / 10
0.1
```

Ricorda solo, anche se il risultato stampato sembra il valore esatto di $1/10$, il valore effettivamente memorizzato è la frazione binaria rappresentabile più vicina.

Interessante è che ci sono molti numeri decimali diversi che condividono la stessa frazione binaria approssimata più vicina. Ad esempio, i numeri `0.1` e `0.100000000000000001` e `0.1000000000000000055511151231257827021181583404541015625` sono tutti approssimati da $3602879701896397 / 2^{55}$. Poiché tutti questi valori decimali condividono la stessa approssimazione, qualsiasi di essi potrebbe essere visualizzato pur mantenendo l'invariante `eval(repr(x)) == x`.

Storicamente, il prompt di Python e la funzione integrata `repr()` avrebbero scelto quella con 17 cifre significative, `0.100000000000000001`. A partire da Python 3.1, Python (sulla maggior parte dei sistemi) è ora in grado di scegliere il più corto di questi e visualizzare semplicemente `0.1`.

Note that this is in the very nature of binary floating point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

Per un output più gradevole, potresti voler usare la formattazione delle stringhe per produrre un numero limitato di cifre significative:

```
>>> format(math.pi, '.12g') # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

È importante rendersi conto che questo è, in un certo senso, un'illusione: stai semplicemente arrotondando la *visualizzazione* del vero valore della macchina.

Un'illusione può generare un'altra. Ad esempio, poiché `0.1` non è esattamente $1/10$, sommare tre valori di `0.1` potrebbe non dare esattamente `0.3`, nemmeno:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Inoltre, poiché lo `0.1` non può avvicinarsi al valore esatto di $1/10$ e lo `0.3` non può avvicinarsi al valore esatto di $3/10$, il pre-arrotondamento con la funzione `round()` non può aiutare:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

Anche se i numeri non possono essere resi più vicini ai loro valori esatti previsti, la funzione `math.isclose()` può essere utile per confrontare valori inesatti:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

In alternativa, la funzione `round()` può essere utilizzata per confrontare approssimazioni grossolane:

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```

Binary floating-point arithmetic holds many surprises like this. The problem with «0.1» is explained in precise detail below, in the «Representation Error» section. See [Examples of Floating Point Problems](#) for a pleasant summary of how binary floating point works and the kinds of problems commonly encountered in practice. Also see [The Perils of Floating Point](#) for a more complete account of other common surprises.

As that says near the end, «there are no easy answers.» Still, don't be unduly wary of floating point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2^{53} per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic and that every float operation can suffer a new rounding error.

Sebbene esistano casi patologici, per la maggior parte dell'uso occasionale dell'aritmetica in virgola mobile otterrai il risultato che ti aspetti alla fine se semplicemente arrotondi la visualizzazione dei tuoi risultati finali al numero di cifre decimali che ti aspetti. `str()` di solito è sufficiente, e per un controllo più fine vedi gli specificatori di formato del metodo `str.format()` in [formatstrings](#).

Per utilizzi che richiedono una rappresentazione decimale esatta, prova ad usare il modulo `decimal` che implementa l'aritmetica decimale adatta per applicazioni contabili e applicazioni ad alta precisione.

Un'altra forma di aritmetica esatta è supportata dal modulo `fractions` che implementa l'aritmetica basata su numeri razionali (così i numeri come $1/3$ possono essere rappresentati esattamente).

Se sei un utente assiduo delle operazioni in virgola mobile, dovresti dare un'occhiata al pacchetto NumPy e a molti altri pacchetti per operazioni matematiche e statistiche forniti dal progetto SciPy. Vedi <https://scipy.org>.

Python fornisce strumenti che possono aiutare in quelle rare occasioni in cui davvero *vuoi* conoscere il valore esatto di un float. Il metodo `float.as_integer_ratio()` esprime il valore di un float come una frazione:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Poiché il rapporto è esatto, può essere utilizzato per ricreare senza perdita il valore originale:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

Il metodo `float.hex()` esprime un float in esadecimale (base 16), dando ancora una volta il valore esatto memorizzato dal tuo computer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Questa rappresentazione esadecimale precisa può essere utilizzata per ricostruire il valore del float esattamente:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Poiché la rappresentazione è esatta, è utile per trasferire valori in modo affidabile tra diverse versioni di Python (indipendenza dalla piattaforma) e scambiare dati con altri linguaggi che supportano lo stesso formato (come Java e C99).

Un altro strumento utile è la funzione `sum()` che aiuta a mitigare la perdita di precisione durante la somma. Usa la precisione estesa per i passaggi di arrotondamento intermedi man mano che i valori vengono aggiunti a un totale incrementale. Questo può fare la differenza in termini di precisione complessiva in modo che gli errori non si accumulino al punto da influenzare il totale finale:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

(continues on next page)

(continua dalla pagina precedente)

```
>>> sum([0.1] * 10) == 1.0
True
```

The `math.fsum()` goes further and tracks all of the «lost digits» as values are added onto a running total so that the result has only a single rounding. This is slower than `sum()` but will be more accurate in uncommon cases where large magnitude inputs mostly cancel each other out leaving a final sum near zero:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...        -143401161400469.7, 266262841.31058735, -0.003244936839808227]
>>> float(sum(map(Fraction, arr))) # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr)                 # Single rounding
8.042173697819788e-13
>>> sum(arr)                       # Multiple roundings in extended precision
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
...     total += x                 # Multiple roundings in standard precision
...
>>> total                          # Straight addition has no correct digits!
-0.0051575902860057365
```

15.1 Errore di Rappresentazione

Questa sezione spiega l'esempio «0.1» in dettaglio, e mostra come puoi eseguire un'analisi esatta di casi come questo da solo. Si assume una familiarità di base con la rappresentazione in virgola mobile binaria.

Errore di rappresentazione si riferisce al fatto che alcune (anzi, la maggior parte) delle frazioni decimali non possono essere rappresentate esattamente come frazioni binarie (base 2). Questa è la principale ragione per cui Python (o Perl, C, C++, Java, Fortran e molti altri) spesso non visualizzeranno il numero decimale esatto che ti aspetti.

Perché questo? $1/10$ non è esattamente rappresentabile come frazione binaria. Dal 2000 in poi, quasi tutte le macchine utilizzano l'aritmetica in virgola mobile binaria IEEE 754, e quasi tutte le piattaforme mappano i float di Python ai valori «doppia precisione» IEEE 754 binary64. I valori IEEE 754 binary64 contengono 53 bit di precisione, quindi in input il computer cerca di convertire 0.1 nella frazione più vicina della forma $J/2^N$ dove J è un intero contenente esattamente 53 bit. Riscrivendo:

```
1 / 10 ~ J / (2**N)
```

come:

```
J ~ 2**N / 10
```

e ricordando che J ha esattamente 53 bit (è $\geq 2^{52}$ ma $< 2^{53}$), il miglior valore per N è 56:

```
>>> 2**52 <= 2**56 // 10 < 2**53
True
```

Cioè, 56 è l'unico valore per N che lascia J con esattamente 53 bit. Il miglior valore possibile per J è quindi quel quoziente arrotondato:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Poiché il resto è più della metà di 10, la migliore approssimazione si ottiene arrotondando verso l'alto:

```
>>> q+1
7205759403792794
```

Pertanto la migliore approssimazione possibile di $1/10$ in doppia precisione IEEE 754 è:

```
7205759403792794 / 2 ** 56
```

Dividendo sia il numeratore che il denominatore per due si riduce la frazione a:

```
3602879701896397 / 2 ** 55
```

Nota che poiché abbiamo arrotondato verso l'alto, questo è in realtà un po' più grande di $1/10$; se non avessimo arrotondato verso l'alto, il quoziente sarebbe stato un po' più piccolo di $1/10$. Ma in nessun caso può essere *esattamente* $1/10$!

Quindi il computer non «vede» mai $1/10$: quello che vede è la frazione esatta data sopra, la migliore approssimazione doppia precisione IEEE 754 che può ottenere:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Se moltiplichiamo quella frazione per 10^{55} , possiamo vedere il valore fino a 55 cifre decimali:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
10000000000000000055511151231257827021181583404541015625
```

il che significa che il numero esatto memorizzato nel computer è pari al valore decimale $0.10000000000000000055511151231257827021181583404541015625$. Invece di visualizzare il valore decimale completo, molti linguaggi (compresi le vecchie versioni di Python), arrotondano il risultato a 17 cifre significative:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

I moduli `fractions` e `decimal` rendono questi calcoli facili:

```
>>> 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('0.10000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17f')
'0.10000000000000001'
```


16.1 Modalità Interattiva

Ci sono due varianti del *REPL* interattivo. L'interprete di base classico è supportato su tutte le piattaforme con capacità minime di controllo di riga.

On Windows, or Unix-like systems with `curses` support, a new interactive shell is used by default. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see `using-on-controlling-color` for details. Function keys provide some additional functionality. `F1` enters the interactive help browser `pydoc`. `F2` allows for browsing command-line history with neither output nor the `>` and `...` prompts. `F3` enters «paste mode», which makes pasting larger blocks of code easier. Press `F3` to return to the regular prompt.

Quando si utilizza la nuova shell interattiva, si può uscire dalla shell digitando `exit` o `quit`. Non è necessario aggiungere le parentesi di chiamata dopo questi comandi.

Se non si vuole usare la nuova shell interattiva, questa può essere disabilitata tramite la variabile di ambiente `PYTHON_BASIC_REPL`.

16.1.1 Gestione degli Errori

Quando si verifica un errore, l'interprete stampa un messaggio di errore e una traccia dello stack. In modalità interattiva, quindi ritorna al prompt principale; quando l'input proviene da un file, esce con uno stato di uscita diverso da zero dopo aver stampato la traccia dello stack. (Le eccezioni gestite da una clausola `except` in un'istruzione `try` non sono errori in questo contesto.) Alcuni errori sono incondizionatamente fatali e causano un'uscita con uno stato di uscita diverso da zero; questo si applica alle incoerenze interne e a certi casi di esaurimento della memoria. Tutti i messaggi di errore sono scritti sul flusso standard degli errori; l'output normale dei comandi eseguiti è scritto sul flusso standard dell'output.

Digitare il carattere d'interruzione (di solito `Control-C` o `Delete`) al prompt principale o secondario cancella l'input e ritorna al prompt principale.¹ Digitando un'interruzione mentre un comando è in esecuzione si solleva l'eccezione `KeyboardInterrupt`, che può essere gestita da un'istruzione `try`.

16.1.2 Script Eseguibili Python

Nei sistemi Unix di tipo BSD, gli script Python possono essere resi direttamente eseguibili, come gli script di shell, aggiungendo la riga

¹ Un problema con il pacchetto GNU Readline potrebbe impedirlo.

```
#!/usr/bin/env python3
```

(supponendo che l'interprete sia nel `PATH` dell'utente) all'inizio dello script e dando al file un permesso eseguibile. Il `#!` deve essere i primi due caratteri del file. Su alcune piattaforme, questa prima riga deve terminare con un fine riga in stile Unix (`'\n'`), non con un fine riga in stile Windows (`'\r\n'`). Nota che il carattere cancelletto, `'#'`, è usato per iniziare un commento in Python.

Lo script può essere reso eseguibile, o avere i permessi, usando il comando `chmod`.

```
$ chmod +x myscript.py
```

Nei sistemi Windows, non esiste il concetto di «modalità eseguibile». L'installatore Python associa automaticamente i file `.py` con `python.exe` in modo che un doppio clic su un file Python lo esegua come uno script. L'estensione può anche essere `.pyw`; in tal caso, la finestra della console che appare normalmente viene soppressa.

16.1.3 Il File di Avvio Interattivo

Quando usi Python in modalità interattiva, è spesso utile avere alcuni comandi standard eseguiti ogni volta che l'interprete viene avviato. Puoi fare ciò impostando una variabile di ambiente chiamata `PYTHONSTARTUP` al nome di un file contenente i tuoi comandi di avvio. Questo è simile alla funzionalità `.profile` delle shell Unix.

Questo file viene letto solo nelle sessioni interattive, non quando Python legge i comandi da uno script, e non quando `/dev/tty` è dato come fonte esplicita dei comandi (che altrimenti si comporta come una sessione interattiva). Viene eseguito nello stesso namespace dove vengono eseguiti i comandi interattivi, quindi gli oggetti che definisce o importa possono essere usati senza qualificazioni nella sessione interattiva. Puoi anche cambiare i prompt `sys.ps1` e `sys.ps2` in questo file.

Se vuoi leggere un file di avvio aggiuntivo dalla directory corrente, puoi programmare questo nel file di avvio globale usando il codice come `if os.path.isfile('pythonrc.py'): exec(open('pythonrc.py').read())`. Se vuoi usare il file di avvio in uno script, devi farlo esplicitamente 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 I Moduli di Personalizzazione

Python fornisce due hook per permetterti di personalizzarlo: `sitecustomize` e `usercustomize`. Per vedere come funziona, devi prima trovare la posizione della directory `user site-packages`. Avvia Python ed esegui questo codice:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.x/site-packages'
```

Ora puoi creare un file chiamato `usercustomize.py` in quella directory e inserire ciò che vuoi in esso. Questo influenzerà ogni invocazione di Python, a meno che non sia avviato con l'opzione `-s` per disabilitare l'importazione automatica.

`sitecustomize` funziona allo stesso modo, ma è tipicamente creato da un amministratore del computer nella directory globale `site-packages` e viene importato prima di `usercustomize`. Per maggiori dettagli, vedi la documentazione del modulo `site`.

Note a pie” di pagina

>>>

The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

...

Can refer to:

- The default Python prompt of the *interactive* shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
- The `Ellipsis` built-in constant.

abstract base class

Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections.abc` module), numbers (in the `numbers` module), streams (in the `io` module), import finders and loaders (in the `importlib.abc` module). You can create your own ABCs with the `abc` module.

annotation

A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a *type hint*.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality. Also see *annotations-howto* for best practices on working with annotations.

argument

A value passed to a *function* (or *method*) when calling the function. There are two kinds of argument:

- *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, 3 and 5 are both keyword arguments in the following calls to `complex()`:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an *iterable* preceded by *. For example, 3 and 5 are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the *parameter* glossary entry, the FAQ question on the difference between arguments and parameters, and **PEP 362**.

asynchronous context manager

An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by **PEP 492**.

asynchronous generator

A function which returns an *asynchronous generator iterator*. It looks like a coroutine function defined with `async def` except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain `await` expressions as well as `async for`, and `async with` statements.

asynchronous generator iterator

An object created by a *asynchronous generator* function.

This is an *asynchronous iterator* which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

asynchronous iterable

An object, that can be used in an `async for` statement. Must return an *asynchronous iterator* from its `__aiter__()` method. Introduced by **PEP 492**.

asynchronous iterator

An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__()` must return an *awaitable* object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by **PEP 492**.

attribute

A value associated with an object which is usually referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

It is possible to give an object an attribute whose name is not an identifier as defined by identifiers, for example using `setattr()`, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with `getattr()`.

awaitable

An object that can be used in an `await` expression. Can be a *coroutine* or an object with an `__await__()` method. See also **PEP 492**.

BDFL

Benevolent Dictator For Life, a.k.a. [Guido van Rossum](#), Python's creator.

binary file

A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode ('rb', 'wb' or 'rb+'), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

See also *text file* for a file object able to read and write `str` objects.

borrowed reference

In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

bytes-like object

An object that supports the bufferobjects and can export a *C-contiguous* buffer. This includes all `bytes`, `bytearray`, and `array.array` objects, as well as many common `memoryview` objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as «read-write bytes-like objects». Example mutable buffer objects include `bytearray` and a `memoryview` of a `bytearray`. Other operations require the binary data to be stored in immutable objects («read-only bytes-like objects»); examples of these include `bytes` and a `memoryview` of a `bytes` object.

bytecode

Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This «intermediate language» is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the `dis` module.

callable

A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

callback

A subroutine function which is passed as an argument to be executed at some point in the future.

class

A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

class variable

A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

closure variable

A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

complex number

An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of -1), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

context

This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

context management protocol

The `__enter__()` and `__exit__()` methods called by the `with` statement. See [PEP 343](#).

context manager

An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See [PEP 343](#).

context variable

A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous

A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

coroutine

Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also [PEP 492](#).

coroutine function

A function which returns a *coroutine* object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords. These were introduced by [PEP 492](#).

CPython

The canonical implementation of the Python programming language, as distributed on python.org. The term «CPython» is used when necessary to distinguish this implementation from others such as Jython or IronPython.

current context

The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

decorator

A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

descriptor

Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see `descriptors` or the `Descriptor How To Guide`.

dictionary

An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary comprehension

A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See `comprehensions`.

dictionary view

The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See `dict-views`.

docstring

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing

A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used («If it looks like a duck and quacks like a duck, it must be a duck.») By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with *abstract base classes*.) Instead, it typically employs `hasattr()` tests or *EAFP* programming.

EAFP

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

expression

A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statements* which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions.

extension module

A module written in C or C++, using Python's C API to interact with the core and with user code.

f-string

String literals prefixed with `'f'` or `'F'` are commonly called «f-strings» which is short for formatted string literals. See also [PEP 498](#).

file object

An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw *binary files*, buffered *binary files* and *text files*. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

file-like object

A synonym for *file object*.

filesystem encoding and error handler

Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

See also the *locale encoding*.

finder

An object that tries to find the *loader* for a module that is being imported.

There are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See `finders-and-loaders` and `importlib` for much more detail.

floor division

Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to 2 in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*. See [PEP 238](#).

free threading

A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See [PEP 703](#).

free variable

Formally, as defined in the language execution model, a free variable is any variable used in a namespace which

is not a local variable in that namespace. See [closure variable](#) for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for [closure variable](#).

function

A series of statements which returns some value to a caller. It can also be passed zero or more [arguments](#) which may be used in the execution of the body. See also [parameter](#), [method](#), and the [function](#) section.

function annotation

An [annotation](#) of a function parameter or return value.

Function annotations are usually used for [type hints](#): for example, this function is expected to take two `int` arguments and is also expected to have an `int` return value:

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

Function annotation syntax is explained in [section function](#).

See [variable annotation](#) and [PEP 484](#), which describe this functionality. Also see [annotations-howto](#) for best practices on working with annotations.

`__future__`

A `future` statement, from `__future__` import <feature>, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection

The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the `gc` module.

generator

A function which returns a [generator iterator](#). It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a `for`-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

generator iterator

An object created by a [generator](#) function.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending `try`-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression

An [expression](#) that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, `range`, and an optional `if` clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

generic function

A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the [single dispatch](#) glossary entry, the `functools.singledispatch()` decorator, and [PEP 443](#).

generic type

A *type* that can be parameterized; typically a container class such as `list` or `dict`. Used for *type hints* and *annotations*.

For more details, see generic alias types, [PEP 483](#), [PEP 484](#), [PEP 585](#), and the `typing` module.

GIL

See *global interpreter lock*.

global interpreter lock

The mechanism used by the *CPython* interpreter to assure that only one thread executes Python *bytecode* at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see [PEP 703](#).

hash-based pyc

A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See *pyc-invalidation*.

hashable

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

Most of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their `id()`.

IDLE

An Integrated Development and Learning Environment for Python. `idle` is a basic editor and interpreter environment which ships with the standard distribution of Python.

immortal

Immortal objects are a CPython implementation detail introduced in [PEP 683](#).

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

immutable

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

import path

A list of locations (or *path entries*) that are searched by the *path based finder* for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package's `__path__` attribute.

importing

The process by which Python code in one module is made available to Python code in another module.

importer

An object that both finds and loads a module; both a *finder* and *loader* object.

interactive

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see *Modalità Interattiva*.

interpreted

Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

interpreter shutdown

When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the *garbage collector*. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the `__main__` module or the script being run has finished executing.

iterable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in *typeiter*.

Dettaglio dell'implementazione di CPython: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

key function

A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example, the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a `lambda` expression

such as `lambda r: (r[0], r[2])`. Also, `operator.attrgetter()`, `operator.itemgetter()`, and `operator.methodcaller()` are three key function constructors. See the [Sorting HOW TO](#) for examples of how to create and use key functions.

keyword argument

See [argument](#).

lambda

An anonymous inline function consisting of a single [expression](#) which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

LBYL

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the [EAFP](#) approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between «the looking» and «the leaping». For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

list

A built-in Python [sequence](#). Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

list comprehension

A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader

An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a [finder](#). See also:

- [finders-and-loaders](#)
- `importlib.abc.Loader`
- **PEP 302**

locale encoding

On Unix, it is the encoding of the `LC_CTYPE` locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: "cp1252").

On Android and VxWorks, Python uses "utf-8" as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the [filesystem encoding and error handler](#).

magic method

An informal synonym for [special method](#).

mapping

A container object that supports arbitrary key lookups and implements the methods specified in the `collections.abc.Mapping` or `collections.abc.MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

meta path finder

A [finder](#) returned by a search of `sys.meta_path`. Meta path finders are related to, but different from [path entry finders](#).

See `importlib.abc.MetaPathFinder` for the methods that meta path finders implement.

metaclass

The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in metaclasses.

method

A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first *argument* (which is usually called `self`). See *function* and *nested scope*.

method resolution order

Method Resolution Order is the order in which base classes are searched for a member during lookup. See `python_2.3_mro` for details of the algorithm used by the Python interpreter since the 2.3 release.

module

An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of *importing*.

See also *package*.

module spec

A namespace containing the import-related information used to load a module. An instance of `importlib.machinery.ModuleSpec`.

See also `module-specs`.

MRO

See *method resolution order*.

mutable

Mutable objects can change their value but keep their `id()`. See also *immutable*.

named tuple

The term «named tuple» applies to any type or class that inherits from `tuple` and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info`:

```
>>> sys.float_info[1]           # indexed access
1024
>>> sys.float_info.max_exp      # named field access
1024
>>> isinstance(sys.float_info, tuple) # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

namespace

The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

namespace package

A [PEP 420 package](#) which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a [regular package](#) because they have no `__init__.py` file.

See also [module](#).

nested scope

The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.

new-style class

Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattr__()`, class methods, and static methods.

object

Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any [new-style class](#).

optimized scope

A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package

A Python [module](#) which can contain submodules or recursively, subpackages. Technically, a package is a Python module with a `__path__` attribute.

See also [regular package](#) and [namespace package](#).

parameter

A named entity in a [function](#) (or method) definition that specifies an [argument](#) (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either [positionally](#) or as a [keyword argument](#). This is the default kind of parameter, for example *foo* and *bar* in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Positional-only parameters can be defined by including a `/` character in the parameter list of the function definition after them, for example *posonly1* and *posonly2* in the following:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example *args* in the following:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example *kwargs* in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the [argument](#) glossary entry, the FAQ question on the difference between arguments and parameters, the `inspect.Parameter` class, the function section, and [PEP 362](#).

path entry

A single location on the [import path](#) which the [path based finder](#) consults to find modules for importing.

path entry finder

A *finder* returned by a callable on `sys.path_hooks` (i.e. a [path entry hook](#)) which knows how to locate modules given a [path entry](#).

See `importlib.abc.PathEntryFinder` for the methods that path entry finders implement.

path entry hook

A callable on the `sys.path_hooks` list which returns a [path entry finder](#) if it knows how to find modules on a specific [path entry](#).

path based finder

One of the default [meta path finders](#) which searches an [import path](#) for modules.

path-like object

An object representing a file system path. A path-like object is either a `str` or `bytes` object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a `str` or `bytes` file system path by calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a `str` or `bytes` result instead, respectively. Introduced by [PEP 519](#).

PEP

Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See [PEP 1](#).

portion

A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in [PEP 420](#).

positional argument

See [argument](#).

provisional API

A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a «solution of last resort» - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See [PEP 411](#) for more details.

provisional package

See [provisional API](#).

Python 3000

Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated «Py3k».

Pythonic

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)) :
    print(food[i])
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print(piece)
```

qualified name

A dotted name showing the «path» from a module's global scope to a class, function or method defined in that module, as defined in [PEP 3155](#). For top-level functions and classes, the qualified name is the same as the object's name:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

regular package

A traditional *package*, such as a directory containing an `__init__.py` file.

See also *namespace package*.

REPL

An acronym for the «read–eval–print loop», another name for the *interactive* interpreter shell.

__slots__

A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence

An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *hashable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see [Common Sequence Operations](#).

set comprehension

A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See [comprehensions](#).

single dispatch

A form of *generic function* dispatch where the implementation is chosen based on the type of a single argument.

slice

An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally.

soft deprecated

A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See [PEP 387: Soft Deprecation](#).

special method

A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in [specialnames](#).

statement

A statement is part of a suite (a «block» of code). A statement is either an *expression* or one of several constructs with a keyword, such as `if`, `while` or `for`.

static type checker

An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also [type hints](#) and the `typing` module.

strong reference

In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

See also [borrowed reference](#).

text encoding

A string in Python is a sequence of Unicode code points (in range U+0000–U+10FFFF). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as «encoding», and recreating the string from the sequence of bytes is known as «decoding».

There are a variety of different text serialization codecs, which are collectively referred to as «text encodings».

text file

A *file object* able to read and write `str` objects. Often, a text file actually accesses a byte-oriented datastream and handles the *text encoding* automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

See also *binary file* for a file object able to read and write *bytes-like objects*.

triple-quoted string

A string which is bound by three instances of either a quotation mark (`»`) or an apostrophe (`“`). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type

The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

type alias

A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying *type hints*. For example:

```
def remove_gray_shades(  
    colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:  
    pass
```

could be made more readable like this:

```
Color = tuple[int, int, int]  
  
def remove_gray_shades(colors: list[Color]) -> list[Color]:  
    pass
```

See *typing* and **PEP 484**, which describe this functionality.

type hint

An *annotation* that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using `typing.get_type_hints()`.

See *typing* and **PEP 484**, which describe this functionality.

universal newlines

A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `bytes.splitlines()` for an additional use.

variable annotation

An *annotation* of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
class C:  
    field: 'annotation'
```

Variable annotations are usually used for *type hints*: for example this variable is expected to take `int` values:

```
count: int = 0
```

Variable annotation syntax is explained in section [annassign](#).

See [function annotation](#), [PEP 484](#) and [PEP 526](#), which describe this functionality. Also see [annotations-howto](#) for best practices on working with annotations.

virtual environment

A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also `venv`.

virtual machine

A computer defined entirely in software. Python's virtual machine executes the *bytecode* emitted by the bytecode compiler.

Zen of Python

Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing `«import this»` at the interactive prompt.

About this documentation

Python's documentation is generated from [reStructuredText](#) sources using [Sphinx](#), a documentation generator originally created for Python and now maintained as an independent project.

Lo sviluppo della documentazione e della sua toolchain è uno lavoro svolto esclusivamente da volontari, proprio come lo stesso Python. Se si desidera contribuire, si prega di dare un'occhiata alla pagina [reporting-bugs](#) per avere maggiori informazioni su come farlo. Nuovi volontari sono sempre i benvenuti!

Molte grazie a:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and author of much of the content;
- the [Docutils](#) project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 Contributors to the Python documentation

Many people have contributed to the Python language, the Python standard library, and the Python documentation. See [Misc/ACKS](#) in the Python source distribution for a partial list of contributors.

È solo con il contributo dei membri della comunità di Python che Python ha una documentazione così meravigliosa — Grazie!

Storia e licenza

C.1 Storia del software

Python è stato creato all'inizio degli anni '90 da Guido van Rossum allo Stichting Mathematisch Centrum (CWI, <https://www.cwi.nl/>) nei Paesi Bassi a partire dal linguaggio ABC. Guido rimane l'autore principale di Python, anche se questo include molti contributi da parte di altre persone.

Nel 1995 Guido ha continuato il suo lavoro su Python presso la Corporation for National Research Initiatives (CNRI, vedi <https://www.cnri.reston.va.us/>) a Reston, Virginia, dove ha rilasciato diverse versioni del software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen Python-Labs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <https://www.zope.org/>). In 2001, the Python Software Foundation (PSF, see <https://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

Tutte le versioni di Python sono Open Source (vedi <https://opensource.org/> per la definizione di Open Source). Storicamente la maggior parte, ma non tutte, le versioni di Python sono state compatibili con la GPL; la tabella seguente riassume le varie versioni.

Rilascio	Derivato da	Anno	Proprietario	Compatibile con la GPL?
Da 0.9.0 a 1.2	n/d	1991-1995	CWI	sì
Da 1.3 a 1.5.2	1.2	1995-1999	CNRI	sì
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	sì
2.1.1	2.1+2.0.1	2001	PSF	sì
2.1.2	2.1.1	2002	PSF	sì
2.1.3	2.1.2	2002	PSF	sì
2.2 e superiori	2.1.1	2001-adesso	PSF	sì

GPL-compatibile non significa che stiamo distribuendo Python sotto la GPL. Tutte le licenze Python, a differenza della GPL, consentono di distribuire una versione modificata senza rendere le modifiche open source. Le licenze compatibili con la GPL permettono di combinare Python con altri software rilasciati sotto la GPL; le altre no.

Grazie ai tanti volontari esterni che hanno lavorato sotto la direzione di Guido per rendere possibili queste *release*.

C.2 Termini e condizioni di accesso o di utilizzo di Python

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenze e riconoscimenti per il software incorporato* for an incomplete list of these licenses.

C.2.1 PSF ACCORDO DI LICENZA PER PYTHON 3.13.1

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.13.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.13.1 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001–2024 Python Software Foundation; All Rights Reserved" are retained in Python 3.13.1 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 3.13.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 3.13.1.
4. PSF is making Python 3.13.1 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.13.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.13.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.13.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 3.13.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.2 CONTRATTO DI LICENZA DI BEOPEN.COM PER PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.3 CNRI CONTRATTO DI LICENZA PER PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works,

(continues on next page)

(continua dalla pagina precedente)

distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the internet using the following URL: <http://hdl.handle.net/1895.22/1013>."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 CWI CONTRATTO DI LICENZA PER PYTHON DA 0.9.0 A 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

(continues on next page)

(continua dalla pagina precedente)

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.1 DOCUMENTATION

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3 Licenze e riconoscimenti per il software incorporato

Questa sezione è una lista incompleta, ma in crescita, di licenze e riconoscimenti per software di terze parti incorporate nella distribuzione Python.

C.3.1 Mersenne Twister

The `_random` C extension underlying the `random` module includes code based on a download from <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)`
or `init_by_array(init_key, key_length)`.

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

(continues on next page)

(continua dalla pagina precedente)

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

C.3.2 Socket

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, <https://www.wide.ad.jp/>.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT

(continues on next page)

(continua dalla pagina precedente)

LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.3 Servizi di socket asincrone

The `test.support.asyncchat` and `test.support.asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.4 Gestione dei cookie

Il modulo `http.cookies` contiene il seguente avviso:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.5 Tracciabilità dell'esecuzione

Il modulo `trace` contiene il seguente avviso:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 Funzioni `UUencode` e `UUdecode`

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
    All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 Chiamate di procedura remota XML

Il modulo `xmlrpc.client` contiene il seguente avviso:

```
The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test.test_epoll` module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

Il modulo `select` contiene il seguente avviso per l'interfaccia `kqueue`:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski's implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
  https://github.com/majek/csiphash/

Solution inspired by code from:
  Samuel Neves (supercop/crypto_auth/siphash24/little)
  djb (supercop/crypto_auth/siphash24/little2)
  Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod e dtoa

The file `Python/dtoa.c`, which supplies C functions `dtoa` and `strtod` for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from <https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c>. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```

/*****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 *****/

```

C.3.12 7.4 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```

                                Apache License
                                Version 2.0, January 2004
                                https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction,
and distribution as defined by Sections 1 through 9 of this document.

"Licenser" shall mean the copyright owner or entity authorized by
the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all
other entities that control, are controlled by, or are under common
control with that entity. For the purposes of this definition,
"control" means (i) the power, direct or indirect, to cause the
direction or management of such entity, whether by contract or
otherwise, or (ii) ownership of fifty percent (50%) or more of the
outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity
exercising permissions granted by this License.

```

(continues on next page)

(continua dalla pagina precedente)

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You

(continues on next page)

(continua dalla pagina precedente)

institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

(continues on next page)

(continua dalla pagina precedente)

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

C.3.13 expat

The pyexpat extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

(continues on next page)

(continua dalla pagina precedente)

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ``Software''), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.15 zlib

L'estensione `zlib` è costruita usando una copia dei sorgenti `zlib` se la versione `zlib` trovata sul sistema è troppo vecchia per essere usata per la build:

Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be

(continues on next page)

(continua dalla pagina precedente)

```
appreciated but is not required.
```

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly
jloup@gzip.org

Mark Adler
madler@alumni.caltech.edu

C.3.16 cfuhash

L'implementazione della tabella hash utilizzata da `tracemalloc` si basa sul progetto cfuhash:

```
Copyright (c) 2005 Don Owens  
All rights reserved.
```

```
This code is released under the BSD license:
```

```
Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions  
are met:
```

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS  
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT  
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS  
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE  
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,  
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES  
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR  
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)  
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,  
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)  
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED  
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without
```

(continues on next page)

(continua dalla pagina precedente)

modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at <https://www.w3.org/TR/xml-c14n2-testcases/> and is distributed under the 3-clause BSD license:

Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018–2021 Microsoft Corporation, Daan Leijen
```

```
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:
```

```
The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from [uvloop 0.16](#), which is distributed under the MIT license:

```
Copyright (c) 2015–2021 MagicStack Inc. http://magic.io
```

```
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
```

```
The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's «Global Unbounded Sequences» safe memory reclamation scheme in `subr_smr.c`. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>
```

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
```

(continues on next page)

(continua dalla pagina precedente)

are met:

1. Redistributions of source code must retain the above copyright notice unmodified, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

APPENDICE D

Copyright

Python e questa documentazione sono protetti da:

Copyright © 2001-2024 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. Tutti i diritti riservati.

Copyright © 1995-2000 Corporation for National Research Initiatives. Tutti i diritti riservati.

Copyright © 1991-1995 Stichting Mathematisch Centrum. Tutti i diritti riservati.

Fare riferimento a [Storia e licenza](#) per informazioni complete su licenza e permessi.

Non-alphabetical

..., [125](#)

(*hash*)

commento, [9](#)

* (*asterisco*)

nelle chiamate di funzione, [32](#)

**

nelle chiamate di funzione, [33](#)

: (*due punti*)

annotazioni delle funzioni, [34](#)

->

annotazioni delle funzioni, [34](#)

>>>, [125](#)

__all__, [56](#)

__future__, [131](#)

__slots__, [138](#)

A

abstract base class, [125](#)

annotation, [125](#)

annotazioni

funzione, [34](#)

apri

funzione incorporata, [63](#)

argument, [125](#)

asynchronous context manager, [126](#)

asynchronous generator, [126](#)

asynchronous generator iterator, [126](#)

asynchronous iterable, [126](#)

asynchronous iterator, [126](#)

attribute, [126](#)

awaitable, [126](#)

B

BDFL, [126](#)

binary file, [127](#)

borrowed reference, [127](#)

builtins

modulo, [53](#)

bytecode, [127](#)

bytes-like object, [127](#)

C

callable, [127](#)

callback, [127](#)

C-contiguous, [128](#)

cerca

percorso, modulo, [51](#)

class, [127](#)

class variable, [127](#)

closure variable, [127](#)

codifica

stile, [34](#)

complex number, [128](#)

context, [128](#)

context management protocol, [128](#)

context manager, [128](#)

context variable, [128](#)

contiguous, [128](#)

coroutine, [128](#)

coroutine function, [128](#)

CPython, [128](#)

current context, [129](#)

D

decorator, [129](#)

definizione

for, [19](#)

descriptor, [129](#)

dictionary, [129](#)

dictionary comprehension, [129](#)

dictionary view, [129](#)

docstring, [129](#)

docstrings, [26](#), [33](#)

duck-typing, [129](#)

E

EAFP, [130](#)

expression, [130](#)

extension module, [130](#)

F

f-string, [130](#)

file

oggetto, [63](#)

file object, [130](#)

file-like object, [130](#)

filesystem encoding and error handler, [130](#)

- finder, [130](#)
- floor division, [130](#)
- for
 - definizione, [19](#)
- formatted string literal, [60](#)
- Fortran contiguous, [128](#)
- free threading, [130](#)
- free variable, [130](#)
- fstring, [60](#)
- f-string, [60](#)
- function, [131](#)
- function annotation, [131](#)
- funzione
 - annotazioni, [34](#)
- funzione built-in
 - help, [93](#)
- funzione incorporata
 - apri, [63](#)

G

- garbage collection, [131](#)
- generator, [131](#)
- generator expression, [131](#)
- generator iterator, [131](#)
- generic function, [131](#)
- generic type, [132](#)
- GIL, [132](#)
- global interpreter lock, [132](#)

H

- hash-based pyc, [132](#)
- hashable, [132](#)
- help
 - funzione built-in, [93](#)

I

- IDLE, [132](#)
- immortal, [132](#)
- immutable, [132](#)
- import path, [132](#)
- importer, [133](#)
- importing, [132](#)
- interactive, [133](#)
- interpolated string literal, [60](#)
- interpreted, [133](#)
- interpreter shutdown, [133](#)
- iterable, [133](#)
- iterator, [133](#)

J

- json
 - modulo, [65](#)

K

- key function, [133](#)
- keyword argument, [134](#)

L

- lambda, [134](#)
- LBYL, [134](#)
- list, [134](#)
- list comprehension, [134](#)
- loader, [134](#)
- locale encoding, [134](#)

M

- magic
 - method, [134](#)
- magic method, [134](#)
- mapping, [134](#)
- meta path finder, [134](#)
- metaclass, [135](#)
- method, [135](#)
 - magic, [134](#)
 - special, [139](#)
- method resolution order, [135](#)
- metodo
 - oggetto, [83](#)
- module, [135](#)
- module spec, [135](#)
- modulo
 - builtins, [53](#)
 - cerca percorso, [51](#)
 - json, [65](#)
 - sys, [52](#)
- MRO, [135](#)
- mutable, [135](#)

N

- named tuple, [135](#)
- namespace, [135](#)
- namespace package, [136](#)
- nested scope, [136](#)
- new-style class, [136](#)
- nome
 - offuscamento, [88](#)

O

- object, [136](#)
- offuscamento
 - nome, [88](#)
- oggetto
 - file, [63](#)
 - metodo, [83](#)
- optimized scope, [136](#)

P

- package, [136](#)
- parameter, [136](#)
- PATH, [51](#), [122](#)
- path based finder, [137](#)
- path entry, [137](#)
- path entry finder, [137](#)
- path entry hook, [137](#)

path-like object, [137](#)
 PEP, [137](#)
 percorso
 modulo cerca, [51](#)
 portion, [137](#)
 positional argument, [137](#)
 provisional API, [137](#)
 provisional package, [137](#)
 Python 3000, [138](#)
 Python Enhancement Proposals
 PEP 1, [137](#)
 PEP 8, [35](#)
 PEP 238, [130](#)
 PEP 278, [140](#)
 PEP 302, [134](#)
 PEP 343, [128](#)
 PEP 362, [126](#), [137](#)
 PEP 411, [137](#)
 PEP 420, [136](#), [137](#)
 PEP 443, [131](#)
 PEP 483, [132](#)
 PEP 484, [34](#), [125](#), [131](#), [132](#), [140](#), [141](#)
 PEP 492, [126](#), [128](#)
 PEP 498, [130](#)
 PEP 519, [137](#)
 PEP 525, [126](#)
 PEP 526, [125](#), [141](#)
 PEP 585, [132](#)
 PEP 636, [25](#)
 PEP 683, [132](#)
 PEP 703, [130](#), [132](#)
 PEP 3107, [34](#)
 PEP 3116, [140](#)
 PEP 3147, [52](#)
 PEP 3155, [138](#)
 PYTHON_BASIC_REPL, [121](#)
 PYTHON_GIL, [132](#)
 Pythonic, [138](#)
 PYTHONPATH, [51](#), [53](#)
 PYTHONSTARTUP, [122](#)

Q

qualified name, [138](#)

R

reference count, [138](#)
 regular package, [138](#)
 REPL, [138](#)
 RFC
 RFC 2822, [98](#)

S

sequence, [139](#)
 set comprehension, [139](#)
 single dispatch, [139](#)
 sitecustomize, [122](#)
 slice, [139](#)
 soft deprecated, [139](#)

special
 method, [139](#)
 special method, [139](#)
 statement, [139](#)
 static type checker, [139](#)
 stile
 codifica, [34](#)
 string
 formatted literal, [60](#)
 interpolated literal, [60](#)
 stringhe di documentazione, [26](#), [33](#)
 stringhe, documentazione, [26](#), [33](#)
 strong reference, [139](#)
 sys
 modulo, [52](#)

T

text encoding, [139](#)
 text file, [140](#)
 triple-quoted string, [140](#)
 type, [140](#)
 type alias, [140](#)
 type hint, [140](#)

U

universal newlines, [140](#)
 usercustomize, [122](#)

V

variabile d'ambiente, PATH, [51](#), [122](#)
 variabile d'ambiente, PYTHON_BASIC_REPL, [121](#)
 variabile d'ambiente, PYTHON_GIL, [132](#)
 variabile d'ambiente, PYTHONPATH, [51](#), [53](#)
 variabile d'ambiente, PYTHONSTARTUP, [122](#)
 variable annotation, [140](#)
 virtual environment, [141](#)
 virtual machine, [141](#)

Z

Zen of Python, [141](#)