JACK FELLERS

Python

INTRODUZIONE E FONDAMENTALI PER IMPARARE A PROGRAMMARE IN PYTHON IN 7 GIORNI

WEBHAWK

PYTHON

Caro lettore, per ringraziarti per la fiducia dimostratami acquistando il mio libro, ecco per te in **regalo**, una guida per fortificare ancora di più la tua conoscenza nella programmazione web!

Scansiona il codice o clicca sul link per riscattarlo in meno di un minuto:

Link alternativo al Qr code:
https://webhawk.tech/optin-it/

Buona lettura!

INDICE

PREMESSA

Scrivere programmi, o programmare, è un'attività molto creativa e gratificante. Puoi scrivere programmi per diversi scopi: guadagnarti da vivere, risolvere un difficile problema di analisi dei dati o aiutare qualcun altro a risolvere un problema.

Questo Ebook presuppone che tutti debbano sapere come programmare e, successivamente, scoprire cosa vuoi fare con le tue nuove skills. Nella nostra quotidianità siamo circondati dalla tecnologia, a partire dai laptop fino ai telefoni cellulari. Possiamo pensare a questi computer come ai nostri "assistenti personali" che possono occuparsi di molte attività per conto nostro. L'hardware dei nostri computer attuali è essenzialmente costruito per farci continuamente la domanda: "Cosa vorresti che facessi dopo?". I nostri computer sono veloci e hanno una grande quantità di memoria che potrebbe esserci molto utile se solo conoscessimo la lingua per interagire con esso.

Se conoscessimo questa lingua, potremmo dire al computer di svolgere attività ripetitive per conto nostro ed è interessante notare che le attività in cui i computer possono fare meglio sono spesso le attività che noi umani troviamo noiose e che intorpidiscono la mente. Ad esempio, se dovessi analizzare due capitoli di un libro per trovare la parola più usata e contare quante volte è stata usata, probabilmente avresti qualche difficoltà. Sei in grado di leggere e comprendere le parole in pochi secondi ma contarle è più complesso perché non è il tipo di problema per cui le menti umane sono progettate. Per un computer, è vero il contrario, la lettura e la comprensione

del testo sono compiti difficili ma contare le parole e trovare la parola più utilizzata sono compiti molto più facili:

Python parole.py

La parola più utilizzata nel testo è: per

Tale parola è stata usata 11 volte

Il nostro programma ci ha rapidamente detto che la parola "per" è stata usata undici volte nel testo che hai già letto finora. Proprio perché i computer eccellono in tutti quei compiti in cui gli esseri umani non sono bravi è il motivo per cui devi diventare un esperto del "linguaggio informatico".

Dopo aver appreso questo nuovo linguaggio, puoi delegare attività banali al tuo computer, lasciandoti più tempo per fare ciò per cui sei particolarmente adatto. Devi usare creatività, intuizione e inventiva in questa fase e, sebbene questo libro non sia destinato a programmatori professionisti, programmare può essere un lavoro molto gratificante sia finanziariamente che personalmente.

Costruire programmi utili, eleganti e intelligenti e che gli altri possano usare è un'attività molto creativa e interessante. Il tuo computer, di solito, contiene molti programmi diversi e creati da diversi gruppi di programmatori, ognuno in competizione per avere la tua attenzione e il tuo interesse. I programmatori fanno del loro meglio per soddisfare le tue esigenze e per offrirti un'ottima esperienza utente durante l'uso.

Per ora, la nostra motivazione principale non è fare soldi o piacere agli utenti finali ma, piuttosto, essere più produttivi nella gestione dei dati e delle informazioni che incontreremo nel corso della nostra vita. Al primo avvio, sarai sia il programmatore che l'utente finale dei tuoi programmi, man mano che acquisisci abilità come programmatore, la programmazione ti sembrerà più creativa e le tue idee potranno orientarsi verso lo sviluppo di programmi per gli altri.

LA PROGRAMMAZIONE

Nel corso di questo e-book proveremo a trasformarti in una persona esperta in programmazione. Alla fine, sarai un programmatore, sicuramente non un programmatore professionista ma almeno avrai le competenze per esaminare un problema e sviluppare un programma per risolverlo. In un certo senso, hai bisogno di due abilità per essere programmatore:

• Innanzitutto, devi conoscere il linguaggio di programmazione (Python in questo caso) che è composto dal vocabolario e dalla grammatica. Devi essere in grado di scrivere correttamente le parole in questa nuova "lingua" e saper costruire "frasi" ben formate in questa nuova "lingua";

• In secondo luogo, è necessario combinare parole e frasi per creare una storia e trasmettere un'idea. Nella programmazione, il nostro programma è la "storia" e il problema che stai cercando di risolvere è l'"idea".

Dopo aver appreso un linguaggio di programmazione come Python, sarà molto più semplice imparare un secondo linguaggio di programmazione come JavaScript o C++. Tutti i linguaggi di programmazione hanno un vocabolario e una grammatica diversi tra loro ma le capacità di risoluzione dei problemi saranno le stesse in tutti i linguaggi di programmazione. Imparerai il "vocabolario" e le "frasi" di Python abbastanza rapidamente in quanto molto semplici. Scrivere un programma coerente per risolvere un problema potrebbe richiedere più tempo perché prima bisogna apprendere le basi. Proprio come abbiamo imparato la lingua italiana, dapprima imparando a scrivere le parole, successivamente saremo in grado di leggere e spiegare i programmi.

Iniziando a programmare ti accorgerai che la programmazione diventerà un processo molto piacevole e creativo. Iniziamo con il vocabolario e la struttura dei programmi Python, sii paziente perché useremo esempi semplici ma che si rivelano davvero utili.

Parole chiave

A differenza dei linguaggi umani, il vocabolario di Python è in realtà piuttosto ridotto. Questo "vocabolario" costituisce le "parole riservate" ovvero parole che hanno un significato molto speciale per Python. Durante la scrittura dei tuoi programmi, potrai creare le tue parole ovvero parole che hanno un significato importante per te e sono chiamate **variabili**.

Potrai scegliere i nomi per le tue variabili ma non potrai usare nessuna delle parole riservate di Python come nome di una variabile. Le parole riservate in Python sono le seguenti:

and del global not with as elif if or yield assert else import pass break except in raise class finally is return continue for lambda try def from nonlocal while

Impareremo pian piano queste parole riservate e il modo in cui vengono utilizzate nel corso di questo e-book. Possiamo imporre a Python di mostrare un messaggio con l'istruzione:

print ('Hello World!')

In questo modo abbiamo scritto la nostra prima frase Python sintatticamente corretta.

La nostra frase inizia con la funzione print seguita da una stringa di testo di nostra scelta racchiusa tra virgolette singole. Le stringhe nelle istruzioni di stampa sono racchiuse tra virgolette che possono essere singole o doppie. La maggior parte dei programmatori usa virgolette singole ma in alcuni casi questo può rappresentare un problema in quanto possono essere interpretate in modo errato da Python.

Ora che abbiamo una parola e una semplice frase che conosciamo in Python, dobbiamo sapere come iniziare una conversazione con Python per poter interagire con esso.

Prima di poter conversare con Python, devi installare il software sul tuo computer e scoprire come avviare Python sul tuo computer. Il processo di installazione è molto semplice e guidato da un’interfaccia grafica curata ed essenziale. Cerca maggiori informazioni sul sito ufficiale https://www.python.org/downloads/. Ti preannuncio una buona notizia: se sei un utente Linux o Mac OS molto probabilmente Python è già installato nel tuo sistema. Ad ogni modo, apri una finestra del terminale e digita il comando

python3 e, se installato, l'interprete Python inizierà l'esecuzione in modalità interattiva e apparirà qualcosa di simile:

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Dec 20 2019, 18:57:59)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Il prompt >>> è il modo in cui l'interprete di Python ti chiede: "Cosa vuoi fare adesso?" Python è pronto per conversare con te. Tutto quello che devi fare è parlare la lingua di Python.

Puoi stampare a video alcune frasi come:

```
print('Buongiorno')
print('Vengo in pace')
print('Mi piacerebbe imparare Python')
print 'perchè è molto potente'
File "<stdin>", line 1
print 'perchè è molto potente
^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

La conversazione è andata abbastanza bene per un po' e poi hai commesso un piccolo errore usando il linguaggio Python. A questo punto, capisci che nonostante Python sia incredibilmente complesso e potente, è molto esigente sulla sintassi che usi. In realtà Python non è così intelligente come credi, stai davvero solo conversando con te stesso ma stai usando la sintassi corretta. In un certo senso, quando usi un programma scritto da qualcun altro, la conversazione è tra te e gli altri programmatori usando Python che agisce come intermediario.

Python viene usato per esprimere come deve procedere la conversazione.

Per terminare una conversazione con Python potresti chiudere la finestra del terminale o, in alternativa, in modo più elegante puoi usare il comando quit():

```
>>> quit()
```

Interprete e compilatore

Python è un linguaggio di alto livello destinato ad essere semplice da leggere e da scrivere per gli esseri umani e semplice da leggere ed elaborare per i computer. Altri linguaggi di alto livello sono Java, C++, PHP, Ruby, Basic, Perl, JavaScript e molti altri. L'hardware all'interno della Central Processing Unit (CPU) non comprende nessuno di questi linguaggi di alto livello infatti la CPU capisce un linguaggio detto **linguaggio macchina**. Il linguaggio macchina è molto semplice e francamente molto noioso da scrivere perché è rappresentato soltanto da zeri e uno che ricordano tanto Matrix:

00101000111010010010101000000111
11100110000011101010010101101101

Il linguaggio macchina sembra piuttosto semplice in superficie, dato che ci sono solo zero e uno ma la sua sintassi è molto più complessa di Python, quindi pochissimi programmatori scrivono in linguaggio macchina. Si tende a costruire diversi traduttori per consentire ai programmatori di scrivere in linguaggi di alto livello come Python o JavaScript e questi traduttori si occupano di convertire i programmi in linguaggio macchina per un'esecuzione effettiva da parte della CPU.

I programmi scritti in linguaggi di alto livello possono essere spostati tra computer diversi utilizzando un interprete diverso sulla nuova macchina o ricompilando il codice per creare una versione del linguaggio del programma per la nuova macchina. Questi traduttori di linguaggio di programmazione rientrano in due categorie generali: **interpreti** e **compilatori**.

Un interprete legge il codice sorgente del programma come scritto dal programmatore, analizza il codice sorgente ed interpreta le istruzioni al volo. Python è un interprete e quando eseguiamo Python in modalità interattiva, possiamo digitare un comando valido di Python (o una frase) ed esso lo elabora immediatamente diventando di nuovo disponibile per aggiungere un'altra richiesta.

Alcune righe dicono a Python di ricordare un valore per il futuro quindi dobbiamo scegliere un nome per memorizzare quel valore. Di solito viene usato il termine **variabile** per fare riferimento alle etichette di questi valori:

```
>>> x = 5
>>> print(x)
5


>>> y = x * 4
>>> print(y)
20
```

In questo esempio, chiediamo a Python di ricordare il valore cinque e di usare l'etichetta x in modo da poter recuperare il valore in un secondo momento. Verifichiamo che Python ha effettivamente ricordato il valore usando print. Quindi chiediamo a Python di recuperare x e moltiplicarlo per quattro e mettere il valore appena calcolato in y, infine, chiediamo a Python di stampare il valore attualmente in y.

Anche se stiamo digitando questi comandi in Python uno alla volta, Python li tratta come una sequenza ordinata di istruzioni dove le istruzioni successive sono in grado di recuperare i dati creati con istruzioni precedenti. Stiamo scrivendo il nostro primo “paragrafo” con quattro frasi in un ordine logico e significativo.

Un compilatore, invece, deve memorizzare l'intero programma in un file, quindi esegue un processo per tradurre il codice sorgente di alto livello in linguaggio macchina e quindi il compilatore inserisce il linguaggio macchina risultante in un file per l'esecuzione. Se si dispone di un sistema Windows, spesso questi programmi di linguaggio macchina eseguibile hanno un suffisso di ".exe" o ".dll" che significano rispettivamente "eseguibile" e "linklibrary dinamico". In Linux e Macintosh, non esiste alcun suffisso che contrassegni in modo univoco un file come eseguibile.

L'interprete Python è scritto in un linguaggio di alto livello chiamato C. Puoi cercare il codice sorgente per l'interprete Python visitando il sito ufficiale a questo indirizzo https://www.python.org/ perché si tratta di un linguaggio open source quindi chiunque può contribuire al suo sviluppo.

Quando hai installato Python sul tuo computer (o il fornitore lo ha installato), hai/ha effettuato una copia del codice macchina del programma Python tradotto per il tuo sistema.

Un programma

La definizione di **programma** nella sua forma più semplice è una sequenza di istruzioni Python che sono state create per risolvere un problema. Anche una sola istruzione print può essere considerata un programma. È un programma ad una riga e non è particolarmente utile ma nella definizione più rigorosa, è un programma Python. Potrebbe essere più facile capire cosa sia un programma pensando al problema per cui il programma è stato costruito, quindi considera il programma come un aiuto a risolvere quel problema.

immaginiamo che stai facendo delle ricerche sui post di Facebook e sei interessato alla parola più utilizzata in una serie di post. Potresti stampare il flusso di post di Facebook e scandagliare il testo cercando la parola più comune, ma ciò richiederebbe molto tempo e sarebbe molto incline agli errori. Si potrebbe scrivere un programma Python per gestire l'attività in modo rapido e accurato ma, soprattutto, in modo da poter terminare questa analisi rapidamente.

Immagina di dover svolgere questo compito osservando milioni di righe di testo. Onestamente, sarebbe più facile e veloce per te imparare Python e scrivere un programma per contare le parole di quanto non sarebbe scansionare manualmente tutte le parole. Non servono centinaia di righe per un programma così semplice, ne bastano meno di 20. Non credi sia possibile? Ecco qui:

```
    nomeFile = input('Inserisci il file:')
    file = open(nomeFile, 'r')
    contatore = dict()


for riga in file:
    parole = riga.split()
    for parola in parole:
    contatore[parola] = contatore.get(parola, 0) + 1


bigcont = None
```

```
    bigparola = None
    for parola, cont in list(contatore.items()):
    if bigcont is None or cont > bigcont:
    bigparola = parola
    bigcont = cont


print(bigparola, bigcont)
```

Leggendo questo codice noterai un misto tra italiano ed inglese, il che è normale in fase di apprendimento perché per rendere il codice più leggibile cerchiamo termini del nostro vocabolario piuttosto che di quello inglese. Tuttavia, è fondamentale imparare a scrivere codice interamente in inglese in modo che possa essere compreso da tutti senza dover usare un traduttore, specialmente se stiamo lavorando a progetti open source.

Infine, ma non meno importanti, esistono alcuni schemi concettuali di basso livello che utilizziamo per costruire programmi. Questi costrutti non sono solo per i programmi Python ma fanno parte di ogni linguaggio di programmazione dal linguaggio macchina fino ai linguaggi di alto livello.

- Esecuzione condizionale: verificare determinate condizioni e quindi eseguire o saltare una sequenza di istruzioni;
- Esecuzione ripetuta: eseguire ripetutamente alcune serie di istruzioni, di solito con qualche variazione;
- Esecuzione sequenziale: eseguire le istruzioni una dopo l'altra nell'ordine in cui si trovano nello script;
- Input: ottenere dati dal "mondo esterno". Questo potrebbe essere la lettura di dati o anche qualche tipo di sensore come un microfono o un GPS. Nel nostro caso, l’input verrà dall'utente che digita i dati sulla tastiera;
- Output: visualizzare i risultati del programma su uno schermo o salvandoli in un file o magari scrivendoli su un dispositivo come un CD per riprodurre musica;
- Riuso: scrivere una serie di istruzioni una volta e assegnare loro un nome in modo da riutilizzarle nel corso del programma.

Sembra quasi troppo semplice per essere vero, e ovviamente non è mai così semplice. L'arte di scrivere un programma consiste nel comporre ed intrecciare questi elementi di base più volte per produrre qualcosa che sia utile per i tuoi utenti. Il programma di conteggio delle parole dell’esempio precedente utilizza direttamente tutti questi schemi tranne uno e alla fine del libro sarai in grado di capire quale.

VARIABILI ED ESPRESSIONI

Un **valore** è uno degli elementi di base con cui un programma può lavorare, ad esempio una lettera o un numero. Finora abbiamo visto solo numeri e stringhe (sequenze di caratteri) come valori, questi appartengono a diversi tipi: numeri interi e stringhe. Tu (e l'interprete) potete identificare le stringhe perché sono racchiuse tra virgolette (singole o doppie) e possono essere stampate con l'istruzione print, che funziona anche per i numeri interi. Usiamo il comando python3 per avviare l'interprete.

```
python3
>>> print(4)
4
```

Se volessi sapere quale tipo abbia un valore, l'interprete può aiutarti come segue:

```
>>> type('Hello World!')
<class 'str'>
>>> type(19)
<class 'int'>
```

Non sorprende che le stringhe appartengano al tipo str e gli interi appartengano al tipo int. In modo meno ovvio, i numeri con un punto decimale appartengono a un tipo chiamato float, perché questi numeri sono rappresentati in un formato detto **virgola mobile** infatti:

```
>>> type(0.2)
<class 'float'>
```

Attenzione, qualsiasi valore posto tra virgolette viene valutato come una stringa infatti:

```
>>> type('13')
<class 'str'>
>>> type('1.2')
<class 'str'>
```

Allo stesso modo, quando digiti un numero intero molto grande, potresti essere tentato dall'uso delle virgole che separano gruppi di tre cifre, come in 1,000,000. Questo non è un numero intero valido in Python, ma non ti restituirà errore:

```
>>> print(1,000,000)
1 0 0
```

Non è affatto quello che ci aspettavamo perché Python interpreta 1,000,000 come una sequenza di numeri interi separati dalla virgola. Francamente sarebbe stato meglio restituire un errore ma questo è il primo esempio di un **errore semantico** ovvero il codice viene eseguito senza produrre un messaggio di errore ma non fa quello che ci aspettiamo.

Variabili

Una delle caratteristiche più potenti di un linguaggio di programmazione è la capacità di manipolare le variabili. Una **variabile** è un nome che fa riferimento a un valore. Un'istruzione di assegnazione crea nuove variabili e fornisce loro i valori:

```
>>> messaggio = 'Benvenuto in Python'
>>> numero = 1522
>>> piGreco = 3.1415926535897931
```

In questo esempio vengono create tre assegnazioni. La prima istruzione assegna una stringa ad una nuova variabile con nome messaggio; la seconda assegna il numero intero 1522 a numero; la terza assegna il valore (approssimativo) di π a piGreco. Per visualizzare il valore di una variabile, è possibile utilizzare un'istruzione print:

```
>>> print(numero)
1522
>>> print(pi)
3.141592653589793
```

I programmatori generalmente scelgono nomi significativi per le variabili e che documentano a cosa serve la variabile. I nomi delle variabili possono essere arbitrariamente lunghi e possono contenere sia lettere che numeri ma non possono iniziare con un numero. È consentito usare lettere maiuscole ma è una buona idea far iniziare i nomi delle variabili con una lettera minuscola.

Il carattere di sottolineatura (_) può apparire nel nome di una variabile infatti viene spesso utilizzato nei nomi con più parole, come pi_greco o nome_utente. I nomi delle variabili possono iniziare con un carattere di sottolineatura ma generalmente evitiamo di farlo, a meno che non stiamo scrivendo codice di una libreria che altri utenti possono usare.

Se si assegna ad una variabile un nome non valido, si ottiene un errore di sintassi:

```
>>> 2orchestra = 'Gran coda'
```
SyntaxError: invalid syntax

Operatori

Gli **operatori** sono simboli speciali che rappresentano calcoli aritmetici come addizione e sottrazione. I valori a cui viene applicato l'operatore sono chiamati **operandi**. Gli operatori +, -, *, / e ** eseguono rispettivamente addizione, sottrazione, moltiplicazione, divisione ed esponente, come nei seguenti esempi:

20 + 32
ora - 1
ora ***** 60 + minuti
minuti / 60
5 ****** 2
(5+9**) * (**15-7**)**

Un'**espressione** è una combinazione di valori, variabili e operatori. Un valore, singolarmente, è considerato un'espressione, così come una variabile quindi, le seguenti sono tutte espressioni legali (supponendo che alla variabile x sia stato assegnato un valore):

12
x
x + 11

Quando più di un operatore appare in un'espressione, l'ordine di valutazione dipende dalle regole di precedenza. Per gli operatori matematici, Python segue le convenzioni matematiche.

L'acronimo *PEMDAS* è un modo utile per ricordare le regole:

- Le parentesi hanno la massima precedenza e possono essere usate per forzare un'espressione a valutare l'ordine desiderato. Poiché le espressioni tra parentesi vengono valutate per prime, 2*(3-1) avrà come risultato 4 e (1+1)**(5-2) avrà come risultato 8. È inoltre possibile utilizzare le parentesi per rendere più semplice la lettura

di un'espressione come in (minuto*100) / 60, anche se non cambia il risultato;

- L'esponente ha la precedenza più alta dopo le parentesi, quindi 2**1+1 restituirà 3 e non 4, così come 3*1**3 restituirà 3, non 27;
- La moltiplicazione e la divisione hanno la stessa precedenza, che è superiore all'addizione e alla sottrazione, che hanno anche la stessa precedenza. Quindi 2*3-1 restituirà 5, non 4 mentre 6+4/2 restituirà 8, non 5;
- Gli operatori con la stessa precedenza vengono valutati da sinistra verso destra. Quindi l'espressione 5-3-1 restituirà 1 e non 3, perché 5-3 si verifica prima e quindi 1 viene sottratto da 2.

L'operatore modulo, invece, lavora su numeri interi e restituisce il resto del primo operando diviso il secondo. In Python, l'operatore modulo è contraddistinto dal segno di percentuale (%). La sintassi è la stessa di altri operatori:

```
>>> quoziente = 7 / 3
>>> print(quoziente)
2,33333333
>>> resto = 7 % 3
>>> print(resto)
1
```

Quindi 7 diviso 3 ha come risultato 2 con resto di 1. L'operatore modulo risulta sorprendentemente utile infatti puoi verificare se un numero è divisibile per un altro dato che se x%y è zero, allora x è divisibile per y. Questo ti consentirebbe di risolvere problemi matematici in modo molto più veloce: sapresti dirmi quante caramelle mi avanzerebbero se volessi distribuirne 12059 tra 2164 bambini? La risposta è 12059%2164 = 1239 caramelle rimaste.

L'operatore + è usato per lavorare anche con le stringhe ma non indica l’addizione in senso matematico. Questo operatore, usato con le stringhe,

esegue la concatenazione ovvero unisce le stringhe collegandole tra loro. Per esempio:

```
>>> numero1 = 10
>>> numero2 = 15
>>> print(numero1+numero2)
25
```

```
>>> numero1 = '100'
>>> numero2 = '150'
>>> print(numero1 + numero2)
100150
```

Allo stesso modo, l'operatore * lavora anche con le stringhe ma moltiplicando il contenuto di una stringa per un numero intero. Per esempio:

```
>>> stringa = 'Prova '
>>> numeroDiRipetizioni = 3
>>> print(stringa * numeroDiRipetizioni)
Prova Prova Prova
```

Recuperare l’input dell’utente

A volte vorremmo recuperare il valore di una variabile dall'utente tramite la sua tastiera. Python fornisce una funzione integrata chiamata input che fa proprio al caso nostro. Quando viene invocata questa funzione, il programma si arresta e attende che l'utente digiti qualcosa. Quando l'utente preme il tasto Invio, il programma riprende e restituisce ciò che l'utente ha digitato come stringa.

```
>>> variabile_utente = input()
Mi piace Python
>>> print(variabile_utente)
Mi piace Python
```

Prima di ricevere un input dall'utente, è una buona idea stampare una domanda o un’indicazione per l’utente. Puoi passare una stringa da visualizzare all'utente prima di mettere in pausa l'input:

```
>>> nome = input('Come ti chiami?\n')
Come ti chiami?
Antonio
>>> print(nome)
Antonio
```

Il carattere speciale \n alla fine del prompt rappresenta una nuova riga ovvero provoca un'interruzione della riga corrente. Ecco perché l'input dell'utente appare sotto il prompt. Se ti aspetti che l'utente digiti un numero intero, puoi provare a convertire il valore restituito in int usando la funzione int():

```
>>> domanda = 'Quanti anni hai?\n'
>>> anni = input(domanda)
Quanti anni hai?
22
>>> int(anni)
22
```

Attenzione, se l’utente digitasse qualcosa di diverso da un numero avresti un errore perché la funzione int() non riesce a tradurre come previsto. Vedremo come gestire le eccezioni nel corso Capitolo 3. Man mano che i programmi diventano più grandi e più complicati, diventano più difficili da leggere. I linguaggi formali sono densi ed è spesso difficile guardare un pezzo di codice e capire cosa si sta facendo o perché.

Per questo motivo, è una buona idea aggiungere dei **commenti** ai programmi per spiegare in linguaggio naturale cosa sta facendo il programma o spiegarne il perché. I commenti in Python iniziano con il simbolo #:

```
# cerco la percentuale di risorse usate
percentuale_disco = (uso_disco * 100) / 60
```

In questo caso il commento è stato posto su una nuova riga ma può essere posto anche sulla stessa riga dell’istruzione. L’interprete Python ignorerà tutto quello che si trova dopo il cancelletto e fino alla fine della riga. I commenti sono molto utili quando documentano caratteristiche non ovvie del codice. È ragionevole supporre che il lettore possa capire cosa fa il codice; è molto più utile spiegarne il perché. Non ha senso spiegare delle semplici assegnazioni, ha più senso spiegare in linguaggio naturale un’espressione complessa (ad esempio, una formula matematica).

Scegli dei nomi di variabili adatti perché possono ridurre la necessità di commenti, tuttavia, i lunghi nomi di variabile possono rendere difficili da leggere espressioni complesse, quindi è fondamentale trovare un compromesso.

ESECUZIONE CONDIZIONALE

Un'espressione booleana è un'espressione vera o falsa. I seguenti esempi utilizzano l'operatore ==, che confronta due operandi e produce True se sono uguali e False in caso contrario:

```
>>> 3 == 3
True
>>> 5 == 2
False
```

True e False sono valori speciali che appartengono alla classe bool infatti non sono stringhe:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Esistono diversi tipi di operatori per eseguire dei confronti:

```
x != y # x non è uguale a y
x > y # x è più grande di y
x < y # x è più piccolo di y
x >= y # x è più grande o uguale a y
x <= y # x è più piccolo o uguale a y
```

Sebbene queste operazioni probabilmente ti siano familiari, i simboli Python sono diversi dai simboli matematici per le stesse operazioni. Un errore comune consiste nell'usare il segno singolo di uguale (=) al posto di un doppio segno di uguale (==). Ricorda che = è un operatore di assegnazione mentre == è un operatore di confronto. Non esiste qualcosa come =< o =>.

Esistono tre operatori logici: **and**, **or**, e **not**. La semantica (significato) di questi operatori è simile al loro significato in inglese. L'operatore and restituisce vero se entrambe le espressioni (alla sua destra e alla sua sinistra) sono vere, falso altrimenti; l'operatore or restituisce vero se almeno una delle due è vera, falso altrimenti; l'operatore not si limita ad invertire il valore quindi falso diventerà vero e viceversa.

```
>>> 1 == 1 and 2 == 2
True
>>> 1 == 1 or 2 == 3
True
>>> 1 == 2 or 2 == 3
False
>>> not(1 == 1)
False
```

if

Per scrivere programmi utili, abbiamo quasi sempre bisogno della capacità di controllare le condizioni e cambiare di conseguenza il comportamento del programma. Le dichiarazioni condizionali ci danno questa capacità e la forma più semplice è l'uso dell'istruzione if:

```
if x > 0 :
print('uso x perché è positiva')
```

L'espressione booleana dopo l'istruzione if è chiamata **condizione**. Terminiamo l'istruzione if con i due punti (:) mentre le righe dopo l'istruzione if sono indentate (rientrate). Se la condizione logica è vera, viene eseguita l'istruzione indentata, se la condizione logica è falsa, l'istruzione indentata viene ignorata.

L'istruzione è costituita da una riga di intestazione che termina con il carattere due punti (:) seguito da un blocco indentato. Le dichiarazioni come questa sono chiamate **istruzioni composte** perché si estendono su più di una riga e non c'è limite al numero di istruzioni che possono apparire nel corpo, l'essenziale è che ne sia presente almeno una. Occasionalmente, è utile avere un corpo senza istruzioni (di solito viene usato in fase di sviluppo per indicare che devi completare quella sezione anche se sarebbe meglio usare un commento). In tal caso, è possibile utilizzare l'istruzione pass, che non esegue alcuna azione ma funge da promemoria.

```
if x > 0 :
print('uso x per i miei calcoli')
```

```
if x < 0:
    # Bisogna gestire i valori negativi
    pass
```

Quando si utilizza l'interprete Python, è necessario lasciare una riga vuota alla fine di un blocco, altrimenti Python restituirà un errore di sintassi simile al seguente:

```
>>> if 10 > 0:
... print('a')
File "<stdin>", line 2
print('a')
^
IndentationError: expected an indented block
```

Come avrai notato, dopo aver digitato l’istruzione condizionale, il prompt di Python è cambiato da >>> a … indicando che si aspetta un’istruzione indentata perché siamo nel corpo di un’istruzione condizionale.

if…else

Una seconda forma dell'istruzione if è l'esecuzione alternativa, in cui esistono due possibilità e la condizione determina quali azioni eseguire. La sintassi è simile alla seguente:

```
if x > 0 :
print('x è maggiore di 0')
else :
print('x è minore o uguale a 0')
```

In questo caso abbiamo ripreso l'esempio precedente applicando una condizione che, se verificata, stampa un messaggio che informa che x è maggiore di 0, in caso contrario informa che x è minore o uguale a 0.

Poiché la condizione deve essere vera o falsa, verrà eseguita necessariamente una delle istruzioni. Le alternative sono chiamate **rami** (o **branch**), perché sono diramazioni del flusso di esecuzione.

if…elif…else

A volte ci sono più di due possibilità e abbiamo bisogno di più di due rami. Assumiamo di voler specializzare meglio il nostro esempio: vogliamo sapere quando x è uguale a 0 perché non basta sapere che è minore o uguale a 0.

```
if x < 0:
print('x è minore di 0')
elif x > 0:
print('x è maggiore di 0')
else:
print('x è uguale a 0')
```

elif è un'abbreviazione di "else if", che viene spesso usato in molti altri linguaggi. Ancora una volta, verrà eseguito esattamente un ramo e ricorda che non c'è limite al numero di dichiarazioni elif. Se esiste una clausola else, deve essere posta alla fine, ma non è necessario che ci sia.

Ogni condizione è verificata nell'ordine descritto. Se la prima è falsa, viene verificata la successiva e così via. Se una delle condizioni è vera, viene eseguito il ramo corrispondente e l'istruzione termina. Attenzione, se più di una condizione è vera, sarà eseguito solo il primo ramo vero; se vuoi eseguire anche gli altri usa solo if.

Eccezioni

In precedenza, abbiamo visto un segmento di codice in cui abbiamo utilizzato le funzioni input e int() per leggere e analizzare un numero intero inserito dall'utente. Abbiamo anche visto quanto potrebbe essere pericoloso farlo perché si può generare un errore ed è fondamentale progettare il proprio codice per gestire gli errori e le eccezioni.

```
>>> domanda = 'Quanti anni hai?\n'
>>> anni = input(domanda)
Quanti anni hai?
test
>>> int(anni)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'test'
```

Quando eseguiamo queste istruzioni nell'interprete Python, riceviamo un nuovo prompt dall'interprete con un **traceback** ovvero un errore. Se si inserisce questo codice in uno script Python e si verifica questo errore, lo script si interrompe immediatamente e non esegue l'istruzione seguente. Per far in modo che il nostro programma gestisca questa situazione possiamo usare il costrutto try…except.

Questa struttura di esecuzione condizionale è incorporata in Python ed è stata creata per gestire tipi di errori previsti o imprevisti. L'idea è di "provare" il codice sapendo che alcune sequenze di istruzioni potrebbero generare un errore quindi si desidera aggiungere alcune istruzioni da eseguire nel caso in cui si verifichi l'errore. Queste istruzioni extra (blocco except) vengono ignorate se non si verificano errori. Puoi pensare a questo costrutto in Python come ad una "polizza assicurativa" su una sequenza di dichiarazioni. Riscriviamo il codice precedente:

```
>>> domanda = 'Quanti anni hai?\n'
>>> anni = input(domanda)
Quanti anni hai?
test
```

```
>>> try:
... int(anni)
... except:
... print('Hai inserito un valore non valido')
...
Hai inserito un valore non valido
```

Python inizia eseguendo la sequenza di istruzioni nel blocco try e se tutto va bene, salta il blocco except e procede. Se si verifica un'eccezione nel blocco try, Python salta fuori da questo blocco ed esegue la sequenza di istruzioni nel blocco except.

Quando si gestisce un'eccezione con un'istruzione try si parla di **catturare un'eccezione**. In questo esempio, la clausola except stampa un messaggio di errore ma in generale, la cattura di un'eccezione ti dà la possibilità di risolvere il problema, o riprovare, o almeno, terminare il programma senza errori.

FUNZIONI

Nel contesto della programmazione, una **funzione** è una sequenza di istruzioni che esegue un calcolo. Quando si definisce una funzione, si specifica il nome e la sequenza di istruzioni. Successivamente, è possibile "chiamare" la funzione per nome per rieseguire quei calcoli. Abbiamo già visto un esempio di una chiamata ad una funzione:

```
>>> type(32)
<class 'int'>
```

Il nome della funzione è type mentre l'espressione tra parentesi si chiama **argomento** della funzione. L'argomento è un valore o una variabile che passiamo alla funzione come input per la funzione stessa. Il risultato, per la funzione type, è il tipo dell'argomento. È comune affermare che una funzione "accetta" un argomento e "restituisce" un risultato, anche detto **valore di ritorno** (o valore restituito).

Python fornisce una serie di importanti funzioni integrate che possiamo usare senza la necessità di incorporare librerie o ricrearle. I creatori di Python hanno scritto una serie di funzioni per risolvere i problemi più comuni e le hanno incluse in Python per consentirci di utilizzarle. Le funzioni max e min, ad esempio, ci restituiscono rispettivamente i valori più grandi e più piccoli in un elenco o stringa:

```
>>> max('abcdefgh')
'h'
>>> min('abcdefgh')
```

'a'

La funzione max indica il "carattere più grande" nella stringa (che risulta essere la lettera h) e la funzione min ci mostra il carattere più piccolo (che risulta essere la lettera a). Un'altra funzione incorporata molto utile è la funzione len che ci dice quanti elementi ci sono nel suo argomento. Se l'argomento di len è una stringa, restituisce il numero di caratteri nella stringa.

```
>>> len('prova')
5
```

Queste funzioni non si limitano a lavorare con le stringhe infatti possono operare su qualsiasi insieme di valori. Dovresti considerare i nomi delle funzioni integrate come parole riservate (evita di usare max come nome di una variabile).

Ma è possibile convertire un valore da un tipo all'altro? Python fornisce anche funzioni integrate che convertono i valori da un tipo ad un altro. La funzione int() accetta qualsiasi valore e lo converte in un numero intero se possibile, in alternativa, restituisce un messaggio d'errore:

```
>>> int('32')
32
>>> int('Prova')
ValueError: invalid literal for int() with base 10: 'Prova'
```

Allo stesso modo puoi usare float() e str() rispettivamente per convertire un valore in un numero a virgola mobile o per ottenere una stringa da un valore numerico.

```
>>> float(32)
32.0
>>> str(32)
'32'
```

Dati gli stessi input, la maggior parte dei programmi per computer genera gli stessi output ogni volta, quindi si dice che sono **deterministici**. Il determinismo è di solito una buona cosa poiché prevediamo che lo stesso calcolo produca sempre lo stesso risultato. Per alcune applicazioni, tuttavia, vogliamo che il computer sia imprevedibile. I giochi sono un esempio ovvio, ma ce ne sono altri come la crittografia.

Rendere un programma veramente non deterministico non è così facile ma esistono dei modi molto utili. Uno di questi consiste nell'utilizzare algoritmi che generano numeri pseudocasuali. I numeri pseudocasuali non sono realmente casuali perché sono generati da un calcolo deterministico ma osservando tali numeri è quasi impossibile distinguerli da quelli casuali. Il modulo random fornisce funzioni che generano numeri pseudocasuali. La funzione random restituisce un float casuale tra 0,0 e 1,0 (incluso 0,0 ma non 1,0). Ogni volta che chiami questa funzione ottieni il numero successivo in una lunga serie. Per vedere un esempio, esegui questo ciclo:

```
>>> import random
>>>
>>> for i in range(10):
... x = random.random()
... print(x)
...
0.052106790620197296
0.688074761676638
0.1162906504127218
0.29023807242781197
0.6002408949875643
0.4784974114988294
0.682726182433228
0.5958148653304466
0.37720316786047214
0.10726917045489204
```

Definire una funzione

Finora abbiamo utilizzato solo le funzioni fornite con Python ma è anche possibile aggiungere nuove funzioni. Una definizione di funzione specifica il nome di una nuova funzione e la sequenza di istruzioni che vengono eseguite quando viene chiamata la funzione. Una volta definita una funzione, possiamo riutilizzarla più volte nel corso del nostro programma, evitando di riscrivere lo stesso codice.

```
>>> nome = 'Antonio'
>>> def salutaUtente(nome):
... print('Benvenuto ' + nome)
...
>>> salutaUtente(nome)
Benvenuto Antonio
```

Come avrai capito, def è una parola chiave che indica una definizione di funzione e il nome della funzione è salutaUtente. Le regole per i nomi delle funzioni sono le stesse dei nomi delle variabili: sono consentite lettere, numeri e alcuni segni di punteggiatura ma il primo carattere non può essere un numero. Non puoi utilizzare una parola chiave come nome di una funzione e dovresti evitare di avere una variabile e una funzione con lo stesso nome.

Le parentesi dopo il nome della funzione servono ad indicare gli argomenti della funzione, se sono vuote indicano che la funzione non accetta alcun argomento. In questo caso la funzione accetta un parametro in ingresso ovvero il nome dell'utente da salutare.

La prima riga della definizione della funzione si chiama **intestazione**; il resto si chiama **corpo**. L'intestazione deve terminare con due punti e il corpo deve essere indentato. Per convenzione, l'indentazione è sempre di quattro spazi. Il corpo può contenere qualsiasi numero di istruzioni infatti se digiti una definizione di funzione in modalità interattiva, l'interprete stampa i puntini di sospensione (...) per informarti che la definizione non è completa. Per terminare la funzione, devi inserire una riga vuota (ciò non è necessario in uno script).

Un aspetto peculiare di Python è che la definizione di una funzione crea una variabile con lo stesso nome e la sintassi per chiamare (o invocare) una funzione da noi creata è la stessa usata per le funzioni incorporate in Python. Dopo aver definito una funzione, è possibile utilizzarla all'interno di un'altra funzione. Ad esempio, potremmo scrivere una funzione per dare il benvenuto all'utente e inviare un codice di verifica per l'accesso:

```
>>> def salutaUtente(nome):
... print('Benvenuto ' + nome)
...
>>> def accessoSicuro(nome):
... salutaUtente(nome)
... print('Pin inviato')
...
>>> accessoSicuro(nome)
Benvenuto Antonio
Pin inviato
```

Le definizioni delle funzioni vengono eseguite esattamente come le altre istruzioni ma l'effetto è quello di creare oggetti funzione. Le istruzioni all'interno della funzione non vengono eseguite finché la funzione non viene chiamata e la definizione della funzione non genera alcun output. Come ci si potrebbe aspettare, è necessario creare una funzione prima di poterla eseguire, in altre parole, la definizione della funzione deve essere eseguita prima della prima chiamata.

Al fine di garantire che una funzione sia definita prima del suo primo utilizzo, è necessario conoscere l'ordine in cui vengono eseguite le istruzioni, tale ordine è chiamato **flusso di esecuzione**. L'esecuzione inizia sempre alla prima istruzione del programma e le dichiarazioni vengono eseguite una alla volta, in ordine dall'alto verso il basso. Le definizioni delle funzioni non alterano il flusso di esecuzione del programma, ma ricordano che le istruzioni all'interno della funzione non vengono eseguite fino a quando non viene chiamata la funzione.

Una chiamata di funzione è come una deviazione nel flusso di esecuzione infatti al posto di passare all'istruzione successiva, il flusso salta al corpo della funzione, esegue tutte le istruzioni presenti nel corpo della funzione e quindi riprende da dove era stato interrotto. Sembra abbastanza

semplice, fino a quando non ricordi che una funzione può chiamarne un'altra.

Nel mezzo di una funzione, il programma potrebbe dover eseguire le istruzioni in un'altra funzione creando così un ciclo complesso. Fortunatamente, Python è bravo a tenere traccia di dove si trova, quindi ogni volta che una funzione viene completata, il programma riprende da dove era stato interrotto nella funzione che lo chiamava. Quando raggiunge la fine del programma, il programma termina. Alla luce di ciò, quando leggi un programma, non limitarti a leggerlo dall'alto verso il basso, a volte ha più senso seguire il flusso dell'esecuzione.

Alcune delle funzioni integrate, così come la funzione appena creata richiedono argomenti. Alcune funzioni accettano anche più di un argomento e all'interno della funzione, tali argomenti sono assegnati a variabili chiamate **parametri**.

I parametri vengono valutati prima che la funzione venga chiamata e bisogna chiarire un aspetto molto importante che può causare confusione, soprattutto per i principianti. Il nome della variabile che passiamo come argomento (nome) non ha nulla a che fare con il nome del parametro (nome). Non importa come è stato denominato il valore nel chiamante; nella funzione il parametro può avere un nome diverso. Modifichiamo l'esempio per rendere il concetto più chiaro:

```
>>> nome = 'Antonio'
>>> def salutaUtente(nome):
... print('Benvenuto ' + nome)
...
>>> def accessoSicuro(nome_utente):
... salutaUtente(nome_utente)
... print('Pin inviato')
...
>>> accessoSicuro(nome)
Benvenuto Antonio
Pin inviato
```

Come vedi abbiamo definito una variabile nome che passiamo in input alla funzione accessoSicuro(). Il parametro con cui è stata definita

accessoSicuro() è nome_utente ma non è fondamentale passare in input una variabile con lo stesso nome.

Potrebbe non essere chiaro il motivo per cui valga la pena dividere un programma in funzioni. Esistono, in realtà, diversi motivi:

- La creazione di una nuova funzione offre l'opportunità d’identificare un gruppo di istruzioni, facilitando la lettura, la comprensione e il debug (identificazione di errori) del programma;
- Le funzioni possono ridurre la lunghezza di un programma eliminando il codice ripetuto. Se hai delle funzioni e apporti una modifica, devi solo apportarla in un unico punto;
- Dividere un lungo programma in funzioni consente di eseguire il debug delle parti una alla volta e quindi assemblarle in un unico insieme funzionante;
- Le funzioni ben progettate sono spesso utili per molti programmi. Dopo averle scritte ed aver eseguito il debug, è possibile riutilizzarle.

Consigliamo fortemente l’uso delle funzioni e, nonostante tu possa essere un po’ restio al loro uso, ben presto ti accorgerai di quanto tempo possono farti risparmiare. Assicurati di creare funzioni con un nome adatto al loro compito e, soprattutto, che eseguano solo un compito in modo da essere atomiche e da non dover incorporare più attività al loro interno.

ITERAZIONI

I computer sono spesso utilizzati per automatizzare attività ripetitive senza fare errori, questo è un compito che, di solito, i computer svolgono bene mentre le persone svolgono male. Poiché l'iterazione è così comune, Python offre diverse funzionalità linguistiche per renderla più semplice. Una forma di iterazione in Python è l'istruzione while. Ecco un semplice programma per dare il “Ciak, si gira” dopo aver effettuato un conto alla rovescia.

```
n = 3
while n > 0:
print(n)
n = n - 1
print('Ciak, si gira!')
```

Puoi quasi leggere l'istruzione while come se fosse inglese infatti significa "Fino a quando n è maggiore di 0, visualizza il valore di n e quindi riduci il valore di n di una unità. Quando arrivi a 0, esci dall'istruzione while e visualizza la frase “Ciak, si gira!”.

Più formalmente, ecco il flusso di esecuzione per un'istruzione while:

1. Valuta la condizione, producendo vero o falso;
2. Se la condizione è falsa, esci dall'istruzione while e continua l'esecuzione con l'istruzione successiva;

3. Se la condizione è vera, esegui il corpo del ciclo e poi torna al punto 1.

Questo tipo di flusso è chiamato **loop** (o ciclo) perché il terzo punto riporta ad uno dei precedenti e ogni volta che eseguiamo il corpo del ciclo è detta **iterazione**. Per il ciclo sopra, diremmo, "Aveva tre iterazioni", il che significa che il corpo del ciclo è stato eseguito tre volte.

Il corpo del ciclo dovrebbe cambiare il valore di una o più variabili in modo che alla fine la condizione diventi falsa e il ciclo possa terminare. La variabile che viene modificata ogni volta che il ciclo viene eseguito e che controlla la terminazione del ciclo è detta **variabile di iterazione**. Se non è presente alcuna variabile di iterazione, il ciclo si ripeterà per sempre, generando un ciclo infinito.

Nel caso del conto alla rovescia, possiamo dimostrare che il ciclo termina perché sappiamo che il valore di n è finito e possiamo vedere che il valore di n si riduce ogni volta attraverso il ciclo, quindi alla fine dobbiamo arrivare a 0.

A volte un ciclo è infinito perché non ha alcuna variabile di iterazione, altre volte non sai che è il momento di terminare un ciclo fino a quando non arrivi a metà del corpo del ciclo stesso. In tal caso è possibile scrivere un loop infinito di proposito e quindi utilizzare l'istruzione break per uscire dal ciclo. Creiamo un ciclo infinito da cui non è possibile uscire:

```
n = 10
while True:
print(n, end=' ')
n = n - 1
print('Ciclo completato')
```

Se commetti un errore simile o esegui questo codice, imparerai subito come fermare un processo Python sul tuo sistema o ben presto dovrai spegnere il tuo computer. Questo programma funzionerà per sempre o fino a quando si esaurirà la batteria del tuo laptop perché l'espressione logica nella parte superiore del ciclo è sempre vera in virtù del fatto che l'espressione è il valore costante True. Sebbene si tratti di un ciclo infinito, possiamo

comunque utilizzare questo modello per creare cicli utili. Forse ti starai chiedendo: ma perché usare True come condizione di un ciclo?

In realtà non è del tutto errato, infatti, questa è una pratica usata nella programmazione client-server. Se hai mai creato un server che accetta delle richieste in input dai client, probabilmente hai già usato un ciclo infinito per far in modo che il server resti in ascolto su una porta in attesa che un client invii una richiesta.

Ovviamente questo non può durare per sempre quindi è fondamentale saper uscire dal ciclo piuttosto che spegnere il server in modo brutale.

Ciclo for

A volte vogliamo iterare su un insieme di elementi come un elenco di parole, le righe di un file o un elenco di numeri. Quando abbiamo un elenco di elementi definito su cui iterare, possiamo costruire un ciclo usando un'istruzione for.

L'istruzione while costituisce un ciclo indefinito perché itera semplicemente fino a quando una condizione non diventa falsa, mentre il ciclo for itera un insieme di elementi già noti in modo da attraversare tutti gli elementi nell'insieme. La sintassi di un ciclo for è simile al ciclo while in quanto esiste un'intestazione e un corpo del ciclo:

```
utenti = ['Antonio', 'Filippo', 'Marco']
for utente in utenti:
print('Buongiorno: ', utente)
print('Buongiorno a tutti!')
```

La variabile utenti, per Python, è un elenco di tre stringhe e il ciclo for itera attraverso l'elenco ed esegue il corpo del ciclo una volta per ciascuna delle tre stringhe nell'elenco fornendo questo output:

```
Buongiorno: Antonio
Buongiorno: Filippo
Buongiorno: Marco
Buongiorno a tutti!
```

Tradurre questo for non è diretto come nel caso di while, ma se pensi agli utenti come un insieme: “Esegui le istruzioni nel corpo del ciclo for una volta per ogni utente nell’insieme chiamato utenti.” Le parole chiave for e in sono riservate in Python mentre utente e utenti sono variabili create da noi.

In particolare, utente è la variabile di iterazione per il ciclo for. Tale variabile cambia per ogni iterazione del ciclo e controlla quando il ciclo è completato. La variabile di iterazione attraversa in successione le tre stringhe memorizzate nella variabile utenti.

Spesso utilizziamo un ciclo for o while per passare in rassegna un elenco di elementi o il contenuto di un file e stiamo cercando qualcosa come il valore più grande o più piccolo nei dati attraverso cui eseguiamo la scansione. Questi cicli sono generalmente costituiti da:

- Inizializzazione di una o più variabili prima dell'inizio del ciclo;
- Esecuzione di un calcolo su ciascun elemento nel corpo del ciclo, possibilmente modificando le variabili nel corpo del ciclo;
- Valutazione delle variabili risultanti al completamento del ciclo.

LISTE

Come una stringa, una lista (o elenco) è una sequenza di valori. In una stringa, i valori sono caratteri; in una lista, i valori possono essere di qualsiasi tipo e tali valori sono chiamati **elementi**. Esistono diversi modi per creare una nuova lista; il più semplice è racchiudere gli elementi tra parentesi quadre:

[234**,** 212**,** 200**,** 123**]**
['Antonio'**,** 'Filippo'**,** 'Marco'**]**

Il primo esempio è una lista di quattro numeri interi mentre il secondo è una lista di tre stringhe. Gli elementi di una lista, tuttavia, possono non essere dello stesso tipo. Il seguente esempio mostra una lista che contiene una stringa, un float, un numero intero e un'altra lista:

['prova'**,** 6.3**,** 100**, [**10**,** 20**]]**

Abbiamo creato una lista con una lista al suo interno, tale lista avrebbe potuto contenere elementi oppure essere vuota. Una lista che non contiene elementi è detta **lista vuota**; puoi crearne una con parentesi vuote [].

La sintassi per accedere agli elementi di una lista è la stessa per accedere ai caratteri di una stringa ovvero l'uso dell'operatore parentesi quadrata. L'espressione tra parentesi specifica l'indice dell'elemento che vogliamo selezionare. Ricorda che gli indici iniziano da 0:

>>> utenti = **[**'Antonio'**,** 'Filippo'**,** 'Marco'**]**

```
>>> print(utenti[0])
Antonio
```

A differenza delle stringhe, le liste sono **mutabili** perché è possibile modificare l'ordine degli elementi all'interno della stessa lista o riassegnare un elemento in una lista. In questo caso vogliamo modificare l'ultimo elemento della lista di utenti sostituendolo con Giovanni:

```
>>> utenti[2] = 'Giovanni'
>>> print(utenti)
['Antonio', 'Filippo', 'Giovanni']
```

Puoi pensare ad una lista come una relazione tra indici ed elementi. Questa relazione è chiamata **mappatura**; ogni indice "mappa" uno degli elementi. Gli indici di una lista funzionano allo stesso modo degli indici di una stringa:

- Qualsiasi espressione intera può essere utilizzata come indice;
- Se si tenta di leggere o scrivere un elemento che non esiste, si ottiene un IndexError;
- Se un indice ha un valore negativo, si recuperano gli elementi a partire dalla fine della lista.

L'operatore in funziona anche nelle liste come puoi notare dall'esempio:

```
>>> utenti = ['Antonio', 'Filippo', 'Marco']
>>> 'Antonio' in utenti
True
```

```
>>> 'Mirko' in utenti
False
```

Operatori e metodi utili

Il modo più comune per scorrere gli elementi di una lista è con un ciclo for. La sintassi è la stessa delle stringhe:

```
for utente in utenti:
print(utente)
```

Questo approccio funziona bene se devi solo leggere gli elementi della lista ma se vuoi scrivere o aggiornare gli elementi, hai bisogno degli indici. Un modo comune per ovviare a questo problema è quello di combinare la funzione len con range:

```
numeri = [10, 25, 66]
for i in range(len(numberi)):
numberi[i] = numberi[i] * 2
```

Questo ciclo attraversa la lista e aggiorna ogni elemento. La funzione len restituisce il numero di elementi nella lista mentre range restituisce un elenco di indici da 0 a n-1, dove n è la lunghezza della lista. Ogni volta, attraverso il ciclo, otteniamo l'indice dell'elemento successivo. L'istruzione di assegnazione nel corpo del ciclo utilizza i per leggere il vecchio valore dell'elemento e per assegnare il nuovo valore.

Con le liste si può fare davvero di tutto, ad esempio, si possono concatenare con l'operatore +, si possono ripetere elementi con l'operatore * o addirittura "tagliare" tramite l'operatore due punti (:).

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

```
>>> [0] * 4
```

```
[0, 0, 0, 0]
>>> [1, 2, 3] * 2
[1, 2, 3, 1, 2, 3]
```

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Python fornisce anche dei metodi che operano sulle liste. Ad esempio, append() aggiunge un nuovo elemento alla fine di una lista:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

Il metodo extend(), invece, prende in input una lista e ne aggiunge tutti gli elementi alla lista su cui è stato invocato:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

Un altro metodo molto usato è sort() che consente di ordinare gli elementi in modo crescente quindi dal più piccolo al più grande:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Così come è possibile aggiungere gli elementi ad una lista è possibile rimuoverli con i metodi pop(), del e remove().

Se conosci l'indice dell'elemento da rimuovere puoi usare pop() che modifica la lista e restituisce l'elemento rimosso. Se non hai la necessità di avere l'ultimo elemento rimosso puoi usare del, mentre se conosci il valore da eliminare ma non la sua posizione, puoi usare remove(). Vediamoli in azione:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

Esistono diverse funzioni integrate che possono essere utilizzate nelle liste e che consentono di consultare rapidamente una lista senza riscrivere un

ciclo:

```
>>> numeri = [3, 41, 12, 9, 74, 15]
>>> print(len(numeri))
6
>>> print(max(numeri))
74
>>> print(min(numeri))
3
>>> print(sum(numeri))
154
>>> print(sum(numeri)/len(numeri))
25
```

La funzione sum() funziona solo quando gli elementi della lista sono numeri mentre le altre funzioni (max(), len(), ecc.) funzionano anche con liste di stringhe e altri tipi che possono essere comparabili.

7

DIZIONARI

Un dizionario è come una lista, ma più generale. In una lista, le posizioni dell'indice devono essere numeri interi mentre in un dizionario, gli indici possono essere (quasi) di qualsiasi tipo. Puoi pensare a un dizionario come una mappatura tra un insieme di indici (che sono chiamati **chiavi**) e un insieme di valori dove ogni chiave è mappata su un valore.

L'associazione di una chiave e un valore viene detta **coppia chiave-valore** o talvolta elemento. Ad esempio, creeremo un dizionario che mappa dei numeri inglesi a quelli italiani, quindi le chiavi e i valori saranno tutte stringhe. La funzione dict crea un nuovo dizionario senza elementi e poiché dict è il nome di una funzione integrata, è consigliabile evitare di utilizzarlo come nome di variabile.

```
>>> engToit = dict()
>>> print(engToit)
{}
```

In questo modo abbiamo creato un dizionario vuoto infatti le parentesi graffe senza alcun valore all'interno indicano proprio un dizionario senza alcuna associazione. Possiamo aggiungere degli elementi come segue:

```
>>> engToit['one'] = 'uno'
>>> print(engToit)
{'one': 'uno'}
```

Abbiamo creato un elemento che viene mappato dalla chiave one con il valore uno. Stampando nuovamente il dizionario, vediamo una coppia chiave-valore con due punti che separano la chiave dal valore.

Proviamo ad aggiungere più coppie al dizionario:

```
>>> engToit = {'one': 'uno', 'two': 'due', 'three': 'tre'}
>>> print(engToit)
{'one': 'uno', 'three': 'tre', 'two': 'due'}
```

L'ordine delle coppie chiave-valore, come avrai notato, non è lo stesso di quando le abbiamo inserite. In effetti, se digiti lo stesso esempio sul tuo computer, potresti ottenere un risultato diverso. In generale, l'ordine degli elementi in un dizionario è imprevedibile ma questo non è un problema perché gli elementi di un dizionario non vengono mai indicizzati con indici interi. Si accede ad ogni elemento utilizzando la chiave pertanto:

```
>>> print(engToit['two'])
'due'
```

Se proviamo a far riferimento ad un elemento non esistente all'interno del dizionario otterremo un KeyError che potremo gestire con un blocco try…except.

Proprio come per le liste possiamo usare le funzioni len() per ottenere il numero di elementi inseriti nel dizionario e la parola chiave in per verificare se un elemento è presente nel dizionario.

```
>>> len(engToit)
3
>>> 'one' in engToit
True
>>> 'uno' in engToit
False
```

Un classico uso dei dizionari consiste nel contare quante volte una parola o una lettera si ripete all'interno di un testo. In realtà esistono diversi modi per farlo:

1. È possibile creare 26 variabili, una per ogni lettera dell'alfabeto. Potresti iterare il testo e, per ogni carattere, incrementare il contatore corrispondente;
2. È possibile creare una lista con 26 elementi. Quindi è possibile convertire ciascun carattere in un numero (utilizzando la funzione incorporata ord()), utilizzando il numero come indice nell'elenco e incrementando il contatore appropriato;
3. È possibile creare un dizionario con caratteri come chiavi e contatori come valori corrispondenti. La prima volta che incontri un carattere, aggiungi un oggetto al dizionario. Se hai già incontrato quel carattere, aumenti il valore per la chiave corrispondente.

Ognuna di queste opzioni esegue lo stesso calcolo, ma ognuna implementa tale calcolo in modo diverso. Un'**implementazione** è un modo per eseguire un calcolo ma alcune implementazioni sono migliori di altre. Ad esempio, un vantaggio dell'implementazione del dizionario è che non dobbiamo sapere in anticipo quali lettere compaiono nella stringa e non andremo a cercare tra tutte le lettere ma useremo solo quelle che effettivamente compaiono nel testo. Ecco come potrebbe essere il codice che sfrutta i dizionari:

testo = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam ac elit eget augue tempus lacinia. Quisque mattis odio tincidunt rutrum placerat'

d = **dict()**

```
for c in testo:
    if c not in d:
    d[c] = 1
    else:
    d[c] = d[c] + 1

print(d)
```

Stiamo effettivamente calcolando un istogramma, che è un termine statistico per un insieme di contatori (o frequenze). Il ciclo for itera la stringa e ogni volta che si attraversa il ciclo, se il carattere c non è nel dizionario, creiamo un nuovo elemento con chiave c e valore iniziale 1 (poiché abbiamo visto questo carattere una sola volta). Se c è già nel dizionario incrementiamo d[c].

8

INSIEMI

Gli insiemi (anche detti set) incarnano un concetto matematico che viene spesso utilizzato nella programmazione in cui è richiesto un gruppo univoco di elementi. In Python, gli insiemi sono molto simili ai dizionari con chiavi ma senza valori corrispondenti. I set hanno il tipo Python **set**. Si applicano le stesse regole di base delle chiavi del dizionario in quanto i valori degli insiemi devono essere immutabili e univoci (infatti è questo che lo rende un insieme!).

Il valore predefinito di set() è l'insieme vuoto e questo è l'unico modo per esprimere un insieme vuoto perché {} è già utilizzato per un dizionario vuoto. La funzione set() accetta qualsiasi tipo di collezione come argomento e la converte in un insieme (i valori del dizionario andranno persi). C'è un altro tipo di set in Python, chiamato **frozenset**, che è immutabile ed è fondamentalmente un set di sola lettura. Il suo costruttore funziona come set(), ma supporta solo un sottoinsieme delle operazioni. Puoi usare un frozenset come elemento di un set standard perché frozenset è immutabile. I valori letterali impostati sono espressi con parentesi graffe che circondano gli elementi separati da virgole:

```
myset = {1,2,3,4,5}
```

I set non supportano l'indicizzazione o lo slicing e, come i dizionari, non hanno alcun ordine intrinseco e la funzione sorted() restituisce un elenco ordinato di valori.

Gli insiemi hanno un sacco di operazioni sugli insiemi in stile matematico che sono distinte dalle altre raccolte. La tabella seguente mostra le operazioni comuni ai tipi set e frozenset:

Operazione

Descrizione

in

Questo verifica se esiste un singolo elemento nel set. Se l'elemento testato è esso stesso un insieme, S1, il risultato sarà vero solo se S1, come insieme, è un elemento dell'insieme di obiettivi.

issubset, <=, <

Questi verificano se un insieme è un sottoinsieme della destinazione, ovvero se tutti i membri dell'insieme testato sono anche membri della destinazione. Se gli insiemi sono identici, il test restituirà True nei primi due casi, ma False per l'operatore <.

issuperset, >=, >

Questi verificano se un set è un superset dell'altro, ovvero se tutti i membri del set di destinazione sono membri dell'origine. Se sono uguali, le prime due operazioni restituiscono True; l'ultima operazione restituirà False.

union, |

Questi restituiscono l'unione di due (o più) insiemi. Il metodo union accetta un elenco di insiemi separati da virgole come argomenti mentre l'operatore pipe viene utilizzato in modo fisso tra gli insiemi.

intersection, &

Questi restituiscono l'insieme di intersezione di due (o più) insiemi. L'utilizzo è simile a union().

difference, -

Questi restituiscono i membri del set di origine che non sono membri del set di destinazione.

symmetric_difference, ^

Restituisce gli elementi di entrambi gli insiemi che non si trovano nell'intersezione. Il metodo funziona solo per due insiemi, ma l'operatore ^ può essere applicato a più insiemi.

Nota bene che le variazioni del metodo nella tabella accetteranno qualsiasi tipo di collezione come argomento mentre le operazioni infisse

funzioneranno solo con gli insiemi.

PROGRAMMY PYTHON

I programmi Python non hanno alcun punto di ingresso predefinito richiesto (ad esempio una funzione main()) e sono semplicemente espressi come codice sorgente in un file di testo che viene letto ed eseguito in ordine a partire dall'inizio del file. Le definizioni, come le funzioni, vengono eseguite nel senso che la funzione viene creata e assegnata a un nome, ma il codice interno non viene mai eseguito finché la funzione non viene invocata. Python non ha alcuna sintassi speciale per indicare se una sorgente file è un programma o un modulo e, come vedrai, un dato file può essere utilizzato in entrambi i ruoli. Un tipico file di programma eseguibile consiste in un insieme di istruzioni import per inserire tutti i moduli di codice necessari, alcune definizioni di funzioni e classi e del codice direttamente eseguibile. In pratica, per un programma non banale, la maggior parte delle definizioni di funzioni e classi esisterà nei file di modulo e sarà inclusa nelle import. Ciò lascia una breve sezione del codice del driver per avviare l'applicazione. Spesso questo codice verrà inserito in una funzione e la funzione sarà spesso chiamata main(), ma questo è puramente un cenno alla convenzione di programmazione, non un requisito di Python.

Infine, è necessario chiamare questa funzione "principale". Questo viene spesso fatto all'interno di una speciale istruzione if alla fine dello script principale e si presenta così:

```
if __name__ == "__main__":
main()
```

Quando Python rileva che un file di programma viene eseguito dall'interprete anziché importato come modulo, imposta la variabile speciale __ name__ (nota i doppi caratteri di sottolineatura su entrambi i lati) su " __ main__ ". Ciò significa che tutto il codice all'interno di questo blocco if viene eseguito solo quando lo script viene eseguito come programma principale e non quando il file viene importato da un altro programma. Se ci si aspetta che il file venga usato solo come modulo, la funzione main() può essere sostituita da una funzione test() che esegue una serie di unit test. Anche in questo caso, il nome effettivo utilizzato non ha alcun significato per Python.

La struttura di programmazione più fondamentale è una sequenza di istruzioni. Normalmente le istruzioni Python si trovano su una riga da sole; quindi, una sequenza è semplicemente una serie di righe.

```
x = 2
y = 12
z = 135
```

In questo caso le dichiarazioni sono tutte assegnazioni. Altre affermazioni valide includono chiamate a funzioni, importazioni di moduli o definizioni. Le definizioni includono funzioni e classi. Valgono anche le diverse strutture di controllo descritte nelle sezioni seguenti.

Python è un linguaggio strutturato a blocchi e i blocchi di codice sono indicati dal livello di indentazione. La quantità di indentazione è abbastanza flessibile e, sebbene la maggior parte dei programmatori Python si attenga all'utilizzo di tre o quattro spazi per ottimizzare la leggibilità, per Python non fa alcuna differenza. Diversi ambienti di sviluppo integrato (detti IDE) ed editor di testo hanno le proprie idee su come viene eseguita l'indentazione. Se si utilizzano più strumenti di programmazione, è possibile che vengano segnalati errori di indentazione poiché gli strumenti hanno utilizzato diverse combinazioni di tabulazioni e spazi nel corso del tempo.

Se possibile, imposta il tuo editor per utilizzare gli spazi anziché . L'eccezione alla regola di indentazione sono i commenti. Un commento Python inizia con un simbolo # e continua fino alla fine della riga. Python accetta commenti che iniziano in qualsiasi punto della riga indipendentemente dal livello di indentazione corrente, ma per convenzione i programmatori tendono a mantenere il livello di indentazione, anche per i

commenti. Python supporta un insieme limitato di opzioni di selezione. La struttura più elementare è il costrutto if/elif/else che abbiamo già analizzato.

Spesso nei programmi è possibile incontrare degli errori, esistono due approcci per rilevare gli errori in Python. Il primo prevede il controllo esplicito di ogni azione mentre viene eseguita; l'altro tenta le operazioni e si affida al sistema che genera una condizione di errore, o un'eccezione, se qualcosa va storto. Sebbene il primo approccio sia appropriato in alcune situazioni, in Python è molto più comune usare il secondo. Python supporta questa tecnica con il costrutto try/except/else/finally. Nella sua forma generale, si presenta così:

try:
A block of application code
except <an error **type**> **as** <anExceptionObject>**:**
A block of error handling code
else:
Another block of application code
finally:
A block of clean-up code

Except, else e finally sono tutti facoltativi, sebbene almeno un except o finally debba esistere se viene utilizzata un'istruzione try. Possono esserci più clausole except, ma solo un else o finally. È possibile omettere la parte as... di una riga di istruzione except se i dettagli dell'eccezione non sono richiesti.

Il blocco try viene eseguito e, se si verifica un errore, viene testata la classe di eccezione. Se esiste un'istruzione except per quel tipo di errore, viene eseguito il blocco corrispondente. Se ci sono più blocchi di eccezioni che specificano lo stesso tipo di eccezione, viene eseguita solo la prima clausola di corrispondenza. Se non viene trovata alcuna istruzione except corrispondente, l'eccezione viene propagata verso l'alto finché non viene raggiunto l'interprete di livello superiore e Python genera il consueto rapporto di errore di traceback.

Si noti che un'istruzione except vuota rileverà qualsiasi tipo di errore; tuttavia, questa è generalmente una cattiva idea perché nasconde il verificarsi di eventuali errori imprevisti.

Il blocco else viene eseguito se il blocco try ha esito positivo senza errori, in pratica else è usato raramente. Indipendentemente dal fatto che un errore venga rilevato o propagato, o che venga eseguita la clausola else, la clausola finally verrà sempre eseguita, fornendo così l'opportunità di rilasciare qualsiasi risorsa di calcolo in uno stato bloccato. Questo è vero anche quando la clausola try/except viene collegata ad un'istruzione break o return.

È possibile utilizzare una singola istruzione di except per elaborare più tipi di eccezione. Puoi farlo elencando le classi di except in una tupla (le parentesi sono obbligatorie). L'oggetto eccezione contiene i dettagli su dove si è verificata l'eccezione e fornisce un metodo di conversione della stringa in modo che possa essere fornito un messaggio di errore significativo dalla stampa dell'oggetto. È possibile anche sollevare eccezioni dal proprio codice. È anche possibile utilizzare uno qualsiasi dei tipi di eccezione esistenti o definirne di propri creando una sottoclasse dalla classe Exception. Puoi anche passare argomenti alle eccezioni sollevate e puoi accedervi nell'oggetto eccezione nella clausola exclude usando l'attributo args dell'oggetto error. Ecco un esempio di generazione di un ValueError standard con un argomento personalizzato e quindi rilevare tale errore e stampare l'argomento specificato.

```
>>> try:
... raise ValueError('wrong value')
... except ValueError as error:
... print (error.args)
...
('wrong value',)
```

Nota che non hai ottenuto un traceback completo ma solo l'output di stampa dal blocco except. Puoi anche sollevare nuovamente l'eccezione originale dopo l'elaborazione semplicemente chiamando raise senza argomenti.

Infine, parliamo del concetto di contesto di runtime in Python. Questo in genere include una risorsa temporanea di qualche tipo con cui il tuo programma vuole interagire. Un tipico esempio potrebbe essere un file aperto o un thread di esecuzione simultaneo. Per gestire questo, Python usa la parola chiave with e un protocollo di gestione del contesto. Questo protocollo ti consente di definire le tue classi di gestione del contesto, ma

utilizzerai principalmente i gestori forniti da Python. Si utilizza un gestore di contesto invocando l'istruzione with:

```
with open(filename, mode) as contextName:
    process file here
```

Il gestore del contesto assicura che il file venga chiuso dopo l'uso. Questo è abbastanza tipico del ruolo di un gestore di contesto: garantire che le risorse vengano liberate dopo l'uso o che vengano prese adeguate precauzioni di condivisione al primo utilizzo. I gestori di contesto spesso eliminano la necessità di un costrutto try/finally. Il modulo contextlib fornisce supporto per la creazione dei propri gestori di contesto.

A questo punto hai approfondito i diversi tipi di dati che Python può elaborare così come le strutture di controllo che puoi usare per eseguire quelle elaborazioni. È giunto il momento di scoprire come ottenere dati da e per i tuoi programmi Python.

Input e output

L'input e l'output basilare dei dati è un requisito importante di qualsiasi linguaggio di programmazione. Devi considerare come i tuoi programmi interagiranno con gli utenti e con i dati memorizzati nei file. Per inviare dati agli utenti tramite stdout, usi la funzione print(), che hai già visto diverse volte. Imparerai come controllare l'output in modo più preciso in questa sezione.

Per leggere i dati dagli utenti, si utilizza la funzione input(), che richiede all'utente l'input e quindi restituisce una stringa di caratteri grezzi da stdin. La funzione print() è più complessa di quanto sembri a prima vista in quanto ha diversi parametri opzionali. Al suo livello più semplice, devi semplicemente passare una stringa e print() la visualizza su stdout seguito da un carattere di fine riga (eol). Il livello successivo di complessità implica il passaggio di dati non stringa che print() converte quindi in una stringa (usando la funzione di tipo str()) prima di visualizzare il risultato. Aumentando un po' la difficoltà, puoi passare più elementi contemporaneamente a print(), li convertirà e li visualizzerà separati da uno spazio. Ecco tre elementi fissi nel comportamento di print():

- Visualizza l'output su stdout;
- Termina con un carattere eol;
- Separa gli elementi con uno spazio.

In effetti, nessuno di questi è realmente corretto e print() ti consente di modificarli alcuni o tutti usando parametri opzionali. È possibile modificare l'output specificando un argomento file; il separatore è definito dall'argomento sep e il carattere di terminazione è definito dall'argomento end. La riga seguente stampa il famigerato messaggio "hello world", specificato come due stringhe, in un file usando un trattino come separatore e la stringa "END" come indicatore di fine:

```
with open("tmp.txt", "w") as tmp:
print("Hello", "World", end="END", sep="-", file=tmp)
```

Il risultato è: “Hello-WorldEND”.

Il metodo format() di string diventa davvero unico se combinato con print(). Questa combinazione è in grado di presentare i tuoi dati in modo ordinato e chiaramente separati l’uno dall’altro. Inoltre, l'uso di format() può spesso essere più efficiente rispetto a cercare di stampare un lungo elenco di oggetti e frammenti di stringhe. Ci sono molti esempi di come usare format() nella documentazione di Python.

È inoltre possibile comunicare con l'utente utilizzando la funzione input() che legge i valori digitati dall'utente in risposta a un determinato prompt sullo schermo. Ricorda che è una tua responsabilità convertire i caratteri restituiti in qualsiasi tipo di dati sia rappresentato e gestire eventuali errori risultanti da tale conversione. Ecco un esempio che chiede all'utente di inserire un numero. Se il numero è troppo alto o troppo basso, stampa un avviso:

```
target = 66
while True :
value = input("Enter an integer between 1 and 100")
try:
value = int(value)
break
except ValueError:
print("I said enter an integer!")
if value > target:
print (value, "is too high")
elif int(value) < target:
print("too low")
else:
print("Pefect")
```

Qui all'utente viene fornita una richiesta per inserire un numero intero nell'intervallo appropriato. Il valore letto viene quindi convertito in un numero intero utilizzando int(). Se la conversione non va a buon fine, viene generata un'eccezione ValueError e viene quindi visualizzato il messaggio di errore. Se la conversione ha esito positivo, è possibile interrompere il ciclo while e procedere a testarlo rispetto al target, certi di disporre di un numero intero valido.

I file di testo sono i cavalli di battaglia della programmazione quando si tratta di salvare i dati e Python supporta diverse funzioni per gestire i file di testo. Hai visto la funzione open() nelle sezioni precedenti che prende un nome file e una modalità come argomenti.

La modalità può essere una qualsiasi tra r, w, rw e a rispettivamente per lettura, scrittura, lettura-scrittura e aggiunta. Ci sono anche alcune altre modalità usate meno spesso così come ci sono alcuni parametri opzionali che controllano come vengono interpretati i dati, vedere la documentazione per i dettagli.

La modalità r richiede che il file esista; le modalità w e rw creano un nuovo file vuoto (o sovrascrivono qualsiasi file esistente con lo stesso nome). La modalità a apre un file esistente o crea un nuovo file vuoto se non esiste già un file con il nome file specificato. L'oggetto file restituito è anche un gestore del contesto e può essere utilizzato in un blocco with come hai visto nella sezione precedente. Se non viene usato un blocco with, dovresti chiudere esplicitamente il file usando il metodo close() quando hai finito di lavorarci, assicurandoti così che tutti i dati che si trovano nei buffer di memoria siano inviati al file fisico sul disco. Il costrutto with chiama automaticamente close(), che è uno dei vantaggi dell'utilizzo dell'approccio del gestore del contesto. Una volta che hai un oggetto file aperto, puoi usare read(), readlines() o readline() come richiesto. La funzione read() legge l'intero contenuto del file come una singola stringa completa di caratteri che indicano l’inizio di nuova riga, readlines() legge riga per riga in un elenco e i caratteri di nuova riga vengono mantenuti, readline() legge la riga successiva nel file, riservando la nuova riga. L'oggetto file è iterabile, quindi puoi usarlo direttamente in un ciclo for senza bisogno di nessuno dei metodi di lettura. Lo schema consigliato per leggere le righe in un file è quindi:

```
with open(filename, mode) as fileobject:
for line in fileobject:
# process line
```

È possibile scrivere su un oggetto file usando i metodi write() o writelines(), che sono gli equivalenti dei metodi read con nomi simili. Nota che non esiste un metodo writeline() per scrivere una singola riga. Se si utilizza la modalità rw, è possibile che ci si voglia spostare in una posizione specifica

nel file per sovrascrivere i dati esistenti. Puoi trovare la tua posizione attuale nel file usando il metodo tell() e puoi posizionarti in una posizione specifica (possibilmente una che hai registrato con tell() in precedenza) usando il metodo seek(). La funzione seek() ha diverse modalità di calcolo della posizione; l'impostazione predefinita è semplicemente l'offset dall'inizio del file.

Ora hai tutte le abilità di base per scrivere programmi Python funzionanti. Tuttavia, per affrontare progetti più grandi, avrai bisogno di estendere le tue capacità di Python.

10

ESTENDERE I PROGRAMMI

Il modo più semplice per estendere l'uso di Python è scrivere le proprie funzioni. Puoi definirle nello stesso file del codice che li utilizza, oppure puoi creare un nuovo modulo e importare le funzioni da esso. Per ora creerai le funzioni e le utilizzerai nello stesso file, in effetti, potresti usare principalmente il prompt interattivo per gli esempi in questa sezione. Il passaggio successivo nella creazione di nuove funzionalità in Python è definire le proprie classi e creare oggetti da esse. Gli esempi qui sono abbastanza semplici che puoi semplicemente usare il prompt di Python.

I programmatori Python usano spesso le stringhe di documentazione nei loro programmi. Le stringhe di documentazione sono stringhe letterali che non sono assegnate a una variabile e rispettano il livello di indentazione a cui sono definite. Le stringhe di documentazione vengono utilizzate per descrivere funzioni, classi o moduli. La funzione help() legge e presenta le stringhe di documentazione.

Funzioni

In Python sono disponibili diversi tipi di funzioni. Puoi definire le funzioni in Python usando la parola chiave def. Il modulo si presenta così:

```
def functionName(parameter1, param2,...):
function block
```

Le funzioni Python restituiscono sempre un valore. È possibile specificare un valore di ritorno esplicito utilizzando la parola chiave return; in caso contrario, Python restituisce None per impostazione predefinita. (Se trovi valori None imprevisti che appaiono nell'output, controlla che la funzione in questione abbia un'istruzione return esplicita nel suo corpo.) Puoi dare dei valori predefiniti ai parametri facendo seguire il nome da un segno di uguale e il valore.

In questo caso stiamo creando una nuova funzione che accetta diversi parametri di input e restituisce un valore. Questa funzione utilizza l'equazione matematica di una retta per restituire la coordinata y corrispondente per un determinato gradiente, coordinata x e costante, quindi utilizzeremo la funzione per generare un insieme di coordinate per una linea.

1. Avviare l'interprete Python.
2. Digitare il seguente codice per definire la funzione:

```
>>> def straight_line(gradient, x, constant):
... ''' returns y coordinate of a straight line
-> gradient * x + constant'''
... return gradient*x + constant
...
>>>
```

1. Ora che hai definito la funzione, provala usando alcuni semplici valori che puoi calcolare a mente. Prova a chiamare la funzione con un gradiente pari a 2, un valore x pari a 4 e una costante -3:

```
>>> # test with a single value first
    >>> straight_line(2,4,-3)
    5
```

1. Proviamo ora un test più complesso della funzione, utilizzando il seguente codice:

```
>>> for x in range(10):
    ... print(x,straight_line(2,x,-3))
    ...
    0 -3
    1 -1
    2 1
    3 3
    4 5
    5 7
    6 9
    7 11
    8 13
    9 15
```

1. Infine, verifica che la funzione help() riconosca correttamente la funzione:

```
>>> help(straight_line)
    Help on function straight_line in module __main__:
    straight_line(gradient, x, constant)
    returns y coordinate of a straight line
    -> gradient * x + constant
    (END)
```

Nella prima riga del passaggio 2 è stata creata la definizione della funzione. L'abbiamo chiamata straight_line e abbiamo dichiarato che aveva tre parametri richiesti: gradiente, x e costante. Questi parametri corrispondono ai valori utilizzati nell'equazione matematica y = mx+c, dove m è il gradiente e c è una costante. La seconda riga è una stringa di documentazione che descrive a cosa serve la funzione e come dovrebbe essere utilizzata. La terza riga è il blocco di codice funzione.

Potrebbe essere arbitrariamente complesso e lungo diverse righe, ma in questo caso è una riga e stai restituendo il risultato; quindi, gli hai aggiunto la parola chiave return. Nota che la riga di codice deve iniziare allo stesso livello di indentazione dell'inizio della stringa di documentazione; in caso contrario, riceverai un errore di indentazione. Hai quindi testato la funzione con alcuni valori semplici, alcuni calcoli mentali confermano che il valore di ritorno 5 è effettivamente uguale a (2*4-3). La funzione sembra funzionare, almeno per un caso semplice. È stata quindi utilizzata la funzione per generare un insieme di coppie di coordinate x,y utilizzando un ciclo for con un valore fisso per il gradiente (2) e una costante (-3) ma fornendo la variabile di ciclo come x. Se hai della carta a portata di mano, puoi provare a tracciare le coordinate elencate per confermare che formano una linea retta. Infine, hai utilizzato la funzione help() per confermare che la stringa della documentazione è stata rilevata e visualizzata correttamente.

Le funzioni generatore sono esattamente come le funzioni standard tranne per il fatto che invece di usare return per restituire un valore, usano la parola chiave yield. (In teoria possono usare return in aggiunta a yield, ma solo le espressioni yield producono il comportamento tipico del generatore). La parte interessante è che queste funzioni si comportano come una fotocamera a fermo immagine, quando una funzione standard raggiunge un'istruzione return, restituisce il valore e la funzione elimina tutti i suoi dati interni. La prossima volta che viene chiamata la funzione, tutto riparte da zero.

La dichiarazione yield fa qualcosa di diverso. Restituisce un valore, proprio come fa return, ma non fa sì che la funzione elimini i suoi dati; invece, tutti i dati vengono preservati. La successiva chiamata della funzione riprende dall'istruzione yield, anche se si trova nel mezzo di un blocco o parte di un ciclo. Possono esserci anche più istruzioni yield in una

singola funzione, poiché l'istruzione yield può trovarsi all'interno di un ciclo, è possibile creare una funzione che restituisca effettivamente una serie infinita di risultati. Ecco un esempio che restituisce una serie crescente di numeri dispari:

```
def odds(start=1):
'" return all odd numbers from start upwards'"
if int(start) % 2 == 0: start = int(start) + 1
while True:
yield start
start += 2
```

In questa funzione controlliamo prima che l'argomento di inizio sia un intero dispari (un intero pari diviso per 2 ha resto zero), e in caso contrario, lo forziamo ad essere il successivo intero dispari più alto aggiungendo 1 unità. Quindi creiamo un ciclo while infinito. Normalmente questa sarebbe una pessima idea perché il tuo programma si bloccherà per sempre, tuttavia, poiché questa è una funzione del generatore, stai usando yield per restituire il valore di inizio in modo che la funzione esca a questo punto, restituendo il valore di inizio in questo momento. La prossima volta che la funzione viene chiamata, l'esecuzione inizia esattamente da dove l'avevi interrotta. Quindi start viene incrementato di 2 unità e il ciclo girerà di nuovo, producendo il numero dispari successivo e uscendo dalla funzione fino alla volta successiva. Python assicura che le funzioni generatore diventino iteratori in modo da poterle utilizzare nei cicli for, in questo modo:

```
for n in odds():
if n > 7: break
else: print(n)
```

Usiamo odds() proprio come una collezione. Ogni volta che il loop viene eseguito, chiama la funzione generatore e riceve il valore dispari successivo. Evitiamo un ciclo infinito inserendo il test di interruzione in modo da non chiamare mai odds() oltre 7. Ora che hai capito come funzionano le funzioni generatore, potresti esserti reso conto che le espressioni del generatore introdotte in precedenza in questo capitolo sono

effettivamente solo funzioni generatore anonime. Le espressioni del generatore sono effettivamente una forma mascherata di una funzione generatore senza un nome. Questo fornisce un perfetto passaggio al tipo di funzione finale di cui impareremo qui: la funzione lambda.

Il termine lambda deriva da una forma di calcolo matematico inventata da Alonzo Church. La buona notizia è che non è necessario sapere nulla di matematica per utilizzare le funzioni lambda! L'idea alla base di una funzione lambda è che si tratta di un blocco funzione anonimo, solitamente molto piccolo, che è possibile inserire nel codice che poi chiama la funzione lambda proprio come una normale funzione. Le funzioni lambda non sono molto usate ma sono molto utili quando altrimenti dovresti creare molte piccole funzioni, utilizzate solo una volta. Sono spesso utilizzati in contesti di programmazione GUI o di rete, in cui il toolkit di programmazione richiede una funzione da richiamare con un risultato. Una funzione lambda è definita in questo modo:

lambda <param1**,** param2**, ...,**paramN> **:** <expression>

Questa è la parola letterale lambda, seguita da un elenco opzionale separato da virgole di nomi di parametri, due punti e un'espressione Python arbitraria che, di solito, utilizza i parametri di input. Nota che l'espressione non ha la parola return davanti ad essa. Alcuni linguaggi consentono alle funzioni lambda di essere arbitrariamente complesse; tuttavia, Python ti limita ad una singola espressione. L'espressione può diventare piuttosto complessa, ma in pratica è meglio creare una funzione standard se necessario perché sarà più facile leggere ed eseguire il debug se le cose vanno male. È possibile assegnare funzioni lambda alle variabili, nel qual caso tali variabili assomigliano esattamente ai nomi delle funzioni Python standard. Ad esempio, ecco la funzione di esempio straight_line re-implementata come funzione lambda:

```
>>> straight_line = lambda m,x,c: m*x+c
>>> straight_line(2,4,-3)
5
```

Ricorda solo che sono semplicemente un modo conciso per esprimere una breve, singola espressione, funzione.

Classi

Python supporta la programmazione orientata agli oggetti utilizzando un approccio tradizionale basato sulle classi. Le classi Python supportano l'ereditarietà multipla e l'overloading degli operatori (ma non l'overloading del metodo), nonché i normali meccanismi di incapsulamento e passaggio di messaggi.

Le classi Python non implementano direttamente l'occultamento dei dati (data hiding), sebbene siano utilizzate alcune convenzioni di denominazione per fornire un sottile livello di privacy per gli attributi e per suggerire quando gli attributi non dovrebbero essere utilizzati direttamente dai client. Sono supportati metodi e dati di classe, nonché i concetti di proprietà e slot. Le classi hanno sia un costruttore (__new__ ()) che un inizializzatore (__init__ ()), nonché un meccanismo di distruzione (__del__ ()), sebbene non sia sempre garantito che quest'ultimo venga invocato.

Le classi fungono da namespace per i metodi e i dati definiti mentre gli oggetti sono istanze di classi. Una classe viene definita usando la parola chiave class seguita dal nome della classe e da un elenco di superclassi tra parentesi. La definizione di classe contiene un insieme di dati di classe e definizioni di metodo. Una definizione di metodo ha come primo parametro un riferimento all'istanza chiamante, tradizionalmente chiamata self. Una semplice definizione di classe è simile a questa:

```
class MyClass(object):
instance_count = 0
def __init__(self, value):
self.__value = value
MyClass.instance_count += 1
print("instance No {} created".format(MyClass.instance_count))
def aMethod(self, aValue):
self.__value *= aValue
def __str__(self):
return "A MyClass instance with value: " + str(self.__value)
def __del__(self):
MyClass.instance_count -= 1
```

Il nome della classe inizia tradizionalmente con una lettera maiuscola. In Python 3 la superclasse è sempre object a meno che non sia specificato diversamente, quindi l'uso di object come superclasse nell'esempio precedente è effettivamente ridondante. L'elemento di dati instance_count è un attributo di classe perché non appare all'interno di nessuno dei metodi della classe. La funzione __init__() è l'inizializzatore infatti i costruttori sono usati raramente in Python a meno che non erediti da una classe incorporata, essa imposta la variabile di istanza self.__ value, incrementa la variabile di classe precedentemente definita instance_count e quindi stampa un messaggio. I doppi caratteri di sottolineatura prima del valore (__) indicano che si tratta effettivamente di dati privati e non devono essere utilizzati direttamente. Il metodo __init__() viene chiamato automaticamente da Python subito dopo la costruzione dell'oggetto. Il metodo di istanza aMethod() modifica l'attributo di istanza creato nel metodo __init__().

Il metodo __str__() è un metodo speciale utilizzato per restituire una stringa formattata in modo tale che le istanze passate alla funzione print, ad esempio, vengano stampate in modo significativo. Il distruttore __del__() decrementa il conteggio dell'istanza della variabile di classe instance_count quando l'oggetto viene distrutto. Puoi creare un'istanza della classe in questo modo:

myInstance = MyClass**(**42**)**

Questo crea un'istanza in memoria e quindi chiama MyClass.__init__() con la nuova istanza come self e 42 come valore. Puoi chiamare il metodo aMethod() usando la dot notation in questo modo:

myInstance**.**aMethod**(**66**)**

Questo si traduce nell'invocazione più esplicita:

MyClass**.**aMethod**(**myInstance**,** 66**)**

e risulta nel comportamento desiderato per cui il valore dell'attributo __value viene regolato. Puoi vedere all’opera il metodo __str__() se stampi l'istanza, in questo modo:

print(myInstance**)**

Questo dovrebbe stampare il messaggio:

A MyClass instance **with** value**:** 2772

Puoi anche stampare il valore dell'istance_count prima e dopo aver creato/distrutto un'istanza:

```
print(MyClass.instance_count)
inst = MyClass(44)
print(MyClass.instance_count)
del(inst)
print(MyClass.instance_count)
```

Questo dovrebbe mostrare che il conteggio viene incrementato e poi nuovamente decrementato, potrebbe verificarsi un leggero ritardo prima che il distruttore venga chiamato, ma dovrebbe essere solo di pochi istanti.

I metodi __init__(), __del__() e __str__() non sono gli unici metodi speciali. Esistono molti altri, tutti riconoscibili dall'uso di doppi trattini bassi (a volte sono chiamati metodi dunder). L'overloading dell'operatore è supportato tramite una serie di questi metodi speciali, tra cui: __add__ (), __sub__ (), __mul__ (), __div__ () e così via. Altri metodi prevedono l'implementazione di protocolli Python come l'iterazione o la gestione del contesto. Puoi sovrascrivere questi metodi nelle tue classi ma non dovresti mai definire i tuoi metodi più difficili; in caso contrario, i futuri miglioramenti di Python potrebbero compromettere l’esecuzione del codice. È possibile sovrascrivere i metodi nelle sottoclassi e le nuove definizioni possono richiamare la versione ereditata del metodo utilizzando la funzione super(), in questo modo:

```
class SubClass(Parent):
def __init__(self, aValue):
super().__init__(aValue)
```

La chiamata a super().__init__() si traduce in una chiamata al metodo __init__() di Parent . L'uso di super() evita problemi, in particolare con

l'ereditarietà multipla, in cui una classe potrebbe essere ereditata più volte, di solito non si desidera che venga inizializzata più di una volta.

Gli slot sono un dispositivo di salvataggio della memoria e vengono richiamati utilizzando l'attributo speciale __slot__ e fornendo un elenco dei nomi degli attributi degli oggetti. Spesso gli __slot__ sono un'ottimizzazione prematura e dovresti usarli solo se hai una specifica esigenza. Le proprietà sono un'altra funzionalità disponibile per gli attributi dei dati e consentono di rendere un attributo di sola lettura (o anche di sola scrittura) forzando l'accesso tramite un insieme di metodi anche se non viene utilizzata la consueta sintassi del metodo. Ecco un esempio in cui crei una classe Circle con un attributo raggio e un metodo area(). Vuoi che il valore del raggio sia sempre positivo, quindi non vuoi che i client lo possano cambiare con un valore negativo. Vuoi anche che area assomigli a un attributo di dati di sola lettura anche se è implementato come metodo. Puoi raggiungere entrambi gli obiettivi creando proprietà di raggio e area.

Iniziamo creando una semplice classe Circle1 che ha un solo attributo e due metodi richiamabili: setRadius() e area(). Quindi crei una seconda classe, Circle2, che crea proprietà di raggio e area. Infine, vedremo come l'uso delle proprietà semplifichi l'uso della classe nel codice client.

1. Avvia il tuo editor di programmazione o IDE preferito e crea un nuovo file chiamato testCircle.py
2. Inserisci questo codice:

```
class Circle1:
def __init__(self, radius):
self.__radius = radius
def setRadius(self,newValue):
if newValue >= 0:
class Circle1:
def __init__(self, radius):
self.__radius = radius
def setRadius(self,newValue):
if newValue >= 0:
self.__radius = newValue
else: raise ValueError("Value must be positive")
def area(self):
```

```
    return 3.14159 * (self.__radius ** 2)
    class Circle2:
    def __init__(self, radius):
    self.__radius = radius
    def __setRadius(self, newValue):
    if newValue >= 0:
    self.__radius = newValue
    else: raise ValueError("Value must be positive")
    radius = property(None, __setRadius)
    @property
    def area(self):
    return 3.14159 * (self.__radius ** 2)
```

1. Salva il codice
2. Esegui l'interprete Python e digita il seguente codice per usare Circle1:

```
>>> import testCircle as tc
    >>> c1 = tc.Circle1(42)
    >>> c1.area()
    5541.76476
    >>> print(c1.__radius)
    Traceback (most recent call last):
    File "<interactive input>", line 1, in <module>
    AttributeError: 'Circle1' object has no attribute '__radius'
    >>> c1.setRadius(66)
    >>> c1.area()
    13684.766039999999
    >>> c1.setRadius(-4)
    Traceback (most recent call last):
    File "<interactive input>", line 1, in <module>
    File "D:\PythonCode\Chapter1\testCircle.py", line 7, in setRadius
    else: raise ValueError("Value must be positive")
    ValueError: Value must be positive
```

1. Usiamo Circle2 con il seguente codice:

```
>>> c2 = Circle2(42)
    >>> c2.area
    5541.76476
    >>> print(c2.radius)
    Traceback (most recent call last):
    File "<interactive input>", line 1, in <module>
    AttributeError: unreadable attribute
    >>> c2.radius = 12
    >>> c2.area
    452.38896
    >>> c2.radius = -4
    Traceback (most recent call last):
    File "<interactive input>", line 1, in <module>
    File "D:\PythonCode\Chapter1\testCircle.py", line 18, in __setRadius
    else: raise ValueError("Value must be positive")
    ValueError: Value must be positive
    >>>
```

In testCircle.py hai creato due classi. La prima, Circle1, ha ottenuto ciò che si voleva fare costringendo l'utente a modificare il valore del raggio tramite il metodo setRadius(). L'hai fatto anteponendo l'attributo self.__radius con due trattini bassi, che è il modo in cui Python rende il tutto privato. È stato quindi creato il metodo setRadius() che ha convalidato il valore fornito prima di applicarlo e ha generato un errore se è stato trovato un valore negativo. Hai anche fornito un metodo area() in modo che l'utente possa valutare l'area usando la consueta tecnica di chiamata al metodo. La seconda classe, Circle2, ha affrontato le cose in modo piuttosto diverso perché ha utilizzato la funzione di definizione delle proprietà di Python per creare un attributo chiamato radius che era di sola scrittura. Ha anche creato il metodo area come attributo di sola lettura. Ciò ha reso il codice utente per Circle2 molto più intuitivo, come hai visto quando hai usato il codice nell'interprete. La chiave si trova nella funzione di tipo property() che hai chiamato in questo modo:

radius = **property(None,** __setRadius**)**

Questo codice accetta come argomenti un insieme di funzioni per la lettura, la scrittura e l'eliminazione (oltre a una stringa di documentazione). Il valore predefinito di ciascuno è None. In questo caso hai creato la proprietà radius con una funzione di lettura None ma con il metodo (ora privato) __setRadius() come funzione di scrittura. Gli altri valori sono stati lasciati al valore predefinito None. Il risultato era che l'utente poteva accedere a radius come se fosse un attributo di dati pubblici durante l'assegnazione di un valore ma, in modo implicito, Python chiamava il metodo __setRadius(). Qualsiasi tentativo di leggere (o eliminare) l'attributo verrebbe ignorato perché l'azione viene indirizzata a None. La proprietà area è leggermente diversa e utilizza un decoratore di proprietà Python (@property), che è solo una scorciatoia per creare una proprietà di sola lettura. Questo è un uso molto comune delle proprietà. Guardando la sessione, hai creato un'istanza Circle1 e stampato l'area usando il metodo area(). Hai quindi provato a stampare il raggio direttamente accedendo a __radius, ma Python ha fatto finta che non avesse tale attributo (a causa dell'impostazione privata del doppio underscore) e ha sollevato un AttributeError. Quando hai usato il metodo setRadius(), tutto andava bene e la stampa dell'area una seconda volta ha mostrato che la modifica ha funzionato. Infine, hai provato a impostare un raggio negativo e, come previsto, il metodo ha sollevato un'eccezione ValueError con un messaggio di errore personalizzato.

Nella sessione che utilizza Circle2, puoi vedere quanto è più semplice il codice. Valuta semplicemente il nome della proprietà dell'area per ottenere l'area e assegna un valore al nome della proprietà radius. Quando si tenta di assegnare un valore negativo, il metodo genera nuovamente un ValueError. La stampa diretta del raggio di nuovo genera un AttributeError, anche se questa volta ha un messaggio leggermente diverso. Le proprietà richiedono un piccolo sforzo extra da parte del programmatore, ma possono semplificare notevolmente l'utilizzo della classe.

Dopo aver visto come estendere le capacità di Python usando funzioni e classi, la sezione successiva mostra come racchiudere queste estensioni in moduli e pacchetti per favorire il riutilizzo.

Moduli e package

I moduli sono fondamentali per la maggior parte degli ambienti di programmazione destinati a compiti di programmazione non banali. Consentono ai programmi di essere suddivisi in blocchi gestibili e forniscono un meccanismo per il riutilizzo del codice tra i progetti.

In Python, i moduli sono semplicemente file sorgente, che terminano con .py e si trovano da qualche parte in cui Python può trovarli. In pratica, ciò significa che il file deve trovarsi nella directory di lavoro corrente o in una cartella elencata nella variabile sys.path. Puoi aggiungere le tue cartelle a questo percorso specificandole nella variabile di ambiente PYTHONPATH per il tuo sistema o in modo dinamico in fase di esecuzione. Sebbene i moduli forniscano un modo utile per impacchettare piccole quantità di codice sorgente per il riutilizzo, non sono del tutto soddisfacenti per progetti più grandi come framework GUI o librerie di funzioni matematiche. Per questi Python fornisce il concetto di pacchetto o package. Un pacchetto è essenzialmente una cartella piena di moduli. L'unico requisito è che la cartella contenga un file chiamato __init__.py, che potrebbe essere vuoto. Per l'utente un pacchetto assomiglia molto a un modulo e i sottomoduli all'interno del pacchetto possono sembrare attributi del modulo.

Accedi ai moduli usando la parola chiave import, che ha molte varianti in Python. Le forme più comuni sono:

import aModule
import aModule **as** anAlias
import firstModule**,** secondModule**,** thirdModule**...**
from aModule **import** anObject
from aModule **import** anObject **as** anAlias
from aModule **import** firstObject**,**secondObject**,** thirdObject**...**
from aModule **import ***

L'ultimo modulo importa tutti i nomi visibili da aModule nel namespace corrente. Ciò comporta il concreto rischio di creare conflitti di denominazione con nomi incorporati o nomi che hai definito, o definirai,

localmente. Si consiglia pertanto di utilizzare solo l'ultimo modulo di importazione per testare i moduli al prompt di Python. La digitazione extra implicata nell'uso degli altri moduli è un piccolo prezzo da pagare rispetto alla confusione che può derivare da uno scontro di nomi. Gli altri from... sono molto più sicuri perché importi solo nomi specificati e, se necessario, li rinomini con un alias, ciò rende molto meno probabili gli scontri con altri nomi locali.

Dopo aver importato un modulo utilizzando una delle prime tre forme, è possibile accedere al suo contenuto anteponendo il nome richiesto al nome del modulo (o alias) utilizzando la notazione a punto (dot notation). Ne hai già visti esempi nelle sezioni precedenti; ad esempio, sys.path è un attributo del modulo sys. Detto che i moduli sono semplicemente file sorgente, in pratica, vediamo alcune cose da fare e da non fare durante la creazione dei moduli.

Dovresti evitare il codice di primo livello che verrà eseguito quando il modulo viene importato, tranne, possibilmente, per alcune inizializzazioni di variabili che possono dipendere dall'ambiente locale. Ciò significa che il codice che si desidera riutilizzare deve essere impacchettato come una funzione o una classe. È anche possibile fornire una funzione di test che esegue tutte le funzioni e le classi nel modulo. Anche i nomi dei moduli sono tradizionalmente solo minuscoli.

Puoi controllare la visibilità degli oggetti all'interno del tuo modulo in due modi. Il primo è simile al meccanismo di privacy utilizzato nelle classi in quanto è possibile prefissare un nome con un carattere di sottolineatura. Tali nomi non verranno esportati quando viene utilizzata un'istruzione: from x import *. L'altro modo per controllare la visibilità è elencare solo i nomi che si desidera esportare in una variabile di primo livello chiamata __all__ , ciò garantisce che solo i nomi che si desidera specificamente esportare siano effettivamente esportati ed è consigliato rispetto al metodo precedente se la visibilità è aspetto importante per te.

Hai scoperto all'inizio di questa sezione che un pacchetto Python è solo una cartella con un file chiamato __init__.py. Tutti gli altri file Python all'interno di quella cartella sono i moduli del pacchetto. Python considera i pacchetti solo come un altro tipo di modulo, il che significa che i pacchetti Python possono contenere altri pacchetti al loro interno con una profondità arbitraria, a condizione che ogni sottopacchetto abbia anche il proprio file __init__.py. Il file __init__.py in sé non è particolarmente speciale; è solo

un altro file Python e verrà caricato quando importi il pacchetto. Ciò significa che il file può essere vuoto, nel qual caso l'importazione del pacchetto dà semplicemente accesso ai moduli inclusi, oppure può contenere codice Python al suo interno come qualsiasi altro modulo. In particolare, può definire un elenco __all__, come descritto in precedenza, per controllare la visibilità, consentendo in modo efficace a un pacchetto di avere file di implementazione privati che non vengono esportati quando un client importa il pacchetto.

Un requisito comune quando si crea un pacchetto è aver incluso un insieme di funzioni o dati condivisi tra tutti i moduli. È possibile ottenere ciò inserendo il codice condiviso in un file di modulo chiamato, ad esempio, common.py al livello più alto del pacchetto e facendo in modo che tutti gli altri moduli importino quello in comune. Sarà visibile a loro come parte del pacchetto, ma se non è esplicitamente incluso nell'elenco __all__, gli utenti esterni che usano i pacchetti non lo vedranno. Quando si tratta di utilizzare un pacchetto, lo tratti come qualsiasi altro modulo. Sono disponibili tutti i consueti stili di importazione, ma lo schema di denominazione viene esteso utilizzando la notazione a punto per specificare quali sottomoduli sono necessari dalla gerarchia dei pacchetti. Quando importi un sottomodulo usando la dot notation, definisci effettivamente due nuovi nomi: il pacchetto e il modulo. Si consideri, ad esempio, questa dichiarazione:

import os**.**path

Questo codice importa il sottomodulo del percorso del pacchetto os, ma rende anche visibile l'intero modulo del sistema operativo. Puoi procedere per accedere alle funzioni di os senza avere un'istruzione di importazione separata per os stesso. Un'implicazione di ciò è che Python esegue automaticamente tutti i file __init__.py che trova nella gerarchia importata. Quindi, nel caso di os.path, esegue os.__init__.py e quindi path.__init__.py. D'altra parte, se usi un alias dopo l'importazione, come questo:

import os**.**path **as** pth

solo il modulo os.path è esposto. Se vuoi usare le funzioni del modulo os, avrai bisogno di un'importazione esplicita. Sebbene sia esposto solo il percorso, come nome pth, entrambi i file os e path__init__.py verranno comunque eseguiti. La libreria standard Python contiene diversi pacchetti incluso il pacchetto os appena menzionato. Altri degni di nota includono i framework dell'interfaccia utente tkinter e curses, il pacchetto email e i pacchetti urllib, http e html focalizzati sul web.

Hai letto la teoria; ora è il momento di metterla in pratica. In questa sezione creeremo un paio di moduli e li raggrupperemo in un pacchetto. L'intenzione è fornire un'interfaccia funzionale per gli operatori logici ed estendere il loro ambito per includere il test dei singoli valori di bit. In questo modo vedrai anche molte delle funzionalità del linguaggio principale di Python che sono state discusse in precedenza. I moduli sviluppati non sono ottimizzati per le prestazioni ma sono progettati per illustrare i concetti. Tuttavia, non sarebbe difficile raffinarli in strumenti veramente utili. Iniziamo creando un modulo semplice e convenzionale basato su input interi, quindi creando un altro modulo che definisce una classe che può essere usata per rappresentare un pezzo di dati binari ed esporre le funzioni bit per bit come metodi. Infine, servirà un pacchetto contenente entrambi i moduli.

1. Crea una cartella chiamata bitwise che eventualmente sarà il tuo package
2. In quella cartella crea uno script Python chiamato bits.py contenente il codice seguente:

```
#! /bin/env python3
''' Functional wrapper around the bitwise operators.
Designed to make their use more intuitive to users not
familiar with the underlying C operators.
Extends the functionality with bitmask read/set operations.
The inputs are integer values and
return types are 16 bit integers or boolean.
bit indexes are zero based
Functions implemented are:
NOT(int) -> int
AND(int, int) -> int
```

```
OR(int,int) -> int
XOR(int, int) -> int
shiftleft(int, num) -> int
shiftright(int, num) -> int
bit(int,index) -> bool
setbit(int, index) -> int
zerobit(int,index) -> int
listbits(int,num) -> [int,int...,int]
'''
def NOT(value):
return ~value
def AND(val1,val2):
return val1 & val2
def OR(val1, val2):
return val1 | val2
def XOR(val1,val2):
return val1^val2
def shiftleft(val, num):
return val << num
def shiftright(val, num):
return val >> num
def bit(val,idx):
mask = 1 << idx # all 0 except idx
return bool(val & 1)
def setbit(val,idx):
mask = 1 << idx # all 0 except idx
return val | mask
def zerobit(val, idx):
mask = ~(1 << idx) # all 1 except idx
return val & mask
def listbits(val):
num = len(bin(val)) - 2
result = []
for n in range(num):
result.append( 1 if bit(val,n) else 0 )
return list( reversed(result) )
```

1. Salva il file e, restando nella cartella bitwise, esegui l'interprete Python
2. Digita il seguente codice per testare il modulo:

```
>>> import bits
    >>> bits.NOT(0b0101)
    -6
    >>> bin(bits.NOT(0b0101))
    '-0b110'
    >>> bin(bits.NOT(0b0101) & 0xF)
    '0b1010'
    >>> bin(bits.AND(0b0101, 0b0011) & 0xF)
    '0b1'
    >>> bin(bits.AND(0b0101, 0b0100) & 0xF)
    '0b100'
    >>> bin(bits.OR(0b0101, 0b0100) & 0xF)
    '0b101'
    >>> bin(bits.OR(0b0101, 0b0011) & 0xF)
    '0b111'
    >>> bin(bits.XOR(0b0101, 0b11) & 0xF)
    '0b110'
    >>> bin(bits.XOR(0b0101, 0b0101) & 0xF)
    '0b0'
    >>> bin(bits.shiftleft(0b10,1))
    '0b100'
    >>> bin(bits.shiftleft(0b10,4))
    '0b100000'
    >>> bin(bits.shiftright(0b1000,2))
    '0b10'
    >>> bin(bits.shiftright(0b1000,6))
    '0b0'
    >>> bits.bit(0b0101,0)
    True
    >>> bits.bit(0b0101,1)
    False
    >>> bin(bits.setbit(0b1000,1))
    '0b1010'
```

```
>>> bin(bits.zerobit(0b1000,1))
'0b1000'
>>> bits.listbits(0b10111)
[1, 0, 1, 1, 1]
```

Il modulo è un elenco abbastanza semplice di funzioni che racchiudono gli operatori bit per bit per not (~) e (&), o (|), xor (^), shift left (<<) e shift right (>). Queste operazioni funzionano su dati binari, ovvero semplicemente una sequenza di 1 e 0 memorizzata come un'unità all'interno del computer. Tutti i dati nel computer sono, per definizione, archiviati in forma binaria. Queste operazioni wrapper sono integrate da una serie di funzioni che verificano se un bit ha un valore 1 (questo è noto come "set"), imposta un bit (su 1) o zero un bit (noto anche come "reset"). Il numero di bit si conta a partire da destra, partendo da zero. I test vengono eseguiti utilizzando un modello di bit (noto anche come maschera di bit) che, in tutti i casi tranne zerobit(), è costituito da tutti gli zeri tranne il bit che si desidera testare o impostare. Hai creato la maschera spostando 1 a sinistra del numero di bit richiesto. La funzione zerobit() usa il complemento bit per bit della solita maschera per crearne una composta da tutti 1 tranne uno 0 in cui si desidera reimpostare il bit. Infine, hai una funzione che elenca i singoli bit del valore dato. Quest'ultima funzione è leggermente più complessa e mostra alcune delle funzionalità di Python.

Per prima cosa determini la lunghezza del numero convertendolo in una stringa binaria con bin() e sottraendo 2 (per tenere conto dei caratteri 0b iniziali). Quindi crei un elenco di risultati vuoto e fai scorrere i bit. Per ogni bit aggiungi 1 o 0, a seconda che il bit sia settato o meno, usando il costrutto dell'espressione condizionale di Python.

Il test del modulo fa sorgere alcune questioni interessanti. Iniziamo importando il tuo nuovo modulo. Poiché ti trovi nella cartella in cui si trova il file, Python può vederlo senza modificare il valore sys.path. Inizi a testare con la funzione NOT() (in modalità prefissa, ovviamente, con il nome del modulo, bits), e subito puoi vedere un'anomalia in quanto l'interprete Python stampa la rappresentazione decimale come risultato. Per aggirare il problema, puoi utilizzare la funzione bin() per convertire il numero in una rappresentazione di stringa binaria, tuttavia, c'è ancora un problema perché il numero è negativo. Questo perché gli interi Python sono con segno, cioè

possono rappresentare numeri positivi o negativi. Python lo fa internamente facendo in modo che il bit più a sinistra rappresenti il segno. Invertendo tutti i bit, inverti anche il segno! Puoi evitare questo problema usando una maschera di bit di 0xF (o decimale 15 se preferisci) per recuperare solo i 4 bit più a destra. Convertendo questo con bin(), ora vedi il modello di bit invertito che ti aspettavi. Ovviamente, se il valore che stavi invertendo era maggiore di 16, dovresti usare una maschera di bit più lunga. Ricorda solo che ogni cifra esadecimale è di 4 bit, quindi tutto ciò che devi fare è aggiungere una F in più alla tua maschera.

Il prossimo set di test, che copre le funzioni AND() fino a shiftleft(), dovrebbe essere semplice e puoi controllare i risultati ispezionando visivamente i modelli di bit di input e i risultati. Gli esempi con shiftright() mostrano un risultato interessante in quanto lo spostamento dei bit troppo a destra produce un risultato zero. In altre parole, Python riempie con zeri lo spazio “vuoto” lasciato dalle operazioni di spostamento. Passando alla nuova funzionalità, hai usato bit(), setbit() e zerobit() per testare e modificare i singoli bit all'interno del valore dato. Ancora una volta, puoi ispezionare visivamente i modelli di input e di risultato per vedere che vengono prodotti i risultati corretti. Ricorda che il parametro index conta da zero partendo da destra.

Infine, hai testato la funzione listbits(). Ancora una volta, puoi facilmente confrontare il modello di input binario con l'elenco di numeri risultante. Ora hai un modulo funzionante che puoi importare e utilizzare proprio come qualsiasi altro modulo in Python. Potresti migliorare ulteriormente il modulo fornendo una funzione di test e racchiudendola in una clausola if __name__ se lo desideri, ma per ora puoi capire come passare da un singolo modulo a un pacchetto. Crea una classe che replica le funzioni in bits.py come un insieme di metodi; quindi, raggruppa entrambi i moduli in un pacchetto.

1. Naviga nella cartella bitwise
2. Crea un nuovo file chiamato bitmask.py con il seguente codice:

```
#! /bin/env python3
''' Class that represents a bit mask.
It has methods representing all
the bitwise operations plus some
```

```
additional features. The methods
return a new BitMask object or
a boolean result. See the bits
module for more on the operations
provided.
'''
class BitMask(int):
def AND(self,bm):
return BitMask(self & bm)
def OR(self,bm):
return BitMask(self | bm)
def XOR(self,bm):
return BitMask(self ^ bm)
def NOT(self):
return BitMask(~self)
def shiftleft(self, num):
return BitMask(self << num)
def shiftright(self, num):
return BitMask(self > num)
def bit(self, num):
mask = 1 << num
return bool(self & mask)
def setbit(self, num):
mask = 1 << num
return BitMask(self | mask)
def zerobit(self, num):
mask = ~(1 << num)
return BitMask(self & mask)
def listbits(self, start=0,end=-1):
end = end if end < 0 else end+2
return [int(c) for c in bin(self)[start+2:end]]
```

1. Ora salvalo in modo da poterlo testare nell'interprete Python
2. Restando nella cartella bitwise, avvia Python e digita il seguente codice:

```
>>> import bitmask
    >>> bm1 = bitmask.BitMask()
    >>> bm1
    0
    >>> bin(bm1.NOT() & 0xf)
    '0b1111'
    >>> bm2 = bitmask.BitMask(0b10101100)
    >>> bin(bm2 & 0xFF)
    '0b10101100'
    >>> bin(bm2 & 0xF)
    '0b1100'
    >>> bm1.AND(bm2)
    0
    >>> bin(bm1.OR(bm2))
    '0b10101100'
    >>> bm1 = bm1.OR(0b110)
    >>> bin(bm1)
    '0b110'
    >>> bin(bm2)
    '0b10101100'
    >>> bin(bm1.XOR(bm2))
    '0b10101010'
    >>> bm3 = bm1.shiftleft(3)
    >>> bin(bm3)
    '0b110000'
    >>> bm1 == bm3.shiftright(3)
    True
    >>> bm4 = bitmask.BitMask(0b11110000)
    >>> bm4.listbits()
    [1, 1, 1, 1, 0, 0, 0]
    >>> bm4.listbits(2,5)
    [1, 1, 0]
    >>> bm4.listbits(2,-2)
    [1, 1, 0, 0]
```

1. Chiudi l’interprete, ora che hai dimostrato che il nuovo modulo funziona, puoi andare avanti e convertire la directory bitwise in un pacchetto Python.
2. Crea un file vuoto __init.py__
3. Per verificare che il pacchetto funzioni, devi cambiare la tua directory di lavoro in quella superiore a bitwise. Ora devi verificare che puoi importare il pacchetto e il suo contenuto e accedere alle funzionalità.
4. Avvia l'interprete Python e digita il seguente codice di test:

```
>>> import bitwise.bits as bits
    >>> from bitwise import bitmask
    >>> bits
    <module 'bitwise.bits' from 'bitwise/bits.py'>
    >>> bitmask
    <module 'bitwise.bitmask' from 'bitwise/bitmask.py'>
    >>> bin(bits.AND(0b1010,0b1100))
    '0b1000'
    >>> bin(bits.OR(0b1010,0b1100))
    '0b1110'
    >>> bin(bits.NOT(0b1010))
    '-0b1011'
    >>> bin(bits.NOT(0b1010) & 0xFF)
    '0b11110101'
    >>> bin(bits.NOT(0b1010) & 0xF)
    '0b101'
    >>> bm = bitmask.BitMask(0b1100)
    >>> bin(bm)
    '0b1100'
    >>> bin(bm.AND(0b1110))
    '0b1100'
    >>> bin(bm.OR(0b1110))
    '0b1110'
    >>> bm.listbits()
    [1, 1, 0]
```

Hai creato una classe basata sul tipo intero integrato in Python, int. Poiché si forniscono solo nuovi metodi per la classe e non si memorizzano attributi di dati aggiuntivi, non è necessario fornire un costruttore __new__() o un inizializzatore __init__(). I metodi sono tutti molto simili alle funzioni scritte in bits.py tranne per il fatto che hai creato un'istanza BitMask come tipo restituito. Il metodo listbits() mostra anche un approccio alternativo per derivare l'elenco utilizzando la rappresentazione di stringa bin() e creare tale elenco utilizzando una conversione da carattere a intero utilizzando int(). La funzione listbits() è stata anche estesa per fornire una coppia di parametri di inizio e fine che per impostazione predefinita corrispondono alla lunghezza completa del numero binario, ma possono essere utilizzati anche per estrarre un sottoinsieme di bit. C'è un po' di lavoro extra a seconda che si tratti di un indice positivo o negativo. Gli indici negativi non richiedono l'aggiunta di due caratteri perché si applicano automaticamente dall'estremità destra; quindi, un'assegnazione condizionale di Python assicura che sia impostato il valore finale corretto.

Dopo aver creato la classe, l'hai quindi testata come modulo standard importandolo dalla directory locale. Hai quindi ripetuto una serie di test simili a quelli che hai eseguito per bits.py. Una cosa importante da considerare: puoi combinare e abbinare i tradizionali operatori bit a bit con le nuove versioni funzionali. Puoi anche confrontare gli oggetti BitMask proprio come qualsiasi altro numero intero, come hai visto nell'esempio shiftright(). Infine, hai dimostrato che il tuo nuovo algoritmo listbits() ha funzionato e che i nuovi argomenti aggiuntivi funzionano come previsto sia per i valori positivi che per quelli negativi.

A questo punto sono stati creati due moduli standard in una cartella. Hai quindi creato un file vuoto __init__.py che ha trasformato la cartella in un pacchetto Python. Per verificare che funzionasse, hai cambiato directory puntando a quella in alto in modo che il pacchetto fosse visibile all'interprete. Confermato che è possibile importare il pacchetto e i moduli al suo interno, è possibile accedere ad alcune delle funzionalità. Sapere come creare e utilizzare i moduli e i pacchetti standard, oltre a quelli creati da te, è un ottimo punto di partenza.

11

SCRIPT PYTHON

Spesso, potresti trovarti a svolgere compiti che implicano molte operazioni ripetitive. Per combattere questa ripetizione del lavoro, potrebbe essere possibile scrivere una macro per automatizzare tali operazioni all'interno di una singola applicazione ma, se le operazioni si estendono su più applicazioni, le macro raramente sono efficaci. Ad esempio, se si esegue il backup e l'archiviazione di un'applicazione Web multimediale di grandi dimensioni, potrebbe essere necessario gestire il contenuto prodotto da uno o più strumenti multimediali, il codice di un IDE e probabilmente anche alcuni file del database. Invece delle macro, è necessario uno strumento di programmazione esterno per guidare ogni applicazione, o utilità, per eseguire la sua parte del tutto. Python è particolarmente adatto a questo tipo di ruolo di orchestrazione.

Scripting è un termine che può avere diversi significati, è importante chiarire in anticipo cosa significa quando lo incontrerai in questo capitolo. Significa coordinare le azioni di altri programmi o applicazioni per eseguire un'attività come la stampa di file in serie o automatizzare un flusso di lavoro, come l'aggiunta di un nuovo utente. Potrebbero essere utilizzati strumenti del sistema operativo o pacchetti generici di grandi dimensioni come una suite per la produttività dell'ufficio. Pensa al modo in cui una sceneggiatura in un'opera teatrale dice agli attori cosa dire, dove stare e quando entrare o uscire dal palco. Questo è ciò che fa lo "scripting" di Python; coordina il comportamento di altri programmi.

In questo capitolo imparerai come usare i moduli Python per controllare le impostazioni dell'utente così come i livelli di accesso alle directory e ai

file; impostare l'ambiente corretto per un'operazione; avviare e controllare i programmi esterni dal tuo script. Scoprirai anche come i moduli Python ti aiutano ad accedere ai dati nei comuni formati di file, come gestire date e orari e, infine, come accedere direttamente alle interfacce di programmazione di basso livello di applicazioni esterne usando il potentissimo modulo ctypes e, per Windows, il pacchetto pywin32. La maggior parte delle attività che un tipico programmatore deve svolgere utilizzando il sistema operativo, ad esempio la raccolta di informazioni sull'utente o la navigazione nel file system, possono essere eseguite in modo generico utilizzando la libreria di moduli standard di Python (ricorda che i moduli sono pezzi di codice riutilizzabili che possono essere condivisi tra più programmi). I moduli chiave sono stati scritti in modo tale che le peculiarità dei comportamenti dei singoli sistemi operativi siano state nascoste dietro un insieme di oggetti e operazioni con un livello di astrazione superiore. I moduli che considereremo in questa sezione sono: os/path, pwd, glob, shutil e subprocess. Il materiale qui si concentra su come utilizzare questi moduli in scenari comuni; non cerca di coprire ogni possibile opzione o scenario disponibile.

Il modulo os, come suggerisce il nome, fornisce l'accesso a molte funzionalità del sistema operativo. Si tratta, infatti, di un pacchetto con un sottomodulo, os.path, che si occupa di gestire percorsi, nomi e tipi di file. Il modulo os è supportato da numerosi altri moduli che vedremo in questo capitolo. Questi numerosi moduli sono indicati collettivamente come i moduli di OS (maiuscolo) e il modulo del sistema operativo effettivo come os (minuscolo). Se hai familiarità con la programmazione di sistemi su un sistema UNIX, o anche con l'utilizzo di una shell UNIX come Bash, molte di queste operazioni ti risulteranno familiari. Il sistema operativo serve principalmente per gestire l'accesso all'hardware del computer sotto forma di CPU, memoria, archiviazione e rete, regola l'accesso a queste risorse e gestisce la creazione, la pianificazione e la rimozione dei processi. Le funzioni del modulo OS forniscono informazioni e controllo su queste attività del sistema operativo. Nelle prossime sezioni, esaminerai queste attività comuni:

- Ritrovare informazioni dell’utente e del sistema
- Gestire processi
- Ritrovare informazioni e manipolare file

- Navigare tra le cartelle

Una delle prime cose che puoi fare quando esplori i moduli di OS è scoprire cosa possono dirti sugli utenti. In particolare, puoi scoprire l'ID utente, il nome di accesso e alcune delle sue impostazioni predefinite. Come la maggior parte delle novità in Python, il modo migliore per acquisire familiarità è tramite il prompt interattivo, quindi avvia l'interprete Python e provalo.

1. Avvia l’interprete Python
2. Digita il seguente codice:

```
>>> import os
>>> os.getlogin()
'agauld'
>>> os.getuid() # Not Windows
1001
>>> import pwd # Not Windows
>>> pwd.getpwuid(os.getuid()) # Not Windows
pwd.struct_passwd(pw_name='agauld', pw_passwd='unused', pw_uid=1001,
pw_gid=513, pw_gecos='Alan Gauld,U-DOCUMENTATION\\agauld,
S-1-5-21-2472883112-933775427-2136723719-1001',
pw_dir='/home/agauld', pw_shell='/bin/bash')
>>> for id in pwd.getpwall():
... print(id[0])
...
SYSTEM
LocalService
NetworkService
Administrators
TrustedInstaller
Administrator
agauld
Guest
HomeGroupUser$
```

```
????????
>>>
```

Dopo aver importato il modulo os nella prima riga, hai ottenuto il nome di accesso come stringa. Questo è generalmente più utile per creare prompt personalizzati o messaggi sullo schermo. Sfortunatamente, per gli utenti Windows, questo è tutto; il resto del codice è adatto solo per sistemi basati su UNIX. Tuttavia, non tutto è perduto perché puoi anche trovare alcune di queste informazioni dalle variabili di ambiente. Se hai un sistema basato su UNIX, usa os.getuid() per ottenere l'ID utente così come lo vede il sistema operativo, ovvero un valore numerico, che puoi quindi utilizzare con varie altre funzioni. Le righe successive importano e utilizzano le funzioni del modulo password, pwd, per tradurre l'ID utente del sistema operativo in un insieme più completo di informazioni che include il nome reale, la shell predefinita e la directory home. Questo è ovviamente molto più informativo, ma richiede l'UID di os.getuid() come punto di partenza. Una funzione alternativa, os.getpwnam(), prende invece il nome di login e restituisce le stesse informazioni. Infine, usando pwd.getpwall() e un ciclo for puoi estrarre tutti i nomi utente per questo sistema.

Così puoi scoprire che tipo di autorizzazioni hanno gli utenti sui file che creano. Questo è importante perché influisce su tutti i file prodotti dal codice. Potrebbe essere necessario modificare temporaneamente le autorizzazioni, ad esempio, se è necessario creare un file da eseguire successivamente nel programma, è necessario che il file disponga dei privilegi di esecuzione. In UNIX, queste impostazioni sono memorizzate in qualcosa noto come umask o maschera utente. È una maschera di bit, come quelle utilizzate nei precedenti capitoli, in cui ogni bit rappresenta un punto dati di accesso dell'utente, come descritto di seguito. Python ti consente di guardare il valore umask, anche su Windows, usando la funzione os.umask(). Tuttavia, la funzione os.umask() ha una leggera stranezza nel suo utilizzo, si aspetta che tu passi un nuovo valore alla funzione; quindi imposta quel valore e restituisce il vecchio valore. Ma se vuoi solo scoprire il valore corrente, non puoi farlo. Invece è necessario impostare umask su un nuovo valore temporaneo, leggere quello vecchio e quindi ripristinare il valore sull'originale. Il formato della maschera è molto compatto, composto da 3 gruppi di 3 bit, 1 gruppo per ciascuno dei permessi Proprietario,

Gruppo e Mondo, rispettivamente. All'interno di un gruppo i 3 bit rappresentano ciascuno un tipo di accesso: lettura, scrittura o esecuzione, scritti più comunemente usando la notazione binaria esplicita.

1. Fai partire l’interprete Python
2. Digita il seguente codice:

```
>>> import os
>>> os.umask(0b111111111) # binary for all false - 111 x 3
18
>>> bin(18)
'0b10010'
>>> os.umask(18)
511
```

Hai iniziato chiamando os.umask() con un valore binario di 111111111. Ciò imposta tutte le autorizzazioni su false, cosa che hai fatto come funzionalità di sicurezza nel caso in cui qualcosa fosse andato storto. È meglio avere la maschera troppo restrittiva piuttosto che lasciare l'utente vulnerabile agli exploit di sicurezza. Python ha quindi stampato il valore decimale 18, chiamando la funzione bin() si vede che 18 ha il valore della maschera binaria pari a 10010. Se lo si riempie con zeri per ottenere la maschera completa a 9 bit e la si divide in gruppi a 3 bit, vedrai che è 000-010-010, questo valore rappresenta l'accesso completo al Proprietario, ma solo in lettura ed esecuzione agli utenti del Gruppo e del Mondo. Infine, hai ripristinato l'impostazione originale dell'utente chiamando nuovamente os.umask() con un argomento di 18 (il valore originale restituito da umask) e il valore della maschera precedente (111111111) che avevi impostato è stato stampato, in decimale, come 511.

A volte vuoi sapere che tipo di sistema informatico è in esecuzione per l'utente, in particolare vuoi sapere i dettagli del sistema operativo stesso. Python ha diversi modi per farlo, ma quello che guarderai per primo è sicuramente la proprietà os.name. Al momento della scrittura, questa proprietà restituisce uno dei seguenti valori: posix, nt, mac, os2, ce o java. Un altro posto in cui cercare il sistema che l'utente sta utilizzando è nel modulo sys e, in particolare, nell'attributo sys.platform. Questo attributo restituisce spesso informazioni leggermente diverse da quelle trovate usando os.name. Ad esempio, Windows viene segnalato come win32

anziché nt o ce. Su UNIX un'altra funzione in os chiamata os.uname() fornisce un po' più di dettagli. Se hai a disposizione diversi sistemi operativi, può essere interessante confrontare i risultati di queste diverse tecniche. Si consiglia di utilizzare l'opzione os.name semplicemente perché è universalmente disponibile e restituisce un insieme ben definito di risultati. Un altro frammento di informazioni che è spesso utile raccogliere è la dimensione del terminale dell'utente in termini di righe e colonne. È possibile utilizzare queste informazioni per modificare la visualizzazione dei messaggi dai propri script. Il modulo shutil fornisce una funzione per questo chiamata shutil.get_terminal_size(), ed è usata in questo modo:

```
>>> import shutil
>>> cols, lines = shutil.get_terminal_size()
>>> cols
80
>>> lines
49
```

Se non è possibile determinare la dimensione del terminale, il valore di ritorno predefinito è 80 × 24. È possibile specificare un valore predefinito diverso come argomento facoltativo, ma 80 × 24 è solitamente un'opzione sensata perché è la dimensione tradizionale per gli emulatori di terminale.

Informazioni sui processi

Può essere utile per un programma sapere qualcosa sul suo stato attuale e sull'ambiente di runtime. Ad esempio, potresti voler conoscere l'identità del processo o se il processo ha una cartella preferita in cui scrivere i suoi file di dati o leggere i dati di configurazione. I moduli OS forniscono funzioni per determinare questi valori. Una di queste fonti di informazioni sul processo è l'ambiente di processo, come definito dalle variabili di ambiente. Il modulo os fornisce un dizionario chiamato os.environ che contiene tutte le variabili di ambiente per il processo corrente. Lo svantaggio delle variabili d'ambiente è che sono altamente volatili, gli utenti possono crearle e rimuoverle. Le applicazioni possono fare lo stesso, quindi è pericoloso fare affidamento sull'esistenza di una variabile di ambiente; dovresti sempre avere un valore predefinito su cui puoi ripiegare. Fortunatamente, alcuni valori sono abbastanza affidabili e generalmente presenti. Tre di questi sono particolarmente utili per gli utenti Windows perché le funzioni pwd.getpwuid() e os.uname() discusse in precedenza non sono disponibili. Queste sono HOME, OS e PROCESSOR_ARCHITECTURE. Se provi ad accedere a una variabile che non è definita, ottieni il consueto Python KeyError. Sulla maggior parte dei sistemi operativi, ma non su tutti, un programma può impostare o modificare variabili di ambiente. Se questa funzionalità è supportata per il tuo sistema operativo, Python riflette tutte le modifiche al dizionario os.environ nell'ambiente del sistema operativo.

Oltre a utilizzare le variabili di ambiente come fonte di informazioni sull'utente, è abbastanza comune usarle per definire i dettagli di configurazione specifici dell'utente su un programma, ad esempio la posizione di un database. Questa pratica è un po' contestata al giorno d'oggi ed è considerato meglio usare un file di configurazione per tali dettagli. Ma se dovessi lavorare con applicazioni meno recenti, potresti dover fare riferimento all'ambiente per questi dettagli.

1. Avvia l’interprete Python
2. Digita il seguente codice:

```
>>> import os
```

```
>>> os.getpid()
16432
>>> os.getppid()
3165
>>> os.getcwd()
/home/agauld
>>> len(os.environ)
48
>>> os.environ['HOME']
'/home/agauld'
>>> os.environ['testing123']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python2.7/UserDict.py", line 23, in __getitem__
raise KeyError(key)
KeyError: 'testing123'
>>> os.environ['testing123'] = 42
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python2.7/os.py", line 471, in __setitem__
putenv(key, item)
TypeError: str expected, not int
>>> os.environ['testing123'] = '42'
>>> os.environ['testing123']
'42'
>>> del(os.environ['testing123'])
>>> os.environ['testing123']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python2.7/UserDict.py", line 23, in __getitem__
raise KeyError(key)
KeyError: 'testing123'
```

Dopo aver importato os nella prima riga, la prima cosa che hai fatto è stata determinare l'ID del processo corrente. Sei nell'interprete Python, quindi è il processo dell'interprete stesso che stai identificando. La riga successiva

legge l'ID del processo padre; in questo caso, questo è il programma della shell del sistema operativo. È possibile utilizzare questi ID quando si interagisce con altri strumenti del sistema operativo. Hai quindi utilizzato la funzione os.getcwd() (che sta per get current working directory) per determinare quale directory è attualmente quella predefinita. Questa è solitamente la directory da cui è stato invocato l'interprete ma, come vedrai, è possibile cambiare directory all'interno del tuo programma. La funzione os.getcwd() è un modo utile per controllare esattamente dove sta lavorando il tuo codice in un dato istante. In seguito hai scoperto quanto era grande il tuo ambiente visualizzando la lunghezza del dizionario os.environ, quindi hai estratto il valore della variabile d'ambiente HOME che mostra la home directory dell'utente. Hai provato ad accedere a una variabile chiamata testing123, ma poiché non esisteva, hai ottenuto un KeyError. Hai tentato di creare una variabile testing123 assegnando il numero 42, ma ciò ha prodotto un TypeError perché le variabili di ambiente devono essere stringhe. Assegnando il valore di stringa '42' alla variabile testing123, sei riuscito a creare una nuova variabile di ambiente per questo processo (ricorda che l'ambiente è locale per questo processo e tutti i sottoprocessi che genera, non è visibile in altri processi esterni). Quindi hai letto di nuovo il valore e questa volta hai ottenuto un valore senza messaggi di errore, confermando che è andato tutto a buon fine. Per ultimo, hai eliminato la variabile e hai dimostrato che era sparita tentando ancora una volta di leggerla, provocando, come previsto, un errore.

Spesso è utile essere in grado di eseguire altri programmi dall'interno di uno script e il modulo subprocess è lo strumento preferito per questo. Il modulo contiene una classe chiamata Popen che fornisce un'interfaccia molto potente e flessibile ai programmi esterni, dispone anche di numerose funzioni utili che è possibile utilizzare quando si preferisce un approccio più semplice. La documentazione descrive come utilizzare tutte queste funzioni; in questa sezione si usa solo la funzione più semplice, subprocess.call(), e la classe Popen. L'uso più basilare del modulo consiste nel chiamare un comando del sistema operativo esterno e consentirgli semplicemente di eseguire il suo corso. L'output viene solitamente visualizzato sullo schermo o memorizzato in un file di dati da qualche parte. Al termine del programma, puoi chiedere all'utente di effettuare una selezione in base a ciò che è stato visualizzato oppure puoi accedere al file di dati direttamente dal tuo codice. È possibile forzare molti strumenti del

sistema operativo, in particolar modo su sistemi basati su UNIX, a produrre un file di dati come output fornendo opzioni della riga di comando appropriate o utilizzando il reindirizzamento del file del sistema operativo. Questa tecnica è un modo molto potente per sfruttare la potenza degli strumenti del sistema operativo in modo tale che Python possa utilizzarli per ulteriori elaborazioni. Questo meccanismo di base per chiamare un programma è racchiuso nella funzione subprocess.call(). Questa funzione ha un elenco di stringhe come primo parametro, seguito da diversi parametri di parole chiave opzionali utilizzati per controllare le posizioni di input e output e poche altre cose. Il modo più semplice per vedere come funziona è provarlo.

1. Crea una directory di test, chiamata root e aggiungi alcuni file di testo (creali da zero o copiali da qualche altra parte). Non importa cosa contengono; sei interessato solo alla loro presenza in questa fase. Per ottenere gli stessi risultati mostrati qui, la struttura dovrebbe apparire così:

```
root
fileA.txt
fileB.txt
```

1. Naviga nella cartella root e avvia l'interprete Python
2. Digita il codice seguente:

```
>>> import subprocess as sub
>>> sub.call(['ls']) # Not Windows
fileA.txt fileB.txt
0
>>> sub.call(['ls'], stdout=open('ls.txt', 'w')) # Not windows
0
>>> sub.call(['cmd', '/c', 'dir', '/b']) # Windows only
fileA.txt
fileB.txt
0
```

```
>>> sub.call(['cmd', '/c', 'dir', '/b'], stdout=open('ls.txt','w')) # Windows only
0
>>> sub.call(['more','ls.txt']) # Not Windows
fileA.txt
fileB.txt
ls.txt
0
>>> sub.call(['cmd','/c','type','ls.txt']) # Windows only
fileA.txt
fileB.txt
ls.txt
0
>>> for line in open('ls.txt'): print(line)
...
fileA.txt
fileB.txt
ls.txt
```

Dopo aver importato subprocess con l'alias sub nella prima riga (risparmia solo un po' di digitazione in seguito!), hai chiamato sub.call() per la prima volta, con un argomento ['ls'] o ['cmd', '/c', 'dir', '/b'] a seconda del tuo sistema operativo (si noti che dir è in realtà un sottocomando del processo della shell cmd.exe, il sistema di guida di Windows spiega cosa fanno i flag /ce /b). L'output viene visualizzato su stdout (il proprio terminale), ma i nomi dei file non sono accessibili da Python. L'unica cosa restituita da call() è il codice di ritorno del sistema operativo, che ti dice se il programma è stato completato correttamente o meno, ma non ti aiuta in alcun modo ad interagire con i dati. Hai quindi utilizzato sub.call() una seconda volta, ma questa volta hai reindirizzato stdout a un nuovo file: ls.txt.

Successivamente è stato utilizzato lo strumento del sistema operativo more (o type su Windows) per visualizzare il file. Il fatto che ls.txt sia un normale file di testo significa anche che puoi accedere ai dati aprendo il file ed elaborandolo nel solito modo usando i comandi Python. In questo caso hai semplicemente eseguito un loop sulle righe e le hai stampate, ma avresti potuto archiviare e utilizzare i dati per qualche altro scopo altrettanto

facilmente. Vale la pena notare che esporre un elenco di file in un file di testo come questo è un potenziale problema di sicurezza e dovresti eliminare il file al più presto dopo averlo elaborato.

Un problema che può verificarsi durante l'esecuzione di programmi esterni è che il sistema operativo non riesca a trovare il comando. In genere viene visualizzato un messaggio di errore quando ciò accade ed è necessario fornire esplicitamente il percorso completo del file di programma, supponendo che esista effettivamente. Infine, considera come interrompere un processo in esecuzione. Per i programmi interattivi, il modo più semplice è che l'utente chiuda il programma esterno normalmente o emetta un segnale di interruzione usando Ctrl+C o Ctrl+Z, o qualunque sia la norma sul sistema operativo dell'utente. Ma per i programmi non interattivi, potrebbe essere necessario intervenire dal sistema operativo, di solito esaminando l'elenco dei processi in esecuzione e terminando esplicitamente il processo errato. Hai appena visto quanto sia facile usare subprocess.call() per avviare un processo esterno. Ora impariamo come il modulo del sottoprocesso ti dà molto più controllo sui processi e, in particolare, come consente al tuo programma di interagire con essi mentre sono in esecuzione, in particolare come leggere l'output del processo direttamente dal tuo script.

Ancora più controllo

È possibile utilizzare la classe Popen per creare un'istanza di un processo o un comando. Sfortunatamente, la documentazione può sembrare piuttosto scoraggiante perché il costruttore Popen ha un bel po' di parametri. La buona notizia è che quasi tutti questi parametri hanno valori predefiniti utili e possono essere ignorati nei casi più semplici. Pertanto, per eseguire semplicemente un comando del sistema operativo dall'interno di uno script, devi solo (gli utenti Windows dovrebbero sostituire il comando dir dell'esempio precedente):

```
>>> import subprocess as sub
>>> sub.Popen(['ls', '*.*'], shell=True)
<subprocess.Popen object at 0x7fd3edec>
>>> book tmp
```

Nota l'argomento shell=True, ciò è necessario per ottenere il comando interpretato dal processore dei comandi del sistema operativo o dalla shell. In questo modo si assicura che i caratteri jolly ('*.*') così come le virgolette di stringa e simili vengano tutti interpretati nel modo previsto. Se non si utilizza il parametro shell, accade questo:

```
>>> sub.Popen(['ls', '*.*'])
<subprocess.Popen object at 0x7fcd328c>
>>> ls: cannot access *.*: No such file or directory
```

Senza specificare shell=True, il sistema operativo tenta di trovare un file con il nome letterale '*.*', che non esiste. Il problema con l'utilizzo di shell=True è che crea anche problemi di sicurezza sotto forma di un potenziale attacco injection; quindi, non usarlo mai se i comandi sono formulati da stringhe create dinamicamente, come quelle lette da un file o da un utente.

Un injection attack avviene quando un utente malintenzionato digita una stringa di input che viene letta e interpretata dal programma ma, anziché contenere dati innocui, contiene comandi potenzialmente dannosi, che

possono portare alla cancellazione di file o peggio. Il modulo shlex contiene una funzione quote() che può mitigare i rischi, ma occorre comunque prestare attenzione quando si eseguono stringhe generate dinamicamente. Per accedere all'output del comando in esecuzione, puoi aggiungere un paio di funzionalità extra alla chiamata, in questo modo:

```
>>> lsout = sub.Popen(['ls', '*.*'], shell=True, stdout=sub.PIPE).stdout
>>> for line in lsout:
... print (line)
```

Qui specifichi che stdout dovrebbe essere un sub.PIPE e quindi assegna l'attributo stdout dell'istanza Popen a lsout (una pipe è solo una connessione dati a un altro processo, in questo caso tra il tuo programma e il comando che stai eseguendo). Fatto ciò, puoi quindi trattare la variabile lsout proprio come un normale file Python, leggerlo e così via. Puoi inviare dati al processo più o meno allo stesso modo specificando che stdin è una pipe su cui puoi quindi scrivere. I valori validi che è possibile assegnare ai vari flussi includono file aperti, descrittori di file o altri flussi (in modo che stderr possa essere visualizzato su stdout, ad esempio). Si noti che è possibile concatenare comandi esterni impostando, ad esempio, l'ingresso del secondo programma come uscita del primo. Ciò produce un effetto simile all'utilizzo del carattere pipe del sistema operativo (|) su una riga di comando.

1. Avvia l’interprete Python nella cartella root
2. Digita il codice:

```
>>> import subprocess as sub
>>> sub.Popen(['ls']) # Windows use: ("cmd /c dir /b")
<subprocess.Popen object at 0x7fd3eecc>
fileA.txt fileB.txt ls.txt
>>> # Windows use: ("cmd /c dir /b", stdout=sub.PIPE)
>>> ls = sub.Popen(['ls'], stdout=sub.PIPE)
>>> for f in ls.stdout: print(f)
...
b'fileA.txt\n'
```

```
b'fileB.txt\n'
b'ls.txt\n'
>>> ex = sub.Popen(['ex', 'test.txt'], stdin=sub.PIPE) # Not Windows
>>> ex.stdin.write(b'i\nthis is a test\n.\n') # Not Windows
19
>>> ex.stdin.write(b'wq\n') # Not Windows
3
>>>
1+ Stopped python3
>>> sub.Popen(['NonExistentFile'])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.2/subprocess.py", line 745, in __init__
restore_signals, start_new_session)
File "/usr/lib/python3.2/subprocess.py", line 1361, in _execute_child
raise child_exception_type(errno_num, err_msg)
OSError: [Errno 2] No such file or directory: 'NonExistentFile'
```

Per iniziare, hai importato il subprocess con l'alias sub. Il primo paio di comandi ha semplicemente duplicato ciò che hai fatto con subprocess.call() in quanto inizialmente hai prodotto un elenco di file su stdout, ma di nuovo non è stato possibile utilizzare quei dati. Il secondo caso è più interessante perché hai reindirizzato stdout a un sub.PIPE che ti ha permesso di leggerlo tramite l'attributo stdout. Nota la differenza tra il parametro stdout, che hai impostato su sub.PIPE all'interno della chiamata al costruttore Popen e l'attributo stdout che usi per leggere i dati dall'istanza Popen e a cui si accede tramite la dot notation.

Per usarlo, hai anche assegnato il risultato della chiamata di Popen a una variabile chiamata ls. Popen è in realtà una classe e il risultato della chiamata è una nuova istanza dell'oggetto Popen. Il risultato finale è molto simile al caso subprocess.call() in cui hai inviato l'output a un file, ls.txt, e hai letto il file ma in questo caso non crei alcun file. Piuttosto, leggi direttamente dal processo, ciò significa che non finirai per ingombrare il tuo file system con piccoli file temporanei che devono essere riordinati in seguito. L'esempio successivo utilizzava l'editor di riga UNIX ex, ma questa volta hai reindirizzato stdin a sub.PIPE e poi hai inserito alcuni comandi

nell'editor per creare un breve file di testo. Hai anche passato un nome di file come argomento fornendo a Popen una seconda stringa nel primo argomento. Si noti che l'input per stdin deve essere una stringa di byte (b'xxxx') anziché una normale stringa di testo.

Il penultimo esempio mostra cosa succede se si tenta di aprire un file inesistente (o non si forniscono le informazioni corrette sul percorso). Ottieni un'eccezione OSError che potresti, ovviamente, catturare usando la struttura try/except di Python.

Negli esempi hai avuto accesso direttamente a stdin e stdout; tuttavia, ciò a volte può causare problemi, soprattutto quando si eseguono processi contemporaneamente o all'interno di thread, portando al riempimento dei pipe e al blocco del processo. Per evitare questi problemi, si consiglia di utilizzare il metodo Popen.communicate() e di indicizzare il flusso appropriato. Questo è leggermente più complesso da usare, ma evita i problemi appena menzionati. La funzione Popen.communicate() prende una stringa di input (equivalente a stdin) e restituisce una tupla con il primo elemento che è il contenuto di stdout e il secondo il contenuto di stderr. Quindi, ripetendo l'esempio di elenco dei file usando Popen.communicate() si presenta così:

```
>>> ls = sub.Popen(['ls'], stdout=sub.PIPE)
>>> lsout = ls.communicate()[0]
>>> print (lsout)
b'fileA.txt\nfileB.txt\nls.txt\n'
>>>
```

Per concludere questa sezione, vale la pena sottolineare che, per semplicità, negli esempi sono stati utilizzati comandi abbastanza basilari, come ls. Molti di questi comandi possono essere eseguiti in modo equivalente dall'interno di Python stesso. Il vero valore di meccanismi come subprocess.call() e Popen() sta nell'esecuzione di programmi molto più complessi come strumenti di conversione file e strumenti batch per l'elaborazione di immagini. Scrivere la funzionalità equivalente di questi strumenti in Python sarebbe un progetto importante; quindi, chiamare il programma esterno è un'alternativa molto più sensata. Dovresti usare Python negli scenari in cui è maggiormente adatto, nell'orchestrare e

convalidare gli input e gli output, ma lascia fare il lavoro sporco alle applicazioni dedicate.

Informazioni dai file

Il modulo os è fortemente influenzato dal modo di fare le cose in UNIX. In quanto tale, tratta dispositivi e file in modo simile. Quindi potresti scoprire dispositivi assomiglia molto a scoprire i file. In questa sezione vedrai ora come determinare lo stato e le autorizzazioni dei file e persino come modificare alcune delle loro proprietà dall'interno dei tuoi programmi. Considera il seguente codice:

```
>>> import os
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'ls.txt', 'test.txt']
>>> os.stat('fileA.txt')
posix.stat_result(st_mode=33204, st_ino=1125899907117103,
st_dev=1491519654, st_nlink=1, st_uid=1001, st_gid=513,
st_size=257, st_atime=1388676837, st_mtime=1388677418,
st_ctime=1388677418)
```

Qui hai controllato l'elenco della directory corrente ('.') con os.listdir(), ora che hai visto os.listdir(), si spera che ti renda conto che il tuo uso di ls o dir in subprocess è stato piuttosto artificiale perché os.listdir() fa lo stesso lavoro direttamente da Python, e lo fa in modo più efficiente. Hai quindi utilizzato la funzione os.stat() per ottenere alcune informazioni su uno dei file, questa funzione restituisce un oggetto tupla con un nome che contiene 10 elementi di interesse. Forse i più utili di questi sono st_uid, st_size e st_mtime. Questi valori rappresentano l'ID utente del proprietario del file, la dimensione e la data/ora dell'ultima modifica. Gli orari sono numeri interi che devono essere decodificati utilizzando il modulo time, in questo modo:

```
>>> import time
>>> time.localtime(1388677418)
time.struct_time(tm_year=2014, tm_mon=1, tm_mday=2, tm_hour=15,
tm_min=43, tm_sec=38, tm_wday=3, tm_yday=2, tm_isdst=0)
>>> time.strftime("%Y-%m-%d", time.localtime(1388677418))
'2014-01-02'
```

Qui hai usato la funzione localtime() del modulo time per convertire il valore intero st_mtime in una tupla che mostra i valori dell'ora locale e da lì in una stringa di data leggibile usando la funzione time.strftime() con una stringa di formato adatta. La tupla a 10 valori restituita da os.stat() è generalmente utile, ma maggiori dettagli sono disponibili tramite os.stat(). Alcuni di questi valori aggiuntivi dipendono dal sistema operativo, come l'attributo st_obtype che si trova sui sistemi RiscOS. Devi fare un po' più di lavoro per estrarli ma è possibile accedere ai dettagli utilizzando la notazione punto sull'attributo dell'oggetto. Forse il campo più interessante a cui puoi accedere da os.stat() è il valore st_mode, che ti dice i permessi di accesso del file. Lo usi in questo modo:

```
>>> import os
>>> stats = os.stat('fileA.txt')
>>> stats.st_mode
33204
```

Ma non è troppo utile; è solo un numero apparentemente casuale! Il segreto sta nei singoli bit che compongono il numero; è un'altra maschera di bit. Potresti ricordare la maschera di bit umask che hai visto in precedenza. La st_mode è concettualmente simile alla umask, ma con i significati dei bit invertiti. Puoi vedere come vengono codificati i dettagli di accesso guardando gli ultimi 9 bit, in questo modo:

```
>>> bin(stats.st_mode)[-9:]
'111111101'
```

Usando la funzione bin() in combinazione con slice, hai estratto la rappresentazione binaria degli ultimi 9 bit. Guardandoli come 3 gruppi di 3, puoi vedere i valori di lettura/scrittura/esecuzione rispettivamente per Proprietario, Gruppo e Mondo. Pertanto, in questo esempio, Owner e Group hanno tutti e tre i bit impostati su 1 (True), ma World ha solo i bit di lettura ed esecuzione impostati su 1 (True) e l'accesso in scrittura è 0 (False). Nota che questi sono l'inverso diretto dei significati dei bit umask; non confondere i due! Anche i bit di ordine superiore hanno significato e il modulo stat contiene una serie di maschere di bit che possono essere

utilizzate per estrarre i dettagli su un base bit per bit. Per la maggior parte degli scopi, i bit di accesso precedenti sono sufficienti e nel modulo os.path esistono funzioni di supporto che consentono di accedere a tali informazioni.

Hai anche molti altri modi per determinare i diritti di accesso a un file in Python. In particolare il modulo os fornisce una funzione utile — os.access() — che accetta un nome di file e una variabile flag (una tra os.F_OK, os.R_OK, os.W_OK o os.X_OK) che restituisce un risultato booleano a seconda che il file esista o sia rispettivamente leggibile, scrivibile o eseguibile. Queste funzioni sono tutte più facili da usare rispetto all'approccio sottostante os.stat() e bitmask, ma è utile sapere da dove le funzioni ottengono i loro dati. Infine, la documentazione del sistema operativo evidenzia un potenziale problema durante il controllo dell'accesso prima di aprire un file.

C'è un periodo molto breve tra le due operazioni in cui il file potrebbe cambiare il suo livello di accesso o il suo contenuto. Quindi, come al solito in Python, è meglio usare try/except per aprire il file e gestire l'errore se si verifica. È quindi possibile utilizzare i controlli di accesso per determinare la causa dell'errore, se necessario. Lo schema consigliato si presenta così:

```
try:
myfile = open('myfile.txt')
except PermissionError:
# test/modify the permissions here
else:
# process the file here
finally:
# close the file here
```

Dopo aver visto come esplorare le proprietà dei singoli file, ora esaminiamo i meccanismi disponibili per il file system, leggere cartelle, copiare, spostare ed eliminare file e così via. Python fornisce funzioni integrate per l'apertura, la lettura e la scrittura di singoli file. Il modulo os aggiunge funzioni per manipolare i file come entità complete, ad esempio rinominare, eliminare e creare collegamenti sono tutti previsti. Tuttavia, il modulo os stesso fornisce solo metà delle funzioni utili quando si tratta di lavorare con

i file. L'altra metà è offerta dal modulo shutil e altri moduli di utilità che funzionano insieme a os.

In Python 3.4 è stato introdotto un nuovo modulo chiamato pathlib, che mira a fornire una vista orientata agli oggetti del file system. Tuttavia, pathlib è contrassegnato come provvisorio, il che significa che le interfacce potrebbero cambiare in modo significativo nelle versioni future o il modulo potrebbe anche essere rimosso dalla libreria. A causa di questa incertezza, pathlib non viene utilizzato in questa sezione. Si inizia con la lettura e la navigazione nel file system. Hai già visto come puoi usare os.listdir() per ottenere un elenco di directory e os.getcwd() per ottenere il nome della directory di lavoro corrente. Puoi usare os.mkdir() per creare una nuova directory e os.chdir() per navigare in una directory diversa.

1. Naviga ed entra nella cartella root
2. Avvia l'interprete ed esegui i comandi:

```
>>> import os
>>> cwd = os.getcwd()
>>> print (cwd)
/home/agauld/book/root
>>> os.listdir(cwd)
['fileA.txt', 'fileB.txt', 'ls.txt']
>>> os.mkdir('subdir')
>>> os.listdir(cwd)
['fileA.txt', 'fileB.txt', 'ls.txt', 'subdir']
>>> os.chdir('subdir')
>>> os.getcwd()
'/home/agauld/book/root/subdir'
>>> os.chdir('..')
>>> os.getcwd()
'/home/agauld/book/root'
```

Dopo aver importato os nella prima riga, hai memorizzato la directory corrente in cwd e poi l'hai stampata per confermare che eri dove pensavi di essere. Quindi ne hai elencato il contenuto. Successivamente, hai creato una

cartella chiamata subdir e hai verificato che lo spostamento è riuscito chiamando nuovamente os.getcwd() dalla nuova cartella. Alla fine, sei tornato alla directory precedente usando la scorciatoia '..' e ancora una volta hai verificato che funzionasse con os.getcwd(). Un problema con la funzione os.mkdir() utilizzata qui è che può creare solo una directory in una directory esistente. Se provi a creare una directory in un luogo che non esiste, fallisce. Python fornisce una funzione alternativa chiamata os.makedirs()—attenzione alla differenza di ortografia—che crea tutte le cartelle intermedie in un percorso se non esistono già. Puoi vedere come funziona con i seguenti comandi:

```
>>> os.mkdir('test2/newtestdir')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory: 'test2/newtestdir'
>>> os.makedirs('test2/newtestdir')
>>> os.chdir('test2/newtestdir')
>>> print( os.getcwd() )
/home/agauld/book/root/test2/newtestdir
```

Qui la chiamata originale os.mkdir() ha prodotto un errore perché la cartella intermedia test2 non esisteva. La chiamata a os.makedirs() è riuscita, tuttavia, creando sia le cartelle test2 che newtestdir e sei stato in grado di passare a newtestdir per dimostrare il punto. Nota che os.makedirs() genera un errore se la cartella di destinazione esiste già. Puoi utilizzare un paio di parametri aggiuntivi per ottimizzare ulteriormente il comportamento, ma i valori predefiniti sono in genere ciò di cui hai bisogno. Un altro modulo, shutil, fornisce una serie di comandi di manipolazione dei file di livello superiore. Questi includono la possibilità di copiare singoli file, copiare interi alberi di directory, eliminare alberi di directory e spostare file o interi alberi di directory. Un'anomalia è la capacità di eliminare un singolo file o un gruppo di file. Questo si trova effettivamente nel modulo os sotto forma della funzione os.remove() per i file (e os.rmdir() per le directory vuote, sebbene shutil.rmtree() sia più potente e di solito quello che vuoi). Un altro modulo utile è glob. Questo modulo fornisce la gestione dei caratteri jolly del nome file. Probabilmente hai familiarità con il ? e * caratteri jolly utilizzati per specificare gruppi di file nei comandi del sistema operativo.

Ad esempio, *.exe specifica tutti i file che terminano con .exe. La funzione glob.glob() fa la stessa cosa nel tuo codice restituendo un elenco dei nomi di file corrispondenti per un dato modello.

Usi le funzioni di os, glob e shutil per manipolare interi file. Segui questi passi:

1. Naviga nella cartella root
2. Crea un file con nome test.py, non importa il suo contenuto
3. Avvia l'interprete e digita:

```
>>> import os,glob,shutil as sh
>>> os.listdir('.') # everything in the folder
['fileA.txt', 'fileB.txt', 'ls.txt', 'subdir', 'test.py', 'test2']
>>> glob.glob('*') # everything in the folder
['fileA.txt', 'fileB.txt', 'ls.txt', 'subdir', 'test.py', 'test2']
>>> glob.glob('*.*') # files with an extension
['fileA.txt', 'fileB.txt', 'ls.txt', 'test.py']
>>> glob.glob('*.txt') # text files only
['fileA.txt', 'fileB.txt', 'ls.txt']
>>> glob.glob('file?.txt') # text files starting with 'file'
['fileA.txt', 'fileB.txt']
>>> glob.glob('*.??') # any file with a 2 letter extension
['test.py']
```

1. Osserva da vicino i diversi set di risultati per vedere l'effetto delle diverse combinazioni di funzione/argomento.
2. Digita il seguente codice:

```
>>> sh.copy('fileA.txt','fileX.txt')
>>> sh.copy('fileB.txt','subdir/fileY.txt')
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'fileX.txt', 'ls.txt', 'subdir', 'test.py', 'test2']
>>> os.listdir('subdir')
['fileY.txt']
>>> sh.copytree('subdir', 'test3')
```

```
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'fileX.txt', 'ls.txt', 'subdir', 'test.py',
'test2', 'test3']
>>> os.listdir('test3')
['fileY.txt']
>>> sh.rmtree('test2')
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'fileX.txt', 'ls.txt', 'subdir', 'test.py', 'test3']
```

1. Esamina l'output dei comandi appena digitati prima di digitare il codice seguente:

```
>>> os.mkdir('test4')
>>> sh.move('subdir/fileY.txt', 'test4')
>>> os.listdir('test4')
['fileY.txt']
>>> os.listdir('subdir')
[]
>>> os.remove('test4/fileY.txt')
>>> os.listdir('test4')
[]
>>> os.remove('test4')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
OSError: [Errno 1] Operation not permitted: 'test4'
>>> sh.rmtree('test4')
>>> sh.rmtree('test3')
>>> os.remove('fileX.txt')
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'ls.txt', 'subdir', 'test.py']
```

Nella prima serie di comandi, hai confrontato l'effetto di os.listdir() con vari modelli nella funzione glob.glob(). Il primo modello ('*') ha replicato ciò che ha fatto os.listdir() elencando l'intero contenuto della cartella. Il

secondo modello ('*.*') elencava tutti i file con un'estensione. A rigor di termini glob() non sa nulla di file o cartelle; funziona rigorosamente con i nomi, quindi in realtà elenca tutti i nomi che avevano un punto incluso indipendentemente dal tipo di oggetto. In seguito, hai usato '*.txt' per trovare tutti i file di testo seguiti da 'file?.txt' per trovare qualsiasi file “txt” il cui nome inizia con file seguito da un singolo carattere. Infine, hai usato una combinazione di caratteri jolly per trovare qualsiasi file il cui nome termina con due caratteri ('*.??'). La seconda serie di comandi ha esaminato i comandi di manipolazione dei file di shutil. Hai iniziato usando shutil.copy() per copiare singoli file: fileA.txt in un nuovo file nella stessa cartella chiamata fileX.txt e fileB.txt nella sottodirectory subdir con un nuovo nome fileY.txt. Hai quindi utilizzato os.listdir() due volte per visualizzare i risultati in ciascuna cartella. Successivamente hai esaminato le operazioni a livello di directory con la funzione shutil.copytree() che ha copiato la directory subdir e il suo contenuto in una nuova cartella, test3, creandola nel processo. Ancora una volta, hai usato os.listdir() due volte per confermare i risultati. Hai usato shutil.rmtree() per eliminare la cartella test2 e il suo contenuto.

Hai iniziato la sequenza successiva creando un'altra nuova cartella, test4, usando os.mkdir(). In questa nuova cartella, hai spostato il file fileY.txt usando sh.move() e anche in questo caso, l'uso di os.listdir() su entrambe le cartelle dimostra che l'operazione è andata a buon fine e il file non esiste più in subdir ma esiste in test4. Infine, hai esaminato la rimozione dei file con os.remove(). Il primo esempio ha rimosso il file da test4 e verificato che fosse stato cancellato. La riga successiva ha tentato di rimuovere test4 stesso, ma ha prodotto un errore perché os.remove() funziona solo su file. Se hai bisogno di rimuovere una directory, devi usare di nuovo shutil.rmtree() (o potresti aver usato la funzione os.rmdir() per lo stesso scopo). Per ultimo usando os.listdir() ancora una volta hai avuto la conferma che la cartella era sparita.

CONCLUSIONE

Imparare a programmare è difficile. Non credere a chi ti dice che è semplice. La grande domanda da porsi è "Vuoi imparare Python?" Certamente, questo libro ti ha aiutato a rispondere a questa domanda, ma preparati a lottare, a fare errori, a sentirti frustrato e a sbattere la testa contro la tastiera.

Adesso tocca a te approfondire gli altri aspetti di Python, scoprire librerie interessanti, creare applicazioni e tanto altro. Se ti piace risolvere problemi difficili, se ti piacciono i Sudoku, le parole crociate o i puzzle, in altre parole, se ti piacciono le sfide, allora potresti amare la programmazione in Python. Non è facile, ma è abbastanza facile iniziare, puoi usare questo e-book quando vuoi per rivedere funzioni o chiarire dei dubbi.

Sii paziente con te stesso e fai finta di essere l'insegnante e lo studente allo stesso tempo. Quando lo studente fallisce, il buon insegnante non insulta o castiga ma incoraggia. Il buon insegnante sa che il fallimento fa parte dell'apprendimento e che superare il fallimento è gratificante e porta ad un maggiore apprendimento. Quindi, sii un buon insegnante.

Ma cerca di essere anche un bravo studente. Quando rileggi questo libro, dovresti essere seduto accanto al tuo computer. Poniti un obiettivo, ad esempio, pianifica di trascorrere almeno due ore ininterrotte o pianifica di risolvere un semplice task all’inizio. Questo processo ti porterà attraverso l'apprendimento di base e da lì in poi, sarà un continuo passaggio dal libro alla visualizzazione, modifica e scrittura del codice. Non si impara a programmare leggendo ma esercitandosi.

Quando hai iniziato a leggere questo e-book non avevi, forse, alcuna conoscenza precedente di Python ma adesso che conosci le basi potrai affrontare argomenti piuttosto avanzati come lavorare con la programmazione orientata agli oggetti, i database o usare Python per il Web.

Adesso è tutto nelle tue mani, hai delle basi, fanne buon uso.