Capitolo 9

Esercitazione: Giochi con le parole

Questo capitolo contiene la seconda esercitazione, in cui dovrete risolvere dei quesiti che consistono nel ricercare parole che hanno delle particolari proprietà. Ad esempio, cercherete i più lunghi palindromi della lingua inglese e le parole le cui lettere sono disposte in ordine alfabetico. Illustrerò anche un'altra tecnica di sviluppo: la riduzione ad un problema già risolto.

9.1 Leggere elenchi di parole

Per gli esercizi di questo capitolo ci serve un elenco di parole in inglese. Ci sono parecchi elenchi di parole disponibili sul Web, ma uno dei più adatti ai nostri scopi è quello raccolto da Grady Ward, di pubblico dominio, parte del progetto lessicale Moby (vedere `http://wikipedia.org/wiki/Moby_Project`). È un elenco di 113.809 parole ufficiali per cruciverba, cioè parole che sono considerate valide in un gioco di parole crociate o altri giochi con le parole. Nella raccolta Moby il nome del file è `113809of.fic`; potete anche scaricare una copia chiamata più semplicemente `words.txt`, dal sito `http://thinkpython2.com/code/words.txt`.

Il file è in testo semplice, e potete aprirlo con qualsiasi editor di testo, ma anche leggerlo con Python: la funzione predefinita `open` richiede come parametro il nome di un file e restituisce un **oggetto file** che potete utilizzare per questo scopo.

```
>>> fin = open('words.txt')
```

`fin` è un nome comunemente usato per un oggetto file usato per operazioni di input.

L'oggetto file comprende alcuni metodi di lettura, come `readline`, che legge i caratteri da un file finché non giunge ad un ritorno a capo, e restituisce il risultato sotto forma di stringa:

```
>>> fin.readline()
'aa\n'
```

La prima parola di questa speciale lista è "aa", che è un tipo di lava vulcanica. La sequenza `\n` rappresenta il carattere di ritorno a capo che separa questa parola dalla successiva.

L'oggetto file tiene traccia del punto in cui si trova all'interno del file, così quando chiamate nuovamente `readline`, ottenete la parola successiva:

```
>>> fin.readline()
'aah\n'
```

La parola successiva è "aah", che è perfettamente valida per cui non fate quella faccia! Oppure, se il carattere di ritorno a capo vi dà fastidio, potete sbarazzarvene con il metodo delle stringhe `strip`:

```
>>> riga = fin.readline()
>>> parola = riga.strip()
>>> parola
'aahed'
```

Potete anche usare un oggetto file all'interno di un ciclo `for`. Questo programma legge `words.txt` e stampa ogni parola, una per riga:

```
fin = open('words.txt')
for riga in fin:
    parola = riga.strip()
    print(parola)
```

9.2 Esercizi

Le soluzioni a questi esercizi sono discusse nel prossimo paragrafo. Tentate almeno di risolverli prima di continuare la lettura.

Esercizio 9.1. *Scrivete un programma che legga il file* `words.txt` *e stampi solo le parole composte da più di 20 caratteri (caratteri spaziatori esclusi).*

Esercizio 9.2. *Nel 1939, Ernest Vincent Wright pubblicò una novella di 50.000 parole dal titolo* Gadsby *che non conteneva alcuna lettera "e". Dato che la "e" è la lettera più comune nella lingua inglese, non è una cosa facile.*

Infatti, in italiano non ho mai composto un piccolo brano siffatto: sono pochi i vocaboli privi tali da riuscirci; finora non ho trovato alcun modo, ma conto di arrivarci in alcuni giorni, pur con un po' di difficoltà! Ma ora, basta così.

Scrivete una funzione di nome `niente_e` *che restituisca* `True` *se una data parola non contiene la lettera "e".*

Modificate il programma del paragrafo precedente in modo che stampi solo le parole dell'elenco prive della lettera "e", e ne calcoli la percentuale sul totale delle parole.

Esercizio 9.3. *Scrivete una funzione di nome* `evita` *che richieda una parola e una stringa di lettere vietate, e restituisca* `True` *se la parola non contiene alcuna lettera vietata.*

Modificate poi il programma in modo che chieda all'utente di inserire una stringa di lettere vietate, e poi stampi il numero di parole che non ne contengono alcuna. Riuscite a trovare una combinazione di 5 lettere vietate che escluda il più piccolo numero di parole?

Esercizio 9.4. *Scrivete una funzione di nome* `usa_solo` *che richieda una parola e una stringa di lettere, e che restituisca* `True` *se la parola contiene solo le lettere indicate. Riuscite a comporre una frase in inglese usando solo le lettere* `acefhlo`*? Diversa da "Hoe alfalfa"?*

Esercizio 9.5. *Scrivete una funzione di nome* `usa_tutte` *che richieda una parola e una stringa di lettere richieste e che restituisca* `True` *se la parola utilizza tutte le lettere richieste almeno una volta. Quante parole ci sono che usano tutte le vocali* `aeiou`*? E* `aeiouy`*?*

Esercizio 9.6. *Scrivete una funzione di nome* `alfabetica` *che restituisca* `True` *se le lettere di una parola compaiono in ordine alfabetico (le doppie valgono). Quante parole "alfabetiche" ci sono?*

9.3 Ricerca

Tutti gli esercizi del paragrafo precedente hanno qualcosa in comune: possono essere risolti con lo schema di ricerca che abbiamo visto nel Paragrafo 8.6. L'esempio più semplice è:

```
def niente_e(parola):
    for lettera in parola:
        if lettera == 'e':
            return False
    return True
```

Il ciclo `for` attraversa i caratteri in `parola`. Se trova la lettera "e", può immediatamente restituire `False`; altrimenti deve esaminare la lettera seguente. Se il ciclo termina normalmente, vuol dire che non è stata trovata alcuna "e", per cui il risultato è `True`.

Si potrebbe scrivere questa funzione in modo più conciso usando l'operatore `in`, ma ho preferito iniziare con questa versione perché dimostra la logica dello schema di ricerca.

`evita` è una versione più generale di `niente_e`, ma la struttura è la stessa:

```
def evita(parola, vietate):
    for lettera in parola:
        if lettera in vietate:
            return False
    return True
```

Possiamo restituire `False` appena troviamo una delle lettere vietate; se arriviamo alla fine del ciclo, viene restituito `True`.

`usa_solo` è simile, solo che il senso della condizione è invertito:

```
def usa_solo(parola, valide):
    for lettera in parola:
        if lettera not in valide:
            return False
    return True
```

Invece di un elenco di lettere vietate, ne abbiamo uno di lettere disponibili. Se in `parola` troviamo una lettera che non è una di quelle `valide`, possiamo restituire `False`.

`usa_tutte` è ancora simile, solo che rovesciamo il ruolo della parola e della stringa di lettere:

```
def usa_tutte(parola, richieste):
    for lettera in richieste:
        if lettera not in parola:
            return False
    return True
```

Invece di attraversare le lettere in `parola`, il ciclo attraversa le lettere richieste. Se una qualsiasi delle lettere richieste non compare nella parola, restituiamo `False`.

Ma se avete pensato davvero da informatici, avrete riconosciuto che `usa_tutte` era un'istanza di un problema già risolto in precedenza, e avrete scritto:

```
def usa_tutte(parola, richieste):
    return usa_solo(richieste, parola)
```

Ecco un esempio di metodo di sviluppo di un programma chiamato **riduzione ad un problema già risolto**, che significa che avete riconosciuto che il problema su cui state lavorando è un'istanza di un problema già risolto in precedenza, al quale potete applicare una soluzione che avevate già sviluppato.

9.4 Cicli con gli indici

Ho scritto le funzioni del paragrafo precedente utilizzando dei cicli `for` perché avevo bisogno solo dei caratteri nelle stringhe e non dovevo fare nulla con gli indici.

Per `alfabetica` dobbiamo comparare delle lettere adiacenti, che è un po' laborioso con un ciclo `for`:

```
def alfabetica(parola):
    precedente = parola[0]
    for c in parola:
        if c < precedente:
            return False
        precedente = c
    return True
```

Un'alternativa è usare la ricorsione:

```
def alfabetica(parola):
    if len(parola) <= 1:
        return True
    if parola[0] > parola[1]:
        return False
    return alfabetica(parola[1:])
```

E un'altra opzione è usare un ciclo `while`:

```
def alfabetica(parola):
    i = 0
    while i < len(parola)-1:
        if parola[i+1] < parola[i]:
            return False
        i = i+1
    return True
```

Il ciclo comincia da `i=0` e finisce a `i=len(parola)-1`. Ogni volta che viene eseguito, il ciclo confronta l' i-esimo carattere (consideratelo come il carattere attuale) con l' $i+1$-esimo carattere (consideratelo come quello successivo).

Se il carattere successivo è minore di quello attuale (cioè viene alfabeticamente prima), allora abbiamo scoperto un'interruzione nella serie alfabetica e la funzione restituisce `False`.

Se arriviamo a fine ciclo senza trovare difetti, la parola ha superato il test. Per convincervi che il ciclo è terminato correttamente, prendete un esempio come `'flossy'`. La lunghezza della parola è 6, quindi l'ultima ripetizione del ciclo si ha quando `i` è 4, che è l'indice del penultimo carattere. Nell'ultima iterazione, il penultimo carattere è comparato all'ultimo, che è quello che vogliamo.

Ecco una variante di `palindromo` (vedere l'Esercizio 6.3) che usa due indici; uno parte dall'inizio e aumenta, uno parte dalla fine e diminuisce.

```
def palindromo(parola):
    i = 0
    j = len(parola)-1

    while i<j:
        if parola[i] != parola[j]:
            return False
        i = i+1
        j = j-1

    return True
```

Oppure, possiamo ridurre ad un problema già risolto e scrivere:

```
def palindromo(parola):
    return al_contrario(parola, parola)
```

Usando `al_contrario` del Paragrafo 8.11.

9.5 Debug

Collaudare i programmi non è facile. Le funzioni di questo capitolo sono relativamente agevoli da provare, perché potete facilmente controllare il risultato da voi. Nonostante ciò, scegliere un insieme di parole che riescano a escludere ogni possibile errore è un qualcosa tra il difficile e l'impossibile.

Prendiamo ad esempio `niente_e`. Ci sono due evidenti casi da controllare: le parole che hanno una o più 'e' devono dare come risultato `False`; quelle che invece non hanno 'e', `True`. E fin qui, in un caso o nell'altro, non c'è niente di particolarmente difficile.

Per ciascun caso ci sono alcuni sottocasi meno ovvi. Tra le parole che contengono "e", dovreste provare parole che iniziano con "e", finiscono con "e", hanno "e" da qualche parte nel mezzo della parola. Dovreste poi provare parole lunghe, parole corte e parole cortissime. Nello specifico, la stringa vuota è un esempio di **caso particolare**, che è uno dei casi meno ovvi dove si nascondono spesso gli errori.

Oltre che con i casi da voi ideati, sarebbe anche bene fare un test del vostro programma con un elenco di parole come `words.txt`. Scansionando l'output potreste intercettare qualche errore, ma attenzione: può trattarsi di un certo tipo di errore (parole che non dovrebbero essere incluse ma invece ci sono) e non di un altro (parole che dovrebbero essere incluse ma non ci sono).

In linea generale, fare dei test può aiutarvi a trovare i bug, ma non è facile generare un buon insieme di casi di prova, e anche se ci riuscite non potete essere certi che il vostro programma sia corretto al 100 per cento.

Secondo un leggendario informatico:

> Il test di un programma può essere usato per dimostrare la presenza di bug, ma mai per dimostrarne l'assenza!
>
> — Edsger W. Dijkstra

9.6 Glossario

oggetto file: Un valore che rappresenta un file aperto.

riduzione ad un problema già risolto: Modo di risolvere un problema esprimendolo come un'istanza di un problema precedentemente risolto.

caso particolare: un caso atipico o non ovvio (e con meno probabilità di essere gestito correttamente) che viene testato.

9.7 Esercizi

Esercizio 9.7. *Questa domanda deriva da un quesito trasmesso nel programma radiofonico* Car Talk *(*`http://www.cartalk.com/content/puzzlers`*):*

> *"Ditemi una parola inglese con tre lettere doppie consecutive. Vi dò un paio di parole che andrebbero quasi bene, ma non del tutto. Per esempio la parola "committee", c-o-m-m-i-t-t-e-e. Sarebbe buona se non fosse per la "i" che si insinua in mezzo. O "Mississippi": M-i-s-s-i-s-s-i-p-p-i. Togliendo le "i" andrebbe bene. Ma esiste una parola che ha tre coppie di lettere uguali consecutive, e per quanto ne so dovrebbe essere l'unica. Magari ce ne sono altre 500, ma me ne viene in mente solo una. Qual è?"*

Scrivete un programma per trovare la parola. Soluzione: `http://thinkpython2.com/code/cartalk1.py`.

Esercizio 9.8. *Ecco un altro quesito di* Car Talk *(*`http://www.cartalk.com/content/puzzlers`*):*

> *"L'altro giorno stavo guidando in autostrada e guardai il mio contachilometri. È a sei cifre, come la maggior parte dei contachilometri, e mostra solo chilometri interi. Se la mia macchina, per esempio, avesse 300.000 km, vedrei 3-0-0-0-0-0."*
>
> *"Quello che vidi quel giorno era interessante. Notai che le ultime 4 cifre erano palindrome, cioè si potevano leggere in modo identico sia da sinistra a destra che viceversa. Per esempio 5-4-4-5 è palindromo, per cui il contachilometri avrebbe potuto essere 3-1-5-4-4-5"*
>
> *"Un chilometro dopo, gli ultimi 5 numeri erano palindromi. Per esempio potrei aver letto 3-6-5-4-5-6. Un altro chilometro dopo, le 4 cifre di mezzo erano palindrome. E tenetevi forte: un altro chilometro dopo tutte e 6 erano palindrome!"*
>
> *"La domanda è: quanto segnava il contachilometri la prima volta che guardai?"*

Scrivete un programma in Python che controlli tutti i numeri a sei cifre e visualizzi i numeri che soddisfano le condizioni sopra indicate. Soluzione: `http://thinkpython2.com/code/cartalk2.py`.

Esercizio 9.9. *Ecco un altro quesito di* Car Talk *(*`http://www.cartalk.com/content/puzzlers`*) che potete risolvere con una ricerca :*

> *"Di recente ho fatto visita a mia madre, e ci siamo accorti che le due cifre che compongono la mia età, invertite, formano la sua. Per esempio, se lei avesse 73 anni, io ne avrei 37. Ci siamo domandati quanto spesso succedesse questo negli anni, ma poi abbiamo divagato su altri discorsi senza darci una risposta."*
>
> *"Tornato a casa, ho calcolato che le cifre delle nostre età sono state sinora invertibili per sei volte. Ho calcolato anche che se fossimo fortunati succederebbe ancora tra pochi anni, e se fossimo veramente fortunati succederebbe un'altra volta ancora. In altre parole, potrebbe succedere per 8 volte in tutto. La domanda è: quanti anni ho io in questo momento?"*

Scrivete un programma in Python che ricerchi la soluzione a questo quesito. Suggerimento: potrebbe esservi utile il metodo delle stringhe `zfill`.

Soluzione: `http://thinkpython2.com/code/cartalk3.py`.

Capitolo 10

Liste

Questo capitolo illustra uno dei più utili tipi predefiniti di Python, le liste. Imparerete anche altri dettagli sugli oggetti, e vedrete cosa succede in presenza di uno stesso oggetto con più nomi.

10.1 Una lista è una sequenza

Come una stringa, una **lista** è una sequenza di valori. Mentre in una stringa i valori sono dei caratteri, in una lista possono essere di qualsiasi tipo. I valori che fanno parte della lista sono chiamati **elementi**.

Ci sono vari modi per creare una nuova lista; quello più semplice consiste nel racchiudere i suoi elementi tra parentesi quadrate (`[` e `]`):

```
[10, 20, 30, 40]
['Primi piatti', 'Secondi piatti', 'Dessert']
```

Il primo esempio è una lista di quattro interi; il secondo è una lista di tre stringhe. Non è necessario che gli elementi di una stessa lista siano tutti dello stesso tipo: la lista che segue contiene una stringa, un numero in virgola mobile, un intero e (meraviglia!) un'altra lista:

```
['spam', 2.0, 5, [10, 20]]
```

Una lista all'interno di un'altra lista è detta lista **nidificata**.

Una lista che non contiene elementi è detta lista vuota; potete crearne una scrivendo le due parentesi quadre vuote, `[]`.

Avrete già intuito che potete assegnare i valori della lista a variabili:

```
>>> formaggi = ['Cheddar', 'Edam', 'Gouda']
>>> numeri = [42, 123]
>>> vuota = []
>>> print(formaggi, numeri, vuota)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

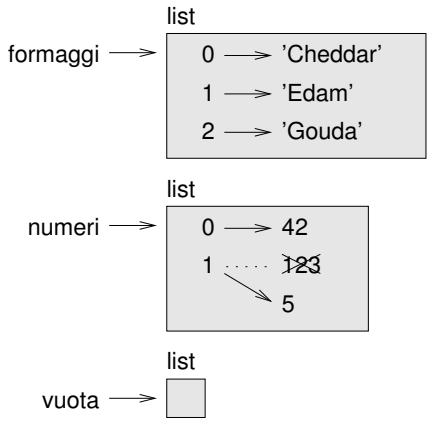

Figura 10.1: Diagramma di stato

10.2 Le liste sono mutabili

La sintassi per accedere agli elementi di una lista è la stessa usata per i caratteri di una stringa: le parentesi quadre, con un'espressione tra parentesi che specifica l'indice dell'elemento. E ricordate che gli indici partono da 0:

```
>>> formaggi[0]
'Cheddar'
```

Ma a differenza delle stringhe, le liste sono mutabili. Quando l'operatore parentesi quadre compare sul lato sinistro di un'assegnazione, identifica l'elemento della lista che sarà riassegnato:

```
>>> numeri = [42, 123]
>>> numeri[1] = 5
>>> numeri
[42, 5]
```

L'elemento di indice 1 di `numeri`, che era 123, ora è 5

La Figura 10.1 mostra il diagramma di stato di `formaggi`, `numeri` e `vuota`:

Le liste possono essere rappresentate da riquadri con la parola "list" all'esterno e i suoi elementi all'interno. `formaggi` si riferisce a una lista con tre elementi di indice 0, 1 e 2. `numeri` contiene due elementi; il diagramma mostra che il valore del secondo elemento è stato riassegnato da 123 a 5. `vuota` si riferisce a una lista senza elementi.

Gli indici delle liste funzionano nello stesso modo di quelli delle stringhe:

- L'indice può essere qualsiasi espressione di tipo intero.
- Se tentate di leggere o modificare un elemento che non esiste, ottenete un messaggio d'errore `IndexError`.
- Con un indice di valore negativo, si conta a ritroso dalla fine della lista.

Anche l'operatore `in` funziona con le liste:

```
>>> formaggi = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in formaggi
True
>>> 'Brie' in formaggi
False
```

10.3 Attraversamento di una lista

Il modo più frequente di attraversare gli elementi di una lista è un ciclo `for`. Anche qui, la sintassi è la stessa delle stringhe:

```
for formaggio in formaggi:
    print(formaggio)
```

Questo metodo funziona bene per leggere gli elementi di una lista, ma se volete scrivere o aggiornare degli elementi vi servono gli indici. Un modo per farlo è usare una combinazione delle funzioni predefinite `range` e `len`:

```
for i in range(len(numeri)):
    numeri[i] = numeri[i] * 2
```

Questo ciclo attraversa la lista e aggiorna tutti gli elementi. `len` restituisce il numero di elementi della lista. `range` restituisce una lista di indici da 0 a $n - 1$, dove n è la lunghezza della lista. Ad ogni ripetizione del ciclo, `i` prende l'indice dell'elemento successivo. L'istruzione di assegnazione nel corpo usa `i` per leggere il vecchio valore dell'elemento e assegnare quello nuovo.

Un ciclo `for` su una lista vuota non esegue mai il corpo:

```
for x in []:
    print('Questo non succede mai.')
```

Sebbene una lista possa contenerne un'altra, quella nidificata conta sempre come un singolo elemento. Quindi la lunghezza di questa lista è quattro:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4 Operazioni sulle liste

L'operatore + concatena delle liste:

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

L'operatore `*` ripete una lista per un dato numero di volte:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Il primo esempio ripete `[0]` per quattro volte. Il secondo ripete la lista `[1, 2, 3]` per tre volte.

10.5 Slicing delle liste

Anche l'operazione di slicing funziona sulle liste:

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

Se omettete il primo indice, lo slicing comincia dall'inizio, mentre se manca il secondo, termina alla fine. Se vengono omessi entrambi, lo slicing è una copia dell'intera lista.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Dato che le liste sono mutabili, spesso è utile farne una copia prima di eseguire operazioni che le modificano.

Un operatore di slicing sul lato sinistro di un'assegnazione, permette di aggiornare più elementi.

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

10.6 Metodi delle liste

Python fornisce dei metodi che operano sulle liste. Ad esempio, `append` aggiunge un nuovo elemento in coda alla lista:

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

`extend` prende una lista come argomento e accoda tutti i suoi elementi:

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

Questo esempio lascia immutata la lista `t2`.

`sort` dispone gli elementi della lista in ordine crescente:

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

La maggior parte dei metodi delle liste sono vuoti: modificano la lista e restituiscono `None`. Se scrivete accidentalmente `t = t.sort()`, il risultato vi deluderà.

10.7 Mappare, filtrare e ridurre

Per sommare tutti i numeri in una lista, potete usare un ciclo come questo:

```
def somma_tutti(t):
    totale = 0
    for x in t:
        totale += x
    return totale
```

`totale` è inizializzato a 0. Ad ogni ripetizione del ciclo, `x` prende un elemento dalla lista. L'operatore += è una forma abbreviata per aggiornare una variabile. Questa **istruzione di assegnazione potenziata**,

```
    totale += x
```

è equivalente a

```
    totale = totale + x
```

Man mano che il ciclo lavora, `totale` accumula la somma degli elementi; una variabile usata in questo modo è detta anche **accumulatore**.

Sommare gli elementi di una lista è un'operazione talmente comune che Python contiene una apposita funzione predefinita, `sum`:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

Una simile operazione che compatta una sequenza di elementi in un singolo valore, è chiamata **riduzione**.

Talvolta è necessario attraversare una lista per costruirne contemporaneamente un'altra. Per esempio, la funzione seguente prende una lista di stringhe e restituisce una nuova lista che contiene le stesse stringhe in lettere maiuscole:

```
def tutte_maiuscole(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` è inizializzata come una lista vuota; ad ogni ripetizione del ciclo viene accodato un elemento. Pertanto `res` è una sorta di accumulatore.

Un'operazione come quella di `tutte_maiuscole` è chiamata anche **mappa**: applica una funzione (in questo caso il metodo `capitalize`) su ciascun elemento di una sequenza.

Un'altra operazione frequente è la selezione di alcuni elementi di una lista per formare una sottolista. Per esempio, la seguente funzione prende una lista di stringhe e restituisce una lista che contiene solo le stringhe scritte in lettere maiuscole:

```
def solo_maiuscole(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` è un metodo delle stringhe che restituisce `True` se la stringa contiene solo lettere maiuscole.

Un'operazione come quella di `solo_maiuscole` è chiamata **filtro** perché seleziona solo alcuni elementi, filtrando gli altri.

La maggior parte delle operazioni sulle liste possono essere espresse come combinazioni di mappa, filtro e riduzione.

10.8 Cancellare elementi

Ci sono vari modi per cancellare elementi da una lista. Se conoscete l'indice dell'elemento desiderato, potete usare `pop`:

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

`pop` modifica la lista e restituisce l'elemento che è stato rimosso. Se omettete l'indice, il metodo cancella e restituisce l'ultimo elemento della lista.

Se non vi serve il valore rimosso, potete usare l'operatore `del`:

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

Se conoscete l'elemento da rimuovere ma non il suo indice, potete usare `remove`:

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

Il valore di ritorno di `remove` è `None`.

Per cancellare più di un elemento potete usare `del` con lo slicing:

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

Come di consueto, lo slicing seleziona gli elementi fino al secondo indice escluso.

10.9 Liste e stringhe

Una stringa è una sequenza di caratteri e una lista è una sequenza di valori, ma una lista di caratteri non è la stessa cosa di una stringa. Per convertire una stringa in una lista di caratteri, potete usare `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

Poiché `list` è una funzione predefinita, va evitato di chiamare una variabile con questo nome. Personalmente evito anche `l` perché somiglia troppo a `1`. Ecco perché di solito uso `t`.

La funzione `list` separa una stringa in singole lettere. Se invece volete spezzare una stringa nelle singole parole, usate il metodo `split`:

```
>>> s = 'profonda nostalgia dei fiordi'
>>> t = s.split()
>>> t
['profonda', 'nostalgia', 'dei', 'fiordi']
```

Un argomento opzionale chiamato **delimitatore** specifica quale carattere va considerato come separatore delle parole. L'esempio che segue usa il trattino come separatore:

```
>>> s = 'spam-spam-spam'
>>> delimita = '-'
>>> t = s.split(delimita)
>>> t
['spam', 'spam', 'spam']
```

`join` è l'inverso di `split`: prende una lista di stringhe e concatena gli elementi. `join` è un metodo delle stringhe, quindi lo dovete invocare per mezzo del delimitatore e passare la lista come parametro:

```
>>> t = ['profonda', 'nostalgia', 'dei', 'fiordi']
>>> delimita = ' '
>>> s = delimita.join(t)
>>> s
'profonda nostalgia dei fiordi'
```

In questo caso il delimitatore è uno spazio, quindi `join` aggiunge uno spazio tra le parole. Per concatenare delle stringhe senza spazi, basta usare come delimitatore la stringa vuota `''`.

10.10 Oggetti e valori

Se eseguiamo queste istruzioni di assegnazione:

```
a = 'banana'
b = 'banana'
```

Sappiamo che `a` e `b` si riferiscono a una stringa, ma non sappiamo se si riferiscono alla *stessa* stringa. Ci sono due possibili stati, illustrati in Figura 10.2.

In un caso, `a` e `b` si riferiscono a due oggetti diversi che hanno lo stesso valore. Nell'altro, si riferiscono allo stesso oggetto.

Per controllare se due variabili si riferiscono allo stesso oggetto, potete usare l'operatore `is`.

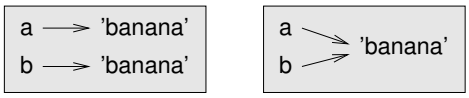

Figura 10.2: Diagramma di stato.

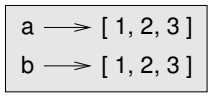

Figura 10.3: Diagramma di stato.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

In questo esempio, Python ha creato un unico oggetto stringa, e sia `a` che `b` fanno riferimento ad esso.

Ma se create due liste, ottenete due oggetti distinti:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

Quindi il diagramma di stato somiglia a quello di Figura 10.3.

In quest'ultimo caso si dice che le due liste sono **equivalenti**, perché contengono gli stessi elementi, ma non **identiche**, perché non sono lo stesso oggetto. Se due oggetti sono identici, sono anche equivalenti, ma se sono equivalenti non sono necessariamente identici.

Fino ad ora abbiamo usato "oggetto" e "valore" indifferentemente, ma è più preciso dire che un oggetto ha un valore. Se valutate `[1,2,3]`, ottenete un oggetto lista il cui valore è una sequenza di interi. Se un'altra lista contiene gli stessi elementi, diciamo che ha lo stesso valore, ma non che è lo stesso oggetto.

10.11 Alias

Se la variabile `a` si riferisce a un oggetto e assegnate `b = a`, allora entrambe le variabili si riferiscono allo stesso oggetto.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

Il diagramma di stato è quello in Figura 10.4.

L'associazione tra una variabile e un oggetto è chiamato **riferimento**. In questo esempio ci sono due riferimenti allo stesso oggetto.

Un oggetto che ha più di un riferimento ha anche più di un nome, e si dice quindi che l'oggetto ha degli **alias**.

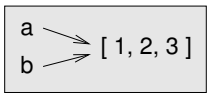

Figura 10.4: Diagramma di stato.

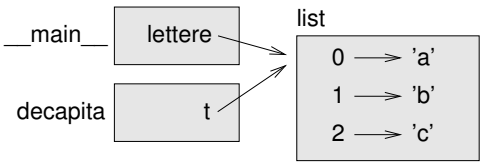

Figura 10.5: Diagramma di stack.

Se l'oggetto munito di alias è mutabile, i cambiamenti provocati da un alias si riflettono anche sull'altro:

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

Sebbene questo comportamento possa essere utile, è anche fonte di errori. In genere è più sicuro evitare gli alias quando si sta lavorando con oggetti mutabili.

Per gli oggetti immutabili come le stringhe, gli alias non sono un problema. In questo esempio:

```
a = 'banana'
b = 'banana'
```

Non fa quasi mai differenza se `a` e `b` facciano riferimento alla stessa stringa o meno.

10.12 Liste come argomenti

Quando passate una lista a una funzione, questa riceve un riferimento alla lista. Se la funzione modifica la lista, il chiamante vede la modifica. Per esempio, `decapita` rimuove il primo elemento di una lista:

```
def decapita(t):
    del t[0]
```

Vediamo come si usa:

```
>>> lettere = ['a', 'b', 'c']
>>> decapita(lettere)
>>> lettere
['b', 'c']
```

Il parametro `t` e la variabile `lettere` sono due alias dello stesso oggetto. Il diagramma di stack è riportato in Figura 10.5.

Dato che la lista è condivisa da due frame, la disegno in mezzo.

È importante distinguere tra operazioni che modificano le liste e operazioni che creano nuove liste. Per esempio il metodo `append` modifica una lista, ma l'operatore + ne crea una nuova.

Ecco un esempio che usa `append`:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

Il valore di ritorno di `append` è `None`.

Un esempio di utilizzo dell'operatore +:

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
```

Il risultato è una nuova lista, e la lista di origine resta immutata.

Questa differenza è importante quando scrivete delle funzioni che devono modificare delle liste. Per esempio, questa funzione *non* cancella il primo elemento della lista:

```
def non_decapita(t):
    t = t[1:]              # SBAGLIATO!
```

L'operatore di slicing crea una nuova lista e l'assegnazione fa in modo che `t` si riferisca ad essa, ma tutto ciò non ha effetti sul chiamante.

```
>>> t4 = [1, 2, 3]
>>> non_decapita(t4)
>>> t4
[1, 2, 3]
```

Alla chiamata di `non_decapita`, `t` e `t4` fanno riferimento alla stessa lista. Alla fine, `t` fa riferimento ad una nuova lista, mentre `t4` continua a fare riferimento alla stessa lista, non modificata.

Un'alternativa valida è scrivere una funzione che crea e restituisce una nuova lista. Per esempio, `ritaglia` restituisce tutti gli elementi di una lista tranne il primo:

```
def ritaglia(t):
    return t[1:]
```

Questa funzione lascia intatta la lista di origine. Ecco come si usa:

```
>>> lettere = ['a', 'b', 'c']
>>> resto = ritaglia(lettere)
>>> resto
['b', 'c']
```

10.13 Debug

Un uso poco accurato delle liste (e degli altri oggetti mutabili) può portare a lunghe ore di debug. Ecco alcune delle trappole più comuni e i modi per evitarle:

1. La maggior parte dei metodi delle liste modificano l'argomento e restituiscono `None`. È il comportamento opposto dei metodi delle stringhe, che restituiscono una nuova stringa e lasciano immutato l'originale.

   Se siete abituati a scrivere il codice per le stringhe così:

   ```
   parola = parola.strip()
   ```

   Può venire spontaneo di scrivere il codice per le liste così:

   ```
   t = t.sort()            # SBAGLIATO!
   ```

   Ma poiché `sort` restituisce `None`, l'operazione successiva che eseguite su `t` con tutta probabilità fallirà.

   Prima di usare i metodi delle liste e gli operatori, leggetene attentamente la documentazione e fate una prova in modalità interattiva.

2. Scegliete un costrutto e usate sempre quello.

   Una parte dei problemi delle liste deriva dal fatto che ci sono molti modi per fare le stesse cose. Per esempio, per rimuovere un elemento da una lista potete usare `pop`, `remove`, `del`, oppure lo slicing.

   Per aggiungere un elemento potete usare il metodo `append` o l'operatore `+`. Supponendo che `t` sia una lista e `x` un elemento, le espressioni seguenti vanno entrambe bene:

   ```
   t.append(x)
   t = t + [x]
   ```

   Mentre queste sono sbagliate:

   ```
   t.append([x])           # SBAGLIATO!
   t = t.append(x)         # SBAGLIATO!
   t + [x]                 # SBAGLIATO!
   t = t + x               # SBAGLIATO!
   ```

   Provate ognuno di questi esempi in modalità interattiva per verificare quello che fanno. Noterete che solo l'ultima espressione causa un errore di esecuzione; le altre sono consentite, ma fanno la cosa sbagliata.

3. Fate copie per evitare gli alias.

   Se volete usare un metodo come `sort` che modifica l'argomento, ma anche mantenere inalterata la lista di origine, potete farne una copia.

   ```
   >>> t = [3, 1, 2]
   >>> t2 = t[:]
   >>> t2.sort()
   >>> t
   [3, 1, 2]
   >>> t2
   [1, 2, 3]
   ```

   In questo esempio, si può anche usare la funzione predefinita `sorted`, che restituisce una nuova lista ordinata e lascia intatta quella di origine.

   ```
   >>> t2 = sorted(t)
   >>> t
   [3, 1, 2]
   >>> t2
   [1, 2, 3]
   ```

10.14 Glossario

lista: Una sequenza di valori.

elemento: Uno dei valori in una lista (o in altri tipi di sequenza).

lista nidificata: Lista che è contenuta come elemento in un'altra lista.

accumulatore: Variabile usata in un ciclo per sommare cumulativamente un risultato.

assegnazione potenziata: Istruzione che aggiorna un valore di una variabile usando un operatore come `+=`.

riduzione: Schema di calcolo che attraversa una sequenza e ne accumula gli elementi in un singolo risultato.

mappa: Schema di calcolo che attraversa una sequenza ed esegue una stessa operazione su ciascun elemento della sequenza.

filtro: Schema di calcolo che attraversa una lista e seleziona solo gli elementi che soddisfano un dato criterio.

oggetto: Qualcosa a cui una variabile può fare riferimento. Un oggetto ha un tipo e un valore.

equivalente: Avente lo stesso valore.

identico: Essere lo stesso oggetto (implica anche l'equivalenza).

riferimento: L'associazione tra una variabile e il suo valore.

alias: Due o più variabili che si riferiscono allo stesso oggetto, con nomi diversi.

delimitatore: Carattere o stringa usato per indicare i punti dove una stringa deve essere spezzata.

10.15 Esercizi

Potete scaricare le soluzioni degli esercizi seguenti all'indirizzo `http://thinkpython2.com/code/list_exercises.py`.

Esercizio 10.1. *Scrivete una funzione di nome* `somma_nidificata` *che prenda una lista di liste di numeri interi e sommi gli elementi di tutte le liste nidificate. Esempio:*

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> somma_nidificata(t)
21
```

Esercizio 10.2. *Scrivete una funzione di nome* `somma_cumulata` *che prenda una lista di numeri e restituisca la somma cumulata, cioè una nuova lista dove l'i-esimo elemento è la somma dei primi* $i + 1$ *elementi della lista di origine. Per esempio:*

```
>>> t = [1, 2, 3]
>>> somma_cumulata(t)
[1, 3, 6]
```

Esercizio 10.3. *Scrivete una funzione di nome* `mediani` *che prenda una lista e restituisca una nuova lista che contenga tutti gli elementi, esclusi il primo e l'ultimo. Esempio:*

```
>>> t = [1, 2, 3, 4]
>>> mediani(t)
[2, 3]
```

Esercizio 10.4. *Scrivete una funzione di nome* `tronca` *che prenda una lista, la modifichi togliendo il primo e l'ultimo elemento, e restituisca* `None`*. Esempio:*

```
>>> t = [1, 2, 3, 4]
>>> tronca(t)
>>> t
[2, 3]
```

Esercizio 10.5. *Scrivete una funzione di nome* `ordinata` *che prenda una lista come parametro e restituisca* `True` *se la lista è ordinata in senso crescente,* `False` *altrimenti. Esempio:*

```
>>> ordinata([1, 2, 2])
True
>>> ordinata(['b', 'a'])
False
```

Esercizio 10.6. *Due parole sono anagrammi se potete ottenerle riordinando le lettere di cui sono composte. Scrivete una funzione di nome* `anagramma` *che riceva due stringhe e restituisca* `True` *se sono anagrammi.*

Esercizio 10.7. *Scrivete una funzione di nome* `ha_duplicati` *che richieda una lista e restituisca* `True` *se contiene elementi che compaiono più di una volta. Non deve modificare la lista di origine.*

Esercizio 10.8. *Questo è un esercizio sul cosiddetto "Paradosso del compleanno"; potete approfondirlo leggendo* `http://it.wikipedia.org/wiki/Paradosso_del_compleanno`*.*

Se in una classe ci sono 23 studenti, quante probabilità ci sono che due di loro compiano gli anni lo stesso giorno? Potete stimare questa probabilità generando alcuni campioni a caso di 23 date e controllando le corrispondenze. Suggerimento: per generare date in modo casuale usate la funzione `randint` *nel modulo* `random`*.*

Potete scaricare la mia soluzione da `http://thinkpython2.com/code/birthday.py`*.*

Esercizio 10.9. *Scrivete una funzione che legga il file* `words.txt` *e crei una lista in cui ogni parola è un elemento. Scrivete due versioni della funzione, una che usi il metodo* `append` *e una il costrutto* `t = t + [x]`*. Quale richiede più tempo di esecuzione? Perché?*

Soluzione: `http://thinkpython2.com/code/wordlist.py`*.*

Esercizio 10.10. *Per controllare se una parola è contenuta in un elenco, è possibile usare l'operatore* `in`*, ma è un metodo lento, perché ricerca le parole seguendo il loro ordine.*

Dato che le parole sono in ordine alfabetico, possiamo accelerare l'operazione con una ricerca binaria (o per bisezione), che è un po' come cercare una parola nel vocabolario. Partite nel mezzo e controllate se la parola che cercate viene prima o dopo la parola di metà elenco. Se prima, cercherete nella prima metà nello stesso modo, se dopo, cercherete nella seconda metà.

Ad ogni passaggio, dimezzate lo spazio di ricerca. Se l'elenco ha 113.809 parole, ci vorranno circa 17 passaggi per trovare la parola o concludere che non c'è.

Scrivete una funzione di nome `bisezione` *che richieda una lista ordinata e un valore da ricercare, e restituisca* `True` *se la parola fa parte della lista, o* `False` *se non è presente.*

Oppure, potete leggere la documentazione del modulo `bisect` *e usare quello! Soluzione:* `http://thinkpython2.com/code/inlist.py`.

Esercizio 10.11. *Una coppia di parole è "bifronte" se l'una si legge nel verso opposto dell'altra. Scrivete un programma che trovi tutte le parole bifronti nella lista di parole. Soluzione:* `http://thinkpython2.com/code/reverse_pair.py`.

Esercizio 10.12. *Due parole si "incastrano" se, prendendo le loro lettere alternativamente dall'una e dall'altra, si forma una nuova parola. Per esempio, le parole inglesi "shoe" and "cold" incastrandosi formano "schooled".*

1. *Scrivete un programma che trovi tutte le coppie di parole che possono incastrarsi. Suggerimento: non elaborate tutte le coppie!*
2. *Riuscite a trovare dei gruppi di tre parole che possono incastrarsi tra loro? Cioè, tre parole da cui, prendendo le lettere una ad una alternativamente, nell'ordine, si formi una nuova parola? (Es. "ace", "bus" e "as" danno "abacuses")*

Soluzione: `http://thinkpython2.com/code/interlock.py`. *Fonte: Questo esercizio è tratto da un esempio di* `http://puzzlers.org`.

Capitolo 11

Dizionari

Questo capitolo illustra un altro tipo predefinito chiamato dizionario. I dizionari sono una delle migliori caratteristiche di Python; sono i mattoni che costituiscono molti eleganti ed efficienti algoritmi.

11.1 Un dizionario è una mappatura

Un **dizionario** è simile ad una lista, ma è più generico. Infatti, mentre in una lista gli indici devono essere numeri interi, in un dizionario possono essere (quasi) di ogni tipo.

Un dizionario contiene una raccolta di indici, chiamati **chiavi**, e una raccolta di valori. Ciascuna chiave è associata ad un unico valore. L'associazione tra una chiave e un valore è detta **coppia chiave-valore** o anche **elemento**.

In linguaggio matematico, un dizionario rappresenta una relazione di corrispondenza, o **mappatura**, da una chiave a un valore, e si può dire pertanto che ogni chiave "mappa in" un valore.

Come esempio, costruiamo un dizionario che trasforma le parole dall'inglese all'italiano, quindi chiavi e valori saranno tutte delle stringhe.

La funzione `dict` crea un nuovo dizionario privo di elementi. Siccome `dict` è il nome di una funzione predefinita, è meglio evitare di usarlo come nome di variabile.

```
>>> eng2it = dict()
>>> eng2it
{}
```

Le parentesi graffe, `{}`, rappresentano un dizionario vuoto. Per aggiungere elementi al dizionario, usate le parentesi quadre:

```
>>> eng2it['one'] = 'uno'
```

Questa riga crea un elemento che contiene una corrispondenza dalla chiave `'one'` al valore `'uno'`. Se stampiamo di nuovo il dizionario, vedremo ora una coppia chiave-valore separati da due punti:

```
>>> eng2it
{'one': 'uno'}
```

Questo formato di output può essere anche usato per gli inserimenti. Ad esempio potete creare un nuovo dizionario con tre elementi:

```
>>> eng2it = {'one': 'uno', 'two': 'due', 'three': 'tre'}
```

Se stampate ancora una volta `eng2it`, avrete una sorpresa:

```
>>> eng2it
{'one': 'uno', 'three': 'tre', 'two': 'due'}
```

L'ordine delle coppie chiave-valore non è necessariamente lo stesso. Se scrivete lo stesso esempio nel vostro computer, potreste ottenere un altro risultato ancora. In genere, l'ordine degli elementi di un dizionario è imprevedibile.

Ma questo non è un problema, perché gli elementi di un dizionario non sono indicizzati con degli indici numerici. Infatti, per cercare un valore si usano invece le chiavi:

```
>>> eng2it['two']
'due'
```

La chiave `'two'` corrisponde correttamente al valore `'due'` e l'ordine degli elementi nel dizionario è ininfluente.

Se la chiave non è contenuta nel dizionario, viene generato un errore::

```
>>> print(eng2it['four'])
KeyError: 'four'
```

La funzione `len` è applicabile ai dizionari, e restituisce il numero di coppie chiave-valore:

```
>>> len(eng2it)
3
```

Anche l'operatore `in` funziona con i dizionari: informa se qualcosa compare come *chiave* nel dizionario (non è condizione sufficiente che sia contenuto come valore).

```
>>> 'one' in eng2it
True
>>> 'uno' in eng2it
False
```

Per controllare invece se qualcosa compare come valore, potete usare il metodo `values`, che restituisce una raccolta dei valori, e quindi usare l'operatore `in`:

```
>>> vals = eng2it.values()
>>> 'uno' in vals
True
```

L'operatore `in` utilizza algoritmi diversi per liste e dizionari. Per le prime, ne ricerca gli elementi in base all'ordine, come nel Paragrafo 8.6. Se la lista si allunga, anche il tempo di ricerca si allunga in proporzione. Per i secondi, Python usa un algoritmo chiamato **tabella hash** che ha notevoli proprietà: l'operatore `in` impiega sempre circa lo stesso tempo, indipendentemente da quanti elementi contiene il dizionario. Rimando la spiegazione di come ciò sia possibile all'Appendice B.4: per capirla, occorre prima leggere qualche altro capitolo.

11.2 Il dizionario come raccolta di contatori

Supponiamo che vi venga data una stringa e che vogliate contare quante volte vi compare ciascuna lettera. Ci sono alcuni modi per farlo:

1. Potete creare 26 variabili, una per lettera dell'alfabeto. Quindi, fare un attraversamento della stringa e per ciascun carattere incrementate il contatore corrispondente, magari usando delle condizioni in serie.
2. Potete creare una lista di 26 elementi, quindi convertire ogni carattere in un numero (usando la funzione predefinita `ord`), utilizzare il numero come indice e incrementare il contatore corrispondente.
3. Potete creare un dizionario con i caratteri come chiavi e i contatori come valore corrispondente. La prima volta che incontrate un carattere, lo aggiungete come elemento al dizionario. Successivamente, incrementerete il valore dell'elemento esistente.

Ciascuna di queste opzioni esegue lo stesso calcolo, ma lo implementa in modo diverso.

Un'**implementazione** è un modo per effettuare un'elaborazione. Le implementazioni non sono tutte uguali, alcune sono migliori di altre: per esempio, un vantaggio dell'implementazione con il dizionario è che non serve sapere in anticipo quali lettere ci siano nella stringa e quali no, dobbiamo solo fare spazio per le lettere che compariranno effettivamente.

Ecco come potrebbe essere scritto il codice:

```
def istogramma(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

Il nome di questa funzione è `istogramma`, che è un termine statistico per indicare un insieme di contatori (o frequenze).

La prima riga della funzione crea un dizionario vuoto. Il ciclo `for` attraversa la stringa. Ad ogni ripetizione, se il carattere `c` non compare nel dizionario crea un nuovo elemento di chiave `c` e valore iniziale 1 (dato che incontra questa lettera per la prima volta). Se invece `c` è già presente, incrementa `d[c]` di una unità.

Vediamo come funziona:

```
>>> h = istogramma('brontosauro')
>>> h
{'a': 1, 'b': 1, 'o': 3, 'n': 1, 's': 1, 'r': 2, 'u': 1, 't': 1}
```

L'istogramma indica che le lettere `'a'` e `'b'` compaiono una volta, la `'o'` tre volte e così via.

I dizionari supportano il metodo `get` che richiede una chiave e un valore predefinito. Se la chiave è presente nel dizionario, `get` restituisce il suo valore corrispondente, altrimenti restituisce il valore predefinito. Per esempio:

```
>>> h = istogramma('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

Come esercizio, usate `get` per scrivere `istogramma` in modo più compatto. Dovreste riuscire a fare a meno dell'istruzione `if`.

11.3 Cicli e dizionari

Se usate un dizionario in un ciclo `for`, quest'ultimo attraversa le chiavi del dizionario. Per esempio, `stampa_isto` visualizza ciascuna chiave e il valore corrispondente:

```
def stampa_isto(h):
    for c in h:
        print(c, h[c])
```

Ecco come risulta l'output:

```
>>> h = istogramma('parrot')
>>> stampa_isto(h)
a 1
p 1
r 2
t 1
o 1
```

Di nuovo, le chiavi sono alla rinfusa. Per attraversare le chiavi disponendole in ordine, si può utilizzare la funzione predefinita `sorted`:

```
>>> for chiave in sorted(h):
...     print(chiave, h[chiave])
a 1
o 1
p 1
r 2
t 1
```

11.4 Lookup inverso

Dato un dizionario `d` e una chiave `k`, è facile trovare il valore corrispondente alla chiave: `v = d[k]`. Questa operazione è chiamata **lookup**.

Ma se invece volete trovare la chiave `k` conoscendo il valore `v`? Avete due problemi: primo, ci possono essere più chiavi che corrispondono al valore `v`. A seconda dell'applicazione, potete riuscire a trovarne uno, oppure può essere necessario ricavare una lista che li contenga tutti. Secondo, non c'è una sintassi semplice per fare un **lookup inverso**; dovete impostare una ricerca.

Ecco una funzione che richiede un valore e restituisce la prima chiave a cui corrisponde quel valore:

```
def inverso_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

Questa funzione è un altro esempio di schema di ricerca, ma usa un'istruzione che non abbiamo mai visto prima, `raise`. L'istruzione `raise` solleva un'eccezione; in questo caso genera un errore `LookupError`, che è un eccezione predefinita usata per indicare che un'operazione di lookup è fallita.

Se arriviamo a fine ciclo, significa che `v` non compare nel dizionario come valore, per cui solleviamo un'eccezione.

Ecco un esempio di lookup inverso riuscito:

```
>>> h = istogramma('parrot')
>>> chiave = inverso_lookup(h, 2)
>>> chiave
'r'
```

E di uno fallito:

```
>>> chiave = inverso_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in inverso_lookup
LookupError
```

Quando generate un errore, l'effetto è lo stesso di quando lo genera Python: viene stampato un traceback con un messaggio di errore.

L'istruzione `raise` può ricevere come parametro opzionale un messaggio di errore dettagliato. Per esempio:

```
>>> raise LookupError('il valore non compare nel dizionario')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: il valore non compare nel dizionario
```

Un lookup inverso è molto più lento di un lookup; se dovete farlo spesso, o se il dizionario diventa molto grande, le prestazioni del vostro programma potrebbero risentirne.

11.5 Dizionari e liste

Le liste possono comparire come valori in un dizionario. Per esempio, se avete un dizionario che fa corrispondere le lettere alle loro frequenze, potreste volere l'inverso; cioè creare un dizionario che a partire dalle frequenze fa corrispondere le lettere. Poiché ci possono essere più lettere con la stessa frequenza, ogni valore del dizionario inverso dovrebbe essere una lista di lettere.

Ecco una funzione che inverte un dizionario:

```
def inverti_diz(d):
    inverso = dict()
    for chiave in d:
        valore = d[chiave]
        if valore not in inverso:
            inverso[valore] = [chiave]
        else:
            inverso[valore].append(chiave)
    return inverso
```

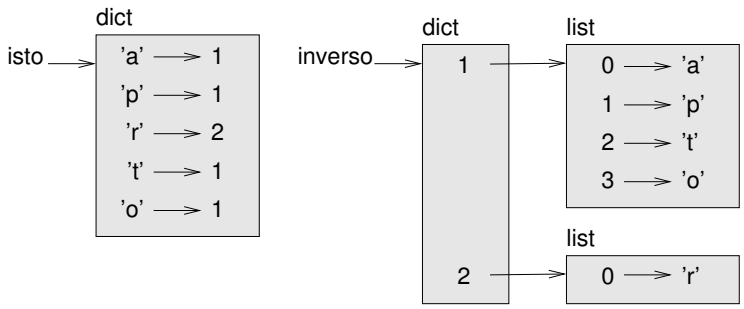

Figura 11.1: Diagramma di stato.

Per ogni ripetizione del ciclo, `chiave` prende una chiave da `d` e `valore` assume il corrispondente valore. Se `valore` non appartiene a `inverso`, vuol dire che non è ancora comparso, per cui creiamo un nuovo elemento e lo inizializziamo con un **singleton** (lista che contiene un solo elemento). Altrimenti, se il valore era già apparso, accodiamo la chiave corrispondente alla lista esistente.

Ecco un esempio:

```
>>> isto = istogramma('parrot')
>>> isto
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverso = inverti_diz(isto)
>>> inverso
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

La Figura 11.1 è un diagramma di stato che mostra `isto` e `inverso`. Un dizionario viene rappresentato come un riquadro con la scritta `dict` sopra e le coppie chiave-valore all'interno. Se i valori sono interi, float o stringhe, li raffiguro dentro il riquadro, lascio invece all'esterno le liste per mantenere semplice il diagramma.

Le liste possono essere valori nel dizionario, come mostra questo esempio, ma non possono essere chiavi. Ecco cosa succede se ci provate:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

Ho accennato che i dizionari sono implementati usando una tabella hash, e questo implica che alle chiavi deve poter essere applicato un **hash**.

Un **hash** è una funzione che prende un valore (di qualsiasi tipo) e restituisce un intero. I dizionari usano questi interi, chiamati valori hash, per conservare e consultare le coppie chiave-valore.

Questo sistema funziona se le chiavi sono immutabili; ma se sono mutabili, come le liste, succedono disastri. Per esempio, nel creare una coppia chiave-valore, Python fa l'hashing della chiave e la immagazzina nello spazio corrispondente. Se modificate la chiave e quindi viene nuovamente calcolato l'hash, si collocherebbe in un altro spazio. In quel caso potreste

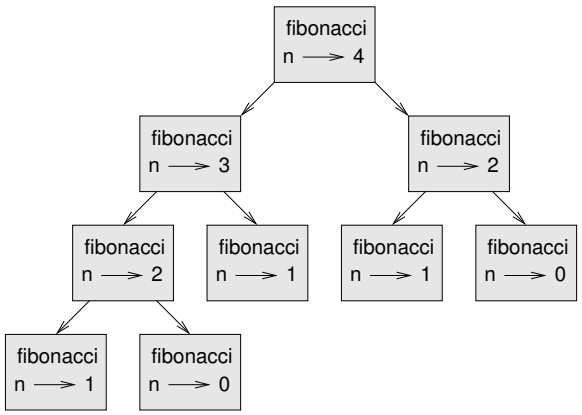

Figura 11.2: Grafico di chiamata.

avere due voci della stessa chiave, oppure non riuscire a trovare una chiave. In ogni caso il dizionario non funzionerà correttamente.

Ecco perché le chiavi devono essere idonee all'hashing, e quelle mutabili come le liste non lo sono. Il modo più semplice per aggirare questo limite è usare le tuple, che vedremo nel prossimo capitolo.

Dato che i dizionari sono mutabili, non possono essere usati come chiavi ma *possono* essere usati come valori.

11.6 Memoizzazione

Se vi siete sbizzarriti con la funzione `fibonacci` del Paragrafo 6.7, avrete notato che più grande è l'argomento che passate, maggiore è il tempo necessario per l'esecuzione della funzione. Inoltre, il tempo di elaborazione cresce rapidamente.

Per capire il motivo, confrontate la Figura 11.2, che mostra il **grafico di chiamata** di `fibonacci` con `n=4`:

Un grafico di chiamata mostra l'insieme dei frame della funzione, con linee che collegano ciascun frame ai frame delle funzioni che chiama a sua volta. In cima al grafico, `fibonacci` con `n=4` chiama `fibonacci` con `n=3` e `n=2`. A sua volta, `fibonacci` con `n=3` chiama `fibonacci` con `n=2` e `n=1`. E così via.

Provate a contare quante volte vengono chiamate `fibonacci(0)` e `fibonacci(1)`. Questa è una soluzione inefficiente del problema, che peggiora ulteriormente al crescere dell'argomento.

Una soluzione migliore è tenere da parte i valori che sono già stati calcolati, conservandoli in un dizionario. La tecnica di conservare per un uso successivo un valore già calcolato, così da non doverlo ricalcolare ogni volta, viene detta **memoizzazione**. Ecco una versione di `fibonacci` che usa la memoizzazione:

```
memo = {0:0, 1:1}

def fibonacci(n):
```

```
    if n in memo:
        return memo[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    memo[n] = res
    return res
```

`memo` è un dizionario che conserva i numeri di Fibonacci già conosciuti. Parte con due elementi: 0 che corrisponde a 0, e 1 che corrisponde a 1.

Ogni volta che `fibonacci` viene chiamata, controlla innanzitutto `memo`. Se quest'ultimo contiene già il risultato, ritorna immediatamente. Altrimenti deve calcolare il nuovo valore, lo aggiunge al dizionario e lo restituisce.

Provate ad eseguire questa versione di `fibonacci` e a confrontarla con l'originale: troverete che è molto più veloce.

11.7 Variabili globali

Nell'esempio precedente, `memo` viene creato esternamente alla funzione, pertanto appartiene al frame speciale chiamato `__main__`. Le variabili di `__main__` sono dette anche **globali** perché ad esse possono accedere tutte le funzioni. A differenza delle variabili locali, che sono distrutte una volta terminata l'esecuzione della loro funzione, quelle globali persistono tra una chiamata di funzione e l'altra.

Di frequente le variabili globali vengono usate come controlli o **flag**; vale a dire, variabili booleane che indicano quando una certa condizione è soddisfatta (True). Per esempio, alcuni programmi usano un flag di nome `verbose` per controllare che livello di dettaglio dare ad un output:

```
verbose = True

def esempio1():
    if verbose:
        print('esempio1 in esecuzione')
```

Se cercate di riassegnare una variabile globale, potreste avere una sorpresa. L'esempio seguente vorrebbe controllare se una funzione è stata chiamata:

```
stata_chiamata = False

def esempio2():
    stata_chiamata = True         # SBAGLIATO
```

Ma se la eseguite vedrete che il valore di `stata_chiamata` non cambia. Il motivo è che la funzione `esempio2` crea una nuova variabile di nome `stata_chiamata`, che è locale, viene distrutta al termine della funzione e non ha effetti sulla variabile globale.

Per riassegnare una variabile globale dall'interno di una funzione, dovete **dichiarare** la variabile globale prima di usarla:

```
stata_chiamata = False

def esempio2():
    global stata_chiamata
    stata_chiamata = True
```

L'istruzione `global` dice all'interprete una cosa del genere: "In questa funzione, quando dico `stata_chiamata`, intendo la variabile globale: non crearne una locale".

Ecco un altro esempio che cerca di aggiornare una variabile globale:

```
conta = 0

def esempio3():
    conta = conta + 1          # SBAGLIATO
```

Se lo eseguite, ottenete:

```
UnboundLocalError: local variable 'conta' referenced before assignment
```

Python presume che `conta` all'interno della funzione sia una variabile locale, e con questa premessa significa che state usando la variabile prima di averla inizializzata. La soluzione è ancora quella di dichiarare `conta` globale.

```
def esempio3():
    global conta
    conta += 1
```

Se una variabile globale fa riferimento ad un valore mutabile, potete modificare il valore senza dichiarare la variabile:

```
noto = {0:0, 1:1}

def esempio4():
    noto[2] = 1
```

Pertanto, potete aggiungere, rimuovere e sostituire elementi di una lista o dizionario globali; tuttavia, se volete riassegnare la variabile, occorre dichiararla:

```
def esempio5():
    global noto
    noto = dict()
```

Le variabili globali possono risultare utili, ma se ce ne sono molte e le modificate di frequente, possono rendere difficile il debug del programma.

11.8 Debug

Se lavorate con banche dati di grosse dimensioni, può diventare oneroso fare il debug stampando e controllando i risultati di output manualmente. Ecco allora alcuni suggerimenti per fare il debug in queste situazioni:

Ridurre l'input: Se possibile, riducete le dimensioni della banca dati. Per esempio, se il programma legge un file di testo, cominciate con le sole prime 10 righe o con il più piccolo campione che riuscite a trovare. Potete anche adattare i file stessi, o (meglio) modificare il programma, in modo che legga solo le prime `n` righe.

Se c'è un errore, potete ridurre `n` al più piccolo valore per il quale si manifesta l'errore, poi aumentarlo gradualmente finché non trovate e correggete l'errore.

Controllare riassunti e tipi: Invece di stampare e controllare l'intera banca dati, prendete in considerazione di stampare riassunti dei dati: ad esempio il numero di elementi in un dizionario o la sommatoria di una lista di numeri.

Una causa frequente di errori in esecuzione è un valore che non è del tipo giusto. Per fare il debug di questo tipo di errori basta spesso stampare il tipo di un valore.

Scrivere controlli automatici: Talvolta è utile scrivere del codice per controllare automaticamente gli errori. Per esempio, se dovete calcolare la media di una lista di numeri, potete controllare che il risultato non sia maggiore dell'elemento più grande della lista e non sia minore del più piccolo. Questo è detto "controllo di congruenza" perché mira a trovare i risultati "incongruenti".

Un altro tipo di controllo confronta i risultati di due calcoli per vedere se collimano. Questo è chiamato "controllo di coerenza".

Stampare gli output in bella copia: Una buona presentazione dei risultati di debug rende più facile trovare un errore. Abbiamo visto un esempio nel Paragrafo 6.9. Uno strumento utile è il modulo `pprint`: esso contiene la funzione `pprint` che mostra i tipi predefiniti in un formato più leggibile (pprint infatti sta per "pretty print").

Ancora, il tempo che impiegate a scrivere del codice temporaneo può essere ripagato dalla riduzione del tempo di debug.

11.9 Glossario

mappatura: Relazione per cui a ciascun elemento di un insieme corrisponde un elemento di un altro insieme.

dizionario: Una mappatura da chiavi nei loro valori corrispondenti.

coppia chiave-valore: Rappresentazione della mappatura da una chiave in un valore.

elemento: In un dizionario, altro nome della coppia chiave-valore.

chiave: Oggetto che compare in un dizionario come prima voce di una coppia chiave-valore.

valore: Oggetto che compare in un dizionario come seconda voce di una coppia chiave-valore. È più specifico dell'utilizzo del termine "valore" fatto sinora.

implementazione: Un modo per effettuare un'elaborazione.

tabella hash: Algoritmo usato per implementare i dizionari in Python.

funzione hash: Funzione usata da una tabella hash per calcolare la collocazione di una chiave.

hash-abile: Un tipo a cui si può applicare la funzione hash. I tipi immutabili come interi, float e stringhe lo sono; i tipi mutabili come liste e dizionari no.

lookup: Operazione su un dizionario che trova il valore corrispondente a una data chiave.

lookup inverso: Operazione su un dizionario che trova una o più chiavi alle quali è associato un dato valore.

singleton: Lista (o altra sequenza) con un singolo elemento.

grafico di chiamata: Diagramma che mostra tutti i frame creati durante l'esecuzione di un programma, con frecce che collegano ciascun chiamante ad ogni chiamata.

memoizzazione: Conservare un valore calcolato per evitarne il successivo ricalcolo.

variabile globale: Variabile definita al di fuori di una funzione, alla quale ogni funzione può accedere.

istruzione global: Istruzione che dichiara globale il nome di una variabile.

controllo o flag: Variabile booleana usata per indicare se una condizione è soddisfatta.

dichiarazione: Istruzione come `global`, che comunica all'interprete un'informazione su una variabile.

11.10 Esercizi

Esercizio 11.1. *Scrivete una funzione che legga le parole in* `words.txt` *e le inserisca come chiavi in un dizionario. I valori non hanno importanza. Usate poi l'operatore* `in` *come modo rapido per controllare se una stringa è contenuta nel dizionario.*

Se avete svolto l'Esercizio 10.10, potete confrontare la velocità di questa implementazione con l'operatore `in` *applicato alla lista e la ricerca binaria.*

Esercizio 11.2. *Leggete la documentazione del metodo dei dizionari* `setdefault` *e usatelo per scrivere una versione più concisa di* `inverti_diz`. *Soluzione:* `http://thinkpython2.com/code/invert_dict.py`.

Esercizio 11.3. *Applicate la memoizzazione alla funzione di Ackermann dell'Esercizio 6.2 e provate a vedere se questa tecnica rende possibile il calcolo della funzione con argomenti più grandi. Suggerimento: no. Soluzione:* `http://thinkpython2.com/code/ackermann_memo.py`.

Esercizio 11.4. *Se avete svolto l'Esercizio 10.7, avete già una funzione di nome* `ha_duplicati` *che richiede come parametro una lista e restituisce* `True` *se ci sono oggetti ripetuti all'interno della lista.*

Usate un dizionario per scrivere una versione più rapida e semplice di `ha_duplicati`. *Soluzione:* `http://thinkpython2.com/code/has_duplicates.py`.

Esercizio 11.5. *Due parole sono "ruotabili" se potete far ruotare le lettere dell'una per ottenere l'altra (vedere* `ruota_parola` *nell'Esercizio 8.5).*

Scrivete un programma che legga un elenco di parole e trovi tutte le coppie di parole ruotabili. Soluzione: `http://thinkpython2.com/code/rotate_pairs.py`.

Esercizio 11.6. *Ecco un altro quesito tratto da* Car Talk *(*`http://www.cartalk.com/content/puzzlers`*):*

> *"Questo ci è stato mandato da un amico di nome Dan O'Leary. Si è recentemente imbattuto in una parola inglese di una sillaba e cinque lettere che ha questa singolare proprietà: se togliete la prima lettera, le lettere restanti formano un omofono della prima parola, cioè un'altra parola che pronunciata suona allo stesso modo. Se poi rimettete la prima lettera e togliete la seconda, ottenete ancora un altro omofono della parola di origine. Qual è questa parola?"*

> *"Facciamo un esempio che non funziona del tutto. Prendiamo la parola 'wrack'; togliendo la prima lettera resta 'rack', che è un'altra parola ma è un perfetto omofono. Se però rimettete la prima lettera e togliete la seconda, ottenete 'wack' che pure esiste ma non è un omofono delle altre due parole."*
>
> *"Esiste comunque almeno una parola, che Dan e noi conosciamo, che dà due parole omofone di quattro lettere, sia che togliate la prima oppure la seconda lettera."*

Potete usare il dizionario dell'Esercizio 11.1 per controllare se esiste una tale stringa nell'elenco di parole.

Per controllare se due parole sono omofone, potete usare il CMU Pronouncing Dictionary, scaricabile da `http://www.speech.cs.cmu.edu/cgi-bin/cmudict` *oppure da* `http://thinkpython2.com/code/c06d` *e potete anche procurarvi* `http://thinkpython2.com/code/pronounce.py`*, che fornisce una funzione di nome* `read_dictionary` *che legge il dizionario delle pronunce e restituisce un dizionario Python in cui a ciascuna parola corrisponde la stringa che ne descrive la pronuncia.*

Scrivete un programma che elenchi tutte le parole che risolvono il quesito. Soluzione: `http://thinkpython2.com/code/homophone.py`*.*

Capitolo 12

Tuple

Questo capitolo illustra un altro tipo di dati predefinito, le tuple, per poi mostrare come liste, tuple e dizionari possono lavorare insieme. Viene inoltre presentata una utile caratteristica per le liste di argomenti a lunghezza variabile: gli operatori di raccolta e spacchettamento.

12.1 Le tuple sono immutabili

Una tupla è una sequenza di valori. I valori possono essere di qualsiasi tipo, sono indicizzati tramite numeri interi, e in questo somigliano moltissimo alle liste. La differenza fondamentale è che le tuple sono immutabili.

Sintatticamente, la tupla è un elenco di valori separati da virgole:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Sebbene non sia necessario, è convenzione racchiudere le tuple tra parentesi tonde:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Per creare una tupla con un singolo elemento, occorre aggiungere una virgola dopo l'elemento:

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

Senza la virgola, infatti, un unico valore tra parentesi non è una tupla ma una stringa:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

Un altro modo di creare una tupla è usare la funzione predefinita `tuple`. Se priva di argomento, crea una tupla vuota:

```
>>> t = tuple()
>>> t
()
```

Se l'argomento è una sequenza (stringa, lista o tupla), il risultato è una tupla con gli elementi della sequenza:

```
>>> t = tuple('lupini')
>>> t
('l', 'u', 'p', 'i', 'n', 'i')
```

Siccome `tuple` è il nome di una funzione predefinita, bisogna evitare di usarlo come nome di variabile.

La maggior parte degli operatori delle liste funzionano anche con le tuple. L'operatore parentesi quadre indicizza un elemento della tupla:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

E l'operatore di slicing seleziona una serie di elementi consecutivi:

```
>>> t[1:3]
('b', 'c')
```

Ma a differenza delle liste, se cercate di modificare gli elementi di una tupla ottenete un messaggio d'errore:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Dato che le tuple sono immutabili, non si può modificarne gli elementi. Ma potete sostituire una tupla con un'altra:

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

Questa istruzione crea una nuova tupla e poi fa in modo che `t` si riferisca ad essa.

Gli operatori di confronto funzionano con le tuple e le altre sequenze; Python inizia a confrontare il primo elemento di ciascuna sequenza. Se sono uguali, passa all'elemento successivo e così via, finché non trova due elementi diversi. Gli eventuali elementi che seguono vengono trascurati (anche se sono molto grandi).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12.2 Assegnazione di tupla

Spesso è utile scambiare i valori di due variabili tra loro. Con le istruzioni di assegnazione convenzionali, dobbiamo usare una variabile temporanea. Per esempio per scambiare `a` e `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Questo metodo è farraginoso; l'utilizzo dell'**assegnazione di tupla** è più elegante:

```
>>> a, b = b, a
```

Sul lato sinistro abbiamo una tupla di variabili; su quello destro, una tupla di espressioni. Ciascun valore viene assegnato alla rispettiva variabile. Tutte le espressioni sul lato destro vengono valutate prima di ogni assegnazione.

Il numero di variabili sulla sinistra deve essere uguale al numero di valori sulla destra:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

Più in generale, sul lato destro può esserci qualsiasi tipo di sequenza (stringhe, liste o tuple). Per esempio, per separare un indirizzo email tra nome utente e dominio, potete scrivere:

```
>>> indirizzo = 'monty@python.org'
>>> nome, dominio = indirizzo.split('@')
```

Il valore di ritorno del metodo `split` è una lista con due elementi; il primo è assegnato alla variabile `nome`, il secondo a `dominio`.

```
>>> nome
'monty'
>>> dominio
'python.org'
```

12.3 Tuple come valori di ritorno

In senso stretto, una funzione può restituire un solo valore di ritorno, ma se il valore è una tupla, l'effetto pratico è quello di restituire valori molteplici. Per esempio, se volete dividere due interi e calcolare quoziente e resto, è poco efficiente calcolare `x/y` e poi `x%y`. Meglio calcolarli entrambi in una volta sola.

La funzione predefinita `divmod` riceve due argomenti e restituisce una tupla di due valori, il quoziente e il resto. E potete memorizzare il risultato con una tupla:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

Oppure, usate l'assegnazione di tupla per conservare gli elementi separatamente:

```
>>> quoziente, resto = divmod(7, 3)
>>> quoziente
2
>>> resto
1
```

Ecco un esempio di funzione che restituisce una tupla:

```
def min_max(t):
    return min(t), max(t)
```

`max` e `min` sono funzioni predefinite che estraggono da una sequenza il valore massimo e quello minimo. `min_max` li estrae entrambi e restituisce una tupla di due valori.

12.4 Tuple di argomenti a lunghezza variabile

Le funzioni possono ricevere un numero variabile di argomenti. Un nome di parametro che comincia con `*`, **raccoglie** gli argomenti in una tupla. Per esempio, `stampatutti` riceve un qualsiasi numero di argomenti e li visualizza:

```
def stampatutti(*args):
    print(args)
```

Il parametro di raccolta può avere qualunque nome, ma per convenzione si usa `args`. Ecco come funziona:

```
>>> stampatutti(1, 2.0, '3')
(1, 2.0, '3')
```

Il contrario della raccolta è lo **spacchettamento**. Se avete una sequenza di valori e volete passarla a una funzione come argomenti multipli, usate ancora l'operatore `*`. Per esempio, `divmod` richiede esattamente due argomenti; passare una tupla non funziona:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

Ma se spacchettate la tupla, funziona:

```
>>> divmod(*t)
(2, 1)
```

Molte funzioni predefinite possono usare le tuple di argomenti a lunghezza variabile. Ad esempio, `max` e `min` ricevono un numero qualunque di argomenti:

```
>>> max(1,2,3)
3
```

Ma con `sum` non funziona.

```
>>> sum(1,2,3)
TypeError: sum expected at most 2 arguments, got 3
```

Per esercizio, scrivete una funzione di nome `sommatutto` che riceva un numero di argomenti a piacere e ne restituisca la somma.

12.5 Liste e tuple

`zip` è una funzione predefinita che riceve due o più sequenze e restituisce una lista di tuple, dove ciascuna tupla contiene un elemento di ciascuna sequenza. Il nome si riferisce alla cerniera-lampo (*zipper*), che unisce due file di dentelli, alternandoli.

Questo esempio abbina una stringa e una lista:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

Il risultato è un **oggetto zip** capace di iterare attraverso le coppie. L'uso più frequente di `zip` è in un ciclo `for`:

```
>>> for coppia in zip(s, t):
...     print(coppia)
...
('a', 0)
('b', 1)
('c', 2)
```

Un oggetto zip è un tipo di **iteratore**, che è un qualsiasi oggetto in grado di iterare attraverso una sequenza. Gli iteratori sono per certi versi simili alle liste, ma a differenza di queste ultime, non si può usare un indice per scegliere un elemento da un iteratore.

Se desiderate usare operatori e metodi delle liste, potete crearne una utilizzando un oggetto zip:

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

Il risultato è una lista di tuple, e in questo esempio ciascuna tupla contiene un carattere della stringa e il corrispondente elemento della lista.

Se le sequenze non sono della stessa lunghezza, il risultato ha la lunghezza di quella più corta:

```
>>> list(zip('Anna', 'Edo'))
[('A', 'E'), ('n', 'd'), ('n', 'o')]
```

Potete usare l'assegnazione di tupla in un ciclo `for` per attraversare una lista di tuple:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for lettera, numero in t:
    print(numero, lettera)
```

Ad ogni ciclo, Python seleziona la tupla successiva all'interno della lista e ne assegna gli elementi a `lettera` e `numero`, quindi li stampa. Il risultato di questo ciclo è:

```
0 a
1 b
2 c
```

Se combinate `zip`, `for` e assegnazione di tupla, ottenete un utile costrutto per attraversare due o più sequenze contemporaneamente. Per esempio, `corrispondenza` prende due sequenze, `t1` e `t2`, e restituisce `True` se esiste almeno un indice `i` tale che `t1[i] == t2[i]`:

```
def corrispondenza(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

Se volete attraversare gli elementi di una sequenza e i loro indici, potete usare la funzione predefinita `enumerate`:

```
for indice, elemento in enumerate('abc'):
    print(indice, elemento)
```

Il risultato di `enumerate` è un oggetto enumerate, che itera una sequenza di coppie; ogni coppia contiene un indice (a partire da 0) e un elemento della sequenza data. In questo esempio l'output è di nuovo:

```
0 a
1 b
2 c
```

12.6 Dizionari e tuple

I dizionari supportano un metodo di nome `items` che restituisce una sequenza di tuple, dove ogni tupla è una delle coppie chiave-valore.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

Il risultato è un oggetto `dict_items`, un iteratore che itera le coppie chiave-valore. Si può usare in un ciclo `for` in questo modo:

```
>>> for chiave, valore in d.items():
...     print(chiave, valore)
...
c 2
a 0
b 1
```

Come di consueto per i dizionari, gli elementi non sono in un ordine particolare. Per altro verso, potete usare una lista di tuple per inizializzare un nuovo dizionario:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

La combinazione di `dict` e `zip` produce un modo conciso di creare un dizionario:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

Anche il metodo dei dizionari `update` prende una lista di tuple e le aggiunge, come coppie chiave-valore, a un dizionario esistente.

L'uso delle tuple come chiavi di un dizionario è frequente (soprattutto perché le liste non si possono usare in quanto mutabili). Per esempio, un elenco telefonico può mappare da coppie di nomi e cognomi nei numeri di telefono. Supponendo di aver definito `cognome`, `nome` e `numero`, possiamo scrivere:

```
elenco[cognome,nome] = numero
```

L'espressione tra parentesi quadre è una tupla. Possiamo usare l'assegnazione di tupla per attraversare questo dizionario.

```
for cognome, nome in elenco:
    print(nome, cognome, elenco[cognome,nome])
```

Questo ciclo attraversa le chiavi in `elenco`, che sono tuple. Assegna gli elementi di ogni tupla a `cognome` e `nome`, quindi stampa il nome completo e il numero di telefono corrispondente.

Ci sono due modi per rappresentare le tuple in un diagramma di stato. La versione più dettagliata mostra gli indici e gli elementi così come compaiono in una lista. Per esempio la tupla `('Cleese', 'John')` comparirebbe come in Figura 12.1.

Ma in un diagramma più ampio è meglio tralasciare i dettagli. Per esempio, quello dell'elenco telefonico può essere come in Figura 12.2.

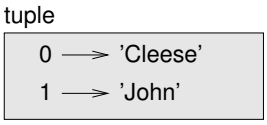

Figura 12.1: Diagramma di stato.

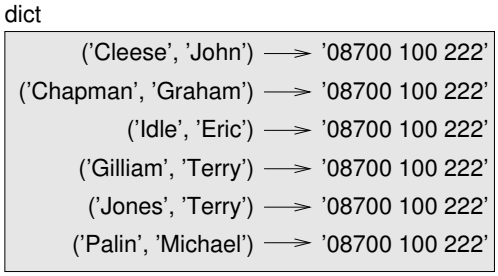

Figura 12.2: Diagramma di stato.

Qui le tuple sono mostrate usando la sintassi di Python come abbreviazione grafica. Il numero di telefono nel diagramma è quello dei reclami della BBC, per cui vi prego, non chiamatelo.

12.7 Sequenze di sequenze

Ci siamo concentrati finora sulle liste di tuple, ma quasi tutti gli esempi di questo capitolo funzionano anche con liste di liste, tuple di tuple, e tuple di liste. Per evitare di elencare tutte le possibili combinazioni, è più semplice usare il termine sequenze di sequenze.

In molti casi, i diversi tipi di sequenze (strighe, liste, tuple) possono essere intercambiabili. E allora, con che criterio usarne una piuttosto di un'altra?

Le stringhe sono ovviamente le più limitate, perché gli elementi devono essere dei caratteri. E sono anche immutabili. Se dovete cambiare i caratteri in una stringa, anziché crearne una nuova, utilizzare una lista di caratteri può essere una scelta migliore.

Le liste sono usate più di frequente delle tuple, soprattutto perché sono mutabili. Ma ci sono alcuni casi in cui le tuple sono preferibili:

1. In certi contesti, come un'istruzione `return`, è sintatticamente più semplice creare una tupla anziché una lista.
2. Se vi serve una sequenza da usare come chiave di un dizionario, dovete per forza usare un tipo immutabile come una tupla o una stringa.
3. Se state passando una sequenza come argomento a una funzione, usare le tuple riduce le possibilità di comportamenti imprevisti dovuti agli alias.

Siccome le tuple sono immutabili, non possiedono metodi come `sort` e `reverse`, che modificano delle liste esistenti. Però Python contiene la funzione `sorted`, che richiede una sequenza e restituisce una nuova lista con gli stessi elementi della sequenza, ordinati, e `reversed`, che prende una sequenza e restituisce un iteratore che attraversa la lista in ordine inverso.

12.8 Debug

Liste, dizionari e tuple sono esempi di **strutture di dati**; in questo capitolo abbiamo iniziato a vedere strutture di dati composte, come liste di tuple, o dizionari che contengono tuple come chiavi e liste come valori. Si tratta di elementi utili, ma soggetti a quelli che io chiamo `errori di formato`; cioè errori causati dal fatto che una struttura di dati è di tipo, dimensione o struttura sbagliati. Ad esempio, se un programma si aspetta una lista che contiene un numero intero e invece gli passate un intero puro e semplice (non incluso in una lista), non funzionerà.

Per facilitare il debug di questo genere di errori, ho scritto un modulo di nome `structshape` che contiene una funzione, anch'essa di nome `structshape`, che riceve come argomento una qualunque struttura di dati e restituisce una stringa che ne riassume il formato. Potete scaricarlo dal sito `http://thinkpython2.com/code/structshape.py`

Questo è il risultato per una lista semplice:

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> structshape(t)
'list of 3 int'
```

Un programma più aggraziato avrebbe scritto "list of 3 int*s*", ma è più semplice non avere a che fare con i plurali. Ecco una lista di liste:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

Se gli elementi della lista non sono dello stesso tipo, `structshape` li raggruppa, in ordine, per tipo:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

Ecco una lista di tuple:

```
>>> s = 'abc'
>>> lt = zip(list(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

Ed ecco un dizionario di 3 elementi in cui corrispondono interi a stringhe

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

Se fate fatica a tenere sotto controllo le vostre strutture di dati, `structshape` può esservi di aiuto.

12.9 Glossario

tupla: Una sequenza di elementi immutabile.

assegnazione di tupla: Assegnazione costituita da una sequenza sul lato destro e una tupla di variabili su quello sinistro. Il lato destro viene valutato, quindi gli elementi vengono assegnati alle variabili sulla sinistra.

raccolta: L'operazione di assemblare una tupla di argomenti a lunghezza variabile.

spacchettamento: L'operazione di trattare una sequenza come una lista di argomenti.

oggetto zip: Il risultato della chiamata della funzione predefinita `zip`; un oggetto che itera attraverso una sequenza di tuple.

iteratore: Un oggetto in grado di iterare attraverso una sequenza, ma che non fornisce operatori e metodi delle liste.

struttura di dati: Una raccolta di valori correlati, spesso organizzati in liste, dizionari, tuple, ecc.

errore di formato: Errore dovuto ad un valore che ha un formato sbagliato, ovvero tipo o dimensioni errati.

12.10 Esercizi

Esercizio 12.1. *Scrivete una funzione di nome* `piu_frequente` *che riceva una stringa e stampi le lettere in ordine di frequenza decrescente. Trovate delle frasi di esempio in diverse lingue e osservate come varia la frequenza delle lettere. Confrontate i vostri risultati con le tabelle del sito* `http://en.wikipedia.org/wiki/Letter_frequencies`*. Soluzione:* `http://thinkpython2.com/code/most_frequent.py`*.*

Esercizio 12.2. *Ancora anagrammi!*

1. *Scrivete un programma che legga un elenco di parole da un file (vedi Paragrafo 9.1) e stampi tutti gli insiemi di parole che sono tra loro anagrammabili.*

   Un esempio di come si può presentare il risultato:

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   Suggerimento: potete costruire un dizionario che faccia corrispondere un gruppo di lettere con una lista di parole che si possono scrivere con quelle lettere. Il problema è: come rappresentare il gruppo di lettere in modo che possano essere usate come chiave?

2. *Modificate il programma in modo che stampi la lista di anagrammi più lunga per prima, seguita dalla seconda più lunga, e così via.*

3. *Nel gioco da tavolo* Scarabeo*, fate un "en-plein" quando giocate tutte le sette lettere sul vostro leggio formando, insieme a una lettera sul tavolo, una parola di otto lettere. Con quale gruppo di 8 lettere si può fare un "en-plein" con maggior probabilità? Suggerimento: il gruppo dà sette combinazioni.*

   Soluzione: `http://thinkpython2.com/code/anagram_sets.py`*.*

Esercizio 12.3. *Si ha una metatesi quando una parola si può ottenere scambiando due lettere di un'altra parola, per esempio: "conversa" e "conserva". Scrivete un programma che trovi tutte le coppie con metatesi nel dizionario. Suggerimento: non provate tutte le possibili coppie di parole e non provate tutti i possibili scambi. Soluzione:* `http://thinkpython2.com/code/metathesis.py`*. Fonte: Esercizio suggerito da un esempio nel sito* `http://puzzlers.org`*.*

Esercizio 12.4. *Ed ecco un altro quesito di Car Talk: (*`http://www.cartalk.com/content/puzzlers`*):*

> *Qual è la più lunga parola inglese che rimane una parola valida se le togliete una lettera alla volta? Le lettere possono essere rimosse sia agli estremi o in mezzo, ma senza spostare le lettere rimanenti. Ogni volta che togliete una lettera, ottenete un'altra parola inglese. Se andate avanti, ottenete un'altra parola. Ora, voglio sapere qual è la parola più lunga possibile e quante lettere ha.*
>
> *Vi faccio un piccolo esempio: Sprite. Partite da sprite, togliete una lettera, una interna, come la r e resta la parola spite, poi togliete la e finale e avete spit, togliamo la s e resta pit, poi it, infine I.*

Scrivete un programma che trovi tutte le parole che sono riducibili in questa maniera, quindi trovate la più lunga.

Questo esercizio è un po' più impegnativo degli altri, quindi eccovi alcuni suggerimenti:

1. *Potete scrivere una funzione che prenda una parola e calcoli una lista di tutte le parole che si possono formare togliendo una lettera. Queste sono le "figlie" della parola.*
2. *Ricorsivamente, una parola è riducibile se qualcuna delle sue figlie è a sua volta riducibile. Come caso base, potete considerare riducibile la stringa vuota.*
3. *L'elenco di parole che ho fornito,* `words.txt`*, non contiene parole di una lettera. Potete quindi aggiungere "I", "a", e la stringa vuota.*
4. *Per migliorare le prestazioni del programma, potete memoizzare le parole che sono risultate riducibili.*

Soluzione: `http://thinkpython2.com/code/reducible.py`*.*

Capitolo 13

Esercitazione: Scelta della struttura di dati

Giunti a questo punto, avete conosciuto le principali strutture di dati di Python, e avete visto alcuni algoritmi che le utilizzano. Se vi interessa saperne di più sugli algoritmi, potrebbe essere un buon momento per leggere l'Appendice B. Non è però necessario per proseguire la lettura: fatelo quando vi pare opportuno.

L'esercitazione di questo capitolo vi aiutèrà ad impratichirvi nella scelta e nell'uso delle strutture di dati.

13.1 Analisi di frequenza delle parole

Come al solito, tentate almeno di risolvere gli esercizi prima di guardare le mie risoluzioni.

Esercizio 13.1. *Scrivete un programma che legga un file di testo, separi da ogni riga le singole parole, scarti gli spazi bianchi e la punteggiatura dalle parole, e converta tutto in lettere minuscole.*

Suggerimento: il modulo `string` *fornisce una stringa chiamata* `whitespace`*, che contiene i caratteri spaziatori come spazio, tabulazione, a capo ecc., e una di nome* `punctuation` *che contiene i caratteri di punteggiatura. Vediamo se Python ce lo conferma:*

```
>>> import string
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

Potete anche fare uso dei metodi delle stringhe `strip`*,* `replace` *e* `translate`*.*

Esercizio 13.2. *Andate sul sito del Progetto Gutenberg (*`http://gutenberg.org`*) e scaricate il libro fuori copyright che preferite, in formato di testo semplice.*

Modificate il programma dell'esercizio precedente in modo che legga il libro da voi scaricato, salti le informazioni di intestazione all'inizio del file, ed elabori il resto come sopra.

Quindi modificate il programma in modo che conti il numero di parole totale del libro, e quante volte è usata ciascuna parola.

Visualizzate il numero di parole diverse usate nel libro. Confrontate libri diversi di diversi autori, scritti in epoche diverse. Quale autore usa il vocabolario più ricco?

Esercizio 13.3. *Modificate il programma dell'esercizio precedente in modo da visualizzare le 20 parole più usate nel libro.*

Esercizio 13.4. *Modificate il programma precedente in modo che acquisisca un elenco di parole (vedi Paragrafo 9.1) e quindi stampi l'elenco delle parole contenute nel libro che non sono presenti nell'elenco di parole. Quante di esse sono errori di stampa? Quante sono parole comuni che* dovrebbero *essere nell'elenco, e quante sono del tutto oscure?*

13.2 Numeri casuali

A parità di dati in ingresso, la maggior parte dei programmi genera gli stessi risultati ad ogni esecuzione, e per questo motivo sono detti deterministici. Di solito il determinismo è una cosa giusta, poiché è ovvio attendersi che gli stessi dati producano gli stessi risultati. Per certe applicazioni, tuttavia, è richiesto che l'elaboratore sia imprevedibile: i videogiochi sono un classico esempio, ma ce ne sono anche altri.

Creare un programma realmente non-deterministico è una cosa piuttosto difficile, ma ci sono dei sistemi per renderlo almeno apparentemente non-deterministico. Uno di questi è utilizzare degli algoritmi che generano dei numeri **pseudocasuali**. Questi numeri non sono veri numeri casuali, dato che sono generati da un elaboratore deterministico, ma a prima vista è praticamente impossibile distinguerli da numeri casuali.

Il modulo `random` contiene delle funzioni che generano numeri pseudocasuali (d'ora in avanti chiamati "casuali" per semplicità).

La funzione `random` restituisce un numero casuale in virgola mobile compreso nell'intervallo tra 0.0 e 1.0 (incluso 0.0 ma escluso 1.0). Ad ogni chiamata di `random`, si ottiene il numero successivo di una lunga serie di numeri casuali. Per vedere un esempio provate ad eseguire questo ciclo:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

La funzione `randint` richiede due parametri interi, uno `inferiore` e uno `superiore`, e restituisce un intero casuale nell'intervallo tra i due parametri (entrambi compresi)

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Per estrarre un elemento a caso da una sequenza, potete usare `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Il modulo `random` contiene anche delle funzioni per generare valori pseudocasuali da distribuzioni continue, incluse gaussiane, esponenziali, gamma, e alcune altre.

Esercizio 13.5. *Scrivete una funzione di nome* `estrai_da_isto` *che prenda un istogramma come definito nel Paragrafo 11.2 e restituisca un valore casuale dall'istogramma, scelto in modo che la probabilità sia proporzionale alla frequenza. Per esempio, dato questo istogramma:*

```
>>> t = ['a', 'a', 'b']
>>> isto = istogramma(t)
>>> isto
{'a': 2, 'b': 1}
```

la vostra funzione dovrebbe restituire `'a'` *con probabilità* $2/3$ *e* `'b'` *con probabilità* $1/3$.

13.3 Istogramma di parole

Provate a risolvere gli esercizi precedenti prima di procedere oltre. Le soluzioni sono scaricabili da `http://thinkpython2.com/code/analyze_book1.py`. Vi servirà anche `http://thinkpython2.com/code/emma.txt`.

Ecco un programma che legge un file e costruisce un istogramma della parole in esso contenute:

```
import string

def elabora_file(nomefile):
    isto = dict()
    fp = open(nomefile)
    for riga in fp:
        elabora_riga(riga, isto)
    return isto

def elabora_riga(riga, isto):
    riga = riga.replace('-', ' ')

    for parola in riga.split():
        parola = parola.strip(string.punctuation + string.whitespace)
        parola = parola.lower()
        isto[parola] = isto.get(parola, 0) + 1

isto = elabora_file('emma.txt')
```

Questo programma legge il file `emma.txt`, che contiene il testo di *Emma* di Jane Austen.

`elabora_file` legge ciclicamente le righe del file, passandole una per volta a `elabora_riga`. L'istogramma `isto` viene usato come un accumulatore.

`elabora_riga` usa il metodo delle stringhe `replace` per sostituire i trattini con gli spazi, prima di usare `split` per suddividere la riga in una lista di stringhe. Attraversa poi la lista di parole e usa `strip` e `lower` per togliere la punteggiatura e convertire in lettere minuscole. (Diciamo per semplicità che le stringhe sono "convertite": essendo queste immutabili, i metodi come `strip` e `lower` in realtà restituiscono nuove stringhe).

Infine, `elabora_riga` aggiorna l'istogramma creando un nuovo elemento o incrementandone uno esistente.

Per contare il numero di parole totali, possiamo aggiungere le frequenze nell'istogramma:

```
def parole_totali(isto):
    return sum(isto.values())
```

Il numero di parole diverse è semplicemente il numero di elementi nel dizionario:

```
def parole_diverse(isto):
    return len(isto)
```

Ed ecco del codice per stampare i risultati:

```
print('Numero totale di parole:', parole_totali(isto))
print('Numero di parole diverse:', parole_diverse(isto))
```

E i relativi risultati:

```
Numero totale di parole: 161080
Numero di parole diverse: 7214
```

13.4 Parole più comuni

Per trovare le parole più comuni, possiamo creare una lista di tuple, in cui ciascuna tupla contiene una parola e la sua frequenza, ed ordinarle:

La funzione seguente prende un istogramma e restituisce una lista di tuple parola-frequenza:

```
def piu_comuni(isto):
    t = []
    for chiave, valore in isto.items():
        t.append((valore, chiave))

    t.sort(reverse=True)
    return t
```

In ogni tupla, la frequenza compare per prima, quindi la lista risultante è ordinata per frequenza. Ecco un ciclo che stampa le dieci parole più comuni:

```
t = piu_comuni(hist)
print('Le parole più comuni sono:')
for freq, parola in t[:10]:
    print(parola, freq, sep='\t')
```

Ho usato l'argomento con nome `sep` per dire a `print` di usare un carattere di tabulazione come "separatore", anziché uno spazio, in modo che la seconda colonna risulti allineata. E questi sono i risultati nel caso di *Emma*:

```
Le parole più comuni sono:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
```

```
it      2529
her     2483
was     2400
she     2364
```

Si potrebbe semplificare il codice utilizzando il parametro `key` della funzione `sort`. Se vi incuriosisce, leggete `https://wiki.python.org/moin/HowTo/Sorting`.

13.5 Parametri opzionali

Abbiamo già visto funzioni predefinite e metodi che ricevono argomenti opzionali. È possibile anche scrivere funzioni personalizzate con degli argomenti opzionali. Ad esempio, questa è una funzione che stampa le parole più comuni in un istogramma:

```
def stampa_piu_comuni(isto, num=10):
    t = piu_comuni(isto)
    print('Le parole più comuni sono:')
    for freq, parola in t[:num]:
        print(parola, freq, sep='\t')
```

Il primo parametro è obbligatorio; il secondo è opzionale. Il **valore di default** di `num` è 10.

Se passate un solo argomento:

```
stampa_piu_comuni(isto)
```

`num` assume il valore predefinito. Se ne passate due:

```
stampa_piu_comuni(isto, 20)
```

`num` assume il valore che avete specificato. In altre parole, l'argomento opzionale **sovrascrive** il valore predefinito.

Se una funzione ha sia parametri obbligatori che opzionali, tutti quelli obbligatori devono essere scritti per primi, seguiti da quelli opzionali.

13.6 Sottrazione di dizionari

Trovare le parole del libro non comprese nell'elenco `words.txt` è un problema che possiamo classificare come sottrazione di insiemi, cioè occorre trovare le parole appartenenti a un insieme (le parole contenute nel libro) che non si trovano nell'altro insieme (l'elenco).

`sottrai` prende i dizionari `d1` e `d2` e ne restituisce uno nuovo che contiene tutte le chiavi di `d1` che non si trovano in `d2`. Siccome non ci interessano affatto i valori, li impostiamo tutti a None.

```
def sottrai(d1, d2):
    res = dict()
    for chiave in d1:
        if chiave not in d2:
            res[chiave] = None
    return res
```

Quindi usiamo `elabora_file` per costruire un istogramma di `words.txt`, per poi sottrarre:

```
parole = elabora_file('words.txt')
diff = sottrai(isto, parole)

print('Parole del libro che non si trovano nell'elenco:')
for parola in diff:
    print(parola, end=' ')
```

Ecco alcuni risultati per *Emma*:

```
Parole del libro che non si trovano nell'elenco:
 rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

Alcune parole sono nomi propri e possessivi. Altre come "rencontre" sono desuete. Ma qualcuna è davvero una parola comune che nell'elenco dovrebbe esserci!

Esercizio 13.6. *Python dispone di una struttura di dati chiamata* `set`*, o insieme, che fornisce molte operazioni comuni sugli insiemi. Al riguardo, potete leggere il Paragrafo 19.5 o la documentazione sul sito* `http://docs.python.org/3/library/stdtypes.html#types-set`*.*

Scrivete un programma che usi la sottrazione di insiemi per trovare le parole del libro che non sono nell'elenco. Soluzione: `http://thinkpython2.com/code/analyze_book2.py`*.*

13.7 Parole a caso

Per scegliere una parola a caso dall'istogramma, l'algoritmo più semplice è costruire una lista che contiene più copie di ciascuna parola, secondo la frequenza osservata, e poi estrarre a caso da questa lista:

```
def parola_caso(h):
    t = []
    for parola, freq in h.items():
        t.extend([parola] * freq)

    return random.choice(t)
```

L'espressione `[parola] * freq` crea una lista con `freq` copie della stringa `parola`. Il metodo `extend` è simile a `append`, con la differenza che l'argomento è una sequenza.

Questo algoritmo funziona, ma non è molto efficiente: ogni volta che estraete una parola, ricostruisce la lista, che è grande come il libro originale. Un ovvio miglioramento è di costruire la lista una sola volta e poi fare estrazioni multiple, ma la lista è ancora grande.

Un'alternativa è:

1. Usare `keys` per ottenere una lista delle parole del libro.
2. Costruire una lista che contiene la somma cumulativa delle frequenze delle parole (vedere l'Esercizio 10.2). L'ultimo elemento della lista è il numero totale delle parole nel libro, n.
3. Scegliere un numero a caso da 1 a n. Usare una ricerca binaria (vedere l'Esercizio 10.10) per trovare l'indice dove il numero casuale si inserirebbe nella somma cumulativa.

4. Usare l'indice per trovare la parola corrispondente nella lista di parole.

Esercizio 13.7. *Scrivete un programma che usi questo algoritmo per scegliere una parola a caso dal libro. Soluzione:* `http://thinkpython2.com/code/analyze_book3.py`.

13.8 Analisi di Markov

Scegliendo a caso delle parole dal libro, potete avere un'idea del vocabolario usato dall'autore, ma difficilmente otterrete una frase di senso compiuto:

```
this the small regard harriet which knightley's it most things
```

Una serie di parole estratte a caso raramente hanno senso, perché non esistono relazioni tra parole successive. In una frase, per esempio, è prevedibile che ad un articolo come "il" segua un aggettivo o un sostantivo, ma non un verbo o un avverbio.

Un modo per misurare questo tipo di relazioni è l'analisi di Markov che, per una data sequenza di parole, descrive la probabilità della parola che potrebbe seguire. Prendiamo la canzone dei Monty Python *Eric, the Half a Bee* che comincia così:

> Half a bee, philosophically,
> Must, ipso facto, half not be.
> But half the bee has got to be
> Vis a vis, its entity. D'you see?
>
> But can a bee be said to be
> Or not to be an entire bee
> When half the bee is not a bee
> Due to some ancient injury?

In questo testo, la frase "half the" è sempre seguita dalla parola "bee," ma la frase "the bee" può essere seguita sia da "has" che da "is".

Il risultato dell'analisi di Markov è una mappatura da ciascun prefisso (come "half the" e "the bee") in tutti i possibili suffissi (come "has" e "is").

Eseguita questa mappatura, potete generare un testo casuale partendo da qualunque prefisso e scegliendo a caso uno dei possibili suffissi. Poi, potete combinare la fine del prefisso e il nuovo suffisso per formare il successivo prefisso, e ripetere l'operazione.

Ad esempio, se partite con il prefisso "Half a," la parola successiva sarà senz'altro "bee," perché il prefisso compare solo una volta nel testo. Il prefisso successivo sarà "a bee," quindi il suffisso successivo potrà essere "philosophically", "be" oppure "due".

In questo esempio, la lunghezza del prefisso è sempre di due parole, ma potete fare l'analisi di Markov con prefissi di qualunque lunghezza.

Esercizio 13.8. *Analisi di Markov:*

1. *Scrivete un programma che legga un testo da un file ed esegua l'analisi di Markov. Il risultato dovrebbe essere un dizionario che fa corrispondere i prefissi a una raccolta di possibili suffissi. La raccolta può essere una lista, tupla o dizionario: a voi valutare la scelta più appropriata. Potete testare il vostro programma con una lunghezza del prefisso di due parole, ma dovreste scrivere il programma in modo da poter provare facilmente anche lunghezze superiori.*

2. *Aggiungete una funzione al programma precedente per generare un testo casuale basato sull'analisi di Markov. Ecco un esempio tratto da* Emma *con prefisso di lunghezza 2:*

   > *He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me? I cannot make speeches, Emma: he soon cut it all himself.*

   In questo esempio, ho lasciato la punteggiatura attaccata alle parole. Il risultato sintatticamente è quasi accettabile, ma non del tutto. Semanticamente, è quasi sensato, ma non del tutto.

   Cosa succede se aumentate la lunghezza del prefisso? Il testo casuale è più sensato?

3. *Ottenuto un programma funzionante, potete tentare un "minestrone": se combinate testi presi da due o più libri, il testo generato mescolerà il vocabolario e le frasi dei sorgenti in modi interessanti.*

Fonte: Questa esercitazione è tratta da un esempio in Kernighan e Pike, The Practice of Programming, *Addison-Wesley, 1999.*

Cercate di svolgere questo esercizio prima di andare oltre; poi potete scaricare la mia soluzione dal sito `http://thinkpython2.com/code/markov.py`. Vi servirà anche `http://thinkpython2.com/code/emma.txt`.

13.9 Strutture di dati

Utilizzare l'analisi di Markov per generare testi casuali è divertente, ma c'è anche un obiettivo in questo esercizio: la scelta della struttura di dati. Per risolverlo, dovevate infatti scegliere:

- Come rappresentare i prefissi.
- Come rappresentare la raccolta di possibili suffissi.
- Come rappresentare la mappatura da ciascun prefisso nella raccolta di suffissi.

L'ultima è facile: un dizionario è la scelta scontata per mappare da chiavi nei corrispondenti valori.

Per i prefissi, le possibili scelte sono: stringa, lista di stringhe o tuple di stringhe. Per i suffissi, un'opzione è una lista, l'altra è un istogramma (cioè un dizionario).

Quale scegliere? Per prima cosa dovete chiedervi quali tipi di operazione dovete implementare per ciascuna struttura di dati. Per i prefissi, ci serve poter rimuovere le parole all'inizio e aggiungerne in coda. Per esempio, se il prefisso attuale è "Half a," e la parola successiva è "bee," dobbiamo essere in grado di formare il prefisso successivo, "a bee".

La prima ipotesi allora potrebbe essere una lista, dato che permette di aggiungere e rimuovere elementi in modo semplice, tuttavia abbiamo anche bisogno di usare i prefissi come chiavi di un dizionario, cosa che esclude le liste. Con le tuple non possiamo aggiungere o rimuovere, ma possiamo sempre usare l'operatore di addizione per formare una nuova tupla:

```
def cambia(prefisso, parola):
    return prefisso[1:] + (parola,)
```

`cambia` prende una tupla di parole, `prefisso`, e una stringa, `parola`, e forma una nuova tupla che comprende tutte le parole in `prefisso` tranne la prima, e `parola` aggiunta alla fine.

Per la raccolta di suffissi, le operazioni che dobbiamo eseguire comprendono l'aggiunta di un nuovo suffisso (o l'incremento della frequenza di un suffisso esistente) e l'estrazione di un elemento a caso.

Aggiungere un nuovo suffisso è ugualmente semplice sia nel caso di implementazione di una lista sia di un istogramma. Estrarre un elemento da una lista è facile, da un istogramma difficile da fare in modo efficiente (vedere Esercizio 13.7).

Sinora abbiamo considerato soprattutto la facilità di implementazione, ma ci sono altri fattori da tenere in considerazione nella scelta delle strutture di dati. Una è il tempo di esecuzione. A volte ci sono ragioni teoriche per attendersi che una struttura sia più veloce di un'altra; per esempio ho già accennato che l'operatore `in` è più rapido nei dizionari che non nelle liste, almeno in presenza di un gran numero di elementi.

Ma spesso non è possibile sapere *a priori* quale implementazione sarà più veloce. Una scelta possibile è implementarle entrambe e provare quale si comporta meglio. Questo approccio è detto **benchmarking**. Un'alternativa pratica è quella di scegliere la struttura di dati più facile da implementare e vedere se è abbastanza veloce per quell'applicazione. Se è così, non c'è bisogno di andare oltre. Altrimenti, ci sono strumenti, come il modulo `profile` che è in grado di segnalare i punti in cui il programma impiega la maggior parte del tempo.

Altro fattore da considerare è lo spazio di archiviazione. Ad esempio, usare un istogramma per la raccolta di suffissi può richiedere meno spazio, perché è necessario memorizzare ogni parola solo una volta, indipendentemente da quante volte compaia nel testo. In qualche caso, risparmiare spazio significa avere un programma più veloce; in casi estremi, il programma può non funzionare affatto se provoca l'esaurimento della memoria. Ma per molte applicazioni, lo spazio è di secondaria importanza rispetto al tempo di esecuzione.

Un'ultima considerazione: in questa discussione, era sottointeso che avremmo dovuto usare una stessa struttura di dati sia per l'analisi che per la generazione. Ma siccome sono fasi separate, nulla vieta di usare un tipo di struttura per l'analisi e poi convertirlo in un'altra struttura per la generazione. Sarebbe un guadagno, se il tempo risparmiato durante la generazione superasse quello impiegato nella conversione.

13.10 Debug

Quando fate il debug di un programma, e specialmente se state affrontando un bug ostico, ci sono cinque cose da provare:

Leggere: Esaminate il vostro codice, rileggetelo e controllate che esprima esattamente quello che voi intendete dire.

Eseguire: Sperimentate facendo modifiche ed eseguendo le diverse versioni. Spesso, se visualizzate la cosa giusta al posto giusto all'interno del programma, il problema diventa evidente; magari occorre spendere un po' di tempo per inserire qualche "impalcatura".

Rimuginare: Prendetevi il tempo per pensarci su! Che tipo di errore è: di sintassi, di runtime o di semantica? Che informazioni si traggono dal messaggio di errore o dall'output del programma? Che tipo di errore potrebbe causare il problema che vedete? Quali modifiche avete fatto prima che si verificasse il problema?

Parlare a una papera di gomma: Spiegando il problema a qualcun altro, talvolta si trova la risposta ancor prima di finire di formulare la domanda. Ma spesso non serve nemmeno un'altra persona: potete semplicemente parlare ad una papera di gomma. E da qui nasce la nota tecnica chiamata **debug con la papera di gomma**. Non me lo sono inventato: date un'occhiata a `https://en.wikipedia.org/wiki/Rubber_duck_debugging`.

Tornare indietro: A un certo punto, la cosa migliore da fare è tornare sui vostri passi, annullare le ultime modifiche, fino a riottenere un programma funzionante e comprensibile. Poi rifate da capo.

I programmatori principianti a volte si fissano su uno di questi punti e tralasciano gli altri. Ciascuno di essi ha dei punti deboli.

Per esempio, leggere il codice va bene se il problema è un errore di battitura, ma non se c'è un fraintendimento concettuale. Se non capite cosa fa il vostro programma, potete leggerlo 100 volte senza riuscire a trovare l'errore, perché l'errore sta nella vostra testa.

Fare esperimenti va bene, specie se si tratta di piccoli, semplici test. Ma se fate esperimenti senza pensare o leggere il codice, potete cascare in uno schema che io chiamo "programmare a tentoni", che significa fare tentativi a casaccio finché il programma non fa la cosa giusta. Inutile dirlo, questo può richiedere un sacco di tempo.

Dovete prendervi il tempo di riflettere. Il debug è come una scienza sperimentale. Dovete avere almeno un'ipotesi di quale sia il problema. Se ci sono due o più possibilità, provate a elaborare un test che ne elimini una.

Ma anche le migliori tecniche di debug falliranno se ci sono troppi errori o se il codice che state cercando di sistemare è troppo grande e complesso. Allora l'opzione migliore è di tornare indietro e semplificare il programma, fino ad ottenere qualcosa di funzionante e che riuscite a capire.

I principianti spesso sono riluttanti a tornare sui loro passi e si spaventano all'idea di cancellare anche una singola riga di codice (anche se è sbagliata). Se vi fa sentire meglio, copiate il programma in un altro file prima di sfrondarlo, potrete così ripristinare i pezzi di codice uno alla volta.

Trovare un bug difficile richiede lettura, esecuzione, rimuginazione e a volte ritornare sui propri passi. Se rimanete bloccati su una di queste attività, provate le altre.

13.11 Glossario

deterministico: Qualità di un programma di fare le stesse cose ogni volte che viene eseguito, a parità di dati di input.

pseudocasuale: Detto di una sequenza di numeri che sembrano casuali, ma sono generati da un programma deterministico.

valore di default: Il valore predefinito di un parametro opzionale quando non viene specificato altrimenti.

sovrascrivere: Sostituire un valore di default con un argomento.

benchmarking: Procedura di scelta tra strutture di dati di vario tipo, implementando le alternative e provandole su un campione di possibili input.

debug con la papera di gomma: Fare il debug spiegando il problema ad un oggetto inanimato, come una papera di gomma. Articolare un problema può aiutare a risolverlo, nonostante la papera di gomma non sappia nulla di Python.

13.12 Esercizi

Esercizio 13.9. *Il "rango" di una parola è la sua posizione in un elenco di parole ordinate in base alla frequenza: la parola più comune ha rango 1, la seconda più comune rango 2, ecc.*

La legge di Zipf descrive una relazione tra rango e frequenza delle parole nei linguaggi naturali (`http://it.wikipedia.org/wiki/Legge_di_Zipf`), in particolare predice che la frequenza, f, della parola di rango r è:

$$f = cr^{-s}$$

dove s e c sono parametri che dipendono dal linguaggio e dal testo. Logaritmizzando ambo i lati dell'equazione, si ottiene:

$$\log f = \log c - s \log r$$

che rappresentata su un grafico con $\log r$ in ascissa e $\log f$ in ordinata, è una retta di coefficiente angolare $-s$ e termine noto $\log c$.

Scrivete un programma che legga un testo da un file, conti le frequenza delle parole e stampi una riga per ogni parola, in ordine decrescente di frequenza, con i valori di $\log f$ e $\log r$. Usate un programma a vostra scelta per costruire il grafico dei risultati e controllare se formano una retta. Riuscite a stimare il valore di s?

Soluzione: `http://thinkpython2.com/code/zipf.py`. Per avviare la mia risoluzione serve il modulo di plotting `matplotlib`*. Se avete installato Anaconda, avete già* `matplotlib`*; altrimenti potrebbe essere necessario installarlo.*

Capitolo 14

File

Questo capitolo spiega il concetto di programma "persistente", che mantiene i propri dati in archivi permanenti, e mostra come usare diversi tipi di archivi, come file e database.

14.1 Persistenza

La maggior parte dei programmi che abbiamo visto finora sono transitori, nel senso che vengono eseguiti per breve tempo e producono un risultato, ma quando vengono chiusi i loro dati svaniscono. Se rieseguite il programma, questo ricomincia da zero.

Altri programmi sono **persistenti**: sono eseguiti per un lungo tempo (o di continuo); mantengono almeno una parte dei loro dati archiviati in modo permanente, come su un disco fisso; e se vengono arrestati e riavviati, riprendono il loro lavoro da dove lo avevano lasciato.

Esempi di programmi persistenti sono i sistemi operativi, eseguiti praticamente ogni volta che un computer viene acceso, e i web server, che lavorano di continuo in attesa di richieste provenienti dalla rete.

Per i programmi, uno dei modi più semplici di mantenere i loro dati è di leggerli e scriverli su file di testo. Abbiamo già visto qualche programma che legge dei file di testo; in questo capitolo ne vedremo alcuni che li scrivono.

Un'alternativa è conservare la situazione del programma in un database. In questo capitolo mostrerò un semplice database e un modulo, `pickle`, che rende agevole l'archiviazione dei dati.

14.2 Lettura e scrittura

Un file di testo è un una sequenza di caratteri salvata su un dispositivo permanente come un disco fisso, una memoria flash o un CD-ROM. Abbiamo già visto come aprire e leggere un file nel Paragrafo 9.1.

Per scrivere un file, lo dovete aprire indicando la modalità `'w'` come secondo parametro:

```
>>> fout = open('output.txt', 'w')
```

Se il file esiste già, l'apertura in modalità scrittura lo ripulisce dai vecchi dati e riparte da zero, quindi fate attenzione! Se non esiste, ne viene creato uno nuovo.

`open` restituisce un oggetto file che fornisce i metodi per lavorare con il file.

Il metodo `write` inserisce i dati nel file.

```
>>> riga1 = "E questa qui è l'acacia,\n"
>>> fout.write(riga1)
25
```

Il valore di ritorno è il numero di caratteri che sono stati scritti. L'oggetto file tiene traccia di dove si trova, e se invocate ancora il metodo `write`, aggiunge i nuovi dati in coda al file.

```
>>> riga2 = "l'emblema della nostra terra.\n"
>>> fout.write(riga2)
30
```

Quando avete finito di scrivere, è opportuno chiudere il file.

```
>>> fout.close()
```

Se non chiudete il file, viene comunque chiuso automaticamente al termine del programma.

14.3 L'operatore di formato

L'argomento di `write` deve essere una stringa, e se volessimo inserire valori di tipo diverso in un file dovremmo prima convertirli in stringhe. Il metodo più semplice per farlo è usare `str`:

```
>>> x = 52
>>> fout.write(str(x))
```

Un'alternativa è utilizzare l'**operatore di formato**, `%`. Quando viene applicato agli interi, `%` rappresenta l'operatore modulo. Ma se il primo operando è una stringa, `%` diventa l'operatore di formato.

Il primo operando è detto **stringa di formato**, che contiene una o più **sequenze di formato**, che specificano il formato del secondo operando. Il risultato è una stringa.

Per esempio, la sequenza di formato `'%d'` significa che il secondo operando dovrebbe essere nel formato di numero intero in base decimale:

```
>>> cammelli = 42
>>> '%d' % cammelli
'42'
```

Il risultato è la stringa `'42'`, che non va confusa con il valore intero `42`.

Una sequenza di formato può comparire dovunque all'interno di una stringa, e così possiamo incorporare un valore in una frase:

```
>>> 'Ho contato %d cammelli.' % cammelli
'Ho contato 42 cammelli.'
```

Se nella stringa c'è più di una sequenza di formato, il secondo operando deve essere una tupla. Ciascuna sequenza di formato corrisponde a un elemento della tupla, nell'ordine.

L'esempio che segue usa `'%d'` per formattare un intero, `'%g'` per formattare un decimale a virgola mobile (floating-point), e `'%s'` per formattare una stringa:

```
>>> 'In %d anni ho contato %g %s.' % (3, 0.1, 'cammelli')
'In 3 anni ho contato 0.1 cammelli.'
```

Naturalmente, il numero degli elementi nella tupla deve essere pari a quello delle sequenze di formato nella stringa, ed i tipi degli elementi devono corrispondere a quelli delle sequenze di formato:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollari'
TypeError: %d format: a number is required, not str
```

Nel primo esempio, non ci sono abbastanza elementi; nel secondo, l'elemento è del tipo sbagliato.

Per saperne di più sull'operatore di formato: `https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting`. Un'alternativa più potente è il metodo di formato delle stringhe, potete leggerne la documentazione sul sito `https://docs.python.org/3/library/stdtypes.html#str.format`.

14.4 Nomi di file e percorsi

Il file sono organizzati in **directory** (chiamate anche "cartelle"). Ogni programma in esecuzione ha una "directory corrente", che è la directory predefinita per la maggior parte delle operazioni che compie. Ad esempio, quando aprite un file in lettura, Python lo cerca nella sua directory corrente.

Il modulo `os` fornisce delle funzioni per lavorare con file e directory ("os" sta per "sistema operativo"). `os.getcwd` restituisce il nome della directory corrente:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

`cwd` sta per "*current working directory*" (directory di lavoro corrente). Il risultato di questo esempio è `/home/dinsdale`, che è la directory home di un utente di nome `dinsdale`.

Una stringa come `'/home/dinsdale'`, che individua la collocazione di un file o una directory, è chiamata **percorso**.

Un semplice nome di file, come `memo.txt` è pure considerato un percorso, ma è un **percorso relativo** perché si riferisce alla directory corrente. Se la directory corrente è `/home/dinsdale`, il nome di file `memo.txt` starebbe per `/home/dinsdale/memo.txt`.

Un percorso che comincia per / non dipende dalla directory corrente; viene chiamato **percorso assoluto**. Per trovare il percorso assoluto del file, si può usare `os.path.abspath`:

I percorsi visti finora sono semplici nomi di file, quindi sono percorsi relativi alla directory corrente. Per avere invece il percorso assoluto, potete usare `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path` fornisce altre funzioni per lavorare con nomi di file e percorsi. Per esempio, `os.path.exists` controlla se un file o una cartella esistono:

```
>>> os.path.exists('memo.txt')
True
```

Se esiste, `os.path.isdir` controlla se è una directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

Similmente, `os.path.isfile` controlla se è un file.

`os.listdir` restituisce una lista dei file e delle altre directory nella cartella indicata:

```
>>> os.listdir(cwd)
['musica', 'immagini', 'memo.txt']
```

Per dimostrare l'uso di queste funzioni, l'esempio seguente "esplora" il contenuto di una directory, stampa il nome di tutti i file e si chiama ricorsivamente su tutte le sottodirectory.

```
def esplora(dirnome):
    for nome in os.listdir(dirnome):
        percorso = os.path.join(dirnome, nome)

        if os.path.isfile(percorso):
            print(percorso)
        else:
            esplora(percorso)
```

`os.path.join` prende il nome di una directory e il nome di un file e li unisce a formare un percorso completo.

Il modulo `os` contiene una funzione di nome `walk` che è simile a questa ma più versatile. Come esercizio, leggetene la documentazione e usatela per stampare i nomi dei file di una data directory e delle sue sottodirectory. Soluzione: `http://thinkpython2.com/code/walk.py`.

14.5 Gestire le eccezioni

Parecchie cose possono andare storte quando si cerca di leggere e scrivere file. Se tentate di aprire un file che non esiste, si verifica un `IOError`:

```
>>> fin = open('file_corrotto')
IOError: [Errno 2] No such file or directory: 'file_corrotto'
```

Se non avete il permesso di accedere al file:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

E se cercate di aprire una directory in lettura, ottenete:

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

Per evitare questi errori, potete usare funzioni come `os.path.exists` e `os.path.isfile`, ma ci vorrebbe molto tempo e molto codice per controllare tutte le possibilità (se "`Errno 21`" significa qualcosa, ci sono almeno 21 cose che possono andare male).

È meglio allora andare avanti e provare—e affrontare i problemi quando si presentano—che è proprio quello che fa l'istruzione `try`. La sintassi è simile a un'istruzione `if...else`:

```
try:
    fin = open('file_corrotto')
except:
    print('Qualcosa non funziona.')
```

Python comincia con l'eseguire la clausola `try`. Se tutto va bene, tralascia la clausola `except` e procede. Se si verifica un'eccezione, salta fuori dalla clausola `try` e va ad eseguire la clausola `except`.

Utilizzare in questo modo l'istruzione `try` viene detto **gestire** un'eccezione. Nell'esempio precedente, la clausola `except` stampa un messaggio di errore che non è di grande aiuto. In genere, gestire un'eccezione vi dà la possibilità di sistemare il problema, o riprovare, o per lo meno arrestare il programma in maniera morbida.

14.6 Database

Un **database** è un file che è progettato per archiviare dati. Molti database sono organizzati come un dizionario, nel senso che fanno una mappatura da chiavi in valori. La grande differenza tra database e dizionari è che i primi risiedono su disco (o altro dispositivo permanente), e persistono quando il programma viene chiuso.

Il modulo `dbm` fornisce un'interfaccia per creare e aggiornare file di database. Come esempio, creerò un database che contiene le didascalie di alcuni file di immagini.

Un database si apre in modo simile agli altri file:

```
>>> import dbm
>>> db = dbm.open('didascalie', 'c')
```

La modalità `'c'` significa che il database deve essere creato se non esiste già. Il risultato è un oggetto database che può essere utilizzato (per la maggior parte delle operazioni) come un dizionario.

Se create un nuovo elemento, `dbm` aggiorna il file di database.

```
>>> db['cleese.png'] = 'Foto di John Cleese.'
```

Quando accedete a uno degli elementi, `dbm` legge il file:

```
>>> db['cleese.png']
b'Foto di John Cleese.'
```

Il risultato è un **oggetto bytes**, ed è per questo che comincia per `b`. Un oggetto bytes è per molti aspetti simile ad una stringa. Quando approfondirete Python la differenza diverrà importante, ma per ora possiamo soprassedere.

Se fate una nuova assegnazione a una chiave esistente, `dbm` sostituisce il vecchio valore:

```
>>> db['cleese.png'] = 'Foto di John Cleese che cammina in modo ridicolo.'
>>> db['cleese.png']
b'Foto di John Cleese che cammina in modo ridicolo.'
```

Certi metodi dei dizionari, come `keys` e `items`, non funzionano con gli oggetti database, ma funziona l'iterazione con un ciclo `for`.

```
for chiave in db:
    print(chiave, db[chiave])
```

Come con gli altri file, dovete chiudere il database quando avete finito:

```
>>> db.close()
```

14.7 Pickling

Un limite di `dbm` è che le chiavi e i valori devono essere delle stringhe, oppure bytes. Se cercate di utilizzare qualsiasi altro tipo, si verifica un errore.

Il modulo `pickle` può essere di aiuto: trasforma quasi ogni tipo di oggetto in una stringa, adatta per essere inserita in un database, e quindi ritrasforma la stringa in oggetto.

`pickle.dumps` accetta un oggetto come parametro e ne restituisce una serializzazione, ovvero una rappresentazione sotto forma di una stringa (`dumps` è l'abbreviazione di "dump string", scarica stringa):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

Il formato non è immediatamente leggibile: è progettato per essere facile da interpretare da parte di `pickle`. In seguito, `pickle.loads` ("carica stringa") ricostruisce l'oggetto:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

Sebbene il nuovo oggetto abbia lo stesso valore di quello vecchio, non è in genere lo stesso oggetto:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

In altre parole, fare una serializzazione con `pickle` e poi l'operazione inversa, ha lo stesso effetto di copiare l'oggetto.

Potete usare `pickle` per archiviare in un database tutto ciò che non è una stringa. In effetti, questa combinazione è tanto frequente da essere stata incapsulata in un modulo chiamato `shelve`.

14.8 Pipe

Molti sistemi operativi forniscono un'interfaccia a riga di comando, nota anche come **shell**. Le shell sono dotate di comandi per spostarsi nel file system e per lanciare le applicazioni. Per esempio, in UNIX potete cambiare directory con il comando `cd`, visualizzarne il contenuto con `ls`, e lanciare un web browser scrivendone il nome, per esempio `firefox`.

Qualsiasi programma lanciabile dalla shell può essere lanciato anche da Python usando un **oggetto pipe**, che rappresenta un programma in esecuzione.

Ad esempio, il comando Unix `ls -l` di norma mostra il contenuto della cartella attuale (in formato esteso). Potete lanciare `ls` anche con `os.popen`[1]:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

L'argomento è una stringa che contiene un comando shell. Il valore di ritorno è un oggetto che si comporta come un file aperto. Potete leggere l'output del processo `ls` una riga per volta con `readline`, oppure ottenere tutto in una volta con `read`:

```
>>> res = fp.read()
```

Quando avete finito, chiudete la pipe come se fosse un file:

```
>>> stat = fp.close()
>>> print(stat)
None
```

Il valore di ritorno è lo stato finale del processo `ls`; `None` significa che si è chiuso normalmente (senza errori).

Altro esempio, in molti sistemi Unix il comando `md5sum` legge il contenuto di un file e ne calcola una checksum . Per saperne di più: `http://it.wikipedia.org/wiki/MD5`. Questo comando è un mezzo efficiente per controllare se due file hanno lo stesso contenuto. La probabilità che due diversi contenuti diano la stessa checksum è piccolissima (per intenderci, è improbabile che succeda prima che l'universo collassi).

Potete allora usare una pipe per eseguire `md5sum` da Python e ottenere il risultato:

```
>>> nomefile = 'book.tex'
>>> cmd = 'md5sum ' + nomefile
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

14.9 Scrivere moduli

Qualunque file che contenga codice Python può essere importato come modulo. Per esempio, supponiamo di avere un file di nome `wc.py` che contiene il codice che segue:

[1] `popen` ora è deprecato, cioè siamo invitati a smettere di usarlo e ad iniziare ad usare invece il modulo `subprocess`. Ma per i casi semplici, trovo che `subprocess` sia più complicato del necessario. Pertanto continuerò ad usare `popen` finché non verrà rimosso definitivamente.

```
def contarighe(nomefile):
    conta = 0
    for riga in open(nomefile):
        conta += 1
    return conta

print(contarighe('wc.py'))
```

Se eseguite questo programma, legge se stesso e stampa il numero delle righe nel file, che è 7. Potete anche importare il file in questo modo:

```
>>> import wc
7
```

Ora avete un oggetto modulo `wc`:

```
>>> wc
<module 'wc' from 'wc.py'>
```

L'oggetto modulo fornisce `contarighe`:

```
>>> wc.contarighe('wc.py')
7
```

Ecco come scrivere moduli in Python.

L'unico difetto di questo esempio è che quando importate il modulo, esegue anche il codice di prova in fondo. Di solito, invece, un modulo definisce solo delle nuove funzioni ma non le esegue.

I programmi che verranno importati come moduli usano spesso questo costrutto:

```
if __name__ == '__main__':
    print(contarighe('wc.py'))
```

`__name__` è una variabile predefinita che viene impostata all'avvio del programma. Se questo viene avviato come script, `__name__` ha il valore `'__main__'`; in quel caso, il codice viene eseguito. Altrimenti, se viene importato come modulo, il codice di prova viene saltato.

Come esercizio, scrivete questo esempio in un file di nome `wc.py` ed eseguitelo come script. Poi avviate l'interprete e scrivete `import wc`. Che valore ha `__name__` quando il modulo viene importato?

Attenzione: Se importate un modulo già importato, Python non fa nulla. Non rilegge il file, anche se è cambiato.

Se volete ricaricare un modulo potete usare la funzione `reload`, ma potrebbe dare delle noie, quindi la cosa più sicura è riavviare l'interprete e importare nuovamente il modulo.

14.10 Debug

Quando leggete e scrivete file, è possibile incontrare dei problemi con gli spaziatori. Questi errori sono difficili da correggere perché spazi, tabulazioni e ritorni a capo di solito non sono visibili.

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

La funzione predefinita `repr` può essere utile: riceve come argomento qualsiasi oggetto e restituisce una rappresentazione dell'oggetto in forma di stringa. Per le stringhe, essa rappresenta gli spaziatori con delle sequenze con barra inversa:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Questa funzione può quindi aiutare nel debug.

Un altro problema in cui potreste imbattervi è che sistemi diversi usano caratteri diversi per indicare la fine della riga. Alcuni usano il carattere di ritorno a capo, rappresentato da `\n`. Altri usano quello di ritorno carrello, rappresentato da `\r`. Alcuni usano entrambi. Se spostate i file da un sistema all'altro, queste incongruenze possono causare errori.

Comunque, esistono per ogni sistema delle applicazioni che convertono da un formato a un altro. Potete trovarne (e leggere altro sull'argomento) sul sito `http://it.wikipedia.org/wiki/Ritorno_a_capo`. Oppure, naturalmente, potete scriverne una voi.

14.11 Glossario

persistente: Di un programma eseguito per un tempo indefinito e che memorizza almeno parte dei suoi dati in dispositivi permanenti.

operatore di formato: Operatore indicato da `%`, che a partire da una stringa di formato e una tupla produce una stringa che include gli elementi della tupla, ciascuno nel formato specificato dalla stringa di formato.

stringa di formato: Stringa usata con l'operatore di formato e che contiene le sequenze di formato.

sequenza di formato: Sequenza di caratteri in una stringa di formato, come `%d`, che specifica in quale formato deve essere un valore.

file di testo: Sequenza di caratteri salvata in un dispositivo di archiviazione permanente come un disco fisso.

directory: Raccolta di file; è dotata di un nome ed è chiamata anche cartella.

percorso: Stringa che localizza un file.

percorso relativo: Un percorso che parte dalla cartella di lavoro attuale.

percorso assoluto: Un percorso che parte dalla cartella principale del file system.

gestire: Prevenire l'arresto di un programma causato da un errore, mediante le istruzioni `try` e `except`.

database: Un file i cui contenuti sono organizzati come un dizionario, con chiavi che corrispondono a valori.

oggetto bytes: Un oggetto simile ad una stringa.

shell: Un programma che permette all'utente di inserire comandi e di eseguirli, avviando altri programmi.

oggetto pipe: Un oggetto che rappresenta un programma in esecuzione e che consente ad un programma Python di eseguire comandi e leggere i risultati.

14.12 Esercizi

Esercizio 14.1. *Scrivete una funzione di nome* `sed` *che richieda come argomenti una stringa modello, una stringa di sostituzione, e due nomi di file. La funzione deve leggere il primo file e scriverne il contenuto nel secondo file (creandolo se necessario). Se la stringa modello compare da qualche parte nel testo del file, la funzione deve sostituirla con la seconda stringa.*

Se si verifica un errore in apertura, lettura, scrittura, chiusura del file, il vostro programma deve gestire l'eccezione, stampare un messaggio di errore e terminare. Soluzione: `http://thinkpython2.com/code/sed.py`.

Esercizio 14.2. *Se avete scaricato la mia soluzione dell'Esercizio 12.2 dal sito* `http://thinkpython2.com/code/anagram_sets.py`, *avrete visto che crea un dizionario che fa corrispondere una stringa ordinata di lettere alla lista di parole che possono essere scritte con quelle lettere. Per esempio,* `'opst'` *corrisponde alla lista* `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Scrivete un modulo che importi `anagram_sets` *e fornisca due nuove funzioni:* `arch_anagrammi` *deve archiviare il dizionario di anagrammi in uno "shelf";* `leggi_anagrammi` *deve cercare una parola e restituire una lista dei suoi anagrammi. Soluzione:* `http://thinkpython2.com/code/anagram_db.py`

Esercizio 14.3. *In una grande raccolta di file MP3 possono esserci più copie della stessa canzone, messe in cartelle diverse o con nomi di file differenti. Scopo di questo esercizio è di ricercare i duplicati.*

1. *Scrivete un programma che cerchi in una cartella e, ricorsivamente, nelle sue sottocartelle, e restituisca un elenco dei percorsi completi di tutti i file con una stessa estensione (come* `.mp3`*). Suggerimento:* `os.path` *contiene alcune funzioni utili per trattare nomi di file e percorsi.*

2. *Per riconoscere i duplicati, potete usare* `md5sum` *per calcolare la "checksum" di ogni file. Se due file hanno la stessa checksum, significa che con ogni probabilità hanno lo stesso contenuto.*

3. *Per effettuare un doppio controllo, usate il comando Unix* `diff`.

Soluzione: `http://thinkpython2.com/code/find_duplicates.py`.