Python for Informatics

Esplorando le Informazioni

Version 0.0.8-d2

Charles Severance

Cronologia di stampa:

Ottobre 2013: revisione maggiore al Capitolo 13 e 14 per spostamento su JSON ed uso di OAuth. Aggiunto nuovo capitolo Visualizzazione.

Settembre 2013: pubblicazione su Amazon CreateSpace.

Gennaio 2010: pubblicazione utilizzando University of Michigan Espresso Book.

Dicembre 2009: revisione maggiore ai capitoli 2-10 da *Think Python: How to Think Like a Computer Scientist* e scrittura dei capitoli 1 e 11-15 per la produzione di *Python for Informatics: Exploring Information*.

Giugno 2008: revisione maggiore, cambio titolo in *Think Python: How to Think Like a Computer Scientist*.

Agosto 2007: revisione maggiore, cambio titolo a *How to Think Like a (Python) Programmer*.

Aprile 2002: prima edizione di *How to Think Like a Computer Scientist*.

La sorgente LATEX per la versione *Think Python: How to Think Like a Computer Scientist* di questo libro è disponibile su http://www.thinkpython.com.

Traduzione a cura:
Mauro di Blasi
Barbara Civati
Cristiano Ferrara
Gianluca Manganiello
Roberto Picco
Mauro Toselli

Prefazione

Python for Informatics : il remix di un Open Book

E' abbastanza naturale per gli accademici, che si sentono dire continuamente “pubblica o muori”, voler sempre creare da zero qualcosa che sia una loro nuova creazione. Questo libro è un esperimento: non partire da zero, ma invece "remixare" il libro dal titolo "Think Python: Come pensare come uno scienziato informatico" scritto da Allen B. Downey, Jeff Elkner e altri.

Nel dicembre del 2009, mi stavo preparando a tenere il corso “SI502 - Networked Programming” presso l'Università del Michigan per il quinto semestre consecutivo e ho deciso che era tempo di scrivere un libro di testo su Python che si concentrasse sull'esplorazione dei dati invece che sulla comprensione di algoritmi ed astrazioni. Il mio obiettivo nel SI502 è quello di insegnare le tecniche fondamentali di analisi dei dati utilizzando Python. Pochi dei miei studenti avevano in progetto di diventare programmatori professionisti. Altri invece pianificavano di diventare bibliotecari, manager, avvocati, biologi, economisti, o altro, e desideravano imparare ad utilizzare abilmente le tecnologie nei rispettivi campi professionali.

Non ho mai trovato per il mio corso un libro su Python che fosse perfettamente orientato alla gestione dei dati, così ho deciso di scriverlo. Fortunatamente, tre settimane prima che iniziassi a lavorarci approfittando delle vacanze, in una riunione di facoltà il Dr. Atul Prakash mi ha mostrato il libro "Think Python" che lui stesso aveva usato per il suo corso. Si trattava di un testo di Informatica ben scritto, focalizzato su dirette e brevi spiegazioni dirette che facilitano l'apprendimento.

La struttura complessiva libro è stata modificata per arrivare a gestire i problemi di analisi dei dati il più rapidamente possibile e per fornire una serie di esercizi ed esempi ed esercizi in merito fin dall'inizio.

I capitoli 2-10 sono simili "Think Python", ma sono state fatte importanti modifiche. Gli esempi e gli esercizi orientati alla gestione di numeri sono stati sostituiti con esercitazioni orientati ai dati. Gli argomenti sono presentati in un ordine tale da fornire soluzioni di analisi dei dati via via sempre più sofisticate. Alcuni argomenti come "try" e "except" sono stati anticipati e presentati come parte del capitolo sull'esecuzione condizionale. Piuttosto che essere trattate già dall'inizio in maniera astratta, le funzioni sono state trattate più superficialmente sino a che non sono diventate necessarie per gestire la complessità dei programmi. Quasi tutte le funzioni definite dall'utente sono state rimosse dai codici di esempio ed esercitazione al di fuori del Capitolo 4.

La parola " ricorsivo"[1] non viene mai utilizzata in nessuna parte del libro.

Nei capitoli 1 e 11-16, tutto il materiale è nuovo di zecca, è focalizzato sull'uso di Python in applicazioni nel mondo reale e fornisce semplici esempi per l'analisi dei dati, comprendendo regolari espressioni per la ricerca e l'analisi, automatizzazione delle attività sul computer, il recupero dei dati attraverso la rete, prelievo di dati da pagine web, utilizzo di servizi web, analisi di dati in formato XML e JSON, e la creazione e l'utilizzo di database utilizzando lo Structured Query Language.

L'obiettivo finale di tutti questi cambiamenti è il passaggio da una Computer Science ad un'informatica il cui focus è quello di includere in un corso di primo livello solo quegli argomenti che potranno tornare utili anche a coloro che non sceglieranno di diventare programmatori professionisti.

Gli studenti che troveranno questo libro interessante e che desiderano esplorare ulteriormente l'argomento dovrebbero considerare il libro di Allen B. Downey "Think Python". Date le molte sovrapposizioni tra i due libri, gli studenti saranno in grado di acquisire rapidamente alcune ulteriori competenze nei settori addizionali sulla tecnica di programmazione tecnica e di pensiero algoritmico che sono parte di "Think Python". Inoltre, dato che i due libri hanno uno stile di scrittura molto simile sarà facile muoversi all'interno del libro.

Come detentore del copyright su "Think Python", Allen mi ha dato il permesso di cambiare la licenza del materiale dal suo libro che viene incluso in questo libro da

GNU Free Documentation License alla più recente di Creative Commons Attribuzione-Condividi allo stesso modo. Questo segue il generale cambiamento nelle licenze di documentazione aperta che si stanno spostando da GFDL a CC BY-SA (vedi Wikipedia). L'utilizzo della licenza CC BY-SA indica ai fruitori dell'opera che essa può essere utilizzata, diffusa e anche modificata liberamente, pur nel rispetto di alcune condizioni essenziali e rende ancora più semplice ai nuovi autori riutilizzare di questo materiale.

Ritengo che questo libro sia un esempio del perché i materiali aperti sono così importanti per il futuro della formazione, voglio ringraziare Allen B. Downey e la Cambridge University Press per la loro decisione lungimirante nel rendere il libro disponibile sotto un open Copyright. Spero che siano soddisfatti dei risultati del mio impegno e, mi auguro lo sia anche il lettore.

---

[1] ad eccezione, ovviamente, di questa riga.

Vorrei ringraziare Allen B. Downey e Lauren Cowles per il loro aiuto, la pazienza, e la guida nell'affrontare e risolvere i problemi di copyright circa questo libro.

Charles Severance

www.dr-chuck.com

Ann Arbor, MI , USA

9 Settembre 2013

Charles Severance è Clinical Associate Professor presso l'Università del Michigan - School of Information.

Preface for “Think Python”

La strana storia di “Think Python”

(Allen B. Downey)

Nel Gennaio 1999 mi stavo preparando a tenere un corso introduttivo sulla programmazione in Java. Lo avevo già tenuto tre volte ed ero frustrato. La percentuale di fallimento del corso era molto alta ed anche gli studenti che completavano il corso con successo il livello generale era troppo basso.

Uno dei problemi che avevo notato erano i libri di testo. Troppo voluminosi, con troppi dettagli non necessari su Java e non abbastanza istruzioni di alto livello su come scrivere i programmi. E tutti risentivano dell’ effetto botola: cominciavano tranquillamente, prcedevano gradualmente e circa al Capitolo 5 la botola si apriva e in molti vi cadevano. Gli studenti si ritrovavano con troppo materiale nuovo, tutto insieme e troppo in fretta e io dovevo spendere il resto del semestre a rimettere insieme i pezzi.

Due settimane prima dell’inizio del corso, ho deciso di scrivere un libro di testo. I miei obiettivi erano:

• Essere conciso. Per gli studenti è molto meglio leggere 10 pagine che non leggerne 50.

• Prestare attenzione al vocabolario. Ho certato di minizzare il gergo e definire ogni termine sin dal primo uso.

• Procedere gradualmente. Per evitare trappole ho preso gli argomenti più difficili e li ho separati in piccoli passi.

• Focalizzarsi sulla programmazione, non sul linguaggio. Ho incluso la minima selezione utile di Java ed ho lasciato fuori il resto.

Mi serviva un titolo e quasi per scherzo ho scelto “How to Think Like a Computer Scientist”.

La prima versione era grezza ma funzionava. Gli studenti leggevano e capivano annastanza per fare in modo che spendessi il tempo in classe sugli argomenti più ostici, importanti e principalmente facendo far pratica agli studenti.

Ho rilasciato il libro sotto GNU Free Documentation License, che permette agli utenti di copiare, modificare e ridistribuire il libro.

Quello che è successo dopo è la parte migliore. Jeff Elkner, un insegnate di scuola

superiore in Virginia, ha adottato il mio libro e lo ha tradotto per Python. Mi ha inviato una copia del suo riadattamento ed ho avuto la strana esperienza dell'impararare Python leggendo il mio stesso libro.

Jeff ed io abbiamo rivisto il libro, incorporato uno studio di ChrisMeyers, e nel 2001

abbiamo rilasciato "How to Think Like a Computer Scientist: Learning with Python", anch'esso sotto GNU Free Documentation License. Come Green Tea Press, ho pubblicato il libro ed iniziato a venderne copie via Amazon.com e la libreria del college. Altri libri di Green Tea Press sono disponibili su greenteapress.com.

Nel 2003 ho iniziato a insegnare all'Olin College ed ho insegnato Python per la prima volta. Il contrasto con Java era inmpressionante. Gli studenti avevano meno difficoltà, imparavano di più, lavoravano a progetti più interessanti e, in generale, si divertivano molto di più.

Nei seguenti 5 anni ho continuato a sviluppare il libro, correggere gli errori, migliorare gli esempi e ad aggiungere materiale, specialmente gli esercizi. Nel 2008 ho iniziato a lavorare ad una importante revisione e allo stesso tempo sono stato contattato da un editore alla Cambridge University Press che era intenzionato a pubblicare la nuova versione del libro. Tempismo eccellente.

Spero che apprezzerete lavorare con questo libro, che vi aiuterà ad imparare a programmare ed a pensare, almeno un pò, come un Computer Scientist.

Ringraziamenti per “Think Python”

(Allen B. Downey)

Prima di tutto, ringrazio Jeff Elkner, che ha tradotto il mio libro su Java in Python, che dato il via a questo progetto e che mi ha introdotto a quello che sarebbe diventato il mio linguaggio favorito..

Ringrazio anche Chris Meyers, che ha contribuito a molte sezioni di “How to Think Like a Computer Scientist”.

E ringrazio la Free Software Foundation per aver sviluppato la GNU Free Documentation License, che ha reso la mia collaborazione con Jeff e Chris possibile.

Ringrazio inoltre i redattori di Lulu che hanno lavorato su “How to Think Like a Computer Scientist”.

Ringrazio tutti gli studenti che hanno lavorato alle prime versioni di questo libro e tutti coloro che hanno contribuito (elencati in Appendice) e che mi hanno spedito le correzioni e suggerimenti.

E ringrazio mia moglie Lisa per il suo lavoro su questo libro, Green Tea Press, e tutti gli altri.

Allen B. Downey

Needham MA

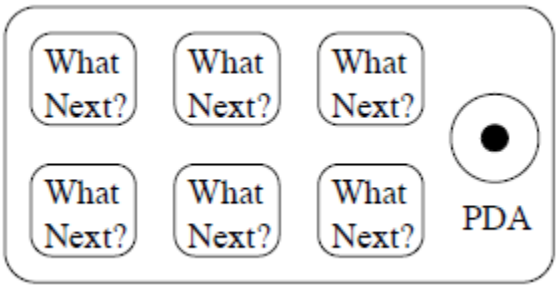
Allen Downey è Associate Professor of Computer Science al Franklin W. Olin College of Engine

Capitolo 1

Perché dovresti imparare a scrivere programmi?

Scrivere programmi (o programmare) è un'attività molto creativa e gratificante. Puoi scrivere programmi per molte ragioni che vanno dal guadagnarsi da vivere al risolvere complessi programmi di analisi dei dati al divertirsi aiutando qualcuno a risolvere un problema. Questo libro dà per scontato che ognuno abbia bisogno di sapere come programmare e che una volta imparato a farlo, troverà come impiegare questa nuova abilità.

Siamo circondati nella vita quotidiana da computers, siano essi portatili o cellulari. Possiamo pensare a questi computers come i nostri "assistenti personali" che possono farsi carico di molte cose al nostro posto. L'hardware nel nostro computer è essenzialmente costruito per porci continuamente la stessa domanda, "Cosa vuoi che faccia ora?"



I programmatori aggiungono un sistema operativo e un insieme di applicazioni all'hardware, e noi ci troviamo con un assistente personale digitale che è piuttosto utile e capace di svolgere molti compiti diversi.

I nostri computers sono veloci e hanno grandi quantità di memoria e potrebbero essere molto utili per noi se solo conoscessimo il linguaggio con cui spiegare al computer cosa noi vorremmo "fare dopo". Se conoscessimo questo linguaggio potremmo dire al computer di compiere azioni ripetitive per conto nostro. Curiosamente, i tipi di cose che i computers fanno meglio sono le cose che noi umani troviamo noiose e alienanti.

Per esempio, osserva i primi tre paragrafi di questo capitolo e dimmi qual è la parola più usata e quante volte essa è ripetuta. Benché tu sia in grado di leggere e capire le parole

in pochi secondi, contarle è piuttosto difficile perché non è il tipo di compito per cui la mente umana è progettata. Per un computer è vero il contrario, leggere e capire un testo scritto su un pezzo di carta è difficile mentre contare le parole e dire quante volte ricorre la parola più usata, è un compito molto agevole:

```
python words.py
Enter file:words.txt
to 16
```

Il nostro "assistente personale per le analisi delle informazioni" ci ha detto velocemente che la parola "to" era usata 16 volte nei primi tre paragrafi di questo capitolo.

Questo stesso fatto che i computers sono bravi a fare le cose che agli umani riescono peggio è il motivo per cui dovresti imparare il "linguaggio dei computer". Una volta acquisito questo linguaggio, potrai delegare i compiti noiosi al tuo compagno (il computer), dedicando il tuo tempo alle cose che ti riescono meglio. Tu porterai creatività, intuizione e inventiva in questa collaborazione.

1.1 Creatività e motivazione

Anche se questo libro non si rivolge a programmatori professionisti, il lavoro di programmatore può essere molto gratificante, sia a livello finanziario che personale. Creare programmi belli, utili ed eleganti affinché gli altri ne traggano beneficio è un'attività molto creativa. Il tuo computer o il tuo PDA spesso contengono molti programmi diversi creati da molti gruppi di programmatori, ciascuno in lotta per ottenere la tua attenzione e il tuo interesse. Essi fanno del loro meglio per venire incontro alle tue necessità e fornirti una straordinaria esperienza.

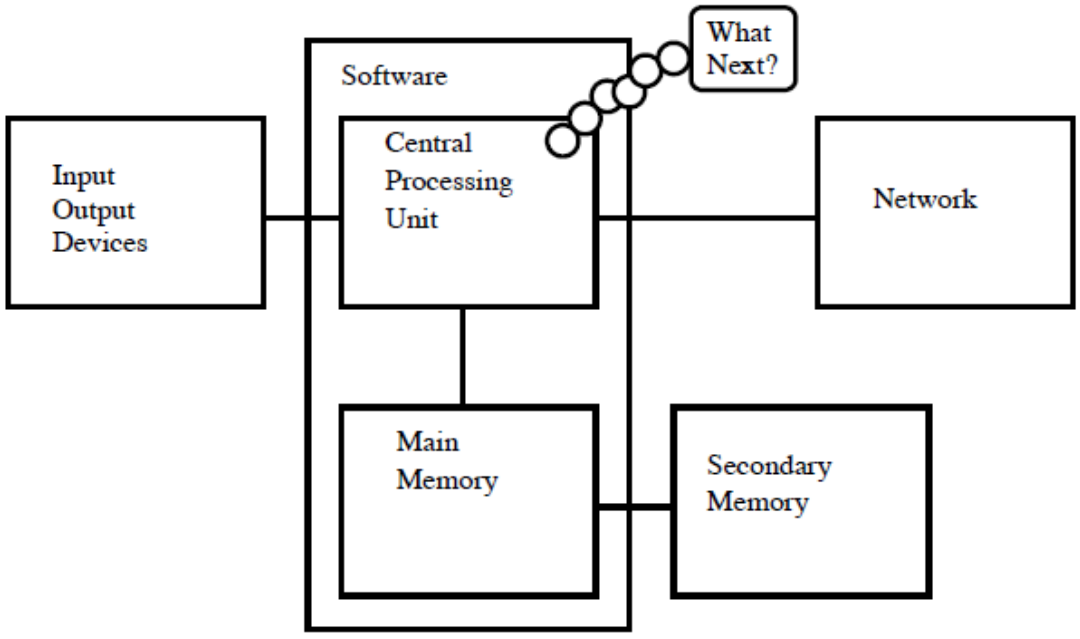
In alcune situazioni, quando scegli un programma, i programmatori sono ricompensati grazie alla tua scelta.

Se pensiamo ai programmi come il prodotto creativo di un gruppo di programmatori, forse l'immagine seguente rappresenta una versione più intelligente del tuo PDA:

Per il momento, la nostra motivazione primaria non sarà fare soldi o soddisfare l'utente finale, ma egoisticamente essere più produttivi nel maneggiare i dati e le informazioni che incontriamo nella nostra vita. Quando inizi per la prima volta, sei contemporaneamente il programmatore e l'utente finale del tuo programma. Man mano che acquisisci competenza come programmatore e la programmazione diventa sempre più un processo creativo per te, potrà nascere in te il desiderio di programmare per gli altri.

1.2 Architettura hardware di un computer

Prima di iniziare ad imparare il linguaggio con cui dare istruzioni ai computers per sviluppare un software, dobbiamo apprendere un paio di nozioni su come i computers sono costruiti. Se smontassi il tuo computer o il tuo cellulare per vedere come sono fatti, troveresti le seguenti parti:



La definizione ad alto livello di queste parti è la seguente:

- L'**unità di elaborazione centrale** (o CPU) è quella parte del computer che è costruita per essere ossessionata dalla domanda "Cosa viene dopo?". Se il tuo computer è targato a 3.0 Gigahertz, significa che la CPU ti chiederà "Cosa viene dopo?" tre miliardi di volte al secondo. Dovrai imparare a parlare velocemente per tenere il passo con la CPU.

- La **memoria principale** è usata per archiviare informazioni di cui la CPU ha immediato bisogno. La memoria principale è veloce quasi quanto la CPU. Bisogna ricordare però che le informazioni archiviate nella memoria centrale svaniscono quando il computer viene spento.

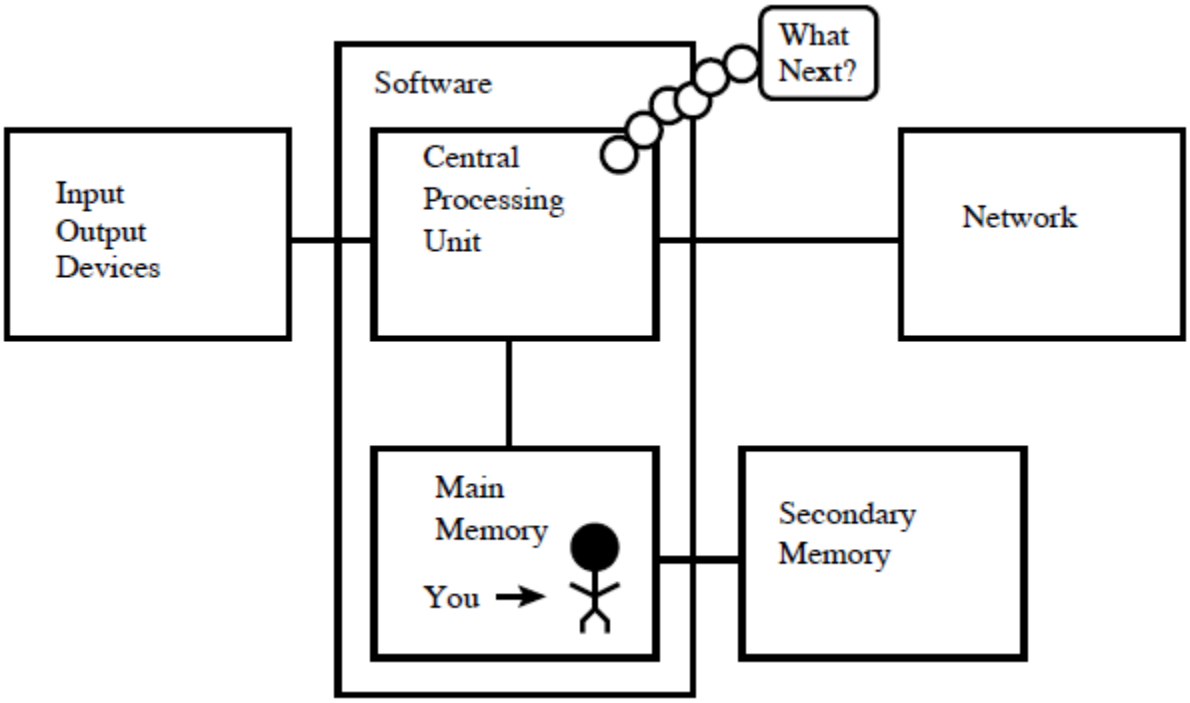
- La **memoria secondaria** è usata per archiviare informazioni, ma a differenza della memoria principale è molto più lenta. Il vantaggio della memoria secondaria è che può mantenere le informazioni anche quando il computer non è alimentato. Esempi di memoria secondaria sono gli hard disk o le memorie flash (tipicamente contenute nelle chiavette USB e nei lettori MP3 portatili).

- Le **periferiche di input e output** sono semplicemente il nostro monitor, la tastiera, il mouse, il microfono, gli altoparlanti, il touchpad, ecc. Essi rappresentano i modi che abbiamo di interagire con il computer.

- Oggigiorno la maggior parte dei computers ha anche una **connessione di rete** per recuperare informazioni da una rete. Possiamo pensare ad una rete come un posto molto lento in cui salvare e recuperare informazioni che potrebbe non essere sempre accessibile. In questo senso la rete è una forma più lenta e a volte meno affidabile di **memoria secondaria**.

La maggior parte dei dettagli su come questi componenti interagiscono è di competenza dei produttori di hardware, ma è bene avere un lessico comune per parlare di questi componenti in maniera corretta quando scriviamo i nostri programmi.

Come programmatore, il tuo lavoro nell'usare e orchestrare ciascuna di queste risorse per risolvere il problema che devi affrontare e analizzare i dati di cui hai bisogno. Come programmatore, nella maggior parte dei casi "parlerai" alla CPU e le dirai cosa fare dopo. A volte dirai alla CPU di usare la memoria principale, la secondaria, la rete o le periferiche di input/output.



Tu devi essere la persona che risponde alla domanda della CPU "Cosa viene dopo?". Sarebbe però molto scomodo rimpicciolirsi ad un'altezza di 5mm e infilarti nel computer così da poter lanciare comandi con una velocità di 3 miliardi di operazioni al secondo. Per questo devi scrivere le tue istruzioni in anticipo. Noi chiamiamo queste istruzioni **programma** e l'atto di scrivere queste istruzioni **programmazione**.

1.3 Capire la programmazione

Nel proseguo di questo libro cercheremo di trasformarti in una persona esperta nella tecnica di programmazione. Alla fine sarai un **programmatore** --- forse non un programmatore professionista, ma almeno avrai la capacità di affrontare un problema di analisi dei dati e sviluppare un programma per risolverlo.

Per certi versi, hai bisogno di due abilità per essere un programmatore:

- In primo luogo è necessario conoscere il linguaggio di programmazione (Python) - è necessario conoscerne il vocabolario e la grammatica. È necessario che tu sia in grado di pronunciare correttamente le parole in questo nuovo linguaggio e di costruire "frasi" sintatticamente corrette in questo nuovo linguaggio.

- In secondo luogo è necessario "raccontare una storia". Nello scrivere una storia, si combinano parole e frasi per dare un'idea al lettore. C'è tecnica e arte nella costruzione del racconto e la capacità di scrivere una storia si migliora stilando

qualche testo e ottenendone un feedback. Nella programmazione, il nostro programma è la "storia" e il problema che si sta tentando di risolvere è "l'idea".

Una volta imparato un linguaggio di programmazione come Python, troverai molto più facile imparare una seconda lingua di programmazione come JavaScript o C + +. Il nuovo linguaggio di programmazione ha un vocabolario e una grammatica molto diversi, ma una volta appresa la capacità di risolvere problemi (*problem solving)*, essa sarà la stessa per tutti i linguaggi di programmazione.

Imparerai il "vocabolario" e le "frasi" di Python abbastanza rapidamente. Ci vorrà più tempo prima di essere in grado di scrivere un programma coerente per risolvere un nuovo problema. Noi insegniamo la programmazione nello stesso modo in cui insegniamo a scrivere. Iniziamo a leggere e spiegare i programmi, poi scriviamo programmi semplici e via via programmi sempre più complessi. Ad un certo punto si "raggiunge l'illuminazione" e si riescono a vedere i modelli ed a comprendere più naturalmente come affrontare un problema e scrivere un programma che lo risolva. Una volta arrivato a quel punto, la programmazione diventa un processo molto piacevole e creativo.

Iniziamo con il vocabolario e la struttura dei programmi Python. Sii paziente quando i semplici esempi ti ricorderanno i tempi in cui iniziasti a leggere.

1.4 Parole e frasi

A differenza dei linguaggi umani, il vocabolario Python è in realtà piuttosto ristretto. Questo "vocabolario" è rappresentato da parole dette **"parole riservate"**. Esse hanno un significato molto speciale per Python. Quando Python incontra queste parole in un programma, esse hanno uno ed un solo significato per Python. In seguito, quando scriverai i tuoi programmi, inventerai le tue parole a cui darai un significato e chiamerai **variabili**. Avrai grande libertà nella scelta dei nomi per le variabili, ma non è possibile utilizzare una qualsiasi delle parole riservate di Python come nome per una variabile.

In un certo senso, quando alleniamo un cane, vorremmo usare parole speciali come, "seduto", "fermo", e "afferra". Anche quando parli ad un cane e di non utilizzi delle parole riservate, il cane ti guarda con lo sguardo perplesso e confuso finché non usi una parola riservata. Ad esempio, se dici "Vorrei che più persone camminassero per migliorare la loro salute generale", quello che la maggior parte dei cani probabilmente sente è: "bla bla bla **camminare** bla bla bla bla". Questo perché "camminare" è una parola riservata in lingua-cane. Molti potrebbero insinuare che la lingua tra umani e

gatti non ha parole riservate[2].

Le parole riservate nella lingua con cui gli umani parlano con Python sono:

```
and   del   for   is   raise
assert   elif   from   lambda   return
break   else   global   not   try
class   except   if   or   while
continue   exec   import   pass   yield
def   nally   in   print
```

A differenza di un cane, Python è perfettamente addestrato e se dici "try", Python eseguirà "try" tutte le volte senza fallire.

Impareremo queste parole riservate e come esse sono utilizzati a tempo debito, per ora ci concentreremo sull'equivalente Python di "parlare" (in una lingua umano-cane). La cosa bella di conversare con Python è che possiamo anche dirgli che cosa dire passandogli un messaggio tra virgolette:

```
print 'Hello world!'
```

Abbiamo scritto la nostra prima frase sintatticamente corretta in Python. La nostra frase inizia con la parola riservata print seguita da una stringa di testo scelta da noi racchiusa da apici semplici.

1.5 Dialogare con Python

Ora che conosciamo una semplice frase in Python, abbiamo bisogno di sapere come iniziare una conversazione con Python per testare le nostre nuove abilità linguistiche.

Prima di poter conversare con Python, devi prima installare Python sul tuo computer e imparare come lanciarlo. Scenderemmo troppo in dettaglio per questo capitolo, quindi ti suggerisco di consultare [www.pythonlearn.com] dove ho lasciato istruzioni dettagliate e degli screencast su come installare e avviare Python su sistemi Macintosh e Windows. Ad un certo punto ti troverai su una finestra terminale o sul command prompt di Windows, scriverai **python** e l'interprete Python verrà lanciato in modalità interattiva, mostrandosi più o meno così:

---

[2] http://xkcd.com/231/

```
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Il prompt `>>>` è il modo dell'interprete Python di chiederti "Cosa vuoi che faccia adesso?". Python è pronto a conversare con te. Tutto quello che devi sapere è come parlare la lingua di Python, in seguito potrai iniziare a colloquiare.

Diciamo, per esempio, che non conosci nemmeno le parole o le frasi più elementarei in Python. Potresti usare la frase standard the usano gli astronauti quando atterrano su un lontano pianeta e cercano di comunicare con i nativi del luogo:

```
>>> I come in peace, please take me to your leader
  File "<stdin>", line 1
    I come in peace, please take me to your leader
         ^
SyntaxError: invalid syntax
>>>
```

Non si sta mettendo bene. A meno che tu non pensi a qualcosa velocemente, gli abitanti del pianeta sono propensi ad infilzarti con le loro lance, metterti su uno spiedo, arrostirti su un fuoco e mangiarti per cena.

Fortunatamente ti sei portato una copia di questo libro e sfogliandolo fino a questa pagina scrivi ancora:

```
>>> print 'Hello world!'
Hello world!
```

Va molto meglio, quindi tenti di comunicare ancora:

```
>>> print 'You must be the legendary god that comes from the sky'
You must be the legendary god that comes from the sky
>>> print 'We have been waiting for you for a long time'
We have been waiting for you for a long time
```

```
>>> print 'Our legend says you will be very tasty with mustard'
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
  File "<stdin>", line 1
    print 'We will have a feast tonight unless you say
                                                      ^
SyntaxError: EOL while scanning string literal
>>>
```

La conversazione stava andando bene, poi hai commesso un errore piccolissimo con il linguaggio e sono rispuntate le lance.

A questo punto dovresti aver capito che, se da un lato Python è sorprendentemente complesso, potente e molto puntiglioso sulla sintassi che usi per comunicare con lui, dall'altro lato non è intelligente. Tu stai di fatto avendo una conversazione con te stesso usando la sintassi appropriata.

In un certo senso, quando usi un programma scritto da qualcun altro, la conversazione è tra te e i programmatori che agiscono da intermediari tra te e Python. Python è un modo per i programmatori di definire come la conversazione dovrebbe procedere. In un paio di capitoli sarai tu uno di quei programmatori e userai Python per parlare agli utenti del tuo programma.

Prima di terminare la nostra prima conversazione con l'interprete Python, devi conoscere il modo appropriato per dire "arrivederci" agli abitanti del pianeta Python

```
>>> good-bye
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined

>>> if you don't mind, I need to leave
  File "<stdin>", line 1
    if you don't mind, I need to leave
             ^
SyntaxError: invalid syntax

>>> quit()
```

Puoi notare che l'errore è diverso per i primi due tentativi. Il secondo errore è diverso

perché **if** è una parola riservata e Python ha notato questa parola e ha pensato che noi stessimo dicendo qualcosa, trovando però la sintassi del costrutto non corretta.

Il modo corretto di dire "arrivederci" a Python è scrivere **quit()** al prompt interattivo. Ci avresti messo un po' ad arrivarci da solo, quindi avere un libro a portata di mano ti è stato utile.

1.6 Terminologia: interprete e compilatore

Python è un linguaggio di **alto livello** pensato per essere scritto e letto in modo intuitivo dagli umani e facilmente letto e interpretato dai computers. Altri linguaggi ad alto livello sono: Java, C++, PHP, Ruby, Basic, Perl, JavaScript e molti altri. L'hardware contenuto nella CPU non è in grado di capire alcuno di questi linguaggi ad alto livello.

La CPU comprende un linguaggio che possiamo chiamare **linguaggio-macchina** . Il linguaggio-macchina è molto semplice e onestamente molto noioso da scrivere perché è rappresentato solo da zeri e uno.

```
01010001110100100101010000001111
11100110000011101010010101101101
...
```

Il linguaggio macchina sembra molto semplice a prima vista poiché si tratta solo di zeri e uno, ma la sua sintassi è ancor più complicata e molto più intricata di quella di Python. Per questo motivo solo pochi programmatori usano il linguaggio macchina. Per facilitarci il compito si costruiscono diversi traduttori per consentire ai programmatori di scrivere usando linguaggi ad alto livello come Python o JavaScript e si lascia il compito a questi traduttori di convertire i programmi in linguaggio macchina per l'esecuzione in CPU.

Dal momento che il linguaggio macchina è legato all'hardware del computer, esso non è **portabile** su diversi tipi di hardware. Al contrario, i programmi scritti in linguaggi di alto livello possono essere usati su macchine diverse usando un interprete diverso sulla nuova macchina o ri-compilando il codice per creare una versione in linguaggio macchina compatibile con il nuovo hardware.

Questi traduttori di linguaggio di programmazione rientrano in due categorie generali: (1) interpreti e (2) compilatori.

Un **interprete** legge il codice sorgente di un programma come è stato scritto dal programmatore, lo passa al setaccio e interpreta le istruzioni al volo. Python è un interprete e quando eseguiamo Python in modalità interattiva possiamo scrivere una

riga di Python (una frase) e Python la processa immediatamente, rendendosi disponibile per un nostro nuovo input.

Alcune delle linee di Python dicono a Python che tu vuoi ricordare alcuni valori per un uso futuro. Abbiamo bisogno di scegliere un nome affinchè quel valore sia ricordato e sia possibile richiamarlo in seguito. Noi usiamo il termine variabile per indicare le etichette che usiamo per riferirci ai valori archiviati.

```
>>> x = 6
>>> print x
6
>>> y = x * 7
>>> print y
42
>>>
```

In questo esempio chiediamo a Python di ricordare il valore 6 e di usare l'etichetta x così da poter recuperare il valore in seguito. Verifichiamo che Python abbia memorizzato il valore usando **print** . Chiediamo poi a Python di recuperare x e di moltiplicarlo per 7 e di salvare il risultato in y. Chiediamo infine a Python di stampare il valore corrente di y .

Anche se stiamo scrivendo queste istruzioni in Python una linea alla volta, Python le tratta come una sequenza ordinata di istruzioni in cui le ultime istruzioni sono in grado di recuperare i dati generati dalle istruzioni precedenti. Stiamo scrivendo il nostro primo semplice paragrafo con quattro frasi in un ordine logico e significativo.

E' nella natura di un interprete essere in grado di sostenere una conversazione interattiva come mostrato in precedenza. A un compilatore deve essere passato l'intero programma in un file e di seguito esso esegue il processo di traduzione del codice sorgente ad alto livello nel codice macchina e alla fine il compilatore restituisce un file con il codice macchina pronto per una futura esecuzione.

Se possiedi un sistema Windows, spesso questi programmi eseguibili in linguaggio macchina hanno un suffisso ".exe" o ".dll" che stanno rispettivamente per "eseguibile" e "libreria caricabile dinamicamente". In Linux e in Macintosh non ci sono suffissi che identificano chiaramente un file come eseguibile.

Se tu aprissi un file eseguibile con un editor di testo, esso sembrerebbe completamente confuso e illeggibile:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
```

```
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
.....
```

Non è facile leggere o scrivere linguaggio macchina, quindi è bello avere **interpreti** e **compilatori** che ci consentono di scrivere in un linguaggio ad alto livello come Python o C.

Ora, a questo punto della discussione su interpreti e compilatori, dovresti porti delle domande sull'interprete di Python. In quale linguaggio è scritto? E' scritto in un linguaggio compilato? Quando scriviamo Python cosa succede realmente?

L'interprete Python è scritto in un linguaggio ad alto livello chiamato "C". Puoi sbirciare il codice sorgente dell'interprete di Python andando su www.python.org e navigare fino al codice sorgente. In buona sostanza Python è un programma a sua volta e è compilato in codice macchina e quando lo installi sul tuo computer (o il rivenditore lo fa per te) tu copi nel tuo sistema una copia di codice macchina del programma Python tradotto. In Windows l'eseguibile per Python è molto probabilmente in un file con un nome simile a:

```
C:\Python27\python.exe
```

Questo è più di quello che tu debba sapere per essere un programmatore Python, ma a volte è utile rispondere a queste piccole domande sin dall'inizio.

1.7 Scrivere un programma

Digitare comandi nell'interprete Python è un gran bel modo per sperimentare le caratteristiche di questo linguaggio, ma non è consigliabile per affrontare problemi più complessi.

Quando vogliamo scrivere un programma, usiamo un editor di testo per scrivere istruzioni Python in un file, file che per definizione viene chiamato script. Per convenzione gli **script** di Python hanno nomi che finiscono con **.py**

Per eseguire lo script, devi comunicare all'interprete di Python il nome del file. In una finestra di comando Unix o Windows , scriveresti python hello.py come segue:

```
csev$ cat hello.py
print 'Hello world!'
```

```
csev$ python hello.py
Hello world!
csev$
```

"csev$" è il prompt del sistema operativo e il comando cat hello.py ci mostra che il file hello.py contiene una linea di codice Python per stampare una stringa.

Invochiamo l'interprete Python e gli diciamo di leggere il codice sorgente dal file hello.py anziché di richiederci i comandi in modo interattivo.

Nota che non c'è necessità di scrivere quit() alla fine del programma Python contenuto nel file. Quando Python legge il tuo codice sorgente dal file, sa che deve fermarsi quando raggiunge la fine del file.

1.8 Cos'è un programma?

La definizione di un programma è, ai minimi termini, una sequenza di istruzioni Python che sono state assiemate per produrre un risultato. Anche il nostro semplice script hello.py è un programma. E' un programma di una sola riga e non particolarmente utile, ma è pur sempre un programma Python, stando alla definizione.

Potrebbe essere più semplice comprendere cos'è un programma pensando prima al problema che il programma dovrebbe risolvere e poi analizzando il programma stesso deputato a risolverlo.

Immaginiamo che tu stia facendo una ricerca in ambito Social Computing sui post di Facebook a tu sia interessato nelle parole più frequentemente usate in una serie di post. Potresti stampare il flusso di messaggi di Facebook e scorrere tutto il testo alla ricerca della parola più usata, ma sarebbe un lavoro molto lungo e darebbe adito facilmente ad errori. Faresti meglio a scrivere un programma in Python per gestire il compito velocemente e accuratamente così potresti trascorrere il weekend facendo qualcosa di divertente.

Per esempio osserva il seguente testo su un pagliaccio e un'auto. Guarda il testo e cerca di trovare la parola più usata e quante volte ricorre.

```
the clown ran after the car and the car ran into the tent and the tent fell down on
the clown and the car
```

Immaginati quindi di portare a termine questo compito per un milione di linee di testo. Onestamente sarebbe più veloce per te imparare Python e scrivere un programma in Python per contare le parole piuttosto che passare in rassegna tutte le parole.

La notizia ancora migliore è che ho già realizzato un semplice programma per trovare la parola più comune nel file di testo. L'ho scritto, l'ho testato e ora lo do a te così che tu possa risparmiare del tempo.

```
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
words = text.split()
counts = dict()

for word in words:
   counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print bigword, bigcount
```

Non devi nemmeno conoscere Python per usare questo programma. Dovrai completare il capitolo 10 di questo libro per capire completamente le meravigliose tecniche Python che sono state usate per creare questo programma. Tu sei l'utente finale, tu semplicemente usi il programma e ti stupisci della sua bravura e di come ti ha risparmiato così tanto lavoro manuale. Tu semplicemente digiti il codice all'interno di un file chiamato **words.py** e lo esegui, o più semplicmente scarichi il file già bell'e pronto da http://www.pythonlearn.com/code/ e lo esegui.

Questo è un buon esempio di come Python e il suo linguaggio agiscano da intermediari tra te (utente finale) e me (programmatore). Python è un modo per scambiarci utili sequenze di istruzioni (i.e. programmi) scritte in un linguaggio comune che possono essere utilizzate da chiunque installi Python sul suo computer. In questo modo nessuno di noi sta parlando a Python ma comunichiamo tra di noi tramite Python.

1.9 Gli elementi costituenti di un programma

Nei prossimi capitoli, approfondiremo il vocabolario, la sintassi, la struttura dei paragrafi e la struttura della storia di Python. Conosceremo le grandi potenzialità di Python e il modo di integrare queste potenzialità per creare programmi utili.

Ci sono alcuni schemi concettuali di basso livello che usiamo per costruire programmi. Questi costrutti non sono tipici solamente dei programmi Python ma sono parte di ogni linguaggio di programmazione, dal linguaggio macchina fino ai linguaggi di alto livello.

input :

> Recupera dati dal "mondo esterno". Questo potrebbe significare leggere dati da un file, o anche da qualche tipo di sensore come un microfono o un GPS. Nei nostri primi programmi, il nostro input verrà dall'utente che digita sulla tastiera.

output :

> Mostra i risultati di un'elaborazione su uno schermo o li archivia in un file o forse li invia ad una periferica come un altoparlante per riprodurre musica o leggere un testo.

esecuzione sequenziale :

> Esegue istruzioni una dopo l'altra nell'ordine in cui sono riportate nello script.

esecuzione condizionale :

> Controlla certe condizioni ed esegue o salta una sequenza di istruzioni.

esecuzione ripetuta :

> Esegue alcuni set di istruzioni ripetutamente, solitamente con alcune variazioni

riutilizzo :

> Scrive un set di istruzioni una volta sola e gli assegna un nome e poi lo riusa al bisogno all'interno di un programma.

Sembra troppo bello per essere vero e infatti spesso le cose non sono così semplici. Sarebbe come dire che camminare è semplicemente "mettere un piede dopo l'altro". L'arte di scrivere un programma risiede nel comporre e tessere assieme questi elementi di base più e più volte fino a produrre qualcosa di utile per gli utenti. Il programma di conta delle parole sopra riportato usa tutti questi schemi concettuali, tranne uno.

1.10 Cosa potrebbe andare storto?

Come abbiamo visto nelle nostre conversazioni iniziali con Python, dobbiamo essere molto precisi quando scriviamo codice Python. La più piccola deviazione o il più piccolo errore potrebbero causare l'interruzione dell'elaborazione.

I programmatori alle prime armi spesso vedono nel fatto che Python non lasci spazio ad errori come la prova che Python è malvagio, pieno di odio e crudele. Mentre Python sembra andare d'accordo con tutti gli altri, Python conosce questi novellini personalmente e prova del risentimento verso di loro. A causa di questo risentimento, Python prende programmi scritti alla perfezione e li rifiuta come "inadatti" solo per tormentarli.

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>> primt 'Hello world'
  File "<stdin>", line 1
    primt 'Hello world'
                      ^
SyntaxError: invalid syntax
>>> I hate you Python!
  File "<stdin>", line 1
    I hate you Python!
         ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
  File "<stdin>", line 1
    if you come out of there, I would teach you a lesson
              ^
SyntaxError: invalid syntax
>>>
```

C'è poco da guadagnare nel litigare con Python. E' uno strumento e non ha emozioni ed è felice e pronto a servirti ogniqualvolta tu ne abbia bisogno. I suoi messaggi di errore sembrano rudi, ma sono solo il modo di Python di chiedere aiuto. Python ha analizzato ciò che hai scritto e semplicemente non può capire cosa hai inserito.

Python assomiglia molto ad un cane, ti ama incondizionatamente, capisce solo poche parole, ti guarda con uno sguardo dolce (>>>) e aspetta che tu gli dica qualcosa che lui possa capire. Quando Python dice *SyntaxError: invalid syntax* stà semplicemente muovendo la coda e dicendo "Ho visto che mi hai detto qualcosa ma io non lo capisco. Per favore continua a provarci (>>>)."

Man mano che i tuoi programmi diventano più sofisticati, incontrerai tre diversi tipi di errore:

Errori di sintassi (syntax error):

> Questi sono i primi errori che farai e saranno i più facili da correggere. Un errore di sintassi significa che hai violato le regole "grammaticali" di Python. Python fa il suo meglio per indicarti la linea e il punto in cui si è trovato in confusione. L'unico aspetto ingannevole degli errori di sintassi è che spesso l'errore stà a monte del punto indicato da Python. Per questo motivo il punto indicato da Python potrebbe essere solo il punto di partenza della tua indagine

Errori logici :

> Un errore logico avviene quando il tuo programma ha una sintassi corretta ma c'è un errore nell'ordine degli enunciati o forse nell'ordine in cui gli enunciati si legano tra loro. Un buon esempio di errore logico potrebbe essere "prendi un sorso dalla tua bottiglia d'acqua, mettila nello zaino, vai in biblioteca e poi metti il tappo sulla bottiglia".

Errori semantici :

> Un errore semantico accade quando la tua descrizione delle azioni da intraprendere è sintatticamente corretta e nel giusto ordine, ma c'è semplicemente un errore nel programma. Il programma è formalmente corretto ma non fa quello che tu vuoi che faccia. Un semplice esempio potrebbe essere questo: stai dando indicazioni ad una persona per raggiungere un ristorante e dici: "... quando raggiungi l'incrocio con il distributore di benzina, gira a sinistra e vai avanti per un miglio, il ristorante è l'edificio rosso alla tua sinistra". Il tuo

amico è molto in ritardo e ti chiama dicendo che è presso una fattoria e stà camminando attorno ad un fienile e del ristorante non c'è nemmeno l'ombra. Gli chiedi quindi "Hai girato a destra o a sinistra al distributore di benzina?" e lui ti risponde "Ho seguito le tue istruzioni alla lettera, le ho scritte, dovevo girare a sinistra e proseguire per un miglio al distributore di benzina". A questo punto tu dici "Mi dispiace, perché anche se le mie istruzioni erano sintatticamente corrette, purtroppo esse contenevano un piccolo ma poco evidente errore semantico".

In tutti i tre tipi di errore, Python sta semplicemente tentando di fare il suo meglio per fare esattamente cosa tu gli hai chiesto di fare.

1.11 Il viaggio dell'apprendimento

Nel leggere questo libro non preoccuparti se alcuni concetti non sembrano quadrare al primo impatto. Quando hai iniziato a parlare, non era un problema per i primi anni emettere solamente dei gorgoglii. Ed è stato normale impiegare sei mesi per passare da un vocabolario semplice a delle frasi semplici, impiegare 5-6 anni per passare da frasi a paragrafi e spendere altri due anni per scrivere piccole storie complete da solo.

Noi vogliamo che tu apprenda Python molto più rapidamente, quindi te lo presenteremo tutto assieme nei prossimi capitoli. E' però il processo di apprendere una nuova lingua che richiede tempo per essere assimilato e compreso prima che tutto sembri naturale. Questo porta ad alcuni fraintendimenti nel vedere e rivedere gli argomenti nel tentativo di farti apprezzare l'insieme mentre definiamo i minuscoli frammenti che compongono il quadro generale. Anche se questo libro è stato scritto con un approccio lineare e tu stai seguendo un corso di formazione, non esitare a procedere in modo totalmente non-lineare quando ti avvicini alla materia. Guarda avanti, indietro, leggi senza scervellarti troppo. Sbirciando i contenuti più avanti in questo testo, pur non comprendendolo appieno, puoi capire meglio il perché di alcune scelte di programmazione. Rivedendo il materiale già acquisito e rifacendo i vecchi esercizi ti renderai conto che hai imparato molto, anche se ciò che stai guardando adesso sembra impenetrabile.

Normalmente, quando si impara per la prima volta un linguaggio di programmazione, ci sono un paio di momenti "Ah-ah!" in cui ti allontani dalla tua opera di martello e scalpello e vedi che stai realmente realizzando una bella scultura.

Se qualcosa sembra particolarmente difficile, non c'è solitamente ragione per passare la notte in bianco a fissarla. Fai una pausa, riposati, mangia qualcosa, spiega a qualcuno (o al tuo cane) perché stai avendo un problema e poi torna a mente fresca. Ti assicuro che

una volta che avrai appreso i concetti della programmazione esposti in questo libro, ti guarderai indietro e realizzerai che era tutto davvero semplice ed elegante e che ci è solamente voluto un po' di tempo per assimilarlo.

1.12 Glossario

bug (bug): Un errore nel programma

Central Processing Unit (CPU): (unità di elaborazione centrale) Il cuore di ogni computer. E' ciò che esegue il codice che scriviamo; si abbrevia in CPU e a volte la si indica come "il processore".

compilare (compile): Tradurre un programma, scritto in un linguaggio ad alto livello, in linguaggio a basso livello, tutto in una volta, in preparazione ad una successiva esecuzione.

linguaggio di alto livello (high-level language): Un linguaggio di programmazione come Python che è progettato per essere facilmente letto e scritto dagli umani.

modalità interattiva (interactive mode): Un modo di usare l'interprete Python introducendo comandi ed espressioni al prompt.

interpretare (interpret): Eseguire un programma in un linguaggio ad alto livello traducendolo una riga alla volta.

linguaggio a basso livello (low-level language): Un linguaggio di programmazione che è disegnato per essere facilmente eseguibile da un computer: chiamato anche "codice macchina" o "linguaggio assembly".

codice macchina (machine code): Il più basso livello per il software, che è anche il linguaggio eseguito dall'unità di elaborazione centrale (CPU).

memoria principale (main memory): Archivia programmi e dati, ma perde le sue informazioni quando l'alimentazione è tolta.

valutare (parse): Esaminare un programma e analizzare la struttura sintattica.

portabilità (portability): La proprietà di un programma che può essere eseguito su più di un tipo di computer.

istruzione di stampa(print statement): Un'istruzione che fa sì che Python mostri a video un valore.

problem solving : Il processo di formulare un problema, trovare una soluzione ed esprimerla.

programma (program): Un insieme di istruzioni che identifica un'elaborazione.

prompt : Quando un programma mostra un messaggio e attende che l'utente inserisca un dato per l'elaborazione.

memoria secondaria (secondary memory): Archivia programmi e dati e li conserva anche quand l'alimentazione è assente. Generalmente più lenta della memoria principale. Esempi della memoria secondaria sono dischi e memorie USB flash.

semantica (semantics): Il significato di un programma.

errore semantico (semantic error): Un errore in un programma che gli fa fare qualcosa di diverso da ciò che il programmatore vuole.

codice sorgente (source code): Un programma scritto in linguaggio ad alto livello.

1.13 Esercizi

Esercizio 1 Qual è la funzione della memoria secondaria in un computer?

a) Eseguire tutta i calcoli e la parte logica di un programma

b) Scaricare pagine web da internet

c) Archiviare informazioni a lungo termine, anche in assenza di alimentazione

d) Ricevere input dall'utente

Esercizio 2 Cos'è un programma?

Esercizio 3 Qual è la differenza tra compilatore e interprete?

Esercizio 4 Quale delle seguenti contiene codice macchina?

a) L'interprete Python

b) La tastiera

c) Il codice sorgente Python

d) Un documento di elaborazione testi

Esercizio 5 Cosa c'è di sbagliato nel codice seguente:

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>>
```

Esercizio 6 Dove è archiviato il contenuto della variabile "X" quando finisce la seguente istruzione Python?

```
x = 123
```

a) CPU

b) Memoria principale

c) Memoria secondaria

d) Periferica di input

e) Periferica di output

Esercizio 7 Cosa stampa questo codice Python?

```
x = 43
x = x + 1
print x
```

a) 43
b) 44
c) x + 1
d) Un errore perché x = x+1 non è possibile matematicamente

Esercizio 8 Spiega ciascuno dei seguenti elementi usando come esempio una caratteristica umana: (1) CPU, (2) Memoria principale, (3) Memoria secondaria, (4) Dispositivo di input, (5) Dispositivo di output. Per esempio:Qual è l'equivalente umano della CPU?

Esercizio 9 Come correggi un errore di sintassi?

Capitolo 2
Variabili , espressioni e istruzioni

2.1 Valori e tipi

Un **valore** è uno degli elementi fondamentali con cui lavorano i programmi, come una lettera o un numero . I valori che abbiamo visto finora sono *1 , 2 ,* e *'Ciao , mondo !'* .

Questi valori appartengono a **tipi** diversi: *2* è un intero, e *'Ciao, Mondo!'* è una **stringa**, viene chiamata così perché contiene, appunto, una "stringa" di lettere. Le stringhe sono identificabili dal fatto che sono racchiuse tra virgolette. L' istruzione print funziona anche per gli interi. Usiamo il comando python per avviare l'interprete.

```
python
>>> Print 4
4
```

Se non sei sicuro a quale tipo appartenga un valore, lo poi chiedere all'interprete.

```
>>> Type ('Ciao, Mondo!')
<type 'str'>
>>> Type (17)
<type 'int'>
```

Chiaramente, le stringhe appartengono al tipo str e i numeri interi appartengono al tipo int. Meno ovvio, i numeri con un punto decimale appartengono a un tipo chiamato float. Questi numeri sono rappresentati in un formato chiamato **virgola mobile (float)** .

```
>>> Tipo ( 3.2)
<type 'float'>
```

Che dire di valori come '17' e '3.2'? Sembrano numeri, ma sono inclusi tra virgolette come le stringhe.

```
>>> type ('17')
<type 'str'>
>>> type ('3.2')
<type 'str'>
```

Sono stringhe a tutti gli effetti.

Quando si digita un grande numero intero, si potrebbe essere tentati di utilizzare le virgole tra gruppi di tre cifre, come ad esempio in 1,000,000. Questo non è un numero intero valido in Python , ma è pur sempre un valore:

```
>>> Print 1.000.000
1 0 0
```

Beh, questo non è per nulla quello che ci aspettavamo! Python interpreta 1,000,000 come una sequenza di numeri interi separata da virgole, e viene stampata con spazi tra di essi.

Questo è il primo esempio in cui abbiamo visto un errore semantico : il codice viene eseguito senza produrre un messaggio di errore, ma non fa la "cosa giusta".

2.2 Variabili

Una delle caratteristiche più importanti di un linguaggio di programmazione è la capacità di manipolare **variabili**. Una variabile è un nome che si riferisce a un valore.

Un' istruzione di **assegnazione** crea nuove variabili e assegna loro dei valori:

```
>>> Messaggio = 'E ora qualcosa di completamente diverso'
>>> n = 17
>>> Pi = 3,1415926535897931
```

Questo esempio esegue tre assegnazioni. Il primo assegna una stringa a una nuova variabile di nome messaggio; il secondo assegna il numero intero 17 a n; la terza assegna il valore (approssimativo) di π a Pi.

Per visualizzare il valore di una variabile, è possibile utilizzare l'istruzione print:

```
>>> Print n
17
>>> Print pi
3,14159265359
```

Il tipo di una variabile è il tipo del valore alla quale si riferisce.

```
>>> Type (messaggio) <type'str'>
>>> Type (n) <type'int'>
>>> Type (pi) <type'float'>
```

2.3 I nomi delle variabili e keywords (parole riservate o chiave)

I programmatori generalmente scelgono nomi significativi per le **variabili**, che documentano per quale scopo viene usata la variabile.

I nomi delle variabili possono essere arbitrariamente lunghi. Possono contenere sia lettere che numeri , ma devono iniziare con una lettera. E' corretto usare le lettere maiuscole, ma è una buona idea per cominciare nomi di variabili con una lettera minuscola (vedrete perché più avanti).

Il carattere di sottolineatura () può apparire in un nome. E' spesso usato nei nomi con più parole, ad esempio *my_name* o *airspeed_of_unladen_swallow* .

Se si dà una variabile un nome non corretto, si ottiene un errore di sintassi:

```
>>> 76strumenti ='Grande parata'
SyntaxError  : invalid syntax
>>> more@ = 1000000
SyntaxError : invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError : invalid syntax
```

76strumenti è sbagliato perché non inizia con una lettera, *more@* è sbagliato perché contiene un carattere non valido, @. Ma cosa c'è di sbagliato con *class*?

In realtà *class* è una delle **parole riservate** di Python.

L'interprete utilizza le keywords per riconoscere la struttura del programma, perciò non possono essere utilizzate come nomi di variabili.

Python riserva **31 keywords**[3] per il suo utilizzo :

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

---

[3] In Python 3.0, *exec* non è più una keyword, ma *nonlocal* sì.

Sarebbe utile tenere questa lista a portata di mano. Se l'interprete si lamenta di uno dei vostri nomi di variabile non sapere spiegarvi perché, probabilmente è presente in questa lista.

2.4 Istruzioni

Un'**istruzione** è una unità di codice che viene eseguita dall'interprete Python. Abbiamo visto due tipi di istruzioni: stampa e assegnazione.

Quando si digita un'istruzione in modalità interattiva, l'interprete esegue e visualizza il risultato, se ce n'è uno.

Uno script di solito contiene una sequenza di istruzioni. Se vi è più di una istruzione, i risultati vengono visualizzati uno alla volta, relativamente alle istruzioni eseguite.

Per esempio, lo script

```
print 1
x = 2
print x
```

produce l'output

```
1
2
```

L'istruzione di assegnazione non produce alcun output.

2.5 Operatori e operandi

Gli operatori sono simboli speciali che rappresentano i calcoli come addizione e moltiplicazioni. I valori ai quali l'operatore viene applicato sono chiamati operandi.

Gli operatori +, -, , / e * eseguono addizioni, sottrazioni, moltiplicazioni, divisioni e elevamento a potenza, come negli esempi seguenti:

```
20+32, hour-1, hour*60+minute, minute/60, 5**2, (5+9)*(15-7)
```

L'operatore di divisione potrebbe non fare quello che vi aspettate:

```
>>> Minuti = 59
>>> minute/60
0
```

Il valore dei minuti è 59, e in aritmetica convenzionale, 59 diviso per 60 è 0,98333, non 0. La ragione della discrepanza è che Python sta eseguendo una **divisione tra interi**[4].

Quando entrambi gli operandi sono interi, il risultato è un numero intero; la divisione tra interi taglia la parte frazionaria, in questo esempio arrotonda a zero.

Se uno degli operandi è un numero in virgola mobile, Python esegue la divisione in virgola mobile, e il risultato è un *float:*

```
>>> Minute/60.0
0,98333333333333328
```

2.6 Espressioni

Un'**espressione** è una combinazione di valori, variabili e operatori. Un valore da solo è considerato un'espressione, e così anche per una variabile, quindi le seguenti sono tutte espressioni valide (assumendo che alla variabile x sia stato assegnato un valore):

```
17
x
x+17
```

Se si digita un'espressione in modalità interattiva, l'interprete la valuta e visualizza il risultato:

```
>>> 1+1
2
```

Ma in uno script, un'espressione da sola non fa nulla! Questo è una fonte di confusione comune tra i principianti.

Esercizio 2.1 Digita le seguenti istruzioni nell l'interprete Python per vedere quello che fanno:

```
5
x = 5
x + 1
```

[4] In Python 3.0, il risultato di questa divisione è a virgola mobile (float). In Python 3.0, il nuovo operatore // esegue la divisione tra interi.

2.7 Ordine delle operazioni

Quando più di un operatore compare in un'espressione , l'ordine di valutazione dipende dalle **regole di priorità**. Per gli operatori matematici , Python segue la convenzione matematica. L'acronimo **PEMDAS** è un modo utile per ricordare le regole :

• Le **P**arentesi hanno la priorità massima e possono essere utilizzate per forzare l'ordine di esecuzione che desideriamo. Poiché le espressioni tra parentesi vengono valutate per prime , *2*(3-1)* è 4 e *(1+1) ** (5-2)* è 8. Potete anche usare le parentesi per rendere l'espressione più facile da leggere , come in *(minuti * 100) / 60* , anche se non cambia il risultato.

• **E**levamento a potenza ha la priorità successiva così *2 ** 1+1* è 3 , non 4 , e *3 * 1 ** 3* è 3, non 27.

• **M**oltiplicazioni e **D**ivisioni hanno la stessa priorità, ed è più alta di **A**ddizioni e **S**ottrazioni, che hanno anche la stessa priorità. Quindi *2 * 3-1* è 5, non 4 e *6+4 / 2* è 8, non 5.

• Gli operatori con la stessa priorità sono valutati da sinistra a destra . Così nell'espressione *5-3-1* è 1, non 3 perché *5-3* viene eseguito prima e poi viene sottratto *1* da *2*.

Nel dubbio utilizzate sempre le parentesi nelle espressioni per assicurarvi esecuzione venga eseguita nell'ordine voluto.

2.8 Operatore modulo

L'operatore **modulo** opera sugli interi e restituisce il resto quando il primo operando viene diviso per il secondo. In Python, l'operatore modulo è un segno di percentuale (%). La sintassi è la stessa che per altri operatori:

```
>>> quoziente = 7/3
>>> print quoziente
2
>>> resto = 7 % 3
>>> print resto
1
```

Quindi 7 diviso 3 è 2 con resto di 1.

L'operatore modulo risulta essere sorprendentemente utile. Ad esempio , è possibile verificare se un numero è divisibile per un altro : se x % y è zero , allora x è divisibile per y.

Inoltre, è possibile estrarre la cifra o le cifre più a destra di un numero. Ad esempio, x % 10 restituisce la cifra più a destra di x (in base 10). Allo stesso modo x % 100 restituisce le ultime due cifre.

2.9 Operazioni sulle stringhe

L'operatore + funziona anche con le stringhe, ma non in senso matematico. Esegue invece la **concatenazione**, che significa unire le stringhe collegandole consecutivamente.

Per esempio :

```
>>> Primo = 10
>>> Secondo = 15
>>> print primo + secondo
25
>>> Primo = '100 '
>>> Secondo = '150 '
>>> print primo + secondo
100150
```

2.10 Richiedere un input all'utente

A volte vorremmo richiedere all'utente di assegnare un valore ad una variabile tramite tastiera. Python fornisce una funzione built-in chiamata *raw_input* che consente l'immissione di valori tramite tastiera. Quando questa funzione viene eseguita, il programma si ferma e attende che l'utente digiti qualcosa. Quando l'utente preme **Return** o **Invio**, il programma riprende e *raw_input* restituisce ciò che l'utente ha digitato, cioè una stringa.

```
>>> input = raw_input()
Alcune cose stupide
>>> print input
Alcune cose stupide
```

Prima di ottenere un input da parte dell'utente, è una buona idea visualizzare un prompt che dica all'utente cosa digitare.

È possibile passare a raw_input un testo da visualizzare all'utente:

```
>>> name = raw_input ( ' Qual è il tuo nome ? \ N')
Qual è il tuo nome ?
Chuck
>>> print name
Chuck
```

La sequenza *\n* alla fine del prompt rappresenta un **ritorno a capo**: è un carattere speciale che provoca un'interruzione di riga. Ecco perché l'input dell'utente appare sotto il prompt.

Se vi aspettate che utente digiti un numero intero, si può provare a convertire il valore restituito in *intero* utilizzando la funzione *int()* :

```
>>> prompt = 'Quale è la velocità di una rondine? \N'
>>> velocita = raw_input(prompt)
Quale è la velocità di una rondine?
17
>>> print int(velocita )
17
>>> print int(velocita ) + 5
22
```

Ma se l'utente digita qualcosa di diverso da una stringa di cifre, si ottiene un errore :

```
>>> Velocità = raw_input (prompt )
Quale è la velocità di una rondine?
Cosa vuoi dire, una rondine africana o europea?
>>> print int(velocità )
ValueError : invalid literal for int()
```

Vedremo come gestire questo tipo di errore più avanti.

2.11 Commenti

Mano a mano che i programmi diventano sempre più lunghi e complessi risultano più

difficile da leggere. I linguaggi formali sono densi ed è spesso difficile guardare un pezzo di codice e capire cosa sta facendo, o perché. Per questo motivo, è una buona idea aggiungere note ai tuoi programmi per spiegare, in linguaggio naturale, quello che sta facendo il programma. Queste note sono chiamate **commenti**, e iniziano con il simbolo #

```
#Calcolare la percentuale di ore trascorse
percentuale = ( minuti * 100 ) / 60
```

In questo caso, il commento su una riga appare da solo. È anche possibile inserire commenti alla fine di una riga:

```
percentuale = ( minuti * 100 ) / 60 # percentuale di un'ora
```

Tutto il testo da # alla fine della riga viene ignorato, non ha alcun effetto sul programma.

I commenti sono più utili quando si documentano caratteristiche non evidenti del codice. E' ragionevole supporre che il lettore possa capire cosa fa il codice ma è molto più utile a spiegare il perché.

Il commento a questo codice è inutile è superfluo:

```
v = 5 # assegna 5 a v
```

Questo commento, invece, contiene informazioni utili che non sono palesi nel codice:

```
v = 5 # velocità in metri / secondo
```

Attribuire buone nomi a delle variabili può ridurre la necessità di commenti, ma nomi lunghi possono rendere le espressioni complesse difficili da leggere: occorre quindi trovare un buon compromesso.

2.12 La scelta di nomi mnemonici per le variabili

Se si seguono alcune semplici regole per nominare le variabili e si evitano le parole riservate, si hanno moltissime scelte per dare un nome alle variabili . All'inizio questa scelta può essere motivo di confusione sia quando si legge un programma che quando si scrivono i propri programmi. Ad esempio, i seguenti tre programmi sono identici in termini di ciò che compiono, ma molto diverso quando li si legge per capire cosa fanno.

```
a = 35.0
b = 12.50
c = a * b
print c

ore = 35.0
tasso = 12.50
paga = tasso ore * tasso
print paga

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3p9afd
print x1q3p9afd
```

L'interprete Python vede tutti e tre questi programmi esattamente allo stesso modo, ma gli esseri umani li vedono e capiscono in modo diverso. Un essere umano capirà più rapidamente l'**intento** del secondo programma perché il programmatore ha scelto nomi delle variabili che riflettono l'intenzione del programmatore circa i dati che verranno memorizzati in ciascuna variabile.

Possiamo chiamare nomi di variabili scelti saggiamente "nomi mnemonici di variabili".

La parola *mnemonico*[5] significa "in aiuto alla memoria". In primo luogo abbiamo scelto nomi mnemonici per le variabili allo scopo di aiutarci a ricordare il motivo per cui abbiamo creato la variabile stessa.

Questo è ottimo, ed è buona pratica utilizzare nomi di variabile mnemonici, tali nomi possono dare un grande aiuto al programmatore principiante che inizia ad analizzare e capire il codice. Questo perché i programmatori al loro inizio non hanno ancora imparato a memoria le parole riservate (sono solo 31) ma, talvolta, le variabili con nomi molto descrittivi possono apparire come parte del linguaggio e non nomi ben scelti.

Date un rapido sguardo al codice Python seguente che cicla su alcuni dati. Ci occuperemo presto di cicli, ma per ora cerca solo di decifrare ciò che significa:

[5] vedi http://en.wikipedia.org/wiki/Mnemonic per ulterione approfondimento

```
for parola in parole:
    print parola
```

Che succede? Quali dei termini *for, parola, in, print* sono parole riservate e quali sono i nomi delle variabili? Come fa Python a capire a livello fondamentale il concetto di “parole”? I programmatori principianti hanno difficoltà a separare quali parti del codice devono essere sempre le stesse, come in questo esempio, e quali altre parti del codice sono semplicemente scelte fatte dal programmatore.

Il codice seguente è equivalente al codice di cui sopra:

```
for fetta in pizza:
    print fetta
```

E' più facile per il programmatore guardare questo codice e individuare quali parti sono parole riservate definite da Python e quali parti sono semplicemente nomi di variabili. È piuttosto evidente che Python non ha una comprensione fondamentale di pizza e fetta o del fatto che una pizza consiste di un insieme di una o più fette.

Ma se il nostro programma riguarda veramente la lettura di dati e la ricerca di parole nei dati, *pizza* e *fetta* sono nomi di variabili poco mnemonici. Scegliendoli come nomi di variabile si crea una possible confusione sul significato del programma.

Non ci vorrà molto tempo per ricordare che le parole riservate più comuni e si inizierà quindi ad individuarle con certezza:

```
for parola in parole:
    print parola
```

Le parti di codice definite da Python (for , in , print e :) sono in grassetto e le variabili scelte dal programmatore (parola e parole) non lo sono. Molti editor di testo riconosceranno la sintassi di Python e coloreranno le parole in modo diverso per dare indizi e per mantenere le variabili e parole riservate separati. Dopo un pò si inizia a leggere Python velocemente ed a determinare quali sono le variabili e quali le parole riservate.

2.13 Debug

A questo punto l'errore di sintassi più probabile è quello di definire un nome scorretto per una variabile, come *class* o *yeld*, che sono keywords , o *posto}lavoro* e *US$*, che contengono caratteri non validi.

Se si mette uno spazio in un nome di variabile, Python pensa che siano due operandi senza un operatore

```
>>> Bad name = 5
SyntaxError : invalid syntax
```

Per gli errori di sintassi, i messaggi di errore non aiutano molto. I messaggi più comuni sono *SyntaxError : invalid syntax* o *SyntaxError : invalid token*, nessuno dei quali è molto informativo.

L'errore di runtime più frequente è "use before def;" cioè si sta cercando di utilizzare una variabile prima di averle assegnato un valore. Questo può accadere se si scrive il nome sbagliato della variabile:

```
>>> principale = 327,68
>>> interesse = principio * tasso
NameError: name 'principio' is not defined
```

I nomi variabili sono *case sensitive*, quindi LaTeX non è lo stesso di latex.

Inoltre, la causa più probabile di un errore di semantica è l'ordine delle operazioni. Ad esempio, per valutare $1/2\pi$, si potrebbe essere tentati di scrivere:

```
>>> 1.0 / 2.0 * pi
```

Ma la divisione avviene prima, così si otterrebbe $\pi / 2$, che non è la stessa cosa! Non c'è modo per Python per sapere cosa si intende ottenere quindi, in questo caso, non avremo un messaggio di errore ma solo la risposta è sbagliata.

2.14 Glossario

Assegnazione : una istruzione che assegna un valore ad una variabile.

Concatenare: unire due operandi consecutivamente.

Commento: informazioni poste nel codice dal programmatore destinate ad altri e che non hanno alcun effetto sull'esecuzione del programma.

Valutare: semplificare un'espressione eseguendo le operazioni allo scopo di ottenere un singolo valore.

Espressione: una combinazione di variabili, operatori e valori che rappresenta un valore unico come risultato.

Float : un tipo di dato che rappresenta numeri con parti frazionarie.

Divisione per numeri interi: l'operazione che divide due numeri e tronca la parte frazionaria.

Integer : Un tipo di dato che rappresenta numeri interi.

Keyword : Una *parola riservata* che viene utilizzata dal compilatore per analizzare un programma; non è possibile utilizzare keywords come *if, def, e while* come nomi di variabili.

Mnemonico : in supporto alla memorizzazione. Noi spesso diamo nomi mnemonici alle variabili per aiutarci a ricordare ciò che è memorizzato nella variabile stessa.

Modulo: un operatore, indicato con un segno di percentuale (%), che funziona su numeri interi e restituisce il resto quando un numero viene diviso per un altro.

Operando: Uno dei valori su cui agisce un operatore.

Operatore: simbolo speciale che rappresenta un'elaborazione semplice come la moltiplicazione, o la concatenazione di stringhe.

Regole di priorità: l'insieme delle norme che disciplinano l'ordine in cui vengono eseguite le espressioni che coinvolgono più operatori e operandi.

Istruzione: una sezione di codice che rappresenta un comando o un'azione. Finora, le istruzioni che abbiamo visto sono le assegnazioni e l'istruzione *print*.

Stringa: un tipo di dato che rappresenta una sequenza di caratteri.

Tipo di dato: una categoria di valori. I tipi che abbiamo visto finora sono interi (type int), numeri in virgola mobile (type float) e le stringhe (type str).

Valore : una delle unità basilari di dati che i programmi gestiscono, come numeri o stringhe.

Variabile : un nome che fa riferimento a un valore.

2.15 Esercizi

Esercizio 2.2 Scrivere un programma che utilizza *raw_input* per richiedere all'utente il proprio nome e restituisce un saluto.

```
Inserisci il tuo nome: Chuck
Ciao Chuck
```

Esercizio 2.3 Scrivere un programma per richiedere all'utente il numero di ore lavorate, la paga oraria e calcolare la retribuzione.

```
Inserire il numero di ore: 35
Inserisci la paga oraria: 2.75
Paga: 96.25
```

Per ora non preoccuparti di avere un risultato con due cifre decimali. Se si desidera, si può sperimentare la funzione *round* per ottenere un risultato a due cifre decimali.

Esercizio 2.4 Eseguiamo le seguenti istruzioni di assegnazione:

```
width = 17
height = 12.0
```

Per ciascuna delle seguenti espressioni, scrivere il valore dell'espressione e il tipo dato (del valore dell'espressione).

```
1. width / 2
2. width/2.0
3. height / 3
4. 1 + 2 * 5
```

Usate Python in modalità interprete per controllare le vostre risposte.

Esercizio 2.5

Scrivete un programma che richiede all'utente una temperatura Celsius, convertire la temperatura Fahrenheit e restituisce la temperatura convertita.

Capitolo 3

Esecuzione Condizionale

3.1 Espressioni booleane

Un'**espressione booleana** è un'espressione che è o vera o falsa. Gli esempi seguenti utilizzano l'operatore == , che mette a confronto due operandi e produce *True* se sono uguali altrimenti *False* :

```
>>> 5 == 5 _True_
>>> 5 == 6 _False_
```

True e *False* sono valori particolari che appartengono al tipo *bool*; non sono stringhe:

```
>>> Type ( True) <type 'bool'>
>>> type ( Falso ) <type 'bool'>
```

L'operatore == è uno degli operatori di confronto; gli altri sono:

```
x != y # x non è uguale a y
x > y # X è maggiore di y
x < y # x è minore di y
x > = y # X è maggiore o uguale a y
x < = y # x è minore o uguale a y
x is y # x è come y
x is not y # x non è come y
```

Anche se queste operazioni ti saranno probabilmente familiari , i simboli Python sono diversi dai simboli matematici. Un errore comune è quello di utilizzare un unico segno di uguale (=) invece uno doppio (==). Ricordate che = è un operatore di assegnazione e == è un operatore di confronto. Non esiste una cosa come =< oppure =>.

3.2 Operatori logici

Ci sono tre operatori logici *AND, OR,* e *NOT*. La semantica (significato) di questi operatori è simile al loro significato in inglese. Per esempio,

```
x> 0 and x <10
```

è TRUE solo se x è maggiore di 0 e minore di 10.

`n% 2 == 0 o n% 3 == 0` è vero se una delle condizioni è vera, cioè, se il numero è divisibile per 2 o 3.

Infine, l'operatore *not* nega una espressione booleana, quindi not (x > y) è TRUE se x > y è FALSE, che è se x è meno di o uguale a y.

In pratica, gli operandi degli operatori logici devono essere espressioni booleane, ma Python non è molto rigoroso. Qualsiasi numero diverso da zero viene interpretato come TRUE.

```
>>> 17 and True
True
```

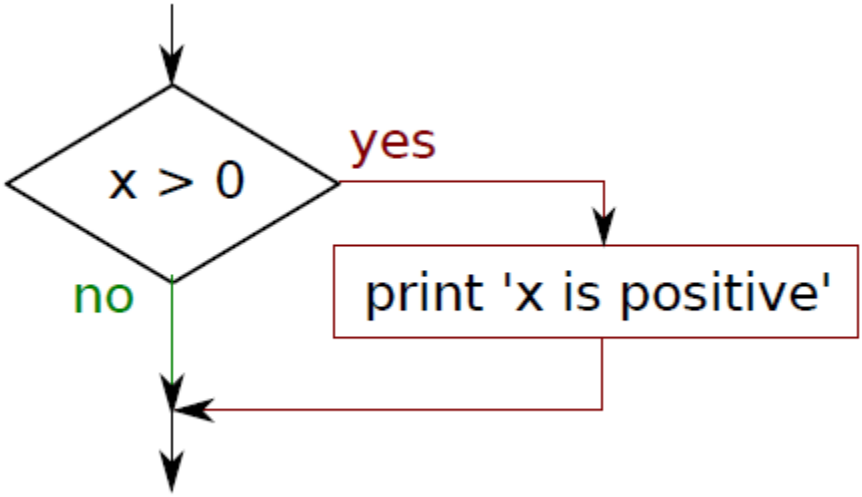
Questa flessibilità può essere utile, ma ci sono alcuni dettagli che potrebbero essere fonte di confusione. Si dovrebbe cercare di non utilizzarla (a meno che non si sappia cosa si sta facendo).

3.3 Esecuzione condizionale

Per scrivere programmi utili , abbiamo quasi sempre la necessità di controllare alcune condizioni e modificare il comportamento del programma di conseguenza. Le **istruzioni condizionali** ci danno questa possibilità. La forma più semplice è l'istruzione *if* :

```
if x > 0 :
    print ' x è positivo '
```

L'espressione booleana dopo l'istruzione *if* è chiamata **condizione**. Terminiamo l'istruzione *if* con un carattere due punti (:) e la linea/linee dopo l'istruzione *if* sono rientrate (indentate).



Se la condizione logica è vero, allora l'istruzione indentata viene eseguita. Se la condizione logica è falsa, l'istruzione indentata viene saltata.

Le istruzioni if hanno la stessa struttura delle definizioni di funzioni o cicli for. La dichiarazione si compone di una riga di intestazione che termina con il carattere due punti (:) seguito da un blocco indentato. Istruzioni come questa sono chiamati istruzioni composte perché si estendono su più di una riga.

Non c'è alcun limite al numero di istruzioni che possono comparire nel corpo, ma deve essercene almeno una. Occasionalmente, è utile avere un blocco indentato senza istruzioni (di solito serve come segnaposto per il codice non è ancora stato scritto). In questo caso, è possibile utilizzare l'istruzione *pass*, che non ha alcun effetto.

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

Se si immette un'istruzione *if* nell'interprete Python, il prompt cambierà da tre >>> a tre punti ... per indicare che siete nel mezzo di un blocco di istruzioni come illustrato di seguito:

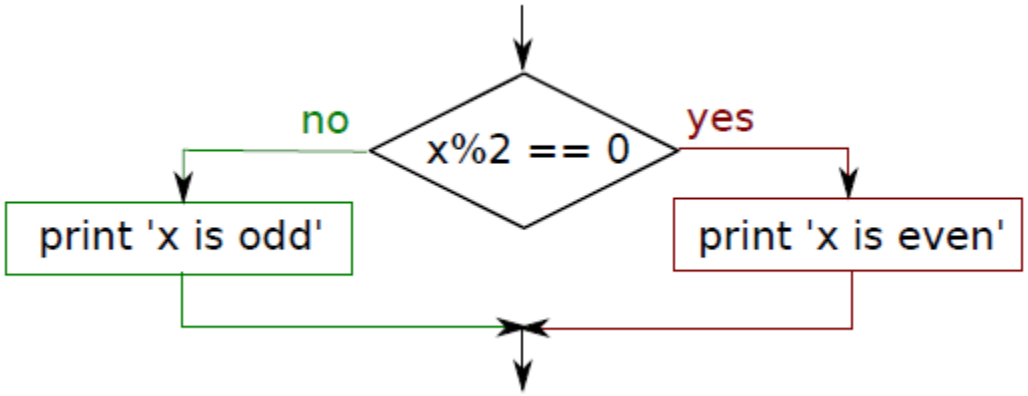
```
>>>x = 3
>>>if x < 10 :
...     print ' Piccolo '
...
Piccolo
>>>
```

3.4 Esecuzione Alternativa

Una seconda forma di **if** è l'esecuzione alternativa, in cui ci sono due possibilità e la condizione determina quale viene eseguita. La sintassi è simile alla seguente:

```
if x%2 == 0:
    print 'x è pari'
else :
    print 'x è dispari'
```

Quando x è diviso per 2 se il resto è 0, allora sappiamo che x è pari, e il programma visualizza un messaggio in tal senso. Se la condizione è falsa, viene eseguito il secondo set di istruzioni.



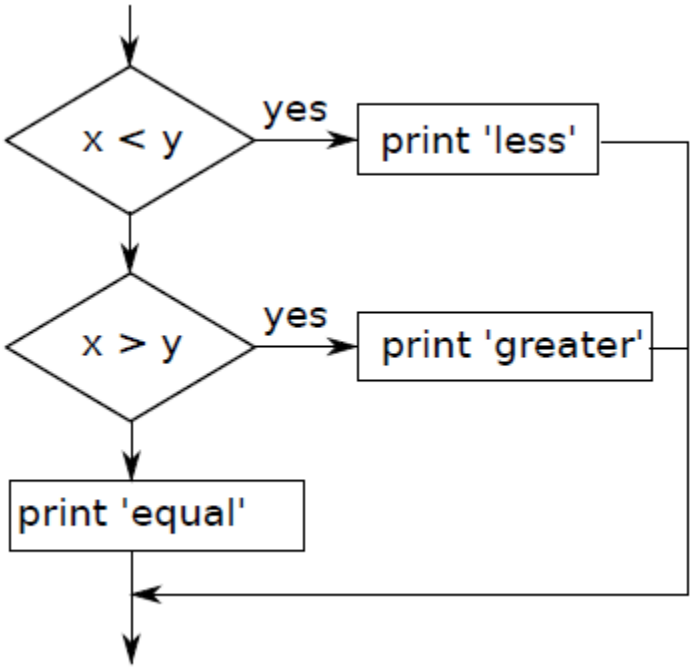
Dato che la condizione deve essere vera o falsa, una delle alternative verrà eseguita esattamente. Le alternative sono chiamate **branches**, perché sono rami nel flusso di esecuzione.

3.5 Esecuzione Condizionale Concatenata

A volte esistono sono più di due possibilità e quindi abbiamo bisogno di più di due ramificazioni. Un modo per descrivere questo tipo di esecuzione è **condizionale concatenata**:

```
if x < y:
    print 'x è minore di y'
elif x > y:
    print 'x è maggiore di y'
else:
    print 'x e y sono uguali'
```

elif è l'abbreviazione di "else if". Anche in questo caso verrà eseguito un solo ramo.



Non c'è un limite al numero di istruzioni elif. Se c'è una clausola else, deve sempre essere alla fine, ma non è necessario che ci sia sempre.

```
if choice == 'a':
    print 'Bad guess'
elif choice == 'b':
    print 'Good guess'
elif choice == 'c':
    print 'Close, but not correct'
```

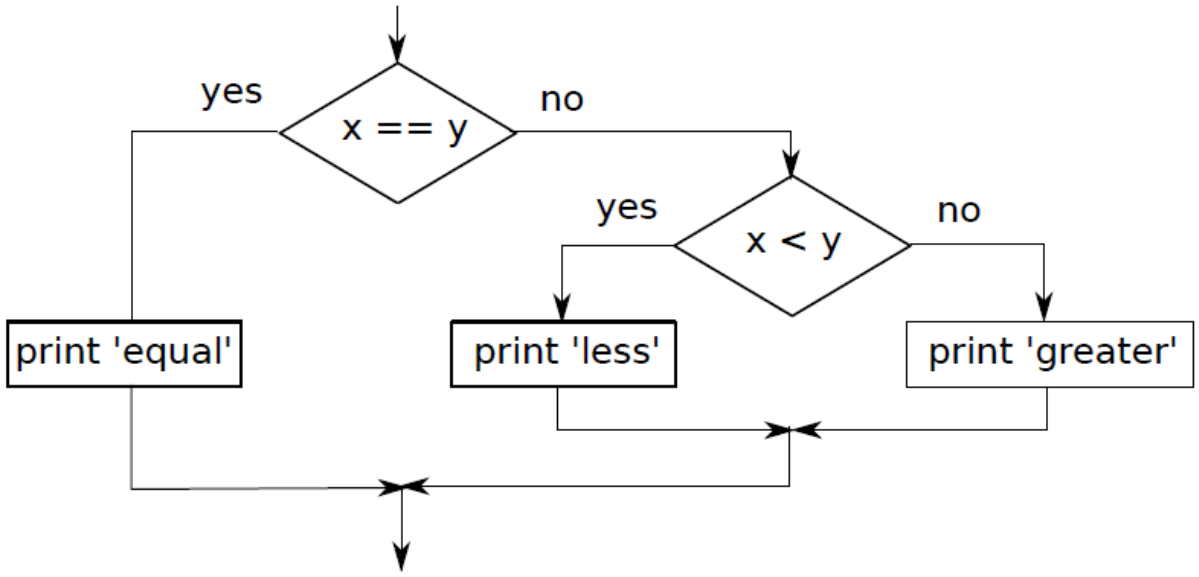
Ogni condizione viene verificata in ordine. Se la prima condizione risulta falsa, viene controllata la successiva e così via. Quando una condizione risulta vera viene eseguito il ramo corrispondente e l'istruzione finisce. Nel caso in cui due o più condizioni dovessero risultare vere, solo la prima di esse, in ordine di controllo, verrà eseguita.

3.6 Esecuzione Condizionale Annidata

Un' esecuzione condizionale può anche essere annidata all'interno di un'altra. Potremmo scrivere un esempio a *tripla* condizione in questo modo:

```
if x == y:
    print 'x and y are equal'
else:
    if x < y:
        print 'x is less than y'
    else:
        print 'x is greater than y'
```

La condizione esterna contiene due rami. Il primo ramo contiene un'istruzione semplice. Il secondo ramo contiene un'altra istruzione *if*, che ha due rami propri. Questi due rami sono entrambi istruzioni semplici, anche se avrebbero potuto essere altre condizioni.



Anche se l'indentazione rende la struttura più leggibile, le condizioni annidate possono diventare difficili da leggere in modo rapido. In generale, è una buona idea evitarlo dove possibile.

Gli operatori logici sono un modo per semplificare le istruzioni condizionali annidate. Ad esempio, possiamo riscrivere il codice riportato di seguito utilizzando una sola condizione:

```
if 0 < x:
    if x < 10:
```

```
        print 'x is a positive single-digit number.'
```

L'istruzione di stampa viene eseguita solo entrambe le condizioni sono vere, quindi possiamo ottenere lo stesso effetto con l'operatore *AND* :

```
if 0 < x and x < 10:
    print 'x is a positive single-digit number.'
```

3.7 Intercettare le eccezioni usando *try* e except

In precedenza abbiamo visto un segmento di codice in cui abbiamo usato le funzioni *raw_input* e int per leggere e analizzare un numero intero immesso dall'utente. Abbiamo anche visto che è poco affidabile:

```
>>> velocità = raw_input (prompt )
Qual è la velocità ... velocità di una rondine?
Cosa vuoi dire , un una rondine africana o europea ?
>>> Int (velocità )
ValueError: invalid literal for int()

>>>
```

Eseguendo queste istruzioni, l'interprete Python ci fa una nuova richiesta, poi pensa "oops" e passa alla prossima istruzione.

Tuttavia, se questo codice fosse in uno script e si verificasse questo errore, lo script si fermerebbe immediatamente con un *traceback* e non eseguirebbe il resto delle istruzioni

Ecco un programma di esempio per convertire una temperatura Fahrenheit in una temperatura Celsius:

```
inp = raw_input('Enter Fahrenheit Temperature:')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print cel
```

Se eseguiamo questo codice e diamo input non valido , si fermerebbe restituendo un messaggio di errore:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
File "fahren.py", line 2, in <module>
fahr = float(inp)
ValueError: invalid literal for float(): fred
```

In Python esiste una struttura di esecuzione condizionale chiamati "*try/except*" proprio per gestire questi tipi errori imprevisti.

L'idea di "try/except" è che sapendo che una sequenza di istruzioni può avere un problema, si desidera aggiungere alcune istruzioni da eseguire se si verifica un errore. Queste istruzioni supplementari (blocco except) vengono ignorate nel caso non si verificheri alcun errore.

Si può pensare al "try/except" in Python come una "polizza di assicurazione" sulla sequenza di istruzioni.

Possiamo quindi riscrivere il nostro convertitore di temperatura come segue:

```
inp = raw_input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print cel
except:
    print 'Please enter a number'
```

Python inizia eseguendo la sequenza di istruzioni nel blocco *try*. Se tutto procede bene, salta il blocco *except* e procede. Se invece si verifica un errore nel blocco *try*, Python esce dal blocco stesso ed esegue la sequenza di istruzioni nel blocco *except*.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

La gestione di un'eccezione con le istruzioni try è chiamato "catturare un'eccezione" (catching an exception).

In questo esempio, la clausola *except* stampa un messaggio di errore ma, in generale, la cattura di un eccezione dà la possibilità di risolvere il problema, provare di nuovo o, quantomeno, terminare correttamente il programma.

3.8 Valutazione corto circuito di espressioni logiche

Quando Python elabora un'espressione logica come `x > = 2 and ( x / y) > 2`, valuta l'espressione da sinistra a destra.

A causa della definizione di *and*, se x è minore di 2, l'espressione `x > = 2` sarà False e quindi l'intera espressione sarà False indipendentemente dal fatto che `(x/y) > 2` restituisca True o False.

Quando Python rileva che non c'è nulla da guadagnare valutando il resto, l'espressione logica ferma alla sua valutazione. Quando la valutazione di un'espressione logica si ferma è perché il valore complessivo è già noto, si definisce come **corto circuito** della valutazione.

Questo può anche sembrare una sottigliezza, ma il comportamento di corto circuito consente una tecnica molto raffinata chiamata **guardian pattern**. Consideriamo l'esecuzione di questa sequenza nell'interprete Python:

```
>>> X = 6
>>> Y = 2
>>> X > = 2 and ( x / y) > 2
True
>>> X = 1
>>> Y = 0
>>> X > = 2 e ( x / y) > 2
False
>>> X = 6
>>> Y = 0
>>> X > = 2 e ( x / y) > 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

Il terzo calcolo non è riuscito perché valutando (x / y) e y era 0 e ha provocato un errore di runtime.

Il secondo esempio però non ha fallito perché la prima parte dell'espressione x > = 2 è risultata falsa così, a causa della regola di corto circuito, (x / y) non è stato eseguito. Siamo quindi in grado di costruire le espressione logiche posizionando strategicamente un **guardian pattern** appena prima della valutazione che potrebbe causare un errore. Ad esempio:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

Nella prima espressione logica x > = 2 è False quindi la valutazione si ferma a and.

Nella seconda espressione logica x > = 2 è vero ma y != 0 è False quindi (x / y) non viene eseguito.

Nella terza espressione logica, y != 0 è dopo il (x / y) quindi il calcolo dell'espresione genera un errore.

Nella seconda espressione, diciamo che y != 0 agisce come una guardiano per assicurare che (x / y) venga eseguito solo se y è diverso da zero.

3.9 Debugging

Quando si verifica un errore la visualizzazione del traceback Python contiene moltissime informazioni e può essere molto estesa, soprattutto quando ci sono molti livelli nel medesimo programma.

Le parti generalmente più utili sono:

• Che tipo di errore si è verificato

• Quando si è verificato

Gli errori di sintassi sono piuttosto facili da trovare, ma ci sono un paio di casi più complicati. Gli errori dovuti a spaziature e tabulazioni sono difficilmente individuabili in quanto invisibili e siamo portati a ignorarli.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
   y = 6
   ^
SyntaxError: invalid syntax
```

In questo esempio, il problema è che la seconda riga rientra di uno spazio, ma il messaggio di errore indica *y*, il che è fuorviante.

In generale i messaggi di errore indicare dove è stato scoperto il problema, ma l'errore effettivo potrebbe essere precedente, a volte una riga prima.

Lo stesso vale per gli errori di runtime. Supponiamo che si sta tentando di calcolare il rapporto segnale/rumore in decibel.

La formula è SNRdb = 10log10 (Psignal / PNOISE)

In Python, si potrebbe scrivere qualcosa del genere:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power / noise_power
decibels = 10 * math.log10(ratio)
print decibels
```

Ma quando lo si esegue, si ottiene un errore:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
OverflowError: math range error
```

Il messaggio di errore indica la linea 5, ma non c'è niente di sbagliato in quella linea. Per trovare il vero errore, potrebbe essere utile stampare il valore di ratio, che si rivelerebbe essere 0.

Il problema è in linea 4, perché esegue una divisione tra due interi. La soluzione è quella di convertire *signal_power* e noise_power in valori con virgola mobile.

Quindi, in generale, i messaggi di errore dicono dove è stato scoperto il problema, ma spesso non dicono dove è stato causato.

3.10 Glossario

corpo (body) : la sequenza di istruzioni all'interno di un'istruzione composta.

espressione booleana : un'espressione il cui valore è True o False.

ramo (branch): una delle sequenze di istruzioni alternative in un'istruzione condizionale.

condizione concatenata (nested conditional): un'istruzione condizionale con una serie di rami alternativi.

operatore di confronto : un operatore che mette a confronto i suoi operandi : == , = , > , < , > = e < =.

istruzione condizionale (conditional statement): una dichiarazione che controlla il flusso di esecuzione in relazione ad una certa condizione.

condizione (condition): l'espressione booleana in un'istruzione condizionale che determina quale ramo viene eseguito.

istruzione composta (compound statement): un'istruzione che consiste di un'intestazione e un corpo. L'intestazione termina con i due punti (:). Il corpo è rientrato rispetto all'intestazione.

guardian pattern: dove costruiamo un'espressione logica usando condizioni particolari per trarre vantaggio del comportamento di corto circuito.

operatore logico: uno degli operatori che combina le espressioni booleane : and, or e not.

condizione annidata (nested conditional): istruzione condizionale che appare in uno dei rami di un'altra istruzione condizionale .

traceback: un elenco di funzioni in esecuzione, che viene restituito quando si verifica un'eccezione (errore).

corto circuito: quando Python è a metà stradanell'esecuzione di una espressione logica e ferma la valutazione in quanto il valore finale per l'espressione è già noto senza la necessità di valutare il resto dell'espressione.

3.11 Esercizi

Esercizio 3.1 Riscrivere il calcolo retributivo in modo da dare al lavoratore 1,5 volte la tariffa oraria per le ore lavorate oltre a 40 ore.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Esercizio 3.2 Riscrivere il programma di calcolo retributivo usando con try e except in modo che il programma gestisca correttamente l'inserimento di dati non numerici restituendo un messaggio ed uscendo dal programma.

Di seguito sono riportate le due esecuzioni del programma :

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input
Enter Hours: forty
Error, please enter numeric input
```

Esercizio 3.3 Scrivere un programma per richiedere un punteggio tra 0,0 e 1,0. Se il punteggio è fuori intervallo ritorna un messaggio. Se il punteggio è tra 0.0 e 1.0, ritorna il grado utilizzando la seguente tabella:

```
Punteggio    Grade
> = 0,9         A
> = 0.8         B
> = 0.7         C
> = 0,6         D
< 0.6           F

Enter score: 0.95
A
Enter score: perfect
Bad score
Enter score: 10.0
Bad score
Enter score: 0.75
C
Enter score: 0.5
F
```

Capitolo 4 - Funzioni

4.1 Invocare una funzione

Nel contesto della programmazione, una **funzione** è una sequenza di espressioni che produce un risultato ed è richiamabile con un nome definito. Quando si definisce una funzione, se ne deve specificare il nome. In seguito, la funzione viene "invocata" con il nome. Abbiamo già visto un esempio di **chiamata**:

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

Il nome della funzione è *type*. L'espressione tra parentesi è chiamata **argomento** della funzione. L'argomento è un valore o una variabile che stiamo passando alla funzione come input alla funzione stessa. Il risultato, per la funzione *type*, è il tipo dell'argomento.

E' prassi affermare che una funzione **riceve** un argomento e **ritorna** un risultato. Il risultato è detto **valore di ritorno**.

4.2 Funzioni predefinite

Python fornisce una serie di importanti funzioni predefinite che possiamo usare senza doverle ridefinire esplicitamente. I creatori di Python hanno elaborato un insieme di funzioni per risolvere i problemi più comuni e lo hanno messo a disposizione di tutti.

Le funzioni max e min ci danno il massimo e minimo valore in una lista, rispettivamente:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

La funzione *max* ci restituisce "il carattere più grande" nella stringa (che risulta essere la lettera "w") e la funzione *min* ci restituisce il carattere più piccolo (che risulta essere uno spazio).

Un'altra funzione predefinita molto comune è la funzione *len* che restituisce il numero di oggetti contenuti nel suo argomento. Se tale argomento è una stringa, essa restituisce il numero di caratteri che compongono la stringa.

```
>>> len('Hello world')
11
>>>
```

Queste funzioni non sono limitate alle stringhe, ma possono operare su un qualsiasi set di valori, come vedremo più avanti.

Dovresti considerare i nomi delle funzioni alla stessa stregua delle parole riservate (es. evitare di usare "max" come nome di variabile).

4.3 Funzioni di conversione di tipo

Python fornisce anche delle funzioni predefinite che convertono valori da un tipo all'altro. La funzioneint prende un valore e lo converte in un intero, se è possibile, altrimenti protesta:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

int può convertire valori in virgola mobile in interi, ma non arrotonda, semplicemente tronca la parte decimale:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

float converte interi e stringhe in numeri in virgola mobile:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Infine *str* converte il suo argomento in stringa:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4 Numeri casuali

Fornendo lo stesso input, la maggior parte dei programmi restituisce lo stesso output ogni volta. Per questa ragione sono definiti **deterministici**. Il determinismo è solitamente una cosa buona, dal momento che ci aspettiamo che calcoli uguali portino a risultati uguali. Tuttavia per certe applicazioni preferiremmo che il computer fosse meno prevedibile. I giochi sono un esempio ovvio, ma ce ne sono altri.

Creare un programma realmente non deterministico risulta essere un compito arduo, ma ci sono modi per farlo almeno sembrare non deterministico. Uno di questi consiste nell'usare algoritmi che generano numeri pseudo casuali. I numeri pseudo casuali non sono realmente casuali perché sono generati da una funzione deterministica, ma a prima vista sono indistinguibili dai numeri casuali.

Il modulo *random* fornisce una funzione che genera numeri pseudo casuali (che chiamerò semplicemente 'casuali' da ora in poi).

La funzione *random* restituisce un numero casuale in virgola mobile compreso tra 0.0 e 1.0 con 0.0 incluso ma non 1.0. In notazione matematica questo si esprime [0.0,1.0). Ogni volta che si invoca la funzione random, viene restituito il numero successivo di una lunga serie di numeri. Per vederne un esempio eseguiamo questo loop:

```
import random

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

Questo programma produce la seguente lista di 10 numeri casuali nell'intervallo [0.0, 1.0)

```
0.301927091705
0.513787075867
0.319470430881
0.285145917252
0.839069045123
0.322027080731
0.550722110248
0.366591677812
0.396981483964
0.838116437404
```

Esercizio 1 Esegui il programma sul tuo computer e osserva quali numeri ottieni. Esegui il programma più volte e confronta i risultati.

La funzione *random* è solo una delle molte funzioni che gestiscono numeri casuali. La funzione *randint* accetta i parametri *low* e *high* e restituisce un intero nell'intervallo tra low e high inclusi gli estremi.

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

Per scegliere un elemento da una sequenza casuale, puoi usare *choice* come segue:

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

Il modulo r*andom* fornisce anche funzioni per generare valori casuali da distribuzioni continue come gaussiana, esponenziale, gamma e un paio di altre ancora.

4.5 Funzioni matematiche

Python ha un **modulo** matematico che fornisce la maggior parte di funzioni matematiche. Prima di poter usare questo modulo lo dobbiamo importare:

```
>>> import math
```

Questa istruzione crea un oggetto modulare chiamato *math*. Se stampi l'oggetto modulare ottieni alcune informazioni:

```
>>> print math
<module 'math' from '/usr/lib/python2.5/lib-dynload/math.so'>
```

L'oggetto modulare contiene le funzioni e le variabili definite nel modulo. Per accedere a una delle funzioni, devi specificare il nome del modulo e il nome della funzione, separati da un punto. Questo formato è detto **dot notation**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

Il primo esempio calcola il logaritmo in base 10 del rapporto segnale/rumore. Il modulo math fornisce anche una funzione detta **log** che calcola il logaritmo naturale.

Il secondo esempio trova il seno di un angolo espresso in radianti. Il nome della variabile suggerisce il fatto che **sin** e le altre funzioni trigonometriche (**cos**, **tan**, etc.) accettano angoli in radianti come argomento. Per convertire da gradi a radianti, dividi per 360 e moltiplica per 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

L'espressione math.pi fornisce la variabile *pi* del modulo *math*. Il valore di questa variabile è un'approssimazione di π precisa fino alla 15esima cifra decimale.

Se conosci la trigonometria, puoi confrontare l'ultimo risultato con la metà della radice di due:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

4.6 Aggiungere nuove funzioni

Fino ad ora abbiamo usato solo funzioni fornite con Python, ma è anche possibile aggiungere nuove funzioni. Una definizione di funzione specifica il nome di una nuova funzione e la sequenza di espressioni che vengono eseguite quando la funzione è invocata. Una volta che definiamo una funzione, possiamo riusare la funzione più volte nel corso del nostro programma.

Ecco un esempio:

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'
```

def è una parola riservata che indica che questa è una definizione di funzione. Il nome della funzione è *print_lyrics*. Le regole per i nomi di funzione sono le stesse in vigore per i nomi di variabile. Non puoi usare una parola chiave come nome di funzione, e si deve evitare di usare lo stesso nome sia per una variabile che per una funzione.

Le parenteri vuote dopo il nome indicano che questa funzione non accetta argomenti. In seguito costruiremo funzioni che accetano argomenti in input.

La prima riga della definizione di funzione è detta **header** (intestazione); il resto è detto **body** (corpo). L'intestazione deve finire con due punti (:) e il corpo deve essere indentato. Per convenzione, l'indentazione è sempre di quattro spazi. Il corpo può contenere un qualsiasi numero di istruzioni.

Le stringhe nelle istruzioni di stampa sono racchiuse da virgolette doppie. Le virgolette semplici e le doppie hanno la stessa funzione: la maggior parte delle persone usa le virgolette semplici tranne nei casi in cui nella stringa appare una virgoletta semplice (come un apostrofo).

Se inserisci una definizione di funzione in modalità interattiva, l'interprete stampa puntini di sospensione (...) per farti capire che la definizione non è completa:

```
>>> def print_lyrics():
...     print "I'm a lumberjack, and I'm okay."
...     print 'I sleep all night and I work all day.'
...
```

Per concludere la funzione devi inserire una riga vuota (cosa non necessaria in uno script).

Definendo una funzione, si crea una variabile con lo stesso nome.

```
>>> print print_lyrics
<function print_lyrics at 0xb7e99e9c>
>>> print type(print_lyrics)
<type 'function'>
```

Il valore di *print_lyrics* è un **oggetto funzione** di tipo 'function'.

La sintassi per invocare la nuova funzione è la stessa delle funzioni predefinite:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Una volta definita una funzione, puoi usarla all'interno di un'altra funzione. Per esempio, per ripetere il ritornello precedente, potremmo scrivere una funzione chiamata *repeat_lyrics*:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

ed invocare *repeat_lyrics*:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Ma questo non è esattamente come fa la canzone...

4.7 Definizioni e usi

Mettendo assieme i pezzi di codice dal paragrafo precedente, il programma nel suo insieme si presenta così:

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

Questo programma contiene due definizioni di funzione: *print_lyrics* and *repeat_lyrics*.

Le definizioni di funzione sono eseguite proprio come le altre istruzioni, ma l'effetto è quello di creare *oggetti funzione*. Le istruzioni all'interno della funzione non vengono eseguite fino a che la funzione è invocata, quindi la definizione della funzione non genera output.

Come potresti aspettarti, devi creare una funzione prima di poterla usare. In altre parole, la definizione della funzione deve essere eseguita prima che la funzione possa venir invocata.

Esercizio 2 Sposta l'ultima riga del programma in cima, così che l'invocazione alla funzione appaia prima della sua definizione. Esegui il programma e vedi qual è il messaggio di errore che ottieni.

Esercizio 3 Rimetti la chiamata alla funzione alla fine del programma e sposta la definizione di *print_lyrics* dopo la definizione di *repeat_lyrics*. Cosa succede quando esegui il programma?

4.8 Flusso di esecuzione

Per assicurarsi del fatto che una funzione sia definita prima di venir usata, si deve conoscere l'ordine secondo cui le istruzioni sono eseguite, che è detto flusso di esecuzione.

L'esecuzione inizia sempre alla prima istruzione del programma. Le istruzioni sono eseguite una alla volta, in ordine dall'alto verso il basso.

Le definizioni di funzione non alterano il flusso dell'esecuzione del programma, ma si tenga presente che le istruzioni all'interno delle funzioni non sono eseguite finché tali funzioni non vengono invocate.

Una chiamata di funzione è come una deviazione nel flusso dell'esecuzione. Anziché proseguire all'istruzione seguente, il flusso salta al corpo della funzione, esegue tutte le istruzioni lì e poi ritorna al punto in cui aveva lasciato.

Questo sembra abbastanza semplice, finché non si parla di funzioni che richiamano altre funzioni. Nel bel mezzo di una funzione, il programma potrebbe dover eseguire istruzioni in un'altra funzione. Ma, mentre esegue quella nuova funzione, il programma potrebbe doverne eseguire una terza!

Fortunatamente, Python è bravo a tenere traccia di dove si trova, così ogni volta che una

funzione termina, il programma riprende da dove aveva deviato. Quando si arriva alla fine del programma, l'esecuzione termina.

Qual è la morale di questa storia? Quando leggi un programma, non ha sempre senso leggerlo dall'inizio alla fine ma a volte ha più significato seguire il flusso dell'esecuzione.

4.9 Parametri e argomenti

Alcune delle funzioni che abbiamo visto richiedono degli argomenti. Per esempio, quando si invoca *math.sin* viene passato un numero come argomento. Alcune funzioni ne richiedono più di uno: *math.pow* ne richiede due, la base e l'esponente.

All'interno della funzione, gli argomenti sono assegnati a variabili chiamate **parametri**. Qui c'è un esempio di una funzione definita dall'utente che accetta un argomento:

```
def print_twice(bruce):
    print bruce
    print bruce
```

Questa funzione assegna l'argomento a un parametro chiamato **bruce**. Quando la funzione è invocata, essa stampa il valore del parametro (qualunque esso sia) due volte.

Questa funzione accetta qualsiasi valore che possa essere visualizzato:

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

Le stesse regole di composizione che si applicano alle funzioni predefinite si applicano alle funzioni definite dall'utente, quindi possiamo usare ogni tipo di espressione come argomento per *print_twice*

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

L'argomento è valutato prima della chiamata alla funzione, quindi negli esempi le espressioni 'Spam'*4e math.cos (math.pi) sono valutate solamente una volta.

Si può anche usare una variabile come argomento:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

Il nome della variabile che passiamo come un argomento (michael) non ha nulla a che fare con il nome del parametro (bruce). Non importa come il valore era chiamato all'inizio: qui in *print_twice* lo chiamiamo tutti **bruce**.

4.10 Funzioni fruitful e funzioni void

Alcune delle funzioni che usiamo, come le funzioni matematiche, producono risultati: in assenza di un nome migliore le chiamo **funzioni fruttuose** o *fruitful functions*. Altre funzioni come *print_twice* compiono delle azioni ma non ritornano un valore. Esse sono chiamate **funzioni vuote** o *void functions*.

Quando invochi una funzione *fruitful*, vuoi praticamente sempre fare qualcosa con il risultato come, ad esempio, assegnarlo ad una variabile o usarlo come parte di un'espressione:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

Quando invochi una funzione in modalità interattiva, Python mostra il risultato:

```
>>> math.sqrt(5)
2.2360679774997898
```

Ma in uno script, se invochi una funzione *fruitful* e non salvi il valore della funzione in una variabile, il valore restituito svanisce nella nebbia!

```
math.sqrt(5)
```

Lo script calcola la radice quadrata di 5, ma dal momento che né salva il risultato in una variabile né lo mostra a video, non risulta essere molto utile.

Le funzioni *void* possono mostrare qualcosa sullo schermo o avere qualche altro effetto,

ma non restituiscono un valore. Se cerchi di assegnare il loro output a una variabile, viene restituito un valore speciale detto **None**.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print result
None
```

Il valore **None** non è la stessa cosa della stringa 'None'. Esso è un valore speciale che ha il suo tipo dedicato:

```
>>> print type(None)
<type 'NoneType'>
```

Per ottenere un risultato da una funzione, usiamo l'istruzione **return** all'interno della nostra funzione. Per esempio, potremmo creare una funzione molto semplice chiamata *addtwo* che somma due numeri e restituisce il risultato di tale somma.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print x
```

Quando lo script viene eseguito, l'istruzione print mostra "8" perché la funzione *addtwo* è stata invocata con gli argomento 3 e 5. La funzione calcola la somma dei due numeri e la assegna alla variabile locale **added** e usa l'istruzione **return** per restituire il valore calcolato al codice che l'ha invocata. Questo valore viene quindi assegnato alla variabile x e visualizzato sullo schermo.

4.11 Perché usare le funzioni?

Ci si potrebbe interrogare sull'utilità di dividere un programma in funzioni. Ci sono diverse ragioni per farlo:

- Creare una nuova funzione dà l'opportunità di dare un nome ad un gruppo di istruzioni, cosa che rende il tuo programma più facile da leggere, capire e debuggare.
- Le funzioni rendono un programma più compatto eliminando le porzioni ripetute

di codice. Se in seguito dovesse capitarti di fare una modifica, potrai farla in un unico posto.

- Dividere un programma lungo in funzioni ti consente di debuggare una parte alla volta e di assemblare il tutto in un insieme globalmente funzionante.
- Funzioni ben scritte sono spesso utili per molti programmi diversi. Una volta scritta e debuggata, una funzione può essere riusata tutte le volte che vuoi.

Per tutto il resto del libro, useremo spesso una funzione per spiegare un concetto. Parte dell'abilità di creare e usare le funzioni risiede nella capacità di catturare efficacemente un'idea come "trovare il valore più piccolo in una lista di valori". Più avanti ti mostreremo il codice che trova il valore più piccolo in una lista di valori e te lo presenteremo come una funzione chiamata min che accetta una lista di valori come argomento e restituisce il valore più piccolo della lista.

4.12 Debugging

Se usi un editor di testi per scrivere i tuoi script, potresti incorrere in fastidi dovuti a spazi e tabulazioni. Il modo migliore per evitare questi problemi consiste nell'usare solo spazi e non tabulazioni. La maggior parte degli editor che supportano Python fanno questo di default, ma non tutti.

Tabulazioni e spazi sono spesso invisibili, il che li rende difficili da debuggare, quindi cerca un editor che gestisca l'indentazione per te in modo efficace.

Non dimenticarti di salvare il programma prima di eseguirlo. Alcuni ambienti di sviluppo lo fanno automaticamente, ma alcuni no. In questo caso il programma che vedi nell'editor di testo non è lo stesso che stai eseguendo.

Il debugging può richiedere molto tempo se continui ad eseguire lo stesso programma sbagliato!

Accertati che il codice che stai vedendo sia quello che stai eseguendo. Se non sei sicuro, metti qualcosa del tipo *print 'hello'* all'inizio del codice ed eseguilo di nuovo. Se non vedi **hello**, non stai eseguendo il programma corretto!

4.13 Glossario

algorithm (algoritmo): un processo generale per risolvere una categoria di problemi

argument (argomento): un valore fornito a una funzione quando essa viene invocata. Il valore è assegnato al parametro corrispondente nella funzione.

body (corpo): la sequenza di istruzioni all'interno di una definizione di funzione.

composition (composizione): usare un'espressione come parte di una espressione più grande, o un'istruzione come parte di un'istruzione più grande.

deterministic (deterministico): attinente ad un programma che fa la stessa cosa ogni volta che viene eseguito, dato lo stesso input.

dot notation (notazione punto): la sintassi per chiamare una funzione in un altro modulo specificando il nome del modulo seguito da un punto (dot) e il nome di funzione.

flow of execution (flusso di esecuzione): l'ordine in cui le istruzioni sono lanciate durante l'esecuzione di un programma.

fruitful function (funzione fruttuosa): una funzione che restituisce un valore.

function (funzione): una sequenza nominata di istruzioni che compie alcune utili operazioni. Le funzioni possono o meno accettare argomenti e possono o meno produrre un risultato.

function call (chiamata a funzione): un'istruzione che esegue una punzione. Consiste nel nome della funzione seguita da una lista di argomenti.

function definition (definizione di funzione): un'istruzione che crea una nuova funzione, specificandone il nome, i parametri e le istruzioni che la compongono.

function object (oggetto funzione): un valore creato da una definizione di funzione. Il nome della funzione è una variabile che si riferisce ad un oggetto funzione.

header (header): la prima riga di una funzione.

import statement (istruzione di importazione): un'istruzione che legge un file modulo e crea un oggetto modulo.

module object (oggetto modulo): un valore creato da un'istruzione di importazione che fornisce accesso ai dati e al codice definito in un modulo.

parameter (parametro): un nome usato all'interno di una funzione per riferirsi al valore passato come argomento.

pseudorandom (pseudocasuale): attinente ad una sequenza di numeri che sembrano essere casuali ma sono generati da un programma deterministico.

return value (valore restituito): il risultato di una funzione. Se la chiamata a funzione è usata come un'espressione, il valore restituito è il valore dell'espressione.

void function (funzione vuota): una funzione che non restituisce un valore.

4.14 Esercizi

Esercizio 4 Qual è lo scopo della parola riservata def in Python?

a) E' un'espressione gergale che significa "il codice seguente è molto cool"

b) Indica l'inizio della definizione di una funzione

c) Indica che la seguente sezione indentata di codice deve essere messa da parte per dopo

d) b e c sono entrambe vere

e) nessuna delle precedenti

Esercizio 5 Cosa stamperà il seguente codice Python?

```
def fred():
   print "Zap"

def jane():
   print "ABC"

jane()
fred()
jane()
```

a) Zap ABC jane fred jane

b) Zap ABC Zap

c) ABC Zap jane

d) ABC Zap ABC

e) Zap Zap Zap

Esercizio 6 Riscrivi il codice per il calcolo della paga con lo straordinario al 150% e crea una funzione chiamata computepay che accetta dua parametri (**hours** e **rate**).

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Esercizio 7 Riscrivi il programma di calcolo dei voti dal capitolo precedente usando una funzione chiamata *computergrade* che accetta il punteggio come parametro e restituisce un voto come stringa

```
Score   Grade
> 0.9     A
> 0.8     B
> 0.7     C
> 0.6     D
<= 0.6    F
```

Esecuzione del programma:

```
Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

Capitolo 5 - Iterazione

5.1 Aggiornare le variabili

Un pattern comune tra le istruzioni di assegnamento è un'istruzione di assegnazione che aggiorna una variabile - dove il nuovo valore della variabile dipende dal vecchio.

```
x = x+1
```

Questo significa "prendi il valore corrente di x, aggiungi uno, e poi aggiorna x con il nuovo valore."

Se cerchi di aggiornare una variabile che non esiste, ricevi un errore, perché Python valuta il lato destro prima di assegnare un valore a x:

```
>>> x = x+1
NameError: name 'x' is not defined
```

Prima che tu possa aggiornare una variabile, devi *inizializzarla*, generalmente mediante una semplice assegnazione:

```
>>> x = 0
>>> x = x+1
```

Aggiornare una variabile aggiungendo 1 è detto *incremento*, mentre l'operazione opposta è detta *decremento*.

5.2 - Il costrutto WHILE

I computers sono spesso utilizzati per automatizzare compiti ripetitivi. Ripetere compiti simili o identici senza fare errori è qualcosa che i computers fanno bene e gli esseri umani no. Siccome l'iterazione è molto comune, Python è dotato di varie istruzioni che la rendono più semplice.

Una forma di iterazione in Python è il costrutto *while*. Ecco un semplice programma che conta da cinque in giù e poi dice "Blastoff!".

```
n = 5
while n > 0:
    print n
    n = n-1
print 'Blastoff!'
```

Si può leggere il costrutto while come se fosse italiano. Quesi codice potrebbe essere tradotto come “Finché n è maggiore di 0, mostra il valore di n e riduci il valore di n di 1. Quando arrivi a 0, esci dal costrutto while e mostra la parola Blastoff!”.

Più formalmente, questo è il flusso di esecuzione di un costrutto while:

1. Valuta la condizione, restituendo True o False.

2. Se la condizione è false, esci dal costrutto while e procedi all'istruzione successiva.

3. Se la condizione è vera, esegui il corpo e poi torna indietro al passo 1.

Questo tipo di flusso è detto ciclo perché al terzo passo ritorna all'inizio. Tutte le volte che viene eseguito il corpo del ciclo, si ha una *iterazione*. Del ciclo (o loop) visto in precedenza diremmo “Aveva cinque iterazioni”, il che significa che il corpo del ciclo è stato eseguito cinque volte.

Il corpo del ciclo dovrebbe cambiare il valore di una o più variabili, di modo che eventualmente la condizione diventi falsa e il ciclo termini. La variabile, che cambia tutte le volte che viene eseguito il ciclo e che controlla che il ciclo termini, è detta variabile di *iterazione*. Se questa variabile non c'è, il ciclo si ripeterà all'infinito, risultando un *ciclo infinito*.

5.3 Cicli infiniti

Una fonte infinita di divertimento per i programmatori è l'osservazione che le indicazioni sullo shampoo “insapona, risciacqua, ripeti” sono un ciclo infinito perché non c'è una *variabile di iterazione* a dirti quante volte eseguire il ciclo.

Nel caso del conto alla rovescia, possiamo provare che il ciclo termina perché sappiamo che il valore di n è finito, e possiamo vedere che il valore di n si riduce tutte le volte attraverso il ciclo, di modo che dobbiamo raggiungere lo 0. Altre volte un ciclo è infinito perché non ha alcuna variabile di iterazione.

5.4 “Cicli infiniti” e break

A volte non sai che è il momento di terminare un ciclo finché non arrivi a metà del ciclo stesso. In questo caso puoi scrivere di proposito un *ciclo infinito* e poi utilizzare break per uscire dal ciclo.

Il ciclo è ovviamente un ciclo infinito perché l'espressione logica è semplicemente la costante logica True:

```
n = 10
while True:
    print n,
    n = n - 1
print 'Done!'
```

Se commetti l'errore di testare questo codice, imparerai velocemente come fermare sul tuo sistema un processo di Python fuori controllo dove si trova il tasto di spegnimento del tuo computer. Questo programma sarà eseguito per sempre finché sarà esaurita la tua batteria perché l'espressione logica in cima al ciclo è sempre vera per via del fatto che l'espressione è il valore costante “true”.

Sebbene questo sia un ciclo infinito disfunzionale, possiamo sempre usarlo per costruire cicli utili, a patto di inserire con cura nel corpo del ciclo, del codice che esplicitamente esca dal ciclo utilizzando break quando si è raggiunta la condizione di uscita.

Per esempio: supponiamo di voler prendere l'input dall'utente finché digita la parola 'done'. Potremmo scrivere:

```
while True:
    line = raw_input('> ')
    if line == 'done':
        break
    print line
print 'Done!'
```

La condizione del ciclo è True, di modo che il ciclo continui finché raggiunge l'istruzione break.

Ad ogni ciclo presenta all’utente un > per richiedere in input. Se l'utente scrive done, l'istruzione break esce dal ciclo. Altrimenti il programma stampa qualsiasi cosa l'utente

scriva e torna all'inizio del ciclo. Qui un piccolo esempio:

```
> hello there
hello there
> finished
finished
> done
Done!
```

Questo modo di scrivere un ciclo while è comune perché puoi verificare la condizione in qualsiasi punto del ciclo (non solo all'inizio) e puoi esprimere la condizione di stop affermativamente (“fermati quando questo accade”) piuttosto che negativamente (“fai questo finché questo accade”).

5.5 Terminare le iterazioni con continue

A volte ti trovi in un ciclo, vuoi terminare l'iterazione corrente e saltare all'iterazione successiva. In quel caso puoi utilizzare l'istruzione continue per saltare alla prossima iterazione senza terminare il blocco delle istruzioni del ciclo per l'iterazione corrente.

Questo è un esempio di ciclo che copia il suo input finché l'utente non scrive “done”, ma tratta le linee che cominciano con un cancelletto come linee che non devono essere stampate (un po' come per i commenti in Python):

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line
print 'Done!'
```

Questo è il risultato del nuovo programma con l'aggiunta di *continue*:

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

Sono stampate tutte le righe, tranne quella che comincia con il cancelletto perché, quando il *continue* è eseguito, termina l'iterazione corrente e torna indietro all'istruzione while per cominciare l'iterazione successiva, saltando la riga che esegue l'istruzione print.

5.6 Cicli definiti utilizzando for

A volte vogliamo ciclare attraverso un set di cose, come una lista di parole, le righe di un file o una lista di numeri. Quando abbiamo una lista di cosa attraverso cui ciclare, possiamo costruire un ciclo definito utilizzando una istruzione *for*. Definiamo il while un ciclo indefinito perché cicla finché una condizione diventa False mentre il ciclo for compie tante iterazioni quanti sono gli elementi di un oggetto.

La sintassi di un ciclo for è simile a quella del ciclo while in quanto ci sono una istruzione for e un corpo del ciclo:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print 'Happy New Year:', friend
print 'Done!'
```

In Python, la variabile friends è una lista di tre stringhe e il ciclo for attraversa la lista ed esegue il corpo una volta per ognuna delle tre stringhe nella lista risultando in questo output:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

Non si può tradurre il ciclo in italiano in maniera così diretta come lo si fa con il while ma se si pensa a friends come ad un set si può più o meno farlo così: "Esegui le istruzioni nel corpo del ciclo for una volta per ognuno degli amici che si trovano nel set friends.".

Guardando il ciclo *for* e *in* sono parole riservate di Python mentre friend e friends sono variabili.

```
for friend in friends:
    print 'Happy New Year', friend
```

In particolare, friend è la variabile di *iterazione* per il ciclo for. La variabile friend

cambia ad ogni iterazione del ciclo e controlla quando viene completato il ciclo. La variabile di iterazione passa in maniera sequenziale attraverso la lista friends.

5.7 I pattern

Spesso utilizziamo un ciclo for o while per ciclare attraverso una lista di oggetti o i contenuti di un file e siamo alla ricerca di qualcosa come il più alto o il più basso valore trovato tra i dati durante la scansione.

Questi cicli sono generalmente costituiti da:

- l'inizializzazione di una o più variabili prima che il ciclo cominci;
- l'esecuzione di qualche calcolo su ciascun item nel corpo del ciclo, magari cambiare le variabili nel corpo del ciclo
- L'osservazione delle variabili al termine del ciclo

Useremo una lista di numeri per dimostrare i concetti e la costruzione di questi pattern.

5.7.1 - Cicli in cui si conta e si somma

Per esempio, per contare il numero di elementi in una lista, dovremmo scrivere il seguente ciclo:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print 'Count: ', count
```

Inizializziamo la variabile count a zero prima che il ciclo cominci, poi scriviamo un ciclo che itera sulla lista di numeri. La nostra variabile di iterazione è detta *itervar* e anche se non usiamo itervar all'interno del ciclo, essa controlla il ciclo e permette al corpo del ciclo di essere eseguito una volta per ciascuno dei valori contenuti nella lista.

Nel corpo del ciclo, aggiungiamo uno al valore corrente di count per ognuno dei valori presenti nella lista. Mentre il ciclo viene eseguito, il valore di count corrisponde al numero di valori che abbiamo incontrato “fino ad ora”.

Una volta che il ciclo termina, il valore di count corrisponde al numero totale degli elementi che si trovano nella lista. Il numero totale <"falls in our lap"> alla fine del ciclo. Costruiamo il ciclo in modo da avere quello che vogliamo quando il ciclo termina.

Un ciclo simile a questo calcola il totale dalla somma degli elementi di una lista:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print 'Total: ', total
```

In questo ciclo utilizziamo la variabile di *iterazione*. Invece di aggiungere semplicemente uno a count come avviene nel ciclo precedente, aggiungiamo il numero corrente (3, 41, 12, ecc.) al totale calcolato di volta in volta durante ciascuna iterazione. Se pensi alla variabile total, essa contiene "il totale dei valori raggiunto fino a questo momento". Quindi, prima che il ciclo venga avviato, total è zero perché non abbiamo ancora settato alcun valore, durante il loop è il totale raggiunto fino a quel momento e alla fine corrisponde alla somma di tutti i numeri presenti nella lista.

Mentre il ciclo viene eseguito, total accumula la somma di tutti gli elementi; una variabile usata in questo modo è chiamata talvolta *accumulatore*.

Sia il primo che il secondo ciclo sono per noi particolarmente utili nella pratica perché ci sono funzioni del linguaggio stesso come len() e sum() che computano il numero di elementi in una lista e il totale dato dalla somma degli elementi di una lista rispettivamente.

5.7.2 - Cicli per calcolare massimo e minimo

Per calcolare il valore più alto in una sequenza, costruiamo il ciclo seguente:

```
largest = None
print 'Before:', largest
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print 'Loop:', itervar, largest
print 'Largest:', largest
```

Eseguendo il programma otterremo il seguente output:

```
Before: None
Loop: 3 3
Loop: 41 41
```

```
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

Possiamo pensare alla variabile largest come al “più alto valore incontrato fino a questo momento". Prima del ciclo, settiamo largest alla costante *None*. *None* è un valore costante speciale che possiamo registrare in una variabile perché venga considerata “vuota”.

Prima che il ciclo cominci, il più alto valore incontrato fino a questo momento è None, dal momento che non abbiamo incontrato ancora alcun valore. Mentre il ciclo è in esecuzione, se largest è None allora viene assegnato a largest il primo valore incontrato. Lo si può vedere nella prima iterazione quando il valore di itervar è 3, largest è None ed immediatamente largest viene settato a 3.

Dopo la prima iterazione, largest non è più None, così la seconda parte della espressione logica composta che verifica che itervar>largest viene innescata solo quando incontriamo un valore che è più alto di quello “più alto fino ad ora”. Quando ne incontriamo uno “ancora più alto”, consideriamo il nuovo valore il più alto. Lo si può vedere dall'output del programma che il valore di largest passa da 3, a 41, a 74.

Alla fine del ciclo, sono stati valutati tutti gli elementi della lista e la variabile largest contiene il valore più alto tra tutti quelli incontrati.

Per calcolare il numero più piccolo, il codice è molto simile, anche se c'è un piccolo cambiamento:

```
smallest = None
print 'Before:', smallest
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print 'Loop:', itervar, smallest
print 'Smallest:', smallest
```

Di nuovo, smallest corrisponde al "più piccolo fino ad ora" prima, durante, e dopo l'esecuzione del ciclo. Quando il ciclo termina, smallest contiene il valore minimo incontrato nella lista.

Anche in questo caso, le funzioni del linguaggio max() e min() rendono non necessario scrivere questi tipo di ciclo.

Quella che segue è una semplice versione della funzione min() incorporata nel linguaggio Python:

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

Abbiamo creato una funzione e rimosso tutte le istruzioni print, in modo da avere l'equivalente della funzione min(), già presente in Python.

5.8 Debug

Man mano che cominci a scrivere programmi più grandi, ti potresti trovare a spendere più tempo a fare il debug. Più codice vuol dire maggiori possibilità di commettere un errore e più posti in cui i bug possono andarsi a nascondere.

Un modo per accorciare il tempo dedicato al debug è il “debug per bisezione”. Per esempio, se ci sono 100 righe nel tuo programma e tu le controlli una alla volta, ciò richiederà 100 step.

Puoi provare, invece, a spezzare il problema in due. Cerca al centro del programma, o da quelle parti, un valore che puoi controllare. Aggiungi una istruzione print (o qualcosa di simile che abbia un effetto verificabile) e lancia il programma.

Se il controllo è errato, il problema deve trovarsi nella prima parte del programma. Se è giusto, deve trovarsi nella seconda.

Tutte le volte che esegui controlli come questo, tu dimezzi il numero di righe tra le quali cercare. Dopo 6 passi (che è molto meno di 100), almeno in teoria, sarai giunto a una o due righe di codice.

In pratica, tuttavia, non è sempre chiaro cosa voglia dire “nel mezzo del programma” e non è sempre possibile verificarlo. Non ha senso contare le righe e trovare l'esatto punto centrale. Invece, pensa pensa ai punti del programma in cui potrebbero annidarsi degli errori e posti in cui è possibile inserire un controllo. Poi scegli un punto in cui pensi che ci sia la possibilità che il bug sia prima o dopo questo punto.

5.9 Glossario

accumulatore: una variabile usata in un ciclo per aggiungere o accumulare un risultato.

contatore: una variabile utilizzata in un ciclo per contare il numero di volte in cui qualcosa è accaduto. Inizializziamo un contatore a zero e poi incrementiamo il contatore tutte le volte che vogliamo "contare" qualcosa.

decremento: Un aggiornamento che fa decrescere il valore di una variabile.

inizializzazione: Una assegnazione che attribuisce un valore iniziale ad una variabile che sarà aggiornata.

incremento: Un aggiornamento che accresce il valore di una variabile (spesso di uno).

ciclo infinito: Un ciclo in cui la condizione di uscita non è mai soddisfatta o per la quale non c'è termine.

iterazione: Esecuzione ripetuta di un gruppo di istruzioni utilizzando o la chiamata ad una funzione ricorsiva o un ciclo.

5.10 Esercizi

Esercizio 5.1: Scrivi un programma che legge ripetutamente numeri finché l'utente inserisce 'done'. Una volta che l'utente ha scritto 'done', scrivi il totale, la somma e la media dei numeri. Se l'utente inserisce qualsiasi altra cosa diversa da un numero, rileva l'errore utilizzando try ed except, stampa un messaggio di errore e passa al numero successivo.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.33333333333
```

Esercizio 5.2: Scrivi un altro programma che richiede l'inserimento di numeri come nell'esempio precedente e alla fine stampa il minimo, il massimo invece che la media.

Capitolo 6 - Stringhe

6.1 Una stringa è una sequenza

Una stringa è una sequenza di caratteri. È possibile accedere ai caratteri uno alla volta con l'operatore con parentesi quadre:

```
>>> frutta = 'banane'
>>> lettera = frutta [1]
```

La seconda istruzione estrae il carattere nella posizione di indice 1 dalla variabile *frutta* e lo assegna alla variabile *lettera*.

L'espressione tra parentesi viene chiamata **indice**. L'indice definisce quale carattere vogliamo nella sequenza desiderata (da qui il nome).

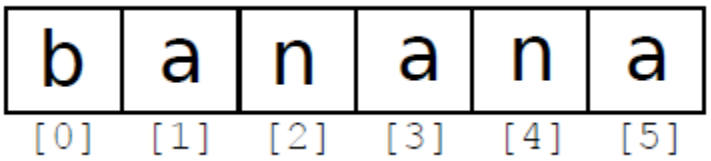
Ma potreste non ottenere quello che vi aspettate:

```
>>> print lettera
a
```

Per molte persone, la prima lettera di *'banana'* è *b*, non a. Ma in Python, l' indice è un offset dall'inizio della stringa e l'offset della prima lettera è zero.

```
>>> lettera = frutta [0]
>>> print lettera
b
```

Quindi b è la lettera 0 ("zero-esima") di 'banane', a è la lettera 1 ("prima"), e n è la lettera 2 ("seconda").



È possibile utilizzare qualsiasi espressione, comprese le variabili e gli operatori, come un indice, ma il valore dell'indice deve essere un numero intero. In caso contrario, si

ottiene:

```
>>> Lettera = frutta [1.5]
IndexError: string index out of range
```

6.2 Come ottenere la lunghezza di una stringa utilizzando *len*

len è una funzione built-in che restituisce il numero di caratteri di una stringa:

```
>>> frutta = 'banane'
>>> len(frutta)
6
```

Per ottenere l'ultima lettera di una stringa, si potrebbe essere tentati di provare qualcosa di simile a questo:

```
>>> lunghezza = len(frutta)
>>> ultima = frutta[lunghezza]
IndexError: string index out of range
```

La ragione per la IndexError è che non c'è nessuna lettera in 'banane' con indice 6. Dato che abbiamo iniziato a contare da zero, le sei lettere sono numerate da 0 a 5. Per ottenere l'ultimo carattere, si deve sottrarre 1 dalla lunghezza:

```
>>> ultima = frutta[length-1]
>>> print ultima
a
```

In alternativa è possibile utilizzare gli indici negativi, che contano a ritroso a partire dalla fine della stringa. L'espressione *frutta[-1]* produce l'ultima lettera, *frutta[-2]* estrae la penultima e così via.

6.3 Eseguire un *traversal* di una stringa con ciclo

Molte elaborazioni richiedono l'elaborazione delle stringhe un carattere alla volta. Spesso selezioniamo dall'inizio un carattere alla volta, facciamo qualcosa, e proseguiamo fino alla fine. Questo modello di elaborazione è chiamato **traversal**. Un modo per scrivere un traversal è con un ciclo *while* :

```
index = 0
while indice < len(frutta):
    lettera = frutta [indice]
    print lettera
```

```
    indice = indice + 1
```

Questo ciclo attraversa la stringa e visualizza ogni lettera in una riga. La condizione del ciclo è *indice < len(frutta)*, così, quando l'indice è uguale alla lunghezza della stringa, la condizione è falsa, e il corpo del ciclo non viene eseguito. L'ultimo carattere cui si accede è quello con indice *len(frutta)-1*, che è l'ultimo carattere della stringa.

Esercizio 6.1 Scrivere un ciclo while che inizia all'ultimo carattere della stringa e procede all'indietro fino al primo carattere, stampando ogni lettera su una riga separata.

Un altro modo per scrivere un traversal è con un ciclo for :

```
for char in frutta:
    print char
```

Ad ogni iterazione del ciclo, il carattere successivo nella stringa viene assegnato alla variabile *char*. Il ciclo continua fino a quando non rimangono più caratteri.

6.4 Porzioni di Stringhe

Un segmento di una stringa è chiamato **slice**. La selezione di una porzione è simile alla selezione di un carattere:

```
>>> S = 'Monty Python' >>>
print s [0:5]
Monty
>>> Print s [6:13]
Python
```

L'operatore [*n:m]* restituisce la parte della stringa dal carattere "n-esimo" al carattere "m-esimo", includendo il primo, ma escludendo l'ultimo.

Se si omette il primo indice, prima dei due punti, la porzione comincia all'inizio della stringa. Se si omette il secondo indice, la porzione arriva alla fine della stringa:

```
>>> frutta = 'banana' >>>
frutta[:3]
'ban'
>>> Frutta [3:]
'ana'
```

Se il primo indice è maggiore o uguale al secondo il risultato è una stringa vuota, rappresentata da due virgolette:

```
>>> frutta = 'banane'
```

```
>>> frutta[3:3]
''
```

Una stringa vuota non contiene caratteri e ha lunghezza 0, ma a parte questo, è lo stesso di qualsiasi altra stringa.

Esercizio 6.2 Dato che frutta è una stringa, cosa significa frutta[:] ?

6.5 Le stringhe sono immutabili

Si può tentare di usare l'operatore *[ ]* sul lato sinistro di un'assegnazione, con l'intenzione di cambiare un carattere in una stringa. Per esempio:

```
>>> Saluto = 'Ciao , mondo!'
>>> Saluto[0] = 'J'
TypeError: object does not support item assignment
```

L' "object" in questo caso è la stringa e "item" è il carattere che si è tentato di assegnare. Per ora, un **object** è la stessa cosa di un valore, ma rivedremo tale definizione più avanti. Un **item** è uno dei valori in una sequenza.

Il motivo dell'errore è che le stringhe sono immutabili, il che significa che non è possibile modificare una stringa esistente. Il meglio che puoi fare è creare una nuova stringa che è una variante dell'originale:

```
>>> Saluto = 'Ciao , mondo!'
>>> nuovo_saluto = 'J' + saluto[1:]
>>> print nuovo_saluto
Jiao , mondo!
```

Questo esempio che consente di concatenare una nuova prima lettera su una porzione di saluto non ha alcun effetto sulla stringa originale.

6.6 Cicli e il conteggio

Il seguente programma conta il numero di volte in cui una lettera compare in una stringa:

```
parola = 'banane'
count = 0
for lettera in parola:
    if lettera == 'a':
```

```
        count = count + 1
print count
```

Questo programma dimostra un altro modello di elaborazione chiamato **counter**. La variabile *count* viene inizializzata a 0 e poi incrementata ogni volta che viene trovata una a. Al termine del ciclo, *count* contiene il numero totale di 'a'.

Esercizio 6.3 Incapsulare questo codice in una funzione denominata conteggio, e strutturatela in modo che accetti la stringa e la lettera come argomenti.

6.7 L'operatore *in*

La parola *in* è un operatore booleano che esamina due stringhe e restituisce *True* se la prima appare come una sottostringa nella seconda:

```
>>> 'a' in ' banane '
True
>>> 'Seme' in 'banane'
False
```

6.8 Confronto tra stringhe

Gli operatori di confronto lavorano sulle stringhe. Per vedere se due stringhe sono uguali:

```
if parola == 'banane':
    print 'Va bene, banane.'
```

Altre operazioni di confronto sono utili per mettere le parole in ordine alfabetico:

```
if parola < 'banane':
    print 'la tua parola, ' + parola + ' , viene prima di banane.'
elif parola > 'banane':
    print 'la tua parola, ' + parola + ' , viene dopo banane.'
else:
    print 'Va bene, banane.'
```

Python non gestisce le lettere maiuscole e minuscole allo stesso modo delle persona. Tutte le lettere maiuscole vengono prima di tutte le lettere minuscole , quindi:

```
La tua parola , Pesca , viene prima di banana.
```

Un modo comune per affrontare questo problema è quello di convertire stringhe in un formato standard, ad esempio in tutto minuscolo, prima di effettuare la comparazione. Tenetelo in mente in caso dobbiate difendervi da un uomo armato di pesche.

6.9 *Metodi* delle stringhe

Le stringhe sono un esempio di **oggetti (object)** Python. Un oggetto contiene sia i dati (la stringa stessa) che i **metodi**, (funzioni incorporate nell'oggetto) e sono a utilizzabili da qualsiasi **istanza** dell'oggetto.

Python ha una funzione chiamata *dir* che elenca i metodi disponibili per un oggetto.

La funzione di *type* indica il tipo di un oggetto e la funzione *dir* mostra i metodi disponibili.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:
capitalize(...)
    S.capitalize() -> string
    Return a copy of the string S with only its first character
    capitalized.
>>>
```

Mentre la funzione *dir* elenca i metodi, ed è possibile utilizzare *help* per ottenere una documentazione su un metodo, una fonte di documentazione più estesa per i metodi delle stringhe è https://docs.python.org/2/library/stdtypes.html#string-methods.

Chiamare un metodo è simile al chiamare una funzione - prende argomenti e restituisce un valore - ma la sintassi è diversa. Chiamiamo un **metodo** aggiungendo il metodo al nome della variabile con un punto come delimitatore.

Ad esempio, il metodo *upper* prende stringa e restituisce una nuova stringa con tutte le lettere in maiuscolo:

Invece della sintassi della funzione *upper(parola)*, si utilizza la sintassi del metodo *word.upper()*.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

Questa forma di notazione con il punto specifica il nome del metodo, *upper*, e il nome della stringa a cui applicare il metodo, *word*. Le parentesi vuote indicano che questo metodo non richiede alcun argomento.

Una chiamata a un metodo viene chiamato una **invocazione**; in questo caso, diremmo che stiamo invocando *upper* su word.

Inoltre, c'è un metodo denominato *find* che è molto simile alla funzione che abbiamo scritto:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

In questo esempio, invochiamo *find* su *word* e passiamo la lettera che stiamo cercando come un parametro.

In realtà, il metodo *find* è più esteso che nella nostra funzione; si possono trovare anche stringhe e non solo caratteri:

```
>>> word.find('na')
2
```

Si può anche usare un secondo parametro: l'indice da dove iniziare la ricerca:

```
>>> word.find('na',3)
4
```

Un'operazione frequente è quella di rimuovere gli spazi bianchi (spazi, tabulazioni o ritorni a capo) all'inizio e alla fine di una stringa utilizzando il metodo *strip*:

```
>>> line = ' Here we go '
>>> line.strip()
'Here we go'
```

Alcuni metodi come *startswith* restituiscono *True*.

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

Noterete che *startswith* discrimina maiuscole e minuscole quindi a volte è utile convertire la stringa in tutto minuscolo con il metodo lower prima di fare qualsiasi controllo.

```
>>> line = 'Please have a nice day'
>>> line.startswith('p')
False
>>> line.lower()
'please have a nice day'
>>> line.lower().startswith('p')
True
```

Nell'ultimo esempio, viene chiamato il metodo *lower* e poi usiamo *startswith* per controllare se la stringa in caratteri minuscoli inizia con la lettera "p". Se prestiamo attenzione all'ordine di esecuzione, si possono fare più chiamate al metodo in un unica espressione.

Esercizio 6.4 C'è un metodo chiamato count che è simile alla funzione nell'esercizio precedente. Leggere la documentazione di questo metodo su docs.python.org/library/string.html e scrivere una invocazione che conta il numero di volte che la lettera a occorre in 'banana'.

6.10 Parsing delle stringhe

Spesso abbiamo la necessità di scorrere una stringa e trovare una sottostringa. Ad esempio, se avessimo una serie di linee formattate come segue:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

e volessimo estrarre solo la seconda metà dell'indirizzo (cioè uct.ac.za) da ogni riga, possiamo farlo utilizzando il metodo *find* e la tecnica di *slicing* della stringa. Prima troveremo la posizione del segno @; poi troveremo la posizione del primo spazio dopo il segno @; poi useremo lo slicing per estrarre la parte della stringa che stiamo cercando.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print atpos
21
>>> sppos = data.find(' ',atpos)
>>> print sppos
31
>>> host = data[atpos+1:sppos]
>>> print host
uct.ac.za
>>>
```

Usiamo una versione del metodo *find* che ci permette di specificare una posizione nella stringa da dove iniziare la ricerca. Quando si effetua lo *slicing*, si estrae il carattere da *"uno dopo @"* fino a*, ma non incluso, lo spazio*.

La documentazione per il metodo find è disponibile all'indirizzo:

docs.python.org/library/string.html.

6.11 Operatore *format*

L'operatore **format** % permette di costruire stringhe attraverso la sostituzione di parti dellla stringa con dati memorizzati in variabili. Quando viene applicato a numeri interi, % è il modulo. Quando il primo operando è una stringa, % è l'operatore format.

Il primo operando è format, che contiene una o più sequenze di formato che specificano come verrà formattato il secondo operando. Il risultato è una stringa. Ad esempio , la

sequenza di **format** '%d' significa che il secondo operando deve essere formattato come un numero intero (d sta per "decimale"):

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

Il risultato è la stringa '42 ', da non confondersi con il valore intero 42.

Una sequenza di **format** può apparire ovunque nella stringa, in modo da poter incorporare un valore in una frase:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

Se vi è più di una sequenza format nella stringa, il secondo argomento deve essere un essere una tuple. Ogni sequenza di format è abbinata ad un elemento del tuple, in ordine.

L'esempio seguente utilizza '%d' formattare un numero intero, '%g' per formattare a virgola mobile (non chiedetemi perché), e '%s' per formattare una stringa :

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

Il numero di elementi nella tuple deve corrispondere al numero di sequenze di format nella stringa . Inoltre , i tipi di elementi devono corrispondere le sequenze di format :

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

Nel primo esempio non ci sono abbastanza elementi; nel secondo l'elemento è di tipo sbagliato. L'operatore format è molto efficace, ma può essere difficile da usare.

Per ulteriori informazioni: https://docs.python.org/2/library/stdtypes.html#string-formatting.

6.12 Debugging

Una capacità che si dovrebbe coltivare quando si programma è di chiedersi sempre: "Che cosa potrebbe andare male qui? " o, in alternativa , "Che cosa pazzesca potrebbe fare il nostro utente per mandare in crash il nostro (apparentemente) programma perfetto?".

Ad esempio, guardiamo il programma che abbiamo usato per dimostrare il ciclo while nel capitolo sull'iterazione:

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line

print 'Done!'
```

Guardate cosa succede quando l'utente immette una riga vuota di ingresso:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
   File "copytildone.py", line 3, in <module>
      if line[0] == '#' :
```

Il codice funziona bene fino a quando non viene inserita una riga vuota: il carattere con indice 0 non esiste e quindi si verifica un traceback . Ci sono due soluzioni per rendere sicura la riga tre del programma, anche se la stringa è vuota.

Una possibilità è quella di utilizzare semplicemente il metodo startswith che restituisce False se la stringa è vuota.

```
if line.startswith('#'):
```

Un altro modo è utilizzare il sistema guardian per fare in modo che la seconda espressione logica venga valutata solo in presenza di almeno un carattere della stringa:

```
if len(line) > 0 and line[0] == '#':
```

6.13 Glossario

count: Variabile usata per contare qualcosa, di solito inizializzata a zero e poi incrementata.

stringa vuota: Stringa senza caratteri e lunghezza 0 , rappresentate da due vilgolette.

operatore format: Un operatore , % , che utilizza un format e un tuple per generere una stringa che include gli elementi del tuple formattati come specificato dal stringa di format.

sequenza di format: Una sequenza di caratteri di un format, come %d, che specifica come deve essere formattato un valore.

stringa format: Stringa, utilizzato con l' operatore format, che contiene le sequenze di format.

flag: Una variabile booleana usata per indicare se una condizione è vera .

invocazione: Un'istruzione che chiama un metodo.

immutabile: La proprietà di una sequenza i cui elementi non possono essere ri-assegnati .

indice: Un valore intero utilizzato per selezionare un elemento in una sequenza , ad esempio un carattere in una stringa.

item: Uno dei valori in una sequenza.

metodo: Una funzione che è associata a un oggetto ed è chiamata usando la notazione punto.

oggetto (object): Qualcosa a cui può fare riferimento. Per ora si può usare " oggetto" e "valore" in modo intercambiabile.

ricerca: Un modello di scorrimento che si ferma quando trova quello che sta cercando.

sequenza: Un insieme ordinato ; cioè un insieme di valori in cui viene identificato ciascun valore da un indice intero .

slice: Una parte di una stringa specificata da una serie di indici .

traverse: Scorrere gli elementi in una sequenza , eseguendo un'operazione simile su ciascuno di essi.

6.14 Esercizi

Esercizio 6.5 Considerare il seguente codice Python che memorizza una stringa:

```
'str = ' X-DSPAM-Confidence: 0,8475'
```

Utilizzare *find* e la tecnica di *slicing* per estrarre la parte della stringa dopo i due punti e quindi utilizzare la funzione float per convertire la stringa estratta in un numero in virgola mobile .

Esercizio 6.6 Leggi la documentazione dei metodi delle stringhe su https://docs.python.org/2/library/stdtypes.html#string-methods.

Potresti sperimentare alcuni metodi per assicurarti di capirne il funzionamento.

strip e *replace* sono particolarmente utili.

La documentazione utilizza una sintassi che potrebbe essere fonte di confusione. Ad esempio , in *find(sub[, start[, end]])*, le parentesi indicano argomenti opzionali, *sub* è necessario, ma *start* è facoltativo , e se si include *start*, poi *end* è opzionale.

Capitolo 7

Files

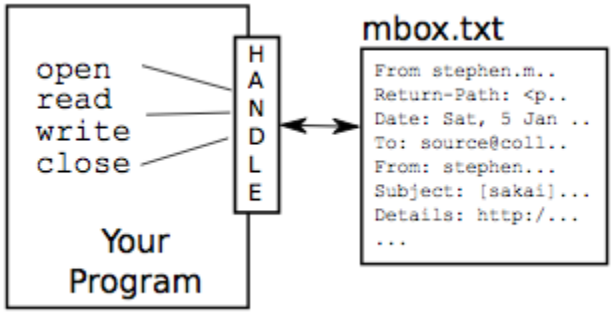
7.1 Persistenza

Abbiamo imparato come scrivere i programmi e comunicare le nostre intenzioni alla CPU attraverso esecuzioni condizionali, funzioni e iterazioni e abbiamo imparato come creare e utilizzare strutture di dati nella memoria centrale.

I nostri programmi funzionano nella CPU e nella memoria: è dove il software "pensa".

Se vi ricordate la nostra discussione sull'architettura dell'hardware, una volta che il PC viene spento tutto ciò che è iin quel momento presente e memorizzato nella CPU o memoria principale viene cancellato.

Fin qui i nostri programmi sono stati solo esercizi divertenti per imparare Python.



In questo capitolo iniziamo a lavorare con la memoria secondaria (o files). La memoria secondaria non viene cancellata se spegnamo il computer. Nel caso di una chiavetta USB, i dati possono essere salvati dai nostri programmi e trasportati su altri computers.

Ci concentreremo principalmente sulla lettura e la scrittura di file di testo come quelli che creiamo in un editor di testo. Più avanti vedremo come lavorare con i files di database che sono files binari specificamente progettati per essere letti e scritti attraverso il software di database.

7.2 Apertura di file

Quando vogliamo leggere o scrivere un file (diciamo dal disco rigido), per prima cosa dobbiamo **aprire (open)** il file. L'operazione di apertura viene fatta in comunicazione con il sistema operativo che conosce l'esatta posizione dei dati. Quando si apre un file, si sta chiedendo al sistema operativo di trovare il file in base al nome e di assicurarsi che il file esista. In questo esempio, apriamo il file *mbox.txt* che deve trovarsi nella stessa cartella in cui ci troviamo quando avvia Python. È possibile scaricare il file dal sito www.py4inf.com/code/mbox.txt

```
>>> Fhand = open ('mbox.txt')
>>> Fhand stampa
<open file 'mbox.txt', mode 'r' at 0x1005088b0>
```

Se l'apertura ha successo, il sistema operativo ci restituisce un **file handle**. Il file handle non rappresenta i dati effettivi contenuti nel file, ma semplicemente un "indirizzo" che possiamo usare per leggere i dati. Viene restituito un'handle se il file richiesto esiste e si dispone delle autorizzazioni appropriate per leggere il file.

Se il file non esiste, allora l'open fallirà, restituirà un traceback e non sarà possibile accedere al contenuto del file

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'stuff.txt'
```

Più avanti vedremo come usare *try* e *except* per gestire questo problema con eleganza.

7.3 file di testo e linee

Possiamo pensare ad un file di testo come una sequenza di linee nello stesso modo in cui, per Python, una stringa può essere una sequenza di caratteri. Ad esempio, questo è un file di testo, è un file che registra l'attività di posta di un team di sviluppo composto da varie persone relativamente ad un progetto open source:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

Il file completo è disponibile a questo indirizzo www.py4inf.com/code/mbox.txt mentre una versione abbreviata del file è disponibile qui: www.py4inf.com/code/mbox-short.txt.

Questi file sono nel formato standard per gli archivi di posta elettronica. Le linee che iniziano con "From" separano i messaggi e le linee che si aprono con "From:" sono parte dei messaggi. Per ulteriori informazioni, vedere en.wikipedia.org/wiki/Mbox .

Per separare il file in linee, esiste un carattere speciale che rappresenta "a capo" (newline). In Python, rappresentiamo questo carattere con una barra rovescia (backslash)-n (\n).

Anche se questi sembrano due caratteri, in realtà viene letto come un singolo carattere. Quando digitiamo una stringa nell'interprete Python, \n viene visualizzato nella stringa, ma quando si usa print \n la spezza in due parti.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print stuff
Hello
World!
>>> stuff = 'X\nY'
>>> print stuff
X
Y
>>> len(stuff)
3
```

Si può anche vedere che la lunghezza della stringa 'X\nY' è di tre caratteri perchè \n viene letto come un singolo carattere.

Così, quando guardiamo le righe in un file, dobbiamo immaginare che ci sia uno speciale carattere invisibile detto newline alla fine di ogni riga.

Così il carattere \n separa i caratteri nel file in linee.

7.4 La lettura dei file

Anche se l'handle non contiene i dati del file, è piuttosto semplice costruire un ciclo *for* per leggere e contare ciascuna delle righe in un file:

```
fhand = open('mbox.txt')
count = 0
for line in fhand:
       count = count + 1
print 'Line Count:', count


python open.py
Line Count: 132045
```

Usiamo l'handle al posto della sequenza nel nostro ciclo for. Il nostro ciclo for semplicemente conta il numero di righe nel file e lo stampa. La traduzione approssimativa del ciclo for è: "per ogni riga nel file rappresentato dal handle, aggiungere 1 alla variabile count".

La ragione per cui la funzione open non legge direttamente l'intero file è che potrebbe essere molto grande, anche molti gigabyte di dati. L'istruzione open impiega sempre il medesimo tempo indipendentemente dalle dimensioni del file. E' il ciclo for che in realtà esegue la lettura dei dati dal file.

Quando il file viene letto utilizzando un ciclo for, Python si prende cura di separare i dati nel file in linee utilizzando il carattere \n. Python legge ogni linea attraverso il ritorno a capo e comprende il ritorno a capo come ultimo carattere della riga per ogni iterazione del ciclo for .

Dato che il ciclo for legge i dati una riga alla volta, si può efficacemente leggere e contare le righe anche di grandi dimensioni senza esaurire la memoria principale per immagazzinare i dati .

Il programma di cui sopra può contare le righe in files di qualsiasi dimensione

utilizzando pochissima memoria perché ogni linea viene letta, contata e quindi scartata.

Se si conosce la dimensione del file ed è relativamente piccola rispetto alle dimensioni della memoria principale, si può leggere l'intero file in una stringa utilizzando il metodo *read*:

```
>>> hand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print len(inp)
94.626
>>> print inp[:20]
from stephen.marquar
```

In questo esempio l'intero contenuto (tutti i 94.626 caratteri) del file mbox-short.txt vengono letti direttamente nella variabile inp. Usiamo la tecnica di *slicing* per stampare i primi 20 caratteri della stringa memorizzata in inp.

Quando il file viene letto in questo modo, tutti i caratteri, tra cui tutte le linee e caratteri di fine riga diventano una grande stringa nella variabile **inp**.

Ricordate che questo utilizzo della funzione open deve essere adottato solo se i dati del file si adattano confortevolmente nella memoria principale del computer.

Se il file è troppo grande per essere contenuto nella memoria principale , si dovrebbe scrivere il programma che legge il file in blocchi utilizzando un ciclo *for* o *while*.

7.5 Ricerca dati in un file

Quando si eseguono ricerche in un file di dati, è una procedura comune scorrere il file ignorando la maggior parte delle righe e considerare solo quelle che soddisfano un particolare criterio. A tale scopo, possiamo combinare un modello di lettura del file con i **metodi** per le stringhe per costruire un semplice meccanismo di ricerca.

Ad esempio, se volessimo leggere un file e stampare solo le righe che iniziano con il prefisso "From:" si potrebbe utilizzare il metodo *startswith* per selezionare solo quelle con il prefisso desiderato:

```
fhand = open('mbox-short.txt')
for line in fhand:
       if line.startswith('From:') :
       print line
```

Quando questo programma viene eseguito si ottiene il seguente output:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
...
```

L'output sembra buono in quanto le uniche righe che stiamo vedendo sono quelle che iniziano con "From:"; ma perché ci sono quelle righe vuote in più? Ciò è dovuto al carattere invisibile \n (**newline**).

Ciascuna delle linee termina con un ritorno a capo, quindi l'istruzione *print* stampa la stringa nella variabile **line** che include \n dunque aggiunge un ritorno a capo, con conseguente effetto di doppia spaziatura che vediamo.

Potremmo usare la tecnica di slicing per tagliare l'ultimo carattere, ma esiste un un approccio più semplice è quello di utilizzare il metodo **rstrip** che mette rimuove gli spazi dal lato destro di una stringa:

```
fhand = open('mbox-short.txt')
for line in fhand:
       line = line.rstrip()
              if line.startswith('From:') :
              print line
```

Quando questo programma viene eseguito, si ottiene il seguente output:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

Man mano che i programmi di elaborazione dei file diventano più complicati, è possibile strutturare i cicli di ricerca utilizzando *continue*. L'idea di base del ciclo è ricercare le righe "interessanti" ed saltare quelle "non interessanti"; poi quando troviamo una riga interessante, fare qualcosa con quella riga.

Possiamo strutturare il ciclo in modo da saltare le righe poco interessanti come segue:

```
fhand = open('mbox-short.txt')
for line in fhand:
	line = line.rstrip()
	# Skip 'uninteresting lines'
	if not line.startswith('From:') :
		continue
	# Process our 'interesting' line
	print line
```

L'output del programma è lo stesso. In pratica, le linee che non iniziano con "From:" non ci interessano e vengono saltate utilizzando *continue*. Le righe che iniziano con "From:", sono quelle che che vogliamo elaborar.

Possiamo usare il metodo find per simulare una ricerca di testo che trova righe che contengono la stringa ricercata. Il metodo find cerca un'occorrenza di una stringa all'interno di un' altra stringa e restituisce -1 se non trova l'occorrenza nella mia stringa. Siamo quindi in grado di scrivere il seguente ciclo che trova e stampa le righe che contengono la sotto-stringa "@uct.ac.za" (cioè provengono dall'Università di Città del Capo in Sud Africa):

```
fhand = open('mbox-short.txt')
for line in fhand:
	line = line.rstrip()
	if line.find('@uct.ac.za') == -1 :
		continue
	print line
```

Che produce il seguente output:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan 4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
7.6. Letting the user choose the file name 85
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

7.6 Lasciamo scegliere all'utente il nome del file

Certamente non vogliano dover modificare il nostro codice Python ogni volta che desideriamo elaborare un file diverso . E' sicuramente più utilie per chiedere all'utente di inserire il nome del file ogni volta che il programma viene eseguito.

E' piuttosto semplice da fare: basta leggere il nome del file inserito dall'utente tramite

raw_input:

```
fname = raw_input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
       if line.startswith('Subject:') :
              count = count + 1
print 'There were', count, 'subject lines in', fname
```

Leggiamo il nome del file inserito dall'utente, lo assegnamo ad una variabile chiamata fname e la usiamo per aprire il file. Ora possiamo eseguire il programma più volte su file diversi.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

Senza guardare nella sezione successiva , diamo un'occhiata al programma sopra e chiediamoci "Che cosa potrebbe andare storto? " o "Cosa potrebbe fare il nostro gentile utente per causare una brusca uscita con un traceback del nostro piccolo programma, facendoci sembrare non-così-abili agli occhi dei nostri utenti?".

7.7 Utilizzo di try, except, e open

Avevo detto di non sbirciare. Questa è la tua ultima possibilità.

Che cosa succede se l'utente inserisce qualcosa che non è un nome di un file?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
File "search6.py", line 2, in <module>
fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
```

```
File "search6.py", line 2, in <module>
fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'na na boo boo'
```

Non ridete, gli utenti faranno ogni cosa possibile per mettere in difficoltà i vostri programmi, a volte anche in malafede. Di fatto, una parte importante di qualsiasi team di sviluppo software è una persona o un gruppo chiamato Quality Assurance (QA in breve) il cui compito principale è quello di fare le cose più folli nel tentativo di rompere il software che il programmatore ha creato.

E' responsabilità del team di QA trovare i difetti nei programmi prima che questi vengano messi in vendita, resi disponibili agli utenti finali o che il lavoro di programmazione venga pagato. Il team QA è il migliore amico del programmatore.

Quindi, ora che vediamo la falla nel programma, possiamo risolverla elegantemente mediante l'uso di try/except nella struttura. Supponendo che *open* potrebbe fallire, è possibile aggiungere delle righe di codice per intercettare il problema e gestirlo sul nascere come nellesempio che segue:

```
fname = raw_input('Enter the file name: ')
try:
      fhand = open(fname)
except:
      print 'File cannot be opened:', fname
      exit()
count = 0
for line in fhand:
      if line.startswith('Subject:') :
            count = count + 1
print 'There were', count, 'subject lines in', fname
```

La funzione *exit* termina il programma. Si tratta di una funzione di "non ritorno". Ora, quando il nostro utente (o gruppo QA) digita cose stupide o nomi di file errati siamo in grado di intercettarle e gestirle con eleganza.

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt
python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

L’uso di try e except per proteggere una chiamata open è un ottimo esempio di “buon codice” Python. Usiamo il termine "Pythonic" quando stiamo facendo qualcosa in "stile Python". Potremmo dire che l'esempio di cui sopra è un modo Pythonic per aprire un file .

Con il migliorare della vostra tecnica di programmazione potrete trovarvi a discutere con altri programmatori per decidere quale di due soluzioni equivalenti ad un problema è "più Pythonic". L'obiettivo di essere "più Pythonic" denota l'idea che la programmazione è in parte Ingegneria e in parte Arte Non siamo sempre solo interessati a far funzionare qualcosa, noi vogliamo che la nostra soluzione sia elegante e che sia considerata raffinata dai nostri colleghi.

#7.8 Scrittura dei file

Per scrivere un file, è necessario aprirlo in modalità 'w' come secondo parametro dell’istruzione *open*:

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

Se il file esiste già, aprendolo in modalità di scrittura i vecchi dati verranno cancellati e si riparte da zero, quindi state attenti! Se il file non esiste, ne viene creato uno nuovo.

Il metodo *write* dell'oggetto handle del file serve a inserire i dati nel file stesso.

```
>>> line1 = 'This here's the wattle,\n'
>>> fout.write(line1)
>
```

Anche in questo caso, l'oggetto file tiene traccia di dove si trova, quindi se si chiama di nuovo **write**, aggiunge i nuovi dati alla fine del file.

Dobbiamo porre attenzione alla gestire dei fine riga, inserendo esplicitamente il carattere \n (nuova riga) quando vogliamo terminarla. L'istruzione print lo aggiunge automaticamente, ma il metodo write no.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
```

Quando abbiamo finito di scrivere, si deve chiudere il file per assicurarsi che l'ultimo bit dei dati sia fisicamente scritto sul disco. In questo modo nessun dato andrà perso anche

in caso di spegnimento del computer.

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

E' possibile chiudere anche i file che apriamo in sola lettura. Se stiamo aprendo pochi file alla volta, potremmo anche evitare di scrivere le istruzioni per la chiusura. Questo perchè Python, al termine di un programma, fà in modo di chiudere tutti i file aperti in lettura. Se però stiamo scrivendo dei file, dobbiamo assicurarci di chiuderli in modo esplicito per evitare perdite di dati.

7.9 Debugging

Durante la lettura e/o scrittura dei files si può incorrere in problemi relativi agli spazi bianchi. Questi sono errori difficili da individuare in quanto gli spazi sono praticamente invisibili.

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

Può essere di aiuto la funzione integrata *repr*. Prende un qualsiasi oggetto come argomento e ritorna una rappresentazione in formato stringa dell'oggetto. Nella stringa visualizzata, gli spazi in questione sono rappresentati da sequenze con backslash \ (barra rovescia):

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

Questo è molto utile per il debug.

Un altro problema in cui si potrebbe incorrere è che diversi sistemi utilizzano caratteri diversi per indicare la fine di una riga. Alcuni sistemi utilizzano un ritorno a capo rappresentato da \n, altri usano un carattere di ritorno rappresentato da \r, alcuni usano entrambi. Se si sposta un file tra sistemi diversi, queste incongruenze potrebbero causare problemi.

Per la maggior parte dei sistemi esistono applicazioni di conversione da un formato all'altro.

Potete trovarli (e saperne di più su questo problema) su wikipedia.org/wiki/Newline. Ovviamente potete anche scriverlo voi stessi.

7.10 Glossario

cattura: Prevenire il termine errato di un programma utilizzando *try* ed *except*.

newline: Un carattere speciale utilizzato nei file e stringhe per indicare la fine di una riga.

Pythonic: Una tecnica che in Python funziona con eleganza. L'utilizzo di *try* ed *except* è un modo *Pythonic* per gestire gli errori.

Quality Assurance: una persona o un team concentrati sulla valutazione della qualità complessiva di un prodotto software. QA è spesso coinvolto nel testare un prodotto ed identificarne i problemi prima che il prodotto venga rilasciato.

file di testo: Una sequenza di caratteri memorizzati in una memoria permanente come un disco rigido.

7.11 Esercizi

Esercizio 7.1 Scrivere un programma per leggere un file e stampare il contenuto del file (riga per riga) tutto in maiuscolo. L'output del programma dovrà essere il seguente:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN 5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
SAT, 05 JAN 2008 09:14:16 -0500
```

È possibile scaricare il file da www.py4inf.com/code/mbox-short.txt

Esercizio 7.2 Scrivete un programma che richiede il nome di file, e quindi leggerlo attraverso il file e cercare le righe con questo formato:

```
X-DSPAM-Confidence: 0,8475
```

Quando si incontra una riga che inizia con "X-DSPAM-Confidence:" eseguire un operazione per estrarre il numero a virgola mobile. Contare queste righe e calcolare il totale dei valori di spam-confidence estratti. Quando si raggiunge la fine del file,

stampare la spam-confidence media

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

Provare il file sui file mbox.txt e mbox-short.txt .

Esercizio 7.3 A volte, quando i programmatori si annoiano o vogliono divertirsi un po' aggiungono un innocuo easter-egg al loro programma (en.wikipedia.org /wiki/Easter_egg_(media)) . Modificare il programma che richiede all'utente il nome file in modo che stampi un messaggio divertente quando l'utente digita esattamente 'Na na Boo Boo' . Il programma dovrebbe comportarsi normalmente per tutti gli altri file che esistono e non esistono. Ecco un esempio di esecuzione del programma:

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt

python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

Non vi stiamo incoraggiando a mettere gli easter-egg nei vostri programmi - questo è solo un esercizio.

Capitolo 8 - Liste

8.1 una lista è una sequenza

Come una stringa, una **lista** è una sequenza di valori. In una stringa, i valori sono caratteri; in una lista, possono essere di ogni tipo. I valori in una lista sono chiamati **elementi** o **items**.

Ci sono vari modi di creare una lista; il modo più semplice è quello di racchiudere gli elementi fra parentesi quadre ([...]):

```
python
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

Il primo esempio è una lista di quattro numeri interi. Il secondo è una lista di tre stringhe. Gli elementi di una lista non devono essere necessariamente tutti dello stesso tipo. La lista seguente contiene una stringa, un numero in virgola mobile (float), un intero e (attenzione!) un'altra lista:

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

Una lista all'interno di un'altra lista viene definita **annidata**.

Una lista che contiene zero elementi è chiamata **lista vuota**; si può creare una lista vuota con delle parentesi quadre vuote, [].

Come prevedibile, si possono assegnare i valori di una lista a delle variabili:

```
python
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

8.2 le liste sono mutabili

La sintassi per accedere agli elementi di una lista è la stessa di quella usata per accedere ai caratteri di una stringa: l'operatore [] .

L'espressione dentro le parentesi quadre specifica l'indice. Ricorda che gli indici partono da 0:

```
python
>>> print cheeses[0]
Cheddar
```

Diversamente dalle stringhe, le liste sono mutabili perché si può cambiare l'ordine degli elementi di una lista oppure riassegnare un elemento in una lista. Quando l'operatore [] appare nella parte sinistra di un'istruzione di assegnazione, identifica l'elemento della lista che sarà assegnato.

```
python
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

Il primo elemento di numbers, che era 123, ora è 5.

Si può considerare una lista come una relazione fra indici ed elementi. Questa relazione è chiamata **mapping**; ogni indice "mappa" uno degli elementi.

Gli indici di una lista funzionano allo stesso modo degli indici di una stringa:

* Qualsiasi espressione che dia come risultato un numero intero, può essere usata come indice.

* Se si prova a leggere o scrivere un elemento che non esiste, si ottiene un IndexError.

* Se un indice ha un valore negativo, conta all'indietro partendo dalla fine della lista.

L'operatore **in** funziona anche con le liste.

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

8.3 Fare il traversing di una lista

Il modo più comune di fare l'**elaborazione trasversale (traversal)** degli elementi di una lista e con l'utilizzo di un ciclo for. La sintassi è la stessa utilizzata con le stringhe:

```
python
for cheese in cheeses:
       print cheese
```

Questo funziona bene solo per la lettura degli elementi di una lista. Ma se si vuole modificare o aggiornare gli elementi, bisogna usare gli indici. Un modo comune per

farlo, è quello di combinare le funzioni range e len:

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

Questo ciclo elabora tutta la lista e aggiorna ogni elemento; len restituisce il numero di elementi nella lista; range restituisce una lista di indici da 0 a _n_-1, dove _n_ è la lunghezza della lista. Ad ogni iterazione del ciclo, *i* corrisponde all'indice del prossimo elemento. L'istruzione di assegnazione nel corpo del ciclo for, usa *i* per leggere il vecchio valore dell'elemento e assegnare il nuovo valore.

Se si applica un ciclo *for* su una lista vuota, le istruzioni al suo interno, non saranno mai eseguite:

```
python
for x in empty:
      print 'This never happens.'
```

Anche se una lista può contenere un'altra lista, la lista annidata viene considerata come un singolo elemento. Nella lista seguente ci sono 4 elementi:

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

8.4 Operazioni sulle liste

L'operatore + concatena le liste:

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

In modo analogo, l'operatore * ripete una lista un dato numero di volte:

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

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

8.5 Porzioni di liste

L'operatore **[:]** si utilizza anche con le liste:

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

Se si omette il primo indice, la porzione parte all'inizio della lista. Se si omette il secondo indice, la porzione arriva fino alla fine della lista. Per cui se si omettono entrambi, la porzione corrisponde ad una copia di tutta la lista.

```
python

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

Dato che le liste sono mutabili, risulta utile farne una copia prima di effettuare qualsiasi tipo di operazione su una lista.

Un operatore [:] sul lato sinistro di una istruzione di assegnazione può aggiornare elementi multipli:

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

8.6 Metodi delle liste

Python prevede metodi che operano sulle liste. Per esempio, **append** aggiunge un nuovo elemento alla fine di una lista:

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

extend prende come argomento una lista e la aggiunge alla fine di un'altra lista:

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

Questo esempio non modifica la lista t2.

sort ordina gli elementi di una lista dal più basso al più alto:

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

Molti metodi delle liste sono di tipo **void**; modificano la lista e restituiscono come valore None. Se provi a scrivere t = t.sort(), rimarrai deluso dal risultato.

8.7 Cancellare elementi da una lista

Ci sono vari modi per cancellare elementi da una lista, se si conosce l'indice dell'elemento che si vuole cancellare, si può usare **pop**:

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

pop modifica la lista e restituisce come valore, l'elemento rimosso; se non si fornisce un indice, cancella e restituisce l'ultimo elemento.

Se non serve il valore dell'elemento rimosso, si può usare l'operatore **del**:

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

Se si conosce l'elemento che si vuole rimuovere, ma non il suo indice, si può usare

remove:

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

remove restituisce come valore None.

Per rimuovere più di un elemento si può usare del con un operatore slice **[:]**:

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

Come al solito, l'operatore slice, seleziona tutti gli elementi dal primo fino al secondo indice, escluso quest'ultimo.

8.8 Liste e funzioni

Ci sono una serie di funzioni predefinite che possono essere usate sulle liste che permettono di esaminare velocemente una lista senza scrivere i propri cicli:

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

La funzione **sum()** si può utilizzare solo quando gli elementi di una lista sono dei numeri. Le altre funzioni (max(), len(), etc.) funzionano con liste di stringhe e altri tipi che possono essere comparati.

Usando una lista, potremmo riscrivere il programma precedente che calcolava la media di una lista di numeri digitati dall'utente.

Questo è il programma per calcolare una media, senza utilizzare una lista:

```
python
total = 0
count = 0
while ( True ) :
      inp = raw_input('Enter a number: ')
      if inp == 'done' : break
      value = float(inp)
      total = total + value
      count = count + 1

average = total / count
print 'Average:', average
```

In questo programma noi abbiamo le variabili count e total per tenere traccia del numero e per aggiornare il totale parziale finchè l'utente continua a inserisce numeri al prompt.

Potremmo invece semplicemente memorizzare ogni numero appena inserito dall'utente e usare le funzioni predefinite per calcolare solo alla fine la somma e il conteggio.

```
python
numlist = list()
while ( True ) :
      inp = raw_input('Enter a number: ')
      if inp == 'done' : break
      value = float(inp)
      numlist.append(value)

average = sum(numlist) / len(numlist)
print 'Average:', average
```

Creiamo una lista vuota prima che il ciclo inizi, e quindi ogni volta che abbiamo un numero, lo aggiungiamo in coda alla lista. Alla fine del programma, semplicemente calcoliamo la somma dei numeri nella lista e la dividiamo per il numero degli elementi presenti nella lista per ricavare la media.

8.9 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 che una stringa. Per convertire una stringa in una lista di caratteri si può usare **list**:

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

dato che list è il nome di una funzione predefinita, bisognerebbe evitare di usarla come nome di variabile. Io evito anche di usare *l*, come nome di variabile, in quanto molto simile a 1. Ecco perché uso *t*.

La funzione list divide una stringa in lettere singole. Se si vuole spezzare una stringa in parole, si può usare il metodo **split:**

```
python
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
>>> print t[2]
the
```

Dopo aver usato split per dividere la stringa in una lista di elementi, si può usare l'operatore indice (parentesi quadre) per esaminare una determinata parola nella lista.

Si può utilizzare split con un argomento opzionale chiamato **delimitatore** che specifica quali caratteri usare per individuare i confini delle parole. Il prossimo esempio, usa un trattino (-) come delimitatore:

```
python
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

join è l'opposto di split. Prende una lista di stringhe e ne concatena gli elementi; join è un metodo che agisce sulle stringhe, per cui bisogna invocarlo dal delimitatore e utilizzare la lista come parametro:

```
python
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

In questo caso il delimitatore è uno spazio, per cui join mette uno spazio fra le parole.

Per concatenare le stringhe senza spazi, si può utilizzare la stringa vuota, **''**, come delimitatore.

8.10 Fare il parsing delle linee

Di solito quando leggiamo un file vogliamo fare qualcos'altro con le linee oltre che semplicemente stampare l'intera linea. Spesso vogliamo isolare delle "linee interessanti" e quindi fare il **parsing** (valutazione) della linea per trovare delle parti interessanti. Come, ad esempio, stampare il giorno della settimana solo dalle linee che hanno come prima parola "From:".

```
python
From stephen.marquard@uct.ac.za <b>Sat</b> Jan 5 09:14:16 2008
```

Il metodo split è molto efficace quando usato per questo tipo di problemi. Possiamo scrivere un piccolo programma che cerca le linee che iniziano con "From" e quindi applicare il metodo split e quindi stampare la terza parola della linea:

```
python
fhand = open('mbox-short.txt')
for line in fhand:
      line = line.rstrip()
      if not line.startswith('From ') : continue
      words = line.split()
      print words[2]
```

nel codice precedente abbiamo utilizzato la forma contratta dell'istruzione *if*, accodando continue sulla stessa linea di if. Questa forma contratta dell'istruzione *if*, funziona esattamente come se continue si trovasse sulla prossima linea e indentato.

Il programma produce il seguente output:

```
python
Sat
Fri
Fri
Fri
      ...
```

In seguito impareremo delle tecniche sempre più sofisticate per individuare le linee da lavorare, per isolarle ed estrapolare solo l'esatta porzione di informazione che ci serve.

8.11 Oggetti e valori

Se eseguiamo queste istruzioni di assegnazione:

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

sappiamo che a e b si riferiscono entrambe a una stringa, ma non sappiamo se si riferiscono alla stessa stringa. Ci sono due possibili stati:

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

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

Nel primo caso a e b si riferiscono a due diversi oggetti che hanno lo stesso valore. Nel secondo caso, si riferiscono allo stesso oggetto.

Per controllare se due variabili si riferiscono allo stesso oggetto, si può usare l'operatore is:

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

In questo esempio, Python ha creato un solo oggetto stringa, e sia a che b si riferiscono ad esso.

Ma quando si creano due liste, si ottengono due oggetti:

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

In questo caso diremmo che le due liste sono **equivalenti**, perché hanno gli stessi elementi, ma non sono **identiche**, perché non sono lo stesso oggetto. Se due oggetti sono identici, allora sono anche equivalenti, mas se sono equivalenti, non sono necessariamente identici.

Finora abbiamo usato "oggetto" e "valore" in maniera intercambiabile, ma è più preciso dire che un oggetto ha un valore. Se si esegue a = [1, 2, 3] , a fa riferimento ad un oggetto di tipi lista, il cui valore è una determinata sequenza di elementi. Se un'altra lista avesse gli stessi elementi, diremmo che ha lo stesso valore.

8.12 Aliasing

Se a fa riferimento ad un oggetto e si esegue l'istruzione di assegnazione b = a, entrambe le variabili fanno riferimento allo stesso oggetto:

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

L'associazione di una variabile con un oggetto è chiamata **riferimento**: in questo esempio, ci sono due riferimenti allo stesso oggetto.

Un oggetto con più di un riferimento ha più di un nome, per cui si dice che l'oggetto ha uno o più **alias**.

Se l'oggetto con alias è mutabile, i cambiamenti fatti ad un alias modificano anche l'altro:

```
python
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

Anche se questa caratteristica può risultare utile, ci sono notevoli possibilità di errore. In generale, è più sicuro evitare di assegnare degli alias quando si sta lavorando con degli oggetti mutevoli.

Per gli oggetti immutevoli come le stringhe, la creazione di alias, non è affatto un problema. In questo esempio:

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

Non è quasi mai importante sapere se a e b si riferiscano allo stesso oggetto o meno.

8.13 Argomenti delle liste

Quando si passa una lista ad una funzione, la funzione fa un riferimento alla lista. Se la funzione modifica un parametro della lista, il cambiamento sarà visibile. Per esempio, **delete_head** rimuove il primo elemento da una lista:

```
python
def delete_head(t):
       del t[0]
```

Ecco come si usa:

```
python
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

Il parametro *t* e la variabile *letters* sono degli aliases per lo stesso oggetto.

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

```
python
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None
>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3]
>>> t2 is t3
False
```

Questa differenza è importante quando si scrivono funzioni che devono modificare le liste. Per esempio, questa funzione non cancella la testa (il primo elemento) di una lista:

```
python
def bad_delete_head(t):
       t = t[1:]           # WRONG!
```

L'operatore [:] crea una nuova lista e l'istruzione di assegnazione, fa sì che *t* faccia riferimento ad essa, ma niente di tutto ciò ha un qualche effetto sulla lista che è stata passata come argomento.

Un'alternativa e scrivere una funzione che creii e restituisca come valore una nuova lista. Per esempio, tail restituisce tutti gli elementi di una lista tranne il primo:

```
python
def tail(t):
       return t[1:]
```

Questa funzione lascia la lista originale immutata. Ecco come si usa:

```
python
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

Esercizio 8.1 Scrivere una funzione chiamata chop che modifica una lista, rimuovendo il primo e l'ultimo elemento e che restituisca come valore None.

Scrivere quindi una funzione chiamata middle che a partire da una lista, restituisca come valore una nuova lista che contiene tutti gli elementi tranne il primo e l'ultimo.

8.14 Debugging

L'utilizzo improprio di liste (e altri oggetti mutabili) può portare a dover affrontare ore e ore di debugging. Ecco alcuni errori comuni e vari modi per evitarli:

1. Non dimenticare che la maggior parte dei metodi delle liste modificano l'argomento e restituiscono come valore None. L'opposto di quello che fanno i metodi delle stringhe, che restituiscono una nuova stringa e lasciano immutata la stringa originale.

Se siete soliti scrivere codice per le stringhe come questo:

```
python
word = word.strip()
```

Vi verrà la tentazione di scrivere codice per le liste come questo:

```
python
t = t.sort()       # WRONG!
```

Ma dato che sort restituisce come valore None, la prossima operazione che utilizzerà t, probabilmente porterà ad un errore.

Prima di usare i metodi e gli operatori delle liste, è utile leggere attentamente la documentazione e fare dei test in modalità interattiva. I metodi e gli operatori delle liste, comuni anche ad altre sequenze (come le stringhe), sono documentati a _docs.python.org/lib/typesseq.html_.

I metodi e gli operatori che si applicano solo alle sequenze mutabili sono documentati a _docs.python.org/lib/typesseq-mutable.html_.

2. Scegli una tecnica e utilizza sempre quella.

Una parte della difficoltà di avere a che fare con le liste, deriva dal fatto che ci sono più modi di fare la stessa operazione. Ad esempio, per rimuovere un elemento da una lista, si può utilizzare, pop, remove, del, o addirittura un operatore [:].

Per aggiungere un elemento, si può usare il metodo append o l'operatore +. Ma non dimenticare che queste operazioni sono giuste:

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

Mentre le seguenti sono sbagliate:

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

Prova ciascuno di questi esempi in modalità interattiva per assicurarti di aver capito quello che fanno. Nota che solo l'ultimo esempio causerà un errore durante l'esecuzione; gli altri tre sono sintatticamente validi, ma portano a un risultato errato.

3. Fai delle copie per evitare di usare degli alias.

Se si vuole usare un metodo come sort che modifica l'argomento, ma al contempo si vuole mantenere inalterata la lista originale, puoi fare una copia.

```
python
orig = t[:]
t.sort()
```

In questo esempio si potrebbe anche utilizzare la funzione predefinita sorted, che restituisce una nuova lista ordinata e lascia inalterata l'originale. Ma in questo caso dovresti evitare di usare sorted come nome di variabile!

4. Liste, split, e files.

Quando leggiamo e facciamo il parsing dei files, ci sono molte possibilità di dover trattare degli input che possono far bloccare l'esecuzione del nostro programma, per cui è una buona idea riutilizzare il metodo del **guardian pattern** quando bisogna scrivere programmi che leggono un file alla ricerca dell' "ago nel pagliaio".

Riprendiamo il programma che cerca nel file, il giorno della settimana nelle linee la cui

prima parola è "From":

```
python
From stephen.marquard@uct.ac.za <b>Sat</b> Jan 5 09:14:16 2008
```

Dato che stiamo suddividendo le linee in singole parole, potremmo evitare di usare startswith e semplicemente esaminare la prima parola della linea per determinare se siamo interessati o meno alla linea in questione. Potremmo utilizzare continue per evitare le linee che non hanno "From" come prima parola, nel modo seguente:

```
python
fhand = open('mbox-short.txt')
for line in fhand:
      words = line.split()
      if words[0] != 'From' : continue
      print words[2]
```

Così sembra molto più semplice e non abbiamo nemmeno bisogno di usare rstrip per rimuovere il carattere di nuova linea alla fine del file. Ma è davvero una soluzione migliore?

```
python
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

Sembra funzionare perché vediamo nell'output, il giorno della prima linea (Sat), ma poi il programma si blocca con un errore. Cosa è andato storto? Quali dati disordinati hanno causato il crash del nostro programma, così elegante e "Pythonico"?

Potresti perderci del tempo e provare a modificare qualcosa, o chiedere aiuto a qualcuno, ma il modo più veloce e intelligente è quello di aggiungere una istruzione print. Il posto migliore dove aggiungere tale istruzione è proprio prima della linea che ha fatto bloccare il programma e che ha mostrato i dati che sembrano causare l'errore.

Con questo approccio si finirà col generare molte linee di output, Ma almeno si avrà immediatamente una idea di quale sia il problema. Aggiungiamo una istruzione print per la variabile words prima della linea cinque. aggiungiamo anche un prefisso "Debug:" alla linea in modo da mantenere l'output regolare separato dall'output di debug.

```
python
for line in fhand:
      words = line.split()
      print 'Debug:', words
      if words[0] != 'From' : continue
      print words[2]
```

Quando eseguiamo il programma, l'output scorrera' in basso nello schermo, ma lla fine, vedremo il nostro output di debug e l'errore, così capiamo esattamente cosa è successo prima dell'errore.

```
python
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

Ogni linea di debug stampa la lista delle parole che otteniamo quando applichiamo il metodo split alla linea. Quando il programma si blocca la lista delle parole è vuota []. Se apriamo il file in un editor di testo e guardiamo il file a quel punto, ecco cosa vediamo:

```
python
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan 5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772

L'errore occorre quando il nostro programma incontra una linea vuota! Naturalmente ci sono "zero parole" in una linea vuota. Come abbiamo fatto a non pensarci quando

abbiamo scritto il codice. Quando il codice cerca la prima parola (word[0]) per vedere se corrisponde a "From", otteniamo un errore "index out of range".

Questo naturalmente è il posto perfetto per aggiungere del codice di tipo **guardian pattern** per evitare di controllare la prima parola se la prima parola non c'è. Ci sono molti modi per proteggere questo codice, noi sceglieremo di far controllare il numero di parole che abbiamo prima di guardare alla prima parola:

```
python
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
      words = line.split()
      # print 'Debug:', words
      if len(words) == 0 : continue
      if words[0] != 'From' : continue
      print words[2]
```

Notare come abbiamo commentato l'istruzione di stampa di debug, piuttosto che rimuoverla, nel caso in cui la nostra modifica dovesse fallire e dovessimo aver bisogno di fare di nuovo il debug. Abbiamo quindi aggiunto una istruzione "di guardia" che controlla se abbiamo "zero parole" e se è il caso, usiamo continue per saltare alla prossima linea nel file.

Possiamo pensare alle due istruzioni continue come un mezzo per aiutarci a restringere il numero di linee che reputiamo "interessanti" e che quindi vogliamo processare ancora. Una linea che ha "zero parole" è "non interessante" per noi, per cui la saltiamo, così come saltiamo una riga che non abbia come prima parola "From".

Il programma così modificato viene eseguito con successo per cui forse è corretto; l'istruzione "di guardia" si assicura che che l'istruzione seguente, words[0], non produca errori, ma forse non è abbastanza. Quando stiamo programmando, dobbiamo sempre chiederci, "cosa potrebbe andare storto?".

Esercizio 8.2 Cercare di capire quale linea del programma precedente è ancora non sufficientemente protetta da errori. Prova a creare un file di testo che faccia bloccare il programma e quindi modifica il programma in modo che la linea venga propriamente protetta e esegui dei test per assicurarti che riesca a completare l'esame del nuovo file di

testo senza errori.

Esercizio 8.3 Riscrivi l'istruzione "di guardia" dell'esempio precedente, senza utilizzare due istruzioni if. Usa invece una espressione logica composta, usando l'operatore logico and e una sola istruzione if.

8.15 Glossario

aliasing: Una circostanza dove due o più variabili fanno riferimento allo stesso oggetto.

delimitatore: Un carattere o una stringa usati per indicare i punti dove una stringa dovrebbe essere suddivisa.

Elemento: Uno dei valori in una lista (o altra sequenza), anche chiamati items.

Equivalente: Che ha lo stesso valore.

Indice: un valore intero che indica un elemento in una lista.

Identico: Essere lo stesso oggetto (il che implica equivalenza).

Lista: Una sequenza di valori.

Traversal di liste: L'elaborazione sequenziale di ogni elemento in una lista.

Lista annidata: Una lista che è un elemento di un'altra lista.

Oggetto: qualcosa al quale una variabile può fare riferimento. Un oggetto ha un tipo e un valore.

Riferimento: l'associazione fra una variabile e il suo valore.

8.16 Esercizi

Esercizio 8.4 Scarica una copia del file da _www.py4inf.com/code/romeo.txt_.

Scrivi un programma per aprire il file romeo.txt e leggilo linea per linea. Per ogni linea, separa la linea in una lista di parole usando la funzione split.

Per ogni parola, controlla se la parola è già in una lista. Se la parola non è nella lista, aggiungila alla lista.

Quando il programma ha finito, ordina e stampa le parole risultanti in ordine alfabetico.

```
python
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

Esercizio 8.5 Scrivi un programma che legga dal file dei messaggi di posta e che quando trova una linea che inizia con "From" (di seguito chiamata - linea "From" -), suddivida la linea in parole usando la funzione split. Ci interessa individuare chi ha spedito il messaggio, ossia la seconda parola nella linea "From".

python

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

Il programma dovrà fare il parsing della linea "From" e stampare la seconda parola per ogni linea "From" e infine dovra anche contare quante sono le linne che iniziano con "From" (non "From:") e stamapre tale numero alla fine.

Questo è un esempio, con alcune righe rimosse, dell'output che dovrebbe produrre il programma:

```
python
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[... output rimosso...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

Esercizio 8.6 Riscrivi il programma che richiede all'utente una lista di numeri e quando l'utente ha digitato "done", stampa il numero più grande e il più piccolo. Fai in modo che il programma, memorizzi i numeri digitati dall'utente, in una lista e usa le funzioni min() e max() per calcolare il massimo e il minimo alla fine del ciclo.

```
python
Enter a number: 6
Enter a number: 2
```

```
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

Capitolo nove

Dizionari

Un dizionario è come una lista, ma più generale. In una lista, le posizioni (indici) devono essere dei numeri interi; in un dizionario gli indici possono essere (quasi) di ogni tipo.

Si può pensare a un dizionario come una relazione fra un insieme di indici (che sono chiamati chiavi) e un insieme di valori. Ogni chiave è in relazione con un valore. L'associazione di una chiave e un valore viene chiamata una coppia chiave-valore o a volte un elemento.

Come esempio, costruiremo un dizionario che mette in relazione parole inglesi e spagnole, quindi sia le chiavi che i valori sono di tipo stringa.

La funzione dict crea un nuovo dizionario senza elementi. Dato che dict è il nome di una funzione predefinita, bisognerebbe evitare di usarla come nome di variabile.

```
>>> eng2sp = dict()
>>> print eng2sp
{}
```

Le parentesi graffe, {}, rappresentano un dizionario vuoto. Per aggiungere elementi al dizionario, si possono usare le parentesi quadre:

```
>>> eng2sp['one'] = 'uno'
```

Questa linea crea un elemento che mette in relazione fra la chiave 'one' e il valore 'uno'. Se stampiamo di nuovo il dizionario vediamo una coppia chiave-valore con un doppio punto fra la chiave e il valore:

```
>>> print eng2sp
{'one': 'uno'}
```

Questo formato di output è anche un formato di input. Per esempio, si può creare un nuovo dizionario con tre elementi:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Ma con l'istruzione print, avresti una sorpresa:

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

L'ordine delle coppie chiave-valore non è lo stesso. Infatti se si digita lo stesso esempio al proprio computer, si otterranno risultati differenti. In generale, l'ordine degli elementi di un dizionario è imprevedibile.

Ma questo non è un problema, dato che gli elementi di un dizionario non sono mai indicizzati con degli interi. si usano le chiavi per cercare i corrispondenti valori:

```
>>> print eng2sp['two']
'dos'
```

Le chiave 'due' è sempre in relazione con il valore 'dos', per cui l'ordine degli elementi non ha importanza.

Se la chiave non è presente nel dizionario si ottiene un errore:

```
>>> print eng2sp['four']
KeyError: 'four'
```

La funzione len si può usare anche con i dizionari; restituisce il numero di copie chiavi-valore:

```
>>> len(eng2sp)
3
```

L'operatore in funziona sui dizionari; ci dice se qualcosa è presente come chiave nel dizionario (non trova invece elementi presenti come valore).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

Per vedere se qualcosa è presente come un valore in un dizionario, si può usare il metodo values, che restituisce i valori sotto forma di lista e si può quindi usare l'operatore in:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

L'operatore in usa diversi algoritmi per le liste e per i dizionari. Per le liste usa un algoritmo di ricerca lineare. Più lunga è la lista, maggiore sarà il tempo di ricerca, si allunga in proporzione diretta alla lunghezza della lista. Per i dizionari, Python usa un algoritmo chiamato 'Hash Table'a che ha una proprietà rimarchevole; l'opeatore in, ci mette lo stesso tempo indifferentemente dal numero di elementi presenti in un dizionario. Non spiegherò perché le funzioni di hash compiono tali magie, maggiori informazioni sono presenti su wikipedia

wikipedia.org/wiki/Hash_table

Esercizio 9.1

Scrivere un programma che legga le parole in words.txt e le memorizzi in un dizionario come chiavi. Non ha importanza quali siano i valori. Quindi usa l'operatore in come un modo veloce per controllare se una stringa si trova nel dizionario.

9.1 Un dizionario come un insieme di contatori.

Supponiamo che ti venga data una stringa e tu voglia contare quante volte ogni lettera appare. Ci sono vari modi di farlo

1.

Potresti scrivere 26 variabili, una per ogni lettera dell'alfabeto. Quindi potresti fare il traverse delle stringhe e, per ogni carattere, incrementare il contatore corrispondente, probabilmente usando una o più strutture condizionali

2.

Potresti creare una lista con 26 elementi. Quindi potresti convertire ogni carattere in un numero (usando la funzione predefinita ord), usare il numero come un indice della lista e incrementare l'apposito contatore.

3.

Potresti creare un dizionario che utilizza i caratteri come chiavi e dei contatori come i corrispondenti valori. La prima volta che si incontra un carattere, si aggiunge un elemento al dizionario, in seguito si procede con l'incrementare il valore di un elemento esistente.

Ognuna di queste opzioni fa una serie di calcoli per giungere allo stesso risultato, ma ognuna di loro con una diversa implementazione.

Una implementazione è un modo di fare un calcolo; alcune sono migliori di altre. Per esempio, il vantaggio della soluzione che utilizza il dizionario è che non dobbiamo sapere prima quali lettere appaiono nella stringa è dobbiamo solo fare spazio per le lettere che effettivamente appaiono.

Ecco un esempio del codice:

```
word = 'brontosaurus'
d = dict()
for c in word:
       if c not in d:
```

```
            d[c] = 1
      else:
            d[c] = d[c] + 1
print d
```

Stiamo effettivamente calcolando un istogramma, che è un termine statistico per indicare un insieme di contatori (o di frequenze).

Il ciclo For passa attraverso tutta la stringa. Ad ogni iterazione se il carattere non è nel dizionario, creiamo un nuovo elemento con chiave c e valore iniziale 1 (dato che abbiamo visto questa lettera per la prima volta). Se è già nel dizionario incrementiamo d[c].

Ecco l'output del programma:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

L'istogramma indica che la lettera a e la lettera b appaiono una volta sola; la lettera o appare due volte e così via.

I dizionari hanno un metodo chiamato get che utilizza una chiave ed un valore di default. Se la chiave appare nel dizionario, get restituisce il corrispondente valore; altrimenti restituisce il valore predefinito. Per esempio:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print counts.get('jan', 0)
100
>>> print counts.get('tim', 0)
0
```

Possiamo usare get per scrivere il ciclo che produce l'istogramma in modo più conciso.

Dato che il metodo get si prende cura automaticamente del caso in cui una chiave non si trova nel dizionario, possiamo condensare quattro linee in una ed eliminare l'istruzione if.

```
word = 'brontosaurus'
d = dict()
for c in word:
       d[c] = d.get(c,0) + 1
print d
```

L'uso del metodo get per semplificare questo ciclo di conteggio, risulta essere un vero e proprio 'idioma', una soluzione molto utilizzata nel Python e lo useremo molte volte lungo tutto il libro. Per cui risulta utile confrontare il ciclo che usa l'istruzione if e l'operatore in con il ciclo che usa il metodo get. Fanno esattamente la stessa cosa, ma uno è molto più conciso dell'altro.

9.2 Dizionari e files

Uno degli usi più comuni di un dizionario è quello di usarlo per contare le occorrenze di una parola in un file di testo. Partiamo con una file molto semplice, con parole prese dal testo di 'Romeo e Giulietta' all'indirizzo

http://shakespeare.mit.edu/Tragedy/romeoandjuliet/romeo_juliet.2.2.html

Per i primi esempi, useremo una versione semplificata e accorciata del testo senza segni di punteggiatura. In seguito lavoreremo con il testo con la punteggiatura inclusa.

But soft what light through yonder window breaks

It is the east and Juliet is the sun

Arise fair sun and kill the envious moon

Who is already sick and pale with grief

Scriveremo un programma per leggere attraverso le righe del testo, spezzeremo ogni riga di testo in una lista di parole e quindi con un ciclo di iterazioni attraverso ognuna delle parole, conteremo ogni parola utilizzando un dizionario.

Utilizzeremo due cicli For. Il ciclo esterno legge le righe del file e il ciclo interno fa delle iterazioni attraverso ognuna delle parole di quella riga particolare. Questo è un un esempio di ciclo nidificato, perché uno dei cicli è il ciclo esterno, l'altro ciclo è interno.

Dato che il ciclo interno esegue tutte le sue iterazioni, ogni volta che il ciclo esterno fa una singola iterazione, possiamo pensare al ciclo interno come a un ciclo "più veloce", che si ripete più velocemente di quanto faccia il ciclo esterno.

La combinazione dei due cicli nidificati, assicura che conteremo ogni parola su ogni riga del file di input.

```
fname = raw_input('Enter the file name: ')
try:
	fhand = open(fname)
except:
	print 'File cannot be opened:', fname
	exit()
counts = dict()
for line in fhand:
	words = line.split()
	for word in words:
		if word not in counts:
			counts[word] = 1
		else:
			counts[word] += 1
print counts
```

Quando eseguiamo il programma, vediamo una prima versione disordinata dei conteggi. (il file romeo.txt è disponibile all'indirizzo)

http://www.py4inf.com/code/romeo.txt

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

È disagevole guardare attraverso il dizionario per trovare le parole più comuni e i loro conteggi, per cui abbiamo bisogno di aggiungere ancora un po' di codice per produrre un output che risulti più utile.

9.3 Cicli di iterazione e dizionari

Se si usa un dizionario come sequenza all'interno di un ciclo for, esso attraverserà le chiavi del dizionario. Questo ciclo stampa ogni chiave e il corrispondente valore:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
       print key, counts[key]
```

Ecco come sarà l'output:

```
jan 100
chuck 1
annie 42
```

Notare come, ancora una volta, le chiavi non seguono un ordine di nessun tipo.

Possiamo usare questo schema , per inplementare i vari modi di fare una iterazione, che abbiamo descritto precedentemente. Per esempio se volessimo trovare tutte gli elementi in un dizionario con un valore maggiore di 10, potremmo scrivere il seguente codice:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
       if counts[key] > 10 :
              print key, counts[key]
```

Il ciclo For fa delle iterazioni attraverso le chiavi del dizionario, per cui dobbiamo usare l'operatore index per recuperare il corrispondente valore per ogni chiave. Ecco l'output:

```
jan 100
annie 42
```

Vediamo solo gli ellementi con un valore maggiore di 10.

Volendo stampare le chiavi in ordine alfabetico, bisogna prima fare una lista che contenga le chiavi del dizionario, usando il metodo keys, metodo esposto dagli oggetti di tipo dictionary, quindi ordinare la lista e attraversare con un ciclo l'intera lista ordinata, cercando ogni chiave e stampando la coppia chiave-valore in ordine, così come segue:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = counts.keys()
print lst
lst.sort()
for key in lst:
       print key, counts[key]
```

Ecco come apparirà l'output:

```
['jan', 'chuck', 'annie']
```

```
annie 42
chuck 1
jan 100
```

Si vede la lista delle parole, senza nessun tipo di ordinazione, che si ottengono con il metodo get.

Quindi vediamo le coppie chiave-valore ordinate grazie al ciclo for.

9.4 Tecniche avanzate di parsing del testo

Nell'esempio precedente, usando il file romeo.txt, abbiamo semplificato il file al massimo, rimuovendo ogni punteggiatura a mano. Il testo reale ha molta punteggiatura così come si può vedere:

But, soft! what light through yonder window breaks?

It is the east, and Juliet is the sun.

Arise, fair sun, and kill the envious moon,

Who is already sick and pale with grief,

Dato che la funzione split cerca degli spazi e tratta le parole come elementi separati da spazi, dovremmo trattare le parole "soft!" e "soft" come parole diverse e creare un elemento diverso per ogni parola all'interno del dizionario.

Considerando anche che ci sono delle parole con lettere in maiuscolo, dovremo trattare "who" e "Who" come parole diverse e avere conteggi diversi.

Possiamo risolvere entrambi i problemi usando i metodi dell'oggetto stringa, lower, punctuation, and translate. il metodo translate è il più sottile fra questi.

Ecco la documentazione:

```
string.translate(s, table[, deletechars])
```

Cancella tutti i caratteri dalla stringa s che si trovano in deletechars (se presente), e quindi traduci i caratteri usando table, che deve essere una stringa di 256 caratteri che dia la traduzione per ogni carattere, indicizzato dal suo ordinale. se il valore di table è None, verrà effettuata solo la cancellazione dei caratteri.

Non specificheremo il valore di table, ma useremo il parametro deletechars per cancellare tutta la punteggiatura. Useremo addirittura Python per farci dire quali cartteri sono considerati "punteggiatura":

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

Faremo le seguenti modifiche al nostro programma:

```
import string                                   # New Code
fname = raw_input('Enter the file name: ')
try:
       fhand = open(fname)
except:
       print 'File cannot be opened:', fname
       exit()
counts = dict()
for line in fhand:
       line = line.translate(None, string.punctuation) # New Code
       line = line.lower()                              # New Code
       words = line.split()
       for word in words:
```

```
            if word not in counts:
                  counts[word] = 1
            else:
                  counts[word] += 1
print counts
```

Usiamo translate per rimuovere tutta la punteggiatura e lower per scrivere l'intera riga in minuscolo. Per il resto, il programma è identico. Da notare, che per Python 2.5 e precedenti, translate non accetta None come primo parametro, per cui si usa questo codice per l'istruzione translate:

```
print a.translate(string.maketrans(' ',' '), string.punctuation
```

Parte dell'imparare "l'arte del Python" o del "pensare in Python" consiste nel comprendere che spesso, Python offre delle soluzioni pre-definite per molti problemi tipici dell'analisi dei dati. Nel tempo vedrete abbastanza codice d'esempio e leggerete abbastanza documentazione da riuscire a sapere dove guardare per capire se qualcuno ha già scritto qualcosa di utile per il vostro scopo.

Quello che segue è una versione abbreviata dell'output:

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Guardare questo output è ancora molto disagevole e possiamo usare Python per darci esattamente quello che stiamo cercando ma per farlo abbiamo bisogno di studiare le tuple. Riprenderemo questo esempio quando le studieremo.

9.5 Debugging

Se si utilizzano dei set di dati molto grandi, può diventare disagevole fare il debugging stampando e controllando i dati a mano. Ecco alcuni suggerimenti per fare il debugging di grandi quantità di dati:

Diminuire i dati di input: se possibile, ridurre la quantità dell'insieme di dati. Per esempio, se il programma legge da un file di testo, partire con solo le prime 10 linee, o con la porzione di testo più piccola che si possa utilizzare. Si può editare direttamente il file o ancora meglio modificare il programma in modo che legga solo le prime n righe.

Se c'è un errore si puo' ridurre n al valore più piccolo che produce l'errore, e quindi incrementarlo gradualmente, lungo tutto il processo di ricerca e correzione degli errori.

Controllare i tipi e utilizzare dei riassunti: invece di stampare e controllare l'intero set di dati, puoò essere utile stampare un riassunto dei dati: per esempio, il numero di elementi in un dizionario o il totale di una lista di numeri.

Una causa comune di errore al momento dell'esecuzione è una valore che non è del tipo giusto. Per il debugging di questo tipo di errori, spesso è sufficiente stampare il tipo di appartenenza di quel determinato valore.

Scrivere codice di autocontrollo: A volte può essere utile scrivere del codice per controllare gli errori automaticamente. Per esempio, se si sta calcolando una media di una lista di numeri, può bastare controllare che il risultato non sia maggiore del più grande elemento della lista o minore del più piccolo. Questo viene chiamato 'controllo coerenza', dato che serve a eliminare risultati che sarebbero illogici.

Un altro tipo di controllo confronta i risultati di due diversi calcoli per vedere se sono consistenti. Questo viene chiamato un "controllo di consistenza".

Formattare l'output in maniera adeguata: Il formattare l'output delle operazioni

di debugging, può contribuire a rendere più facile la ricerca degli errori.

Il tempo che si spende nel concepire l'impalcatura del programma in maniera adeguata, è tempo risparmiato nelle operazioni di debugging.

9.6 Glossario

Dizionario (dictionary): è un insieme di relazioni fra un insieme di chiavi e i corrispondenti valori.

Hashtable: è l'algoritmo usato per implementare i dizionari.

Funzione Hash: una funzione usata da una HashTable per dare la posizione di una chiave all'interno dell'indica.

Istogramma (histogram): un insieme di contatori.

Implementazione: un modo di effettuare una elaborazione.

Elemento (item): un altro nome per una coppia chiave-valore.

Chiave (key): un oggetto all'interno di un dizionario come prima parte di una coppia chiave-valore.

Coppia chiave-valore (key-value): la rappresentazione di una relazione fra una chiave ed un valore.

Ricerca (lookup): un'operazione su un dizionario che a partire da una chiave trova il corrispondente valore.

Cicli nidificati (nested loops): Quando ci sono uno o più cicli, all'interno di un altro ciclo. Il ciclo interno, effettua tutte le iterazioni, nel tempo impiegato dal ciclo esterno a compierne una sola.

Valore (value): un oggetto all'interno di un dizionario come prima parte di una coppia chiave-valore. Questo è un significato più specifico di quello precedentemente visto per la parola valore.

9.7 Esercizi

Esercizio 9.2

Scrivi un programma che categorizzi ogni messaggio in base al giorno della settimana in cui è stato spedito. Per fare questo cerca le righe che iniziano con "From", quindi cerca la terza parola e tieni un conteggio in corso, per ognuno dei giorni della settimana. Alla fine del programma stampa i contenuti del tuo dizionario (l'ordine non ha importanza).

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

Sample Execution:
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

Esercizio 9.3

Scrivi un programma che legga un mail log, e costruisci un istogramma usando un dizionario che conti quanti messaggi sono arrivati da ogni indirizzo email e stampa il

dizionario.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

Esercizio 9.4

Aggiungi del codice al programma precedente per capire chi abbia ricevuto il maggior numero di messaggi.

Dopo che tutti i dati sono stati letti e il dizionario è stato creato, guarda attraverso il dizionario usando un ciclo maximum (guarda la sezione 5.7.2) per cercare chi ha il maggior numero di messaggi e stampa il numero di messaggi di quella persona.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5
Enter a file name: mbox.txt
zqian@umich.edu 195
```

Esercizio 9.5

Questo programma registra il nome di dominio (invece dell'indirizzo) dove il messaggio è stato spedito invece del nome del mittente (ad esempio, l'intero indirizzo mail). Alla fine del programma, stampa il contenuto del dizionario.

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
```

'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}