Pensare da informatico

Imparare con Python

Pensare da informatico

Imparare con Python

Allen Downey
Jeffrey Elkner
Chris Meyers

Green Tea Press
Wellesley, Massachusetts

Elaborato da Shannon Turlington e Lisa Cutler. Copertina di Rebecca Gimenez.

Edizioni:

Aprile 2002: Prima edizione.

Aprile 2003: Traduzione in lingua italiana (ver. 1.0c - luglio 2003).

Green Tea Press
1 Grove St.
P.O. Box 812901
Wellesley, MA 02482

La forma originale di questo libro è in LaTeX . La compilazione del sorgente LaTeX ha l'effetto di generare una rappresentazione di un testo indipendente dal dispositivo che può essere successivamente convertito in altri formati e stampato. Il sorgente LaTeX di questo libro può essere ottenuto da `http://www.thinkpython.com`

Informazioni di catalogo della versione originale in lingua inglese (fornita da Quality Books, Inc.)

Downey, Allen
How to think like a computer scientist : learning with Python / Allen Downey, Jeffrey Elkner, Chris Meyers. – 1st ed.
p. cm.
Includes index.
ISBN 0-9716775-0-6
LCCN 2002100618

1. Python (Computer program language) I. Elkner, Jeffrey. II. Meyers, Chris. III. Title

QA76.73.P98D69 2002 005.13'3
QBI02-200031

Introduzione

Di David Beazley

In qualità di educatore, ricercatore e autore di libri, sono lieto di assistere alla conclusione della stesura di questo testo. Python è un linguaggio di programmazione divertente e semplice da usare, la cui popolarità è andata via via crescendo nel corso degli ultimi anni. Python è stato sviluppato più di dieci anni fa da Guido van Rossum che ne ha derivato semplicità di sintassi e facilità d'uso in gran parte da ABC, un linguaggio dedicato all'insegnamento sviluppato negli anni '80. Oltre che per questo specifico contesto, Python è stato creato per risolvere problemi reali, dimostrando di possedere un'ampia varietà di caratteristiche tipiche di linguaggi di programmazione quali C++, Java, Modula-3 e Scheme. Questo giustifica una delle sue più rimarchevoli caratteristiche: l'ampio consenso nell'ambito degli sviluppatori professionisti di software, in ambiente scientifico e di ricerca, tra i creativi e gli educatori.

Nonostante l'interesse riscosso da Python in ambienti così disparati, potresti ancora chiederti "Perché Python?" o "Perché insegnare la programmazione con Python?". Rispondere a queste domande non è cosa semplice, specialmente quando l'interesse generale è rivolto ad alternative più masochistiche quali C++ e Java. Penso comunque che la risposta più diretta sia che la programmazione in Python è semplice, divertente e più produttiva.

Quando tengo corsi di informatica, il mio intento è quello di spiegare concetti importanti interessando ed intrattenendo nel contempo gli studenti. Sfortunatamente nei corsi introduttivi c'è la tendenza a focalizzare troppo l'attenzione sull'astrazione matematica e nel caso degli studenti a sentirsi frustrati a causa di fastidiosi problemi legati a dettagli di basso livello della sintassi, della compilazione e dall'imposizione di regole poco intuitive. Sebbene questa astrazione e questo formalismo siano importanti per il progettista di software professionale e per gli studenti che hanno intenzione di proseguire i loro studi di informatica, questo approccio in un corso introduttivo porta solitamente a rendere l'informatica noiosa. Quando tengo un corso non voglio avere davanti una classe di studenti annoiati: preferirei piuttosto vederli impegnati a risolvere problemi interessanti esplorando idee diverse, approcci non convenzionali, infrangendo le regole e imparando dai loro stessi errori.

Inoltre non voglio sprecare mezzo semestre a risolvere oscuri problemi di sintassi, cercando di capire messaggi del compilatore generalmente incomprensibili o di far fronte al centinaio di modi in cui un programma può generare un "general protection fault".

Una delle ragioni per cui mi piace Python è che esso permette un ottimo equilibrio tra l'aspetto pratico e quello concettuale. Dato che Python è interpretato, gli studenti possono fare qualcosa quasi subito senza perdersi in problemi di compilazione e link. Inoltre Python è fornito di un'ampia libreria di moduli che possono essere usati in ogni sorta di contesto, dalla programmazione web alla grafica. Questo aspetto pratico è un ottimo sistema per impegnare gli studenti e permette loro di portare a termine progetti non banali. Python può anche servire come eccellente punto di partenza per introdurre importanti concetti di informatica: dato che supporta procedure e classi, possono essere gradualmente introdotti argomenti quali l'astrazione procedurale, le strutture di dati e la programmazione ad oggetti, tutti solitamente relegati a corsi avanzati di Java o C++. Python prende a prestito un certo numero di caratteristiche da linguaggi di programmazione funzionali e può essere quindi usato per introdurre concetti che sarebbero normalmente trattati in dettaglio in corsi di Scheme o di Lisp.

Leggendo la prefazione di Jeffrey sono rimasto colpito da un suo commento: Python gli ha permesso di ottenere "un livello generale di successo più elevato ed un minore livello di frustrazione", e gli è stato possibile muoversi "con maggiore velocità e con risultati migliori". Questi commenti si riferiscono al suo corso introduttivo: io uso Python per queste stesse ragioni in corsi di informatica avanzata all'Università di Chicago. In questi corsi sono costantemente messo di fronte alla difficoltà di dover coprire molti argomenti complessi in un periodo di appena nove settimane. Sebbene sia certamente possibile per me infliggere un bel po' di sofferenza usando un linguaggio come il C++, ho spesso trovato che questo approccio è controproducente, specialmente nel caso di corsi riguardanti la semplice programmazione. Ritengo che usare Python mi permetta di focalizzare meglio l'attenzione sull'argomento reale della lezione, consentendo nel contempo agli studenti di completare progetti concreti.

Sebbene Python sia un linguaggio ancora giovane ed in continua evoluzione, credo che esso abbia un futuro nel campo dell'insegnamento. Questo libro è un passo importante in questa direzione.

David Beazley, autore di *Python Essential Reference*
Università di Chicago

Prefazione

Di Jeff Elkner

Questo libro deve la sua esistenza alla collaborazione resa possibile da Internet e dal movimento free software. I suoi tre autori, un professore universitario, un docente di scuola superiore ed un programmatore professionista, non si sono ancora incontrati di persona, ma ciononostante sono riusciti a lavorare insieme a stretto contatto, aiutati da molte persone che hanno donato il proprio tempo e le loro energie per rendere questo libro migliore.

Noi pensiamo che questo libro rappresenti la testimonianza dei benefici e delle future possibilità di questo tipo di collaborazione, la cui struttura è stata creata da Richard Stallman e dalla Free Software Foundation.

Come e perché sono arrivato ad usare Python

Nel 1999 per la prima volta venne usato il linguaggio C++ per l'esame di informatica del College Board's Advanced Placement (AP). Come in molte scuole secondarie della nazione, la decisione di cambiare linguaggio ebbe un impatto diretto sul curriculum del corso di informatica alla Yorktown High School di Arlington, Virginia, dove insegno. Fino a quel momento il Pascal era stato il linguaggio di insegnamento sia per il primo anno che per i corsi AP. Per continuare con la tradizione di insegnare ai nostri studenti uno stesso linguaggio per due anni, decidemmo di passare al C++ con gli studenti del primo anno nel '97/'98 così da metterli al passo con il cambio nel corso AP dell'anno successivo.

Due anni più tardi io ero convinto che il C++ fosse una scelta non adeguata per introdurre gli studenti all'informatica: mentre da un lato il C++ è certamente un linguaggio molto potente, esso si dimostra tuttavia essere estremamente difficile da insegnare ed imparare. Mi trovavo costantemente alle prese con la difficile sintassi del C++ e stavo inoltre inutilmente perdendo molti dei miei studenti. Convinto che ci dovesse essere un linguaggio migliore per il nostro primo anno iniziai a cercare un'alternativa al C++.

Avevo bisogno di un linguaggio che potesse girare tanto sulle macchine Linux del nostro laboratorio quanto sui sistemi Windows e Macintosh che la maggior parte degli studenti aveva a casa. Lo volevo open-source, così che potesse essere usato dagli studenti indipendentemente dalle loro possibilità economiche. Cercavo un linguaggio che fosse usato da programmatori professionisti e che avesse un'attiva comunità di sviluppatori. Doveva supportare tanto la programmazione procedurale che quella orientata agli oggetti. Cosa più importante, doveva essere facile da insegnare ed imparare. Dopo avere vagliato le possibili alternative con questi obiettivi in mente, Python sembrò il migliore candidato.

Chiesi ad uno tra gli studenti più dotati di Yorktown, Matt Ahrens, di provare Python. In due mesi egli non solo imparò il linguaggio ma scrisse un'applicazione, chiamata pyTicket, che dava la possibilità al nostro staff di stendere report concernenti problemi tecnici via Web.

Sapevo che Matt non avrebbe potuto realizzare un'applicazione di tale portata in un tempo così breve in C++, ed il suo successo, insieme al suo giudizio positivo sul linguaggio, suggerirono che Python era la soluzione che andavo cercando.

Trovare un libro di testo

Avendo deciso di usare Python nel corso introduttivo in entrambi i miei corsi di informatica l'anno successivo, la mancanza di un libro di testo si fece il problema più pressante.

Il materiale disponibile gratuitamente venne in mio aiuto. In precedenza, in quello stesso anno, Richard Stallman mi aveva fatto conoscere Allen Downey. Entrambi avevamo scritto a Richard esprimendo il nostro interesse nello sviluppare dei testi educativi gratuiti e Allen aveva già scritto un testo di informatica per il primo anno, *How to Think Like a Computer Scientist*. Quando lessi quel libro seppi immediatamente che volevo usarlo nelle mie lezioni. Era il testo di informatica più chiaro ed utile che avessi visto: il libro enfatizzava il processo di pensiero coinvolto nella programmazione piuttosto che le caratteristiche di un particolare linguaggio. Il solo fatto di leggerlo mi rese un insegnante migliore.

How to Think Like a Computer Scientist non solo era un libro eccellente, ma aveva la licenza pubblica GNU: questo significava che esso poteva essere usato liberamente e modificato per far fronte alle esigenze dei suoi utilizzatori. Deciso a usare Python, dovevo tradurre in questo linguaggio la versione originale basata su Java del testo di Allen. Mentre non sarei mai stato capace di scrivere un libro basandomi sulle mie sole forze, il fatto di avere il libro di Allen da usare come base mi rese possibile farlo, dimostrando nel contempo che il modello di sviluppo cooperativo usato così bene nel software poteva funzionare anche in ambito educativo.

Lavorare su questo libro negli ultimi due anni è stata una ricompensa sia per me che per i miei studenti, e proprio i miei studenti hanno giocato un ruolo importante nel processo. Dato che potevo modificare il testo non appena qualcuno trovava un errore o riteneva troppo difficile un passaggio, io li incoraggiai a cercare errori dando loro un punto aggiuntivo ogniqualvolta una loro osservazione comportava il cambiamento del testo. Questo aveva il doppio scopo di incoraggiarli a leggere il testo più attentamente e di passare il libro al vaglio dei suoi critici più severi: gli studenti impegnati ad imparare l'informatica.

Per la seconda parte del libro riguardante la programmazione ad oggetti, sapevo che sarebbe stato necessario trovare qualcuno con un'esperienza di programmazione reale più solida della mia. Il libro rimase incompiuto per buona parte dell'anno, finché la comunità open source ancora una volta fornì i mezzi per il suo completamento.

Ricevetti un'email da Chris Meyers che esprimeva interesse per il libro. Chris è un programmatore professionista che aveva iniziato a tenere un corso di programmazione con Python l'anno precedente presso il Lane Community College di Eugene, Oregon. La prospettiva di tenere il corso aveva portato il libro alla conoscenza di Chris, così che quest'ultimo cominciò ad aiutarci immediatamente. Prima della fine dell'anno aveva creato un progetto parallelo chiamato *Python for Fun* sul sito `http://www.ibiblio.org/obp` e stava lavorando con alcuni dei miei studenti più avanzati guidandoli dove io non avrei potuto portarli.

Introduzione alla programmazione con Python

Il processo di traduzione e uso di *How to Think Like a Computer Scientist* nei due anni scorsi ha confermato che Python è adatto all'insegnamento agli studenti del primo anno. Python semplifica enormemente gli esempi di programmazione e rende più semplici le idee importanti.

Il primo esempio illustra bene il punto. È il tradizionale programma "hello, world", la cui versione C++ nel libro originale è la seguente:

```
#include <iostream.h>

void main()
{
  cout << "Hello, World!" << endl;
}
```

Nella versione Python diventa:

```
print "Hello, World!"
```

I vantaggi di Python saltano subito all'occhio anche in questo esempio banale. Il corso di informatica I a Yorktown non necessita di prerequisiti, così molti studenti vedendo questo esempio stanno in realtà guardando il loro primo programma. Qualcuno di loro è sicuramente un po' nervoso, avendo saputo che la programmazione è difficile da imparare. La versione in C++ mi ha sempre costretto a scegliere tra due opzioni ugualmente insoddisfacenti: o spiegare le istruzioni `#include`, `void main()`, { e }, rischiando di intimidire e mettere in confusione qualcuno degli studenti già dall'inizio, o dire loro "Non preoccupatevi di questa roba per adesso; ne parleremo più avanti" e rischiare di ottenere lo stesso risultato. Gli obiettivi a questo punto del corso sono quelli di introdurre gli studenti all'idea di istruzione di programma e di portarli a scrivere il loro primo programma, così da introdurli nell'ambiente della programmazione. Python ha esattamente ciò che è necessario per fare questo e niente di più.

Confrontare il testo esplicativo del programma in ognuna delle due versioni del libro illustra ulteriormente ciò che questo significa per lo studente alle prime armi: ci sono tredici paragrafi nella spiegazione di "Hello, world!" nella versione C++ e solo due in quella Python. Da notare che gli undici paragrafi aggiuntivi non trattano delle “grandi idee” della programmazione, ma riguardano i particolari connessi alla sintassi del C++. Ho visto questo accadere lungo tutto il corso del libro, così che interi paragrafi semplicemente sono scomparsi dalla versione Python del testo perché la sintassi del linguaggio, molto più chiara, li ha resi inutili.

L’uso di un linguaggio di altissimo livello come Python permette all’insegnante di posporre la trattazione di dettagli di basso livello sino al momento in cui gli studenti non sono in possesso delle basi che permettono loro di comprenderli appieno. Ciò dà la possibilità di procedere con ordine. Uno degli esempi migliori è il modo in cui Python tratta le variabili. In C++ una variabile è un nome che identifica un posto che contiene qualcosa: le variabili devono essere dichiarate anticipatamente perché la grandezza del posto cui si riferiscono deve essere predeterminata tanto che l’idea di una variabile è legata all’hardware della macchina. Il concetto potente e fondamentale di variabile è già sufficientemente difficile per studenti alle prime armi (tanto in informatica che in algebra): byte e indirizzi non aiutano certo a comprendere l’argomento. In Python una variabile è un nome che fa riferimento ad una cosa. Questo è un concetto decisamente più intuitivo per gli studenti e molto più vicino a ciò che essi hanno imparato in matematica. Ho dovuto affrontare difficoltà molto minori nell’insegnare le variabili quest’anno che in passato e ho dovuto trascorrere meno tempo aiutando gli studenti a destreggiarsi con esse.

Un altro esempio di come Python aiuti tanto nell’insegnamento quanto nell’apprendimento della programmazione è la sua sintassi per le funzioni. I miei studenti hanno sempre avuto difficoltà a capire le funzioni: il problema verte sulla differenza tra la definizione di una funzione e la sua chiamata e la relativa distinzione tra un parametro ed un argomento. Python viene in aiuto con una sintassi che non manca di eleganza. La definizione di una funzione inizia con `def`, così dico ai miei studenti: “Quando definite una funzione iniziate con `def`, seguito dal nome della funzione; quando volete chiamare la funzione basta inserire il suo nome.” I parametri vanno con le definizioni, gli argomenti con le chiamate. Non ci sono tipi di ritorno, tipi del parametro, o riferimenti, così posso insegnare le funzioni in minor tempo e con una migliore comprensione.

L’uso di Python ha migliorato l’efficacia del nostro corso di informatica. Ottengo dai miei studenti un livello generale di successo più elevato ed un minore livello di frustrazione, rispetto al biennio in cui ho insegnato il C++. Mi muovo con maggior velocità e con migliori risultati. Un maggior numero di studenti terminano il corso con la capacità di creare programmi significativi e con un’attitudine positiva verso l’esperienza della programmazione.

Costruire una comunità

Ho ricevuto email da tutto il mondo da gente che usa questo libro per imparare o insegnare la programmazione. Una comunità di utilizzatori ha iniziato ad emergere, e molte persone hanno contribuito al progetto spedendo materiale al sito `http://www.thinkpython.com`.

Con la pubblicazione del libro in forma stampata mi aspetto che la comunità di utilizzatori si espanda. La nascita di questa comunità e la possibilità che essa suggerisce riguardo collaborazioni tra insegnanti sono state le cose che più mi hanno coinvolto in questo progetto. Lavorando insieme possiamo migliorare la qualità del materiale disponibile e risparmiare tempo prezioso. Ti invito a unirti a noi e attendo di ricevere tue notizie: scrivi agli autori all'indirizzo `feedback@thinkpython.com`.

Jeffrey Elkner
Yorktown High School
Arlington, Virginia

Lista dei collaboratori

Questo libro è nato grazie ad una collaborazione che non sarebbe stata possibile senza la GNU Free Documentation License. Vorremmo ringraziare la Free Software Foundation per aver sviluppato questa licenza e per avercela resa disponibile.

Vorremmo anche ringraziare il centinaio di lettori che ci hanno spedito suggerimenti e correzioni nel corso degli ultimi due anni. Nello spirito del software libero abbiamo deciso di esprimere la nostra gratitudine aggiungendo la lista dei collaboratori. Sfortunatamente la lista non è completa, ma stiamo facendo del nostro meglio per tenerla aggiornata.

Se avrai modo di scorrere lungo la lista riconoscerai che ognuna di queste persone ha risparmiato a te e agli altri lettori la confusione derivante da errori tecnici o da spiegazioni non troppo chiare.

Anche se sembra impossibile dopo così tante correzioni, ci possono essere ancora degli errori in questo libro. Se per caso dovessi trovarne uno, speriamo tu possa spendere un minuto per farcelo sapere. L'indirizzo email al quale comunicarcelo è `feedback@thinkpython.com`. Se faremo qualche cambiamento a seguito del tuo suggerimento anche tu sarai inserito nella lista dei collaboratori, sempre che tu non chieda altrimenti. Grazie!

- Lloyd Hugh Allen, per una correzione nella sezione 8.4.
- Yvon Boulianne, per una correzione di un errore di semantica al capitolo 5.
- Fred Bremmer, per una correzione alla sezione 2.1.
- Jonah Cohen, per lo script Perl di conversione del codice LaTeX di questo libro in HTML.
- Michael Conlon, per una correzione grammaticale nel capitolo 2, per il miglioramento dello stile nel capitolo 1 e per aver iniziato la discussione sugli aspetti tecnici degli interpreti.
- Benoit Girard, per la correzione di un errore nella sezione 5.6.
- Courtney Gleason e Katherine Smith, per aver scritto `horsebet.py`, usato in una versione precedente del libro come caso di studio. Il loro programma può essere trovato sul sito.

- Lee Harr, per aver sottoposto una serie di correzioni che sarebbe troppo lungo esporre qui. Dovrebbe essere citato come uno dei maggiori revisori del libro.
- James Kaylin è uno studente che ha usato il libro ed ha sottoposto numerose correzioni.
- David Kershaw, per aver reso funzionante del codice nella sezione 3.10.
- Eddie Lam, per aver spedito numerose correzioni ai primi tre capitoli, per aver sistemato il makefile così da creare un indice alla prima compilazione e per averci aiutato nella gestione delle versioni.
- Man-Yong Lee, per aver spedito una correzione al codice di esempio nella sezione 2.4.
- David Mayo, per una correzione grammaticale al capitolo 1.
- Chris McAloon, per le correzioni nelle sezioni 3.9 e 3.10.
- Matthew J. Moelter, per essere stato uno dei collaboratori al progetto, e per aver contribuito con numerose correzioni e commenti.
- Simon Dicon Montford, per aver fatto notare una mancata definizione di funzione e numerosi errori di battitura nel capitolo 3 e per aver aver trovato gli errori nella funzione `Incrementa` nel capitolo 13.
- John Ouzts, per aver corretto la definizione di "valore di ritorno" nel capitolo 3.
- Kevin Parks, per aver contribuito con validi commenti e suggerimenti su come migliorare la distribuzione del libro.
- David Pool, per la correzione di un errore di battitura al capitolo 1 e per averci spedito parole di incoraggiamento.
- Michael Schmitt, per una correzione nel capitolo sui file e le eccezioni.
- Robin Shaw, per aver trovato un errore nella sezione 13.1 dove una funzione veniva usata senza essere stata preventivamente definita.
- Paul Sleigh, per aver trovato un errore nel capitolo 7, ed un altro nello script Perl per la generazione dell'HTML.
- Craig T. Snydal, che sta usando il testo in un corso alla Drew University. Ha contribuito con numerosi suggerimenti e correzioni.
- Ian Thomas ed i suoi studenti che hanno usato il testo in un corso di programmazione. Sono stati i primi a controllare i capitoli nella seconda parte del libro, fornendo numerose correzioni ed utili suggerimenti.
- Keith Verheyden, per una correzione nel capitolo 3.
- Peter Winstanley, per una correzione nel capitolo 3.
- Chris Wrobel, per le correzioni al codice nel capitolo sui file e le eccezioni.

- Moshe Zadka, per il suo prezioso contributo al progetto. Oltre ad aver scritto la prima stesura del capitolo sui dizionari ha fornito una continua assistenza nelle fasi iniziali del libro.
- Christoph Zwerschke, per aver spedito numerose correzioni e suggerimenti, e per aver spiegato la differenza tra *gleich* e *selbe*.
- James Mayer, per la lista di correzioni di errori tipografici e di spelling.
- Hayden McAfee, per aver notato una potenziale incoerenza tra due esempi.
- Angel Arnal, fa parte di un gruppo internazionale di traduttori che sta lavorando sulla versione in lingua spagnola del libro. Ha anche riferito di una serie di errori nella versione inglese.
- Tauhidul Hoque e Lex Berezhny hanno creato le illustrazioni del capitolo 1 e migliorato molte delle altre illustrazioni.
- Dr. Michele Alzetta, per aver corretto un errore nel capitolo 8 e aver inviato una serie di utili commenti e suggerimenti concernenti Fibonacci e Old Maid.
- Andy Mitchell, per aver corretto un errore tipografico nel capitolo 1 ed un esempio non funzionante nel capitolo 2.
- Kalin Harvey, per aver suggerito un chiarimento nel capitolo 7 e aver corretto alcuni errori di battitura.
- Christopher P. Smith, per la correzione di numerosi errori di battitura e per l'aiuto nell'aggiornamento del libro alla versione 2.2 di Python.
- David Hutchins, per la correzione di un errore di battitura nella Prefazione.
- Gregor Lingl sta insegnando Python in una scuola superiore di Vienna e lavorando alla traduzione in tedesco. Ha corretto un paio di errori nel capitolo 5.
- Julie Peters, per la correzione di un errore di battitura nella prefazione.

Note sulla traduzione

Di Alessandro Pocaterra

Chi si trova a tradurre un testo da una lingua all'altra deve necessariamente fare delle scelte, dato che nel caso delle lingue naturali non è quasi mai possibile ottenere una perfetta corrispondenza tra testo originale e testo tradotto. Questo vale più che mai nel caso della traduzione di testi tecnici, soprattutto in campi così "giovani" come l'informatica: questo settore è nato pescando a destra e a manca termini dalla lingua inglese, e in buona parte questi sono traducibili in italiano solo in modo ridicolo (si veda il "baco" malamente ottenuto dall'originale "bug"), inadeguato o, quel che è peggio, inesatto. Partendo dal fatto che io sono un programmatore senior, il mio approccio è decisamente diverso da quello dello studente "moderno" che si appresta allo studio dell'informatica: solo dieci anni fa era praticamente impossibile trovare termini tecnici in informatica che non fossero rigorosamente in inglese e pertanto ho deciso di conservarli dove ho ritenuto fosse necessario (come nel caso di "bug", "debug", "parsing" per fare qualche esempio). In questa traduzione ho cercato di rispettare il più possibile il testo originale mantenendone il tono discorsivo e le frasi brevi tipiche della lingua inglese. Ho avuto il permesso degli autori a togliere (poche) frasi che avrebbero perso il loro significato perché basate su giochi di parole intraducibili e a rimaneggiare in qualche punto l'organizzazione del testo.

Una nota che invece riguarda la notazione numerica. Chiunque abbia mai preso in mano una calcolatrice si sarà accorto che la virgola dei decimali tanto cara alla nostra maestra delle elementari si è trasformata in un punto. Naturalmente questo cambio non è casuale: nei paesi anglosassoni l'uso di virgola e punto nei numeri è esattamente l'opposto di quello cui siamo abituati: se per noi ha senso scrivere 1.234.567,89 (magari con il punto delle migliaia in alto), in inglese questo numero viene scritto come 1,234,567.89. In informatica i separatori delle migliaia sono di solito trascurati e per la maggior parte dei linguaggi di programmazione considerati illegali: per il nostro computer lo stesso numero sarà quindi 1234567.89. Un po' di pratica e ci si fa l'abitudine. In relazione al codice presente nel testo, per non uscire dai margini del documento, sono state spezzate le righe che davano problemi con l'inserimento del carattere \ come fine riga. Siete quindi fin d'ora avvertiti che, ove trovaste quel carattere, in realtà la riga andrebbe scritta comprendendo anche quella successiva. In altri casi piuttosto evidenti è stato omesso il carattere \.

Ringraziamenti

Naturalmente ringrazio i tre autori del testo originale Allen Downey, Jeffrey Elkner e Chris Meyers, senza i quali questo libro non avrebbe mai visto la luce. Devo ringraziare per l'aiuto mia moglie Sara che si è volonterosamente prestata alla rilettura e correzione del libro. Ringrazio in modo particolare Ferdinando Ferranti che si è prodigato nella revisione, ma soprattutto nel rivedere il codice LaTeX in ogni sua parte, aiutandomi a suddividere il documento così come nell'originale, correggendo gli errori di compilazione che il codice restituiva e realizzando così anche una versione HTML funzionante. Oltre a questo ha anche modificato l'impaginazione ed il Makefile ottenendo così una versione del documento la cui stampa è più funzionale rispetto all'originale, pensato per formati di carta differenti. Ringrazio anche Nicholas Wieland, "Pang" e Nicola La Rosa per il loro aiuto insostituibile in fase di revisione. Ringrazio tutti quelli, Dario Cavedon e Giovanni Panozzo in testa, che mi hanno fatto scoprire il mondo Linux, il Free Software e la Free Documentation. Un ringraziamento particolare a tutti quelli che si sono rimboccati le maniche ed hanno dato vita a quell'incredibile strumento che è LaTeX .

La traduzione di questo libro è stata un passatempo ed un divertimento. Dato che sicuramente qualcosa non gira ancora come dovrebbe, vi chiedo di mandarmi i vostri commenti a riguardo all'indirizzo `a.pocaterra@libero.it`. In caso di refusi o imprecisioni ricordate di citare sempre la pagina e la versione di questo documento (versione 1.0b).

Nelle versioni successive si cercherà per quanto possibile di tenere il passo con la bibliografia: qualsiasi indicazione al riguardo sarà sempre bene accetta.

Buona fortuna!
Alessandro Pocaterra

Indice

Capitolo 1

Imparare a programmare

L'obiettivo di questo libro è insegnarti a pensare da informatico. Questo modo di pensare combina alcune delle migliori caratteristiche della matematica, dell'ingegneria e delle scienze naturali. Come i matematici, gli informatici usano linguaggi formali per denotare idee (nella fattispecie elaborazioni). Come gli ingegneri progettano cose, assemblano componenti in sistemi e cercano compromessi tra le varie alternative. Come gli scienziati osservano il comportamento di sistemi complessi, formulano ipotesi e verificano previsioni.

La più importante capacità di un informatico è quella di **risolvere problemi**. Risolvere problemi significa avere l'abilità di schematizzarli, pensare creativamente alle possibili soluzioni ed esprimerle in modo chiaro ed accurato. Da ciò emerge che il processo di imparare a programmare è un'eccellente opportunità di mettere in pratica l'abilità di risolvere problemi.

Da una parte ti sarà insegnato a programmare, già di per sé un'utile capacità. Dall'altra userai la programmazione come un mezzo rivolto ad un fine. Mentre procederemo quel fine ti diverrà più chiaro.

1.1 Il linguaggio di programmazione Python

Il linguaggio di programmazione che imparerai è il Python. Python è un esempio di **linguaggio di alto livello**; altri linguaggi di alto livello di cui puoi aver sentito parlare sono il C, il C++, il Perl ed il Java.

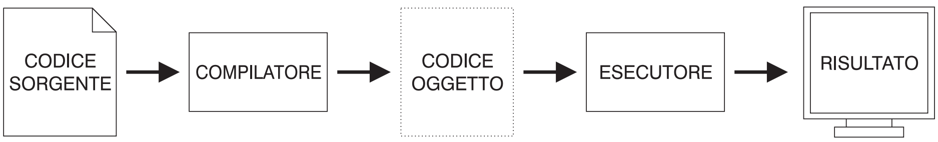
Come puoi immaginare sentendo la definizione "linguaggio di alto livello" esistono anche **linguaggi di basso livello**, talvolta chiamati "linguaggi macchina" o "linguaggi assembly". In modo non del tutto corretto si può affermare che i computer possono eseguire soltanto programmi scritti in linguaggi di basso livello: i programmi scritti in un linguaggio di alto livello devono essere elaborati prima di poter essere eseguiti. Questo processo di elaborazione impiega del tempo e rappresenta un piccolo svantaggio dei linguaggi di alto livello.

I vantaggi sono d'altra parte enormi. In primo luogo è molto più facile programmare in un linguaggio ad alto livello: questi tipi di programmi sono più veloci da scrivere, più corti e facilmente leggibili, ed è più probabile che siano corretti. In secondo luogo i linguaggi di alto livello sono portabili: **portabilità** significa che essi possono essere eseguiti su tipi di computer diversi con poche o addirittura nessuna modifica. I programmi scritti in linguaggi di basso livello possono essere eseguiti solo su un tipo di computer e devono essere riscritti per essere trasportati su un altro sistema.

Questi vantaggi sono così evidenti che quasi tutti i programmi sono scritti in linguaggi di alto livello, lasciando spazio ai linguaggi di basso livello solo in poche applicazioni specializzate.

I programmi di alto livello vengono trasformati in programmi di basso livello eseguibili dal computer tramite due tipi di elaborazione: l'**interpretazione** e la **compilazione**. Un interprete legge il programma di alto livello e lo esegue, trasformando ogni riga di istruzioni in un'azione. L'interprete elabora il programma un po' alla volta, alternando la lettura delle istruzioni all'esecuzione dei comandi che le istruzioni descrivono:

Un compilatore legge il programma di alto livello e lo traduce completamente in basso livello, prima che il programma possa essere eseguito. In questo caso il programma di alto livello viene chiamato **codice sorgente**, ed il programma tradotto **codice oggetto** o **eseguibile**. Dopo che un programma è stato compilato può essere eseguito ripetutamente senza che si rendano necessarie ulteriori compilazioni finché non ne viene modificato il codice.



Python è considerato un linguaggio interpretato perché i programmi Python sono eseguiti da un interprete. Ci sono due modi di usare l'interprete: a linea di comando o in modo script. In modo "linea di comando" si scrivono i programmi Python una riga alla volta: dopo avere scritto una riga di codice alla pressione di Invio (o Enter, a seconda della tastiera) l'interprete la analizza subito ed elabora immediatamente il risultato, eventualmente stampandolo a video:

```
$ python
Python 1.5.2 (#1, Feb 1 2000, 16:32:16)
Copyright 1991-1995 Stichting Mathematish Centrum, Amsterdam
>>> print 1 + 1
2
```

La prima linea di questo esempio è il comando che fa partire l'interprete Python in ambiente Linux e può cambiare leggermente a seconda del sistema operativo utilizzato. Le due righe successive sono semplici informazioni di copyright del programma.

La terza riga inizia con `>>>`: questa è l'indicazione (chiamata "prompt") che l'interprete usa per indicare la sua disponibilità ad accettare comandi. Noi

abbiamo inserito `print 1 + 1` e l'interprete ha risposto con `2`.

In alternativa alla riga di comando si può scrivere un programma in un file (detto **script**) ed usare l'interprete per eseguire il contenuto del file. Nell'esempio seguente abbiamo usato un editor di testi per creare un file chiamato `pippo.py`:

```
print 1 + 1
```

Per convenzione, i file contenenti programmi Python hanno nomi che terminano con `.py`.

Per eseguire il programma dobbiamo dire all'interprete il nome dello script:

```
$ python pippo.py
2
```

In altri ambienti di sviluppo i dettagli dell'esecuzione dei programmi possono essere diversi.

La gran parte degli esempi di questo libro sono eseguiti da linea di comando: lavorare da linea di comando è conveniente per lo sviluppo e per il test del programma perché si possono inserire ed eseguire immediatamente singole righe di codice. Quando si ha un programma funzionante lo si dovrebbe salvare in uno script per poterlo eseguire o modificare in futuro senza doverlo riscrivere da capo ogni volta. Tutto ciò che viene scritto in modo "linea di comando" è irrimediabilmente perso nel momento in cui usciamo dall'ambiente Python.

1.2 Cos'è un programma?

Un **programma** è una sequenza di istruzioni che specificano come effettuare una elaborazione. L'elaborazione può essere sia di tipo matematico (per esempio la soluzione di un sistema di equazioni o il calcolo delle radici di un polinomio) che simbolico (per esempio la ricerca e sostituzione di un testo in un documento).

I dettagli sono diversi per ciascun linguaggio di programmazione, ma un piccolo gruppo di istruzioni è praticamente comune a tutti:

- **input:** ricezione di dati da tastiera, da file o da altro dispositivo.
- **output:** scrittura di dati su video, su file o trasmissione ad altro dispositivo.
- **matematiche:** esecuzione di semplici operazioni matematiche, quali l'addizione e la sottrazione.
- **condizionali:** controllo di alcune condizioni ed esecuzione della sequenza di istruzioni appropriata.
- **ripetizione:** ripetizione di un'azione, di solito con qualche variazione.

Che ci si creda o meno, questo è più o meno tutto quello che c'è. Ogni programma che hai usato per quanto complesso possa sembrare (anche il tuo videogioco preferito) è costituito da istruzioni che assomigliano a queste. Possiamo affermare che la programmazione altro non è che la suddivisione di un compito grande e complesso in una serie di sotto-compiti via via più piccoli, finché questi sono sufficientemente semplici da essere eseguiti da una di queste istruzioni fondamentali.

Questo concetto può sembrare un po' vago, ma lo riprenderemo quando parleremo di **algoritmi**.

1.3 Cos'è il debug?

La programmazione è un processo complesso e dato che esso è fatto da esseri umani spesso comporta errori. Per ragioni bizzarre gli errori di programmazione sono chiamati **bug** ed il processo della loro ricerca e correzione è chiamato **debug**.

Sono tre i tipi di errore nei quali si incorre durante la programmazione: gli errori di sintassi, gli errori in esecuzione e gli errori di semantica. È utile distinguerli per poterli individuare più velocemente.

1.3.1 Errori di sintassi

Python può eseguire un programma solo se il programma è sintatticamente corretto, altrimenti l'elaborazione fallisce e l'interprete ritorna un messaggio d'errore. La **sintassi** si riferisce alla struttura di un programma e alle regole concernenti la sua struttura. In italiano, per fare un esempio, una frase deve iniziare con una lettera maiuscola e terminare con un punto. *questa frase contiene un* **errore di sintassi**. *E anche questa*

Per la maggior parte dei lettori qualche errore di sintassi non è un problema significativo, tanto che possiamo leggere le poesie di E.E.Cummings (prive di punteggiatura) senza "messaggi d'errore". Python non è così permissivo: se c'è un singolo errore di sintassi da qualche parte nel programma Python stamperà un messaggio d'errore e ne interromperà l'esecuzione, rendendo impossibile proseguire. Durante le prime settimane della tua carriera di programmatore probabilmente passerai molto tempo a ricercare errori di sintassi. Via via che acquisirai esperienza questi si faranno meno numerosi e sarà sempre più facile rintracciarli.

1.3.2 Errori in esecuzione

Il secondo tipo di errore è l'**errore in esecuzione** (o "runtime"), così chiamato perché l'errore non appare finché il programma non è eseguito. Questi errori sono anche chiamati **eccezioni** perché indicano che è accaduto qualcosa di eccezionale nel corso dell'esecuzione (per esempio si è cercato di dividere un numero per zero).

Gli errori in esecuzione sono rari nei semplici programmi che vedrai nei primissimi capitoli, così potrebbe passare un po' di tempo prima che tu ne incontri uno.

1.3.3 Errori di semantica

Il terzo tipo di errore è l'**errore di semantica**. Se c'è un errore di semantica il programma verrà eseguito senza problemi nel senso che il computer non genererà messaggi d'errore durante l'esecuzione, ma il risultato non sarà ciò che ci si aspettava. Sarà qualcosa di diverso, e questo qualcosa è esattamente ciò che è stato detto di fare al computer.

Il problema sta nel fatto che il programma che è stato scritto non è quello che si desiderava scrivere: il significato del programma (la sua semantica) è sbagliato. L'identificazione degli errori di semantica è un processo complesso perché richiede di lavorare in modo inconsueto, guardando i risultati dell'esecuzione e cercando di capire cosa il programma ha fatto di sbagliato per ottenerli.

1.3.4 Debug sperimentale

Una delle più importanti abilità che acquisirai è la capacità di effettuare il debug (o "rimozione degli errori"). Sebbene questo possa essere un processo frustrante è anche una delle parti più intellettualmente vivaci, stimolanti ed interessanti della programmazione.

In un certo senso il debug può essere paragonato al lavoro investigativo. Sei messo di fronte agli indizi e devi ricostruire i processi e gli eventi che hanno portato ai risultati che hai ottenuto.

Il debug è una scienza sperimentale: dopo che hai avuto un'idea di ciò che può essere andato storto, modifichi il programma e lo provi ancora. Se la tua ipotesi era corretta allora puoi predire il risultato della modifica e puoi avvicinarti di un ulteriore passo all'avere un programma funzionante. Se la tua ipotesi era sbagliata devi ricercarne un'altra. Come disse Sherlock Holmes "Quando hai eliminato l'impossibile ciò che rimane, per quanto improbabile, deve essere la verità" (A.Conan Doyle, *Il segno dei quattro*)

Per qualcuno la programmazione e il debug sono la stessa cosa, intendendo con questo che la programmazione è un processo di rimozione di errori finché il programma fa ciò che ci si aspetta. L'idea è che si dovrebbe partire da un programma che fa *qualcosa* e facendo piccole modifiche ed eliminando gli errori man mano che si procede si dovrebbe avere in ogni momento un programma funzionante sempre più completo.

Linux, per fare un esempio, è un sistema operativo che contiene migliaia di righe di codice, ma esso è nato come un semplice programma che Linus Torvalds usò per esplorare il chip 80386 Intel. Secondo Larry Greenfields, "uno dei progetti iniziali di Linus era un programma che doveva cambiare una riga di AAAA in BBBB e viceversa. Questo in seguito diventò Linux." (*The Linux Users' Guide* Beta Version 1)

I capitoli successivi ti forniranno ulteriori suggerimenti sia per quanto riguarda il debug che per altre pratiche di programmazione.

1.4 Linguaggi formali e naturali

I **linguaggi naturali** sono le lingue parlate, tipo l'inglese, l'italiano, lo spagnolo. Non sono stati "progettati" da qualcuno e anche se è stato imposto un certo ordine nel loro sviluppo si sono evoluti naturalmente.

I **linguaggi formali** sono linguaggi progettati per specifiche applicazioni.

Per fare qualche esempio, la notazione matematica è un linguaggio formale particolarmente indicato ad esprimere relazioni tra numeri e simboli; i chimici usano un linguaggio formale per rappresentare la struttura delle molecole; cosa più importante dal nostro punto di vista, *i linguaggi di programmazione sono linguaggi formali che sono stati progettati per esprimere elaborazioni.*

I linguaggi formali tendono ad essere piuttosto rigidi per quanto riguarda la sintassi: $3 + 3 = 6$ è una dichiarazione matematica sintatticamente corretta, mentre $3 = \div 6\$$ non lo è. H_2O è un simbolo chimico sintatticamente corretto contrariamente a ${}_2Zz$.

Le regole sintattiche si possono dividere in due categorie: la prima riguarda i **token**, la seconda la **struttura**. I token sono gli elementi di base del linguaggio (quali possono essere le parole in letteratura, i numeri in matematica e gli elementi chimici in chimica). Uno dei problemi con $3 = \div 6\$$ è che \$ non è un token valido in matematica; ${}_2Zz$ non è valido perché nessun elemento chimico è identificato dal simbolo Zz.

Il secondo tipo di regola riguarda la struttura della dichiarazione, cioè il modo in cui i token sono disposti. La dichiarazione $3 = \div 6\$$ è strutturalmente non valida perché un segno $\div$ non può essere posto immediatamente dopo un segno $=$. Allo stesso modo l'indice nelle formule chimiche deve essere indicato dopo il simbolo dell'elementi chimico, non prima, e quindi l'espressione ${}_2Zz$ non è valida.

> *Come esercizio crea quella che può sembrare una frase in italiano con dei token non riconoscibili. Poi scrivi un'altra frase con tutti i token validi ma con una struttura non valida.*

Quando leggi una frase in italiano o una dichiarazione in un linguaggio formale devi capire quale sia la struttura della dichiarazione. Questo processo (chiamato **parsing**) in un linguaggio naturale viene realizzato in modo inconscio e spesso non ci si rende conto della sua intrinseca complessità.

Per esempio, quando senti la frase "La scarpa è caduta", capisci che "la scarpa" è il soggetto e che "è caduta" è il verbo. Quando hai analizzato la frase puoi capire cosa essa significa (cioè la semantica della frase). Partendo dal presupposto che tu sappia cosa sia una "scarpa" e cosa significhi "cadere" riesci a comprendere il significato generale della frase.

Anche se i linguaggi formali e quelli naturali condividono molte caratteristiche (token, struttura, sintassi e semantica) ci sono tuttavia molte differenze:

Ambiguità: i linguaggi naturali ne sono pieni ed il significato viene ottenuto anche grazie ad indizi ricavati dal contesto. I linguaggi formali sono progettati per essere completamente non ambigui e ciò significa che ciascuna dichiarazione ha esattamente un significato, indipendente dal contesto.

Ridondanza: per evitare l'ambiguità e ridurre le incomprensioni i linguaggi naturali impiegano molta ridondanza. I linguaggi formali sono meno ridondanti e più concisi.

Letteralità: i linguaggi naturali fanno uso di paragoni e metafore, e possiamo parlare in termini astratti intuendo immediatamente che ciò che sentiamo ha un significato simbolico. I linguaggi formali invece esprimono esattamente ciò che dicono.

Anche se siamo cresciuti apprendendo un linguaggio naturale, la nostra lingua madre, spesso abbiamo difficoltà ad adattarci ai linguaggi formali. In un certo senso la differenza tra linguaggi naturali e formali è come quella esistente tra poesia e prosa, ma in misura decisamente più evidente:

Poesia: le parole sono usate tanto per il loro suono che per il loro significato, e la poesia nel suo complesso crea un effetto o una risposta emotiva. L'ambiguità è non solo frequente, ma spesso addirittura cercata.

Prosa: il significato delle parole è estremamente importante, con la struttura che contribuisce a fornire maggior significato. La prosa può essere soggetta ad analisi più facilmente della poesia, ma può risultare ancora ambigua.

Programmi: il significato di un programma per computer è non ambiguo e assolutamente letterale, può essere compreso nella sua interezza con l'analisi dei token e della struttura.

Qui sono esposti alcuni suggerimenti per la lettura di programmi e di altri linguaggi formali.

- Ricorda che i linguaggi formali sono molto più ricchi di significato dei linguaggi naturali, così è necessario più tempo per leggerli e comprenderli.

- La struttura dei linguaggi formali è molto importante e solitamente non è una buona idea leggerli dall'alto in basso, da sinistra a destra, come avviene per un testo letterario: impara ad analizzare il programma nella tua testa, identificandone i token ed interpretandone la struttura.

- I dettagli sono importanti: piccole cose come errori di ortografia e cattiva punteggiatura sono spesso trascurabili nei linguaggi naturali, ma possono fare una gran differenza in quelli formali.

1.5 Il primo programma

Per tradizione il primo programma scritto in un nuovo linguaggio è chiamato "Hello, World!" perché tutto ciò che fa è scrivere le parole `Hello, World!` a video e nient'altro. In Python questo programma è scritto così:

```
>>> print "Hello, World!"
```

Questo è un esempio di **istruzione di stampa**, che in effetti non stampa nulla su carta limitandosi invece a scrivere un valore sullo schermo. In questo caso ciò che viene "stampato" sono le parole

```
Hello, World!
```

Le virgolette segnano l'inizio e la fine del valore da stampare ed esse non appaiono nel risultato.

Alcune persone giudicano la qualità di un linguaggio di programmazione dalla semplicità del programma "Hello, World!": da questo punto di vista Python sembra essere quanto di meglio sia realizzabile.

1.6 Glossario

Soluzione di problemi: il processo di formulare un problema, trovare una soluzione ed esprimerla.

Linguaggio ad alto livello: un linguaggio di programmazione tipo Python che è progettato per essere facilmente leggibile e utilizzabile dagli esseri umani.

Linguaggio di basso livello: un linguaggio di programmazione che è progettato per essere facilmente eseguibile da un computer; è anche chiamato "linguaggio macchina" o "linguaggio assembly".

Portabilità: caratteristica di un programma di poter essere eseguito su computer di tipo diverso.

Interpretare: eseguire un programma scritto in un linguaggio di alto livello traducendolo ed eseguendolo immediatamente, una linea alla volta.

Compilare: tradurre un programma scritto in un linguaggio di alto livello in un programma di basso livello come preparazione alla successiva esecuzione.

Codice sorgente: un programma di alto livello prima di essere compilato.

Codice oggetto: il risultato ottenuto da un compilatore dopo aver tradotto il codice sorgente.

Eseguibile: altro nome per indicare il codice oggetto pronto per essere eseguito.

Script: programma memorizzato in un file, solitamente destinato ad essere interpretato.

Programma: serie di istruzioni che specificano come effettuare un'elaborazione.

Algoritmo: processo generale usato per risolvere una particolare categoria di problemi.

Bug: errore in un programma (detto anche "baco").

Debug: processo di ricerca e di rimozione di ciascuno dei tre tipi di errori di programmazione.

Sintassi: struttura di un programma.

Errore di sintassi: errore in un programma che rende impossibile la continuazione dell'analisi del codice (il programma non può quindi essere interpretato interamente o compilato).

Errore in esecuzione: errore che non è riconoscibile finché il programma non è stato eseguito e che impedisce la continuazione della sua esecuzione.

Eccezione, errore runtime: altri nomi per indicare un errore in esecuzione.

Errore di semantica: errore nel programma che fa ottenere risultati diversi da quanto ci si aspettava.

Semantica: significato di un programma.

Linguaggio naturale: ognuno dei linguaggi parlati evoluti nel tempo.

Linguaggio formale: ognuno dei linguaggi che sono stati progettati per scopi specifici, quali la rappresentazione di idee matematiche o programmi per computer (tutti i linguaggi per computer sono linguaggi formali).

Token: uno degli elementi di base della struttura sintattica di un programma analogo alla parola nei linguaggi naturali.

Parsing: esame e analisi della struttura sintattica di un programma.

Istruzione di stampa: istruzione che ordina all'interprete Python di scrivere un valore sullo schermo.

Capitolo 2

Variabili, espressioni ed istruzioni

2.1 Valori e tipi

Un **valore** è una delle cose fondamentali manipolate da un programmatore, come lo sono una lettera dell'alfabeto nella scrittura o un numero in matematica. I valori che abbiamo visto finora sono `"Hello, World!"` e `2`, quest'ultimo il risultato ottenuto quando abbiamo sommato `1+1`.

Questi valori appartengono a **tipi** diversi: `2` è un intero, e `"Hello, World!"` è una **stringa**, così chiamata perché contiene una serie (o "stringa") di caratteri. L'interprete può identificare le stringhe perché sono racchiuse da virgolette.

L'istruzione `print` funziona sia per le stringhe che per gli interi.

```
>>> print 4
4
```

Se non sei sicuro del tipo di un valore, l'interprete te lo può dire:

```
>>> type("Hello, World!")
<type 'string'>
>>> type(17)
<type 'int'>
```

Ovviamente le stringhe appartengono al tipo `string` e gli interi al tipo `int`. Non è invece intuitivo il fatto che i numeri con il punto decimale appartengano al tipo `float`: questi numeri sono rappresentati in un formato chiamato **virgola mobile** o **floating-point**.

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

Cosa dire di numeri come `"17"` e `"3.2"`? Sembrano effettivamente dei numeri, ma sono racchiusi tra virgolette e questo sicuramente significa qualcosa. Infatti non siamo in presenza di numeri ma di stringhe:

```
>>> type("17")
<type 'string'>
>>> type("3.2")
<type 'string'>
```

Quando scrivi numeri grandi puoi essere tentato di usare dei punti per delimitare i gruppi di tre cifre, come in `1.000.000`. Questa in effetti non è una cosa consentita in Python ed il valore numerico in questo caso non è valido. È invece corretta una scrittura del tipo

```
>>> print 1,000,000
1 0 0
```

...anche se probabilmente questo risultato non è quello che ci si aspettava! Python interpreta `1,000,000` come una lista di tre valori da stampare (1, 0 e 0). Ricordati di non inserire virgole nei tuoi interi.

2.2 Variabili

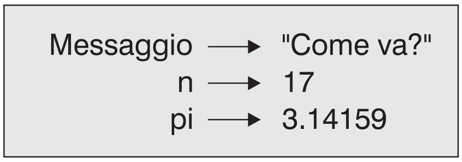
Una delle caratteristiche più potenti in un linguaggio di programmazione è la capacità di manipolare **variabili**. Una variabile è un nome che si riferisce ad un valore.

L'**istruzione di assegnazione** crea nuove variabili e assegna loro un valore:

```
>>> messaggio = "Come va?"
>>> n = 17
>>> pi = 3.14159
```

Questo esempio effettua tre assegnazioni. La prima assegna la stringa `Come va?` ad una nuova variabile chiamata `messaggio`. La seconda assegna l'intero `17` alla variabile `n` e la terza assegna il valore in virgola mobile `3.14159` alla variabile `pi`.

Un modo comune di rappresentare le variabili sulla carta è scriverne il nome con una freccia che punta al valore della variabile. Questo tipo di figura è chiamato **diagramma di stato** perché mostra lo stato in cui si trova la variabile. Questo diagramma mostra il risultato dell'istruzione di assegnazione:



L'istruzione `print` funziona anche con le variabili:

```
>>> print messaggio
Come va?
>>> print n
17
>>> print pi
3.14159
```

ed in ogni caso il risultato è il valore della variabile.

Anche le variabili hanno il tipo; ancora una volta possiamo chiedere all'interprete a quale tipo ogni variabile appartenga:

```
>>> type(message)
<type 'string'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

Il tipo di una variabile è il tipo di valore cui essa si riferisce.

2.3 Nomi delle variabili e parole riservate

I programmatori generalmente scelgono dei nomi significativi per le loro variabili, documentando così a che cosa servono.

I nomi delle variabili possono essere lunghi quanto si desidera e possono contenere sia lettere che numeri, ma devono sempre iniziare con una lettera. È legale usare sia lettere maiuscole che minuscole. Ricorda comunque che l'interprete le considera diverse così che `Numero`, `NUmEro` e `numero` sono a tutti gli effetti variabili diverse.

Il carattere di sottolineatura (_) può far parte di un nome ed è spesso usato in nomi di variabile composti da più parole (per esempio `il_mio_nome` e `prezzo_del_the`. In alternativa le parole possono essere composte usando l'iniziale maiuscola per ciascuna di esse, con il resto dei caratteri lasciati in minuscolo come in `IlMioNome` e `PrezzoDelThe`. Sembra che tra i due metodi quest'ultimo sia il più diffuso così lo adotteremo gradualmente nel corso delle lezioni.

Assegnando un nome illegale alla variabile otterrai un messaggio d'errore di sintassi:

```
>>> 76strumenti = "grande banda"
SyntaxError: invalid syntax
>>> milione$ = 1000000
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
SyntaxError: invalid syntax
```

`76strumenti` è illegale perché non inizia con una lettera. `milione$` è illegale perché contiene un carattere non valido (il segno di dollaro `$`). Ma cosa c'è di sbagliato in `class`?

`class` è una delle **parole riservate** di Python. Le parole riservate definiscono le regole del linguaggio e della struttura e non possono essere usate come nomi di variabili.

Python ha 28 parole riservate:

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

Sarebbe meglio tenere questa lista a portata di mano: se l'interprete ha problemi con il nome che vuoi assegnare ad una variabile e non ne capisci il motivo, prova a controllare se si trova in questa lista.

2.4 Istruzioni

Un'istruzione è un'operazione che l'interprete Python può eseguire. Abbiamo già visto due tipi di istruzioni: istruzioni di stampa [1] e di assegnazione.

Quando scrivi un'istruzione sulla riga di comando, Python la esegue e se previsto stampa il risultato a video. Un'istruzione di assegnazione di per sé non produce risultati visibili mentre il risultato di un'istruzione di stampa è un valore mostrato a video.

Uno script di solito contiene una sequenza di istruzioni: se sono presenti più istruzioni i loro risultati appariranno via via che le singole istruzioni saranno eseguite.

Per esempio lo script:

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

produce questa stampa:

```
1
2
```

2.5 Valutazione delle espressioni

Un'espressione è una combinazione di valori, variabili e operatori. Se scrivi un'espressione sulla riga di comando l'interprete la **valuta** e mostra a video il risultato:

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

Sia un valore (numerico o stringa) che una variabile sono già di per sé delle espressioni:

```
>>> 17
17
>>> x
2
```

[1] D'ora in poi si parlerà di "stampa a video" invece che di "scrittura a video"

La differenza tra “valutare un’espressione” e stamparne il valore è sottile ma importante:

```
>>> messaggio = "Come va?"
>>> messaggio
"Come va?"
>>> print messaggio
Come va?
```

Quando Python mostra il valore di un’espressione usa lo stesso formato che si userebbe per inserirla: nel caso delle stringhe ciò significa che include le virgolette di delimitazione. L’istruzione `print` invece stampa il valore dell’espressione, che nel caso delle stringhe corrisponde al loro contenuto. Le virgolette sono quindi rimosse.

In uno script un valore preso da solo è legale, anche se non fa niente e non produce alcun risultato:

```
17
3.2
"Hello, World!"
1 + 1
```

> *Lo script dell’esempio non produce alcun risultato. Come lo modificheresti per mostrare i quattro valori?*

2.6 Operatori e operandi

Gli **operatori** sono simboli speciali che rappresentano elaborazioni di tipo matematico, quali la somma e la moltiplicazione. I valori che l’operatore usa nei calcoli sono chiamati **operandi**.

Le seguenti espressioni sono tutte legali in Python, ed il loro significato dovrebbe esserti chiaro:

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

L’uso dei simboli `+`, `-`, `/` e delle parentesi sono uguali a all’uso che se ne fa in matematica. L’asterisco (`*`) è il simbolo della moltiplicazione ed il doppio asterisco (`**`) quello dell’elevamento a potenza.

Quando una variabile compare al posto di un operando essa è rimpiazzata dal valore che rappresenta prima che l’operazione sia eseguita.

Addizione, sottrazione, moltiplicazione ed elevamento a potenza fanno tutto ciò che potresti aspettarti, ma la divisione potrebbe non sembrare così intuitiva. L’operazione seguente ha infatti un risultato inatteso:

```
>>> minuti = 59
>>> minuti/60
0
```

Il valore di `minuti` è 59, e 59 diviso 60 è 0.98333, non zero. La ragione di questa differenza sta nel fatto che Python sta facendo una **divisione tra numeri interi**.

Quando entrambi gli operandi sono numeri interi il risultato è sempre un numero intero e per convenzione la divisione tra numeri interi restituisce sempre un numero arrotondato all'intero inferiore (*arrotondamento verso il basso*), anche nel caso in cui il risultato sia molto vicino all'intero superiore.

Una possibile soluzione a questo problema potrebbe essere il calcolo della percentuale, piuttosto che del semplice valore decimale:

```
>>> minuti*100/60
98
```

Ancora una volta il valore è arrotondato per difetto, ma almeno la risposta è approssimativamente corretta. Un'altra alternativa è l'uso della divisione in virgola mobile che tratteremo nella sezione 3.

2.7 Ordine delle operazioni

Quando più operatori compaiono in un'espressione, l'ordine di valutazione dipende dalle **regole di precedenza**. Python segue le stesse regole di precedenza usate in matematica:

- **Parentesi**: hanno il più alto livello di precedenza e possono essere usate per far valutare l'espressione in qualsiasi ordine. Dato che le espressioni tra parentesi sono valutate per prime, `2*(3-1)` dà come risultato 4, e `(1+1)**(5-2)` dà 8. Puoi usare le parentesi per rendere più leggibile un'espressione come in `(minuti*100)/60`, anche se questo non influisce sul risultato.

- **Elevamento a potenza**: ha la priorità successiva così `2**1+1` fa 3 e non 4, e `3*1**3` fa 3 e non 27.

- **Moltiplicazione** e **Divisione** hanno la stessa priorità, superiore a somma e sottrazione. `2*3-1` dà 5 e non 4, e `2/3-1` fa `-1`, e non `1` (ricorda che la divisione intera `2/3` restituisce 0).

- **Addizione** e **Sottrazione**, anch'esse con la stessa priorità.

- Gli operatori con la stessa priorità sono valutati da sinistra verso destra, così che nell'espressione `minuti*100/60`, la moltiplicazione è valutata per prima, ottenendo `5900/60`, che a sua volta restituisce `98`. Se le operazioni fossero state valutate da destra a sinistra il risultato sarebbe stato sbagliato: `59*1=59`.

2.8 Operazioni sulle stringhe

In generale non puoi effettuare operazioni matematiche sulle stringhe, anche se il loro contenuto sembra essere un numero. Se supponiamo che `messaggio` sia di tipo `string` gli esempi proposti di seguito sono illegali:

```
messaggio-1   "Ciao"/123   messaggio*"Ciao"   "15"+2
```

L'operatore `+` funziona con le stringhe anche se la sua funzione è diversa da quella cui siamo abituati in matematica: infatti nel caso di stringhe l'operatore `+` rappresenta il **concatenamento**, cioè l'aggiunta del secondo operando alla fine del primo. Per esempio:

```
frutta = "banana"
verdura = " pomodoro"
print frutta + verdura
```

Il risultato a video di questo programma è `banana pomodoro`. Lo spazio davanti alla parola `pomodoro` è parte della stringa ed è necessario per produrre lo spazio tra le due stringhe concatenate.

Anche l'operatore `*` lavora sulle stringhe pur con un significato diverso rispetto a quello matematico: infatti causa la ripetizione della stringa. Per fare un esempio, `"Casa"*3` è `"CasaCasaCasa"`. Uno degli operandi deve essere una stringa, l'altro un numero intero.

Da una parte questa interpretazione di `+` e di `*` ha senso per analogia con l'addizione e la moltiplicazione in matematica. Così come `4*3` è equivalente a `4+4+4`, ci aspettiamo che `"Casa"*3` sia lo stesso di `"Casa"+"Casa"+"Casa"`, ed effettivamente è così. D'altro canto c'è un particolare sostanziale che rende diverse la somma e la moltiplicazione di numeri e di stringhe.

> *Riesci ad immaginare una proprietà che somma e moltiplicazione tra numeri non condividono con concatenamento e ripetizione di stringhe?*

2.9 Composizione

Finora abbiamo guardato agli elementi di un programma (variabili, espressioni e istruzioni) prendendoli isolatamente, senza parlare di come combinarli.

Una delle più utili caratteristiche dei linguaggi di programmazione è la loro capacità di prendere piccoli blocchi di costruzione e di **comporli**.

Sappiamo già sommare e stampare dei numeri e possiamo fare le due operazioni nello stesso momento:

```
>>>  print 17 + 3
20
```

In realtà l'addizione è stata portata a termine prima della stampa, così che le due operazioni non stanno avvenendo contemporaneamente. Qualsiasi operazione che ha a che fare con i numeri, le stringhe e le variabili può essere usata all'interno di un'istruzione di stampa. Hai già visto un esempio a riguardo:

```
print "Numero di minuti da mezzanotte: ", ore*60+minuti
```

Puoi anche inserire espressioni arbitrarie nella parte destra di un'istruzione di assegnazione:

```
percentuale = (minuti * 100) / 60
```

Questa capacità può non sembrare particolarmente impressionante, ma vedrai presto altri esempi in cui la composizione permette di esprimere elaborazioni complesse in modo chiaro e conciso.

Attenzione: ci sono dei limiti su "dove" puoi usare certe espressioni. Per esempio la parte sinistra di un'istruzione di assegnazione può solo essere una variabile, e non un'espressione. `minuti*60 = ore` è illegale.

2.10 Commenti

Man mano che il programma cresce di dimensioni diventa sempre più difficile da leggere. I linguaggi formali sono ricchi di significato, e può risultare difficile capire a prima vista cosa fa un pezzo di codice o perché è stato scritto in un certo modo.

Per questa ragione è una buona idea aggiungere delle note ai tuoi programmi per spiegare con un linguaggio naturale cosa sta facendo il programma nelle sue varie parti. Queste note sono chiamate **commenti**, e sono marcati dal simbolo `#`:

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

In questo caso il commento appare come una linea a sé stante. Puoi eventualmente inserire un commento alla fine di una riga:

```
percentuale = (minuti*100)/60   # attenzione: divisione intera
```

Qualsiasi cosa scritta dopo il simbolo `#` e fino alla fine della riga viene trascurata nell'esecuzione del programma. Il commento serve al programmatore o ai futuri programmatori che dovranno usare questo codice. In questo ultimo esempio il commento ricorda al lettore che ci potrebbe essere un comportamento inatteso dovuto all'uso della divisione tra numeri interi.

2.11 Glossario

Valore: numero o stringa (o altri tipi di dato che vedremo in seguito) che può essere memorizzato in una variabile o usato in una espressione.

Tipo: formato di un valore che determina come esso possa essere usato nelle espressioni. Finora hai visto i numeri interi (tipo `int`), i numeri in virgola mobile (tipo `float`) e le stringhe (tipo `string`).

Virgola mobile: formato di dati che rappresenta i numeri con parte decimale; è anche detto "floating-point".

Variabile: nome che si riferisce ad un valore.

Istruzione: sezione di codice che rappresenta un comando o un'azione. Finora hai visto istruzioni di assegnazione e di stampa.

Assegnazione: istruzione che assegna un valore ad una variabile.

Diagramma di stato: rappresentazione grafica di una serie di variabili e dei valori cui esse si riferiscono.

Parola riservata: parola che ha un significato particolare per il linguaggio e non può essere usata come nome di variabile o di funzione.

Operatore: simbolo speciale che rappresenta un'elaborazione semplice tipo l'addizione, la moltiplicazione o il concatenamento di stringhe.

Operando: uno dei valori sui quali agisce un operatore.

Espressione: combinazione di variabili, operatori e valori che sono sostituibili da un unico valore equivalente.

Valutazione: semplificazione di un'espressione seguendo una serie di operazioni per produrre un singolo valore.

Divisione tra numeri interi: operazione che divide un numero intero per un altro intero.

Regole di precedenza: insieme di regole che determinano l'ordine nel quale vengono analizzate espressioni complesse dove sono presenti più operandi ed operatori.

Concatenamento: unione di due stringhe tramite l'accodamento della seconda alla prima.

Composizione: capacità di combinare espressioni semplici in istruzioni composite in modo da rappresentare elaborazioni complesse in forma chiara e concisa.

Commento: informazione riguardante il significato di una parte del programma; non ha alcun effetto sull'esecuzione del programma ma serve solo per facilitarne la comprensione.

Capitolo 3

Funzioni

3.1 Chiamate di funzioni

Hai già visto un esempio di **chiamata di funzione**:

```
>>> type("32")
<type 'string'>
```

Il nome della funzione è `type` e mostra il tipo di valore della variabile. Il valore della variabile, che è chiamato **argomento** della funzione, deve essere racchiuso tra parentesi. È comune dire che una funzione "prende" o "accetta" un argomento e "ritorna" o "restituisce" un risultato. Il risultato è detto **valore di ritorno**. Invece di stampare il valore di ritorno possiamo assegnarlo ad una variabile:

```
>>> betty = type("32")
>>> print betty
<type 'string'>
```

Come esempio ulteriore, la funzione `id` prende un valore o una variabile e ritorna un intero che agisce come un identificatore unico del valore:

```
>>> id(3)
134882108
>>> betty = 3
>>> id(betty)
134882108
```

Ogni valore ha un `id` unico che rappresenta dove è depositato nella memoria del computer. L'`id` di una variabile è l'`id` del valore della variabile cui essa si riferisce.

3.2 Conversione di tipo

Python fornisce una raccolta di funzioni interne che converte valori da un tipo all'altro. La funzione `int` prende ogni valore e lo converte, se possibile, in intero. Se la conversione non è possibile mostra un messaggio d'errore:

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

`int` può anche convertire valori in virgola mobile in interi, ma ricorda che nel farlo tronca (cioè toglie) la parte decimale.

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

La funzione `float` converte interi e stringhe in numeri in virgola mobile:

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

Infine `str` converte al tipo stringa:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

Può sembrare strano il fatto che Python distingua il valore intero `1` dal corrispondente valore in virgola mobile `1.0`. Questi rappresentano effettivamente uno stesso numero ma appartengono a tipi differenti (rispettivamente intero e in virgola mobile) e quindi vengono rappresentati in modo diverso all'interno della memoria del computer.

3.3 Forzatura di tipo

Per tornare ad un esempio del capitolo precedente (la divisione di *minuti* per 60), ora che sappiamo convertire i tipi abbiamo un modo ulteriore per gestire la divisione tra interi. Supponiamo di dover calcolare la frazione di ora che è trascorsa: l'espressione più ovvia, `minuti/60`, lavora con numeri interi, così il risultato è sempre 0 anche se sono trascorsi 59 minuti.

Una delle soluzioni è quella di convertire `minuti` in virgola mobile e calcolare il risultato della divisione in virgola mobile:

```
>>> minuti = 59
>>> float(minuti) / 60.0
0.983333333333
```

In alternativa possiamo avvantaggiarci delle regole di conversione automatica dei tipi chiamate **forzature di tipo**. Nel caso di operatori matematici se uno degli operandi è `float`, l'altro è automaticamente convertito a `float`:

```
>>> minuti = 59
>>> minuti / 60.0
0.983333333333
```

Convertendo il denominatore a valore in virgola mobile forziamo Python a calcolare il risultato di una divisione in virgola mobile.

3.4 Funzioni matematiche

In matematica hai probabilmente visto funzioni del tipo sin e log, ed hai imparato a calcolare espressioni quali $sin(\pi/2)$ e $log(1/x)$. Innanzitutto devi calcolare il valore dell'espressione tra parentesi (l'argomento). Nell'esempio $\pi/2$ è approssimativamente 1.571 e se x vale 10.0, $1/x$ è 0.1.

Poi valuti la funzione stessa tramite calcoli o tabelle. `sin` di 1.571 è circa 1, e `log` in base 10 di 0.1 è -1.

Questo processo può essere applicato ripetutamente per valutare espressioni complesse del tipo $log(1/sin(\pi/2))$. In questo caso devi iniziare dall'espressione più interna $\pi/2$, calcolando poi il seno con sin, seguito dall'inverso del seno $1/x$ e dal logaritmo dell'inverso $log(x)$.

Python è provvisto di un modulo matematico che permette di eseguire le più comuni operazioni matematiche. Un **modulo** è un file che contiene una raccolta di funzioni raggruppate.

Prima di poter usare le funzioni di un modulo dobbiamo dire all'interprete di caricare il modulo in memoria. Questa operazione viene detta "importazione":

```
>>> import math
```

Per chiamare una funzione di un modulo dobbiamo specificare il nome del modulo che la contiene e il nome della funzione separati da un punto. Questo formato è chiamato **notazione punto**.

```
>>> decibel = math.log10 (17.0)
>>> angolo = 1.5
>>> altezza = math.sin(angolo)
```

La prima istruzione assegna a `decibel` il logaritmo di 17 in base 10. È anche disponibile la funzione `log` che calcola il logaritmo naturale di un numero.

La terza istruzione trova il seno del valore della variabile `angolo`. `sin` e le altre funzioni trigonometriche (`cos`, `tan`, etc.) accettano argomenti in radianti e non i gradi. Per convertire da gradi in radianti devi dividere per 360 e moltiplicare per 2 π. Per esempio, per calcolare il seno di 45 gradi, prima trasforma l'angolo in radianti e poi usa la funzione seno:

```
>>> gradi = 45
>>> angolo = gradi * 2 * math.pi / 360.0
>>> math.sin(angolo)
```

La costante `pi` fa già parte del modulo matematico `math`. Se conosci un po' di geometria puoi verificare il risultato confrontandolo con $\sqrt{2}/2$:

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

3.5 Composizione

Così come in matematica anche in Python le funzioni possono essere composte, facendo in modo che il risultato di una possa essere usato come argomento di un'altra:

```
>>> x = math.cos(angolo + math.pi/2)
```

Questa istruzione prende il valore di π (`math.pi`), lo divide per 2 e somma il quoziente ad `angolo`. La somma è poi passata come argomento alla funzione `cos` che ne calcola il coseno.

```
>>> x = math.exp(math.log(10.0))
```

In quest'altro esempio l'istruzione `log` calcola il logaritmo naturale (in base e) di 10 e poi eleva e al valore precedentemente calcolato. Il risultato viene assegnato ad `x`.

3.6 Aggiungere nuove funzioni

Finora abbiamo soltanto usato funzioni che fanno parte di Python, ma è possibile aggiungerne di nuove. La creazione di nuove funzioni per risolvere un particolare problema è infatti una tra le cose più utili di un linguaggio di programmazione generale, intendendo con "generale" che il linguaggio non è destinato ad un settore di applicazioni particolari, quale può essere quello scientifico o finanziario, ma che può essere usato in ogni campo).

Nel contesto della programmazione una **funzione** è una sequenza di istruzioni che esegue una determinata operazione. Questa azione è descritta in una **definizione di funzione**. Le funzioni che abbiamo usato finora sono state definite per noi e le loro definizioni sono rimaste nascoste: questa è una cosa positiva in quanto possiamo usarle senza doverci preoccupare di come sono state definite da chi le ha scritte.

La sintassi per la definizione di una funzione è:

```
def NOME( LISTA_DEI_PARAMETRI ):
  ISTRUZIONI
```

Puoi usare qualsiasi nome per una funzione, fatta eccezione per le parole riservate di Python. La lista dei parametri di una funzione specifica quali informazioni, sempre che ne sia prevista qualcuna, desideri fornire alla funzione per poterla usare.

All'interno della funzione sono naturalmente presenti delle istruzioni e queste devono essere indentate rispetto al margine sinistro. Di solito il rientro è di un paio di spazi, ma questa è solo una convenzione: per questioni puramente estetiche potresti volerne usare di più. Mentre nella maggior parte dei linguaggi il rientro è facoltativo e dipende da come il programmatore vuole organizzare visivamente il suo codice, in Python il rientro è obbligatorio. Questa scelta può sembrare un vincolo forzoso, ma ha il vantaggio di garantire una certa uniformità di stile e per quanto disordinato possa essere un programmatore il codice conserverà sempre un minimo di ordine.

La prima coppia di funzioni che stiamo per scrivere non ha parametri e la sintassi è:

```
def UnaRigaVuota():
  print
```

Questa funzione si chiama `UnaRigaVuota`. Le parentesi vuote stanno ad indicare che non ci sono parametri. La funzione è composta da una singola riga che stampa una riga vuota (questo è ciò che succede quando usi il comando `print` senza argomenti).

La sintassi per richiamare la funzione che hai appena definito è la stessa che hai usato per richiamare le funzioni predefinite:

```
print "Prima riga."
UnaRigaVuota()
print "Seconda riga."
```

Il risultato del programma è una scrittura a video:

```
Prima riga.

Seconda riga.
```

Nota lo spazio tra le due righe. Cosa avresti dovuto fare se c'era bisogno di più spazio? Ci sono varie possibilità. Avresti potuto chiamare più volte la funzione:

```
print "Prima riga."
UnaRigaVuota()
UnaRigaVuota ()
UnaRigaVuota ()
print "Seconda riga."
```

o avresti potuto creare una nuova funzione chiamata `TreRigheVuote` che stampa tre righe vuote:

```
def TreRigheVuote():
  UnaRigaVuota()
```

```
  UnaRigaVuota()
  UnaRigaVuota()

print "Prima riga."
TreRigheVuote()
print "Seconda riga."
```

Questa funzione contiene tre istruzioni, tutte indentate di due spazi proprio per indicare che fanno parte della definizione della funzione. Dato che dopo la definizione, alla fine del terzo `UnaRigaVuota()`, la riga successiva, `print "Prima riga."` non ha più indentazione, ciò significa che questa non fa più parte della definizione e che la definizione deve essere considerata conclusa.

Puoi notare alcune cose riguardo questo programma:

1. Puoi chiamare più volte la stessa procedura. È abbastanza comune e utile farlo.
2. Una funzione può chiamare altre funzioni al suo interno: in questo caso `TreRigheVuote` chiama `UnaRigaVuota`.

Può non essere ancora chiaro perché sia il caso di creare tutte queste nuove funzioni. Effettivamente di ragioni ce ne sono tante, qui ne indichiamo due:

- Creare una funzione ti dà l'opportunità di raggruppare e identificare con un nome un gruppo di istruzioni. Le funzioni possono semplificare un programma nascondendo un'elaborazione complessa dietro un singolo comando, e usando parole comprensibili per richiamarla invece di codice difficile da capire.
- La creazione di funzioni rende più piccolo il programma, eliminando le parti ripetitive. Per fare un esempio, se vogliamo stampare 9 righe vuote, possiamo chiamare 9 volte la funzione `UnaRigaVuota` o 3 volte la funzione `TreRigheVuote`.

  Esercizio: scrivi una funzione chiamata `NoveRigheVuote` *che usa* `TreRigheVuote` *per scrivere 9 righe bianche. Cosa faresti poi per scrivere 27 righe bianche?*

3.7 Definizioni e uso

Raggruppando assieme i frammenti di codice della sezione precedente il programma diventa:

```
def UnaRigaVuota():
  print

def TreRigheVuote():
  UnaRigaVuota()
  UnaRigaVuota()
```

```
  UnaRigaVuota()

print "Prima riga."
TreRigheVuote()
print "Seconda riga."
```

Questo programma contiene la definizione di due funzioni: `UnaRigaVuota` e `TreRigheVuote`. Le definizioni di funzione sono eseguite come le altre istruzioni ma il loro effetto è quello di creare una nuova funzione. Le istruzioni all'interno di una definizione non sono eseguite finché la funzione non è chiamata e la definizione in sé non genera alcun risultato. Come puoi facilmente immaginare, prima di poter usare una funzione devi averla definita: la definizione della funzione deve sempre precedere la sua chiamata.

> *Esercizio: sposta le ultime tre righe del programma all'inizio, per fare in modo che la chiamata alle funzioni appaia prima della loro definizione. Esegui il programma e vedi che tipo di messaggio d'errore ottieni.*

> *Esercizio: inizia con il programma funzionante e sposta la definizione di* `UnaRigaVuota` dopo la definizione di `TreRigheVuote`. Cosa succede quando esegui il programma?

3.8 Flusso di esecuzione

Per assicurarti che una funzione sia definita prima del suo uso devi conoscere l'ordine in cui le istruzioni sono eseguite cioè il **flusso di esecuzione** del programma.

L'esecuzione inizia sempre alla prima riga del programma e le istruzioni sono eseguite una alla volta dall'alto verso il basso.

La definizione di funzioni non altera il flusso di esecuzione del programma ma ricorda che le istruzioni all'interno delle funzioni non sono eseguite finché la funzione non viene chiamata. Sebbene questo non sia una cosa che avviene frequentemente, puoi anche definire una funzione all'interno di un'altra funzione. In questo caso la funzione più interna non sarà eseguita finché non viene chiamata anche quella più esterna.

La chiamata alle funzioni è una deviazione nel flusso di esecuzione: invece di proseguire con l'istruzione successiva, il flusso salta alla prima riga della funzione chiamata ed esegue tutte le sue istruzioni; alla fine della funzione il flusso riprende dal punto dov'era stato deviato dalla chiamata di funzione.

Questo è abbastanza comprensibile ma non devi dimenticare che una funzione ne può chiamare un'altra al suo interno. Può succedere che il programma principale chiami una funzione che a sua volta ne chiama un'altra: alla fine della seconda funzione il flusso torna alla prima, dov'era stato lasciato in sospeso, e quando anche la prima funzione è stata completata il flusso di esecuzione torna al programma principale.

Fortunatamente Python è sufficientemente intelligente da ricordare dove il flusso di esecuzione viene via via interrotto e sa dove riprendere quando una funzione è conclusa. Se il flusso di programma giunge all'ultima istruzione, dopo la sua esecuzione il programma è terminato.

Qual è il senso di tutto questo discorso? Quando leggi un programma non limitarti a farlo dall'alto in basso, come stessi leggendo un libro: cerca invece di seguire il flusso di esecuzione, con i suoi salti all'interno delle procedure.

3.9 Parametri e argomenti

Alcune delle funzioni che devi usare richiedono argomenti, i valori che controllano come la funzione deve portare a termine il proprio compito. Per esempio, se vuoi trovare il seno di un numero devi indicare quale sia questo numero: `sin` si aspetta infatti un valore numerico come argomento.

Alcune funzioni prendono due o più parametri: `pow` si aspetta due argomenti che sono la base e l'esponente in un'operazione di elevamento a potenza. Dentro la funzione i valori che sono passati vengono assegnati a variabili chiamate **parametri**.

Eccoti un esempio di definizione di una funzione con un parametro:

```
def Stampa2Volte(Valore):
  print Valore, Valore
```

Questa funzione si aspetta un unico argomento e lo assegna ad un parametro chiamato `Valore`. Il valore del parametro (a questo punto del programma non sappiamo nemmeno di che tipo sarà, se stringa, intero o di altro tipo) è stampato due volte. La stampa è poi conclusa con un ritorno a capo. Il nome `Valore` è stato scelto per ricordarti che sta a te sceglierne uno sufficientemente esplicativo, e di solito ne sceglierai qualcuno che ricordi l'uso della funzione o della variabile.

La funzione `Stampa2Volte` funziona per ogni tipo di dato che può essere stampato:

```
>>> Stampa2Volte('Pippo')
Pippo Pippo
>>> Stampa2Volte(5)
5 5
>>> Stampa2Volte(3.14159)
3.14159 3.14159
```

Nella prima chiamata di funzione l'argomento è una stringa, nella seconda un intero e nella terza un numero in virgola mobile (`float`).

Le stesse regole per la composizione che sono state descritte per le funzioni predefinite valgono anche per le funzioni definite da te, così che possiamo usare una qualsiasi espressione valida come argomento per `Stampa2Volte`:

```
>>> Stampa2Volte("Pippo"*4)
PippoPippoPippoPippo PippoPippoPippoPippo
>>> Stampa2Volte(math.cos(math.pi))
-1.0 -1.0
```

Come al solito, l'espressione passata come argomento è valutata prima dell'esecuzione della funzione, così nell'esempio appena proposto `Stampa2Volte` ritorna il risultato `PippoPippoPippoPippo PippoPippoPippoPippo` invece di `"Pippo"*4 "Pippo"*4`.

Una nota per quanto riguarda le stringhe: le stringhe possono essere racchiuse sia da virgolette `"ABC"` che da apici `'ABC'`. Il tipo di delimitatore NON usato per delimitare la stringa, l'apice se si usano le virgolette, le virgolette se si usa l'apice, può essere usato all'interno della stringa. Ad esempio sono valide le stringhe `"apice ' nella stringa"` e `'virgoletta " nella stringa'`, ma non lo sono `'apice ' nella stringa'` e `"virgoletta " nella stringa"`, dato che in questo caso l'interprete non riesce a stabilire quale sia il fine stringa desiderato dal programmatore.

> *Esercizio: scrivi una chiamata a* `Stampa2Volte` *che stampa a video la stringa* `"Pippo"*4 "Pippo"*4` *così com'è scritta.*

Naturalmente possiamo usare una variabile come argomento di una funzione:

```
>>> Messaggio = 'Come va?'
>>> Stampa2Volte(Messaggio)
Come va? Come va?
```

Il nome della variabile che passiamo come argomento (`Messaggio`) non ha niente a che fare con il nome del parametro nella definizione della funzione (`Valore`). Non ha importanza conoscere il nome originale con cui sono stati identificati i parametri durante la definizione della funzione.

3.10 Variabili e parametri sono locali

Quando crei una **variabile locale** all'interno di una funzione, essa esiste solo all'interno della funzione e non puoi usarla all'esterno. Per esempio:

```
def StampaUnite2Volte(Parte1, Parte2):
  Unione = Parte1 + Parte2
  Stampa2Volte(Unione)
```

Questa funzione prende due argomenti, li concatena e poi ne stampa il risultato due volte. Possiamo chiamare la funzione con due stringhe:

```
>>> Strofa1 = "Nel mezzo "
>>> Strofa2 = "del cammin"
>>> StampaUnite2Volte(Strofa1, Strofa2)
Nel mezzo del cammin Nel mezzo del cammin
```

Quando `StampaUnite2Volte` termina, la variabile `Unione` è distrutta. Se proviamo a stamparla quando il flusso di esecuzione si trova all'esterno della funzione `StampaUnite2Volte` otterremo un messaggio d'errore:

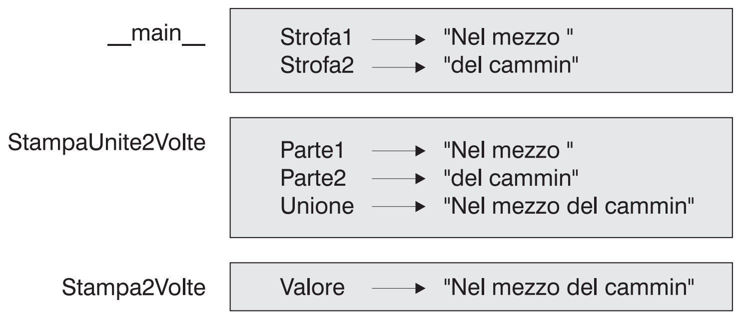
```
>>> print Unione
NameError: Unione
```

Anche i parametri sono locali: al di fuori della funzione `StampaUnite2Volte`, non esiste alcuna cosa chiamata `messaggio`. Se proverai ad usarla al di fuori della funzione dov'è definita Python ti mostrerà ancora una volta un messaggio d'errore.

3.11 Diagrammi di stack

Per tenere traccia di quali variabili possono essere usate è talvolta utile disegnare un **diagramma di stack**. Come i diagrammi di stato, i diagrammi di stack mostrano il valore di ciascuna variabile e indicano a quale funzione essa appartenga.

Ogni funzione è rappresentata da un **frame**, un rettangolo con il nome della funzione a fianco e la lista dei parametri e delle variabili al suo interno. Il diagramma di stack nel caso dell'esempio precedente è:



L'ordine dello stack mostra chiaramente il flusso di esecuzione. Possiamo vedere che `Stampa2Volte` è chiamata da `StampaUnite2Volte` e che `StampaUnite2Volte` è chiamata da `__main__`. `__main__` è un nome speciale che indica il programma principale che di per sé (non essendo definito con `def` come si fa per le funzioni) non ha un nome vero e proprio. Quando crei una variabile all'esterno di ogni funzione, essa appartiene a `__main__`.

Ogni parametro si riferisce al valore che ha l'argomento corrispondente. Così `Parte1` ha lo stesso valore di `Strofa1`, `Parte2` ha lo stesso valore di `Strofa2` e `Valore` lo stesso di `Unione`.

Se avviene un errore durante la chiamata di una funzione, Python mostra il nome della funzione, il nome della funzione che l'ha chiamata, il nome della funzione che ha chiamato quest'ultima e così via, fino a raggiungere il primo livello che è sempre `__main__`.

Ad esempio se cerchiamo di chiamare `Unione` dall'interno di `Stampa2Volte`, otteniamo un errore di tipo `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    StampaUnite2Volte(Parte1, Parte2)
  File "test.py", line 5, in StampaUnite2Volte
    Stampa2Volte(Unione)
  File "test.py", line 9, in Stampa2Volte
    print Unione
NameError: Unione
```

Questa lista temporale delle chiamate delle funzioni è detta **traccia**. La traccia ti dice in quale file è avvenuto l'errore, che riga all'interno del file si stava eseguendo in quel momento ed il riferimento alla funzione che ha causato l'errore.

Nota che c'è una notevole somiglianza tra traccia e diagramma di stack e questa somiglianza non è certamente una coincidenza.

3.12 Funzioni con risultati

Puoi notare come alcune delle funzioni che hai usato, tipo le funzioni matematiche, restituiscono dei risultati. Altre funzioni, come `UnaRigaVuota`, eseguono un'azione senza ritornare alcun valore. Questa differenza solleva qualche domanda:

1. Cosa succede se chiami una funzione e non fai niente con il risultato che viene restituito (per esempio non lo assegni ad una variabile e non lo usi come parte di una espressione)?

2. Cosa succede se usi una funzione che non produce risultato come parte di un'espressione (per esempio `UnaRigaVuota() + 7`)?

3. Puoi scrivere funzioni che producono risultati, o sei costretto a limitarti a semplici funzioni tipo `UnaRigaVuota` e `Stampa2Volte` che eseguono azioni in questo caso piuttosto banali?

La risposta alla terza domanda la troveremo al capitolo 5.

> *Esercizio: trova la risposta alle altre due domande provando i due casi. Quando non hai chiaro cosa sia legale e cosa non lo sia è buona regola provare per vedere come reagisce l'interprete.*

3.13 Glossario

Chiamata di funzione: istruzione che esegue una funzione. Consiste di un nome di funzione seguito da una serie di argomenti racchiuso tra parentesi.

Argomento: valore fornito alla funzione quando questa viene chiamata. Il valore è assegnato al corrispondente parametro della funzione.

Valore di ritorno: risultato di una funzione.

Conversione di tipo: istruzione esplicita che prende un valore di un tipo e lo converte nel corrispondente valore di un altro tipo.

Forzatura di tipo: conversione automatica di tipo secondo le regole di forzatura di Python.

Modulo: file che contiene una raccolta di funzioni correlate.

Notazione punto: sintassi per la chiamata di una funzione definita in un altro modulo, specificando il nome del modulo di appartenenza, seguito da un punto e dal nome della funzione con gli eventuali argomenti tra parentesi.

Funzione: sequenza di istruzioni identificata da un nome che svolge qualche operazione utile. Le funzioni possono avere o meno dei parametri e possono produrre o meno un risultato.

Definizione della funzione: istruzioni che creano una nuova funzione, specificandone il nome, i parametri e le operazioni che essa deve eseguire.

Flusso di esecuzione: ordine in cui le istruzioni sono interpretate quando il programma viene eseguito.

Parametro: nome usato all'interno della funzione per riferirsi al valore passato come argomento.

Variabile locale: variabile definita all'interno di una funzione. Una variabile locale può essere usata unicamente all'interno della funzione dov'è definita.

Diagramma di stack: rappresentazione grafica delle funzioni, delle loro variabili e dei valori cui esse si riferiscono.

Frame: rettangolo che in un diagramma di stack rappresenta una chiamata di funzione. Indica le variabili locali e i parametri della funzione.

Traccia: lista delle funzioni in corso di esecuzione stampata in caso di errore in esecuzione.

Capitolo 4

Istruzioni condizionali e ricorsione

4.1 L'operatore modulo

L'**operatore modulo** opera sugli interi (e sulle espressioni intere) e produce il resto della divisione del primo operando diviso per il secondo. In Python l'operatore modulo è rappresentato dal segno percentuale (`%`). La sintassi è la stessa degli altri operatori matematici:

```
>>> Quoziente = 7 / 3
>>> print Quoziente
2
>>> Resto = 7 % 3
>>> print Resto
1
```

Così 7 diviso 3 dà 2, con il resto di 1.

L'operatore modulo è molto utile in quanto ti permette di controllare se un numero è divisibile per un altro: se `x % y` è 0, allora `x` è divisibile per `y`.

Inoltre può essere usato per estrarre la cifra più a destra di un numero: `x%10` restituisce la cifra più a destra in base 10. Allo stesso modo `x%100` restituisce le ultime due cifre.

4.2 Espressioni booleane

Un'**espressione booleana** è un'espressione che è o *vera* o *falsa*. In Python un'espressione che è *vera* ha valore 1, un'espressione *falsa* ha valore 0.

L'operatore `==` confronta due valori e produce un risultato di tipo booleano:

```
>>> 5 == 5
1
>>> 5 == 6
0
```

Nella prima riga i due operandi sono uguali, così l'espressione vale 1 (*vero*); nella seconda riga 5 e 6 non sono uguali, così otteniamo 0 (*falso*).

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

```
x != y       # x è diverso da 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?
```

Sebbene queste operazioni ti possano sembrare familiari, i simboli Python sono diversi da quelli usati comunemente in matematica. Un errore comune è quello di usare il simbolo di uguale (`=`) invece del doppio uguale (`==`): ricorda che `=` è un operatore di assegnazione e `==` un operatore di confronto. Inoltre in Python non esistono simboli del tipo `=<` e `=>`, ma solo gli equivalenti `<=` e `>=`.

4.3 Operatori logici

Ci sono tre **operatori logici**: `and`, `or` e `not`. Il significato di questi operatori è simile al loro significato in italiano: per esempio, `(x>0) and (x<10)` è vera solo se `x` è più grande di 0 *e* meno di 10.

`(n%2==0) or (n%3==0)` è vera se si verifica almeno una delle due condizioni e cioè se il numero è divisibile per 2 *o* per 3.

Infine, l'operatore `not` nega il valore di un'espressione booleana, trasformando in falsa un'espressione vera e viceversa. Così se `x>y` è vera (x è maggiore di y), `not(x>y)` è falsa.

A dire il vero gli operatori booleani dovrebbero restituire un valore *vero* o *falso*, ma da questo punto di vista Python (come parte dei linguaggi di programmazione) non sembra essere troppo fiscale: infatti ogni valore diverso da zero viene considerato *vero* e lo zero è considerato *falso*.

```
>>>  x = 5
>>>  x and 1
1
>>>  y = 0
>>>  y and 1
0
```

In generale, le righe appena viste pur essendo lecite non sono considerate un buon esempio di programmazione: se vuoi confrontare un valore con zero è sempre meglio farlo in modo esplicito, con un'espressione del tipo

```
>>>  x != 0
```

4.4 Esecuzione condizionale

Per poter scrivere programmi di una certa utilità dobbiamo essere messi in grado di valutare delle condizioni e di far seguire differenti percorsi al flusso di esecuzione a seconda del risultato della valutazione. Le **istruzioni condizionali** ci offrono questa possibilità. La forma più semplice di istruzione `if` è la seguente:

```
if x > 0:
  print "x e' positivo"
```

L'espressione booleana dopo l'istruzione `if` è chiamata **condizione**. L'istruzione indentata che segue i due punti della riga `if` viene eseguita solo se la condizione è vera. Se la condizione è falsa non viene eseguito alcunché.

Come nel caso di altre istruzioni composte, l'istruzione `if` è costituita da un'intestazione e da un blocco di istruzioni:

```
INTESTAZIONE:
  PRIMA RIGA DI ISTRUZIONI
  ...
  ULTIMA RIGA DI ISTRUZIONI
```

L'intestazione inizia su di una nuova riga e termina con il segno di due punti. La serie di istruzioni indentate che seguono sono chiamate **blocco di istruzioni**. La prima riga di istruzioni non indentata marca la fine del blocco di istruzioni e non ne fa parte. Un blocco di istruzioni all'interno di un'istruzione composta è anche chiamato **corpo** dell'istruzione.

Non c'è un limite al numero di istruzioni che possono comparire nel corpo di un'istruzione `if` ma deve sempre essercene almeno una. In qualche occasione può essere utile avere un corpo vuoto, ad esempio quando il codice corrispondente non è ancora stato scritto ma si desidera ugualmente poter provare il programma. In questo caso puoi usare l'istruzione `pass`, che è solo un segnaposto e non fa niente:

```
if x > 0:
  pass
```

4.5 Esecuzione alternativa

Una seconda forma di istruzione `if` è l'esecuzione alternativa, nella quale ci sono due possibilità di azione e il valore della condizione determina quale delle due debba essere scelta. La sintassi è:

```
if x%2 == 0:
  print x, "e' pari"
else:
  print x, "e' dispari"
```

Se il resto della divisione intera di `x` per 2 è zero allora sappiamo che `x` è pari e il programma mostra il messaggio corrispondente. Se la condizione è falsa viene eseguita la serie di istruzioni descritta dopo la riga `else` (che in inglese significa "altrimenti").

Le due alternative sono chiamate **ramificazioni** perché rappresentano delle ramificazioni nel flusso di esecuzione del programma, e solo una di esse verrà effettivamente eseguita.

Una nota: se hai bisogno di controllare la parità di un numero (vedere se il numero è pari o dispari), potresti desiderare di creare una funzione apposita da poter riutilizzare in seguito:

```
def StampaParita(x):
  if x%2 == 0:
    print x, "e' pari"
  else:
    print x, "e' dispari"
```

Così per ogni valore intero di `x`, `StampaParita` mostra il messaggio appropriato. Quando chiami questa funzione puoi fornire qualsiasi espressione intera come argomento.

```
>>> StampaParita(17)
>>> StampaParita(y+1)
```

4.6 Condizioni in serie

Talvolta ci sono più di due possibilità per la continuazione del programma, così possiamo aver bisogno di più di due ramificazioni. Un modo per esprimere questo caso sono le **condizioni in serie**:

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

`elif` è l'abbreviazione di "else if", che in inglese significa "altrimenti se". Anche in questo caso solo uno dei rami verrà eseguito, a seconda del confronto tra `x` e `y`. Non c'è alcun limite al numero di istruzioni `elif` ma è eventualmente possibile inserire un'unica istruzione `else` che deve essere l'ultima dell'elenco e che rappresenta l'azione da eseguire quando nessuna delle condizioni precedenti è stata soddisfatta. La presenza di un'istruzione `else` è facoltativa.

```
if scelta == 'A':
  FunzioneA()
elif scelta == 'B':
  FunzioneB()
elif scelta == 'C':
  FunzioneC()
else:
  print "Scelta non valida"
```

Le condizioni sono controllate nell'ordine in cui sono state scritte. Se la prima è falsa viene provata la seconda e così via. Non appena una è verificata viene eseguito il ramo corrispondente e l'intera istruzione `if` viene conclusa. In ogni caso, anche se fossero vere altre condizioni, dopo l'esecuzione della prima queste vengono trascurate. Se nessuna condizione è vera ed è presente un `else` verrà eseguito il codice corrispondente; se non è presente non verrà eseguito niente.

> *Esercizio: scrivi due funzioni basate sugli esempi proposti, una che confronta x e y (*`Confronta(x, y)`*) e l'altra che controlla se un valore passato come parametro appartiene ad una lista di valori validi (*`ElaboraScelta(scelta)`*).*

4.7 Condizioni annidate

Un'espressione condizionale può anche essere inserita nel corpo di un'altra espressione condizionale: un'espressione di questo tipo viene detta "condizione annidata".

```
if x == y:
  print x, "e", y, "sono uguali"
else:
  if x < y:
    print x, "e' minore di", y
  else:
    print x, "e' maggiore di", y
```

La prima condizione (`if x == y`) contiene due rami: il primo è scelto quando x e y sono uguali, il secondo quando sono diversi. All'interno del secondo (subito sotto il primo `else:`) troviamo un'altra istruzione `if`, che a sua volta prevede un'ulteriore ramificazione. Entrambi i rami del secondo `if` sono istruzioni di stampa ma potrebbero contenere a loro volta ulteriori istruzioni condizionali.

Sebbene l'indentazione delle istruzioni renda evidente la struttura dell'esempio, le istruzioni condizionali annidate in livelli sempre più profondi diventano sempre più difficili da leggere, quindi è una buona idea evitarle quando è possibile.

Gli operatori logici permettono un modo molto semplice di semplificare le espressioni condizionali annidate:

```
if 0 < x:
  if x < 10:
    print "x e' un numero positivo."
```

L'istruzione di stampa `print` è eseguita solo se entrambe le condizioni (`x>0` e `x<10`) sono verificate contemporaneamente. Possiamo quindi usare l'operatore booleano `and` per combinarle:

```
if 0<x and x<10:
  print "x e' un numero positivo."
```

Questo tipo di condizione è così frequente che Python permette di usare una forma semplificata che ricorda da vicino quella corrispondente usata in matematica:

```
if 0 < x < 10:
  print "x e' un numero positivo."
```

A tutti gli effetti i tre esempi sono equivalenti per quanto riguarda la semantica (il significato) del programma.

4.8 L'istruzione `return`

L'istruzione `return` ti permette di terminare l'esecuzione di una funzione prima di raggiungerne la fine. Questo può servire quando viene riconosciuta una condizione d'errore:

```
import math

def StampaLogaritmo(x):
  if x <= 0:
    print "Inserire solo numeri positivi!"
    return

  risultato = math.log(x)
  print "Il logaritmo di",x,"e'", risultato
```

La funzione `StampaLogaritmo` accetta un parametro chiamato `x`. La prima operazione controlla che esso sia positivo; in caso contrario stampa un messaggio d'errore e termina prematuramente la funzione con `return`.

Ricorda che dovendo usare una funzione del modulo `math` è necessario importare il modulo.

4.9 Ricorsione

Abbiamo detto che è perfettamente lecito che una funzione ne chiami un'altra e di questo hai avuto modo di vedere parecchi esempi. Abbiamo invece trascurato di dirti che è anche lecito che una funzione possa chiamare sé stessa. Può non essere immediatamente ovvio il motivo per cui questo sia utile, ma questa è una delle cose più interessanti che un programma possa fare. Per fare un esempio dai un'occhiata a questa funzione:

```
def ContoAllaRovescia(n):
  if n == 0:
    print "Partenza!"
  else:
    print n
    ContoAllaRovescia(n-1)
```

`ContoAllaRovescia` si aspetta che il parametro sia un intero positivo. Se `n` vale 0, viene stampata la scritta `Partenza!`. Altrimenti stampa `n` e poi chiama la funzione `ContoAllaRovescia` (cioè sé stessa) con un argomento che vale `n-1`.

Cosa succede quando chiamiamo una funzione come questa?

```
>>> ContoAllaRovescia(3)
```

L'esecuzione di `ContoAllaRovescia` inizia con `n=3`. Dato che `n` non è 0, essa stampa il valore 3, e poi richiama sé stessa...

> L'esecuzione di `ContoAllaRovescia` inizia con `n=2`. Dato che `n` non è 0, essa stampa il valore 2, poi richiama sé stessa...
>
> > L'esecuzione di `ContoAllaRovescia` inizia con `n=1`. Dato che `n` non è 0, essa stampa il valore 1, poi richiama sé stessa...
> >
> > > L'esecuzione di `ContoAllaRovescia` inizia con il valore di `n=0`. Dal momento che `n` è 0, essa stampa il testo "Partenza!" e poi ritorna.
> >
> > La funzione `ContoAllaRovescia` che aveva `n=1`; e poi ritorna.
>
> La funzione `ContoAllaRovescia` che aveva `n=2`; e poi ritorna.

E quindi torna in `__main__` (questo è un trucco). Il risultato è questo:

```
3
2
1
Partenza!
```

Come secondo esempio torniamo alle funzioni `UnaRigaVuota` e `TreRigheVuote`:

```
def UnaRigaVuota():
  print

def TreRigheVuote():
  UnaRigaVuota()
  UnaRigaVuota()
  UnaRigaVuota()
```

Sebbene funzionino correttamente non sarebbero di molto aiuto nel momento in cui vogliamo stampare due righe vuote o magari 106. Una alternativa migliore potrebbe essere questa:

```
def NRigheVuote(n):
  if n > 0:
    print
    NRigheVuote(n-1)
```

Questo programma è simile a `ContoAllaRovescia`: finché `n` è maggiore di 0, la funzione stampa una riga vuota e poi chiama sé stessa con un argomento `n` diminuito di 1.

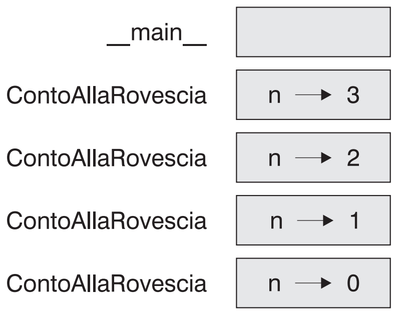
Il processo di una funzione che richiama sé stessa è detto **ricorsione**, e la funzione è definita ricorsiva.

4.10 Diagrammi di stack per funzioni ricorsive

Nella sezione 3.11, abbiamo usato un diagramma di stack per rappresentare lo stato di un programma durante una chiamata di funzione. Lo stesso tipo di diagramma può aiutare a capire come lavora una funzione ricorsiva.

Ogni volta che una funzione viene chiamata, Python crea un nuovo frame della funzione, contenente le variabili locali definite all'interno della funzione ed i suoi parametri. Nel caso di una funzione ricorsiva possono esserci più frame riguardanti una stessa funzione allo stesso tempo.

La figura mostra il diagramma dello stack della funzione `ContoAllaRovescia` chiamata con `n=3`:



Come al solito il livello superiore dello stack è il frame per `__main__`. Questo frame è vuoto perché in questo caso non abbiamo creato alcuna variabile locale e non abbiamo passato alcun parametro.

I quattro frame di `ContoAllaRovescia` hanno valori diversi per il parametro `n`. Il livello inferiore dello stack, quando `n=0`, è chiamato lo **stato di base**. Esso non effettua ulteriori chiamate ricorsive, così non ci sono ulteriori frame.

> *Esercizio: disegna il diagramma dello stack per la funzione* `NRigheVuote` *chiamata con* `n=4`*.*

4.11 Ricorsione infinita

Se una ricorsione non raggiunge mai il suo stato di base la chiamata alla funzione viene eseguita all'infinito ed in teoria il programma non giunge mai alla fine. Questa situazione è conosciuta come **ricorsione infinita** e non è generalmente considerata una buona cosa. Questo è un programma minimo che genera una ricorsione infinita:

```
def Ricorsione():
  Ricorsione()
```

Nella maggior parte degli ambienti un programma con una ricorsione infinita non viene eseguito senza fine, dato che ogni chiamata ad una funzione impegna un po' di memoria del computer e questa memoria prima o poi finisce. Python stampa un messaggio d'errore quando è stato raggiunto il massimo livello di ricorsione possibile:

```
  File "<stdin>", line 2, in Ricorsione
  ...
  File "<stdin>", line 2, in Ricorsione
RuntimeError: Maximum recursion depth exceeded
```

Questa traccia è un po' più lunga di quella che abbiamo visto nel capitolo precedente. Quando è capitato l'errore c'erano moltissime ricorsioni nello stack.

> *Esercizio: scrivi una funzione con ricorsione infinita ed eseguila nell'interprete Python.*

4.12 Inserimento da tastiera

I programmi che abbiamo scritto finora sono piuttosto banali, nel senso che non accettano inserimenti di dati da parte dell'operatore, limitandosi a eseguire sempre le stesse operazioni.

Python fornisce un insieme di funzioni predefinite che permettono di inserire dati da tastiera. La più semplice di esse è `raw_input`. Quando questa funzione è chiamata il programma si ferma ed attende che l'operatore inserisca qualcosa, confermando poi l'inserimento con Invio (o Enter). A quel punto il programma riprende e `raw_input` ritorna ciò che l'operatore ha inserito sotto forma di `stringa`:

```
>>> Inserimento = raw_input ()
Testo inserito
>>> print Inserimento
Testo inserito
```

Prima di chiamare `raw_input` è una buona idea stampare un messaggio che avvisa l'operatore di ciò che deve essere inserito. Questo messaggio è chiamato **prompt**. L'operazione è così comune che il messaggio di prompt può essere passato come argomento a `raw_input`:

```
>>> Nome = raw_input ("Qual e' il tuo nome? ")
Qual e' il tuo nome? Arturo
>>> print Nome
Arturo
```

Se il valore da inserire è un intero possiamo usare la funzione `input`:

```
Prompt = "A che velocita'viaggia il treno?\n"
Velocita = input(Prompt)
```

Se l'operatore inserisce una serie di cifre questa è convertita in un intero ed assegnata a `Velocita`. Sfortunatamente se i caratteri inseriti dall'operatore non rappresentano un numero, il programma stampa un messaggio d'errore e si blocca:

```
>>> Velocita = input (Prompt)
A che velocita'viaggia il treno?
ottanta all'ora
SyntaxError: invalid syntax
```

Per evitare questo tipo di errori è generalmente meglio usare la funzione `raw_input` per ottenere una stringa di caratteri e poi usare le funzioni di conversione per ottenere gli altri tipi.

4.13 Glossario

Operatore modulo: operatore matematico denotato con il segno di percentuale (`%`) che restituisce il resto della divisione tra due operandi interi.

Espressione booleana: espressione che è o vera o falsa.

Operatore di confronto: uno degli operatori che confrontano due valori: `==`, `!=`, `>`, `<`, `>=` e `<=`.

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

Istruzione condizionale: istruzione che controlla il flusso di esecuzione del programma a seconda del verificarsi di certe condizioni.

Condizione: espressione booleana in una istruzione condizionale che determina quale ramificazione debba essere seguita dal flusso di esecuzione.

Istruzione composta: istruzione che consiste di un'intestazione terminante con i due punti (:) e di un corpo composto di una o più istruzioni indentate rispetto all'intestazione.

Blocco: gruppo di istruzioni consecutive con la stessa indentazione.

Corpo: blocco che segue l'intestazione in un'istruzione composta.

Annidamento: particolare struttura di programma interna ad un'altra, come nel caso di una istruzione condizionale inserita all'interno di un'altra istruzione condizionale.

Ricorsione: richiamo di una funzione che è già in esecuzione.

Stato di base: ramificazione di un'istruzione condizionale posta in una funzione ricorsiva e che non esegue alcuna chiamata ricorsiva.

Ricorsione infinita: funzione che chiama sé stessa ricorsivamente senza mai raggiungere lo stato di base. L'occupazione progressiva della memoria che avviene ad ogni successiva chiamata causa ad un certo punto un errore in esecuzione.

Prompt: suggerimento visivo che specifica il tipo di dati atteso come inserimento da tastiera.

Capitolo 5

Funzioni produttive

5.1 Valori di ritorno

Alcune delle funzioni predefinite che abbiamo usato finora producono dei risultati: la chiamata della funzione con un particolare argomento genera un nuovo valore che viene in seguito assegnato ad una variabile o viene usato come parte di un'espressione.

```
e = math.exp(1.0)
Altezza = Raggio * math.sin(Angolo)
```

Nessuna delle funzioni che abbiamo scritto sino a questo momento ha ritornato un valore.

In questo capitolo scriveremo funzioni che ritornano un valore e che chiamiamo **funzioni produttive**. Il primo esempio è `AreaDelCerchio` che ritorna l'area di un cerchio per un dato raggio:

```
import math

def AreaDelCerchio(Raggio):
  temp = math.pi * Raggio**2
  return temp
```

Abbiamo già visto l'istruzione `return`, ma nel caso di una funzione produttiva questa istruzione prevede un **valore di ritorno**. Questa istruzione significa: "ritorna immediatamente da questa funzione a quella chiamante e usa questa espressione come valore di ritorno". L'espressione che rappresenta il valore di ritorno può essere anche complessa, così che l'esempio visto in precedenza può essere riscritto in modo più conciso:

```
def AreaDelCerchio(raggio):
  return math.pi * Raggio**2
```

D'altra parte una **variabile temporanea** come `temp` spesso rende il programma più leggibile e ne semplifica il debug.

Talvolta è necessario prevedere delle istruzioni di ritorno multiple, ciascuna all'interno di una ramificazione di un'istruzione condizionale:

```
def ValoreAssoluto(x):
  if x < 0:
    return -x
  else:
    return x
```

Dato che queste istruzioni `return` sono in rami diversi della condizione solo una di esse verrà effettivamente eseguita.

Il codice che è posto dopo un'istruzione `return`, o in ognuno dei posti dove non può essere raggiunto dal flusso di esecuzione, è denominato **codice morto**.

In una funzione produttiva è una buona idea assicurarci che ognuna delle ramificazioni possibili porti ad un'uscita dalla funzione con un'istruzione di `return`. Per esempio:

```
def ValoreAssoluto(x):
  if x < 0:
    return -x
  elif x > 0:
    return x
```

Questo programma non è corretto in quanto non è prevista un'uscita con `return` nel caso `x` sia 0. In questo caso il valore di ritorno è un valore speciale chiamato `None`:

```
>>> print ValoreAssoluto(0)
None
```

> *Esercizio: scrivi una funzione* `Confronto` *che ritorna* `1` *se* `x>y`, `0` *se* `x==y` *e* `-1` *se* `x<y`.

5.2 Sviluppo del programma

A questo punto sei già in grado di leggere funzioni complete e capire cosa fanno. Inoltre se hai fatto gli esercizi che ti ho suggerito hai già scritto qualche piccola funzione. A mano a mano che scriverai funzioni di complessità maggiore comincerai ad incontrare qualche difficoltà soprattutto con gli errori di semantica e di esecuzione.

Per fare fronte a questi programmi via via più complessi ti suggerisco una tecnica chiamata **sviluppo incrementale**. Lo scopo dello sviluppo incrementale è evitare lunghe sessioni di debug, aggiungendo e testando continuamente piccole parti di codice alla volta.

Come programma di esempio supponiamo che tu voglia trovare la distanza tra due punti conoscendone le coordinate (x_1, y_1) e (x_2, y_2). Con il teorema di Pitagora sappiamo che la distanza è

$$distanza = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1}$$

La prima cosa da considerare è l'aspetto che la funzione `DistanzaTraDuePunti` deve avere in Python chiarendo subito quali siano i parametri che si vogliono passare alla funzione e quale sia il risultato da ottenere: quest'ultimo può essere tanto un valore numerico da utilizzare all'interno di una espressione o da assegnare ad una variabile, tanto una stampa a video o altro.

Nel nostro caso è chiaro che le coordinate dei due punti sono i nostri parametri, e la distanza calcolata un valore numerico in virgola mobile.

Possiamo così delineare un primo abbozzo di funzione:

```
def DistanzaTraDuePunti(x1, y1, x2, y2):
  return 0.0
```

Ovviamente questa prima versione non calcola distanze, in quanto ritorna sempre 0. Ma è già una funzione sintatticamente corretta e può essere eseguita: è il caso di eseguire questo primo test prima di procedere a renderla più complessa.

Per testare la nuova funzione proviamo a chiamarla con dei semplici valori:

```
>>> DistanzaTraDuePunti(1, 2, 4, 6)
0.0
```

Abbiamo scelto questi valori così che la loro distanza orizzontale è 3 e quella verticale è 4. Con il teorema di Pitagora è facile vedere che il valore atteso è pari a 5 (5 è la lunghezza dell'ipotenusa di un triangolo rettangolo i cui cateti sono 3 e 4). Quando testiamo una funzione è sempre utile conoscere il risultato di qualche caso particolare per verificare se stiamo procedendo sulla strada giusta.

A questo punto abbiamo verificato che la funzione è sintatticamente corretta e possiamo così cominciare ad aggiungere linee di codice. Dopo ogni aggiunta la testiamo ancora per vedere che non ci siano problemi evidenti. Dovesse presentarsi un problema almeno sapremo che questo è dovuto alle linee inserite dopo l'ultimo test che ha avuto successo.

Un passo logico per risolvere il nostro problema è quello di trovare le differenze $x_2 - x_1$ e $y_2 - y_1$. Memorizzeremo queste differenze in variabili temporanee chiamate `dx` e `dy` e le stamperemo a video.

```
def DistanzaTraDuePunti(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  print "dx vale", dx
  print "dy vale", dy
  return 0.0
```

Se la funzione lavora correttamente, quando la richiamiamo con i valori di prima dovremmo trovare che `dx` e `dy` valgono rispettivamente 3 e 4. Se i risultati coincidono siamo sicuri che la funzione carica correttamente i parametri ed elabora altrettanto correttamente le prime righe. Nel caso il risultato non fosse quello atteso, dovremo concentrarci solo sulle poche righe aggiunte dall'ultimo test e non sull'intera funzione.

Proseguiamo con il calcolo della somma dei quadrati di `dx` e `dy`:

```
def DistanzaTraDuePunti(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  DistQuadrata = dx**2 + dy**2
  print "DistQuadrata vale ", DistQuadrata
  return 0.0
```

Nota come i due `print` che avevamo usato prima siano stati rimossi in quanto ci sono serviti per testare quella parte di programma ma adesso sarebbero inutili. Un codice come questo è chiamato **codice temporaneo** perché è utile durante la costruzione del programma ma alla fine deve essere rimosso in quanto non fa parte delle funzioni richieste alla versione definitiva della nostra funzione.

Ancora una volta eseguiamo il programma. Se tutto funziona dovremmo trovare un risultato pari a 25 (la somma dei quadrati costruiti sui cateti di lato 3 e 4).

Non ci resta che calcolare la radice quadrata. Se abbiamo importato il modulo matematico `math` possiamo usare la funzione `sqrt` per elaborare il risultato:

```
def DistanzaTraDuePunti(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  DistQuadrata = dx**2 + dy**2
  Risultato = math.sqrt(DistQuadrata)
  return Risultato
```

Stavolta se tutto va bene abbiamo finito. Potresti anche stampare il valore di `Risultato` prima di uscire dalla funzione con `return`.

Soprattutto all'inizio non dovresti mai aggiungere più di poche righe di programma alla volta. Man mano che la tua esperienza di programmatore cresce ti troverai a scrivere pezzi di codice sempre più grandi. In ogni caso nelle prime fasi il processo di sviluppo incrementale ti farà risparmiare un bel po' di tempo.

Ecco gli aspetti chiave del processo di sviluppo incrementale:

1. Inizia con un programma funzionante e fai piccoli cambiamenti: questo ti permetterà di scoprire facilmente dove siano localizzati gli eventuali errori.
2. Usa variabili temporanee per memorizzare i valori intermedi, così da poterli stampare e controllare.
3. Quando il programma funziona perfettamente rimuovi le istruzioni temporanee e consolida le istruzioni in espressioni composite, sempre che questo non renda il programma difficile da leggere.

Esercizio: usa lo sviluppo incrementale per scrivere una funzione chiamata `Ipotenusa` *che ritorna la lunghezza dell'ipotenusa di un triangolo rettangolo, passando i due cateti come parametri. Registra ogni passo del processo di sviluppo man mano che esso procede.*

5.3 Composizione

È possibile chiamare una funzione dall'interno di un'altra funzione. Questa capacità è chiamata **composizione**.

Scriveremo ora una funzione che accetta come parametri il centro ed un punto sulla circonferenza di un cerchio e calcola l'area del cerchio.

Il centro del cerchio è memorizzato nelle variabili `xc` e `yc` e le coordinate del punto sulla circonferenza in `xp` e `yp`. Il primo passo è trovare il raggio del cerchio, che è equivalente alla distanza tra i due punti: la funzione `DistanzaTraDuePunti` che abbiamo appena scritto servirà proprio a questo:

```
Raggio = DistanzaTraDuePunti(xc, yc, xp, yp)
```

Il secondo passo è trovare l'area del cerchio e restituirla:

```
Risultato = AreaDelCerchio(Raggio)
return Risultato
```

Assemblando il tutto in una funzione abbiamo:

```
def AreaDelCerchio2(xc, yc, xp, yp):
  Raggio = DistanzaTraDuePunti(xc, yc, xp, yp)
  Risultato = AreaDelCerchio(Raggio)
  return Risultato
```

Abbiamo chiamato questa funzione `AreaDelCerchio2` per distinguerla dalla funzione `AreaDelCerchio` definita in precedenza. Non possono esistere due funzioni con lo stesso nome all'interno di un modulo.

Le variabili temporanee `Raggio` e `Risultato` sono utili per lo sviluppo e il debug ma quando il programma funziona possiamo riscrivere la funzione in modo più conciso componendo le chiamate alle funzioni:

```
def AreaDelCerchio2(xc, yc, xp, yp):
  return AreaDelCerchio(DistanzaTraDuePunti(xc, yc, xp, yp))
```

> *Esercizio: scrivi una funzione* `Pendenza(x1, y1, x2, y2)` *che ritorna il valore della pendenza della retta passante per i punti* $(x1, y1)$ *e* $(x2, y2)$*. Poi usa questa funzione in una seconda funzione chiamata* `IntercettaY(x1, y1, x2, y2)` *che ritorna il valore delle ordinate quando la retta determinata dagli stessi punti ha X uguale a zero.*

5.4 Funzioni booleane

Le funzioni possono anche ritornare valori booleani (*vero* o *falso*) e questo è molto utile per mascherare al loro interno test anche complicati.

```
def Divisibile(x, y):
  if x % y == 0:
    return 1       # x e' divisibile per y: ritorna vero
  else:
    return 0       # x non e' divisibile per y: ritorna falso
```

Il nome di questa funzione è `Divisibile` (sarebbe comodo poterla chiamare `E'Divisibile` ma purtroppo gli accenti e le lettere accentate non sono caratteri validi nei nomi di variabili e di funzioni). È consuetudine assegnare dei nomi che sembrano domande con risposta si/no alle funzioni booleane: Divisibile? Bisestile? NumeroPari? Nel nostro caso `Divisibile` ritorna `1` o `0` per indicare se `x` è divisibile o meno per `y`. Vale il discorso già fatto in precedenza: 0 indica *falso*, qualsiasi valore diverso da 0 *vero*.

Possiamo rendere le funzioni ancora più concise avvantaggiandoci del fatto che la condizione nell'istruzione `if` è anch'essa di tipo booleano:

```
def Divisibile(x, y):
  return x%y == 0
```

Questa sessione mostra la nuova funzione in azione:

```
>>>   Divisibile(6, 4)
0
>>>   Divisibile(6, 3)
1
```

Le funzioni booleane sono spesso usate in istruzioni condizionali:

```
if Divisibile(x, y):
  print x, "e' divisibile per", y
else:
  print x, "non e' divisibile per", y
```

> *Esercizio: scrivi una funzione* `CompresoTra(x,y,z)` *che ritorna 1 se $y \leq x \leq z$, altrimenti ritorna 0.*

5.5 Ancora ricorsione

Finora hai imparato una piccola parte di Python, ma potrebbe interessarti sapere che questo sottoinsieme è già di per sé un linguaggio di programmazione *completo*: questo significa che con gli elementi che già conosci puoi esprimere qualsiasi tipo di elaborazione. Aggiungendo solo qualche comando di controllo per gestire tastiera, mouse, dischi, ecc. qualsiasi tipo di programma potrebbe già essere riscritto usando solo le caratteristiche del linguaggio che hai imparato finora.

La prova di questa affermazione è un esercizio non banale e fu dimostrata per la prima volta da Alan Turing, uno dei primi teorici dell'informatica (qualcuno potrebbe obiettare che in realtà era un matematico, ma molti degli informatici di allora erano dei matematici). Di conseguenza la dimostrazione è chiamata Teorema di Turing.

Per darti un'idea di che cosa puoi fare con ciò che hai imparato finora proveremo a valutare un po' di funzioni matematiche definite ricorsivamente. Una funzione ricorsiva è simile ad una definizione circolare, nel senso che la sua definizione contiene un riferimento alla cosa che viene definita. Una definizione circolare non è poi troppo utile, tanto che se ne trovassi una consultando un vocabolario ciò ti darebbe fastidio:

zurloso: aggettivo usato per descrivere qualcosa di zurloso.

D'altra parte se guardi la definizione della funzione matematica fattoriale (indicata da un numero seguito da un punto esclamativo) ti accorgi che la somiglianza è notevole:

$$0! = 1$$
$$n! = n(n-1)!$$

Questa definizione stabilisce che il fattoriale di 0 è 1 e che il fattoriale di ogni altro valore n è n moltiplicato per il fattoriale di $n-1$.

Così 3! è 3 moltiplicato 2!, che a sua volta è 2 moltiplicato 1!, che a sua volta è 1 moltiplicato 0!, che per definizione è 1. Mettendo tutto assieme 3! è uguale a 3 per 2 per 1, e cioè pari a 6.

Se scrivi una definizione ricorsiva, solitamente puoi anche scrivere un programma Python per valutarla. Il primo passo è quello di decidere quali siano i parametri da passare alla funzione.

`Fattoriale` ha un solo parametro:

```
def Fattoriale(n):
```

Se l'argomento è 0 dobbiamo ritornare il valore 1:

```
def Fattoriale(n):
  if n == 0:
    return 1
```

Altrimenti, e questa è la parte interessante, dobbiamo fare una chiamata ricorsiva per trovare il fattoriale di $n-1$ e poi moltiplicare questo valore per n:

```
def Fattoriale(n):
  if n == 0:
    return 1
  else:
    FattorialeMenoUno = Fattoriale(n-1)
    Risultato = n * FattorialeMenoUno
    return Risultato
```

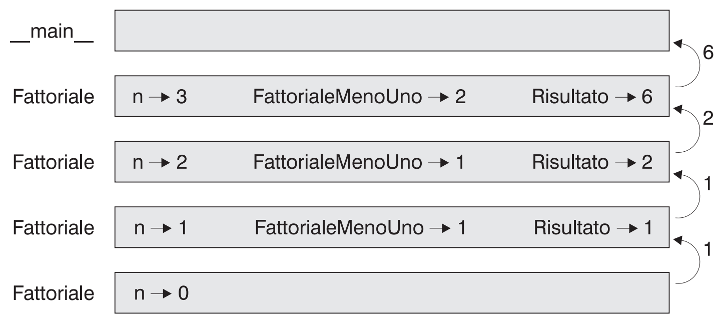
Il flusso di esecuzione del programma è simile a quello di `ContoAllaRovescia` nella sezione 4.9. Se chiamiamo `Fattoriale` con il valore 3:

Dato che 3 non è 0, seguiamo il ramo `else` e calcoliamo il fattoriale di `n=3-1=2`...

> Dato che 2 non è 0, seguiamo il ramo `else` e calcoliamo il fattoriale di `n=2-1=1`...
>
> > Dato che 1 non è 0, seguiamo il ramo `else` e calcoliamo il fattoriale di `n=1-1=0`...
> >
> > > Dato che 0 *è* 0 ritorniamo 1 senza effettuare ulteriori chiamate ricorsive.
> >
> > Il valore di ritorno (1) è moltiplicato per n (1) e il risultato (1) restituito alla funzione chiamante.
>
> Il valore di ritorno (1) è moltiplicato per n (2) e il risultato (2) restituito alla funzione chiamante.

Il valore di ritorno (2) è moltiplicato per n (3) e il risultato (6) diventa il valore di ritorno della funzione che ha fatto partire l'intero processo.

Questo è il diagramma di stack per l'intera serie di funzioni:



I valori di ritorno sono mostrati mentre vengono passati di chiamata in chiamata verso l'alto. In ogni frame il valore di ritorno è `Risultato`, che è il prodotto di `n` per `FattorialeMenoUno`.

Nota come nell'ultimo frame le variabili locali `FattorialeMenoUno` e `Risultato` non esistono, perché il ramo che le crea non viene eseguito.

5.6 Accettare con fiducia

Seguire il flusso di esecuzione è un modo di leggere i programmi, ma può dimostrarsi piuttosto difficile da seguire man mano che le dimensioni del codice aumentano. Un modo alternativo è ciò che potremmo chiamare *accettazione con fiducia*: quando arrivi ad una chiamata di funzione invece di seguire il flusso di esecuzione *parti dal presupposto* che la funzione chiamata si comporti correttamente e che ritorni il valore che ci si attende.

In ogni modo stai già praticando questa *accettazione con fiducia* quando usi le funzioni predefinite: quando chiami `math.cos` o `math.exp` non vai a controllare l'implementazione delle funzioni, assumendo che chi le ha scritte fosse un buon programmatore e che le funzioni siano corrette.

Lo stesso si può dire per le funzioni che scrivi tu stesso: quando abbiamo scritto la funzione `Divisibile`, che controlla se un numero è divisibile per un altro, e abbiamo verificato che la funzione è corretta controllando il codice possiamo usarla senza doverla ricontrollare ancora.

Quando hai chiamate ricorsive invece di seguire il flusso di programma puoi partire dal presupposto che la chiamata ricorsiva funzioni (producendo il risultato corretto) chiedendoti in seguito: "Supponendo che si riesca a trovare il fattoriale di $n - 1$, posso calcolare il fattoriale di n?" In questo caso è chiaro che puoi farlo moltiplicandolo per n. È certamente strano partire dal presupposto che una funzione lavori correttamente quando non è ancora stata finita, non è vero?

5.7 Un esempio ulteriore

Nell'esempio precedente abbiamo usato delle variabili temporanee per identificare meglio i singoli passaggi e per facilitare la lettura del codice, ma avremmo potuto risparmiare qualche riga:

```
def Fattoriale(n):
  if n == 0:
    return 1
  else:
    return n * Fattoriale(n-1)
```

D'ora in poi in questo libro tenderemo ad usare la forma più concisa, ma ti consiglio di usare quella più esplicita finché non avrai un po' di esperienza nello sviluppo del codice.

Dopo il `Fattoriale`, l'esempio di funzione ricorsiva più comune è la funzione `Fibonacci` che ha questa definizione:

$$\begin{aligned}
&fibonacci(0) = 1\\
&fibonacci(1) = 1\\
&fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);
\end{aligned}$$

Tradotta in Python:

```
def Fibonacci (n):
  if n == 0 or n == 1:
    return 1
  else:
    return Fibonacci(n-1) + Fibonacci(n-2)
```

Con una funzione del genere il flusso di esecuzione diventa praticamente impossibile da seguire anche per piccoli valori di n. In questo caso ed in casi analoghi vale la pena di adottare l'accettazione con fiducia partendo dal presupposto che le due chiamate ricorsive funzionino correttamente e che quindi la somma dei loro valori di ritorno sia corretta.

5.8 Controllo dei tipi

Cosa succede se chiamiamo `Fattoriale` e passiamo 1.5 come argomento?

```
>>> Fattoriale(1.5)
RuntimeError: Maximum recursion depth exceeded
```

A prima vista sembra una ricorsione infinita. Ma come può accadere? C'è un caso base (quando `n==0`) che dovrebbe fermare la ricorsione, ma il problema è che non tutti i possibili valori di `n` verificano la condizione di fermata prevista dal caso base.

Se proviamo a seguire il flusso di esecuzione, alla prima chiamata il valore di `n` passa a 0.5. Alla successiva diventa -0.5. Da lì in poi, sottraendo 1 di volta in volta, il valore passato alla funzione è sempre più piccolo ma non sarà mai lo 0 che ci aspettiamo nel caso base.

Abbiamo due scelte: possiamo generalizzare la funzione `Fattoriale` per farla lavorare anche nel caso di numeri in virgola mobile, o possiamo far controllare alla funzione dopo la sua chiamata se il parametro passato è del tipo corretto. La prima possibilità è chiamata in matematica *funzione gamma* (il fattoriale definito nei numeri reali) ed è decisamente al di là degli scopi di questo libro, così sceglieremo la seconda alternativa.

Possiamo usare `type` per controllare se il parametro è di tipo intero. Già che ci siamo mettiamo anche un controllo per essere sicuri che il numero sia positivo:

```
def Fattoriale(n):
  if type(n) != type(1):
    print "Il fattoriale è definito solo per i valori interi."
    return -1
  elif n < 0:
    print "Il fattoriale è definito solo per interi positivi."
    return -1
  elif n == 0:
    return 1
  else:
    return n * Fattoriale(n-1)
```

Nel primo confronto abbiamo confrontato il "tipo di `n`" con il "tipo del numero intero 1" per vedere se `n` è intero.

Ora abbiamo tre casi: il primo blocca i valori non interi; il secondo gli interi negativi ed il terzo calcola il fattoriale di un numero che a questo punto è sicuramente un intero positivo o uguale a zero. Nei primi due casi dato che il calcolo non è possibile viene stampato un messaggio d'errore e la funzione ritorna il valore -1, per indicare che qualcosa non ha funzionato:

```
>>> Fattoriale("AAA")
Il fattoriale è definito solo per i valori interi.
-1
>>> Fattoriale (-2)
Il fattoriale è definito solo per gli interi positivi.
```

```
-1
```

Se il flusso di programma passa attraverso entrambi i controlli siamo certi che `n` è un intero positivo e sappiamo che la ricorsione avrà termine.

Questo programma mostra il funzionamento di una **condizione di guardia**. I primi due controlli agiscono da "guardiani", proteggendo il codice che segue da circostanze che potrebbero causare errori. Le condizioni di guardia rendono possibile provare la correttezza del codice in modo estremamente semplice ed affidabile.

5.9 Glossario

Funzione produttiva: funzione che produce un valore.

Valore di ritorno: valore restituito da una funzione.

Variabile temporanea: variabile usata per memorizzare un risultato intermedio durante un calcolo complesso.

Codice morto: parte di un programma che non può mai essere eseguita, spesso perché compare dopo un'istruzione di `return`.

Valore `None`: valore speciale ritornato da una funzione che non ha un'istruzione `return`, o se l'istruzione `return` non specifica un valore di ritorno.

Sviluppo incrementale: sistema di sviluppo del programma inteso ad evitare lunghe sessioni di debug alla ricerca degli errori aggiungendo e testando solo piccole porzioni di codice alla volta.

Codice temporaneo: codice inserito solo nella fase di sviluppo del programma e che non è richiesto nella versione finale.

Condizione di guardia: condizione che controlla e gestisce le circostanze che possono causare un errore.

Capitolo 6

Iterazione

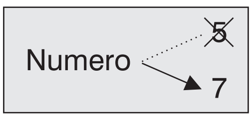
6.1 Assegnazione e confronto

Come puoi avere già scoperto è possibile assegnare più valori ad una stessa variabile, con la variabile che assume sempre l'ultimo valore assegnato:

```
Numero = 5
print Numero,
Numero = 7
print Numero
```

La stampa di questo programma è `5 7`, perché la prima volta che `Numero` è stampato il suo valore è 5, la seconda 7. La virgola dopo la prima istruzione `print` evita il ritorno a capo dopo la stampa così che entrambi i valori appaiono sulla stessa riga.

Questo è il diagramma di stato per quest'assegnazione:



Nel caso di assegnazioni ripetute è particolarmente importante distinguere tra operazioni di assegnazione e controlli di uguaglianza. Python usa (`=`) per l'assegnazione e si potrebbe essere tentati di interpretare l'istruzione `a = b` come un controllo di equivalenza, ma non lo è!

In primo luogo l'equivalenza è commutativa mentre l'assegnazione non lo è: in matematica se $a = 7$ allora $7 = a$; in Python l'istruzione `a=7` è legale mentre `7=a` produce un errore di sintassi.

Inoltre in matematica un'uguaglianza è sempre vera: se $a = b$, a sarà sempre uguale a b. In Python un'assegnazione può rendere due variabili uguali ma raramente l'uguaglianza sarà mantenuta a lungo:

```
a = 5
b = a    # a e b sono uguali
a = 3    # ora a e b sono diversi
```

La terza riga cambia il valore di `a` ma non cambia il valore di `b`. In qualche linguaggio di programmazione sono usati simboli diversi per l'assegnazione, tipo `<-` o `:=`, per evitare ogni malinteso.

6.2 L'istruzione `while`

I computer sono spesso usati per automatizzare compiti ripetitivi: il noiosissimo compito di ripetere operazioni identiche o simili un gran numero di volte senza fare errori è qualcosa che riesce bene ai computer.

Abbiamo visto due programmi, `NRigheVuote` e `ContoAllaRovescia`, che usano la ricorsione per eseguire una ripetizione. Questa ripetizione è più comunemente chiamata **iterazione**. Dato che l'iterazione è così comune, Python fornisce vari sistemi per renderla più semplice da implementare. Il primo sistema è l'istruzione `while`.

Ecco come `ContoAllaRovescia` viene riscritto usando l'istruzione `while`:

```
def ContoAllaRovescia(n):
  while n > 0:
    print n
    n = n-1
  print "Partenza!"
```

La chiamata ricorsiva è stata rimossa e quindi questa funzione ora non è più ricorsiva.

Puoi leggere il programma con l'istruzione `while` come fosse scritto in un linguaggio naturale: "Finché (while) `n` è più grande di 0 stampa il valore di `n` e poi diminuiscilo di 1. Quando arrivi a 0 stampa la stringa `Partenza!`".

In modo più formale ecco il flusso di esecuzione di un'istruzione `while`:

1. Valuta la condizione controllando se essa è vera (`1`) o falsa (`0`).

2. Se la condizione è falsa esci dal ciclo `while` e continua l'esecuzione dalla prima istruzione che lo segue.

3. Se la condizione è vera esegui tutte le istruzioni nel corpo del `while` e torna al passo 1.

Il corpo del ciclo `while` consiste di tutte le istruzioni che seguono l'intestazione e che hanno la stessa indentazione.

Questo tipo di flusso è chiamato **ciclo** o loop. Nota che se la condizione è falsa al primo controllo, le istruzioni del corpo non sono mai eseguite.

Il corpo del ciclo dovrebbe cambiare il valore di una o più variabili così che la condizione possa prima o poi diventare falsa e far così terminare il ciclo. In caso contrario il ciclo si ripeterebbe all'infinito, determinando un **ciclo infinito**.

Nel caso di `ContoAllaRovescia` possiamo essere certi che il ciclo è destinato a terminare visto che `n` è finito ed il suo valore diventa via via più piccolo così da diventare, prima o poi, pari a zero. In altri casi può non essere così facile stabilire se un ciclo avrà termine:

```
def Sequenza(n):
  while n != 1:
    print n,
    if n%2 == 0:        # se n e' pari
      n = n/2
    else:               # se n e' dispari
      n = n*3+1
```

La condizione per questo ciclo è `n!=1` cosicché il ciclo si ripeterà finché `n` è diverso da `1`.

Ogni volta che viene eseguito il ciclo il programma stampa il valore di `n` e poi controlla se è pari o dispari. Se è pari, `n` viene diviso per 2. Se dispari, è moltiplicato per 3 e gli viene sommato 1. Se il valore passato è 3, la sequenza risultante è 3, 10, 5, 16, 8, 4, 2, 1.

Dato che `n` a volte sale e a volte scende in modo abbastanza casuale non c'è una prova ovvia che `n` raggiungerà 1 in modo da far terminare il ciclo. Per qualche particolare valore di `n` possiamo facilmente determinare a priori il suo termine (per esempio per le potenze di 2) ma per gli altri nessuno è mai riuscito a trovare la dimostrazione che il ciclo ha termine.

> *Esercizio: riscrivi la funzione* `NRigheVuote` *della sezione 4.9 usando un'iterazione invece che la ricorsione.*

6.3 Tabelle

Una delle cose per cui sono particolarmente indicati i cicli è la generazione di tabulati. Prima che i computer fossero comunemente disponibili si dovevano calcolare a mano logaritmi, seni, coseni e i valori di tante altre funzioni matematiche. Per rendere più facile il compito i libri di matematica contenevano lunghe tabelle di valori la cui stesura comportava enormi quantità di lavoro molto noioso e grosse possibilità di errore.

Quando apparvero i computer l'idea iniziale fu quella di usarli per generare tabelle prive di errori. La cosa che non si riuscì a prevedere fu il fatto che i computer sarebbero diventati così diffusi e disponibili a tutti da rendere quei lunghi tabulati cartacei del tutto inutili. Per alcune operazioni i computer usano ancora tabelle simili in modo del tutto nascosto dall'operatore: vengono usate per ottenere risposte approssimate che poi vengono rifinite per migliorarne la precisione. In qualche caso ci sono stati degli errori in queste tabelle "interne", il più famoso dei quali ha avuto come protagonista il Pentium Intel con un errore nel calcolo delle divisioni in virgola mobile.

Sebbene la tabella dei logaritmi non sia più utile come lo era in passato rimane tuttavia un buon esempio di iterazione. Il programma seguente stampa una sequenza di valori nella colonna di sinistra e il loro logaritmo in quella di destra:

```
x = 1.0
while x < 10.0:
  print x, '\t', math.log(x)
  x = x + 1.0
```

La stringa `'\t'` rappresenta un carattere di **tabulazione**.

A mano a mano che caratteri e stringhe sono mostrati sullo schermo un marcatore invisibile chiamato **cursore** tiene traccia di dove andrà stampato il carattere successivo. Dopo un'istruzione `print` il cursore normalmente si posiziona all'inizio della riga successiva.

Il carattere di tabulazione sposta il cursore a destra finché quest'ultimo raggiunge una delle posizione di stop delle tabulazioni. Queste posizioni si ripetono a distanze regolari, tipicamente ogni 4 o 8 caratteri. Le tabulazioni sono utili per allineare in modo semplice le colonne di testo. Ecco il prodotto del programma appena visto:

```
1.0     0.0
2.0     0.69314718056
3.0     1.09861228867
4.0     1.38629436112
5.0     1.60943791243
6.0     1.79175946923
7.0     1.94591014906
8.0     2.07944154168
9.0     2.19722457734
```

Se questi valori sembrano strani ricorda che la funzione `log` usa il logaritmo dei numeri naturali e. Dato che le potenze di due sono così importanti in informatica possiamo avere la necessità di calcolare il logaritmo in base 2. Per farlo usiamo questa formula:

$$\log_2 x = \frac{log_e x}{log_e 2} \tag{6.1}$$

Modificando una sola riga di programma:

```
   print x, '\t',  math.log(x)/math.log(2.0)
```

otteniamo:

```
1.0     0.0
2.0     1.0
3.0     1.58496250072
```

```
4.0     2.0
5.0     2.32192809489
6.0     2.58496250072
7.0     2.80735492206
8.0     3.0
9.0     3.16992500144
```

Possiamo vedere che 1, 2, 4 e 8 sono potenze di due perché i loro logaritmi in base 2 sono numeri interi.

Per continuare con le modifiche, invece di sommare qualcosa a `x` ad ogni ciclo e ottenere così una serie aritmetica, possiamo moltiplicare `x` per qualcosa ottenendo una serie geometrica. Se vogliamo trovare il logaritmo di altre potenze di due possiamo modificare ancora il programma:

```
x = 1.0
while x < 100.0:
  print x, '\t', math.log(x)/math.log(2.0)
  x = x * 2.0
```

Il risultato in questo caso è:

```
1.0     0.0
2.0     1.0
4.0     2.0
8.0     3.0
16.0    4.0
32.0    5.0
64.0    6.0
```

Il carattere di tabulazione fa in modo che la posizione della seconda colonna non dipenda dal numero di cifre del valore nella prima.

Anche se i logaritmi possono non essere più così utili per un informatico, conoscere le potenze di due è fondamentale.

> *Esercizio: modifica questo programma per fare in modo che esso produca le potenze di due fino a 65536 (cioè 2^{16}). Poi stampale e imparale a memoria!*

Il carattere di backslash `'\'` in `'\t'` indica l'inizio di una **sequenza di escape**. Le sequenze di escape sono usate per rappresentare caratteri invisibili come la tabulazione (`'\t'`) e il ritorno a capo (`'\n'`). Può comparire in qualsiasi punto di una stringa: nell'esempio appena visto la tabulazione è l'unica cosa presente nella stringa del print.

Secondo te, com'è possibile rappresentare un carattere di backslash in una stringa?

> *Esercizio: scrivi una stringa singola che quando stampata*

```
produca
        questo
                risultato.
```

6.4 Tabelle bidimensionali

Una tabella bidimensionale è una tabella dove leggi un valore all'intersezione tra una riga ed una colonna, la tabella della moltiplicazione ne è un buon esempio. Immaginiamo che tu voglia stampare la tabella della moltiplicazione per i numeri da 1 a 6.

Un buon modo per iniziare è scrivere un ciclo che stampa i multipli di 2 tutti su di una stessa riga:

```
i = 1
while i <= 6:
  print 2*i, '   ',
  i = i + 1
print
```

La prima riga inizializza una variabile chiamata `i` che agisce come contatore o **indice del ciclo**. Man mano che il ciclo viene eseguito `i` passa da 1 a 6. Quando `i` è 7 la condizione non è più soddisfatta ed il ciclo termina. Ad ogni ciclo viene mostrato il valore di `2*i` seguito da tre spazi.

Ancora una volta vediamo come la virgola in `print` faccia in modo che il cursore rimanga sulla stessa riga evitando un ritorno a capo. Quando il ciclo sui sei valori è stato completato una seconda istruzione `print` ha lo scopo di portare il cursore a capo su una nuova riga.

Il risultato del programma è:

```
2       4       6       8       10      12
```

6.5 Incapsulamento e generalizzazione

L'`incapsulamento` è il processo di inserire un pezzo di codice all'interno di una funzione così da permetterti di godere dei vantaggi delle funzioni. Hai già visto due esempi di incapsulamento: `StampaParita` nella sezione 4.5 e `Divisibile` nella sezione 5.4.

`Generalizzare` significa prendere qualcosa di specifico per farlo diventare generale: nel nostro caso prendere il programma che calcola i multipli di 2 e fargli calcolare i multipli di un qualsiasi numero intero.

Questa funzione incapsula il ciclo visto in precedenza e lo generalizza per stampare i primi 6 multipli di `n`:

```
def StampaMultipli(n):
  i = 1
  while i <= 6:
    print n*i, '\t',
    i = i + 1
  print
```

Per incapsulare dobbiamo solo aggiungere la prima linea che dichiara il nome della funzione e la lista dei parametri. Per generalizzare dobbiamo sostituire il valore 2 con il parametro `n`.

Se chiamiamo la funzione con l'argomento 2 otteniamo lo stesso risultato di prima. Con l'argomento 3 il risultato è:

```
3       6       9       12      15      18
```

Con l'argomento 4:

```
4       8       12      16      20      24
```

Avrai certamente indovinato come stampare la tabella della moltiplicazione. Chiamiamo ripetutamente `StampaMultipli` con argomenti diversi all'interno di un secondo ciclo:

```
i = 1
while i <= 6:
  StampaMultipli(i)
  i = i + 1
```

Nota come siano simili questo ciclo e quello all'interno di `StampaMultipli`: tutto quello che abbiamo fatto è stato sostituire l'istruzione `print` con una chiamata di funzione.

Il risultato di questo programma è la tabella della moltiplicazione:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

6.6 Ancora incapsulamento

Per provare ancora con l'incapsulamento andiamo a prendere il codice della sezione precedente e inseriamolo in una funzione:

```
def TabellaMoltiplicazione6x6():
  i = 1
  while i <= 6:
    StampaMultipli(i)
    i = i + 1
```

Il processo appena illustrato è un **piano di sviluppo** piuttosto comune: si sviluppa del codice controllando poche righe in ambiente interprete; solo quando queste righe sono perfettamente funzionanti le inseriamo in una funzione.

Questo modo di procedere è particolarmente utile se all'inizio della stesura del tuo programma non sai come lo dividerai in funzioni. Questo tipo di approccio ti permette di progettare il codice mentre procedi con la stesura.

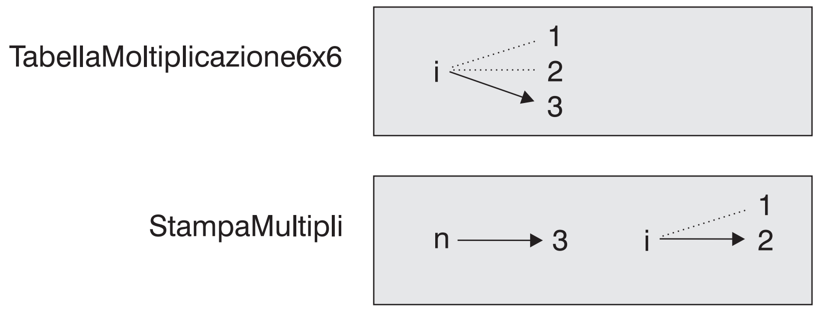
6.7 Variabili locali

Potresti chiederti com'è possibile che si possa usare la stessa variabile `i` sia in `StampaMultipli` che in `TabellaMoltiplicazione6x6`. Non ci sono problemi quando una funzione cambia il valore della variabile?

La risposta è no dato che la variabile `i` usata in `StampaMultipli` e la `i` in `TabellaMoltiplicazione6x6` *non sono* la stessa variabile.

Le variabili create all'interno della definizione di una funzione sono locali e non puoi accedere al valore di una variabile locale al di fuori della funzione che la ospita. Ciò significa che sei libero di avere variabili con lo stesso nome sempre che non si trovino all'interno di una stessa funzione.

Il diagramma di stack per questo programma mostra che le due variabili chiamate `i` non sono la stessa variabile. Si riferiscono a valori diversi e cambiandone una l'altra resta invariata.



Il valore di `i` in `TabellaMoltiplicazione6x6` va da 1 a 6. Nel diagramma ha valore 3 e al prossimo ciclo varrà 4. Ad ogni ciclo `TabellaMoltiplicazione6x6` chiama `StampaMultipli` con il valore corrente di `i` come argomento. Quel valore viene assegnato al parametro `n`.

All'interno di `StampaMultipli` il valore di `i` copre l'intervallo che va da 1 a 6. Nel diagramma è 2 e cambiandolo non ci sono effetti collaterali per la variabile `i` in `TabellaMoltiplicazione6x6`.

È comune e perfettamente legale avere variabili locali con lo stesso nome. In particolare nomi come `i` e `j` sono usati frequentemente come indici per i cicli.

6.8 Ancora generalizzazione

Se vogliamo generalizzare ulteriormente `TabellaMoltiplicazione6x6` potremmo estendere il risultato ad una tabella di moltiplicazione di qualsiasi grandezza, e non solo fino al 6x6. A questo punto dobbiamo anche passare un argomento per stabilire la grandezza desiderata:

```
def TabellaMoltiplicazioneGenerica(Grandezza):
  i = 1
  while i <= Grandezza:
    StampaMultipli(i)
    i = i + 1
```

Abbiamo sostituito il valore 6 con il parametro `Grandezza`. Se chiamiamo `TabellaMoltiplicazioneGenerica` con l'argomento 7 il risultato è:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
7       14      21      28      35      42
```

Il risultato è corretto fatta eccezione per il fatto che sarebbe meglio avere lo stesso numero di righe e di colonne. Per farlo dobbiamo modificare `StampaMultipli` per specificare quante colonne la tabella debba avere.

Tanto per essere originali chiamiamo anche questo parametro `Grandezza`, dimostrando ancora una volta che possiamo avere parametri con lo stesso nome all'interno di funzioni diverse. Ecco l'intero programma:

```
def StampaMultipli(n, Grandezza):
  i = 1
  while i <= Grandezza:
    print n*i, '\t',
    i = i + 1
  print

def TabellaMoltiplicazioneGenerica(Grandezza):
  i = 1
  while i <= Grandezza:
    StampaMultipli(i, Grandezza)
    i = i + 1
```

Quando abbiamo aggiunto il nuovo parametro abbiamo cambiato la prima riga della funzione (l'intestazione) ed il posto dove la funzione è chiamata in `TabellaMoltiplicazioneGenerica`.

Questo programma genera correttamente la tabella 7x7:

```
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
```

Quando generalizzi una funzione nel modo più appropriato, spesso ottieni capacità che inizialmente non erano state previste. Per esempio dato che $ab = ba$, tutti i numeri compresi nella tabella (fatta eccezione per quelli della diagonale) sono presenti due volte. In caso di necessità puoi modificare una linea in `TabellaMoltiplicazioneGenerica` per stamparne solo metà. Cambia :

```
    StampaMultipli(i, Grandezza)
```

in

```
    StampaMultipli(i, i)
```

per ottenere

```
1
2       4
3       6       9
4       8       12      16
5       10      15      20      25
6       12      18      24      30      36
7       14      21      28      35      42      49
```

> *Esercizio: il compito consiste nel tracciare l'esecuzione di questa versione* `TabellaMoltiplicazioneGenerica` *e cerca di capire come funziona.*

6.9 Funzioni

Abbiamo già menzionato i motivi per cui è consigliato l'uso delle funzioni, senza però entrare nel merito. Adesso ti starai chiedendo a che cosa ci stessimo riferendo. Eccone qualcuno:

- Dare un nome ad una sequenza di istruzioni per rendere il tuo programma più semplice da leggere e correggere.
- Dividere un grosso programma in tante piccole parti che possono essere testate singolarmente e poi ricomposte in un tutto unico.
- Facilitare sia la ricorsione che l'iterazione.
- Riutilizzare parti di programma: quando una funzione è stata scritta e testata può essere riutilizzata anche in altri programmi.

6.10 Glossario

Assegnazione ripetuta: assegnazione alla stessa variabile di più valori nel corso del programma.

Iterazione: ripetizione di una serie di istruzioni usando una funzione ricorsiva o un ciclo.

Ciclo: istruzione o gruppo di istruzioni che vengono eseguite ripetutamente finché è soddisfatta una condizione.

Ciclo infinito: ciclo nel quale la condizione di terminazione non è mai soddisfatta.

Corpo: gruppo di istruzioni all'interno di un ciclo.

Indice del ciclo: variabile usata nella condizione di terminazione di un ciclo.

Tabulazione: carattere speciale (`'\t'`) che in un'istruzione di stampa sposta il cursore alla prossima posizione di stop nella riga corrente.

Ritorno a capo: carattere speciale (`'\n'`) che in un'istruzione di stampa sposta il cursore all'inizio della prossima riga.

Cursore: marcatore non visibile che tiene traccia di dove andrà stampato il prossimo carattere.

Sequenza di escape: carattere (\\) seguito da uno o più caratteri, usato per designare dei caratteri non stampabili.

Incapsulare: dividere un programma complesso in componenti più semplici, tipo le funzioni, e isolarne i componenti uno dall'altro usando variabili locali.

Generalizzare: sostituire qualcosa di specifico (come un valore costante) con qualcosa di più generale (come un parametro o una variabile).

Piano di sviluppo: processo per lo sviluppo di un programma. In questo capitolo abbiamo mostrato uno stile di sviluppo basato sulla scrittura di un semplice programma capace di svolgere un compito specifico, per poi estenderlo con l'incapsulamento e la generalizzazione.

Capitolo 7

Stringhe

7.1 Tipi di dati composti

Finora abbiamo visto tre tipi di dati: `int`, `float` e `string`. Le stringhe sono qualitativamente diverse dagli altri tipi di dati poiché sono composte di unità più piccole: i caratteri.

I tipi di dati che sono fatti di elementi più piccoli sono detti **tipi di dati composti**. A seconda di ciò che stiamo facendo possiamo avere la necessità di trattare un tipo composto come fosse una singola entità o possiamo voler agire sulle sue singole parti. Questa duplice possibilità è molto utile.

L'operatore porzione seleziona dei caratteri da una stringa:

```
>>> Frutto = "banana"
>>> Lettera = Frutto[1]
>>> print Lettera
```

L'espressione `Frutto[1]` seleziona il carattere numero 1 dalla stringa `Frutto`. La variabile `Lettera` contiene il risultato. Quando stampiamo `Lettera` abbiamo però una sorpresa:

```
a
```

La prima lettera di `"banana"` logicamente non è `"a"`: in informatica i conteggi partono spesso da 0 e non da 1 come potrebbe sembrare normale e per accedere al primo carattere di una stringa dobbiamo quindi richiedere il numero 0, per il secondo il numero 1 e così via. Sembra un po' illogico ma ci farai facilmente l'abitudine perché questo è il modo normale di contare in molti linguaggi di programmazione. Quindi se vogliamo sapere l'iniziale della stringa scriviamo:

```
>>> Lettera = Frutto[0]
>>> print Lettera
b
```

L'espressione tra parentesi quadrate è chiamata **indice**. Un indice identifica un particolare elemento di un insieme ordinato che nel nostro caso è l'insieme dei caratteri di una stringa. L'indice può essere una qualsiasi espressione intera.

7.2 Lunghezza

La funzione `len` ritorna il numero di caratteri di una stringa:

```
>>> Frutto = "banana"
>>> len(Frutto)
6
```

Per ottenere l'ultimo carattere di una stringa potresti essere tentato di fare qualcosa di simile a:

```
Lunghezza = len(Frutto)
Ultimo = Frutto[Lunghezza]        # ERRORE!
```

ma c'è qualcosa che non va: infatti ottieni un errore `IndexError: string index out of range` dato che stai facendo riferimento all'indice 6 quando quelli validi vanno da 0 a 5. Per ottenere l'ultimo carattere dovrai quindi scrivere:

```
Lunghezza = len(Frutto)
Ultimo = Frutto[Lunghezza-1]
```

In alternativa possiamo usare indici negativi che in casi come questo sono più comodi, contando a partire dalla fine della stringa: l'espressione `Frutto[-1]` ritorna l'ultimo carattere della stringa, `Frutto[-2]` il penultimo e così via.

7.3 Elaborazione trasversale e cicli `for`

Molti tipi di elaborazione comportano un'azione su una stringa un carattere per volta. Spesso queste elaborazioni iniziano dal primo carattere, selezionano un carattere per volta e continuano fino al completamento della stringa. Questo tipo di elaborazione è definita **elaborazione trasversale** o **attraversamento**, in quanto attraversa la stringa dall'inizio alla fine. Un modo per implementare un'elaborazione trasversale è quello di usare un ciclo `while`:

```
Indice = 0
while Indice < len(Frutto):
  Lettera = Frutto[Indice]
  print Lettera
  Indice = Indice + 1
```

Questo ciclo attraversa la stringa e ne mostra una lettera alla volta, una per riga. La condizione del ciclo è `Indice < len(Frutto)` così che quando `Indice` è uguale alla lunghezza della stringa la condizione diventa falsa, il corpo del ciclo non è eseguito ed il ciclo termina. L'ultimo carattere cui si accede è quello con indice `len(Frutto)-1` che è l'ultimo carattere della stringa.

> *Esercizio: scrivi una funzione che prende una stringa come argomento e la stampa un carattere per riga partendo dall'ultimo carattere.*

Usare un indice per attraversare un insieme di valori è un'operazione così comune che Python fornisce una sintassi ancora più semplice: il ciclo `for`.

```
for Lettera in Frutto:
  print Lettera
```

Ad ogni ciclo, `Lettera` assume il valore del prossimo carattere della stringa `Frutto`, così che `Frutto` viene attraversata completamente finché non rimangono più caratteri da analizzare.

L'esempio seguente mostra come usare il concatenamento e un ciclo `for` per generare una serie alfabetica, e cioè una lista di valori nei quali gli elementi appaiono in ordine alfabetico. Per esempio nel libro *Make Way for Ducklings* di Robert McCloskey i nomi dei protagonisti sono Jack, Kack, Lack, Mack, Nack, Ouack, Pack e Quack. Questo ciclo restituisce i nomi in ordine:

```
Prefissi = "JKLMNOPQ"
Suffisso = "ack"

for Lettera in Prefissi:
  print Lettera + Suffisso
```

Il risultato del programma è:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Non è del tutto corretto dato che `Ouack` e `Quack` sono scritti in modo errato.

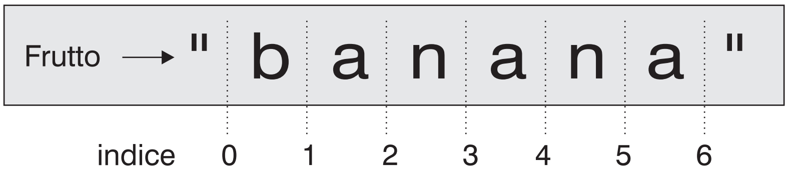
Esercizio: modifica il programma per correggere questo errore.

7.4 Porzioni di stringa

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

```
>>> s = "Pietro, Paolo e Maria"
>>> print s[0:6]
Pietro
>>> print s[8:13]
Paolo
>>> print s[16:21]
Maria
```

L'operatore `[n:m]` ritorna la stringa a partire dall' "n-esimo" carattere incluso fino all' "m-esimo" escluso. Questo comportamento non è intuitivo, e per comprenderlo è meglio immaginare i puntatori *tra* i caratteri, come nel diagramma seguente:



Se non è specificato il primo indice (prima dei due punti :) la porzione parte dall'inizio della stringa. Senza il secondo indice la porzione finisce con il termine della stringa:

```
>>> Frutto = "banana"
>>> Frutto[:3]
'ban'
>>> Frutto[3:]
'ana'
```

Secondo te cosa significa `Frutto[:]`?

7.5 Confronto di stringhe

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

```
if Parola == "BANANA":
  print  "stai parlando di un frutto!"
```

Altri operatori di confronto sono utili per mettere le parole in ordine alfabetico:

```
if Parola < "BANANA":
  print "la tua parola" + Parola + "viene prima di BANANA."
elif Parola > "BANANA":
  print "la tua parola" + Parola + "viene dopo BANANA."
else:
  print "hai inserito la parola BANANA"
```

Devi comunque fare attenzione al fatto che Python non gestisce le parole maiuscole e minuscole come facciamo noi in modo intuitivo: in un confronto le lettere maiuscole vengono sempre prima delle minuscole, così che:

```
"BANANA" < "BAnana" < "Banana" < "bANANA" < "banana"
"ZEBRA" < "banana"
```

Un modo pratico per aggirare il problema è quello di convertire le stringhe ad un formato standard (tutto maiuscole o tutto minuscole) prima di effettuare il confronto.

7.6 Le stringhe sono immutabili

Si può essere tentati di usare l'operatore porzione `[]` alla sinistra di un'assegnazione, con l'intenzione di cambiare un carattere di una stringa:

```
Saluto = "Ciao!"
Saluto[0] = 'M'            # ERRORE!
print Saluto
```

Invece di ottenere `Miao!` questo codice stampa il messaggio d'errore `TypeError: object doesn't support item assignment`.

Le stringhe sono infatti **immutabili** e ciò significa che non puoi cambiare una stringa esistente. L'unica cosa che puoi eventualmente fare è creare una nuova stringa come variante di quella originale:

```
Saluto = "Ciao!"
NuovoSaluto = 'M' + Saluto[1:]
print NuovoSaluto
```

Abbiamo concatenato la nuova prima lettera ad una porzione di `Saluto`, e questa operazione non ha avuto alcun effetto sulla stringa originale.

7.7 Funzione `Trova`

Secondo te cosa fa questa funzione?

```
def Trova(Stringa, Carattere):
  Indice = 0
  while Indice < len(Stringa):
    if Stringa[Indice] == Carattere:
      return Indice
    Indice = Indice + 1
  return -1
```

In un certo senso questa funzione `Trova` è l'opposto dell'operatore porzione `[]`: invece di prendere un indice e trovare il carattere corrispondente cerca in una stringa la posizione dove appare un carattere e ne restituisce l'indice. Se il carattere non è presente la funzione restituisce `-1`.

Questo è il primo esempio di `return` all'interno di un ciclo. Se `Stringa[Indice] == Carattere` il ciclo viene interrotto prematuramente. Se il carattere non fa parte della stringa il programma termina normalmente e ritorna `-1`.

> *Esercizio: modifica la funzione* `Trova` *per accettare un terzo parametro che rappresenta la posizione dove si deve cominciare a cercare all'interno della stringa.*

7.8 Cicli e contatori

Questo programma conta il numero di volte in cui la lettera `'a'` compare in una stringa, usando un **contatore**:

```
Frutto = "banana"
Conteggio = 0
for Carattere in Frutto:
  if Carattere == 'a':
    Conteggio = Conteggio + 1
print Conteggio
```

La variabile `Conteggio` è inizializzata a 0 e poi incrementata ogni volta che è trovata una `'a'` (**incrementare** significa aumentare di 1; è l'opposto di **decrementare**). Al termine del ciclo `Conteggio` contiene il risultato e cioè il numero totale di lettere `a` nella stringa.

> *Esercizio: incapsula questo codice in una funzione* `ContaLettera` *e fai in modo che questa accetti sia la stringa che la lettera da cercare come parametri.*

> *Esercizio: riscrivi la funzione* `ContaLettera` *in modo che invece di elaborare completamente la stringa faccia uso della versione a tre parametri di* `Trova`*.*

7.9 Il modulo `string`

Il modulo `string` contiene funzioni molto utili per la manipolazione delle stringhe. Come abbiamo già visto prima di poter usare un modulo lo dobbiamo importare:

```
>>> import string
```

Il modulo `string` include una funzione chiamata `find` che fa le stesse cose della nostra funzione `Trova`. Per poterla usare, dopo avere importato il modulo, dobbiamo chiamarla usando la notazione punto
(*NomeDelModulo.NomeDellaFunzione*):

```
>>> Frutto = "banana"
>>> Posizione = string.find(Frutto, "a")
>>> print Posizione
1
```

In realtà `string.find` è più generale della nostra `Trova`. In primo luogo possiamo usarla per cercare stringhe e non soltanto caratteri:

```
>>> string.find("banana", "na")
2
```

Inoltre ammette un argomento ulteriore per specificare da dove vogliamo iniziare la nostra ricerca:

```
>>> string.find("banana", "na", 3)
4
```

Ancora, può prendere due argomenti che specificano il dominio di ricerca, cioè la porzione di stringa originale dove vogliamo effettuare la ricerca:

```
>>> string.find("bob", "b", 1, 2)
-1
```

In questo esempio la ricerca fallisce perché la lettera `'b'` non appare nel dominio definito dagli indici `1` e `2` (da `1` incluso fino a `2` escluso).

7.10 Classificazione dei caratteri

È spesso necessario esaminare un carattere e controllare se questo è maiuscolo, minuscolo, o se si tratta di una cifra o di uno spazio bianco. A questo scopo il modulo `string` fornisce parecchie costanti molto utili.

La stringa `string.lowercase` contiene tutti i caratteri che il sistema considera minuscoli. Allo stesso modo `string.uppercase` contiene tutti i caratteri maiuscoli. Guarda cosa contengono queste stringhe:

```
>>> print string.lowercase
>>> print string.uppercase
>>> print string.digits
```

Possiamo usare queste costanti e la funzione `find` per classificare i caratteri. Per esempio se `find(string.lowercase, Carattere)` ritorna un valore diverso da `-1` allora `Carattere` è minuscolo (un valore diverso da -1 indicherebbe infatti la posizione del carattere trovato):

```
def Minuscolo(Carattere):
  return string.find(string.lowercase, Carattere) != -1
```

In alternativa possiamo usare l'operatore `in` che determina se un carattere compare in una stringa:

```
def Minuscolo(Carattere):
  return Carattere in string.lowercase
```

o il consueto operatore di confronto:

```
def Minuscolo(Carattere):
  return 'a' <= Carattere <= 'z'
```

Se `Carattere` è compreso tra `'a'` e `'z'` deve per forza trattarsi di una lettera minuscola.

> *Esercizio: prova a determinare quale di queste versioni è la più veloce. Puoi pensare ad altre ragioni, a parte la velocità, per preferire una versione piuttosto che un'altra?*

Un'altra costante definita nel modulo `string` può sorprenderti quando provi a stamparla:

```
>>> print string.whitespace
```

I caratteri **spazi bianchi** infatti muovono il cursore senza stampare nulla: sono questi che creano lo spazio bianco tra i caratteri visibili. La costante `string.whitespace` contiene tutti gli spazi bianchi inclusi lo spazio, la tabulazione (`\t`) ed il ritorno a capo (`\n`).

Ci sono molte altre utili funzioni nel modulo `string` ma questo libro non è inteso per essere un manuale di riferimento come invece lo è la *Python Library Reference*, disponibile al sito ufficiale del linguaggio Python `www.python.org`.

7.11 Glossario

Tipo di dati composto: un tipo di dati costruito con componenti che sono essi stessi dei valori.

Attraversare: elaborare tutti gli elementi di un insieme dal primo all'ultimo effettuando su tutti la stessa operazione.

Indice: variabile o valore usati per selezionare un elemento di un insieme ordinato come un carattere in una stringa.

Porzione: parte di una stringa specificata da due indici.

Mutabile: tipo di dati composto al quale possono essere assegnati nuovi valori.

Contatore: variabile usata per contare qualcosa, di solito inizializzata a 0 e successivamente incrementata.

Incrementare: aumentare di 1 il valore di una variabile.

Decrementare: diminuire di 1 il valore di una variabile.

Spazio bianco: ciascuno dei caratteri che se stampato si limita a muovere il cursore senza stampare caratteri visibili. La costante `string.whitespace` contiene tutti gli spazi bianchi.

Capitolo 8

Liste

Una **lista** è una serie ordinata di valori, ognuno identificato da un indice. I valori che fanno parte della lista sono chiamati **elementi**. Le liste sono simili alle stringhe essendo insiemi ordinati di caratteri, fatta eccezione per il fatto che gli elementi di una lista possono essere di tipo qualsiasi. Liste e stringhe (e altri tipi di dati che si comportano da insiemi ordinati) sono chiamate **sequenze**.

8.1 Valori della lista

Ci sono parecchi modi di creare una lista nuova, e quello più semplice è racchiudere i suoi elementi tra parentesi quadrate (`[` e `]`):

```
[10, 20, 30, 40]
["Pippo", "Pluto", "Paperino"]
```

Il primo esempio è una lista di quattro interi, il secondo una lista di tre stringhe. Gli elementi di una stessa lista non devono necessariamente essere tutti dello stesso tipo. Questa lista contiene una stringa, un numero in virgola mobile, un intero ed un'altra lista:

```
["ciao", 2.0, 5, [10, 20]]
```

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

Le liste che contengono numeri interi consecutivi sono così comuni che Python fornisce un modo semplice per crearle:

```
>>> range(1,5)
[1, 2, 3, 4]
```

La funzione `range` prende due argomenti e ritorna una lista che contiene tutti gli interi a partire dal primo (incluso) fino al secondo (escluso).

Ci sono altre due forme per `range`. Con un solo argomento crea una lista a partire da 0:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Se è presente un terzo argomento questo specifica l'intervallo tra valori successivi, chiamato **passo**. Questo esempio mostra come ottenere una stringa dei numeri dispari tra 1 e 10:

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Infine esiste una lista speciale che non contiene alcun elemento: è chiamata lista vuota ed è indicata da `[]`.

Con tutti questi modi di creare liste sarebbe un peccato non poter variare il contenuto di una lista o poter passare liste come parametri di funzioni. Infatti entrambe queste cose possono essere fatte:

```
>>> Vocabolario = ["amico", "casa", "telefono"]
>>> Numeri = [17, 123]
>>> ListaVuota = []
>>> print Vocabolario, Numeri, ListaVuota
['amico', 'casa', 'telefono'] [17, 123] []
```

8.2 Accesso agli elementi di una lista

La sintassi per l'accesso agli elementi di una lista è la stessa che abbiamo già visto per i caratteri di una stringa: anche in questo caso facciamo uso dell'operatore porzione (`[]`). L'espressione tra parentesi quadrate specifica l'indice dell'elemento (non dimenticare che gli indici partono da 0!):

```
>>> print Numeri[0]
17
>>> Numeri[1] = 5
```

L'operatore porzione può comparire in qualsiasi posto di un'espressione: quando è alla sinistra di un'assegnazione cambia uno degli elementi della lista (nell'esempio appena visto l'elemento 123 è diventato 5).

Come indice possiamo inoltre usare qualsiasi espressione che produca un intero:

```
>>> Numeri[3-2]
5
>>> Numeri[1.0]
TypeError: sequence index must be integer
```

Provando a leggere o modificare un elemento che non esiste si ottiene un messaggio d'errore:

```
>>> Numeri[2] = 5
IndexError: list assignment index out of range
```

Se un indice ha valore negativo il conteggio parte dalla fine della lista:

```
>>> Numeri[-1]
5
>>> Numeri[-2]
17
>>> Numeri[-3]
IndexError: list index out of range
```

`Numeri[-1]` è quindi l'ultimo elemento della lista, `Numeri[-2]` il penultimo e `Numeri[-3]` non esiste essendo la nostra lista composta di 2 soli elementi.

È comune usare una variabile di ciclo come indice di una lista:

```
Squadre = ["Juventus", "Inter", "Milan", "Roma"]

i = 0
while i < 4:
  print Squadre[i]
  i = i + 1
```

Questo ciclo `while` conta da 0 a 4: quando l'indice del ciclo `i` vale 4 la condizione diventa falsa e il ciclo termina. Il corpo del ciclo è eseguito solo quando `i` è 0, 1, 2 e 3.

Ad ogni ciclo la variabile `i` è usata come indice della lista: questo tipo di elaborazione è chiamata **elaborazione trasversale di una lista** o **attraversamento di una lista**.

8.3 Lunghezza di una lista

La funzione `len` ritorna la lunghezza di una lista. È sempre bene usare `len` per conoscere il limite superiore in un ciclo, piuttosto che usare un valore costante: in questo modo se la lunghezza della lista dovesse cambiare non dovrai scorrere il programma per modificarne i cicli, e sicuramente `len` funzionerà correttamente per liste di ogni lunghezza:

```
Squadre = ["Juventus", "Inter", "Milan", "Roma"]

i = 0
while i < len(Squadre):
  print Squadre[i]
  i = i + 1
```

L'ultima volta che il ciclo è eseguito `i` vale `len(Squadre) - 1` che è l'indice dell'ultimo elemento della lista. Quando al successivo incremento `i` diventa `len(Squadre)` la condizione diventa falsa ed il corpo non è eseguito, dato che `len(Squadre)` non è un indice valido.

Sebbene una lista possa contenere a sua volta un'altra lista questa lista annidata conta come un singolo elemento indipendentemente dalla sua lunghezza. La lunghezza della lista seguente è 4:

```
['ciao!', 1, ['mela', 'pera', 'banana'], [1, 2, 3]]
```

> *Esercizio: scrivi un ciclo che attraversa la lista precedente e stampa la lunghezza di ogni elemento.*

8.4 Appartenenza ad una lista

`in` è un operatore booleano (restituisce vero o falso) che controlla se un valore è presente in una lista. L'abbiamo già usato con le stringhe nella sezione 7.10 ma funziona anche con le liste e con altri tipi di sequenze:

```
>>> Squadre = ['Juventus', 'Inter', 'Milan', 'Roma']
>>> 'Inter' in Squadre
1
>>> 'Arsiero' in Squadre
0
```

Dato che `Inter` è un membro della lista `Squadre` l'operatore `in` ritorna *vero*; `Arsiero` non fa parte della lista e l'operazione `in` ritorna *falso*.

Possiamo usare `not` in combinazione con `in` per controllare se un elemento non fa parte di una lista:

```
>>> 'Arsiero' not in Squadre
1
```

8.5 Liste e cicli `for`

Il ciclo `for` che abbiamo visto nella sezione 7.3 funziona anche con le liste. La sintassi generica per il ciclo `for` in questo caso è:

```
for VARIABILE in LISTA:
  CORPO
```

Questa istruzione è equivalente a:

```
i = 0
while i < len(LISTA):
  VARIABILE = LISTA[i]
  CORPO
  i = i + 1
```

Il ciclo `for` è più conciso perché possiamo eliminare l'indice del ciclo `i`. Ecco il ciclo di uno degli esempi appena visti riscritto con il `for`:

```
for Squadra in Squadre:
  print Squadra
```

Si legge quasi letteralmente: "Per (ciascuna) Squadra in (nella lista di) Squadre, stampa (il nome della) Squadra".

Nel ciclo `for` può essere usata qualsiasi espressione che produca una lista:

```
for Numero in range(20):
  if Numero % 2 == 0:
    print Numero

for Frutto in ["banana", "mela", "pera"]:
  print "Mi piace la" + Frutto + "!"
```

Il primo esempio stampa tutti i numeri pari tra 0 e 19. Il secondo esprime l'entusiasmo per la frutta.

8.6 Operazioni sulle liste

L'operatore + concatena le liste:

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

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

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

Nel primo esempio abbiamo ripetuto `[0]` quattro volte. Nel secondo abbiamo ripetuto la lista `[1, 2, 3]` tre volte.

8.7 Porzioni di liste

Le porzioni che abbiamo già visto alla sezione 7.4 lavorano anche con le liste:

```
>>> Lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> Lista[1:3]
['b', 'c']
>>> Lista[:4]
['a', 'b', 'c', 'd']
>>> Lista[3:]
['d', 'e', 'f']
>>> Lista[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

8.8 Le liste sono mutabili

A differenza delle stringhe le liste sono mutabili e ciò significa che gli elementi possono essere modificati. Usando l'operatore porzione nella parte sinistra dell'assegnazione possiamo aggiornare un elemento:

```
>>> Frutta = ["banana", "mela", "susina"]
>>> Frutta[0] = "pera"
>>> Frutta[-1] = "arancia"
>>> print Frutta
['pera', 'mela', 'arancia']
```

Con l'operatore porzione possiamo modificare più elementi alla volta:

```
>>> Lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> Lista[1:3] = ['x', 'y']
>>> print Lista
['a', 'x', 'y', 'd', 'e', 'f']
```

Possiamo rimuovere elementi da una lista assegnando loro una lista vuota:

```
>>> Lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> Lista[1:3] = []
>>> print Lista
['a', 'd', 'e', 'f']
```

Possono essere aggiunti elementi ad una lista inserendoli in una porzione vuota nella posizione desiderata:

```
>>> Lista = ['a', 'd', 'f']
>>> Lista[1:1] = ['b', 'c']
>>> print Lista
['a', 'b', 'c', 'd', 'f']
>>> Lista[4:4] = ['e']
>>> print Lista
['a', 'b', 'c', 'd', 'e', 'f']
```

8.9 Cancellazione di liste

Usare le porzioni per cancellare elementi delle liste non è poi così pratico ed è facile sbagliare. Python fornisce un'alternativa molto più leggibile.

`del` rimuove un elemento da una lista:

```
>>> a = ['uno', 'due', 'tre']
>>> del a[1]
>>> a
['uno', 'tre']
```

Come puoi facilmente immaginare `del` gestisce anche gli indici negativi e avvisa con messaggio d'errore se l'indice è al di fuori dei limiti ammessi.

Puoi usare una porzione come indice di `del`:

```
>>> Lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del Lista[1:5]
>>> print Lista
['a', 'f']
```

Come abbiamo già visto la porzione indica tutti gli elementi a partire dal primo indice incluso fino al secondo indice escluso.

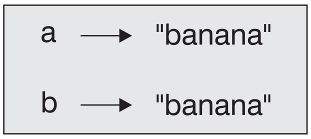
8.10 Oggetti e valori

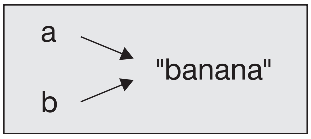
Se eseguiamo queste istruzioni

```
a = "banana"
b = "banana"
```

sappiamo che sia `a` che `b` si riferiscono ad una stringa contenente le lettere `"banana"`. A prima vista non possiamo dire se puntano alla *stessa stringa* in memoria.

I possibili casi sono due:





Nel primo caso `a` e `b` si riferiscono a due diverse "cose" che hanno lo stesso valore. Nel secondo caso si riferiscono alla stessa "cosa". Queste "cose" hanno un nome: **oggetti**. Un oggetto è un qualcosa cui può far riferimento una variabile.

Ogni oggetto ha un **identificatore** unico che possiamo ricavare con la funzione `id`. Stampando l'identificatore di `a` e di `b` possiamo dire subito se le due variabili si riferiscono allo stesso oggetto:

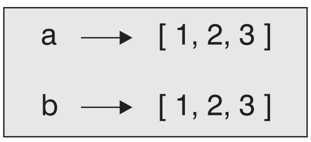
```
>>> id(a)
135044008
>>> id(b)
135044008
```

Otteniamo lo stesso identificatore e ciò significa che Python ha creato in memoria un'unica stringa cui fanno riferimento entrambe le variabili `a` e `b`.

In questo ambito le liste si comportano diversamente dalle stringhe, dato che quando creiamo due liste queste sono sempre oggetti diversi:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

Il diagramma di stato in questo caso è



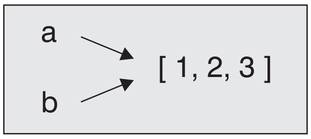
`a` e `b` hanno lo stesso valore ma non si riferiscono allo stesso oggetto.

8.11 Alias

Dato che le variabili si riferiscono ad oggetti quando assegniamo una variabile ad un'altra entrambe le variabili si riferiscono allo stesso oggetto:

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

In questo caso il diagramma di stato è



La stessa lista in questo caso ha due nomi differenti, `a` e `b`, e diciamo che questi sono due **alias**. Dato che l'oggetto cui entrambi si riferiscono è lo stesso è indifferente quale degli alias si usi per effettuare un'elaborazione:

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Sebbene questo comportamento possa essere desiderabile è nella maggior parte dei casi difficilmente controllabile e può portare a effetti indesiderati e inattesi. In generale è buona norma evitare gli alias in caso di oggetti mutabili, mentre per quelli immutabili non ci sono problemi. Ecco perché Python si permette di usare la stessa stringa con diversi alias quando si tratta di risparmiare memoria senza che questo fatto causi alcun problema. La stringa è un oggetto immutabile e quindi non può essere modificata: non c'è quindi il rischio di causare spiacevoli effetti collaterali.

8.12 Clonare le liste

Se vogliamo modificare una lista e mantenere una copia dell'originale dobbiamo essere in grado di copiare il contenuto della lista e non solo di creare un suo alias. Questo processo è talvolta chiamato **clonazione** per evitare l'ambiguità insita nella parola "copia".

Il modo più semplice per clonare una lista è quello di usare l'operatore porzione:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

Il fatto di prendere una porzione di `a` crea una nuova lista. In questo caso la porzione consiste degli elementi dell'intera lista, dato che non sono stati specificati gli indici iniziale e finale.

Ora siamo liberi di modificare `b` senza doverci preoccupare di `a`:

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

> *Esercizio: disegna un diagramma di stato per* `a` *e per* `b` *prima e dopo questa modifica.*

8.13 Parametri di tipo lista

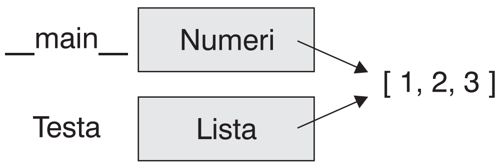
Se passiamo una lista come parametro di funzione in realtà passiamo un suo riferimento e non una sua copia. Per esempio la funzione `Testa` prende una lista come parametro e ne ritorna il primo elemento:

```
def Testa(Lista):
  return Lista[0]
```

Ecco com'è usata:

```
>>> Numeri = [1, 2, 3]
>>> Testa(Numeri)
1
```

Il parametro `Lista` e la variabile `Numeri` sono alias dello stesso oggetto. Il loro diagramma di stato è



Dato che l'oggetto lista è condiviso da due frame l'abbiamo disegnato a cavallo di entrambi.

Se una funzione modifica una lista passata come parametro, viene modificata la lista stessa e non una sua copia. Per esempio `CancellaTesta` rimuove il primo elemento da una lista:

```
def CancellaTesta(Lista):
  del Lista[0]
```

Ecco com'è usata `CancellaTesta`:

```
>>> Numeri = [1, 2, 3]
>>> CancellaTesta(Numeri)
>>> print Numeri
[2, 3]
```

Quando una funzione ritorna una lista in realtà viene ritornato un riferimento alla lista stessa. Per esempio `Coda` ritorna una lista che contiene tutti gli elementi di una lista a parte il primo:

```
def Coda(Lista):
  return Lista[1:]
```

Ecco com'è usata `Coda`:

```
>>> Numeri = [1, 2, 3]
>>> Resto = Coda(Numeri)
>>> print Resto
[2, 3]
```

Dato che il valore ritornato è stato creato con l'operatore porzione stiamo restituendo una nuova lista. La creazione di `Resto` ed ogni suo successivo cambiamento non ha alcun effetto sulla lista originale `Numeri`.

8.14 Liste annidate

Una lista annidata è una lista che compare come elemento di un'altra lista. Nell'esempio seguente il quarto elemento della lista (ricorda che stiamo parlando dell'elemento numero 3 dato che il primo ha indice 0) è una lista:

```
>>> Lista = ["ciao", 2.0, 5, [10, 20]]
```

Se stampiamo `Lista[3]` otteniamo `[10, 20]`. Per estrarre un elemento da una lista annidata possiamo procedere in due tempi:

```
>>> Elemento = Lista[3]
>>> Elemento[0]
10
```

O possiamo combinare i due passi in un'unica istruzione:

```
>>> Lista[3][0]
10
```

L'operatore porzione viene valutato da sinistra verso destra così questa espressione ricava il quarto elemento (indice 3) di `Lista` ed essendo questo una lista ne estrae il primo elemento (indice 0).

8.15 Matrici

Le liste annidate sono spesso usate per rappresentare matrici. Per esempio la matrice

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

può essere rappresentata come

```
>>> Matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`Matrice` è una lista di tre elementi dove ciascuno è una riga della matrice. Possiamo selezionare una singola riga nel solito modo:

```
>>> Matrice[1]
[4, 5, 6]
```

O estrarre una singola cella usando il doppio indice:

```
>>> Matrice[1][1]
5
```

Il primo indice seleziona la riga ed il secondo la colonna. Questo è un modo comune di rappresentare le matrici ma non è l'unico: una variante è quella di usare una lista di colonne invece che di righe. Vedremo in seguito un'alternativa completamente diversa quando avremo visto i dizionari.

8.16 Stringhe e liste

Due delle funzioni più utili nel modulo `string` hanno a che fare con le liste di stringhe. La funzione `split` spezza una stringa in una lista di parole singole, considerando un qualsiasi carattere di spazio bianco come punto di interruzione tra parole consecutive:

```
>>> import string
>>> Verso = "Nel mezzo del cammin..."
>>> string.split(Verso)
['Nel', 'mezzo', 'del', 'cammin...']
```

Può anche essere usato un argomento opzionale per specificare quale debba essere il **delimitatore** da considerare. In questo esempio usiamo la stringa `el` come delimitatore:

```
>>> string.split(Verso, 'el')
['N', ' mezzo d', ' cammin...']
```

Il delimitatore non appare nella lista.

La funzione `join` si comporta in modo inverso rispetto a `split`: prende una lista di stringhe e ne concatena gli elementi inserendo uno spazio tra ogni coppia:

```
>>> Lista = ['Nel', 'mezzo', 'del', 'cammin...']
>>> string.join(Lista)
'Nel mezzo del cammin...'
```

Come nel caso di `split`, `join` accetta un argomento opzionale che rappresenta il delimitatore da inserire tra gli elementi. Il delimitatore di default è uno spazio ma può essere cambiato:

```
>>> string.join(Lista, '_')
'Nel_mezzo_del_cammin...'
```

Esercizio: descrivi la relazione tra la lista `Verso` *e cosa ottieni da* `string.join(string.split(Verso))`. *Sono le stesse per tutte le stringhe o in qualche caso possono essere diverse?*

8.17 Glossario

Lista: collezione di oggetti identificata da un nome dove ogni oggetto è selezionabile grazie ad un indice.

Indice: variabile intera o valore che indica un elemento all'interno di una lista.

Elemento: valore in una lista (o in altri tipi di sequenza). L'operatore porzione seleziona gli elementi di una lista.

Sequenza: ognuno dei tipi di dati che consiste in una lista ordinata di elementi identificati da un indice.

Lista annidata: lista che è un elemento di un'altra lista.

Attraversamento di una lista: accesso in sequenza di tutti gli elementi di una lista.

Oggetto: zona di memoria cui si può riferire una variabile.

Alias: più variabili che si riferiscono allo stesso oggetto con nomi diversi.

Clonare: creare un nuovo oggetto che ha lo stesso valore di un oggetto già esistente.

Delimitatore: carattere o stringa usati per indicare dove una stringa deve essere spezzata.

Capitolo 9

Tuple

9.1 Mutabilità e tuple

Finora hai visto due tipi composti: le stringhe (sequenze di caratteri) e le liste (sequenze di elementi di tipo qualsiasi). Una delle differenze che abbiamo notato è che le gli elementi di una lista possono essere modificati, mentre non possono essere alterati i caratteri in una stringa: le stringhe sono infatti **immutabili** mentre le liste sono **mutabili**.

C'è un altro tipo di dati in Python, simile alla lista eccetto per il fatto che è immutabile: la **tupla**. La tupla è una lista di valori separati da virgole:

```
>>> tupla = 'a', 'b', 'c', 'd', 'e'
```

Sebbene non sia necessario, è convenzione racchiudere le tuple tra parentesi tonde per ragioni di chiarezza:

```
>>> tupla = ('a', 'b', 'c', 'd', 'e')
```

Per creare una tupla con un singolo elemento dobbiamo aggiungere la virgola finale dopo l'elemento:

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

Senza la virgola, infatti, Python tratterebbe `('a')` come una stringa tra parentesi:

```
>>> t2 = ('a')
>>> type(t2)
<type 'string'>
```

Sintassi a parte le operazioni sulle tuple sono identiche a quelle sulle liste. L'operatore indice seleziona un elemento dalla tupla:

```
>>> tupla = ('a', 'b', 'c', 'd', 'e')
>>> tupla[0]
'a'
```

e l'operatore porzione seleziona una serie di elementi consecutivi:

```
>>> tupla[1:3]
('b', 'c')
```

A differenza delle liste se cerchiamo di modificare gli elementi di una tupla otteniamo un messaggio d'errore:

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

Naturalmente anche se non possiamo modificare gli elementi di una tupla possiamo sempre rimpiazzarla con una sua copia modificata:

```
>>> tupla = ('A',) + tupla[1:]
>>> tupla
('A', 'b', 'c', 'd', 'e')
```

9.2 Assegnazione di tuple

Di tanto in tanto può essere necessario scambiare i valori di due variabili. Con le istruzioni di assegnazione convenzionali dobbiamo usare una variabile temporanea. Per esempio per scambiare `a` e `b`:

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

Questo approccio è poco intuitivo e l'uso dell'**assegnazione di tuple** lo rende decisamente più comprensibile:

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

La parte sinistra dell'assegnazione è una tupla di variabili; la parte destra una tupla di valori. Ogni valore è assegnato alla rispettiva variabile. Tutte le espressioni sulla destra sono valutate prima delle assegnazioni. Questa caratteristica rende le tuple estremamente versatili.

Ovviamente il numero di variabili sulla sinistra deve corrispondere al numero di valori sulla destra:

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

9.3 Tuple come valori di ritorno

Le funzioni possono ritornare tuple. Per fare un esempio possiamo scrivere una funzione che scambia due valori:

```
def Scambia(x, y):
  return y, x
```

e in seguito possiamo assegnare il valore di ritorno della funzione ad una tupla di due variabili:

```
a, b = Scambia(a, b)
```

In questo caso non c'è una grande utilità nel rendere `Scambia` una funzione. Anzi occorre stare attenti ad uno dei pericoli insiti nell'incapsulamento di `Scambia`:

```
def Scambia(x, y):      # versione non corretta
  x, y = y, x
```

Se chiamiamo la funzione con:

```
Scambia(a, b)
```

apparentemente tutto sembra corretto, ma quando controlliamo i valori di `a` e `b` prima e dopo lo "scambio" in realtà ci accorgiamo che questi non sono cambiati. Perché? Perché quando chiamiamo questa funzione non vengono passate le variabili `a` e `b` come argomenti, ma i loro valori. Questi valori vengono assegnati a `x` e `y`; al termine della funzione, quando `x` e `y` vengono rimosse perché variabili locali, qualsiasi valore in esse contenuto viene irrimediabilmente perso.

Questa funzione non produce messaggi d'errore ma ciononostante non fa ciò che noi volevamo farle fare: questo è un esempio di errore di semantica.

> *Esercizio: disegna il diagramma di stato della funzione* `Scambia` *così da capire perché non funziona.*

9.4 Numeri casuali

La maggior parte dei programmi fanno la stessa cosa ogni volta che vengono eseguiti e sono detti per questo **deterministici**. Di solito un programma deterministico è una cosa voluta in quanto a parità di dati in ingresso ci attendiamo lo stesso risultato. Per alcune applicazioni, invece, abbiamo bisogno che l'esecuzione sia imprevedibile: i videogiochi sono un esempio lampante, ma ce ne sono tanti altri.

Creare un programma realmente non deterministico (e quindi imprevedibile) è una cosa piuttosto difficile, ma ci sono dei sistemi per renderlo abbastanza casuale da soddisfare la maggior parte delle esigenze in tal senso. Uno dei sistemi è quello di generare dei numeri casuali ed usarli per determinare i risultati prodotti dal programma. Python fornisce delle funzioni di base che generano numeri **pseudocasuali**: questi numeri non sono realmente casuali in senso matematico ma per i nostri scopi saranno più che sufficienti.

Il modulo `random` contiene una funzione chiamata `random` che restituisce un numero in virgola mobile compreso tra 0.0 (compreso) e 1.0 (escluso). Ad ogni chiamata di `random` si ottiene il numero seguente di una lunga serie di numeri pseudocasuali. Per vedere un esempio prova ad eseguire questo ciclo:

```
import random

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

Per generare un numero casuale (lo chiameremo così d'ora in poi, anche se è sottinteso che la casualità ottenuta non è assoluta) compreso tra 0.0 (compreso) ed un limite superiore `Limite` (escluso) moltiplica `x` per `Limite`.

> *Esercizio: tenta di generare un numero casuale compreso tra il* `LimiteInferiore` *(compreso) ed il* `LimiteSuperiore` *(escluso).*
>
> *Esercizio addizionale: genera un numero intero compreso tra il* `LimiteInferiore` *ed il* `LimiteSuperiore` *comprendendo entrambi questi limiti.*

9.5 Lista di numeri casuali

Proviamo a scrivere un programma che usa i numeri casuali, iniziando con la costruzione di una lista di questi numeri. `ListaCasuale` prende un parametro intero `Lungh` e ritorna una lista di questa lunghezza composta di numeri casuali. Iniziamo con una lista di `Lungh` zeri e sostituiamo in un ciclo un elemento alla volta con un numero casuale:

```
def ListaCasuale(Lungh):
  s = [0] * Lungh
  for i in range(Lungh):
    s[i] = random.random()
  return s
```

Testiamo la funzione generando una lista di otto elementi: per poter controllare i programmi è sempre bene partire con insiemi di dati molto piccoli.

```
>>> ListaCasuale(8)
[0.11421081445000203, 0.38367479346590505, 0.16056841528993915,
0.29204721527340882, 0.75201663462563095, 0.31790165552578986,
0.43858231029411354, 0.27749268689939965]
```

I numeri casuali generati da `random` si ritengono distribuiti uniformemente tanto che ogni valore è egualmente probabile.

Se dividiamo la gamma dei valori generati in intervalli della stessa grandezza e contiamo il numero di valori casuali che cadono in ciascun intervallo dovremmo ottenere, approssimativamente, la stessa cifra in ciascuno, sempre che l'esperimento sia effettuato un buon numero di volte.

Possiamo testare questa affermazione scrivendo un programma per dividere la gamma dei valori in intervalli e contare il numero di valori in ciascuno di essi.

9.6 Conteggio

Un buon approccio a questo tipo di problemi è quello di dividere il problema in sottoproblemi e applicare a ciascun sottoproblema uno schema di soluzione già visto in precedenza.

In questo caso vogliamo attraversare una lista di numeri e contare il numero di volte in cui un valore cade in un determinato intervallo. Questo suona familiare: nella sezione 7.8 abbiamo già scritto un programma che attraversa una stringa e conta il numero di volte in cui appare una determinata lettera. Possiamo allora copiare il vecchio programma e adattarlo al problema corrente. Il programma originale era:

```
Conteggio = 0
for Carattere in Frutto:
  if Carattere == 'a':
    Conteggio = Conteggio + 1
print Conteggio
```

Il primo passo è quello di sostituire `Frutto` con `Lista` e `Carattere` con `Numero`. Questo non cambia il programma ma semplicemente lo rende più leggibile.

Il secondo passo è quello di cambiare la condizione dato che siamo interessati a verificare se `Numero` cade tra `LimiteInferiore` e `LimiteSuperiore`.

```
Conteggio = 0
for Numero in Lista
  if LimiteInferiore < Numero < LimiteSuperiore:
    Conteggio = Conteggio + 1
print Conteggio
```

L'ultimo passo è quello di incapsulare questo codice in una funzione chiamata `NellIntervallo`. I parametri della funzione sono la lista da controllare ed i valori `LimiteInferiore` and `LimiteSuperiore`:

```
def NellIntervallo(Lista, LimiteInferiore, LimiteSuperiore):
  Conteggio = 0
  for Numero in Lista:
    if LimiteInferiore < Numero < LimiteSuperiore:
      Conteggio = Conteggio + 1
  return Conteggio
```

Copiando e modificando un programma esistente siamo stati capaci di scrivere questa funzione velocemente risparmiando un bel po' di tempo di test. Questo tipo di piano di sviluppo è chiamato **pattern matching**: se devi cercare una soluzione a un problema che hai già risolto riusa una soluzione che avevi già trovato, modificandola per adattarla quel tanto che serve in base alle nuove circostanze.

9.7 Aumentare il numero degli intervalli

A mano a mano che il numero degli intervalli cresce `NellIntervallo` diventa poco pratica da gestire. Con due soli intervalli ce la caviamo ancora bene:

```
Intervallo1 = NellIntervallo(a, 0.0, 0.5)
Intervallo2 = NellIntervallo(a, 0.5, 1.0)
```

ma con quattro intervalli è facile commettere errori sia nel calcolo dei limiti sia nella battitura dei numeri:

```
Intervallo1 = NellIntervallo(a, 0.0, 0.25)
Intervallo2 = NellIntervallo(a, 0.25, 0.5)
Intervallo3 = NellIntervallo(a, 0.5, 0.75)
Intervallo4 = NellIntervallo(a, 0.75, 1.0)
```

Ci sono due ordini di problemi: il primo è che dobbiamo creare un nome di variabile per ciascun risultato; il secondo è che dobbiamo calcolare a mano i limiti inferiore e superiore per ciascun intervallo prima di chiamare la funzione.

Risolveremo innanzitutto questo secondo problema: se il numero degli intervalli che vogliamo considerare è `NumIntervalli` allora l'ampiezza di ogni intervallo è `1.0 / NumIntervalli`.

Possiamo usare un ciclo per calcolare i limiti inferiore e superiore per ciascun intervallo, usando `i` come indice del ciclo da 0 a `NumIntervalli-1`:

```
AmpiezzaIntervallo = 1.0 / NumIntervalli
for i in range(NumIntervalli):
  LimiteInferiore = i * AmpiezzaIntervallo
  LimiteSuperiore = LimiteInferiore + AmpiezzaIntervallo
  print "da", LimiteInferiore, "a", LimiteSuperiore
```

Per calcolare il limite inferiore di ogni intervallo abbiamo moltiplicato l'indice del ciclo per l'ampiezza di ciascun intervallo; per ottenere il limite superiore abbiamo sommato al limite inferiore la stessa ampiezza.

Con `NumIntervalli = 8` il risultato è:

```
da 0.0 a 0.125
da 0.125 a 0.25
da 0.25 a 0.375
da 0.375 a 0.5
da 0.5 a 0.625
da 0.625 a 0.75
da 0.75 a 0.875
da 0.875 a 1.0
```

Puoi vedere come ogni intervallo sia della stessa ampiezza, come tutta la gamma da 0.0 a 1.0 sia presente e come non ci siano intervalli che si sovrappongono.

Ora torniamo al primo problema: abbiamo bisogno di memorizzare 8 interi senza dover creare variabili distinte. Le liste ci vengono in aiuto e l'indice del ciclo sembra essere un ottimo sistema per selezionare di volta in volta un elemento della lista.

Creiamo la lista dei conteggi all'esterno del ciclo dato che la dobbiamo creare una sola volta (e non ad ogni ciclo). All'interno del ciclo chiameremo ripetutamente `NellIntervallo` e aggiorneremo l'`i`-esimo elemento della lista dei conteggi:

```
NumIntervalli = 8
Conteggio = [0] * NumIntervalli
AmpiezzaIntervallo = 1.0 / NumIntervalli
for i in range(NumIntervalli):
  LimiteInferiore = i * AmpiezzaIntervallo
  LimiteSuperiore = LimiteInferiore + AmpiezzaIntervallo
  Conteggio[i] = NellIntervallo(Lista, LimiteInferiore, \
                 LimiteSuperiore)
print Conteggio
```

Con una lista di 1000 valori questo programma produce una lista di conteggi di questo tipo:

```
[138, 124, 128, 118, 130, 117, 114, 131]
```

Ci aspettavamo per ogni intervallo un valore medio di 125 (1000 numeri divisi per 8 intervalli) ed in effetti ci siamo andati abbastanza vicini da poter affermare che il generatore di numeri casuali si comporta in modo sufficientemente realistico.

> *Esercizio: prova questa funzione con liste più lunghe per vedere se il conteggio di valori in ogni intervallo tende o meno a livellarsi (maggiore è il numero di prove più i valori dovrebbero diventare simili).*

9.8 Una soluzione in una sola passata

Anche se il programma funziona correttamente non è ancora sufficientemente efficiente. Ogni volta che il ciclo chiama `NellIntervallo` viene attraversata l'intera lista. A mano a mano che il numero di intervalli cresce questo implica un gran numero di attraversamenti di liste.

Sarebbe meglio riuscire a fare un singolo attraversamento della lista ed elaborare direttamente in quale intervallo cade ogni elemento, incrementando il contatore opportuno.

Nella sezione precedente abbiamo preso un indice `i` e lo abbiamo moltiplicato per `AmpiezzaIntervallo` per trovare il limite inferiore di un determinato intervallo. Quello che vogliamo fare ora è ricavare direttamente l'indice dell'intervallo cui un valore appartiene.

Questo problema è esattamente l'inverso del precedente: dobbiamo indovinare in quale intervallo cade un valore dividendo quest'ultimo per `AmpiezzaIntervallo` invece di moltiplicare un indice per `AmpiezzaIntervallo`.

Dal momento che `AmpiezzaIntervallo = 1.0 / NumIntervalli`, dividere per `AmpiezzaIntervallo` è lo stesso di moltiplicare per `NumIntervalli`. Se moltiplichiamo un numero nella gamma da 0.0 a 1.0 per `NumIntervalli` otteniamo un numero compreso tra 0.0 e `NumIntervalli`. Se arrotondiamo questo risultato all'intero inferiore otteniamo proprio quello che stavamo cercando: l'indice dell'intervallo dove cade il valore.

```
NumIntervalli = 8
Conteggio = [0] * NumIntervalli
for i in Lista:
  Indice = int(i * NumIntervalli)
  Conteggio[Indice] = Conteggio[Indice] + 1
```

Abbiamo usato la funzione `int` per convertire un numero in virgola mobile in un intero.

> *Esercizio: È possibile per questo calcolo produrre un indice che sia fuori dalla gamma di numeri ammessa (negativo o più grande di* `len(Conteggio)-1`*)?*

Una lista come `Conteggi` che contiene il numero dei valori per una serie di intervalli è chiamata **istogramma**.

> *Esercizio: scrivi una funzione chiamata* `Istogramma` *che prende una lista ed il numero di intervalli da considerare e ritorna l'istogramma della distribuzione dei valori per ciascun intervallo.*

9.9 Glossario

Tipo immutabile: tipo in cui i singoli elementi non possono essere modificati. L'operazione di assegnazione ad elementi o porzioni produce un errore.

Tipo mutabile: tipo di dati in cui gli elementi possono essere modificati. Liste e dizionari sono mutabili; stringhe e tuple non lo sono.

Tupla: tipo di sequenza simile alla lista con la differenza di essere immutabile. Le tuple possono essere usate dovunque serva un tipo immutabile, per esempio come chiave in un dizionario.

Assegnazione ad una tupla: assegnazione di tutti gli elementi della tupla usando un'unica istruzione di assegnazione.

Programma deterministico: programma che esegue le stesse operazioni ogni volta che è eseguito.

Pseudocasuale: sequenza di numeri che sembra essere casuale ma in realtà è il risultato di un'elaborazione deterministica.

Istogramma: lista di interi in cui ciascun elemento conta il numero di volte in cui una determinata condizione si verifica.

Pattern matching: piano di sviluppo del programma che consiste nell'identificare un tracciato di elaborazione già visto e modificarlo per ottenere la soluzione di un problema simile.

Capitolo 10

Dizionari

I tipi di dati composti che abbiamo visto finora (stringhe, liste e tuple) usano gli interi come indici. Qualsiasi tentativo di usare altri tipi di dati produce un errore.

I **dizionari** sono simili agli altri tipi composti ma si differenziano per il fatto di poter usare qualsiasi tipo di dato immutabile come indice. Se desideriamo creare un dizionario per la traduzione di parole dall'inglese all'italiano è utile poter usare la parola inglese come indice di ricerca della corrispondente italiana. Gli indici usati sono in questo caso delle `stringhe`.

Un modo per creare un dizionario è partire con un dizionario vuoto e aggiungere via via gli elementi. Il dizionario vuoto è indicato da `{}`:

```
>>> Eng2Ita = {}
>>> Eng2Ita['one'] = 'uno'
>>> Eng2Ita['two'] = 'due'
```

La prima assegnazione crea un dizionario chiamato `Eng2Ita`; le altre istruzioni aggiungono elementi al dizionario. Possiamo stampare il valore del dizionario nel solito modo:

```
>>> print Eng2Ita
{'one': 'uno', 'two': 'due'}
```

Gli elementi di un dizionario appaiono in una sequenza separata da virgole. Ogni voce contiene un indice ed il corrispondente valore separati da due punti. In un dizionario gli indici sono chiamati **chiavi** e un elemento è detto **coppia chiave-valore**.

Un altro modo di creare un dizionario è quello di fornire direttamente una serie di coppie chiave-valore:

```
>>> Eng2Ita = {'one': 'uno', 'two': 'due', 'three': 'tre'}
```

Se stampiamo ancora una volta il valore di `Eng2Ita` abbiamo una sorpresa:

```
>>> print Eng2Ita
{'one': 'uno', 'three': 'tre', 'two': 'due'}
```

Le coppie chiave-valore non sono in ordine! Per fortuna non c'è ragione di conservare l'ordine di inserimento dato che il dizionario non fa uso di indici numerici. Per cercare un valore usiamo infatti una chiave:

```
>>> print Eng2Ita['two']
'due'
```

La chiave `'two'` produce correttamente `'due'` anche se appare in terza posizione nella stampa del dizionario.

10.1 Operazioni sui dizionari

L'istruzione `del` rimuove una coppia chiave-valore da un dizionario. Vediamo di fare un esempio pratico creando un dizionario che contiene il nome di vari tipi di frutta (la chiave) ed il numero di frutti corrispondenti in magazzino (il valore):

```
>>> Magazzino = {'mele': 430, 'banane': 312, 'arance': 525,
                 'pere': 217}
>>> print Magazzino
{'banane': 312, 'arance': 525, 'pere': 217, 'mele': 430}
```

Dovessimo togliere la scorta di pere dal magazzino possiamo direttamente rimuovere la voce dal dizionario:

```
>>> del Magazzino['pere']
>>> print Magazzino
{'banane': 312, 'arance': 525, 'mele': 430}
```

o se intendiamo solo cambiare il numero di pere senza rimuoverne la voce dal dizionario possiamo cambiare il valore associato:

```
>>> Magazzino['pere'] = 0
>>> print Magazzino
{'banane': 312, 'arance': 525, 'pere': 0, 'mele': 430}
```

La funzione `len` opera anche sui dizionari ritornando il numero di coppie chiave-valore:

```
>>> len(Magazzino)
4
```

10.2 Metodi dei dizionari

Un **metodo** è simile ad una funzione, visto che prende parametri e ritorna valori, ma la sintassi di chiamata è diversa. Il metodo `keys` prende un dizionario e ritorna la lista delle sue chiavi: invece di invocarlo con la sintassi delle funzioni `keys(Eng2Ita)` usiamo la sintassi dei metodi `Eng2Ita.keys()`:

```
>>> Eng2Ita.keys()
['one', 'three', 'two']
```

Questa forma di notazione punto specifica il nome della funzione `keys` ed il nome dell'oggetto cui applicare la funzione `Eng2Ita`. Le parentesi vuote indicano che questo metodo non prende parametri.

Una chiamata ad un metodo è detta **invocazione**; in questo caso diciamo che stiamo invocando `keys` sull'oggetto `Eng2Ita`.

Il metodo `values` funziona in modo simile: ritorna la lista dei valori in un dizionario:

```
>>> Eng2Ita.values()
['uno', 'tre', 'due']
```

Il metodo `items` ritorna entrambi nella forma di una lista di tuple, una per ogni coppia chiave-valore:

```
>>> Eng2Ita.items()
[('one','uno'), ('three', 'tre'), ('two', 'due')]
```

La sintassi fornisce utili informazioni sul tipo ottenuto invocando `items`: le parentesi quadrate indicano che si tratta di una lista; le parentesi tonde che gli elementi della lista sono tuple.

Se un metodo prende un argomento usa la stessa sintassi delle chiamate di funzioni. Il metodo `has_key` prende come argomento una chiave e ritorna *vero* (1) se la chiave è presente nel dizionario:

```
>>> Eng2Ita.has_key('one')
1
>>> End2Ita.has_key('deux')
0
```

Se provi a invocare un metodo senza specificare l'oggetto cui si fa riferimento ottieni un errore:

```
>>> has_key('one')
NameError: has_key
```

Purtroppo il messaggio d'errore a volte, come in questo caso, non è del tutto chiaro: Python cerca di dirci che la funzione `has_key` non esiste, dato che con questa sintassi abbiamo chiamato la funzione `has_key` e non invocato il metodo `has_key` dell'oggetto.

10.3 Alias e copia

Visto che i dizionari sono mutabili devi stare molto attento agli alias: quando due variabili si riferiscono allo stesso oggetto un cambio effettuato su una influenza immediatamente il contenuto dell'altra.

Se desideri poter modificare un dizionario e mantenere una copia dell'originale usa il metodo `copy`. Per fare un esempio costruiamo un dizionario `Opposti` che contiene coppie di parole dal significato opposto:

```
>>> Opposti = {'alto': 'basso', 'giusto': 'sbagliato',
               'vero': 'falso'}
>>> Alias = Opposti
>>> Copia = Opposti.copy()
```

`Alias` e `Opposti` si riferiscono allo stesso oggetto; `Copia` si riferisce ad una copia del dizionario nuova di zecca. Se modifichiamo `Alias`, `Opposti` viene modificato:

```
>>> Alias['giusto'] = 'errato'
>>> Opposti['giusto']
'errato'
```

`Opposti` resta immutato se modifichiamo `Copia`:

```
>>> Copia['giusto'] = 'errato'
>>> Opposti['giusto']
'sbagliato'
```

10.4 Matrici sparse

Nella sezione 8.15 abbiamo usato una lista di liste per rappresentare una matrice. Questa è una buona scelta quando si tratta di rappresentare matrici i cui valori sono in buona parte diversi da zero, ma c'è un tipo di matrice detta "sparsa" i cui valori sono di tipo particolare:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

La rappresentazione sotto forma di lista di questa matrice contiene molti zeri:

```
>>> Matrice = [ [0,0,0,1,0],
                [0,0,0,0,0],
                [0,2,0,0,0],
                [0,0,0,0,0],
                [0,0,0,3,0] ]
```

L'alternativa in questo caso è quella di usare un dizionario, usando come chiavi tuple composte dalla coppia riga/colonna. Ecco la stessa matrice rappresentata con un dizionario:

```
>>> Matrice = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

In questo caso abbiamo solo 3 coppie chiave-valore, una per ciascun elemento diverso da zero nella matrice. Ogni chiave è una tupla ed ogni valore un intero.

Per l'accesso ad un elemento della matrice possiamo usare l'operatore `[]`:

```
>>> Matrice[(0,3)]
1
>>> Matrice[0,3]    # questa sintassi e' equivalente
1
```

Nota come la sintassi per la rappresentazione della matrice sotto forma di dizionario sia diversa da quella della lista di liste: invece di due valori indice usiamo un unico indice che è una tupla di interi.

C'è un problema: se cerchiamo un elemento che è pari a zero otteniamo un errore, dato che non c'è una voce nel dizionario corrispondente alla tupla con quelle coordinate:

```
>>> Matrice[1,3]
KeyError: (1, 3)
```

Il metodo `get` risolve il problema:

```
>>> Matrice.get((0,3), 0)
1
```

Il primo argomento è la tupla-chiave, il secondo il valore che `get` deve ritornare nel caso la chiave non sia presente nel dizionario:

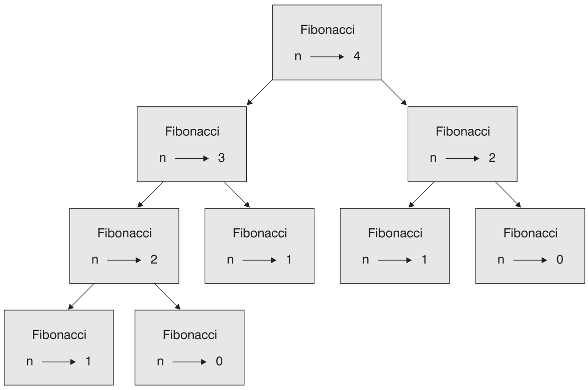
```
>>> Matrice.get((1,3), 0)
0
```

Anche se la sintassi di `get` non è la più intuitiva almeno abbiamo un modo efficiente per accedere ad una matrice sparsa.

10.5 Suggerimenti

Se hai fatto qualche prova con la funzione di `Fibonacci` nella sezione 5.7 avrai notato che man mano che l'argomento passato alla funzione cresce il tempo trascorso prima di ottenere il risultato aumenta molto rapidamente. Mentre `Fibonacci(20)` termina quasi istantaneamente `Fibonacci(30)` impiega qualche secondo e `Fibonacci(40)` impiega un tempo lunghissimo.

Per comprenderne il motivo consideriamo questo **grafico delle chiamate** per la funzione `Fibonacci` con `n=4`:



Un grafico delle chiamate mostra una serie di frame (uno per ogni funzione) con linee che collegano ciascun frame alle funzioni chiamate. A iniziare dall'alto `Fibonacci` con `n=4` chiama `Fibonacci` con `n=3` e `n=2`. A sua volta `Fibonacci` con `n=3` chiama `Fibonacci` con `n=2` e `n=1`. E così via.

Se conti il numero di volte in cui `Fibonacci(0)` e `Fibonacci(1)` sono chiamate ti accorgerai facilmente che questa soluzione è evidentemente inefficiente e le sue prestazioni tendono a peggiorare man mano che l'argomento diventa più grande.

Una buona soluzione è quella di tenere traccia in un dizionario di tutti i valori già calcolati per evitare il ricalcolo in tempi successivi. Un valore che viene memorizzato per un uso successivo è chiamato **suggerimento**. Ecco un'implementazione di `Fibonacci` fatta usando i "suggerimenti":

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

def Fibonacci(n):
  if Precedenti.has_key(n):
    return Precedenti[n]
  else:
    NuovoValore = Fibonacci(n-1) + Fibonacci(n-2)
    Precedenti[n] = NuovoValore
    return NuovoValore
```

Il dizionario `Precedenti` tiene traccia dei numeri di Fibonacci già calcolati. Lo creiamo inserendo solo due coppie: 0 associato a 1 (`Fibonacci(0) = 1`) e 1 associato a 1 (`Fibonacci(1) = 1`).

La nuova funzione `Fibonacci` prima di tutto controlla se nel dizionario è già presente il valore cercato: se c'è viene restituito senza ulteriori elaborazioni. Nel caso non sia presente deve essere calcolato il nuovo valore che poi viene aggiunto al dizionario (per poter essere usato in momenti successivi) prima che la funzione termini.

Usando questa funzione di `Fibonacci` ora riusciamo a calcolare `Fibonacci(40)` in un attimo. Ma quando chiamiamo `Fibonacci(50)` abbiamo un altro tipo di problema:

```
>>> Fibonacci(50)
OverflowError: integer addition
```

La risposta che volevamo ottenere è 20365011074 ed il problema è che questo numero è troppo grande per essere memorizzato in un intero di Python. Durante il calcolo otteniamo un **overflow** che non è altro che uno “sbordamento” dall'intero. Fortunatamente in questo caso la soluzione è molto semplice.

10.6 Interi lunghi

Python fornisce un tipo chiamato `long int` che può maneggiare interi di qualsiasi grandezza.

Ci sono due modi per creare un valore intero lungo. Il primo consiste nello scrivere un intero immediatamente seguito da una `L` maiuscola:

```
>>> type(1L)
<type 'long int'>
```

Il secondo è l'uso della funzione `long` per convertire un valore in intero lungo. `long` può accettare qualsiasi tipo di numero e anche una stringa di cifre:

```
>>> long(1)
1L
>>> long(3.9)
3L
>>> long('57')
57L
```

Tutte le operazioni matematiche operano correttamente sugli interi lunghi così non dobbiamo fare molto per adattare `Fibonacci`:

```
>>> Precedenti = {0:1L, 1:1L}
>>> Fibonacci(50)
20365011074L
```

Solamente cambiando il valore iniziale di `Precedenti` abbiamo cambiato il comportamento di `Fibonacci`. I primi elementi della sequenza sono interi lunghi così tutti i numeri successivi diventano dello stesso tipo.

> *Esercizio: converti* `Fattoriale` *così da produrre interi lunghi come risultato.*

10.7 Conteggio di lettere

Nel capitolo 7 abbiamo scritto una funzione che conta il numero di lettere in una stringa. Una possibile estensione è la creazione di un istogramma della stringa per mostrare la frequenza di ciascuna lettera.

Questo tipo di istogramma può essere utile per comprimere un file di testo: dato che le lettere compaiono con frequenza diversa possiamo usare codici brevi per le lettere più frequenti e codici via via più lunghi per le meno frequenti.

I dizionari permettono di realizzare istogrammi in modo elegante:

```
>>> ConteggioLettere = {}
>>> for Lettera in "Mississippi":
...   ConteggioLettere [Lettera] = ConteggioLettere.get \
                                    (Lettera, 0) + 1
...
>>> ConteggioLettere
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

Siamo partiti con un dizionario vuoto e per ogni lettera della stringa abbiamo incrementato il corrispondente conteggio. Alla fine il dizionario contiene coppie formate da lettera e frequenza e queste coppie rappresentano il nostro istogramma.

Può essere più interessante mostrare l'istogramma in ordine alfabetico, e in questo caso facciamo uso dei metodi `items` e `sort`:

```
>>> ConteggioLettere = ConteggioLettere.items()
>>> ConteggioLettere.sort()
>>> print ConteggioLettere
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Abbiamo visto già il metodo `items` ma `sort` è il primo metodo che incontriamo ad essere applicabile alle liste. Ci sono parecchi altri metodi applicabili alle liste (tra gli altri `append`, `extend` e `reverse`). Puoi consultare la documentazione di Python per avere ulteriori informazioni a riguardo.

10.8 Glossario

Dizionario: collezione di coppie chiave-valore dove si associa ogni chiave ad un valore. Le chiavi devono essere immutabili; i valori possono essere di qualsiasi tipo.

Chiave: valore usato per cercare una voce in un dizionario.

Coppia chiave-valore: elemento di un dizionario.

Metodo: tipo di funzione chiamata con una sintassi particolare ed invocata *su* un oggetto.

Invocare: chiamare un metodo.

Suggerimento: deposito temporaneo di valori precalcolati per evitare elaborazioni inutili.

Overflow: errore generato quando un risultato è troppo grande per essere rappresentato da un determinato formato numerico.

Capitolo 11

File ed eccezioni

Quando un programma è in esecuzione i suoi dati sono in memoria; nel momento in cui il programma termina o il computer viene spento tutti i dati in memoria vengono irrimediabilmente persi. Per conservare i dati devi quindi memorizzarli in un **file**, solitamente memorizzato su hard disk, floppy o CD-ROM.

Lavorando con un gran numero di file è logico cercare di organizzarli: questo viene fatto inserendoli in **cartelle** (dette anche “folder” o “directory”). Ogni file all'interno di una cartella è identificato da un nome unico.

Leggendo e scrivendo file i programmi possono scambiare informazioni e anche generare documenti stampabili usando il formato PDF o altri formati simili.

Lavorare con i file è molto simile a leggere un libro: per usarli li devi prima aprire e quando hai finito li chiudi. Mentre il libro è aperto lo puoi leggere o puoi scrivere una nota sulle sue pagine, sapendo in ogni momento dove ti trovi al suo interno. La maggior parte delle volte leggerai il libro in ordine, ma nulla ti vieta di saltare a determinate pagine facendo uso dell'indice.

Questa metafora può essere applicata ai file. Per aprire un file devi specificarne il nome e l'uso che intendi farne (lettura o scrittura).

L'apertura del file crea un oggetto file: nell'esempio che segue useremo la variabile `f` per riferirci all'oggetto file appena creato.

```
>>> f = open("test.dat","w")
>>> print f
<open file 'test.dat', mode 'w' at fe820>
```

La funzione `open` prende due argomenti: il primo è il nome del file ed il secondo il suo “modo”. Il modo `"w"` significa che stiamo aprendo il file in scrittura.

Nel caso non dovesse esistere un file chiamato `test.dat` l'apertura in scrittura farà in modo di crearlo vuoto. Nel caso dovesse già esistere, la vecchia copia verrà rimpiazzata da quella nuova e definitivamente persa.

Quando stampiamo l'oggetto file possiamo leggere il nome del file aperto, il modo e la posizione dell'oggetto in memoria.

Per inserire dati nel file invochiamo il metodo `write`:

```
>>> f.write("Adesso")
>>> f.write("chiudi il file")
```

La chiusura del file avvisa il sistema che abbiamo concluso la scrittura e rende il file disponibile alla lettura:

```
>>> f.close()
```

Solo dopo aver chiuso il file possiamo riaprirlo in lettura e leggerne il contenuto. Questa volta l'argomento di modo è `"r"`:

```
>>> f = open("test.dat","r")
```

Se cerchiamo di aprire un file che non esiste otteniamo un errore:

```
>>> f = open("test.cat","r")
IOError: [Errno 2] No such file or directory: 'test.cat'
```

Il metodo `read` legge dati da un file. Senza argomenti legge l'intero contenuto del file:

```
>>> Testo = f.read()
>>> print Testo
Adessochiudi il file
```

Non c'è spazio tra `Adesso` e `chiudi` perché non abbiamo scritto uno spazio tra le due stringhe al momento della scrittura.

`read` accetta anche un argomento che specifica quanti caratteri leggere:

```
>>> f = open("test.dat","r")
>>> print f.read(5)
Adess
```

Se non ci sono caratteri sufficienti nel file, `read` ritorna quelli effettivamente disponibili. Quando abbiamo raggiunto la fine del file `read` ritorna una stringa vuota:

```
>>> print f.read(1000006)
ochiudi il file
>>> print f.read()

>>>
```

La funzione che segue copia un file leggendo e scrivendo fino a 50 caratteri per volta. Il primo argomento è il nome del file originale, il secondo quello della copia:

```
def CopiaFile(Originale, Copia):
  f1 = open(Originale, "r")
  f2 = open(Copia, "w")
  while 1:
    Testo = f1.read(50)
```

```
    if Testo == "":
      break
    f2.write(Testo)
  f1.close()
  f2.close()
  return
```

L'istruzione `break` è nuova: la sua esecuzione interrompe immediatamente il loop saltando alla prima istruzione che lo segue (in questo caso `f1.close()`).

Il ciclo `while` dell'esempio è apparentemente infinito dato che la sua condizione ha valore `1` ed è quindi sempre vera. L'unico modo per uscire da questo ciclo è di eseguire un `break` che viene invocato quando `Testo` è una stringa vuota e cioè quando abbiamo raggiunto la fine del file in lettura.

11.1 File di testo

Un **file di testo** è un file che contiene caratteri stampabili e spazi bianchi, organizzati in linee separate da caratteri di ritorno a capo. Python è stato specificatamente progettato per elaborare file di testo e fornisce metodi molto efficaci per rendere facile questo compito.

Per dimostrarlo creeremo un file di testo composto da tre righe di testo separate da dei ritorno a capo:

```
>>> f = open("test.dat","w")
>>> f.write("linea uno\nlinea due\nlinea tre\n")
>>> f.close()
```

Il metodo `readline` legge tutti i caratteri fino al prossimo ritorno a capo:

```
>>> f = open("test.dat","r")
>>> print f.readline()
linea uno

>>>
```

`readlines` ritorna tutte le righe rimanenti come lista di stringhe:

```
>>> print f.readlines()
['linea due\n', 'linea tre\n']
```

In questo caso il risultato è in formato lista e ciò significa che le stringhe appaiono racchiuse tra apici e i caratteri di ritorno a capo come sequenze di escape.

Alla fine del file `readline` ritorna una stringa vuota e `readlines` una lista vuota:

```
>>> print f.readline()

>>> print f.readlines()
[]
```

Quello che segue è un esempio di elaborazione di un file: `FiltraFile` fa una copia del file `Originale` omettendo tutte le righe che iniziano con `#`:

```
def FiltraFile(Originale, Nuovo):
  f1 = open(Originale, "r")
  f2 = open(Nuovo, "w")
  while 1:
    Linea = f1.readline()
    if Linea == "":
      break
    if Linea[0] == '#':
      continue
    f2.write(Linea)
  f1.close()
  f2.close()
  return
```

L'istruzione `continue` termina l'iterazione corrente e continua con il prossimo ciclo: il flusso di programma torna all'inizio del ciclo, controlla la condizione e procede di conseguenza.

Non appena `Linea` è una stringa vuota il ciclo termina grazie al `break`. Se il primo carattere di `Linea` è un carattere cancelletto l'esecuzione continua tornando all'inizio del ciclo. Se entrambe le condizioni falliscono (non siamo in presenza della fine del file né il primo carattere è un cancelletto) la riga viene copiata nel nuovo file.

11.2 Scrittura delle variabili

L'argomento di `write` dev'essere una stringa così se vogliamo inserire altri tipi di valore in un file li dobbiamo prima convertire in stringhe. Il modo più semplice è quello di usare la funzione `str`:

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

Un'alternativa è quella di usare l'**operatore di formato** `%`. Quando è applicato agli interi `%` è l'operatore modulo, ma quando il primo operando è una stringa `%` identifica l'operatore formato.

Il primo operando in questo caso è la **stringa di formato** ed il secondo una tupla di espressioni. Il risultato è una stringa formattata secondo la stringa di formato.

Cominciamo con un esempio semplice: la **sequenza di formato** `"%d"` significa che la prima espressione della tupla che segue deve essere formattata come un intero:

```
>>> NumAuto = 52
>>> "%d" % NumAuto
'52'
```

Il risultato è la stringa '52' che non deve essere confusa con il valore intero 52.

Una sequenza di formato può comparire dovunque all'interno di una stringa di formato così possiamo inserire un valore in una frase qualsiasi:

```
>>> NumAuto = 52
>>> "In luglio abbiamo venduto %d automobili." % NumAuto
'In luglio abbiamo venduto 52 automobili.'
```

La sequenza di formato "%f" formatta il valore nella tupla in un numero in virgola mobile e "%s" lo formatta come stringa:

```
>>> "Ricavo in %d giorni: %f milioni di %s." % (34,6.1,'euro')
'Ricavo in 34 giorni: 6.100000 milioni di euro.'
```

Per default la formattazione in virgola mobile %f stampa sei cifre decimali.

Il numero delle espressioni nella tupla deve naturalmente essere corrispondente a quello delle sequenze di formato nella stringa di formato, ed i tipi delle espressioni devono corrispondere a quelli delle sequenze di formato:

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

Nel primo esempio la stringa di formato si aspetta tre espressioni ma nella tupla ne abbiamo passate solo due. Nel secondo vogliamo stampare un intero ma stiamo passando una stringa.

Per avere un miglior controllo del risultato possiamo modificare le sequenze di formato aggiungendo delle cifre:

```
>>> "%6d" % 62
'    62'
>>> "%12f" % 6.1
'    6.100000'
```

Il numero dopo il segno di percentuale è il numero minimo di caratteri che dovrà occupare il nostro valore dopo essere stato convertito. Se la conversione occuperà un numero di spazi minori verranno aggiunti spazi alla sinistra. Se il numero è negativo sono aggiunti spazi dopo il numero convertito:

```
>>> "%-6d" % 62
'62    '
```

Nel caso dei numeri in virgola mobile possiamo anche specificare quante cifre devono comparire dopo il punto decimale:

```
>>> "%12.2f" % 6.1
'        6.10'
```

In questo esempio abbiamo stabilito di creare una stringa di 12 caratteri con due cifre dopo il punto decimale. Questo tipo di formattazione è utile quando si devono stampare dei valori contabili in forma tabellare con il punto decimale allineato.

Immaginiamo un dizionario che contiene il nome (la chiave) e tariffa oraria (il valore) per una serie di lavoratori. Ecco una funzione che stampa il contenuto del dizionario in modo formattato:

```
def Report(Tariffe) :
  Lavoratori = Tariffe.keys()
  Lavoratori.sort()
  for Lavoratore in Lavoratori:
    print "%-20s %12.02f" % (Lavoratore, Tariffe[Lavoratore])
```

Per provare questa funzione creiamo un piccolo dizionario e stampiamone il contenuto:

```
>>> Tariffe = {'Maria': 6.23, 'Giovanni': 5.45, 'Alberto': 4.25}
>>> Report(Tariffe)
Alberto                      4.25
Giovanni                     5.45
Maria                        6.23
```

Controllando la larghezza di ciascun valore garantiamo che le colonne saranno perfettamente allineate, sempre che il nome rimanga al di sotto dei 20 caratteri e la tariffa oraria al di sotto delle 12 cifre...

11.3 Directory

Quando crei un nuovo file aprendolo e scrivendoci qualcosa, questo viene memorizzato nella directory corrente e cioè in quella dalla quale sei partito per eseguire l'interprete Python. Allo stesso modo se richiedi la lettura di un file Python lo cercherà nella directory corrente.

Se vuoi aprire il file da qualche altra parte dovrai anche specificare il **percorso** per raggiungerlo, e cioè il nome della directory di appartenenza:

```
>>>   f = open("/usr/share/dict/words","r")
>>>   print f.readline()
Aarhus
```

In questo esempio apriamo un file chiamato `words` che risiede in una directory chiamata `dict` che risiede in un'altra directory chiamata `share` che a sua volta risiede in `usr`. Quest'ultima risiede nella directory principale del sistema, `/`, secondo il formato Linux.

Non puoi usare `/` come parte di un nome di file proprio perché questo è un carattere delimitatore che viene inserito tra i vari nomi delle directory nella definizione del percorso.

11.4 Pickling

Abbiamo visto come per mettere dei valori in un file di testo li abbiamo dovuti preventivamente convertire in stringhe. Hai già visto come farlo usando `str`:

```
>>> f.write (str(12.3))
>>> f.write (str(4.567))
>>> f.write (str([1,2,3]))
```

Il problema è che quando cerchi di recuperare il valore dal file ottieni una stringa, e non l'informazione originale che avevi memorizzato. Oltretutto non c'è nemmeno il modo di sapere dove inizia o termina di preciso la stringa che definisce il valore nel file:

```
>>>   f.readline()
'12.34.567[1, 2, 3]'
```

La soluzione è il **pickling** (che letteralmente significa "conservazione sotto vetro") chiamato così perché "preserva" le strutture dei dati. Il modulo `pickle` contiene tutti i comandi necessari. Importiamo il modulo e poi apriamo il file nel solito modo:

```
>>> import pickle
>>> f = open("test.pck","w")
```

Per memorizzare una struttura di dati usa il metodo `dump` e poi chiudi il file:

```
>>> pickle.dump(12.3, f)
>>> pickle.dump(4.567, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

Puoi riaprire il file e caricare le strutture di dati memorizzati con il metodo `load`:

```
>>> f = open("test.pck","r")
>>> x = pickle.load(f)
>>> x
12.3
>>> type(x)
<type 'float'>
>>> x2 = pickle.load(f)
>>> x2
4.567
>>> type(x2)
<type 'float'>
>>> y = pickle.load(f)
>>> y
[1, 2, 3]
>>> type(y)
<type 'list'>
```

Ogni volta che invochiamo `load` otteniamo un singolo valore completo del suo tipo originale.

11.5 Eccezioni

Se il programma si blocca a causa di un errore in esecuzione viene creata un'**eccezione**: l'interprete si ferma e mostra un messaggio d'errore.

Le eccezioni più comuni per i programmi che hai visto finora possono essere:

- la divisione di un valore per zero:

  ```
  >>> print 55/0
  ZeroDivisionError: integer division or modulo
  ```

- la richiesta di un elemento di una lista con un indice errato:

  ```
  >>> a = []
  >>> print a[5]
  IndexError: list index out of range
  ```

- la richiesta di una chiave non esistente in un dizionario:

  ```
  >>> b = {}
  >>> print b['pippo']
  KeyError: pippo
  ```

In ogni caso il messaggio d'errore è composto di due parti: la categoria dell'errore e le specifiche separati dai due punti. Normalmente Python stampa la traccia del programma al momento dell'errore ma in questi esempi sarà omessa per questioni di leggibilità.

Molte operazioni possono generare errori in esecuzione ma in genere desideriamo che il programma non si blocchi quando questo avviene. La soluzione è quella di **gestire** l'eccezione usando le istruzioni `try` ed `except`.

Per fare un esempio possiamo chiedere ad un operatore di inserire il nome di un file per poi provare ad aprirlo. Se il file non dovesse esistere non vogliamo che il programma si blocchi mostrando un messaggio di errore; così cerchiamo di gestire questa possibile eccezione:

```
NomeFile = raw_input('Inserisci il nome del file: ')
try:
  f = open (NomeFile, "r")
except:
  print 'Il file', NomeFile, 'non esiste'
```

L'istruzione `try` esegue le istruzioni nel suo blocco. Se non si verificano eccezioni (e cioè se le istruzioni del blocco `try` sono eseguite senza errori) l'istruzione `except` ed il blocco corrispondente vengono saltate ed il flusso del programma prosegue dalla prima istruzione presente dopo il blocco `except`. Nel caso si verifichi qualche eccezione (nel nostro caso la più probabile è che il file richiesto non esiste) viene interrotto immediatamente il flusso del blocco `try` ed eseguito il blocco `except`.

Possiamo incapsulare questa capacità in una funzione: `FileEsiste` prende un nome di un file e ritorna *vero* se il file esiste, *falso* se non esiste.

```
def FileEsiste(NomeFile):
  try:
    f = open(NomeFile)
    f.close()
    return 1
  except:
    return 0
```

Puoi anche usare blocchi di `except` multipli per gestire diversi tipi di eccezioni. Vedi a riguardo il *Python Reference Manual*.

Con `try/except` possiamo fare in modo di continuare ad eseguire un programma in caso di errore. Possiamo anche "sollevare" delle eccezioni nel corso del programma con l'istruzione **raise** in modo da generare un errore in esecuzione quando qualche condizione non è verificata:

```
def InputNumero() :
  x = input ('Dimmi un numero: ')
  if x > 16 :
    raise 'ErroreNumero', 'mi aspetto numeri minori di 17!'
  return x
```

In questo caso viene generato un errore in esecuzione quando è introdotto un numero maggiore di 16.

L'istruzione `raise` prende due argomenti: il tipo di eccezione e l'indicazione specifica del tipo di errore. `ErroreNumero` è un nuovo tipo di eccezione che abbiamo inventato appositamente per questa applicazione.

Se la funzione che chiama `InputNumero` gestisce gli errori il programma continua, altrimenti Python stampa il messaggio d'errore e termina l'esecuzione:

```
>>> InputNumero()
Dimmi un numero: 17
ErroreNumero: mi aspetto numeri minori di 17!
```

Il messaggio di errore include l'indicazione del tipo di eccezione e l'informazione aggiuntiva che è stata fornita.

> *Esercizio: scrivi una funzione che usa* `InputNumero` *per inserire un numero da tastiera e gestisce l'eccezione* `ErroreNumero` *quando il numero non è corretto.*

11.6 Glossario

File: entità identificata da un nome solitamente memorizzata su hard disk, floppy disk o CD-ROM, e contenente una serie di dati.

Directory: contenitore di file; è anche chiamata *cartella* o *folder*.

Percorso: sequenza di nomi di directory che specifica l'esatta locazione di un file.

File di testo: file che contiene solo caratteri stampabili organizzati come serie di righe separate da caratteri di ritorno a capo.

Istruzione `break`: istruzione che causa l'interruzione immediata del flusso del programma e l'uscita da un ciclo.

Istruzione `continue`: istruzione che causa l'immediato ritorno del flusso del programma all'inizio del ciclo senza completarne il corpo.

Operatore formato: operatore indicato da '`%`' che produce una stringa di caratteri in base ad una stringa di formato passata come argomento.

Stringa di formato: stringa che contiene caratteri stampabili e stabilisce come debbano essere formattati una serie di valori in una stringa.

Sequenza di formato: sequenza di caratteri che inizia con `%` e che stabilisce, all'interno di una stringa di formato, come debba essere convertito in stringa un singolo valore.

Pickling: operazione di scrittura su file di un valore assieme alla descrizione del suo tipo, in modo da poterlo recuperare facilmente in seguito.

Eccezione: errore in esecuzione.

Gestire un'eccezione: prevedere i possibili errori in esecuzione per fare in modo che questi non interrompano l'esecuzione del programma.

Sollevare un'eccezione: segnalare la presenza di una situazione anomala facendo uso dell'istruzione `raise`.

Capitolo 12

Classi e oggetti

12.1 Tipi composti definiti dall'utente

Abbiamo usato alcuni dei tipi composti predefiniti e ora siamo pronti per crearne uno tutto nostro: il tipo `Punto`.

Considerando il concetto matematico di punto nelle due dimensioni, il punto è definito da una coppia di numeri (le coordinate). In notazione matematica le coordinate dei punti sono spesso scritte tra parentesi con una virgola posta a separare i due valori. Per esempio $(0, 0)$ rappresenta l'origine e (x, y) il punto che si trova x unità a destra e y unità in alto rispetto all'origine.

Un modo naturale di rappresentare un punto in Python è una coppia di numeri in virgola mobile e la questione che ci rimane da definire è in che modo raggruppare questa coppia di valori in un oggetto composto: un sistema veloce anche se poco elegante sarebbe l'uso di una tupla, anche se possiamo fare di meglio.

Un modo alternativo è quello di definire un nuovo tipo composto chiamato **classe**. Questo tipo di approccio richiede un po' di sforzo iniziale, ma i suoi benefici saranno subito evidenti.

Una definizione di classe ha questa sintassi:

```
class Punto:
  pass
```

Le definizioni di classe possono essere poste in qualsiasi punto di un programma ma solitamente per questioni di leggibilità sono poste all'inizio, subito sotto le istruzioni `import`. Le regole di sintassi per la definizione di una classe sono le stesse degli altri tipi composti: la definizione dell'esempio crea una nuova classe chiamata `Punto`. L'istruzione **pass** non ha effetti: è stata usata per il solo fatto che la definizione prevede un corpo che deve ancora essere scritto.

Creando la classe `Punto` abbiamo anche creato un nuovo tipo di dato chiamato con lo stesso nome. I membri di questo tipo sono detti **istanze** del tipo o **oggetti**. La creazione di una nuova istanza è detta **istanziazione**: solo al momento dell'istanziazione parte della memoria è riservata per depositare il valore dell'oggetto. Per creare un oggetto di tipo `Punto` viene chiamata una funzione chiamata `Punto`:

```
P1 = Punto()
```

Alla variabile `P1` è assegnato il riferimento ad un nuovo oggetto `Punto`. Una funzione come `Punto`, che crea nuovi oggetti e riserva quindi della memoria per depositarne i valori, è detta **costruttore**.

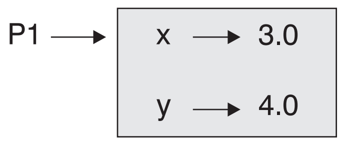
12.2 Attributi

Possiamo aggiungere un nuovo dato ad un'istanza usando la notazione punto:

```
>>> P1.x = 3.0
>>> P1.y = 4.0
```

Questa sintassi è simile a quella usata per la selezione di una variabile appartenente ad un modulo, tipo `math.pi` e `string.uppercase`. In questo caso stiamo selezionando una voce da un'istanza e queste voci che fanno parte dell'istanza sono dette **attributi**.

Questo diagramma di stato mostra il risultato delle assegnazioni:



La variabile `P1` si riferisce ad un oggetto Punto che contiene due attributi ed ogni attributo (una coordinata) si riferisce ad un numero in virgola mobile.

Possiamo leggere il valore di un attributo con la stessa sintassi:

```
>>> print P1.y
4.0
>>> x = P1.x
>>> print x
3.0
```

L'espressione `P1.x` significa "vai all'oggetto puntato da `P1` e ottieni il valore del suo attributo `x`". In questo caso assegniamo il valore ad una variabile chiamata `x`: non c'è conflitto tra la variabile locale `x` e l'attributo `x` di `P1`: lo scopo della notazione punto è proprio quello di identificare la variabile cui ci si riferisce evitando le ambiguità.

Puoi usare la notazione punto all'interno di ogni espressione così che le istruzioni proposte di seguito sono a tutti gli effetti perfettamente lecite:

```
print '(' + str(P1.x) + ', ' + str(P1.y) + ')'
DistanzaAlQuadrato = P1.x * P1.x + P1.y * P1.y
```

La prima riga stampa `(3.0, 4.0)`; la seconda calcola il valore 25.0.

Potresti essere tentato di stampare direttamente il valore di `P1`:

```
>>> print P1
<__main__.Punto instance at 80f8e70>
```

Il risultato indica che `P1` è un'istanza della classe `Punto` e che è stato definito in `__main__`. `80f8e70` è l'identificatore univoco dell'oggetto, scritto in base 16 (esadecimale). Probabilmente questo non è il modo più pratico di mostrare un oggetto `Punto` ma vedrai subito come renderlo più comprensibile.

> *Esercizio: crea e stampa un oggetto* `Punto` *e poi usa* `id` *per stampare l'identificatore univoco dell'oggetto. Traduci la forma esadecimale dell'identificatore in decimale e verifica che i due valori trovati coincidono.*

12.3 Istanze come parametri

Puoi passare un'istanza come parametro ad una funzione nel solito modo:

```
def StampaPunto(Punto):
  print '(' + str(Punto.x) + ', ' + str(Punto.y) + ')'
```

`StampaPunto` prende un oggetto `Punto` come argomento e ne stampa gli attributi in forma standard. Se chiami `StampaPunto(P1)` la stampa è `(3.0, 4.0)`.

> *Esercizio: riscrivi la funzione* `DistanzaTraDuePunti` *che abbiamo già visto alla sezione 5.2 così da accettare due oggetti di tipo* `Punto` *invece di quattro numeri.*

12.4 Uguaglianza

La parola "uguale" sembra così intuitiva che probabilmente non hai mai pensato più di tanto a cosa significa veramente.

Quando dici "Alberto ed io abbiamo la stessa auto" naturalmente vuoi dire che entrambi possedete un'auto dello stesso modello ed è sottinteso che stai parlando di due auto diverse e non di una soltanto. Se dici "Alberto ed io abbiamo la stessa madre" è sottinteso che la madre è la stessa e voi siete fratelli[1]. L'idea stessa di uguaglianza dipende quindi dal contesto.

Quando parli di oggetti abbiamo la stessa ambiguità: se due oggetti di tipo `Punto` sono gli stessi, significa che hanno semplicemente gli stessi dati (coordinate) o che si sta parlando di un medesimo oggetto?

Per vedere se due riferimenti fanno capo allo stesso oggetto usa l'operatore `==`:

[1] Non tutte le lingue soffrono di questa ambiguità: per esempio il tedesco ha parole diverse per indicare tipi diversi di similarità: "la stessa auto" in questo contesto è traducibile con "gleiche Auto" e "la stessa madre" con "selbe Mutter".

```
>>> P1 = Punto()
>>> P1.x = 3
>>> P1.y = 4
>>> P2 = Punto()
>>> P2.x = 3
>>> P2.y = 4
>>> P1 == P2
0
```

Anche se `P1` e `P2` hanno le stesse coordinate non fanno riferimento allo stesso oggetto ma a due oggetti diversi. Se assegniamo `P1` a `P2` allora le due variabili sono alias dello stesso oggetto:

```
>>> P2 = P1
>>> P1 == P2
1
```

Questo tipo di uguaglianza è detta **uguaglianza debole** perché si limita a confrontare solo i riferimenti delle variabili e non il contenuto degli oggetti.

Per confrontare il contenuto degli oggetti (**uguaglianza forte**) possiamo scrivere una funzione chiamata `StessoPunto`:

```
def StessoPunto(P1, P2) :
  return (P1.x == P2.x) and (P1.y == P2.y)
```

Se creiamo due differenti oggetti che contengono gli stessi dati possiamo ora usare `StessoPunto` per verificare se entrambi rappresentano lo stesso punto:

```
>>> P1 = Punto()
>>> P1.x = 3
>>> P1.y = 4
>>> P2 = Punto()
>>> P2.x = 3
>>> P2.y = 4
>>> StessoPunto(P1, P2)
1
```

Logicamente se le due variabili si riferiscono allo stesso punto e sono alias l'una dell'altra allo stesso tempo garantiscono l'uguaglianza debole e quella forte.

12.5 Rettangoli

Se volessimo creare una classe per rappresentare un rettangolo quali informazioni dovremmo fornire per specificarlo in modo univoco? Per rendere le cose più semplici partiremo con un rettangolo orientato lungo gli assi.

Ci sono poche possibilità tra cui scegliere: potremmo specificare il centro del rettangolo e le sue dimensioni (altezza e larghezza); oppure specificare un angolo di riferimento e le dimensioni (ancora altezza e larghezza); o ancora specificare le coordinate di due punti opposti. Una scelta convenzionale abbastanza comune è quella di specificare il punto in alto a sinistra e le dimensioni.

Definiamo la nuova classe:

```
class Rettangolo:
  pass
```

Per istanziare un nuovo oggetto rettangolo:

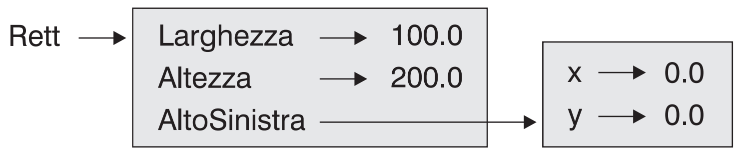
```
Rett = Rettangolo()
Rett.Larghezza = 100.0
Rett.Altezza = 200.0
```

Questo codice crea un nuovo oggetto `Rettangolo` con due attributi in virgola mobile. Ci manca solo il punto di riferimento in alto a sinistra e per specificarlo possiamo inserire un oggetto all'interno di un altro oggetto:

```
Rett.AltoSinistra = Punto()
Rett.AltoSinistra.x = 0.0;
Rett.AltoSinistra.y = 0.0;
```

L'operatore punto è usato per comporre l'espressione: `Rett.AltoSinistra.x` significa "vai all'oggetto cui si riferisce `Rett` e seleziona l'attributo chiamato `AltoSinistra`; poi vai all'oggetto cui si riferisce `AltoSinistra` e seleziona l'attributo chiamato `x`."

La figura mostra lo stato di questo oggetto:



12.6 Istanze come valori di ritorno

Le funzioni possono ritornare istanze. Possiamo quindi scrivere una funzione `TrovaCentro` che prende un oggetto `Rettangolo` come argomento e restituisce un oggetto `Punto` che contiene le coordinate del centro del rettangolo:

```
def TrovaCentro(Rettangolo):
  P = Punto()
  P.x = Rettangolo.AltoSinistra.x + Rettangolo.Larghezza/2.0
  P.y = Rettangolo.AltoSinistra.y + Rettangolo.Altezza/2.0
  return P
```

Per chiamare questa funzione passa `Rett` come argomento e assegna il risultato ad una variabile:

```
>>> Centro = TrovaCentro(Rett)
>>> StampaPunto(Centro)
(50.0, 100.0)
```

12.7 Gli oggetti sono mutabili

Possiamo cambiare lo stato di un oggetto facendo un'assegnazione ad uno dei suoi attributi. Per fare un esempio possiamo cambiare le dimensioni di `Rett`:

```
Rett.Larghezza = Rett.Larghezza + 50
Rett.Altezza = Rett.Altezza + 100
```

Incapsulando questo codice in un metodo e generalizzandolo diamo la possibilità di aumentare le dimensioni di qualsiasi rettangolo:

```
def AumentaRettangolo(Rettangolo, AumentoLargh, AumentoAlt) :
  Rettangolo.Larghezza = Rettangolo.Larghezza + AumentoLargh;
  Rettangolo.Altezza = Rettangolo.Altezza + AumentoAlt;
```

Le variabili `AumentoLargh` e `AumentoAlt` indicano di quanto devono essere aumentate le dimensioni del rettangolo. Invocare questo metodo ha lo stesso effetto di modificare il `Rettangolo` che è passato come argomento.

Creiamo un nuovo rettangolo chiamato `R1` e passiamolo a `AumentaRettangolo`:

```
>>> R1 = Rettangolo()
>>> R1.Larghezza = 100.0
>>> R1.Altezza = 200.0
>>> R1.AltoSinistra = Punto()
>>> R1.AltoSinistra.x = 0.0;
>>> R1.AltoSinistra.y = 0.0;
>>> AumentaRettangolo(R1, 50, 100)
```

Mentre stiamo eseguendo `AumentaRettangolo` il parametro `Rettangolo` è un alias per `R1`. Ogni cambiamento apportato a `Rettangolo` modifica direttamente `R1` e viceversa.

> *Esercizio: scrivi una funzione chiamata* `MuoviRettangolo` *che prende come parametri un* `Rettangolo` *e due valori* `dx` *e* `dy`*. La funzione deve spostare le coordinate del punto in alto a sinistra sommando alla posizione x il valore* `dx` *e alla posizione y il valore* `dy`*.*

12.8 Copia

Abbiamo già visto che gli alias possono rendere il programma difficile da leggere perché una modifica può cambiare il valore di variabili che apparentemente non hanno nulla a che vedere con quelle modificate. Man mano che le dimensioni del programma crescono diventa difficile tenere a mente quali variabili si riferiscano ad un dato oggetto.

La copia di un oggetto è spesso una comoda alternativa all'alias. Il modulo `copy` contiene una funzione `copy` che permette di duplicare qualsiasi oggetto:

```
>>> import copy
>>> P1 = Punto()
```

```
>>> P1.x = 3
>>> P1.y = 4
>>> P2 = copy.copy(P1)
>>> P1 == P2
0
>>> StessoPunto(P1, P2)
1
```

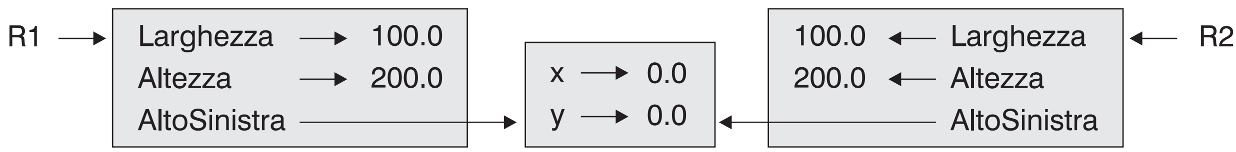
Dopo avere importato il modulo `copy` possiamo usare il metodo `copy` in esso contenuto per creare un nuovo oggetto `Punto`. `P1` e `P2` non solo sono lo stesso punto ma contengono gli stessi dati.

Per copiare un semplice oggetto come `Punto` che non contiene altri oggetti al proprio interno `copy` è sufficiente. Questa è chiamata **copia debole**:

```
>>> Punto2 = copy.copy(Punto1)
```

Quando abbiamo a che fare con un `Rettangolo` che contiene al proprio interno un riferimento ad un altro oggetto `Punto`, `copy` non lavora come ci si aspetta dato che viene copiato il riferimento a `Punto` così che sia il vecchio che il nuovo `Rettangolo` si riferiscono allo stesso oggetto invece di averne uno proprio per ciascuno.

Se creiamo il rettangolo `R1` nel solito modo e ne facciamo una copia `R2` usando `copy` il diagramma di stato risultante sarà:



Quasi certamente non è questo ciò che vogliamo. In questo caso, invocando `AumentaRettangolo` su uno dei rettangoli non si cambieranno le dimensioni dell'altro, ma `MuoviRettangolo` sposterà entrambi! Questo comportamento genera parecchia confusione e porta facilmente a commettere errori.

Fortunatamente il modulo `copy` contiene un altro metodo chiamato `deepcopy` che copia correttamente non solo l'oggetto ma anche gli eventuali oggetti presenti al suo interno:

```
>>> Oggetto2 = copy.deepcopy(Oggetto1)
```

Ora `Oggetto1` e `Oggetto2` sono oggetti completamente separati e occupano diverse zone di memoria.

Possiamo usare `deepcopy` per riscrivere completamente `AumentaRettangolo` così da non cambiare il `Rettangolo` originale ma restituire una copia con le nuove dimensioni:

```
def AumentaRettangolo(Rettangolo, AumentoLargh, AumentoAlt) :
  import copy
  NuovoRett = copy.deepcopy(Rettangolo)
```

```
NuovoRett.Larghezza = NuovoRett.Larghezza + AumentoLargh
NuovoRett.Altezza = NuovoRett.Altezza + AumentoAlt;
return NuovoRett
```

Esercizio: riscrivi `MuoviRettangolo` *per creare e restituire un nuovo rettangolo invece di modificare quello originale.*

12.9 Glossario

Classe: tipo di dato composto definito dall'utente.

Istanziare: creare un'istanza di una determinata classe.

Istanza: oggetto che appartiene ad una classe.

Oggetto: tipo di dato composto che è spesso usato per definire un concetto o una cosa del mondo reale.

Costruttore: metodo usato per definire nuovi oggetti.

Attributo: uno dei componenti che costituiscono un'istanza.

Uguaglianza debole: uguaglianza di riferimenti che si verifica quando due variabili si riferiscono allo stesso oggetto.

Uguaglianza forte: uguaglianza di valori che si verifica quando due variabili si riferiscono a oggetti che hanno lo stesso valore.

Copia debole: copia del contenuto di un oggetto includendo ogni riferimento ad eventuali oggetti interni, realizzata con la funzione `copy` del modulo `copy`.

Copia forte: copia sia del contenuto di un oggetto che degli eventuali oggetti interni e degli oggetti eventualmente contenuti in essi; è realizzata dalla funzione `deepcopy` del modulo `copy`.

Capitolo 13

Classi e funzioni

13.1 Tempo

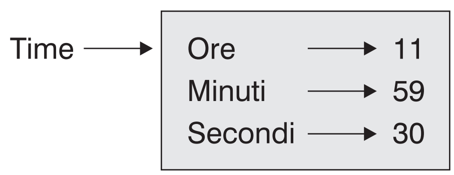
Definiamo ora una classe chiamata `Tempo` che permette di registrare un'ora del giorno:

```
class Tempo:
  pass
```

Possiamo creare un nuovo oggetto `Tempo` assegnando gli attributi per le ore, i minuti e i secondi:

```
Time = Tempo()
Time.Ore = 11
Time.Minuti = 59
Time.Secondi = 30
```

Il diagramma di stato per l'oggetto `Time` è:



> *Esercizio: scrivi una funzione* `StampaTempo` *che prende un oggetto* `Tempo` *come argomento e ne stampa il risultato nella classica forma* `ore:minuti:secondi`.

> *Esercizio: scrivi una funzione booleana* `Dopo` *che prende come argomenti due oggetti* `Tempo` *(*`Tempo1` *e* `Tempo2`*) e ritorna* vero *se* `Tempo1` *segue cronologicamente* `Tempo2` *e* falso *in caso contrario.*

13.2 Funzioni pure

Nelle prossime funzioni scriveremo due versioni di una funzione che chiameremo `SommaTempi` che calcola la somma di due oggetti `Tempo`. Questo permetterà di mostrare due tipi di funzioni: le funzioni pure e i modificatori.

Questa è una versione grezza di `SommaTempi`:

```
def SommaTempi(T1, T2):
  Somma = Tempo()
  Somma.Ore = T1.Ore + T2.Ore
  Somma.Minuti = T1.Minuti + T2.Minuti
  Somma.Secondi = T1.Secondi + T2.Secondi
  return Somma
```

La funzione crea un nuovo oggetto `Tempo`, inizializza i suoi attributi e ritorna un riferimento al nuovo oggetto. Questa viene chiamata **funzione pura** perché non modifica in alcun modo gli oggetti passati come suoi parametri e non ha effetti collaterali (del tipo richiedere l'immissione di un valore da parte dell'operatore o stampare un valore a video).

Ecco un esempio che mostra come usare questa funzione: creiamo due oggetti `Tempo`, `OraCorrente` che contiene l'ora corrente e `TempoCottura` che indica il tempo necessario a preparare un pasto. Se non hai ancora scritto `StampaTempo` guarda la sezione 14.2 prima di continuare:

```
>>> OraCorrente = Tempo()
>>> OraCorrente.Ore = 9
>>> OraCorrente.Minuti = 14
>>> OraCorrente.Secondi=  30

>>> TempoCottura = Tempo()
>>> TempoCottura.Ore =  3
>>> TempoCottura.Minuti = 35
>>> TempoCottura.Secondi = 0

>>> PastoPronto = SommaTempi(OraCorrente, TempoCottura)
>>> StampaTempo(PastoPronto)
```

La stampa di questo programma è `12:49:30` ed il risultato è corretto. D'altra parte ci sono dei casi in cui il risultato è sbagliato: riesci a pensarne uno?

Il problema è che nella nostra funzione non abbiamo tenuto conto del fatto che le somme dei secondi e dei minuti possono andare oltre il 59. Quando capita questo dobbiamo conteggiare il riporto come facciamo con le normali addizioni.

Ecco una seconda versione corretta della funzione:

```
def SommaTempi(T1, T2):
  Somma = Tempo()
  Somma.Ore = T1.Ore + T2.Ore
  Somma.Minuti = T1.Minuti + T2.Minuti
  Somma.Secondi = T1.Secondi + T2.Secondi
```

```
  if Somma.Secondi >= 60:
    Somma.Secondi = Somma.Secondi - 60
    Somma.Minuti = Somma.Minuti + 1

  if Somma.Minuti >= 60:
    Somma.Minuti = Somma.Minuti - 60
    Somma.Ore = Somma.Ore + 1

  return Somma
```

Questa funzione è corretta e comincia ad avere una certa lunghezza. In seguito ti mostreremo un approccio alternativo che ti permetterà di ottenere un codice più corto.

13.3 Modificatori

Ci sono dei casi in cui è utile una funzione che possa modificare gli oggetti passati come suoi parametri. Quando questo si verifica la funzione è detta **modificatore**.

La funzione `Incremento` che somma un certo numero di secondi a `Tempo` può essere scritta in modo molto intuitivo come modificatore. La prima stesura potrebbe essere questa:

```
def Incremento(Tempo, Secondi):
  Tempo.Secondi = Tempo.Secondi + Secondi

  if Tempo.Secondi >= 60:
    Tempo.Secondi = Tempo.Secondi - 60
    Tempo.Minuti = Tempo.Minuti + 1

  if Tempo.Minuti >= 60:
    Tempo.Minuti = Tempo.Minuti - 60
    Tempo.Ore = Tempo.Ore + 1
```

La prima riga calcola il valore mentre le successive controllano che il risultato sia nella gamma di valori accettabili come abbiamo già visto.

Questa funzione è corretta? Cosa succede se il parametro `Secondi` è molto più grande di 60? In questo caso non è abbastanza il riporto 1 tra secondi e minuti, e quindi dobbiamo riscrivere il controllo per fare in modo di continuarlo finché `Secondi` non diventa minore di 60. Una possibile soluzione è quella di sostituire l'istruzione `if` con un ciclo `while`:

```
def Incremento(Tempo, Secondi):
  Tempo.Secondi = Tempo.Secondi + Secondi

  while Tempo.Secondi >= 60:
    Tempo.Secondi = Tempo.Secondi - 60
```

```
    Tempo.Minuti = Tempo.Minuti + 1

  while Tempo.Minuti >= 60:
    Tempo.Minuti = Tempo.Minuti - 60
    Tempo.Ore = Tempo.Ore + 1
```

La funzione è corretta, ma certamente non si tratta ancora della soluzione più efficiente possibile.

> *Esercizio: riscrivi questa funzione facendo in modo di non usare alcun tipo di ciclo.*

> *Esercizio: riscrivi* `Incremento` *come funzione pura e scrivi delle chiamate di funzione per entrambe le versioni.*

13.4 Qual è la soluzione migliore?

Qualsiasi cosa che può essere fatta con i modificatori può anche essere fatta con le funzioni pure e alcuni linguaggi di programmazione non prevedono addirittura i modificatori. Si può affermare che le funzioni pure sono più veloci da sviluppare e portano ad un minor numero di errori, anche se in qualche caso può essere utile fare affidamento sui modificatori.

In generale raccomandiamo di usare funzioni pure quando possibile e usare i modificatori come ultima risorsa solo se c'è un evidente vantaggio nel farlo. Questo tipo di approccio può essere definito **stile di programmazione funzionale**.

13.5 Sviluppo prototipale e sviluppo pianificato

In questo capitolo abbiamo mostrato un approccio allo sviluppo del programma che possiamo chiamare **sviluppo prototipale**: siamo partiti con lo stendere una versione grezza (prototipo) che poteva effettuare solo i calcoli di base, migliorandola e correggendo gli errori man mano che questi venivano trovati.

Sebbene questo approccio possa essere abbastanza efficace può portare a scrivere un codice inutilmente complesso (perché ha a che fare con molti casi speciali) e inaffidabile (dato che è difficile sapere se tutti gli errori sono stati rimossi).

Un'alternativa è lo **sviluppo pianificato** nel quale lo studio preventivo del problema da affrontare rende la programmazione molto più semplice. Nel nostro caso potremmo accorgerci che a tutti gli effetti l'oggetto `Tempo` è rappresentabile da tre cifre in base numerica 60.

Quando abbiamo scritto `SommaTempi` e `Incremento` stavamo effettivamente calcolando una somma in base 60 e questo è il motivo per cui dovevamo gestire il riporto tra secondi e minuti, e tra minuti e ore quando la somma era maggiore di 59.

Questa osservazione ci suggerisce un altro tipo di approccio al problema: possiamo convertire l'oggetto `Tempo` in un numero singolo ed avvantaggiarci del fatto che il computer lavora bene con le operazioni aritmetiche. Questa funzione converte un oggetto `Tempo` in un intero:

```
def ConverteInSecondi(Orario):
  Minuti = Orario.Ore * 60 + Orario.Minuti
  Secondi = Minuti * 60 + Orario.Secondi
  return Secondi
```

Tutto quello che ci serve è ora un modo per convertire da un intero ad un oggetto `Tempo`:

```
def ConverteInTempo(Secondi):
  Orario = Tempo()
  Orario.Ore = Secondi/3600
  Secondi = Secondi - Orario.Ore * 3600
  Orario.Minuti = Secondi/60
  Secondi = Secondi - Orario.Minuti * 60
  Orario.Secondi = Secondi
  return Orario
```

Forse dovrai pensarci un po' su per convincerti che questa tecnica per convertire un numero da una base all'altra è formalmente corretta. Comunque ora puoi usare queste funzioni per riscrivere `SommaTempi`:

```
def SommaTempi(T1, T2):
  Secondi = ConverteInSecondi(T1) + ConverteInSecondi(T2)
  return ConverteInTempo(Secondi)
```

Questa versione è molto più concisa dell'originale e ed è molto più facile dimostrare la sua correttezza.

> *Esercizio: riscrivi* `Incremento` *usando lo stesso principio.*

13.6 Generalizzazione

Sicuramente la conversione numerica da base 10 a base 60 e viceversa è meno intuitiva da capire, data la sua astrazione. Il nostro intuito ci aveva portato a lavorare con i tempi in un modo molto più comprensibile anche se meno efficace.

Malgrado lo sforzo iniziale abbiamo progettato il nostro programma facendo in modo di trattare i tempi come numeri in base 60, il tempo investito nello scrivere le funzioni di conversione viene abbondantemente recuperato quando riusciamo a scrivere un programma molto più corto, facile da leggere e correggere, e soprattutto più affidabile.

Se il programma è progettato in modo oculato è anche più facile aggiungere nuove caratteristiche. Per esempio immagina di sottrarre due tempi per determinare l'intervallo trascorso. L'approccio iniziale avrebbe portato alla necessità di dover implementare una sottrazione con il prestito. Con le funzioni di conversione, scritte una sola volta ma riutilizzate in varie funzioni, è molto più facile e rapido avere un programma funzionante anche in questo caso.

Talvolta il fatto di rendere un problema più generale e quindi leggermente più difficile da implementare permette di gestirlo in modo più semplice dato che ci sono meno casi speciali da gestire e di conseguenza minori possibilità di errore.

13.7 Algoritmi

Quando trovi una soluzione ad una classe di problemi, invece che ad un singolo problema, hai a che fare con un **algoritmo**. Abbiamo già usato questa parola in precedenza ma non l'abbiamo propriamente definita ed il motivo risiede nel fatto che non è facile trovare una definizione. Proveremo un paio di approcci.

Consideriamo qualcosa che non è un algoritmo: quando hai imparato a moltiplicare due numeri di una cifra hai sicuramente memorizzato la tabella della moltiplicazione. In effetti si è trattato di memorizzare 100 soluzioni specifiche: questo tipo di lavoro non è un algoritmo.

Ma se sei stato "pigro" probabilmente hai trovato delle scorciatoie che ti hanno permesso di alleggerire il lavoro. Per fare un esempio per moltiplicare n per 9 potevi scrivere $n-1$ come prima cifra, seguito da $10-n$ come seconda cifra. Questo sistema è una soluzione generale per moltiplicare ogni numero di una cifra maggiore di zero per 9: in questo caso ci troviamo a che fare con un algoritmo.

Le varie tecniche che hai imparato per calcolare la somma col riporto, la sottrazione con il prestito, la moltiplicazione, la divisione sono tutti algoritmi. Una delle caratteristiche degli algoritmi è che non richiedono intelligenza per essere eseguiti in quanto sono processi meccanici nei quali ogni passo segue il precedente secondo un insieme di regole più o meno semplice.

D'altro canto la progettazione di algoritmi è molto interessante ed intellettualmente stimolante rappresentando il fulcro di ciò che chiamiamo programmazione.

Alcune delle cose più semplici che facciamo naturalmente, senza difficoltà o pensiero cosciente, sono tra le cose più difficili da esprimere sotto forma di algoritmo. Comprendere un linguaggio naturale è un buon esempio: lo sappiamo fare tutti ma finora nessuno è stato in grado di spiegare *come* ci riusciamo esprimendolo sotto forma di algoritmo.

13.8 Glossario

Funzione pura: funzione che non modifica gli oggetti ricevuti come parametri. La maggior parte delle funzioni pure sono produttive.

Modificatore: funzione che cambia uno o più oggetti ricevuti come parametri. La maggior parte dei modificatori non restituisce valori di ritorno.

Stile di programmazione funzionale: stile di programmazione dove la maggior parte delle funzioni è pura.

Sviluppo prototipale: tipo di sviluppo del programma a partire da un prototipo che viene gradualmente testato, esteso e migliorato.

Sviluppo pianificato: tipo di sviluppo del programma che prevede uno studio preventivo del problema da risolvere.

Algoritmo: serie di passi per risolvere una classe di problemi in modo meccanico.

Capitolo 14

Classi e metodi

14.1 Funzionalità orientate agli oggetti

Python è un **linguaggio di programmazione orientato agli oggetti** il che significa che fornisce il supporto alla **programmazione orientata agli oggetti**.

Non è facile definire cosa sia la programmazione orientata agli oggetti ma abbiamo già visto alcune delle sue caratteristiche:

- I programmi sono costituiti da definizioni di oggetti e definizioni di funzioni e la gran parte dell'elaborazione è espressa in termini di operazioni sugli oggetti.
- Ogni definizione di oggetto corrisponde ad un oggetto o concetto del mondo reale e le funzioni che operano su un oggetto corrispondono a modi reali di interazione tra cose reali.

Per esempio la classe `Tempo` definita nel capitolo 13 corrisponde al modo in cui tendiamo a pensare alle ore del giorno e le funzioni che abbiamo definite corrispondono al genere di operazioni che facciamo con gli orari. Le classi `Punto` e `Rettangolo` sono estremamente simili ai concetti matematici corrispondenti.

Finora non ci siamo avvantaggiati delle funzionalità di supporto della programmazione orientata agli oggetti fornite da Python. Sia ben chiaro che queste funzionalità non sono indispensabili in quanto forniscono solo una sintassi alternativa per fare qualcosa che possiamo fare in modi più tradizionali, ma in molti casi questa alternativa è più concisa e accurata.

Per esempio nel programma `Tempo` non c'è una chiara connessione tra la definizione della classe e le definizioni di funzioni che l'hanno seguita: un esame superficiale è sufficiente per accorgersi che tutte queste funzioni prendono almeno un oggetto `Tempo` come parametro.

Questa osservazione giustifica la presenza dei **metodi**. Ne abbiamo già visto qualcuno nel caso dei dizionari, quando abbiamo invocato `keys` e `values`. Ogni metodo è associato ad una classe ed è destinato ad essere invocato sulle istanze di quella classe.

I metodi sono simili alle funzioni con due differenze:

- I metodi sono definiti all'interno della definizione di classe per rendere più esplicita la relazione tra la classe ed i metodi corrispondenti.
- La sintassi per invocare un metodo è diversa da quella usata per chiamare una funzione.

Nelle prossime sezioni prenderemo le funzioni scritte nei due capitoli precedenti e le trasformeremo in metodi. Questa trasformazione è puramente meccanica e puoi farla seguendo una serie di semplici passi: se sei a tuo agio nel convertire tra funzione e metodo e viceversa riuscirai anche a scegliere di volta in volta la forma migliore.

14.2 StampaTempo

Nel capitolo 13 abbiamo definito una classe chiamata `Tempo` e scritto una funzione `StampaTempo`:

```
class Tempo:
  pass

def StampaTempo(Orario):
  print str(Orario.Ore) + ":" +
        str(Orario.Minuti) + ":" +
        str(Orario.Secondi)
```

Per chiamare la funzione abbiamo passato un oggetto `Tempo` come parametro:

```
>>> OraAttuale = Tempo()
>>> OraAttuale.Ore = 9
>>> OraAttuale.Minuti = 14
>>> OraAttuale.Secondi = 30
>>> StampaTempo(OraAttuale)
```

Per rendere `StampaTempo` un metodo tutto quello che dobbiamo fare è muovere la definizione della funzione all'interno della definizione della classe. Fai attenzione al cambio di indentazione:

```
class Tempo:
  def StampaTempo(Orario):
    print str(Orario.Ore) + ":" +   \
          str(Orario.Minuti) + ":" +  \
          str(Orario.Secondi)
```

Ora possiamo invocare `StampaTempo` usando la notazione punto.

```
>>> OraAttuale.StampaTempo()
```

Come sempre l'oggetto su cui il metodo è invocato appare prima del punto ed il nome del metodo subito dopo.

L'oggetto su cui il metodo è invocato è automaticamente assegnato al primo parametro, quindi nel caso di `OraAttuale` è assegnato a `Orario`.

Per convenzione il primo parametro di un metodo è chiamato `self`, traducibile in questo caso come "l'oggetto stesso".

Come nel caso di `StampaTempo(OraAttuale)`, la sintassi di una chiamata di funzione tradizionale suggerisce che la funzione sia l'agente attivo: equivale pressappoco a dire "`StampaTempo`! C'è un oggetto per te da stampare!"

Nella programmazione orientata agli oggetti sono proprio gli oggetti ad essere considerati l'agente attivo: un'invocazione del tipo `OraAttuale.StampaTempo()` significa "`OraAttuale`! Invoca il metodo per stampare il tuo valore!"

Questo cambio di prospettiva non sembra così utile ed effettivamente negli esempi che abbiamo visto finora è così. Comunque lo spostamento della responsabilità dalla funzione all'oggetto rende possibile scrivere funzioni più versatili e rende più immediati il mantenimento ed il riutilizzo del codice.

14.3 Un altro esempio

Convertiamo `Incremento` (dalla sezione 13.3) da funzione a metodo. Per risparmiare spazio eviteremo di riscrivere il metodo `StampaTempo` che abbiamo già definito ma tu lo devi tenere nella tua versione del programma:

```
class Tempo:
  ...
  def Incremento(self, Secondi):
    self.Secondi = Secondi + self.Secondi

    while self.Secondi >= 60:
      self.Secondi = self.Secondi - 60
      self.Minuti = self.Minuti + 1

    while self.Minuti >= 60:
      self.Minuti = self.Minuti - 60
      self.Ore = self.Ore + 1
```

D'ora in poi i tre punti di sospensione `...` all'interno del codice indicheranno che è stata omessa per questioni di leggibilità una parte del codice già definito in precedenza.

La trasformazione, come abbiamo già detto, è puramente meccanica: abbiamo spostato la definizione di una funzione all'interno di una definizione di classe e cambiato il nome del primo parametro.

Ora possiamo invocare `Incremento` come metodo.

```
OraAttuale.Incremento(500)
```

Ancora una volta l'oggetto su cui il metodo è invocato viene automaticamente assegnato al primo parametro, `self`. Il secondo parametro, `Secondi`, vale `500`.

> *Esercizio: converti* `ConverteInSecondi` *della sezione 13.5 a metodo della classe* `Tempo`*.*

14.4 Un esempio più complesso

La funzione `Dopo` è leggermente più complessa perché opera su due oggetti `Tempo` e non soltanto su uno com'è successo per i metodi appena visti. Uno dei parametri è chiamato `self`; l'altro non cambia:

```
class Tempo:
  ...
  def Dopo(self, Tempo2):
    if self.Ore > Tempo2.Ore:
      return 1
    if self.Ore < Tempo2.Ore:
      return 0

    if self.Minuti > Tempo2.Minuti:
      return 1
    if self.Minuti < Tempo2.Minuti:
      return 0

    if self.Secondi > Tempo2.Secondi:
      return 1
    return 0
```

Invochiamo questo metodo su un oggetto e passiamo l'altro come argomento:

```
if TempoCottura.Dopo(OraAttuale):
  print "Il pranzo e' pronto"
```

14.5 Argomenti opzionali

Abbiamo già visto delle funzioni predefinite che accettano un numero variabile di argomenti: `string.find` accetta due, tre o quattro argomenti.

Possiamo scrivere funzioni con una lista di argomenti opzionali. Scriviamo la nostra versione di `Trova` per farle fare la stessa cosa di `string.find`.

Ecco la versione originale che abbiamo scritto nella sezione 7.7:

```
def Trova(Stringa, Carattere):
  Indice = 0
  while Indice < len(Stringa):
    if Stringa[Indice] == Carattere:
      return Indice
```

```
    Indice = Indice + 1
  return -1
```

Questa è la versione aggiornata e migliorata:

```
def Trova(Stringa, Carattere, Inizio=0):
  Indice = Inizio
  while Inizio < len(Stringa):
    if Stringa[Indice] == Carattere:
      return Indice
    Indice = Indice + 1
  return -1
```

Il terzo parametro, `Inizio`, è opzionale perché abbiamo fornito il valore `0` di default. Se invochiamo `Trova` con solo due argomenti usiamo il valore di default per il terzo così da iniziare la ricerca dall'inizio della stringa:

```
>>> Trova("Mela", "l")
2
```

Se forniamo un terzo parametro questo **sovrascrive** il valore di default:

```
>>> Trova("Mela", "l", 3)
-1
```

> *Esercizio: aggiungi un quarto parametro,* `Fine`*, che specifica dove interrompere la ricerca.*
>
> *Attenzione: questo esercizio non è semplice come sembra. Il valore di default di* `Fine` *dovrebbe essere* `len(Stringa)` *ma questo non funziona. I valori di default sono valutati al momento della definizione della funzione, non quando questa è chiamata: quando* `Trova` *viene definita,* `Stringa` *non esiste ancora così non puoi conoscere la sua lunghezza. Trova un sistema per aggirare l'ostacolo.*

14.6 Il metodo di inizializzazione

Il **metodo di inizializzazione** è un metodo speciale invocato quando si crea un oggetto. Il nome di questo metodo è `__init__` (due caratteri di sottolineatura, seguiti da `init` e da altri due caratteri di sottolineatura). Un metodo di inizializzazione per la classe `Tempo` potrebbe essere:

```
class Tempo:
  def __init__(self, Ore=0, Minuti=0, Secondi=0):
    self.Ore = Ore
    self.Minuti = Minuti
    self.Secondi = Secondi
```

Non c'è conflitto tra l'attributo `self.Ore` e il parametro `Ore`. La notazione punto specifica a quale variabile ci stiamo riferendo.

Quando invochiamo il costruttore `Tempo` gli argomenti che passiamo sono girati a `__init__`:

```
>>> OraAttuale = Tempo(9, 14, 30)
>>> OraAttuale.StampaTempo()
>>> 9:14:30
```

Dato che i parametri sono opzionali possiamo anche ometterli:

```
>>> OraAttuale = Tempo()
>>> OraAttuale.StampaTempo()
>>> 0:0:0
```

Possiamo anche fornire solo il primo parametro:

```
>>> OraAttuale = Tempo(9)
>>> OraAttuale.StampaTempo()
>>> 9:0:0
```

o i primi due parametri:

```
>>> OraAttuale = Tempo(9, 14)
>>> OraAttuale.StampaTempo()
>>> 9:14:0
```

Infine possiamo anche passare un sottoinsieme dei parametri nominandoli esplicitamente:

```
>>> OraAttuale = Tempo(Secondi = 30, Ore = 9)
>>> OraAttuale.StampaTempo()
>>> 9:0:30
```

14.7 La classe `Punto` rivisitata

Riscriviamo la classe `Punto` che abbiamo già visto alla sezione 12.1 in uno stile più orientato agli oggetti:

```
class Punto:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

Il metodo di inizializzazione prende `x` e `y` come parametri opzionali. Il loro valore di default è 0.

Il metodo `__str__` ritorna una rappresentazione di un oggetto `Punto` sotto forma di stringa. Se una classe fornisce un metodo chiamato `__str__` questo sovrascrive il comportamento abituale della funzione `str` di Python.

```
>>> P = Punto(3, 4)
>>> str(P)
'(3, 4)'
```

La stampa di un oggetto `Punto` invoca `__str__` sull'oggetto: la definizione di `__str__` cambia dunque anche il comportamento di `print`:

```
>>> P = Punto(3, 4)
>>> print P
(3, 4)
```

Quando scriviamo una nuova classe iniziamo quasi sempre scrivendo `__init__` (la funzione che rende più facile istanziare oggetti) e `__str__` (utile per il debug).

14.8 Ridefinizione di un operatore

Alcuni linguaggi consentono di cambiare la definizione degli operatori predefiniti quando applicati a tipi definiti dall'utente. Questa caratteristica è chiamata **ridefinizione dell'operatore** (o "overloading dell'operatore") e si rivela molto utile soprattutto quando vogliamo definire nuovi tipi di operazioni matematiche.

Se vogliamo ridefinire l'operatore somma `+` scriveremo un metodo chiamato `__add__`:

```
class Punto:
  ...
  def __add__(self, AltroPunto):
    return Punto(self.x + AltroPunto.x, self.y + AltroPunto.y)
```

Come al solito il primo parametro è l'oggetto su cui è invocato il metodo. Il secondo parametro è chiamato `AltroPunto` per distinguerlo da `self`. Ora sommiamo due oggetti `Punto` restituendo la somma in un terzo oggetto `Punto` che conterrà la somma delle coordinate x e delle coordinate y.

Quando applicheremo l'operatore `+` ad oggetti `Punto` Python invocherà il metodo `__add__`:

```
>>>   P1 = Punto(3, 4)
>>>   P2 = Punto(5, 7)
>>>   P3 = P1 + P2
>>>   print P3
(8, 11)
```

L'espressione `P1 + P2` è equivalente a `P1.__add__(P2)` ma ovviamente più elegante.

> *Esercizio: aggiungi il metodo* `__sub__(self, AltroPunto)` *che ridefinisca l'operatore sottrazione per la classe* `Punto`*.*

Ci sono parecchi modi per ridefinire l'operatore moltiplicazione, aggiungendo il metodo `__mul__` o `__rmul__` o entrambi.

Se l'operatore a sinistra di `*` è un `Punto` Python invoca `__mul__` assumendo che anche l'altro operando sia un oggetto di tipo `Punto`. In questo caso si dovrà calcolare il **prodotto punto** dei due punti secondo le regole dell'algebra lineare:

```
def __mul__(self, AltroPunto):
  return self.x * AltroPunto.x + self.y * AltroPunto.y
```

Se l'operando a sinistra di `*` è un tipo primitivo (e quindi diverso da un oggetto `Punto`) e l'operando a destra è di tipo `Punto` Python invocherà `__rmul__` per calcolare una **moltiplicazione scalare**:

```
def __rmul__(self, AltroPunto):
  return Punto(AltroPunto * self.x,  AltroPunto * self.y)
```

Il risultato della moltiplicazione scalare è un nuovo punto le cui coordinate sono un multiplo di quelle originali. Se `AltroPunto` è un tipo che non può essere moltiplicato per un numero in virgola mobile `__rmul__` produrrà un errore in esecuzione.

Questo esempio mostra entrambi i tipi di moltiplicazione:

```
>>> P1 = Punto(3, 4)
>>> P2 = Punto(5, 7)
>>> print P1 * P2
43
>>> print 2 * P2
(10, 14)
```

Cosa accade se proviamo a valutare `P2 * 2`? Dato che il primo parametro è un `Punto` Python invoca `__mul__` con `2` come secondo argomento. All'interno di `__mul__` il programma prova ad accedere la coordinata `x` di `AltroPunto` e questo tentativo genera un errore dato che un numero intero non ha attributi:

```
>>> print P2 * 2
AttributeError: 'int' object has no attribute 'x'
```

Questo messaggio d'errore è effettivamente troppo sibiliino per risultare di una qualche utilità, e questo è ottimo esempio delle difficoltà che puoi incontrare nella programmazione ad oggetti: non è sempre semplice capire quale sia il codice che ha causato l'errore.

Per un trattato più esauriente sulla ridefinizione degli operatori vedi l'appendice B.

14.9 Polimorfismo

La maggior parte dei metodi che abbiamo scritto finora lavorano solo per un tipo specifico di dati. Quando crei un nuovo oggetto scrivi dei metodi che lavorano su oggetti di quel tipo.

Ci sono comunque operazioni che vorresti poter applicare a molti tipi come ad esempio le operazioni matematiche che abbiamo appena visto. Se più tipi di dato supportano lo stesso insieme di operazioni puoi scrivere funzioni che lavorano indifferentemente con ciascuno di questi tipi.

Per esempio l'operazione `MoltSomma` (comune in algebra lineare) prende tre parametri: il risultato è la moltiplicazione dei primi due e la successiva somma del terzo al prodotto. Possiamo scriverla così:

```
def MoltSomma(x, y, z):
  return x * y + z
```

Questo metodo lavorerà per tutti i valori di `x` e `y` che possono essere moltiplicati e per ogni valore di `z` che può essere sommato al prodotto.

Possiamo invocarla con valori numerici:

```
>>> MoltSomma(3, 2, 1)
7
```

o con oggetti di tipo `Punto`:

```
>>> P1 = Punto(3, 4)
>>> P2 = Punto(5, 7)
>>> print MoltSomma(2, P1, P2)
(11, 15)
>>> print MoltSomma(P1, P2, 1)
44
```

Nel primo caso il punto `P1` è moltiplicato per uno scalare e il prodotto è poi sommato a un altro punto (`P2`). Nel secondo caso il prodotto punto produce un valore numerico al quale viene sommato un altro valore numerico.

Una funzione che accetta parametri di tipo diverso è chiamata **polimorfica**.

Come esempio ulteriore consideriamo il metodo `DirittoERovescio` che stampa due volte una stringa, prima direttamente e poi all'inverso:

```
def DirittoERovescio(Stringa):
  import copy
  Rovescio = copy.copy(Stringa)
  Rovescio.reverse()
  print str(Stringa) + str(Rovescio)
```

Dato che il metodo `reverse` è un modificatore si deve fare una copia della stringa prima di rovesciarla: in questo modo il metodo `reverse` non modificherà la lista originale ma solo una sua copia.

Ecco un esempio di funzionamento di `DirittoERovescio` con le liste:

```
>>>   Lista = [1, 2, 3, 4]
>>>   DirittoERovescio(Lista)
[1, 2, 3, 4][4, 3, 2, 1]
```

Era facilmente intuibile che questa funzione riuscisse a maneggiare le liste. Ma può lavorare con oggetti di tipo `Punto`?

Per determinare se una funzione può essere applicata ad un tipo nuovo applichiamo la regola fondamentale del polimorfismo:

> **Se tutte le operazioni all'interno della funzione possono essere applicate ad un tipo di dato allora la funzione stessa può essere applicata al tipo.**

Le operazioni nel metodo `DirittoERovescio` includono `copy`, `reverse` e `print`.

`copy` funziona su ogni oggetto e abbiamo già scritto un metodo `__str__` per gli oggetti di tipo `Punto` così l'unica cosa che ancora ci manca è il metodo `reverse`:

```
def reverse(self):
  self.x , self.y = self.y, self.x
```

Ora possiamo passare `Punto` a `DirittoERovescio`:

```
>>>   P = Punto(3, 4)
>>>   DirittoERovescio(P)
(3, 4)(4, 3)
```

Il miglior tipo di polimorfismo è quello involontario, quando scopri che una funzione già scritta può essere applicata ad un tipo di dati per cui non era stata pensata.

14.10 Glossario

Linguaggio orientato agli oggetti: linguaggio che è dotato delle caratteristiche che facilitano la programmazione orientata agli oggetti, tipo la possibilità di definire classi e l'ereditarietà.

Programmazione orientata agli oggetti: stile di programmazione nel quale i dati e le operazioni che li manipolano sono organizzati in classi e metodi.

Metodo: funzione definita all'interno di una definizione di classe invocata su istanze di quella classe.

Ridefinire: rimpiazzare un comportamento o un valore di default, scrivendo un metodo con lo stesso nome o rimpiazzando un parametro di default con un valore particolare.

Metodo di inizializzazione: metodo speciale invocato automaticamente nel momento in cui viene creato un nuovo oggetto e usato per inizializzare gli attributi dell'oggetto stesso.

Ridefinizione dell'operatore: estensione degli operatori predefiniti (`+`, `-`, `*`, `>`, `<`, ecc.) per farli lavorare con i tipi definiti dall'utente.

Prodotto punto: operazione definita nell'algebra lineare che moltiplica due punti e produce un valore numerico.

Moltiplicazione scalare: operazione definita nell'algebra lineare che moltiplica ognuna delle coordinate di un punto per un valore numerico.

Funzione polimorfica: funzione che può operare su più di un tipo di dati. Se tutte le operazioni in una funzione possono essere applicate ad un tipo di dato allora la funzione può essere applicata al tipo.

Capitolo 15

Insiemi di oggetti

15.1 Composizione

Uno dei primi esempi di composizione che hai visto è stato l'uso di un'invocazione di un metodo all'interno di un'espressione. Un altro esempio è stata la struttura di istruzioni annidate, con un `if` all'interno di un ciclo `while` all'interno di un altro `if` e così via.

Dopo aver visto questo modo di operare e aver analizzato le liste e gli oggetti, non dovresti essere sorpreso del fatto che puoi anche creare liste di oggetti. Non solo: puoi creare oggetti che contengono liste come attributi, o liste che contengono liste, oggetti che contengono oggetti e così via.

In questo capitolo e nel prossimo vedremo alcuni esempi di queste combinazioni usando l'oggetto `Carta`.

15.2 Oggetto `Carta`

Se non hai dimestichezza con le comuni carte da gioco adesso è il momento di prendere in mano un mazzo di carte, altrimenti questo capitolo non avrà molto senso. Per i nostri scopi considereremo un mazzo di carte americano: questo mazzo è composto da 52 carte, ognuna delle quali appartiene a un seme (picche, cuori, quadri, fiori, nell'ordine di importanza nel gioco del bridge) ed è identificata da un numero da 1 a 13 (detto "rango"). I valori rappresentano, in ordine crescente, l'Asso, la serie numerica da 2 a 10, il Jack, la Regina ed il Re. A seconda del gioco a cui stai giocando il valore dell'Asso può essere considerato inferiore al 2 o superiore al Re.

Volendo definire un nuovo oggetto per rappresentare una carta da gioco è ovvio che gli attributi devono essere il `rango` ed il `seme`. Non è invece evidente di che *tipo* debbano essere gli attributi. Una possibilità è quella di usare stringhe contenenti il seme (`"Cuori"`) e il rango (`"Regina"`) solo che in questo modo non c'è un sistema semplice per vedere quale carta ha il rango o il seme più elevato.

Un'alternativa è quella di usare gli interi per **codificare** il rango e il seme. Con "codifica" non intendiamo crittografie o traduzioni in codice segreto ma semplicemente la definizione che lega una sequenza di numeri agli oggetti che essi vogliono rappresentare. Per esempio:

Picche	$\mapsto$	3
Cuori	$\mapsto$	2
Quadri	$\mapsto$	1
Fiori	$\mapsto$	0

Un utile effetto pratico di questa mappatura è il fatto che possiamo confrontare i semi tra di loro determinando subito quale vale di più. La mappatura per il rango è abbastanza ovvia: per le carte numeriche il rango è il numero della carta mentre per le carte figurate usiamo queste associazioni:

Asso	$\mapsto$	1
Jack	$\mapsto$	11
Regina	$\mapsto$	12
Re	$\mapsto$	13

Cominciamo con il primo abbozzo di definizione di `Carta` e come sempre forniamo anche un metodo di inizializzazione dei suoi attributi:

```
class Carta:
  def __init__(self, Seme=0, Rango=0):
    self.Seme = Seme
    self.Rango = Rango
```

Per creare un oggetto che rappresenta il 3 di fiori useremo:

```
TreDiFiori = Carta(0, 3)
```

dove il primo argomento (`0`) rappresenta il *seme* fiori ed il secondo (`3`) il *rango* della carta.

15.3 Attributi della classe e metodo `__str__`

Per stampare oggetti di tipo `Carta` in un modo facilmente comprensibile possiamo mappare i codici interi con stringhe. Assegniamo pertanto due liste di stringhe all'inizio della definizione della classe:

```
class Carta:

  ListaSemi = ["Fiori", "Quadri", "Cuori", "Picche"]
  ListaRanghi = ["impossibile", "Asso", "2", "3", "4", "5", "6",\
                 "7", "8", "9", "10", "Jack", "Regina", "Re"]

  def __init__(self, Seme=0, Rango=0):
    self.Seme = Seme
    self.Rango = Rango
```

```
  def __str__(self):
    return (self.ListaRanghi[self.Rango] + " di " +
            self.ListaSemi[self.Seme])
```

Le due liste sono in questo caso degli `attributi di classe` che sono definiti all'esterno dei metodi della classe e possono essere utilizzati da qualsiasi metodo della classe.

All'interno di `__str__` possiamo allora usare `ListaSemi` e `ListaRanghi` per far corrispondere i valori numerici di `Seme` e `Rango` a delle stringhe. Per fare un esempio l'espressione `self.ListaSemi[self.Seme]` significa "usa l'attributo `Seme` dell'oggetto `self` come indice nell'attributo di classe chiamato `ListaSemi` e restituisci la stringa appropriata".

Il motivo della presenza dell'elemento `"impossibile"` nel primo elemento di `ListaRanghi` è di agire come segnaposto per l'elemento 0 che non dovrebbe mai essere usato dato che il rango ha valori da 1 a 13. Meglio sprecare un elemento della lista piuttosto che dover scalare tutti i ranghi di una posizione e dover far corrispondere l'asso allo 0, il due all'1, il tre al 2, eccetera, con il rischio di sbagliare.

Con i metodi che abbiamo scritto finora possiamo già creare e stampare le carte:

```
>>> Carta1 = Carta(1, 11)
>>> print Carta1
Jack di Quadri
```

Gli attributi di classe come `ListaSemi` sono condivisi da tutti gli oggetti `Carta`. Il vantaggio è che possiamo usare qualsiasi oggetto `Carta` per accedere agli attributi di classe:

```
>>> Carta2 = Carta(1, 3)
>>> print Carta2
3 di Quadri
>>> print Carta2.ListaSemi[1]
Quadri
```

Lo svantaggio sta nel fatto che se modifichiamo un attributo di classe questo cambiamento si riflette in ogni istanza della classe. Per esempio se decidessimo di cambiare il seme "Quadri" in "Bastoni"...

```
>>> Carta1.ListaSemi[1] = "Bastoni"
>>> print Carta1
Jack di Bastoni
```

...*tutti* i Quadri diventerebbero dei Bastoni:

```
>>> print Carta2
3 di Bastoni
```

Non è solitamente una buona idea modificare gli attributi di classe.

15.4 Confronto tra carte

Per i tipi primitivi sono già definiti operatori condizionali (`<`, `>`, `==`, ecc.) che confrontano i valori e determinano se un operatore è più grande, più piccolo o uguale ad un altro. Per i tipi definiti dall'utente possiamo ridefinire il comportamento di questi operatori aggiungendo il metodo `__cmp__`. Per convenzione `__cmp__` prende due parametri, `self` e `Altro`, e ritorna 1 se il primo è il più grande, -1 se è più grande il secondo e 0 se sono uguali.

Alcuni tipi sono completamente ordinati, il che significa che puoi confrontare due elementi qualsiasi e determinare sempre quale sia il più grande tra di loro. Per esempio i numeri interi e quelli in virgola mobile sono completamente ordinati. Altri tipi sono disordinati, nel senso che non esiste un modo logico per stabilire quale sia il più grande, così come non è possibile stabilire tra una serie di colori quale sia il "minore".

L'insieme delle carte da gioco è parzialmente ordinato e ciò significa che qualche volta puoi confrontare due carte e qualche volta no. Per fare un esempio sai che il 3 di Fiori è più alto del 2 di Fiori e il 3 di Quadri più alto del 3 di Fiori. Fino a questo punto il loro valore relativo e il conseguente ordine sono chiari. Ma qual è la carta più alta se dobbiamo scegliere tra 3 di Fiori e 2 di Quadri? Una ha il rango più alto, l'altra il seme.

Per rendere confrontabili le carte dobbiamo innanzitutto decidere quale attributo sia il più importante, se il rango o il seme. La scelta è arbitraria e per il nostro studio decideremo che il seme ha priorità rispetto al rango.

Detto questo possiamo scrivere `__cmp__`:

```
def __cmp__(self, Altro):

  # controlla il seme
  if self.Seme > Altro.Seme: return 1
  if self.Seme < Altro.Seme: return -1

  # se i semi sono uguali controlla il rango
  if self.Rango > Altro.Rango: return 1
  if self.Rango < Altro.Rango: return -1

  # se anche i ranghi sono uguali le carte sono uguali!
  return 0
```

In questo tipo di ordinamento gli Assi hanno valore più basso dei 2.

> *Esercizio: modifica* `__cmp__` *così da rendere gli Assi più importanti dei Re.*

15.5 Mazzi

Ora che abbiamo oggetti per rappresentare le carte il passo più logico è quello di definire una classe per rappresentare il `Mazzo`. Il mazzo è composto di carte così ogni oggetto `Mazzo` conterrà una lista di carte come attributo.

Quella che segue è la definizione di classe della classe `Mazzo`. Il metodo di inizializzazione crea l'attributo `Carte` e genera le 52 carte standard:

```
class Mazzo:
  def __init__(self):
    self.Carte = []
    for Seme in range(4):
      for Rango in range(1, 14):
        self.Carte.append(Carta(Seme, Rango))
```

Il modo più semplice per creare un mazzo è per mezzo di un ciclo annidato: il ciclo esterno numera i semi da 0 a 3, quello interno i ranghi da 1 a 13. Dato che il ciclo esterno viene eseguito 4 volte e quello interno 13 il corpo è eseguito un totale di 52 volte (4 per 13). Ogni iterazione crea una nuova istanza di `Carta` con seme e rango correnti ed aggiunge la carta alla lista `Carte`.

Il metodo `append` lavora sulle liste ma non sulle tuple (che sono immutabili).

15.6 Stampa del mazzo

Com'è consueto dopo aver creato un nuovo tipo di oggetto è utile scrivere un metodo per poterne stampare il contenuto. Per stampare `Mazzo` attraversiamo la lista stampando ogni elemento `Carta`:

```
class Mazzo:
  ...
  def StampaMazzo(self):
    for Carta in self.Carte:
      print Carta
```

Come alternativa a `StampaMazzo` potremmo anche riscrivere il metodo `__str__` per la classe `Mazzo`. Il vantaggio nell'uso di `__str__` sta nel fatto che è più flessibile. Piuttosto che limitarsi a stampare il contenuto di un oggetto `__str__` genera infatti una rappresentazione sotto forma di stringa che altre parti del programma possono manipolare o che può essere memorizzata in attesa di essere usata in seguito.

Ecco una versione di `__str__` che ritorna una rappresentazione di un `Mazzo` come stringa. Tanto per aggiungere qualcosa facciamo anche in modo di indentare ogni carta rispetto alla precedente:

```
class Mazzo:
  ...
  def __str__(self):
    s = ""
    for i in range(len(self.Carte)):
      s = s + " "*i + str(self.Carte[i]) + "\n"
    return s
```

Questo esempio mostra un bel po' di cose.

Prima di tutto invece di attraversare `self.Carte` e assegnare ogni carta ad una variabile stiamo usando `i` come variabile del ciclo e come indice della lista delle carte.

In secondo luogo stiamo usando l'operatore di moltiplicazione delle stringhe per indentare le carte. L'espressione `" "*i` infatti produce un numero di spazi pari a `i`.

Terzo, invece di usare un comando `print` per stampare le carte usiamo la funzione `str`. Passare un oggetto come argomento a `str` è equivalente ad invocare il metodo `__str__` sull'oggetto.

Infine stiamo usando la variabile `s` come **accumulatore**. Inizialmente `s` è una stringa vuota. Ogni volta che passiamo attraverso il ciclo viene generata e concatenata a `s` una nuova stringa. Quando il ciclo termina `s` contiene la rappresentazione completa dell'oggetto `Mazzo` sotto forma di stringa:

```
>>> Mazzo1 = Mazzo()
>>> print Mazzo1
Asso di Fiori
 2 di Fiori
  3 di Fiori
   4 di Fiori
    5 di Fiori
     6 di Fiori
      7 di Fiori
       8 di Fiori
        9 di Fiori
         10 di Fiori
          Jack di Fiori
           Regina di Fiori
            Re di Fiori
             Asso di Quadri
              ...
```

Anche se il risultato appare come una serie di 52 righe (una per ogni carta) in realtà si tratta di una singola stringa che contiene caratteri di ritorno a capo per poter essere stampata su più righe.

15.7 Mescolare il mazzo

Se un mazzo è perfettamente mescolato ogni carta ha la stessa probabilità di comparire in una posizione qualsiasi.

Per mescolare il mazzo useremo la funzione `randrange` del modulo `random`. `randrange` prende due argomenti interi (`a` e `b`) e sceglie un numero casuale intero nell'intervallo `a <= x < b`. Dato che il limite superiore è escluso possiamo usare la lunghezza di una lista come secondo parametro avendo la garanzia della validità dell'indice. Questa espressione sceglie l'indice di una carta casuale nel mazzo:

```
random.randrange(0, len(self.Carte))
```

Un modo utile per mescolare un mazzo è scambiare ogni carta con un'altra scelta a caso. È possibile che la carta possa essere scambiata con se stessa ma questa situazione è perfettamente accettabile. Infatti se escludessimo questa possibilità l'ordine delle carte sarebbe meno casuale:

```
class Mazzo:
  ...
  def Mescola(self):
    import random
    NumCarte = len(self.Carte)
    for i in range(NumCarte):
      j = random.randrange(i, NumCarte)
      self.Carte[i], self.Carte[j] = self.Carte[j], self.Carte[i]
```

Piuttosto che partire dal presupposto che le carte del mazzo siano sempre 52 abbiamo scelto di ricavare la lunghezza della lista e memorizzarla in `NumCarte`.

Per ogni carta del mazzo abbiamo scelto casualmente una carta tra quelle non ancora mescolate. Poi abbiamo scambiato la carta corrente (`i`) con la carta selezionata (`j`). Per scambiare le due carte abbiamo usato un'assegnazione di una tupla, come si è già visto nella sezione 9.2:

```
self.Carte[i], self.Carte[j] = self.Carte[j], self.Carte[i]
```

> *Esercizio: riscrivi questa riga di codice senza usare un'assegnazione di una tupla.*

15.8 Rimuovere e distribuire le carte

Un altro metodo utile per la classe `Mazzo` è `RimuoviCarta` che permette di rimuovere una carta dal mazzo ritornando *vero* (1) se la carta era presente e *falso* (0) in caso contrario:

```
class Mazzo:
  ...
  def RimuoviCarta(self, Carta):
    if Carta in self.Carte:
      self.Carte.remove(Carta)
      return 1
    else:
      return 0
```

L'operatore `in` ritorna *vero* se il primo operando è contenuto nel secondo. Quest'ultimo deve essere una lista o una tupla. Se il primo operando è un oggetto, Python usa il metodo `__cmp__` dell'oggetto per determinare l'uguaglianza tra gli elementi della lista. Dato che `__cmp__` nella classe `Carta` controlla l'uguaglianza forte il metodo `RimuoviCarta` usa anch'esso l'uguaglianza forte.

Per distribuire le carte si deve poter rimuovere la prima carta del mazzo e il metodo delle liste `pop` fornisce un ottimo sistema per farlo:

```
class Mazzo:
  ...
  def PrimaCarta(self):
    return self.Carte.pop()
```

In realtà `pop` rimuove *l'ultima* carta della lista, così stiamo in effetti togliendo dal fondo del mazzo, ma dal nostro punto di vista questa anomalia è indifferente.

Una operazione che può essere utile è la funzione booleana `EVuoto` che ritorna *vero* (1) se il mazzo non contiene più carte:

```
class Mazzo:
  ...
  def EVuoto(self):
    return (len(self.Carte) == 0)
```

15.9 Glossario

Mappare: rappresentare un insieme di valori usando un altro insieme di valori e costruendo una mappa di corrispondenza tra i due insiemi.

Codificare: in campo informatico sinonimo di mappare.

Attributo di classe: variabile definita all'interno di una definizione di classe ma al di fuori di qualsiasi metodo. Gli attributi di classe sono accessibili da ognuno dei metodi della classe e sono condivisi da tutte le istanze della classe.

Accumulatore: variabile usata in un ciclo per accumulare una serie di valori, concatenati sotto forma di stringa o sommati per ottenere un valore totale.

Capitolo 16

Ereditarietà

16.1 Ereditarietà

La caratteristica più frequentemente associata alla programmazione ad oggetti è l'**ereditarietà** che è la capacità di definire una nuova classe come versione modificata di una classe già esistente.

Il vantaggio principale dell'ereditarietà è che si possono aggiungere nuovi metodi ad una classe senza dover modificare la definizione originale. È chiamata "ereditarietà" perché la nuova classe "eredita" tutti i metodi della classe originale. Estendendo questa metafora la classe originale è spesso definita "genitore" e la classe derivata "figlia" o "sottoclasse".

L'ereditarietà è una caratteristica potente e alcuni programmi possono essere scritti in modo molto più semplice e conciso grazie ad essa, dando inoltre la possibilità di personalizzare il comportamento di una classe senza modificare l'originale. Il fatto stesso che la struttura dell'ereditarietà possa riflettere quella del problema può rendere in qualche caso il programma più semplice da capire.

D'altro canto l'ereditarietà può rendere più difficile la lettura del programma, visto che quando si invoca un metodo non è sempre chiaro dove questo sia stato definito (se all'interno del genitore o delle classi da questo derivate) con il codice che deve essere rintracciato all'interno di più moduli invece che essere in un unico posto ben definito. Molte delle cose che possono essere fatte con l'ereditarietà possono essere di solito gestite elegantemente anche senza di essa, ed è quindi il caso di usarla solo se la struttura del problema la richiede: se usata nel momento sbagliato può arrecare più danni che apportare benefici.

In questo capitolo mostreremo l'uso dell'ereditarietà come parte di un programma che gioca a Old Maid, un gioco di carte piuttosto meccanico e semplice. Anche se implementeremo un gioco particolare uno dei nostri scopi è quello di scrivere del codice che possa essere riutilizzato per implementare altri tipi di giochi di carte.

16.2 Una mano

Per la maggior parte dei giochi di carte abbiamo la necessità di rappresentare una mano di carte. La mano è simile al mazzo, dato che entrambi sono insiemi di carte e richiedono metodi per aggiungere e rimuovere carte. Inoltre abbiamo bisogno sia per la mano che per il mazzo di poter mescolare le carte.

La mano si differenzia dal mazzo perché, a seconda del gioco, possiamo avere la necessità di effettuare su una mano alcuni tipi di operazioni che per un mazzo non avrebbero senso: nel poker posso avere l'esigenza di classificare una mano (full, colore, ecc.) o confrontarla con un'altra mano mentre nel bridge devo poter calcolare il punteggio di una mano per poter effettuare una puntata.

Questa situazione suggerisce l'uso dell'ereditarietà: se creiamo `Mano` come sottoclasse di `Mazzo` avremo immediatamente disponibili tutti i metodi di `Mazzo` con la possibilità di riscriverli o di aggiungerne altri.

Nella definizione della classe figlia il nome del genitore compare tra parentesi:

```
class Mano(Mazzo):
  pass
```

Questa istruzione indica che la nuova classe `Mano` eredita dalla classe già esistente `Mazzo`.

Il costruttore `Mano` inizializza gli attributi della mano, che sono il `Nome` e le `Carte`. La stringa `Nome` identifica la mano ed è probabilmente il nome del giocatore che la sta giocando: è un parametro opzionale che per default è una stringa vuota. `Carte` è la lista delle carte nella mano, inizializzata come lista vuota:

```
class Mano(Mazzo):
  def __init__(self, Nome=""):
    self.Carte = []
    self.Nome = Nome
```

In quasi tutti i giochi di carte è necessario poter aggiungere e rimuovere carte dalla mano. Della rimozione ce ne siamo già occupati, dato che `Mano` eredita immediatamente `RimuoviCarta` da `Mazzo`. Dobbiamo invece scrivere `AggiungeCarta`:

```
class Mano(Mazzo):

  def __init__(self, Nome=""):
    self.Carte = []
    self.Nome = Nome

  def AggiungeCarta(self,Carta) :
    self.Carte.append(Carta)
```

Il metodo di lista `append` aggiunge una nuova carta alla fine della lista di carte.

16.3 Distribuire le carte

Ora che abbiamo una classe `Mano` vogliamo poter spostare delle carte dal `Mazzo` alle singole mani. Non è immediatamente ovvio se questo metodo debba essere inserito nella classe `Mano` o nella classe `Mazzo` ma dato che opera su un mazzo singolo e (probabilmente) su più mani è più naturale inserirlo in `Mazzo`.

Il metodo `Distribuisci` dovrebbe essere abbastanza generale da poter essere usato in vari giochi e deve permettere la distribuzione tanto dell'intero mazzo che di una singola carta.

`Distribuisci` prende due argomenti: una lista (o tupla) di mani e il numero totale di carte da distribuire. Se non ci sono carte sufficienti per la distribuzione il metodo distribuisce quelle in suo possesso e poi si ferma:

```
class Mazzo:
  ...
  def Distribuisci(self, ListaMani, NumCarte=999):
    NumMani = len(ListaMani)
    for i in range(NumCarte):
      if self.EVuoto(): break         # si ferma se non ci sono
                                      # ulteriori carte
      Carta = self.PrimaCarta()       # prende la carta superiore
                                      # del mazzo
      Mano = ListaMani[i % NumMani]   # di chi e' il prossimo
                                      # turno?
      Mano.AggiungeCarta(Carta)       # aggiungi la carta alla
                                      # mano
```

Il secondo parametro, `NumCarte`, è opzionale; il valore di default è molto grande per essere certi che vengano distribuite tutte le carte del mazzo.

La variabile del ciclo `i` va da `0` a `NumCarte-1`. Ogni volta che viene eseguito il corpo del ciclo, la prima carta del mazzo viene rimossa usando il metodo di lista `pop` che rimuove e ritorna l'ultimo valore di una lista.

L'operatore modulo (`%`) ci permette di distribuire le carte in modo corretto, una carta alla volta per ogni mano: quando `i` è uguale al numero delle mani nella lista l'espressione `i % NumMani` restituisce 0 permettendo di ricominciare dal primo elemento della lista delle mani.

16.4 Stampa di una mano

Per stampare il contenuto di una mano possiamo avvantaggiarci dei metodi `StampaMazzo` e `__str__` ereditati da `Mazzo`. Per esempio:

```
>>> Mazzo1 = Mazzo()
>>> Mazzo1.Mescola()
>>> Mano1 = Mano("pippo")
>>> Mazzo1.Distribuisci([Mano1], 5)
>>> print Mano1
```

```
2 di Picche
 3 di Picche
  4 di Picche
   Asso di Cuori
    9 di Fiori
```

Anche se è comodo ereditare da metodi esistenti può essere necessario modificare il metodo `__str__` nella classe `Mano` per aggiungere qualche informazione, ridefinendo il metodo omonimo ereditato dalla classe `Mazzo`:

```
class Mano(Mazzo)
  ...
  def __str__(self):
    s = "La mano di " + self.Nome
    if self.EVuoto():
      s = s + " e' vuota\n"
    else:
      s = s + " contiene queste carte:\n"
    return s + Mazzo.__str__(self)
```

`s` è una stringa che inizialmente indica chi è il proprietario della mano. Se la mano è vuota vengono aggiunte ad `s` le parole `"e' vuota"` e viene ritornata `s`. IN caso contrario vengono aggiunte le parole `"contiene queste carte"` e la rappresentazione della mano sotto forma di stringa già vista in `Mazzo`, elaborata invocando il metodo `__str__` della classe `Mazzo` su `self`.

Potrebbe sembrarti strano il fatto di usare `self`, che si riferisce alla mano corrente, con un metodo appartenente alla classe `Mazzo`: ricorda che `Mano` è un tipo di `Mazzo`. Gli oggetti `Mano` possono fare qualsiasi cosa di cui è capace `Mazzo` e così è legale invocare un metodo `Mazzo` con la mano `self`.

In genere è sempre legale usare un'istanza di una sottoclasse invece di un'istanza della classe genitore.

16.5 La classe `GiocoDiCarte`

La classe `GiocoDiCarte` si occupa delle operazioni comuni in tutti i giochi di carte, quali possono essere la creazione del mazzo ed il mescolamento delle sue carte:

```
class GiocoDiCarte:
  def __init__(self):
    self.Mazzo = Mazzo()
    self.Mazzo.Mescola()
```

In questo primo caso abbiamo visto come il metodo di inizializzazione non si limiti ad assegnare dei valori agli attributi, ma esegua una elaborazione significativa.

Per implementare dei giochi specifici possiamo successivamente ereditare da `GiocoDiCarte` e aggiungere a questa classe le caratteristiche del nuovo gioco. Per fare un esempio scriveremo una simulazione di Old Maid.

L'obiettivo di Old Maid è quello di riuscire a sbarazzarsi di tutte le carte che si hanno in mano. Questo viene fatto eliminando coppie di carte che hanno lo stesso rango e colore: il 4 di fiori viene eliminato con il 4 di picche perché entrambi i segni sono neri; il jack di cuori con il jack di quadri perché entrambi sono rossi.

Per iniziare il gioco la Regina di Fiori è tolta dal mazzo per fare in modo che la Regina di Picche non possa essere eliminata durante la partita. Le 51 carte sono poi tutte distribuite una alla volta in senso orario ai giocatori e dopo la distribuzione tutti i giocatori scartano immediatamente quante più carte possibili eliminando le coppie presenti nella mano appena distribuita.

Quando non si possono più scartare carte il gioco ha inizio. A turno ogni giocatore pesca senza guardarla una carta dal giocatore che, in senso orario, ha ancora delle carte in mano. Se la carta scelta elimina una carta in mano la coppia viene rimossa. In caso contrario la carta scelta rimane in mano.

Alla fine della partita tutte le eliminazioni saranno state fatte ed il perdente è chi rimane con la Regina di Picche in mano.

Nella nostra simulazione del gioco il computer giocherà tutte le mani. Sfortunatamente alcune sottigliezze del gioco verranno perse: nel gioco reale chi si trova in mano la Regina di Picche farà di tutto per fare in modo che questa venga scelta da un vicino, disponendola in modo da facilitare un successo in tal senso. Il computer invece sceglierà le carte completamente a caso.

16.6 Classe `ManoOldMaid`

Una mano per giocare a Old Maid richiede alcune capacità che vanno oltre rispetto a quelle fornite da `Mano`. Sarà opportuno quindi definire una nuova classe `ManoOldMaid`, che erediterà i metodi da `Mano` e a questi metodi ne verrà aggiunto uno (`RimuoveCoppie`) per rimuovere le coppie di carte:

```
class ManoOldMaid(Mano):

  def RimuoveCoppie(self):
    Conteggio = 0
    CarteOriginali = self.Carte[:]
    for CartaOrig in CarteOriginali:
      CartaDaCercare = Carta(3-CartaOrig.Seme, CartaOrig.Rango)
      if CartaDaCercare in self.Carte:
        self.Carte.remove(CartaOrig)
        self.Carte.remove(CartaDaCercare)
        print "Mano di %s : %s elimina %s" %
              (self.Nome,CartaOrig,CartaDaCercare)
        Conteggio = Conteggio + 1
    return Conteggio
```

Iniziamo facendo una copia della lista di carte, così da poter attraversare la copia finché non rimuoviamo l'originale: dato che `self.Carte` viene modificata durante l'attraversamento, non possiamo di certo usarla per controllare tutti i suoi elementi. Python potrebbe essere confuso dal fatto di veder cambiare la lista che sta attraversando!

Per ogni carta della mano andiamo a controllare se quella che la elimina è presente nella stessa mano. La carta "eliminante" ha lo stesso rango e l'altro seme dello stesso colore di quella "eliminabile": l'espressione `3-Carta.Seme` serve proprio a trasformare una carta di Fiori (seme 0) in Picche (seme 3) e viceversa; una carta di Quadri (seme 1) in Cuori (seme 2) e viceversa.

Se entrambe le carte sono presenti sono rimosse con `RimuoveCoppie`:

```
>>> Partita = GiocoDiCarte()
>>> Mano1 = ManoOldMaid("Franco")
>>> Partita.Mazzo.Mescola([Mano1], 13)
>>> print Mano1
La mano di Franco contiene queste carte:
Asso di Picche
 2 di Quadri
  7 di Picche
   8 di Fiori
    6 di Cuori
     8 di Picche
      7 di Fiori
       Regina di Fiori
        7 di Quadri
         5 di Fiori
          Jack di Quadri
           10 di Quadri
            10 di Cuori

>>> Mano1.RimuoveCoppie()
Mano di Franco: 7 di Picche elimina 7 di Fiori
Mano di Franco: 8 di Picche elimina 8 di Fiori
Mano di Franco: 10 di Quadri elimina 10 di Cuori
>>> print Mano1
La mano di Franco contiene queste carte:
Asso di Picche
 2 di Quadri
  6 di Cuori
   Regina di Fiori
    7 di Quadri
     5 di Fiori
      Jack di Quadri
```

Nota che non c'è un metodo di inizializzazione `__init__` per la classe `ManoOldMaid` dato che l'abbiamo ereditato da `Mano`.

16.7 Classe GiocoOldMaid

Ora possiamo dedicarci al gioco vero e proprio: GiocoOldMaid è una sottoclasse di GiocoDiCarte con un metodo Giocatori che prende una lista di giocatori come parametro.

Dato che __init__ è ereditato da GiocoDiCarte un nuovo oggetto GiocoOldMaid contiene un mazzo già mescolato:

```
class GiocoOldMaid(GiocoDiCarte):

  def Partita(self, Nomi):

    # rimozione della regina di fiori
    self.Mazzo.RimuoviCarta(Carta(0,12))

    # creazione di una mano per ogni giocatore
    self.Mani = []
    for Nome in Nomi:
      self.Mani.append(ManoOldMaid(Nome))

    # distribuzione delle carte
    self.Mazzo.Distribuisci(self.Mani)
    print "---------- Le carte sono state distribuite"
    self.StampaMani()

    # toglie le coppie iniziali
    NumCoppie = self.RimuoveTutteLeCoppie()
    print "---------- Coppie scartate, inizia la partita"
    self.StampaMani()

    # gioca finche' non sono state fatte 25 coppie
    Turno = 0
    NumMani = len(self.Mani)
    while NumCoppie < 25:
      NumCoppie = NumCoppie + self.GiocaUnTurno(Turno)
      Turno = (Turno + 1) % NumMani

    print "---------- La partita e' finita"
    self.StampaMani()
```

Alcuni dei passi della partita sono stati separati in metodi singoli per ragioni di chiarezza anche se dal punto di vista del programma questo non era strettamente necessario.

RimuoveTutteLeCoppie attraversa la lista di mani e invoca RimuoveCoppie su ognuna:

```
class GiocoOldMaid(GiocoDiCarte):
  ...
  def RimuoveTutteLeCoppie(self):
```

```
    Conteggio = 0
    for Mano in self.Mani:
      Conteggio = Conteggio + Mano.RimuoveCoppie()
    return Conteggio
```

> *Esercizio: scrivi* `StampaMani` *che attraversa* `self.Mani` *e stampa ciascuna mano.*

`Conteggio` è un accumulatore che tiene traccia del numero di coppie rimosse dall'inizio della partita: quando il numero totale di coppie raggiunge 25 sono state rimosse dalle mani esattamente 50 carte, e ciò significa che è rimasta solo una carta (la Regina di Picche) ed il gioco è finito.

La variabile `Turno` tiene traccia di quale giocatore debba giocare. Parte da 0 e viene incrementata di 1 ad ogni mano. Quando arriva a `NumMani` l'operatore modulo `%` la riporta a 0.

Il metodo `GiocaUnTurno` prende un parametro dal giocatore che sta giocando. Il valore ritornato è il numero di coppie rimosse durante il turno:

```
class GiocoOldMaid(GiocoDiCarte):
  ...
  def GiocaUnTurno(self, Giocatore):
    if self.Mani[Giocatore].EVuoto():
      return 0
    Vicino = self.TrovaVicino(Giocatore)
    CartaScelta = self.Mani[Vicino].PrimaCarta()
    self.Mani[Giocatore].AggiungeCarta(CartaScelta)
    print "Mano di", self.Mani[Giocatore].Nome, \
          ": scelta", CartaScelta
    Conteggio = self.Mani[Giocatore].RimuoveCoppie()
    self.Mani[Giocatore].Mescola()
    return Conteggio
```

Se la mano di un giocatore è vuota quel giocatore è fuori dal gioco e non fa nulla. Il valore di ritorno in questo caso è 0.

In caso contrario un turno consiste nel trovare il primo giocatore in senso orario che abbia delle carte in mano, prendergli una carta e cercare coppie da rimuovere dopo avere aggiunto la carta scelta alla mano. Prima di tornare le carte in mano devono essere mescolate così che la scelta del prossimo giocatore sia ancora una volta casuale.

Il metodo `TrovaVicino` inizia con il giocatore all'immediata sinistra e continua in senso orario finché non trova qualcuno che ha ancora carte in mano:

```
class GiocoOldMaid(GiocoDiCarte):
  ...
  def TrovaVicino(self, Giocatore):
    NumMani = len(self.Mani)
    for Prossimo in range(1,NumMani):
      Vicino = (Giocatore + Prossimo) % NumMani
```

```
      if not self.Mani[Vicino].EVuoto():
        return Vicino
```

Se `TrovaVicino` dovesse effettuare un giro completo dei giocatori senza trovare qualcuno con delle carte in mano tornerebbe `None` e causerebbe un errore da qualche parte del programma. Fortunatamente possiamo provare che questo non succederà mai, sempre che la condizione di fine partita sia riconosciuta correttamente.

Abbiamo omesso il metodo `StampaMani` dato che puoi scriverlo tu senza problemi.

La stampa che mostriamo in seguito mostra una partita effettuata usando le sole quindici carte di valore più elevato (i 10, i jack, le regine ed i re), ed è stata ridotta per questioni di spazio. La partita ha visto come protagonisti tre giocatori: Allen, Jeff e Chris. Con un mazzo così piccolo il gioco si ferma dopo aver rimosso 7 coppie invece delle consuete 25.

```
>>> import Carte
>>> Gioco = Carte.GiocoOldMaid()
>>> Gioco.Partita(["Allen","Jeff","Chris"])
---------- Le carte sono state distribuite
La mano di Allen contiene queste carte:
Re di Cuori
 Jack di Fiori
  Regina di Picche
   Re di Picche
    10 di Quadri

La mano di Jeff contiene queste carte:
Regina di Cuori
 Jack di Picche
  Jack di Cuori
   Re di Quadri
    Regina di Quadri

La mano di Chris contiene queste carte:
Jack di Quadri
 Re di Fiori
  10 di Picche
   10 di Cuori
    10 di Fiori

Mano di Jeff: Regina di Cuori elimina Regina di Quadri
Mano di Chris: 10 di Picche elimina 10 di Fiori
---------- Coppie scartate, inizia la partita
La mano di Allen contiene queste carte:
Re di Cuori
 Jack di Fiori
  Regina di Picche
   Re di Picche
```

```
    10 di Quadri

La mano di Jeff contiene queste carte:
Jack di Picche
 Jack di Cuori
  Re di Quadri

La mano di Chris contiene queste carte:
Jack di Quadri
 Re di Fiori
  10 di Cuori

Mano di Allen: scelta Re di Quadri
Mano di Allen: Re di Cuori elimina Re di Quadri
Mano di Jeff: scelta 10 di Cuori
Mano di Chris: scelta Jack di Fiori
Mano di Allen: scelta Jack di Cuori
Mano di Jeff: scelta Jack di Quadri
Mano di Chris: scelta Regina di Picche
Mano di Allen: scelta Jack di Quadri
Mano di Allen: Jack di Cuori elimina Jack di Quadri
Mano di Jeff: scelta Re di Fiori
Mano di Chris: scelta Re di Picche
Mano di Allen: scelta 10 di Cuori
Mano di Allen: 10 di Quadri elimina 10 di Cuori
Mano di Jeff: scelta Regina di Picche
Mano di Chris: scelta Jack di Picche
Mano di Chris: Jack di Fiori elimina Jack di Picche
Mano di Jeff: scelta Re di Picche
Mano di Jeff: Re di Fiori elimina Re di Picche
---------- La partita e' finita
La mano di Allen e' vuota

La mano di Jack contiene queste carte:
Regina di Picche

La mano di Chris e' vuota
```

Così Jeff ha perso.

16.8 Glossario

Ereditarietà: capacità di definire una nuova classe come versione modificata di una classe precedentemente definita.

Classe genitore: classe da cui si deriva un'altra classe.

Classe figlia: nuova classe creata derivandola da una classe già esistente; è anche chiamata "sottoclasse".

Capitolo 17

Liste linkate

17.1 Riferimenti interni

Abbiamo visto esempi di attributi che si riferiscono ad altri oggetti (**riferimenti interni**, vedi sezione 12.8). Una struttura di dati piuttosto comune, la **lista linkata**, fa uso di questa caratteristica.

Le liste linkate sono costituite da **nodi** ed ognuno di questi nodi contiene il riferimento al successivo nodo della lista ed un'unità di dati utili chiamata **contenuto**.

Una lista linkata è considerata una **struttura di dati ricorsiva** perché la sua definizione è di per sé ricorsiva:

> *Una lista linkata è:*
>
> - *una lista vuota, rappresentata da* `None`*, oppure*
> - *un nodo che contiene un oggetto "contenuto" ed un riferimento ad una lista linkata.*

Le strutture di dati di tipo ricorsivo sono gestite da metodi ricorsivi.

17.2 La classe `Nodo`

Come abbiamo già visto in occasione della scrittura di nuove classi, cominciamo a scrivere la classe Nodo dalla sua inizializzazione e dal metodo `__str__` così da poter testare immediatamente il meccanismo di creazione e visualizzazione del nuovo tipo:

```
class Nodo:
  def __init__(self, Contenuto=None, ProssimoNodo=None):
    self.Contenuto = Contenuto
    self.ProssimoNodo  = ProssimoNodo
```

```
  def __str__(self):
    return str(self.Contenuto)
```

Abbiamo definito come opzionali i parametri per il metodo di inizializzazione: di default sia `Contenuto` che il link `ProssimoNodo` hanno valore `None`.

La rappresentazione a stringa del nodo è solo la stampa del suo contenuto: dato che alla funzione `str` può essere passato qualsiasi tipo di valore possiamo memorizzare nella lista ogni tipo di dato.

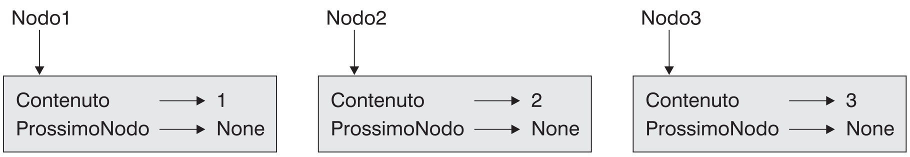
Per testare l'implementazione possiamo creare un `Nodo` e stamparne il valore:

```
>>> Nodo1 = Nodo("test")
>>> print Nodo1
test
```

Per rendere il tutto più interessante abbiamo bisogno di una lista che contiene più di un nodo:

```
>>> Nodo1 = Nodo(1)
>>> Nodo2 = Nodo(2)
>>> Nodo3 = Nodo(3)
```

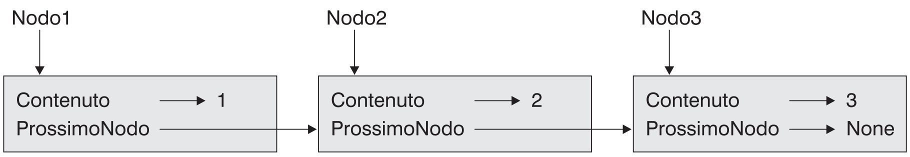
Questo codice crea tre nodi ma non siamo in presenza di una lista dato che questi nodi non sono **linkati** (collegati uno all'altro). Il diagramma di stato in questo caso è:



Per linkare i nodi dobbiamo fare in modo che il primo si riferisca al secondo, ed il secondo al terzo:

```
>>> Nodo1.ProssimoNodo = Nodo2
>>> Nodo2.ProssimoNodo = Nodo3
```

Il riferimento del terzo nodo è `None` e questo indica che ci troviamo alla fine della lista. Ecco il nuovo diagramma di stato:



Ora sai come creare nodi e come linkarli in liste. Ciò che probabilmente è meno chiaro è il motivo per cui questo possa rivelarsi utile.

17.3 Liste come collezioni

Le liste sono utili perché forniscono un modo per assemblare più oggetti in una entità singola talvolta chiamata **collezione**. Nell'esempio che abbiamo visto il primo nodo serve come riferimento all'intera lista dato che ne rappresenta il punto di partenza.

Per passare una lista di questo tipo come parametro ad una funzione dobbiamo passare quindi soltanto il riferimento al suo primo nodo. Per fare un esempio, la funzione `StampaLista` prende un singolo nodo come argomento, considerandolo l'inizio della lista e stampa il contenuto di ogni nodo finché non viene raggiunta la fine della lista:

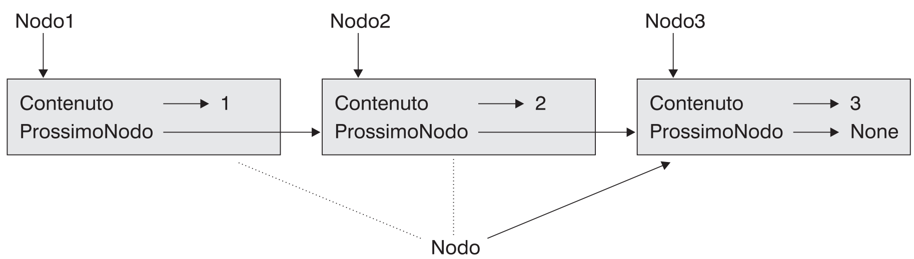
```
def StampaLista(Nodo):
  while Nodo:
    print Nodo,
    Nodo = Nodo.ProssimoNodo
  print
```

Per invocare questo metodo passiamo un riferimento al primo nodo:

```
>>> StampaLista(Nodo1)
1 2 3
```

All'interno di `StampaLista` abbiamo un riferimento al primo nodo della lista ma non c'è alcuna variabile che si riferisce agli altri nodi: per passare da un nodo al successivo usiamo il valore `Nodo.ProssimoNodo`, usando la variabile `Nodo` per riferirsi ad ognuno dei nodi in successione.

Questo diagramma mostra il valore di `Lista` ed il valore assunto da `Nodo`:



> *Esercizio: per convenzione le liste sono stampate tra parentesi quadrate con virgole che ne separano gli elementi, come in* `[1, 2, 3]`*. Modifica* `StampaLista` *così da generare una stampa in questo formato.*

17.4 Liste e ricorsione

Data la sua natura ricorsiva è intuitivo esprimere molte operazioni sulle liste con metodi ricorsivi. Questo è un algoritmo per stampare una lista a partire dall'ultimo elemento:

1. Separa la lista in due parti: il primo nodo (chiamato testa) ed il resto (la coda).

2. Stampa la coda in ordine inverso.

3. Stampa la testa.

Logicamente il passo 2, la chiamata ricorsiva, parte dal presupposto che ci sia un metodo per stampare la lista al contrario. Se partiamo dal presupposto che la chiamata ricorsiva funziona correttamente questo algoritmo lavora in modo corretto.

Tutto ciò di cui abbiamo bisogno è un caso base ed un modo per verificare che per ogni tipo di lista riusciremo ad arrivare al caso base per interrompere la serie di chiamate ricorsive. Data la definizione ricorsiva della lista un caso base intuitivo è la lista vuota, rappresentata da `None`:

```
def StampaInversa(Lista):
  if Lista == None: return
  Testa = Lista
  Coda = Lista.ProssimoNodo
  StampaInversa(Coda)
  print Testa,
```

La prima riga gestisce il caso base senza fare niente. Le due righe successive dividono la lista in due parti (`Testa` e `Coda`). Le ultime due righe stampano la lista. Ricorda che la virgola alla fine del `print` evita la stampa del ritorno a capo tra un nodo e l'altro.

Invochiamo questo metodo come abbiamo fatto con `StampaLista`:

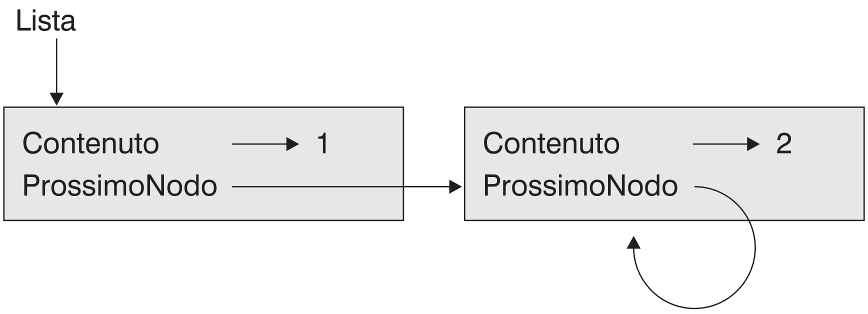
```
>>> StampaInversa(Nodo1)
3 2 1
```

Potresti chiederti perché `StampaLista` e `StampaInversa` sono funzioni e non metodi nella classe `Nodo`. La ragione è che vogliamo usare il valore `None` per rappresentare la lista vuota e non è lecito invocare un metodo su `None`. Questa limitazione in effetti rende poco pulito il codice, costringendo alla sua implementazione senza poter fare uso di uno stile orientato agli oggetti.

17.5 Liste infinite

Possiamo provare che `StampaInversa` giungerà sempre alla fine, raggiungendo il caso base? La risposta è no e infatti la sua chiamata causerà un errore in esecuzione nel caso in cui la lista passata come parametro sia di tipo particolare.

Non c'è nulla che vieti ad un nodo di fare riferimento ad un nodo precedente della lista o addirittura a se stesso. Questa figura mostra una lista di due nodi ognuno dei quali si riferisce a se stesso:



Se invocassimo `StampaLista` o `StampaInversa` su questa lista si creerebbe una ricorsione infinita: questo tipo di comportamento rende particolarmente difficile lavorare con le liste...

Ciononostante le liste infinite possono rivelarsi molto utili in (poche) occasioni particolari, come quando vogliamo rappresentare un numero come lista di cifre usando una lista infinita per la descrizione della parte decimale periodica.

Ci rimane comunque il problema che non possiamo dimostrare che `StampaLista` e `StampaInversa` raggiungono sempre il caso base. Il meglio che possiamo fare è stabilire una **precondizione**, assumendo che "se non sono presenti anelli all'interno della lista questi metodi termineranno". La precondizione impone una limitazione ai parametri e descrive il comportamento di un metodo nel caso essa venga soddisfatta. Vediamo subito qualche esempio.

17.6 Il teorema dell'ambiguità fondamentale

Una parte di `StampaInversa` aveva qualcosa di sospetto:

```
Testa = Lista
Coda = Lista.ProssimoNodo
```

Dopo la prima assegnazione `Testa` e `Lista` hanno lo stesso tipo e lo stesso valore. Perché dunque abbiamo creato una nuova variabile?

La ragione è che le due variabili giocano ruoli differenti. Pensiamo a `Testa` come riferimento ad un singolo nodo e a `Lista` come riferimento al primo nodo della lista. Questi "ruoli" non sono espressamente necessari al programma, ma sono molto utili per chiarire il concetto al programmatore.

In generale non possiamo dire quale ruolo giochi una variabile semplicemente guardando un programma. Spesso si usano nomi come `Nodo` e `Lista` per documentare l'uso della variabile e si introducono variabili addizionali solo per rendere meno ambiguo il codice al momento della lettura.

Avremmo anche potuto scrivere `StampaInversa` senza `Testa` e `Coda`. Il risultato sarebbe stato più conciso, ma decisamente meno chiaro:

```
def StampaInversa(Lista) :
  if Lista == None : return
  StampaInversa(Lista.ProssimoNodo)
  print Lista,
```

Con un'attenzione alle due chiamate di funzione è necessario ricordarci che `StampaInversa` tratta il suo argomento `Lista` come una collezione e `print` il proprio come un oggetto singolo.

Il **teorema dell'ambiguità fondamentale** descrive l'ambiguità inerente al riferimento ad un nodo:

> **Una variabile che si riferisce ad un nodo può trattare il nodo come oggetto singolo o come primo elemento di una lista di nodi linkati.**

17.7 Modifica delle liste

Ci sono due modi per modificare una lista linkata: possiamo cambiare il contenuto di uno dei nodi o aggiungere, rimuovere o riordinare i nodi.

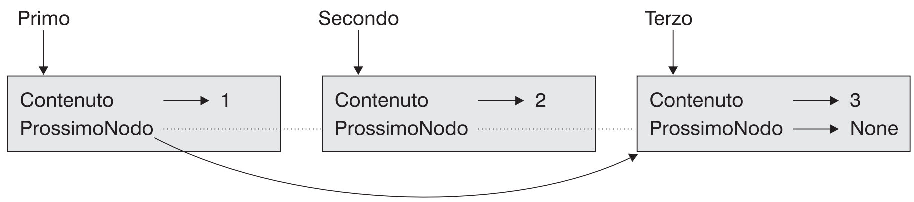
Come esempio scriviamo un metodo per rimuovere il secondo nodo di una lista, ritornando un riferimento al nodo rimosso:

```
def RimuoviSecondo(Lista):
  if Lista == None: return
  Primo = Lista
  Secondo = Lista.ProssimoNodo
  # il primo nodo deve riferirsi al terzo
  Primo.ProssimoNodo = Secondo.ProssimoNodo
  # separa il secondo nodo dal resto della lista
  Secondo.ProssimoNodo = None
  return Secondo
```

Ancora una volta abbiamo usato delle variabili temporanee per rendere il codice più leggibile. Ecco come usare questo metodo:

```
>>> StampaLista(Nodo1)
1 2 3
>>> Rimosso = RimuoviSecondo(Nodo1)
>>> StampaLista(Rimosso)
2
>>> StampaLista(Nodo1)
1 3
```

Questo diagramma di stato mostra l'effetto dell'operazione:



Cosa succede se invochi questo metodo e passi una lista composta da un solo elemento (**elemento singolo**)? Cosa succede se passi come argomento una lista vuota? C'è una precondizione per questo metodo? Se esiste riscrivi il metodo per gestire gli eventuali problemi.

17.8 Metodi contenitore e aiutante

Spesso è utile dividere un'operazione su una lista in due metodi. Per esempio per stampare una lista al contrario secondo il formato convenzionale `[3, 2, 1]` possiamo usare il metodo `StampaInversa` per stampare `3, 2,` ma abbiamo bisogno di un metodo diverso per stampare le parentesi ed il primo nodo. Chiamiamo questo metodo `StampaInversaFormato`:

```
def StampaInversaFormato(Lista) :
  print "[",
  if Lista != None :
    Testa = Lista
    Coda = Lista.ProssimoNodo
    StampaInversa(Coda)
    print Testa,
  print "]",
```

Ancora una volta è una buona idea testare questo metodo per vedere se funziona correttamente anche in casi particolari, quando cioè una lista è vuota o composta da un solo elemento.

Quando usiamo questo metodo da qualche parte nel programma invochiamo direttamente `StampaInversaFormato` e questa invoca a sua volta `StampaInversa`. In questo senso `StampaInversaFormato` agisce come un **contenitore** che usa `StampaInversa` come **aiutante**.

17.9 La classe `ListaLinkata`

Dal modo in cui abbiamo implementato le liste sorgono dei problemi concettuali piuttosto sottili. Procedendo in modo diverso dal consueto proporremo un'implementazione alternativa spiegando solo in seguito quali problemi vengono risolti da questa nuova versione.

Creiamo innanzitutto una nuova classe chiamata `ListaLinkata`. I suoi attributi sono un intero che contiene la lunghezza della lista e il riferimento al primo nodo. Gli oggetti `ListaLinkata` ci serviranno per gestire liste di oggetti `Nodo`:

```
class ListaLinkata:
  def __init__(self) :
    self.Lunghezza = 0
    self.Testa = None
```

Una cosa positiva per quanto concerne la classe `ListaLinkata` è che fornisce un posto naturale dove inserire funzioni contenitore quale `StampaInversaFormato` e che possiamo far diventare metodi della classe:

```
class ListaLinkata:
  ...
  def StampaInversa(self):
    print "[",
    if self.Testa != None:
      self.Testa.StampaInversa()
    print "]",

class Nodo:
  ...
  def StampaInversa(self):
    if self.ProssimoNodo != None:
      Coda = self.ProssimoNodo
      Coda.StampaInversa()
    print self.Contenuto,
```

Per rendere le cose più interessanti, rinominiamo `StampaInversaFormato`. Ora abbiamo due metodi chiamati `StampaInversa`: quello nella classe `Nodo` che è l'aiutante e quello nella classe `ListaLinkata` che è il contenitore. Quando il metodo contenitore invoca `self.Testa.StampaInversa` sta in effetti invocando l'aiutante dato che `self.Testa` è un oggetto di tipo `Nodo`.

Un altro beneficio della classe `ListaLinkata` è che rende semplice aggiungere o rimuovere il primo elemento di una lista. `AggiuntaPrimo` è il metodo di `ListaLinkata` per aggiungere un contenuto all'inizio di una lista:

```
class ListaLinkata:
  ...
  def AggiuntaPrimo(self, Contenuto):
    NodoAggiunto = Nodo(Contenuto)
    NodoAggiunto.ProssimoNodo = self.Testa
    self.Testa = NodoAggiunto
    self.Lunghezza = self.Lunghezza + 1
```

Come sempre occorre verificare che questo codice funzioni correttamente anche nel caso di liste speciali: cosa succede se la lista è inizialmente vuota?

17.10 Invarianti

Alcune liste sono "ben formate" mentre altre non lo sono. Se una lista contiene un anello questo può creare problemi a un certo numero dei nostri metodi, tanto che potremmo richiedere solo liste che non contengono anelli al loro interno. Un altro prerequisito è che il valore di `Lunghezza` nell'oggetto `ListaLinkata` corrisponda sempre al numero di nodi della lista.

Prerequisiti come questi sono chiamati **invarianti** perché dovrebbero essere sempre verificati in ogni momento per ogni oggetto della classe. Specificare gli invarianti degli oggetti è una pratica di programmazione molto indicata in quanto consente di rendere molto più facile la verifica del codice, il controllo dell'integrità delle strutture e il riconoscimento degli errori.

Una cosa che può rendere confusi per quanto riguarda gli invarianti è che a volte i prerequisiti che essi rappresentano possono essere violati, anche se solo temporaneamente: nel metodo `AggiungiPrimo`, dopo aver aggiunto il nodo ma prima di avere aggiornato `Lunghezza`, il prerequisito invariante non è soddisfatto. Questo tipo di violazione è accettabile, dato che spesso è impossibile modificare un oggetto senza violare un invariante almeno per un breve istante. Normalmente richiediamo che qualsiasi metodo che si trovi a violare un invariante lo ripristini non appena possibile.

Se l'invariante è violato in una parte significativa del codice è molto importante commentare questo comportamento anomalo per evitare che, anche a distanza di tempo, possano essere richieste delle operazioni che dipendono dall'integrità dei dati proprio dove questi dati non sono corretti.

17.11 Glossario

Riferimento interno: riferimento depositato in un attributo di un oggetto.

Lista linkata: struttura di dati che implementa una collezione usando una sequenza di nodi linkati.

Nodo: elemento di una lista solitamente implementato come un oggetto che contiene un riferimento ad un altro oggetto dello stesso tipo.

Contenuto: insieme dei dati utili contenuti in un nodo.

Link: riferimento interno ad un oggetto usato per legarlo ad un altro oggetto.

Precondizione: condizione che deve essere vera per permettere ad un metodo di funzionare in modo corretto.

Teorema dell'ambiguità fondamentale: il riferimento ad un nodo di una lista può essere considerato sia un singolo oggetto che il primo di una lista di nodi.

Elemento singolo: lista linkata composta da un singolo nodo.

Contenitore: metodo che agisce da interfaccia tra una funzione chiamante e un metodo aiutante, spesso semplificando l'uso del metodo aiutante o rendendo l'invocazione più immune da errori.

Aiutante: metodo che non è invocato direttamente da una funzione chiamate ma che è usato da un altro metodo per portare a termine una parte di un'operazione.

Invariante: condizione che deve essere vera per un oggetto in ogni momento, con l'unica eccezione degli istanti in cui l'oggetto è in fase di modifica.

Capitolo 18

Pile

18.1 Tipi di dati astratti

Tutti i tipi di dati che hai visto finora sono concreti, nel senso che abbiamo completamente specificato quale sia la loro implementazione. La classe `Carta` rappresenta una carta da gioco usando due numeri interi: come abbiamo detto durante lo sviluppo della classe questa non è l'unica implementazione possibile ma ne esistono infinite altre.

Un **tipo di dato astratto** (TDA) specifica un insieme di operazioni (o metodi) e la loro semantica (cosa fa ciascuna operazione) ma senza specificare la loro implementazione: questa caratteristica è ciò che lo rende astratto.

Per che cosa è utile questa "astrazione"?

- Semplifica il compito di specificare un algoritmo, dato che puoi decidere cosa dovranno fare le operazioni senza dover pensare allo stesso tempo a come implementarle.

- Ci sono molti modi per implementare un TDA e può essere utile scrivere un solo algoritmo in grado di funzionare per ciascuna delle possibili implementazioni.

- TDA molto ben conosciuti, tipo la Pila (o Stack) che vedremo in questo capitolo, sono spesso implementati nelle librerie standard dei vari linguaggi di programmazione così da poter essere usati da molti programmatori senza dover essere reinventati ogni volta.

- Le operazioni sui TDA forniscono un linguaggio di alto livello che consente di specificare e descrivere gli algoritmi.

Quando parliamo di TDA spesso distinguiamo il codice che usa il TDA (**cliente**) dal codice che lo implementa (**fornitore**).

18.2 Il TDA Pila

In questo capitolo esamineremo la **pila**, un tipo di dato astratto molto comune. Una pila è una collezione e cioè una struttura di dati che contiene elementi multipli. Altre collezioni che abbiamo già visto sono i dizionari e le liste.

Un TDA è definito dalle operazioni che possono essere effettuate su di esso e che sono chiamate **interfaccia**. L'interfaccia per una pila consiste di queste operazioni:

`__init__`: Inizializza un pila vuota.

`Push`: Aggiunge un elemento alla pila.

`Pop`: Rimuove e ritorna un elemento dalla pila. L'elemento tornato è sempre l'ultimo inserito.

`EVuota`: Controlla se la pila è vuota.

Una pila è spesso chiamata struttura di dati LIFO ("last in/first out", ultimo inserito, primo fuori) perché l'ultimo elemento inserito in ordine di tempo è il primo ad essere rimosso: un esempio è una serie di piatti da cucina sovrapposti, ai quali aggiungiamo ogni ulteriore piatto appoggiandolo sopra agli altri, ed è proprio dall'alto che ne preleviamo uno quando ci serve.

18.3 Implementazione delle pile con le liste di Python

Le operazioni che Python fornisce per le liste sono simili a quelle definite per la nostra pila. L'interfaccia non è proprio quella che ci si aspetta ma scriveremo del codice per tradurla nel formato utile al nostro TDA Pila.

Questo codice è chiamato **implementazione** del TDA Pila. Più in generale un'implementazione è un insieme di metodi che soddisfano la sintassi e la semantica dell'interfaccia richiesta.

Ecco un'implementazione della Pila con le liste predefinite in Python:

```
class Pila:
  def __init__(self):
    self.Elementi = []

  def Push(self, Elemento) :
    self.Elementi.append(Elemento)

  def Pop(self):
    return self.Elementi.pop()

  def EVuota(self):
    return (self.Elementi == [])
```

L'oggetto `Pila` contiene un attributo chiamato `Elementi` che è la lista di oggetti contenuta nella pila. Il metodo `__init__` inizializza `Elementi` come lista vuota.

`Push` inserisce un nuovo elemento nella pila aggiungendolo a `Elementi`. `Pop` esegue l'operazione inversa, rimuovendo e ritornando l'ultimo elemento inserito nella pila.

Per controllare se la pila è vuota `EVuota` confronta `Elementi` con una lista vuota e ritorna *vero/falso*.

Un'implementazione di questo tipo in cui i metodi sono solo una semplice invocazione di metodi già esistenti viene detta **maschera**. Nella vita reale la maschera (o impiallacciatura) è tra le altre cose quello strato di legno di buona qualità che copre un legno di bassa qualità sottostante. In informatica è un pezzo di codice che nasconde i dettagli di un'implementazione per fornire un'interfaccia più semplice e standard.

18.4 Push e Pop

Una pila è una **struttura di dati generica** dato che possiamo aggiungere qualsiasi tipo di dato al suo interno. Gli esempi seguenti aggiungono due interi ed una stringa alla pila:

```
>>> P = Pila()
>>> P.Push(54)
>>> P.Push(45)
>>> P.Push("+")
```

Possiamo usare `EVuota` e `Pop` per rimuovere e stampare tutti gli elementi della pila:

```
while not P.EVuota() :
  print P.Pop(),
```

Il risultato è `+ 45 54`. In altre parole abbiamo usato la pila per stampare gli elementi in ordine inverso! Anche se questo non è il formato standard per la stampa di una lista usando una pila è stato comunque facile ottenerla.

Confronta questo codice con l'implementazione di `StampaInversa` nella sezione 17.4. Le due versioni sono molto più simili di ciò che sembra a prima vista, dato che entrambe fanno uso dello stesso meccanismo: mentre nell'implementazione della classe `Pila` appena scritta l'uso della pila è evidente, nella versione ricorsiva vista in precedenza il carico della gestione della pila era delegato all'interprete stesso. Ad ogni chiamata di funzione infatti viene usata una pila interna all'interprete che tiene conto della successione delle chiamate alle funzioni.

18.5 Uso della pila per valutare espressioni postfisse

Nella maggior parte dei linguaggi di programmazione le espressioni matematiche sono scritte con l'operatore tra i due operandi, come nella consueta `1+2`. Questo formato è chiamato **notazione infissa**. Un modo alternativo che ha avuto qualche successo in passato in particolari modelli di calcolatrici tascabili ma ora è usato meno frequentemente, è chiamato **notazione postfissa**: nella notazione postfissa l'operatore segue gli operandi, tanto che l'espressione appena vista sarebbe scritta in questo modo: `1 2 +`.

Il motivo per cui la notazione postfissa può rivelarsi utile è che c'è un modo del tutto naturale per valutare espressioni postfisse con l'uso della pila:

- A partire dall'inizio dell'espressione ricava un termine (operatore o operando) alla volta.
  - Se il termine è un operando aggiungilo all'inizio della pila.
  - Se il termine è un operatore estrai dalla pila il numero di operandi previsto per l'operatore, elabora il risultato dell'operazione su di essi e aggiungi il risultato all'inizio della pila.
- Quando tutta l'espressione è stata elaborata nella pila ci dovrebbe essere un solo elemento che rappresenta il risultato.

Esercizio: applica questo algoritmo all'espressione `1 2 + 3 *`.

Questo esempio mostra uno dei vantaggi della notazione postfissa: non sono necessarie parentesi per controllare l'ordine delle operazioni. Per ottenere lo stesso risultato con la notazione infissa avremmo dovuto scrivere `(1 + 2) * 3`.

Esercizio: scrivi l'espressione postfissa equivalente a `1+2*3`.

18.6 Parsing

Per implementare l'algoritmo di valutazione dell'espressione dobbiamo essere in grado di attraversare una stringa e di dividerla in una serie di operandi e operatori. Questo processo è un esempio di **parsing** e il risultato è una serie di elementi chiamati **token**. Abbiamo già visto questi termini all'inizio del libro.

Python fornisce un metodo `split` in due moduli, sia in `string` (per la gestione delle stringhe) che in `re` (per le espressioni regolari). La funzione `string.split` divide una stringa scomponendola in una lista di token e usando un singolo carattere come **delimitatore**. Per esempio:

```
>>> import string
>>> string.split("Nel mezzo del cammin"," ")
['Nel', 'mezzo', 'del', 'cammin']
```

In questo caso il delimitatore è il carattere spazio così che la stringa viene spezzata ad ogni spazio.

La funzione `re.split` è molto più potente, permettendo l'uso di una espressione regolare invece di un delimitatore singolo. Un'espressione regolare è un modo per specificare un insieme di stringhe e non soltanto un'unica stringa: `[A-Z]` è l'insieme di tutte le lettere maiuscole dell'alfabeto, mentre `[0-9]` è l'insieme di tutti i numeri. L'operatore ^ effettua la negazione dell'insieme così che [^0-9] rappresenta l'insieme di tutto ciò che non è un numero. Questi sono soltanto gli esempi più semplici di ciò che possono fare le espressioni regolari e per le nostre necessità ci fermeremo qui: infatti abbiamo già ricavato l'espressione regolare che ci serve per dividere un'espressione postfissa:

```
>>> import re
>>> re.split("([^0-9])", "123+456*/")
['123', '+', '456', '*', '', '/', '']
```

Nota come l'ordine degli operandi sia diverso da quello di `string.split` in quanto i delimitatori sono indicati prima della stringa da dividere.

La lista risultante include gli operandi `123` e `456`, e gli operatori `*` e `/`. Include inoltre due stringhe vuote inserite dopo gli operandi.

18.7 Valutazione postfissa

Per valutare un'espressione postfissa useremo il parser e l'algoritmo che abbiamo visto nelle sezioni precedenti. Per cominciare dalle cose più semplici inizialmente implementeremo solo gli operatori `+` e `*`:

```
def ValutaPostfissa(Espressione):
  import re
  ListaToken = re.split("([^0-9])", Espressione)
  Pila = Pila()
  for Token in ListaToken:
    if  Token == '' or Token == ' ':
      continue
    if  Token == '+':
      Somma = Pila.Pop() + Pila.Pop()
      Pila.Push(Somma)
    elif Token == '*':
      Prodotto = Pila.Pop() * Pila.Pop()
      Pila.Push(Prodotto)
    else:
      Pila.Push(int(Token))
  return Pila.Pop()
```

La prima condizione tiene a bada gli spazi e le stringhe vuote. Le due condizioni successive gestiscono gli operatori, partendo dal presupposto che qualsiasi altra cosa sia un operatore valido. Logicamente dovremo controllare la validità dell'espressione da valutare ed eventualmente mostrare un messaggio di errore se ci fossero dei problemi, ma questo lo faremo più avanti.

Testiamola per valutare l'espressione postfissa di `(56+47)*2`:

```
>>> print ValutaPostfissa("56 47 + 2 *")
206
```

18.8 Clienti e fornitori

Uno degli obiettivi fondamentali di un TDA è quello di separare gli interessi del *fornitore*, che scrive il codice del TDA, da quelli del *cliente*, che usa il TDA. Il fornitore deve solo preoccuparsi di verificare che l'implementazione sia corretta, secondo le specifiche del TDA, e non ha idea di *come* sarà usato il suo codice.

D'altra parte il cliente *parte dal presupposto* che l'implementazione del TDA sia corretta e non si preoccupa dei dettagli già considerati dal fornitore. Quando stai usando dei tipi predefiniti in Python hai il vantaggio di dover pensare solo da cliente, senza doverti preoccupare di verificare la corretta implementazione del codice.

Logicamente nel momento in cui implementi un TDA (e quindi sei il fornitore) devi scrivere del codice cliente per testarlo, e questo fatto può mettere un po' in confusione dato che si devono giocare entrambi i ruoli.

18.9 Glossario

Tipo di dato astratto (TDA): tipo di dato (solitamente una collezione di oggetti) definito da una serie di operazioni e che può essere implementato in una varietà di modi diversi.

Interfaccia: insieme di operazioni che definiscono un TDA.

Implementazione: codice che soddisfa i prerequisiti di sintassi e semantica di un'interfaccia.

Cliente: programma (o persona che scrive un programma) che usa un TDA.

Fornitore: programma (o persona che scrive un programma) che implementa un TDA.

Maschera: definizione di classe che implementa un TDA con definizioni di metodi che sono invocazioni di altri metodi, talvolta con l'apporto di semplici trasformazioni. Le maschere non fanno un lavoro significativo, ma migliorano o standardizzano l'interfaccia usata dal cliente.

Struttura di dati generica: struttura di dati che può contenere dati di ogni tipo.

Notazione infissa: modo di scrivere espressioni matematiche con gli operatori tra gli operandi, eventualmente con l'uso di parentesi.

Notazione postfissa: modo di scrivere espressioni matematiche con gli operatori posti dopo gli operandi (detta anche "notazione polacca inversa").

Parsing: lettura di una stringa di caratteri per l'analisi dei token e della struttura grammaticale.

Token: serie di caratteri che viene trattata come un'unità nell'operazione di parsing, allo stesso modo delle parole in un linguaggio naturale.

Delimitatore: carattere usato per separare i token, allo stesso modo della punteggiatura in un linguaggio naturale.

Capitolo 19

Code

Questo capitolo presenta due tipi di dati astratti (TDA): la Coda e la Coda con priorità. Nella vita reale un esempio di **coda** può essere la linea di clienti in attesa di un servizio di qualche tipo. Nella maggior parte dei casi il primo cliente della fila è quello che sarà servito per primo, anche se ci possono essere delle eccezioni. All'aeroporto ai clienti il cui volo sta per partire può essere concesso di passare davanti a tutti, indipendentemente dalla loro posizione nella fila. Al supermercato un cliente può scambiare per cortesia il suo posto con qualcuno che deve pagare solo pochi prodotti.

La regola che determina chi sarà il prossimo ad essere servito si chiama **politica di accodamento**. Quella più semplice è la **FIFO** ("first in, first out") dove il primo che arriva è il primo ad essere servito. La politica di accodamento più generale è l' **accodamento con priorità** dove a ciascun cliente è assegnata una priorità ed il cliente con la massima priorità viene servito per primo indipendentemente dall'ordine di arrivo. Diciamo che questa politica di accodamento è la più generale perché la priorità può essere basata su qualsiasi fattore: l'orario di partenza dell'aereo, la quantità di prodotti da pagare ad una cassa, l'importanza del cliente (!), la gravità dello stato di un paziente al pronto soccorso. Logicamente non tutte le politiche di accodamento sono "giuste"...

I tipi di dati astratti Coda e Coda con priorità condividono lo stesso insieme di operazioni. La differenza sta soltanto nella loro semantica: una Coda usa la politica FIFO, mentre la Coda con priorità, come suggerisce il nome stesso, usa la politica di accodamento con priorità.

19.1 Il TDA Coda

Il TDA Coda è definito dalle operazioni seguenti:

`__init__`**:** Inizializza una nuova coda vuota.

`Inserimento`**:** Aggiunge un nuovo elemento alla coda.

`Rimozione`: Rimuove e ritorna un elemento dalla coda. L'elemento ritornato è il primo inserito nella coda in ordine di tempo.

`EVuota`: Controlla se la coda è vuota.

19.2 Coda linkata

La prima implementazione del TDA Coda a cui guarderemo è chiamata **coda linkata** perché è composta di oggetti `Nodo` linkati. Ecco una definizione della classe:

```
class Coda:
  def __init__(self):
    self.Lunghezza = 0
    self.Testa = None

  def EVuota(self):
    return (self.Lunghezza == 0)

  def Inserimento(self, Contenuto):
    NodoAggiunto = Nodo(Contenuto)
    NodoAggiunto.ProssimoNodo = None
    if self.Testa == None:
      # se la lista e' vuota il nodo e' il primo
      self.Testa = Nodo
    else:
      # trova l'ultimo nodo della lista
      Ultimo = self.Testa
      while Ultimo.ProssimoNodo: Ultimo = Ultimo.ProssimoNodo
      # aggiunge il nuovo nodo
      Ultimo.ProssimoNodo = NodoAggiunto
    self.Lunghezza = self.Lunghezza + 1

  def Rimozione(self):
    Contenuto = self.Testa.Contenuto
    self.Testa = self.Testa.ProssimoNodo
    self.Lunghezza = self.Lunghezza - 1
    return Contenuto
```

I metodi `EVuota` e `Rimozione` sono identici a quelli usati in `ListaLinkata`. Il metodo `Inserimento` è nuovo ed un po' più complicato.

Vogliamo inserire nuovi elementi alla fine della lista: se la coda è vuota facciamo in modo che `Testa` si riferisca al nuovo nodo.

Altrimenti attraversiamo la lista fino a raggiungere l'ultimo nodo e attacchiamo a questo il nuovo nodo. Possiamo identificare facilmente l'ultimo nodo della lista perché è l'unico il cui attributo `ProssimoNodo` vale `None`.

Ci sono due invarianti per un oggetto `Coda` ben formato: il valore di `Lunghezza` dovrebbe essere il numero di nodi nella coda e l'ultimo nodo dovrebbe avere l'attributo `ProssimoNodo` uguale a `None`. Prova a studiare il metodo implementato verificando che entrambi gli invarianti siano sempre soddisfatti.

19.3 Performance

Normalmente quando invochiamo un metodo non ci interessa quali siano i dettagli della sua implementazione. Ma c'è uno di questi dettagli che invece dovrebbe interessarci: le performance del metodo. Quanto impiega ad essere eseguito? Come cambia il tempo di esecuzione man mano che la collezione aumenta di dimensioni?

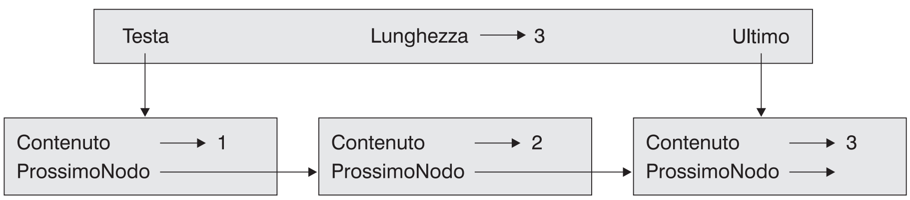
Diamo un'occhiata a `Rimozione`. Non ci sono cicli o chiamate a funzione, e ciò suggerisce che il tempo di esecuzione sarà lo stesso ogni volta. Questo tipo di metodo è definito **operazione a tempo costante**. In realtà il metodo potrebbe essere leggermente più veloce quando la lista è vuota dato che tutto il corpo della condizione viene saltato, ma la differenza in questo caso non è molto significativa e può essere tranquillamente trascurata.

La performance di `Inserimento` è molto diversa. Nel caso generale dobbiamo attraversare completamente la lista per trovarne l'ultimo elemento.

Questo attraversamento impiega un tempo che è proporzionale alla grandezza della lista: dato che il tempo di esecuzione in funzione lineare rispetto alla lunghezza, diciamo che questo metodo è un'**operazione a tempo lineare**. Se confrontato ad un'operazione a tempo costante il suo comportamento è decisamente peggiore.

19.4 Lista linkata migliorata

Logicamente un'implementazione del TDA Coda che può eseguire tutte le operazioni in un tempo costante è preferibile, dato che in questo caso il tempo di esecuzione è indipendente dalla grandezza della lista elaborata. Un modo per fare questo è quello di modificare la classe Coda per fare in modo che venga tenuta traccia tanto del primo che dell'ultimo elemento della lista, come mostrato in questa figura:



L'implementazione di `CodaMigliorata` potrebbe essere:

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

```
    self.Lunghezza = 0
    self.Testa = None
    self.UltimoNodo = None

  def EVuota(self):
    return (self.Lunghezza == 0)
```

Finora l'unico cambiamento riguarda l'aggiunta dell'attributo `UltimoNodo`. Questo attributo è usato dai metodi `Inserimento` e `Rimozione`:

```
class CodaMigliorata:
  ...
  def Inserimento(self, Contenuto):
    NodoAggiunto = Nodo(Contenuto)
    NodoAggiunto.ProssimoNodo = None
    if self.Lunghezza == 0:
      # se la lista e' vuota il nuovo nodo e'
      # sia la testa che la coda
      self.Testa = self.UltimoNodo = NodoAggiunto
    else:
      # trova l'ultimo nodo
      Ultimo = self.UltimoNodo
      # aggiunge il nuovo nodo
      Ultimo.ProssimoNodo = NodoAggiunto
      self.UltimoNodo = NodoAggiunto
    self.Lunghezza = self.Lunghezza + 1
```

Dato che `UltimoNodo` tiene traccia dell'ultimo nodo non dobbiamo più attraversare la lista per cercarlo. Come risultato abbiamo fatto diventare questo metodo un'operazione a tempo costante.

Comunque dobbiamo pagare un prezzo per questa modifica: quando dobbiamo rimuovere l'ultimo nodo con `Rimozione` dovremo assegnare `None` a `UltimoNodo`:

```
class CodaMigliorata:
  ...
  def Rimozione(self):
    Contenuto = self.Testa.Contenuto
    self.Testa = self.Testa.ProssimoNodo
    self.Lunghezza = self.Lunghezza - 1
    if self.Lunghezza == 0:
      self.UltimoNodo = None
    return Contenuto
```

Questa implementazione è più complessa di quella della coda linkata ed è più difficile dimostrare che è corretta, Il vantaggio che abbiamo comunque ottenuto è l'aver reso sia `Inserimento` che `Rimozione` operazioni a tempo costante.

> *Esercizio: scrivi un'implementazione del TDA Coda usando una lista di Python. Confronta le performance di questa implementazione con quelle di* `CodaMigliorata` *per una serie di lunghezze diverse della coda.*

19.5 Coda con priorità

Il TDA *Coda con priorità* ha la stessa interfaccia del TDA *Coda* ma una semantica diversa. L'interfaccia è sempre:

`__init__`: Inizializza una nuova coda vuota.

`Inserimento`: Aggiungi un elemento alla coda.

`Rimozione`: Rimuovi un elemento dalla coda. L'elemento da rimuovere e ritornare è quello con la priorità più alta.

`EVuota`: Controlla se la coda è vuota.

La differenza di semantica è che l'elemento da rimuovere non è necessariamente il primo inserito in coda, ma quello che ha la priorità più alta. Cosa siano le priorità e come siano implementate sono fatti non specificati dall'implementazione, dato che questo dipende dal genere di elementi che compongono la coda.

Per esempio se gli elementi nella coda sono delle stringhe potremmo estrarle in ordine alfabetico. Se sono punteggi del bowling dal più alto al più basso, e viceversa nel caso del golf. In ogni caso possiamo rimuovere l'elemento con la priorità più alta da una coda soltanto se i suoi elementi sono confrontabili tra di loro.

Questa è un'implementazione di una coda con priorità che usa una lista Python come attributo per contenere gli elementi della coda:

```
class CodaConPriorita:
  def __init__(self):
    self.Elementi = []

  def EVuota(self):
    return self.Elementi == []

  def Inserimento(self, Elemento):
    self.Elementi.append(Elemento)
```

I metodi `__init__`, `EVuota` e `Inserimento` sono tutte maschere delle operazioni su liste. L'unico metodo "interessante" è `Rimozione`:

```
class CodaConPriorita:
  ...
  def Rimozione(self):
    Indice = 0
    for i in range(1,len(self.Elementi)):
      if self.Elementi[i] > self.Elementi[Indice]:
        Indice = i
    Elemento = self.Elementi[Indice]
    self.Elementi[Indice:Indice+1] = []
    return Elemento
```

All'inizio di ogni iterazione `Indice` contiene l'indice dell'elemento con priorità massima. Ad ogni ciclo viene confrontato questo elemento con l'`i`-esimo elemento della lista: se il nuovo elemento ha priorità maggiore (nel nostro caso è maggiore), il valore di `Indice` diventa `i`.

Quando il ciclo `for` è stato completato `Indice` è l'indice dell'elemento con priorità massima. Questo elemento è rimosso dalla lista e ritornato.

Testiamo l'implementazione:

```
>>> q = CodaConPriorita()
>>> q.Inserimento(11)
>>> q.Inserimento(12)
>>> q.Inserimento(14)
>>> q.Inserimento(13)
>>> while not q.EVuota(): print q.Rimozione()
14
13
12
11
```

Se la coda contiene solo numeri o stringhe questi vengono rimossi in ordine numerico o alfabetico, dal più alto al più basso. Python può sempre trovare il numero o la stringa più grande perché può confrontare coppie di questi operandi con operatori di confronto predefiniti.

Se la coda contenesse un oggetto di tipo non predefinito è necessario fornire anche un metodo `__cmp__` per poter effettuare il confronto. Quando `Rimozione` usa l'operatore `>` per confrontare gli elementi in realtà invoca `__cmp__` per uno degli operandi e passa l'altro come parametro. La Coda con priorità funziona come ci si aspetta solo se il metodo `__cmp__` opera correttamente.

19.6 La classe `Golf`

Come esempio di oggetto con una definizione inusuale di priorità implementiamo una classe chiamata `Golf` che tiene traccia dei nomi e dei punteggi di un gruppo di golfisti. Partiamo con la definizione di `__init__` e `__str__`:

```
class Golf:
  def __init__(self, Nome, Punteggio):
    self.Nome = Nome
    self.Punteggio = Punteggio

  def __str__(self):
    return "%-16s: %d" % (self.Nome, self.Punteggio)
```

`__str__` usa l'operatore di formato per stampare i nomi ed i punteggi in forma tabellare su colonne ordinate.

Poi definiamo una versione di `__cmp__` dove il punteggio minore ottiene la priorità più alta: come abbiamo già visto in precedenza `__cmp__` ritorna 1 se `self` è più grande di `Altro`, -1 se `self` è minore di `Altro`, e 0 se i due valori sono uguali.

```
class Golf:
  ...
  def __cmp__(self, Altro):
    if self.Punteggio < Altro.Punteggio: return  1
    if self.Punteggio > Altro.Punteggio: return -1
    return 0
```

Ora siamo pronti a testare la coda con priorità sulla classe `Golf`:

```
>>> tiger = Golf("Tiger Woods",    61)
>>> phil  = Golf("Phil Mickelson", 72)
>>> hal   = Golf("Hal Sutton",     69)
>>>
>>> pq = CodaConPriorità()
>>> pq.Inserimento(tiger)
>>> pq.Inserimento(phil)
>>> pq.Inserimento(hal)
>>> while not pq.EVuota(): print pq.Rimozione()
Tiger Woods    : 61
Hal Sutton     : 69
Phil Mickelson : 72
```

> *Esercizio: scrivi un'implementazione di un TDA Coda con priorità facendo uso di una lista linkata. Dovrai tenere la lista sempre ordinata per fare in modo che la rimozione di un elemento sia un'operazione a tempo costante. Confronta le performance di questa implementazione con l'implementazione delle liste in Python.*

19.7 Glossario

Coda: insieme di oggetti in attesa di un servizio di qualche tipo; abbiamo implementato un TDA Coda che esegue le comuni operazioni su una coda.

Politica di accodamento: regole che determinano quale elemento di una coda debba essere rimosso per primo.

FIFO: "First In, First Out" (primo inserito, primo rimosso) politica di accodamento nella quale il primo elemento a essere rimosso è il primo ad essere stato inserito.

Coda con priorità: politica di accodamento nella quale ogni elemento ha una priorità determinata da fattori esterni. L'elemento con la priorità più alta è il primo ad essere rimosso. Abbiamo implementato un TDA Coda con priorità che definisce le comuni operazioni richieste da una coda con priorità.

Coda linkata: implementazione di una coda realizzata usando una lista linkata.

Operazione a tempo costante: elaborazione il cui tempo di esecuzione non dipende (o dipende in minima parte) dalla dimensione della struttura di dati da elaborare.

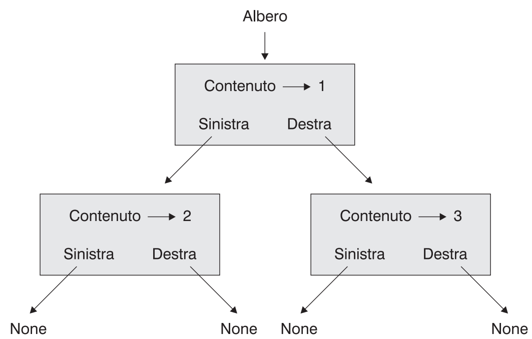
Operazione a tempo lineare: elaborazione il cui tempo di esecuzione è proporzionale alla dimensione della struttura di dati da elaborare.

Capitolo 20

Alberi

Come nel caso delle altre liste linkate, un albero è costituito di nodi. Un tipo comune di albero è l' **albero binario** nel quale ciascun nodo fa riferimento a due altri nodi che possono anche avere valore *None* (in questo caso è prassi comune non indicarli nei diagrammi di stato). Questi riferimenti vengono normalmente chiamati "rami" (o "sottoalberi") sinistro e destro.

Come nel caso dei nodi degli altri tipi di lista anche in questo caso un nodo possiede un contenuto. Ecco un diagramma di stato per un albero:



Il nodo principale dell'albero è chiamato **radice**, gli altri nodi **rami** e quelli terminali **foglie**. Si noti come l'albero viene generalmente disegnato capovolto, con la radice in alto e le foglie in basso.

Per rendere le cose più confuse vengono talvolta usate delle terminologie alternative che fanno riferimento ad un albero genealogico o alla geometria. Nel primo caso il nodo alla sommità è detto **genitore** e i nodi cui esso si riferisce **figli**; nodi con gli stessi genitori sono detti **fratelli**. Nel secondo caso parliamo di nodi a "sinistra" e "destra", in "alto" (verso il genitore/radice) e in "basso" (verso i figli/foglie).

Indipendentemente dai termini usati tutti i nodi che hanno la stessa distanza dalla radice appartengono allo stesso **livello**.

Come nel caso delle liste linkate gli alberi sono strutture di dati ricorsive:

Un albero è:

- un albero vuoto, rappresentato da None oppure
- un nodo che contiene un riferimento ad un oggetto e due riferimenti ad alberi.

20.1 La costruzione degli alberi

Il processo di costruzione degli alberi è simile a quello che abbiamo già visto nel caso delle liste linkate. Ogni invocazione del costruttore aggiunge un singolo nodo:

```
class Albero:
  def __init__(self, Contenuto, Sinistra=None, Destra=None):
    self.Contenuto = Contenuto
    self.Sinistra  = Sinistra
    self.Destra = Destra

  def __str__(self):
    return str(self.Contenuto)
```

Il Contenuto può essere di tipo qualsiasi ma sia Sinistra che Destra devono essere nodi di un albero. Sinistra e Destra sono opzionali ed il loro valore di default è None, significando con questo che non sono linkati ad altri nodi.

Come per gli altri nodi che abbiamo visto precedentemente, la stampa di un nodo dell'albero mostra soltanto il contenuto del nodo stesso.

Un modo per costruire un albero è quello di partire dal basso verso l'alto, allocando per primi i nodi figli:

```
FiglioSinistra = Albero(2)
FiglioDestra = Albero(3)
```

Poi creiamo il nodo genitore collegandolo ai figli:

```
Albero = Albero(1, FiglioSinistra, FiglioDestra);
```

Possiamo anche scrivere in modo più conciso questo codice invocando un costruttore annidato:

```
>>> Albero = Albero(1, Albero(2), Albero(3))
```

In ogni caso il risultato è l'albero presentato graficamente all'inizio del capitolo.

20.2 Attraversamento degli alberi

Ogni volta che vedi una nuova struttura la tua prima domanda dovrebbe essere "come posso attraversarla?". Il modo più intuitivo per attraversare un albero è quello di usare un algoritmo ricorsivo.

Per fare un esempio, se il nostro albero contiene interi questa funzione ne restituisce la somma:

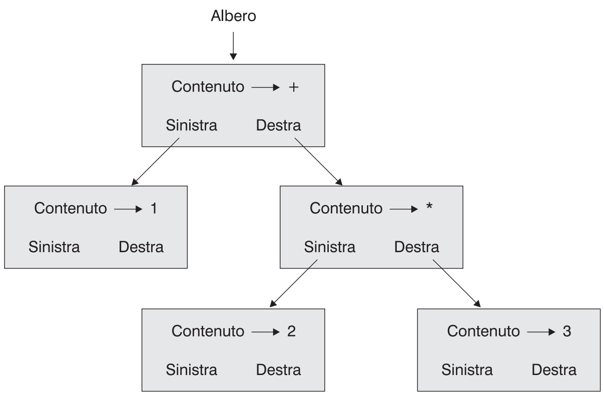
```
def Totale(Albero):
  if Albero == None: return 0
  return Albero.Contenuto + Totale(Albero.Sinistra) + \
         Totale(Albero.Destra)
```

Il caso base è l'albero vuoto che non ha contenuto e che quindi ha valore 0. Il passo successivo chiama due funzioni ricorsive per calcolare la somma dei rami figli. Quando la serie di chiamate ricorsiva è completa la funzione ritorna il totale.

20.3 Albero di espressioni

Un albero è un modo naturale per rappresentare una struttura di espressioni e a differenza di altre notazioni può rappresentare la loro elaborazione in modo non ambiguo (l'espressione infissa `1 + 2 * 3` è ambigua a meno che non si sappia che la moltiplicazione deve essere elaborata prima dell'addizione).

Ecco l'albero che rappresenta questa espressione:



I nodi dell'albero possono essere operandi come `1` e `2` o operatori come `+` e `*`. Gli operandi sono i nodi foglia, e i nodi operatore contengono i riferimenti ai rispettivi operandi. È importante notare che tutte queste operazioni sono **binarie** nel senso che hanno esattamente due operandi.

Possiamo costruire alberi come questo:

```
>>> Albero = Albero('+', Albero(1), Albero('*', Albero(2), \
             Albero(3)))
```

Guardando la figura non c'è assolutamente alcun problema nel determinare l'ordine delle operazioni: la moltiplicazione deve essere eseguita per prima per ottenere un risultato necessario all'addizione.

Gli alberi di espressioni hanno molti usi tra i quali possiamo citare la rappresentazione di espressioni matematiche postfisse e infisse (come abbiamo appena visto), e le operazioni di parsing, ottimizzazione e traduzione dei programmi nei compilatori.

20.4 Attraversamento di un albero

Potremmo attraversare un espressione ad albero e stampare il suo contenuto con:

```
def StampaAlberoPre(Albero):
  if Albero == None: return
  print Albero.Contenuto,
  StampaAlberoPre(Albero.Sinistra)
  StampaAlberoPre(Albero.Destra)
```

Per stampare questo albero abbiamo deciso di stamparne la radice, poi l'intero ramo di sinistra e poi quello di destra. Questo modo di attraversare l'albero è detto con **preordine** perché la radice appare sempre *prima* del contenuto dei figli. La stampa nel nostro caso è:

```
>>> Albero = Albero('+', Albero(1), Albero('*', Albero(2), \
             Albero(3)))
>>> StampaAlberoPre(Albero)
+ 1 * 2 3
```

Questo formato di stampa è diverso sia da quello che ci saremmo aspettati dalla notazione postfissa sia da quella infissa: si tratta infatti di una notazione chiamata **prefissa** nella quale gli operatori compaiono prima dei loro operandi.

Avrai già capito che cambiando l'ordine di attraversamento dell'albero sarà possibile ricavare le altre notazioni equivalenti. Se stampiamo prima i rami e poi il nodo radice otteniamo:

```
def StampaAlberoPost(Albero):
  if Albero == None: return
  StampaAlberoPost(Albero.Sinistra)
  StampaAlberoPost(Albero.Destra)
  print Albero.Contenuto,
```

Il risultato è `1 2 3 * +` in notazione postfissa. Questo tipo di attraversamento è chiamato **postordine**.

L'ultimo caso da considerare è l'attraversamento dell'albero con **inordine**, dove stampiamo il ramo sinistro, poi la radice ed infine il ramo destro:

```
def StampaAlberoIn(Albero):
  if Albero == None: return
  StampaAlberoIn(Albero.Sinistra)
  print Albero.Contenuto,
  StampaAlberoIn(Albero.Destra)
```

Il risultato è `1 + 2 * 3` in notazione infissa.

Ad essere onesti dovremmo menzionare una complicazione molto importante sulla quale abbiamo sorvolato. Talvolta è necessario l'uso delle parentesi per conservare l'ordine delle operazioni nelle espressioni infisse, così che un attraversamento con inordine non è sufficiente a generare un'espressione infissa corretta.

Ciononostante, e con poche modifiche, l'albero delle espressioni e tre diversi attraversamenti ricorsivi ci hanno permesso di tradurre diverse espressioni da una notazione all'altra.

> *Esercizio: modifica* `StampaAlberoIn` *così da mettere un paio di parentesi che racchiuda ogni coppia di operandi ed il loro operatore. Il risultato può essere considerato a questo punto corretto e non ambiguo? Sono sempre necessarie le parentesi?*

Se attraversiamo con inordine e teniamo traccia di quale livello dell'albero ci troviamo possiamo generare una rappresentazione grafica dell'albero:

```
def StampaAlberoIndentato(Albero, Livello=0):
  if Albero == None: return
  StampaAlberoIndentato(Albero.Destra, Livello+1)
  print '  '*Livello + str(Albero.Contenuto)
  StampaAlberoIndentato(Albero.Sinistra, Livello+1)
```

Il parametro `Livello` tiene traccia di dove ci troviamo nell'albero e per default vale inizialmente 0. Ogni volta che effettuiamo una chiamata ricorsiva passiamo `Livello+1` perché il livello del figlio è sempre più grande di 1 rispetto a quello del genitore. Ogni elemento è indentato di due spazi per ogni livello.

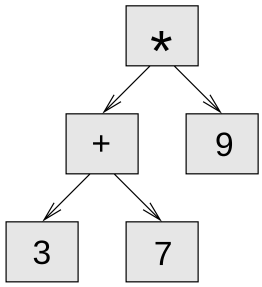
Il risultato del nostro albero di esempio è:

```
>>> StampaAlberoIndentato(Albero)
    3
  *
    2
+
  1
```

Guardando la figura dopo aver girato il foglio vedrai una versione semplificata della figura originale.

20.5 Costruire un albero di espressione

In questa sezione effettueremo il parsing di un'espressione infissa e costruiremo il corrispondente albero. L'espressione `(3+7)*9` produce questo diagramma, dove non sono stati indicati i nomi degli attributi:



Il parser che scriveremo dovrà riuscire a gestire espressioni contenenti numeri, parentesi e gli operatori + e *. Partiamo dal presupposto che la stringa da analizzare sia già stata spezzata in token che nel nostro caso sono elementi di una lista:

```
['(', 3, '+', 7, ')', '*', 9, 'end']
```

Il token `end` è stato aggiunto per fare il modo che il parser non continui la lettura al termine della lista.

> *Esercizio: scrivi una funzione che accetta un'espressione e la converte in una lista di token.*

La prima funzione che scriveremo è `ControllaToken` che prende come parametri una lista di token e un token atteso: dopo aver confrontato il token atteso con il primo elemento della lista, se i due coincidono l'elemento della lista viene rimosso e viene ritornato il valore *vero*; in caso contrario viene ritornato *falso*.

```
def ControllaToken(ListaToken, TokenAtteso):
  if ListaToken[0] == TokenAtteso:
    del ListaToken[0]
    return 1
  else:
    return 0
```

Dato che `ListaToken` si riferisce ad un oggetto mutabile i cambiamenti fatti sono visibili da qualsiasi altra variabile che si riferisce allo stesso oggetto.

La prossima funzione, `ControllaNumero`, gestisce gli operandi: se il prossimo elemento in `ListaToken` è un numero `ControllaNumero` lo rimuove dalla lista e ritorna un nodo foglia contenente il numero; in caso contrario viene ritornato `None`:

```
def ControllaNumero(ListaToken):
  x = ListaToken[0]
  if type(x) != type(0): return None
  del ListaToken[0]
  return Albero(x, None, None)
```

Prima di continuare è buona cosa testare isolatamente `ControllaNumero`. Assegniamo una lista di numeri a `ListaToken`, ne estraiamo il primo, stampiamo il risultato e ciò che rimane della lista di token:

```
>>> Lista = [9, 11, 'end']
>>> x = ControllaNumero(Lista)
>>> StampaAlberoPost(x)
9
>>> print Lista
[11, 'end']
```

Il prossimo metodo di cui avremo bisogno è `EsprProdotto` che costruisce un albero di espressione per le moltiplicazioni del tipo `3*7`.

Ecco una versione di `EsprProdotto` che gestisce prodotti semplici:

```
def EsprProdotto(ListaToken):
  a = ControllaNumero(ListaToken)
  if ControllaToken(ListaToken, '*'):
    b = ControllaNumero(ListaToken)
    return Albero('*', a, b)
  else:
    return a
```

Se `ControllaNumero` ha successo e ritorna un nodo assegniamo il primo operando ad `a`. Se il carattere successivo è `*` ricaviamo il secondo numero e costruiamo un albero con `a`, `b` e l'operatore moltiplicazione. Se il secondo carattere è qualcos'altro ritorniamo il nodo foglia con contenuto pari ad `a`.

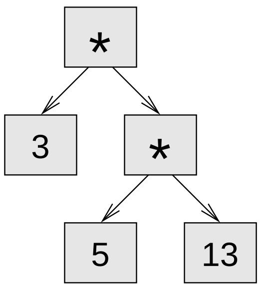
Ecco un paio di esempi:

```
>>> ListaToken = [9, '*', 11, 'end']
>>> Albero = EsprProdotto(ListaToken)
>>> StampaAlberoPost(Albero)
9 11 *
```

```
>>> ListaToken = [9, '+', 11, 'end']
>>> Albero = EsprProdotto(ListaToken)
>>> StampaAlberoPost(Albero)
9
```

Il secondo esempio mostra che noi consideriamo un singolo operando come una moltiplicazione valida. Questa definizione di "prodotto" non è proprio intuitiva, ma risulta esserci molto utile in questo caso.

Ora vediamo di gestire i prodotti composti, come in `3*5*13`. Tratteremo questa espressione come prodotto di prodotti, e cioè `3*(5*13)`. L'albero risultante è:



Con un piccolo cambiamento in `EsprProdotto` possiamo gestire prodotti arbitrariamente lunghi:

```
def EsprProdotto(ListaToken):
  a = ControllaNumero(ListaToken)
  if ControllaToken(ListaToken, '*'):
    b = EsprProdotto(ListaToken)       # questa linea e' cambiata
    return Albero('*', a, b)
  else:
    return a
```

In altre parole un prodotto può essere o un valore singolo o un albero con `*` alla radice, un numero a sinistra e un prodotto alla destra. Ormai dovresti cominciare a sentirti a tuo agio con questa definizione ricorsiva.

Testiamo la nuova versione con un prodotto composto:

```
>>> ListaToken = [2, '*', 3, '*', 5 , '*', 7, 'end']
>>> Albero = EsprProdotto(ListaToken)
>>> StampaAlberoPost(Albero)
2 3 5 7 * * *
```

Continuiamo con la nostra implementazione andando a gestire le somme. Ancora una volta useremo una definizione di somma che non è del tutto intuitiva: una somma può essere un albero con `+` alla radice, un prodotto a sinistra e una somma a destra. Inoltre consideriamo come "somma" anche un prodotto.

Se non riesci a comprenderne il significato, è possibile immaginare qualsiasi espressione priva di parentesi (ricorda che stiamo lavorando solo su addizioni e moltiplicazioni) come somme di prodotti. Questa proprietà rappresenta la base del nostro algoritmo di parsing.

`EsprSomma` prova a costruire un albero con un prodotto a sinistra e una somma a destra; nel caso non riesca a trovare un operatore `+` restituisce il prodotto.

```
def EsprSomma(ListaToken):
  a = EsprProdotto(ListaToken)
  if ControllaToken(ListaToken, '+'):
    b = EsprSomma(ListaToken)
    return Albero('+', a, b)
  else:
    return a
```

Proviamo con `9 * 11 + 5 * 7`:

```
>>> ListaToken = [9, '*', 11, '+', 5, '*', 7, 'end']
>>> Albero = EsprSomma(ListaToken)
>>> StampaAlberoPost(Albero)
9 11 * 5 7 * +
```

Ora ci mancano solo le parentesi. Dovunque in un'espressione compaia un numero, lì può essere racchiusa un'intera somma tra parentesi. Dobbiamo solo modificare `ControllaNumero` per gestire le **sub-espressioni**:

```
def ControllaNumero(ListaToken):
  if ControllaToken(ListaToken, '('):
    x = EsprSomma(ListaToken)         # ricava la sub-espressione
    ControllaToken(ListaToken, ')')   # rimuove la parentesi
                                      # chiusa
    return x
  else:
    x = ListaToken[0]
    if type(x) != type(0): return None
    ListaToken[0:1] = []
    return Albero(x, None, None)
```

Testiamo questa nuova funzione con `9 * (11 + 5) * 7`:

```
>>> ListaToken = [9, '*', '(', 11, '+', 5, ')', '*', 7, 'end']
>>> Albero = EsprSomma(ListaToken)
>>> StampaAlberoPost(Albero)
9 11 5 + 7 * *
```

Il parser ha gestito correttamente le parentesi e l'addizione viene eseguita prima della moltiplicazione.

Nella versione finale del programma è una buona idea dare a `ControllaNumero` un nuovo nome più coerente con il suo nuovo ruolo.

20.6 Gestione degli errori

Le espressioni che dobbiamo passare al parser devono essere ben formate. Se abbiamo raggiunto la fine di una sub-espressione ci aspettiamo una parentesi chiusa: nel caso questa non sia presente sarebbe il caso di gestire questa condizione d'errore.

```
def ControllaNumero(ListaToken):
  if ControllaToken(ListaToken, '('):
    x = EsprSomma(ListaToken)
    if not ControllaToken(ListaToken, ')'):
      raise 'BadExpressionError', 'manca la parentesi'
    return x
  else:
    # omettiamo il resto della funzione
```

L'istruzione `raise` crea un'eccezione: in questo caso abbiamo creato un nuovo tipo di errore chiamato `BadExpressionError`. Se la funzione che chiama `ControllaNumero` o una delle altre funzioni indicate in traccia al momento dell'errore gestisce le eccezioni allora il programma può continuare; in caso contrario Python mostra il messaggio di errore e si interrompe.

> *Esercizio: trova altri posti in queste funzioni in cui possono verificarsi errori e aggiungi le istruzioni* `raise` *appropriate. Testa successivamente il tuo codice passando alle funzioni delle espressioni errate.*

20.7 L'albero degli animali

In questa sezione svilupperemo un piccolo programma che usa un albero per rappresentare un sistema di conoscenze e aumentando la sua ampiezza grazie all'interazione con l'operatore.

Il programma interagisce con l'operatore per creare un albero di domande e di nomi di animali. Ecco un esempio del suo funzionamento:

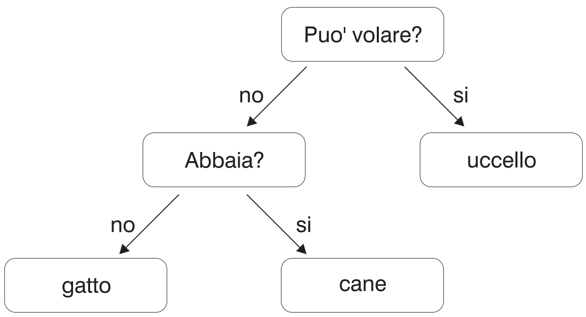
```
Stai pensando ad un animale? s
E' un uccello? n
Qual e' il nome dell'animale? cane
Che domanda permette di distinguere tra un cane e un \
                                uccello? Puo' volare
Se l'animale fosse un cane quale sarebbe la risposta? n

Stai pensando ad un animale? s
Puo' volare? n
E' un cane? n
Qual e' il nome dell'animale? gatto
Che domanda permette di distinguere tra un gatto e un\
                                         cane? Abbaia
Se l'animale fosse un gatto quale sarebbe la risposta? n

Stai pensando ad un animale? s
Puo' volare? n
Abbaia? s
E' un cane? s
Ho indovinato!

Stai pensando ad un animale? n
```

Ecco un albero costruito da questo dialogo:



All'inizio di ogni round il programma parte alla radice dell'albero e pone la prima domanda. A seconda della risposta si muove a destra o a sinistra lungo l'albero e continua fino a raggiungere una foglia. A questo punto tira a indovinare: se la sua ipotesi non è corretta chiede il nome dell'animale pensato dall'operatore e una domanda per poterlo distinguere dall'animale trovato nel nodo foglia. Poi aggiunge il nuovo animale come nodo all'albero, assieme alla nuova domanda.

Ecco il codice:

```
def Animale():
  # parte con una lista composta di un solo elemento
  Radice = Albero("uccello")

  # continua finche' l'operatore non abbandona
  while 1:
```

```
    print
    if not RispostaAffermativa("Stai pensando ad un \
                               animale? "): break

    # percorre l'albero
    SottoAlbero = Radice
    while SottoAlbero.RamoSinistro() != None:
      Messaggio = SottoAlbero.OttieniContenuto() + "? "
      if RispostaAffermativa(Messaggio):
        SottoAlbero = SottoAlbero.RamoDestro()
      else:
        SottoAlbero = SottoAlbero.RamoSinistro()

    # prova a indovinare
    Ipotesi = SottoAlbero.OttieniContenuto()
    Messaggio = "E' un " + Ipotesi + "? "
    if RispostaAffermativa(Messaggio):
      print "Ho indovinato!"
      continue

    # ottiene nuove informazioni
    Messaggio = "Qual e' il nome dell'animale? "
    Animale  = raw_input(Messaggio)
    Messaggio  = "Che domanda permette di distinguere tra \
                  un %s e un %s? "
    Domanda = raw_input(Messaggio % (Animale, Ipotesi))

    # aggiunge le nuove informazioni all'albero
    SottoAlbero.SettaContenuto(Domanda)
    Messaggio = "Se l'animale fosse un %s quale sarebbe la \
                 risposta? "
    if RispostaAffermativa(Messaggio % Animale):
      SottoAlbero.SettaRamoSinistro(Albero(Ipotesi))
  SottoAlbero.SettaRamoDestro(Albero(Animale))
    else:
  SottoAlbero.SettaRamoSinistro(Albero(Animale))
  SottoAlbero.SettaRamoDestro(Albero(Ipotesi))
```

La funzione `RispostaAffermativa` è solo un'aiutante e serve a stampare un messaggio attendendo la risposta dall'operatore. Se la risposta inizia con *s* o *S* la funzione ritorna vero:

```
def RispostaAffermativa(Domanda):
  from string import lower
  Risposta = lower(raw_input(Domanda))
  return (Risposta[0] == 's')
```

La condizione del ciclo esterno è `1` e questo significa che il ciclo verrà eseguito finchè non si incontra un'istruzione `break`, nel caso l'operatore non stia pensando ad un animale.

Il ciclo `while` interno serve a percorrere l'albero dall'alto in basso, guidato dalle risposte dell'operatore.

Se dopo aver raggiunto un nodo foglia ci troviamo a dover inserire un nuovo animale (con la rispettiva domanda per distinguerlo da quello rappresentato dal nodo foglia), viene effettuata una serie di operazioni:

- viene sostituito il contenuto del nodo foglia con la domanda appena inserita
- il nodo originale (che era il nodo foglia) col rispettivo contenuto viene aggiunto come figlio del nodo foglia
- il nodo rappresentato dal nuovo animale viene aggiunto come figlio allo stesso ex-nodo foglia.

Un "piccolo" problema con questo programma è che non appena termina la sua esecuzione tutto quello che gli abbiamo insegnato viene dimenticato...

> *Esercizio: pensa ai vari modi in cui potresti salvare l'albero su file e poi implementa quello che ritieni sia il più semplice.*

20.8 Glossario

Albero binario: albero in cui ogni nodo si riferisce a zero, uno o due nodi dipendenti.

Nodo radice: nodo senza genitori in un albero.

Nodo foglia: nodo senza figli in un albero.

Nodo genitore: nodo che si riferisce ad un dato nodo.

Nodo figlio: uno dei nodi cui si riferisce un altro nodo.

Nodi fratelli: nodi che hanno uno stesso genitore.

Livello: insieme dei nodi equidistanti dalla radice.

Operatore binario: operatore che prende due operandi.

Sub-espressione: espressione tra parentesi che agisce come singolo operando in una espressione più grande.

Preordine: modo di attraversamento di un albero in cui si visita ogni nodo prima dei suoi figli.

Notazione prefissa: notazione matematica dove ogni operatore compare prima dei suoi operandi.

Postordine: modo di attraversamento di un albero in cui si visitano i figli di un nodo prima del nodo stesso.

Inordine: modo di attraversamento di un albero in cui si visita prima il figlio a sinistra, poi la radice ed infine il figlio a destra.

Appendice A

Debug

In un programma possono manifestarsi diversi tipi di errore ed è sempre utile saperli distinguere per poterli rimuovere velocemente:

- Gli errori di sintassi sono prodotti da Python durante la fase di traduzione del codice sorgente prima dell'esecuzione. Di solito indicano che c'è qualcosa di sbagliato nella sintassi del programma. Esempio: omettere i due punti alla fine dell'istruzione `def` porta al messaggio `SyntaxError: invalid syntax`.

- Gli errori in esecuzione (runtime) sono errori che si verificano mentre il programma sta lavorando. La maggior parte delle volte i messaggi di errore indicano quale sia la causa dell'errore e quali le funzioni in esecuzione nel momento in cui si è verificato. Esempio: una ricorsione infinita causa un errore in esecuzione quando si raggiunge il massimo livello di ricorsione ammesso.

- Gli errori di semantica sono i più difficili da rintracciare dato che il programma viene eseguito ma non porta a termine le operazioni corrette. Esempio: un'espressione può non essere valutata nell'ordine corretto tanto da portare ad un risultato inaspettato.

Il primo passo per rimuovere un errore è capire con che tipo di errore hai a che fare. Sebbene le sezioni seguenti siano organizzate in base al tipo di errore alcune tecniche sono applicabili a più di una situazione.

A.1 Errori di sintassi

Gli errori di sintassi sono facili da eliminare quando hai capito dov'è il problema. Sfortunatamente i messaggi di errore non sono sempre utili: i messaggi più comuni sono `SyntaxError: invalid syntax` (sintassi non valida) e `SyntaxError: invalid token` (token non valido) che di per sé non sono di grande aiuto.

Il messaggio fornisce indicazioni sulla riga di programma dove il problema si verifica, anche se questa riga in realtà è il punto in cui Python si è accorto dell'errore e non necessariamente dove questo si trova. Molto spesso l'errore è nella riga precedente rispetto a quella indicata dal messaggio.

Se stai scrivendo in modo incrementale il tuo programma è facile indicare immediatamente dove risiede il problema dato che deve trovarsi nelle ultime righe che hai aggiunto.

Se stai copiando il codice da un libro controlla accuratamente se l'originale è uguale a ciò che hai scritto. Controlla ogni carattere senza dimenticare che il codice riportato dal libro potrebbe anche essere sbagliato. Se vedi qualcosa che sembra un errore di sintassi, probabilmente lo è.

Ecco qualche sistema per evitare gli errori di sintassi più comuni:

1. Controlla di non usare una parola riservata di Python come nome di variabile.

2. Controlla di aver messo i due punti alla fine dell'intestazione di ogni istruzione composta, includendo le istruzioni `for`, `while`, `if` e `def`.

3. Controlla che l'indentazione sia consistente: puoi indentare indifferentemente con spazi o tabulazioni ma è buona norma non usarli contemporaneamente. Ogni livello dovrebbe essere indentato della stessa quantità di spazi: l'uso di due spazi per livello è abbastanza consolidato all'interno della comunità Python.

4. Controlla che i delimitatori delle stringhe siano appaiati correttamente.

5. Se hai stringhe su righe multiple (delimitate da virgolette o apici tripli) controlla di averle terminate in modo appropriato. Una stringa non terminata può causare un errore `invalid token` alla fine del programma o può trattare il resto del programma come fosse parte della stringa. In casi particolari potrebbe non essere neanche mostrato un messaggio d'errore.

6. Una parentesi non chiusa (`(`, `{` o `[`) costringe Python a cercare nelle righe successive credendole parte dell'istruzione corrente. Generalmente un errore di questo tipo viene individuato alla riga seguente.

7. Controlla che non sia presente un `=` invece del `==` all'interno di una condizione.

Se malgrado i controlli non hai ottenuto risultati passa alla sezione seguente.

A.1.1 Non riesco a far funzionare il programma indipendentemente da ciò che faccio

Se il compilatore si ostina a dire che c'è un errore e tu non lo vedi probabilmente state guardando due diversi pezzi di codice. Controlla se il programma su cui stai lavorando è lo stesso che Python cerca di eseguire. Se non sei sicuro introduci un errore di sintassi proprio all'inizio ed eseguilo di nuovo: se il compilatore non vede il nuovo errore di sintassi con ogni probabilità state guardando due cose diverse.

Se questo accade, un approccio standard è quello di ricominciare da zero con un nuovo programma tipo “Hello, World!” e controllare che questo venga eseguito. Poi gradualmente aggiungi pezzi di codice fino ad arrivare a quello definitivo.

A.2 Errori in esecuzione

Quando il tuo programma è sintatticamente corretto Python lo può importare ed eseguire. Cosa potrebbe andare storto?

A.2.1 Il mio programma non fa assolutamente niente

Questo problema è comune quando il tuo codice consiste di classi e funzioni ma non c'è del codice che inizia l'esecuzione con una chiamata. Questo può essere un comportamento voluto quando il tuo codice deve essere importato in un altro modulo per fornire classi e funzioni.

Se questo non è intenzionale controlla di invocare una funzione per iniziare l'esecuzione o eseguila direttamente dal prompt interattivo. Vedi anche la sezione “Flusso di esecuzione” in seguito.

A.2.2 Il mio programma si blocca

Se il programma si ferma e sembra di essere bloccato senza fare nulla diciamo che è “in blocco”. Spesso questo significa che il flusso del programma è all'interno di un ciclo o di una ricorsione infiniti.

- Se i tuoi sospetti cadono su un ciclo in particolare aggiungi una istruzione `print` immediatamente prima di entrare nel ciclo (“entrata nel ciclo”) ed una subito dopo l'uscita (“uscita dal ciclo”).

  Esegui il programma: se leggi il primo messaggio ma non il secondo sei in un loop infinito. Vai alla sezione “Ciclo infinito” descritta in seguito.

- La maggior parte delle volte una ricorsione infinita permetterà al programma di lavorare per un po' e poi produrrà un messaggio d'errore “RuntimeError: Maximum recursion depth exceeded”. Vedi a riguardo la sezione “Ricorsione infinita”.

  Se non ottieni questo tipo di errore ma sospetti che ci sia qualche problema con un metodo o una funzione ricorsivi puoi sempre usare le tecniche descritte nella sezione “Ricorsione infinita”.

- Se questi passi non hanno dato risultati inizia a controllare altri cicli e funzioni ricorsive.

- Se ancora non ottieni risultati è possibile che ti stia sfuggendo come si evolve il flusso del programma. Vedi la sezione “Flusso di esecuzione” in seguito.

Ciclo infinito

Quando hai a che fare con cicli sospetti puoi sempre aggiungere alla fine del corpo del ciclo un'istruzione `print` per stampare i valori delle variabili usate nella condizione ed il valore della condizione.

Per esempio:

```
while x > 0 and y < 0 :
  # fai qualcosa con x
  # fai qualcosa con y

  print  "x: ", x
  print  "y: ", y
  print  "condizione: ", (x > 0 and y < 0)
```

Quando esegui il programma sono stampate tre righe ogni volta che viene reiterato il ciclo. La condizione d'uscita sarà falsa solo nell'ultima esecuzione. Se il ciclo continua ad essere eseguito potrai controllare i valori di `x` e `y` e forse potrai capire perché non vengono aggiornati correttamente.

Ricorsione infinita

La maggior parte delle volte una ricorsione infinita porterà all'esaurimento della memoria disponibile: il programma funzionerà finché ci sarà memoria disponibile, poi verrà mostrato un messaggio d'errore `Maximum recursion depth exceeded`.

Se sospetti che una funzione o un metodo stiano causando una ricorsione infinita, inizia a controllare il caso base: deve sempre essere presente una condizione all'interno della funzione che possa ritornare senza effettuare un'ulteriore chiamata alla funzione stessa. Se il caso base non è presente è il caso di ripensare l'algoritmo.

Se è presente il caso base ma sembra che il flusso di programma non lo raggiunga mai aggiungi un'istruzione `print` all'inizio della funzione o metodo per stamparne i parametri. Se durante l'esecuzione i parametri non si muovono verso il caso base probabilmente significa che la ricorsione non funziona.

Flusso di esecuzione

Se non sei sicuro che il flusso di esecuzione si stia muovendo lungo il programma aggiungi un'istruzione `print` all'inizio di ogni funzione per stampare un messaggio del tipo "entro nella funzione `xxx`" dove `xxx` è il nome della funzione. Quando il programma è in esecuzione avrai una traccia del suo flusso.

A.2.3 Quando eseguo il programma ottengo un'eccezione

Se qualcosa va storto durante l'esecuzione Python stampa un messaggio che include il nome dell'eccezione, la linea di programma dove si è verificato il problema e una traccia.

La traccia identifica la funzione che si stava eseguendo al momento dell'errore, la funzione che l'aveva chiamata, la funzione che aveva chiamato quest'ultima e così a ritroso fino ad arrivare al livello superiore. Mostra cioè il cammino che ha portato all'interno della funzione malfunzionante. Come ulteriori informazioni sono anche indicate le linee di programma dove ciascuna funzione viene chiamata.

Il primo passo è quello di esaminare il posto nel programma dove l'errore si è verificato e cercare di capire cos'è successo. Questi sono i più comuni errori in esecuzione:

NameError: stai provando ad usare una variabile che non esiste in questo ambiente. Ricorda che le variabili sono locali e non puoi riferirti ad esse al di fuori della funzione dove sono state definite.

TypeError: ci sono parecchie possibili cause:

- Stai cercando di usare un valore in maniera impropria, per esempio riferendoti agli elementi di una lista usando un indice che non è intero.
- C'è una discrepanza tra gli elementi di una stringa di formato e i valori passati per la conversione. Questo può succedere sia se il numero degli elementi è diverso sia se il tipo richiede una conversione non valida.
- Stai passando un numero di argomenti errato ad una funzione o a un metodo. Per i metodi controlla la definizione del metodo e che il primo parametro sia `self`. Poi guarda all'invocazione del metodo; controlla se stai invocando il metodo su un oggetto del tipo giusto e se stai fornendo in modo corretto gli altri argomenti.

KeyError: stai cercando di accedere ad un elemento di un dizionario usando una chiave non conosciuta dal dizionario.

AttributeError: stai provando ad accedere ad un attributo o metodo che non esiste.

IndexError: stai usando un indice troppo grande per accedere ad una lista, ad una stringa o ad una tupla. Prima della posizione dell'errore aggiungi un'istruzione `print` per mostrare il valore dell'indice e la lunghezza dell'array. L'array è della lunghezza corretta? L'indice ha il valore corretto?

A.2.4 Ho aggiunto così tante istruzioni `print` da essere sommerso dalle stampe

Uno dei problemi con l'uso dell'istruzione `print` durante il debug è che puoi rimanere letteralmente sommerso da una valanga di messaggi. Ci sono due modi per procedere: semplificare le stampe o semplificare il programma.

Per semplificare le stampe puoi rimuovere o commentare le istruzioni `print` che non servono più, combinarle o formattare la stampa per ottenere una forma più semplice da leggere.

Per semplificare il programma ci sono parecchie cose che puoi fare. Prima di tutto riduci il programma per farlo lavorare su un piccolo insieme di dati. Se stai ordinando un array usa un array *piccolo*. Se il programma accetta un inserimento di dati dall'operatore cerca il più piccolo pacchetto di dati che genera il problema.

In secondo luogo ripulisci il programma. Rimuovi il codice morto e riorganizza il programma per renderlo il più leggibile possibile. Se sospetti che il problema risieda in una parte profondamente annidata del codice prova a riscrivere quella parte in modo più semplice. Se sospetti di una funzione complessa prova a dividerla in funzioni più piccole da testare separatamente.

Spesso il solo processo di trovare un insieme di dati che causa il problema ti porta a scoprirne la causa. Se trovi che il programma funziona in una situazione ma non in un'altra questo ti dà un notevole indizio di cosa stia succedendo.

Riscrivere un pezzo di codice può aiutarti a trovare piccoli bug difficili da individuare soprattutto se fai un cambiamento nel codice che credi non vada ad influire nel resto del programma e invece lo fa.

A.3 Errori di semantica

Gli errori di semantica sono i più difficili da scovare perché il compilatore e l'interprete non forniscono informazioni riguardo che cosa ci sia di sbagliato nel tuo programma. L'unica cosa che sai è ciò che il programma dovrebbe fare e che questo non è ciò che il programma effettivamente fa.

Il primo passo è quello di comprendere la connessione tra il codice ed il comportamento che stai osservando, con la conseguente creazione di ipotesi per giustificare ciò che vedi. Una delle cose che rendono il tutto così difficile è il fatto che il computer sia così veloce.

Spesso ti capiterà di desiderare di poter rallentare il programma fino ad una velocità più "umana" ed effettivamente questo è ciò che fanno alcuni programmi appositamente studiati per il debug. Il tempo trascorso a inserire qualche istruzione `print` ben piazzata è comunque breve se confrontato con tutta la procedura che occorre mettere in atto per configurare opportunamente il debugger, per inserire ed eliminare i punti di interruzione nel programma e per verificare passo per passo tutta l'esecuzione del programma.

A.3.1 Il mio programma non funziona

Dovresti farti qualche domanda:

- C'è qualcosa che il programma dovrebbe fare e sembra non venga fatto? Trova la sezione del codice incaricato di eseguire quella particolare funzione e verifica che questo venga eseguito quando ci si aspetta.

- Succede qualcosa che non dovrebbe accadere? Trova il pezzo di codice che esegue quella particolare funzione e controlla se esso viene eseguito quando non dovrebbe.

- C'è una sezione del codice che non fa quello che ci si aspetta? Controlla questo codice verificando di aver ben capito cosa fa, soprattutto se fa uso di altri moduli Python. Leggi la documentazione per le funzioni che invochi. Prova a testarle una ad una creando piccoli esempi e controllando i risultati.

Per poter programmare devi avere un modello mentale di come il programma lavora: se ottieni un programma che non si comporta come desideri spesso la colpa non è nel programma in sé ma nel tuo modello mentale.

Il modo migliore per correggere il tuo modello mentale è quello di spezzare i suoi componenti (di solito le funzioni ed i metodi) e testare ogni componente in modo indipendente. Quando hai trovato la discrepanza tra il tuo modello e la realtà puoi risolvere il problema.

Dovresti costruire e testare i componenti man mano che sviluppi il programma così ti troveresti, in caso di problemi, soltanto con piccole parti di codice da controllare.

A.3.2 Ho un'espressione piuttosto lunga che non fa ciò che dovrebbe

Scrivere espressioni complesse va bene finché queste sono leggibili ma ricorda che possono rendere problematico il debug. È sempre una buona norma spezzare un'espressione complessa in una serie di assegnazioni anche usando variabili temporanee.

Per esempio:

```
self.Mano[i].AggiungeCarta \
         (self.Mano[self.TrovaVicino(i)].ProssimaCarta())
```

Può essere riscritta come:

```
Vicino = self.TrovaVicino(i)
Prossima = self.Mano[Vicino].ProssimaCarta()
self.Mano[i].AggiungeCarta (Prossima)
```

La versione esplicita è più semplice da leggere poiché i nomi delle variabili forniscono una documentazione aggiuntiva ed è anche più semplice da controllare in fase di debug dato che puoi stampare i valori delle variabili intermedie.

Un altro problema collegato alle espressioni complesse è che talvolta l'ordine di valutazione può non essere ciò che ci si aspetta. Per tradurre l'espressione $\frac{x}{2\pi}$ in Python devi scrivere:

```
y = x / 2 * math.pi;
```

Questo non è corretto perché moltiplicazione e divisione hanno la stessa precedenza e sono valutate da sinistra a destra. Così questa espressione viene valutata $x\pi/2$.

Un buon sistema per effettuare il debug delle espressioni è aggiungere le parentesi per rendere esplicita la valutazione:

```
y = x/(2*math.pi);
```

Quando non sei sicuro dell'ordine di valutazione usa le parentesi. Non solo il programma sarà corretto ma sarà anche più leggibile da parte di chi non ha memorizzato le regole di precedenza.

A.3.3 Ho una funzione o un metodo che non restituisce ciò che mi aspetto

Se hai un'istruzione `return` con un'espressione complessa non hai modo di stampare il valore restituito da una funzione. Anche stavolta è il caso di usare una variabile temporanea. Invece di:

```
return self.Mano[i].TrisRimossi()
```

puoi scrivere:

```
Conteggio = self.Mano[i].TrisRimossi()
return Conteggio
```

Ora hai modo di stampare il valore di `Conteggio` prima di ritornare dalla funzione.

A.3.4 Sono bloccato e ho bisogno di aiuto!

Prova innanzitutto a staccarti dal computer per qualche minuto. I computer emettono onde elettromagnetiche che influenzano il cervello causando un bel po' di effetti spiacevoli:

- Frustrazione e/o rabbia.
- Superstizione ("il computer mi odia") e pensieri poco ortodossi ("il programma funzionava quando indossavo il mio cappello al contrario").
- Programmazione "casuale" (il tentativo poco probabile di scrivere il programma corretto scrivendo un gran numero di programmi assolutamente casuali)

Se credi di soffrire di qualcuno di questi sintomi alzati e vatti a fare quattro passi. Quando ti sei calmato torna a pensare al programma. Cosa sta facendo? Quali sono le possibili cause? Quand'è stata l'ultima volta che si è comportato a dovere? Cos'è stato fatto in seguito?

Talvolta è necessario parecchio tempo per trovare un bug ed è molto più efficace una ricerca fatta dopo aver lasciato sgombra la mente per qualche tempo. Alcuni tra i posti migliori per trovare i bug (senza bisogno di un computer!) sono i treni, le docce ed il letto, appena prima di addormentarsi.

Se il problema si verifica in ufficio di venerdì pomeriggio fai finta di lavorarci, ma pensa ad altro: non fare modifiche che potresti rimpiangere il lunedì mattina...

A.3.5 Niente scherzi: ho veramente bisogno di aiuto

Capita. Anche i migliori programmatori a volte rimangono bloccati: qualche volta lavori su un programma così a lungo che non riesci più a vedere l'errore. due occhi "freschi" sono ciò che ci vuole.

Prima di tirare dentro qualcun altro nella caccia al bug devi essere sicuro di aver esaurito ogni possibile tecnica qui descritta. Il tuo programma dovrebbe essere il più semplice possibile e dovresti lavorare sul più piccolo insieme di dati che causa il problema. Dovresti avere una serie di istruzioni `print` nei posti appropriati con una stampa comprensibile dei rispettivi valori di controllo. Dovresti aver capito il problema tanto da poterlo esprimere in modo conciso.

Quando chiedi l'aiuto di qualcuno ricorda di dare tutte le informazioni di cui può aver bisogno:

- Se c'è un messaggio d'errore che cosa e che parte del programma indica?
- Cos'è stata l'ultima cosa che hai fatto prima che si verificasse l'errore? Quali sono le ultime righe di codice che hai scritto o quali sono i nuovi dati che fanno fallire il test?
- Cosa hai già provato e che ipotesi hai già escluso con le tue prove?

Quando hai trovato il bug fermati un secondo e cerca di capire come avresti potuto trovarlo più in fretta. La prossima volta che riscontrerai un comportamento simile, questo sarà molto utile.

Appendice B

Creazione di un nuovo tipo di dato

La programmazione orientata agli oggetti permette al programmatore di creare nuovi tipi di dato che si comportano come quelli predefiniti. Esploreremo questa capacità costruendo una classe `Frazione` che possa lavorare come i tipi di dato numerico predefiniti (intero, intero lungo e virgola mobile).

Le frazioni, conosciute anche come numeri razionali, sono numeri che si possono esprimere come rapporto tra numeri interi, come nel caso di 5/6. Il numero superiore si chiama numeratore, quello inferiore denominatore.

Iniziamo con una definizione della classe `Frazione` con un metodo di inizializzazione che fornisce un numeratore ed un denominatore interi.

```
class Frazione:
  def __init__(self, Numeratore, Denominatore=1):
    self.Numeratore = Numeratore
    self.Denominatore = Denominatore
```

Il denominatore è opzionale: se la frazione è creata (istanziata) con un solo parametro rappresenta un intero così che se il numeratore è n costruiremo la frazione $n/1$.

Il prossimo passo è quello di scrivere il metodo `__str__` per stampare le frazioni in un modo che sia comprensibile. La forma "numeratore/denominatore" è probabilmente quella più "naturale":

```
class Frazione:
  ...
  def __str__(self):
    return "%d/%d" % (self.Numeratore, self.Denominatore)
```

Per testare ciò che abbiamo fatto finora scriviamo tutto in un file chiamato `frazione.py` (o qualcosa di simile; l'importante è che per te abbia un nome che ti permetta di rintracciarlo in seguito) e lo importiamo nell'interprete Python. Poi passiamo a creare un oggetto `Frazione` e a stamparlo:

```
>>> from Frazione import Frazione
>>> f = Frazione(5,6)
>>> print "La frazione e'", f
La frazione e' 5/6
```

Come abbiamo già visto il comando `print` invoca il metodo `__str__` implicitamente.

B.1 Moltiplicazione di frazioni

Ci interessa poter applicare le consuete operazioni matematiche a operandi di tipo Frazione. Per farlo procediamo con la ridefinizione degli operatori matematici quali l'addizione, la sottrazione, la moltiplicazione e la divisione.

Iniziamo dalla moltiplicazione perché è la più semplice da implementare. Il risultato della moltiplicazione di due frazioni è una frazione che ha come numeratore il prodotto dei due numeratori, e come denominatore il prodotto dei denominatori. `__mul__` è il nome usato da Python per indicare l'operatore `*`:

```
class Frazione:
  ...
  def __mul__(self, Altro):
    return Frazione(self.Numeratore * Altro.Numeratore,
                    self.Denominatore * Altro.Denominatore)
```

Possiamo testare subito questo metodo calcolando il prodotto di due frazioni:

```
>>> print Frazione(5,6) * Frazione(3,4)
15/24
```

Funziona, ma possiamo fare di meglio. Possiamo infatti estendere il metodo per gestire la moltiplicazione di una frazione per un intero, usando la funzione `type` per controllare se `Altro` è un intero. In questo caso prima di procedere con la moltiplicazione lo si convertirà in frazione:

```
class Frazione:
  ...
  def __mul__(self, Altro):
    if type(Altro) == type(5):
      Altro = Frazione(Altro)
    return Frazione(self.Numeratore * Altro.Numeratore,
                    self.Denominatore * Altro.Denominatore)
```

La moltiplicazione tra frazioni e interi ora funziona, ma solo se la frazione compare alla sinistra dell'operatore:

```
>>> print Frazione(5,6) * 4
20/6
>>> print 4 * Frazione(5,6)
TypeError: unsupported operand type(s) for *: 'int' and 'instance
```

Per valutare un operatore binario come la moltiplicazione Python controlla l'operando di sinistra per vedere se questo fornisce un metodo `__mul__` che supporta il tipo del secondo operando. Nel nostro caso l'operatore moltiplicazione predefinito per gli interi non supporta le frazioni (com'è giusto, dato che abbiamo appena inventato noi la classe `Frazione`).

Se il controllo non ha successo Python passa a controllare l'operando di destra per vedere se è stato definito un metodo `__rmul__` che supporta il tipo di dato dell'operatore di sinistra. Visto che non abbiamo ancora scritto `__rmul__` il controllo fallisce e viene mostrato il messaggio di errore.

Esiste comunque un metodo molto semplice per scrivere `__rmul__`:

```
class Frazione:
  ...
  __rmul__ = __mul__
```

Con questa assegnazione diciamo che il metodo `__rmul__` è lo stesso di `__mul__`, così che per valutare `4 * Fraction(5,6)` Python invoca `__rmul__` sull'oggetto `Frazione` e passa 4 come parametro:

```
>>> print 4 * Frazione(5,6)
20/6
```

Dato che `__rmul__` è lo stesso di `__mul__` e che quest'ultimo accetta parametri interi è tutto a posto.

B.2 Addizione tra frazioni

L'addizione è più complessa della moltiplicazione ma non troppo: la somma di a/b e c/d è infatti la frazione `(a*d+c*b)/b*d`.

Usando il codice della moltiplicazione come modello possiamo scrivere `__add__` e `__radd__`:

```
class Frazione:
  ...
  def __add__(self, Altro):
    if type(Altro) == type(5):
      Altro = Frazione(Altro)
    return Fraction(self.Numeratore   * Altro.Denominatore +
                    self.Denominatore * Altro.Numeratore,
                    self.Denominatore * Altro.Denominatore)

  __radd__ = __add__
```

Possiamo testare questi metodi con frazioni e interi:

```
>>> print Frazione(5,6) + Frazione(5,6)
60/36
>>> print Frazione(5,6) + 3
23/6
```

```
>>> print 2 + Frazione(5,6)
17/6
```

I primi due esempi invocano `__add__`; l'ultimo `__radd__`.

B.3 Algoritmo di Euclide

Nell'esempio precedente abbiamo calcolato la somma 5/6+5/6 e ottenuto 60/36. Il risultato è corretto ma quella ottenuta non è la sua migliore rappresentazione. Per **ridurre** la frazione ai suoi termini più semplici dobbiamo dividere il numeratore ed il numeratore per il loro **massimo comune divisore (MCD)** che è 12. Il risultato diventa quindi 5/3.

In generale quando creiamo e gestiamo un oggetto `Frazione` dovremmo sempre dividere numeratore e denominatore per il loro MCD. Nel caso di una frazione già ridotta il MCD è 1.

Euclide di Alessandria (circa 325–265 A.C.) inventò un algoritmo per calcolare il massimo comune divisore tra due numeri interi m e n:

> Se n divide perfettamente m allora il MCD è n. In caso contrario il MCD è il MCD tra n ed il resto della divisione di m diviso per n.

Questa definizione ricorsiva può essere espressa in modo conciso con una funzione:

```
def MCD(m, n):
  if m % n == 0:
    return n
  else:
    return MCD(n, m%n)
```

Nella prima riga del corpo usiamo l'operatore modulo per controllare la divisibilità. Nell'ultima riga lo usiamo per calcolare il resto della divisione.

Dato che tutte le operazioni che abbiamo scritto finora creano un nuovo oggetto `Frazione` come risultato potremmo inserire la riduzione nel metodo di inizializzazione:

```
class Frazione:
  def __init__(self, Numeratore, Denominatore=1):
    mcd = MCD(numeratore, Denominatore)
    self.Numeratore   = Numeratore / mcd
    self.Denominatore = Denominatore / mcd
```

Quando creiamo una nuova `Frazione` questa sarà immediatamente ridotta alla sua forma più semplice:

```
>>> Frazione(100,-36)
-25/9
```

Una bella caratteristica di `MCD` è che se la frazione è negativa il segno meno è sempre spostato automaticamente al numeratore.

B.4 Confronto di frazioni

Supponiamo di dover confrontare due oggetti di tipo `Frazione`, `a` e `b` valutando `a == b`. L'implementazione standard di `==` ritorna vero solo se `a` e `b` sono lo stesso oggetto, effettuando un confronto debole.

Nel nostro caso vogliamo probabilmente ritornare *vero* se a e b hanno lo stesso valore e cioè fare un confronto forte. Ne abbiamo già parlato nella sezione 12.4.

Dobbiamo quindi insegnare alle frazioni come confrontarsi tra di loro. Come abbiamo visto nella sezione 15.4, possiamo ridefinire tutti gli operatori di confronto in una volta sola fornendo un nuovo metodo `__cmp__`.

Per convenzione il metodo `__cmp__` ritorna un numero negativo se `self` è minore di `Altro`, zero se sono uguali e un numero positivo se `self` è più grande di `Altro`.

Il modo più semplice per confrontare due frazioni è la moltiplicazione incrociata: se $a/b > c/d$ allora $ad > bc$. Con questo in mente ecco quindi il codice per `__cmp__`:

```
class Frazione:
  ...
  def __cmp__(self, Altro):
    Differenza = (self.Numeratore  * Altro.Denominatore -
            Altro.Numeratore * self.Denominatore)
    return Differenza
```

Se `self` è più grande di `Altro` allora `Differenza` è positiva. Se `Altro` è maggiore allora `Differenza` è negativa. Se sono uguali `Differenza` è zero.

B.5 Proseguiamo

Logicamente non abbiamo ancora finito. Dobbiamo ancora implementare la sottrazione ridefinendo `__sub__` e la divisione con il corrispondente metodo `__div__`.

Un modo per gestire queste operazioni è quello di implementare la negazione ridefinendo `__neg__` e l'inversione con `__invert__`: possiamo infatti sottrarre sommando al primo operando la negazione del secondo, e dividere moltiplicando il primo operando per l'inverso del secondo. Poi dobbiamo fornire `__rsub__` e `__rdiv__`.

Purtroppo non possiamo usare la scorciatoia già vista nel caso di addizione e moltiplicazione dato che sottrazione e divisione non sono commutative. Non possiamo semplicemente assegnare `__rsub__` e `__rdiv__` a lle corrispondenti `__sub__` e `__div__`, dato che in queste operazioni l'ordine degli operandi fa la differenza...

Per gestire la **negazione unaria**, che non è altro che l'uso del segno meno con un singolo operando (da qui il termine "unaria" usato nella definizione), sarà necessario ridefinire il metodo `__neg__`.

Potremmo anche calcolare le potenze ridefinendo `__pow__` ma l'implementazione in questo caso è un po' complessa: se l'esponente non è un intero, infatti, può non essere possibile rappresentare il risultato come `Frazione`. Per fare un esempio, `Frazione(2) ** Frazione(1,2)` non è nient'altro che la radice di 2 che non è un numero razionale e quindi non può essere rappresentato come frazione. Questo è il motivo per cui non è così facile scrivere una versione generale di `__pow__`.

C'è un'altra estensione della classe `Frazione` che potrebbe rivelarsi utile: finora siamo partiti dal presupposto che numeratore e denominatore sono interi, ma nulla ci vieta di usare interi lunghi.

> *Esercizio: completa l'implementazione della classe* `Frazione` *per gestire sottrazione, divisione ed elevamento a potenza, con interi lunghi al numeratore e denominatore.*

B.6 Glossario

Massimo comune divisore(MCD): il più grande numero positivo intero che divide senza resto sia il numeratore che il denominatore di una frazione.

Riduzione: trasformazione di una frazione nella sua forma più semplice, grazie alla divisione di numeratore e denominatore per il loro MCD.

Negazione unaria: operazione che calcola un inverso additivo, solitamente indicato da un segno meno anteposto ad un numero. È chiamata "unaria" perché agisce su un unico operando, a differenza di altri operatori, quali la sottrazione "binaria", che agiscono su due operandi.

Appendice C

Listati dei programmi

C.1 class `Punto`

```
class Punto:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'

  def __add__(self, AltroPunto):
    return Punto(self.x + AltroPunto.x, self.y + AltroPunto.y)

  def __sub__(self, AltroPunto):
    return Punto(self.x - AltroPunto.x, self.y - AltroPunto.y)

  def __mul__(self, AltroPunto):
    return self.x * AltroPunto.x + self.y * AltroPunto.y

  def __rmul__(self, AltroPunto):
    return Punto(AltroPunto * self.x,  AltroPunto * self.y)

  def reverse(self):
    self.x , self.y = self.y, self.x

  def DirittoERovescio(Stringa):
    import copy
    Rovescio = copy.copy(Stringa)
    Rovescio.reverse()
    print str(Stringa) + str(Rovescio)
```

C.2 class Tempo

```
# -------------------
# Versione funzionale
# -------------------
#   t = Tempo(3,14)
#   s = Tempo(8,12,15)
#   print SommaTempi(s, t)

def ConverteInSecondi(Orario):
  Minuti = Orario.Ore * 60 + Orario.Minuti
  Secondi = Minuti * 60 + Orario.Secondi
  return Secondi

def ConverteInTempo(Secondi):
  Orario = Tempo()
  Orario.Ore = Secondi / 3600
  Secondi = Secondi - Orario.Ore * 3600
  Orario.Minuti = Secondi / 60
  Secondi = Secondi - Orario.Minuti * 60
  Orario.Secondi = Secondi
  return Orario

def SommaTempi(Tempo1, Tempo2):
  Secondi = ConverteInSecondi(Tempo1) + ConverteInSecondi(Tempo2)
  return ConverteInTempo(Secondi)

# ------------------
# Versione a oggetti
# ------------------
# con modifica di uno degli oggetti:
#   t = Tempo(3,14)
#   s = Tempo(8,12,15)
#   t.AggiungiTempo(s)
#   print t
# in alternativa, senza modificare t:
#   a = Tempo()
#   a.SommaTempi(t, s)
#   print a

class Tempo:
  def __init__(self, Ore=0, Minuti=0, Secondi=0):
    self.Ore = Ore
    self.Minuti = Minuti
    self.Secondi = Secondi

  def __str__(self):
    return str(self.Ore) + ":" + str(self.Minuti) + ":" + \
```

```
                                str(self.Secondi)

  def Incremento(self, Secondi):
    Secondi = Secondi + self.Secondi + self.Minuti*60 + \
              self.Ore*3600
    self.Ore = Secondi / 3600
    Secondi = Secondi % 3600
    self.Minuti = Secondi / 60
    Secondi = Secondi % 60
    self.Secondi = Secondi

  def ConverteInSecondi(self):
    Minuti = self.Ore * 60 + self.Minuti
    Secondi = Minuti * 60 + self.Secondi
    return Secondi

  def ConverteInTempo(self, Secondi):
    self.Ore = Secondi / 3600
    Secondi = Secondi - self.Ore * 3600
    self.Minuti = Secondi / 60
    Secondi = Secondi - self.Minuti * 60
    self.Secondi = Secondi

  def AggiungiTempo(self, Tempo2):
    Secondi = self.ConverteInSecondi() + \
              Tempo2.ConverteInSecondi()
    self.ConverteInTempo(Secondi)  # l'oggetto self e' stato
                                   # modificato!

  def SommaTempi(self, Tempo1, Tempo2):
    Secondi = Tempo1.ConverteInSecondi() + \
              Tempo2.ConverteInSecondi()
    self.ConverteInTempo(Secondi)
```

C.3 Carte, mazzi e giochi

```
import random

class Carta:

  ListaSemi = ["Fiori", "Quadri", "Cuori", "Picche"]
  ListaRanghi = ["impossibile", "Asso", "2", "3", "4", "5", "6",
                 "7", "8", "9", "10", "Jack", "Regina", "Re"]

  def __init__(self, Seme=0, Rango=0):
    self.Seme = Seme
    self.Rango = Rango
```

```
  def __str__(self):
    return (self.ListaRanghi[self.Rango] + " di " + \
            self.ListaSemi[self.Seme])

  def __cmp__(self, Altro):

    # controlla il seme
    if self.Seme > Altro.Seme: return 1
    if self.Seme < Altro.Seme: return -1

    # se i semi sono uguali controlla il rango
    if self.Rango > Altro.Rango: return 1
    if self.Rango < Altro.Rango: return -1

    # se anche i ranghi sono uguali le carte sono uguali!
    return 0

class Mazzo:

  def __init__(self):
    self.Carte = []
    for Seme in range(4):
      for Rango in range(1, 14):
        self.Carte.append(Carta(Seme, Rango))

  def StampaMazzo(self):
    for Carta in self.Carte:
      print Carta

  def __str__(self):
    s = ""
    for i in range(len(self.Carte)):
      s = s + " "*i + str(self.Carte[i]) + "\n"
    return s

  def Mischia(self):
    import random
    NumCarte = len(self.Carte)
    for i in range(NumCarte):
      j = random.randrange(i, NumCarte)
      self.Carte[i], self.Carte[j] = self.Carte[j], self.Carte[i]

  def RimuoviCarta(self, Carta):
    if Carta in self.Carte:
      self.Carte.remove(Carta)
      return 1
    else:
      return 0
```

```
  def PrimaCarta(self):
    return self.Carte.pop()

  def EVuoto(self):
    return (len(self.Carte) == 0)

  def Distribuisci(self, ListaMani, NumCarte=999):
    NumMani = len(ListaMani)
    for i in range(NumCarte):
      if self.EVuoto(): break         # si ferma se non ci sono
                                      # piu` carte
      Carta = self.PrimaCarta()       # prende la carta
                                      # superiore del mazzo
      Mano = ListaMani[i % NumMani]   # di chi e` il prossimo
                                      # turno?
      Mano.AggiungiCarta(Carta)       # aggiungi la carta
                                      # alla mano


class Mano(Mazzo):

  def __init__(self, Nome=""):
    self.Carte = []
    self.Nome = Nome

  def AggiungiCarta(self,Carta) :
    self.Carte.append(Carta)

  def __str__(self):
    s = "La mano di " + self.Nome
    if self.EVuoto():
      s = s + " e' vuota\n"
    else:
      s = s + " contiene queste carte:\n"
    return s + Mazzo.__str__(self)


class GiocoCarte:

  def __init__(self):
    self.Mazzo = Mazzo()
    self.Mazzo.Mischia()


class ManoOldMaid(Mano):

  def RimuoviCoppie(self):
    Conteggio = 0
```

```
    CarteOriginali = self.Carte[:]
    for CartaOrig in CarteOriginali:
      CartaDaCercare = Carta(3-CartaOrig.Seme, CartaOrig.Rango)
      if CartaDaCercare in self.Carte:
        self.Carte.remove(CartaOrig)
        self.Carte.remove(CartaDaCercare)
        print "Mano di %s: %s elimina %s" % \
               (self.Nome,CartaOrig,CartaDaCercare)
        Conteggio = Conteggio + 1
    return Conteggio

class GiocoOldMaid(GiocoCarte):

  def Partita(self, Nomi):

    # rimozione della regina di fiori
    self.Mazzo.RimuoviCarta(Carta(0,12))

    # creazione di una mano per ogni giocatore
    self.Mani = []
    for Nome in Nomi:
      self.Mani.append(ManoOldMaid(Nome))

    # distribuzione delle carte
    self.Mazzo.Distribuisci(self.Mani)
    print "---------- Le carte sono state distribuite"
    self.StampaMani()

    # toglie le coppie iniziali
    NumCoppie = self.RimuoveTutteLeCoppie()
    print "---------- Coppie scartate, inizia la partita"
    self.StampaMani()

    # gioca finche' sono state fatte 50 coppie
    Turno = 0
    NumMani = len(self.Mani)
    while NumCoppie < 25:
      NumCoppie = NumCoppie + self.GiocaUnTurno(Turno)
      Turno = (Turno + 1) % NumMani

    print "---------- La partita e' finita"
    self.StampaMani()

  def RimuoveTutteLeCoppie(self):
    Conteggio = 0
    for Mano in self.Mani:
      Conteggio = Conteggio + Mano.RimuoveCoppie()
    return Conteggio
```

```
  def GiocaUnTurno(self, Giocatore):
    if self.Mani[Giocatore].EVuoto():
      return 0
    Vicino = self.TrovaVicino(Giocatore)
    CartaScelta = self.Mani[Vicino].SceltaCarta()
    self.Mani[Giocatore].AggiungeCarta(CartaScelta)
    print "Mano di", self.Mani[Giocatore].Nome, ": scelta", \
          CartaScelta
    Conteggio = self.Mani[Giocatore].RimuoveCoppie()
    self.Mani[Giocatore].Mischia()
    return Conteggio

  def TrovaVicino(self, Giocatore):
    NumMani = len(self.Mani)
    for Prossimo in range(1,NumMani):
      Vicino = (Giocatore + Prossimo) % NumMani
      if not self.Mani[Vicino].EVuoto():
        return Vicino
```

C.4 Liste linkate

```
def StampaLista(Nodo):
    while Nodo:
      print Nodo,
      Nodo = Nodo.ProssimoNodo
    print

def StampaInversa(Lista):
    if Lista == None: return
    Testa = Lista
    Coda = Lista.ProssimoNodo
    StampaInversa(Coda)
    print Testa,

def StampaInversaFormato(Lista) :
    print "[",
    if Lista != None :
      Testa = Lista
      Coda = Lista.ProssimoNodo
      StampaInversa(Coda)
      print Testa,
    print "]",

def RimuoviSecondo(Lista):
    if Lista == None: return
    Primo = Lista
    Secondo = Lista.ProssimoNodo
```

```
    # il primo nodo deve riferirsi al terzo
    Primo.ProssimoNodo = Secondo.ProssimoNodo
    # separa il secondo nodo dal resto della lista
    Secondo.ProssimoNodo = None
    return Secondo

class Nodo:

  def __init__(self, Contenuto=None, ProssimoNodo=None):
    self.Contenuto = Contenuto
    self.ProssimoNodo  = ProssimoNodo

  def __str__(self):
    return str(self.Contenuto)

  def StampaInversa(self):
    if self.ProssimoNodo != None:
      Coda = self.ProssimoNodo
      Coda.StampaInversa()
    print self.Contenuto,

class ListaLinkata:
  def __init__(self) :
    self.Lunghezza = 0
    self.Testa = None

  def StampaInversa(self):
    print "[",
    if self.Testa != None:
      self.Testa.StampaInversa()
    print "]",

  def AggiuntaPrimo(self, Contenuto):
    NodoAggiunto = Nodo(Contenuto)
    NodoAggiunto.ProssimoNodo = self.Testa
    self.Testa = NodoAggiunto
    self.Lunghezza = self.Lunghezza + 1
```

C.5 class Pila

```
class Pila:
  def __init__(self):
    self.Elementi = []

  def Push(self, Elemento) :
    self.Elementi.append(Elemento)
```

```
  def Pop(self):
    return self.Elementi.pop()

  def EVuota(self):
    return (self.Elementi == [])

def ValutaPostfissa(Espressione):
  import re
  ListaToken = re.split("([^0-9])", Espressione)
  Pila1 = Pila()
  for Token in ListaToken:
    if  Token == '' or Token == ' ':
      continue
    if  Token == '+':
      Somma = Pila1.Pop() + Pila1.Pop()
      Pila1.Push(Somma)
    elif Token == '*':
      Prodotto = Pila1.Pop() * Pila1.Pop()
      Pila1.Push(Prodotto)
    else:
      Pila1.Push(int(Token))
  return Pila1.Pop()
```

C.6 Alberi

```
class Albero:
  def __init__(self, Contenuto, Sinistra=None, Destra=None):
    self.Contenuto = Contenuto
    self.Sinistra  = Sinistra
    self.Destra = Destra

  def __str__(self):
    return str(self.Contenuto)

  def OttieniContenuto(self): return self.Contenuto
  def RamoDestro(self):       return self.Destra
  def RamoSinistro(self):     return self.Sinistra

  def SettaContenuto(self, Contenuto): self.Contenuto = Contenuto
  def SettaRamoDestro(self, Nodo):     self.Destra = Nodo
  def SettaRamoSinistro(self, Nodo):   self.Sinistra = Nodo

def Totale(Albero):
  if Albero == None: return 0
  return Albero.Contenuto + Totale(Albero.Sinistra) + \
                            Totale(Albero.Destra)
```

```
def StampaAlberoPre(Albero):
  if Albero == None: return
  print Albero.Contenuto,
  StampaAlberoPre(Albero.Sinistra)
  StampaAlberoPre(Albero.Destra)

def StampaAlberoPost(Albero):
  if Albero == None: return
  StampaAlberoPost(Albero.Sinistra)
  StampaAlberoPost(Albero.Destra)
  print Albero.Contenuto,

def StampaAlberoIn(Albero):
  if Albero == None: return
  StampaAlberoIn(Albero.Sinistra)
  print Albero.Contenuto,
  StampaAlberoIn(Albero.Destra)

def StampaAlberoIndentato(Albero, Livello=0):
  if Albero == None: return
  StampaAlberoIndentato(Albero.Destra, Livello+1)
  print '  '*Livello + str(Albero.Contenuto)
  StampaAlberoIndentato(Albero.Sinistra, Livello+1)

def ControllaToken(ListaToken, TokenAtteso):
  if ListaToken[0] == TokenAtteso:
    del ListaToken[0]
    return 1
  else:
    return 0

def ControllaNumero(ListaToken):
  if ControllaToken(ListaToken, '('):
    x = EsprSomma(ListaToken)               # ricava la
                                            # sub-espressione
    if not ControllaToken(ListaToken, ')'): # rimuove la
                                            # parentesi chiusa
      raise 'BadExpressionError', 'manca la parentesi'
    return x
  else:
    x = ListaToken[0]
    if type(x) != type(0): return None
    ListaToken[0:1] = []
    return Albero(x, None, None)

def EsprProdotto(ListaToken):
  a = ControllaNumero(ListaToken)
  if ControllaToken(ListaToken, '*'):
    b = EsprProdotto(ListaToken)
```

```
    return Albero('*', a, b)
  else:
    return a

def EsprSomma(ListaToken):
  a = EsprProdotto(ListaToken)
  if ControllaToken(ListaToken, '+'):
    b = EsprSomma(ListaToken)
    return Albero('+', a, b)
  else:
    return a
```

C.7 Indovina l'animale

```
def RispostaAffermativa(Domanda):
  from string import lower
  Risposta = lower(raw_input(Domanda))
  return (Risposta[0] == 's')


def Animale():
  # parte con una lista composta di un solo elemento
  Radice = Albero("uccello")

  # continua finche' l'operatore non abbandona
  while 1:
    print
    if not RispostaAffermativa("Stai pensando ad un \
           animale? "): break

    # percorre l'albero
    SottoAlbero = Radice
    while SottoAlbero.RamoSinistro() != None:
      Messaggio = SottoAlbero.OttieniContenuto() + "? "
      if RispostaAffermativa(Messaggio):
        SottoAlbero = SottoAlbero.RamoDestro()
      else:
        SottoAlbero = SottoAlbero.RamoSinistro()

    # prova a indovinare
    Ipotesi = SottoAlbero.OttieniContenuto()
    Messaggio = "E' un " + Ipotesi + "? "
    if RispostaAffermativa(Messaggio):
      print "Ho indovinato!"
      continue
```

```
    # ottiene nuove informazioni
    Messaggio = "Qual e' il nome dell'animale? "
    Animale  = raw_input(Messaggio)
    Messaggio  = "Che domanda permette di distinguere tra un %s \
                  e un %s? "
    Domanda = raw_input(Messaggio % (Animale, Ipotesi))

    # aggiunge le nuove informazioni all'albero
    SottoAlbero.SettaContenuto(Domanda)
    Messaggio = "Se l'animale fosse un %s quale sarebbe la \
                                        risposta? "
    if RispostaAffermativa(Messaggio % Animale):
      SottoAlbero.SettaRamoSinistro(Albero(Ipotesi))
      SottoAlbero.SettaRamoDestro(Albero(Animale))
    else:
      SottoAlbero.SettaRamoSinistro(Albero(Animale))
      SottoAlbero.SettaRamoDestro(Albero(Ipotesi))
```

C.8 class Frazione

```
def MCD(m, n):
  if m % n == 0:
    return n
  else:
    return MCD(n, m%n)


class Frazione:

  def __init__(self, Numeratore, Denominatore=1):
    mcd = MCD(Numeratore, Denominatore)
    self.Numeratore   = Numeratore / mcd
    self.Denominatore = Denominatore / mcd

  def __str__(self):
    return "%d/%d" % (self.Numeratore, self.Denominatore)

  def __mul__(self, Altro):
    if type(Altro) == type(1):
      Altro = Frazione(Altro)
    return Frazione(self.Numeratore * Altro.Numeratore, \
                    self.Denominatore * Altro.Denominatore)
  __rmul__ = __mul__

  def __add__(self, Altro):
    if type(Altro) == type(5):
      Altro = Frazione(Altro)
```

```
    return Frazione(self.Numeratore   * Altro.Denominatore +
                    self.Denominatore * Altro.Numeratore,
                    self.Denominatore * Altro.Denominatore)

  __radd__ = __add__

  def __cmp__(self, Altro):
    Differenza = (self.Numeratore  * Altro.Denominatore - \
            Altro.Numeratore * self.Denominatore)
    return Differenza

  def __repr__(self):
    return self.__str__()
```

Appendice D

Altro materiale

Arrivati a questo punto qual è la direzione da prendere? Le possibilità sono molte e vanno dall'ampliamento della conoscenza dell'informatica in generale, all'applicazione di Python in campi specifici.

Gli esempi proposti in questo libro sono stati deliberatamente semplici ma non hanno mostrato appieno quelle che sono le capacità più entusiasmanti del linguaggio. Ecco un campionario di estensioni di Python e di suggerimenti per progetti che le usano.

- La programmazione dell'interfaccia grafica (detta anche "GUI", graphical user interface) permette al tuo programma di interagire con l'operatore sotto forma di ambiente grafico.

  Il primo pacchetto grafico nato per Python è stato Tkinter, basato sui linguaggi di scripting Tcl e Tk di Jon Ousterhout. Tkinter è sempre presente nelle distribuzioni di Python.

  Un'altra piattaforma piuttosto conosciuta è wxPython. Questa è essenzialmente una maschera per facilitare l'uso di wxWindows, un pacchetto scritto in C++ che implementa un sistema a finestre usando un'interfaccia nativa in ambiente Windows e Unix (Linux incluso). Le finestre ed i controlli in wxPython tendono ad essere più semplici da programmare rispetto ai corrispondenti Tkinter.

  Qualsiasi tipo di programmazione con interfaccia grafica ti porterà ad un ambiente di programmazione controllato dall'evento, dove non è tanto il programmatore ma l'operatore a decidere il flusso di esecuzione. Questo stile di programmazione necessita di un po' di pratica per poter essere gestito nel modo migliore e talvolta può comportare una completa riscrittura del programma.

- La programmazione Web integra Python con Internet. Possiamo, per esempio, costruire programmi web client che aprono e leggono una pagina remota in modo abbastanza semplice, tanto che le difficoltà sono confrontabili con quelle (minime) che si possono incontrare durante l'apertura di un file su disco locale.

Ci sono moduli Python che permettono l'accesso a file remoti via ftp e moduli che consentono di ricevere e spedire email. Python è ampiamente usato anche per la gestione di form di introduzione dati nei web server.

- I database sono paragonabili a dei super-file dove i dati sono memorizzati secondo schemi predefiniti e sono accessibili in vari modi. Python è dotato di un certo numero di moduli per accedere a dati di diversi tipi di database, sia Open Source che commerciali.

- La programmazione a thread permette di eseguire diversi flussi di programma allo stesso tempo a partire da un unico programma. Se hai presente come funziona un browser per Internet puoi farti un'idea di cosa questo significhi: in un browser vengono caricate più pagine contemporaneamente e mentre ne guardi una il caricamento delle altre prosegue in modo quasi del tutto trasparente.

- Quando ci troviamo alle prese con necessità particolari ed è indispensabile una maggiore velocità di esecuzione Python può essere integrato da moduli scritti in altri linguaggi, tipo il C ed il C++. Queste estensioni formano la base dei moduli presenti nelle librerie standard di Python. Anche se le procedure per l'integrazione di questi moduli possono essere piuttosto complesse esiste uno strumento chiamato SWIG (Simplified Wrapper and Interface Generator) che permette di semplificare enormemente l'operazione.

D.1 Siti e libri su Python

Prima di procedere con le raccomandazioni degli autori, per quel che riguarda le risorse disponibili in Internet, ti consiglio di dare un'occhiata ai siti `www.zonapython.it`, `www.python.it` e `python.programmazione.it` che possono rappresentare un buon trampolino di lancio grazie anche (e soprattutto) al fatto di essere in lingua italiana.

- L'homepage di Python, `www.python.org` è il luogo dove iniziare ogni ricerca: troverai aiuto, documentazione, link ad altri siti e mailing list dei SIG (Special Interest Group) alle quali puoi eventualmente associarti.

- L'Open Book Project `www.ibiblio.com/obp` contiene non soltanto questo libro, ma versioni simili per Java e C++ scritti da Allen Downey. Inoltre potrai trovare una serie di altri documenti che spaziano dai circuiti elettrici a Python (*Python for Fun* di Chris Meyers), passando per il sistema operativo Linux (*The Linux Cookbook* by Michael Stultz, con 300 pagine di suggerimenti e tecniche). La versione tradotta in italiano e gli aggiornamenti saranno raggiungibili dal sito `www.zonapython.it`.

- Se poi vai su Google e cerchi "python -snake -monty" potrai rimanere stupito della mole di informazioni disponibili.

Per quanto concerne i libri, la bibliografia su Python, in italiano, si sta via via ampliando. Prova a chiedere in libreria: non ha neanche tanto senso indicare dei titoli quando il materiale disponibile è soggetto a variazioni così repentine. Non dimenticare che Python è un linguaggio giovane ed è soggetto a continue modifiche.

Per quanto concerne i libri in lingua inglese tra gli altri si distinguono quelli del nostro Alex Martelli. Una ricerca in www.amazon.com presenta circa 200 titoli disponibili. Tra questi consigliamo:

- *Python in a Nutshell* di Alex Martelli, è un ottimo riferimento per programmatori. Risolve brillantemente le difficoltà che insorgono quando è necessario ricordare la sintassi del linguaggio e dei suoi molti moduli, tratta sia le parti più usate delle librerie standard che le estensioni più conosciute.

- *Python Cookbook* di Alex Martelli e David Ascher, è un'ottima raccolta di "ricette" basate su esempi pratici e offre la soluzione a oltre 200 problemi in ogni campo di applicazione.

- *Core Python Programming* di Wesley Chun (750 pagine), copre il linguaggio a partire dai concetti fondamentale per arrivare a trattare di tecniche avanzate.

- *Python Essential Reference (2nd edition)* di David M. Beazley e Guido Van Rossum è molto ben fatto, tratta il linguaggio ed i moduli della libreria standard.

- *Python Pocket Reference* di Mark Lutz really, sebbene non così completo come la *Python Essential Reference* è un ottimo riferimento per le funzioni usate più frequentemente.

- *Python Programming on Win32* di Mark Hammond e Andy Robinson deve far parte della biblioteca di chiunque si appresti a programmare in Python in ambiente Windows.

D.2 Informatica in generale

Ecco qualche suggerimento per ulteriori letture, inclusi molti dei libri favoriti dagli autori. Trattano delle tecniche di programmazione da preferire e dell'informatica in generale.

- *The Practice of Programming* di Kernighan e Pike, oltre alla progettazione e alla codifica degli algoritmi e delle strutture di dati, tratta del debug, del test e del miglioramento delle performance dei programmi. Non ci sono esempi in Python.

- *The Elements of Java Style* di Al Vermeulen è un altro piccolo libro che tratta dei punti caratteristici della buona programmazione usando come riferimento il linguaggio Java.

- *Programming Pearls* di Jon Bentley è un classico e consiste di una raccolta di casi reali trattati dall'autore nella sua rubrica *Communications of the ACM*. Ciò che emerge è il concetto che raramente la prima idea per lo sviluppo di un programma è quella ottimale. È piuttosto datato (1986) ed è stato seguito da un secondo volume.
- *The New Turing Omnibus* di A.K Dewdney fornisce un'introduzione indolore a 66 argomenti correlati all'informatica, dall'elaborazione parallela ai virus, passando per gli algoritmi genetici. Tutti gli argomenti sono trattati in modo divertente e succinto.
- *Turtles, Termites and Traffic Jams* di Mitchel Resnick mostra come il comportamento complesso può nascere dal coordinamento di semplici attività delegate a molteplici agenti. Molti degli esempi del libro sono scritti in StarLogo, sono stati sviluppati da studenti e possono essere riscritti usando Python.
- *Gödel, Escher, Bach: un'eterna ghirlanda brillante* di Douglas Hofstadter. Se ti piace la ricorsione la troverai come protagonista di questo libro, pubblicato anche in lingua italiana[1]. L'autore dimostra la relazione esistente tra la musica di J.S.Bach, le immagini di Cornelius Escher ed il teorema dell'incompletezza di Gödel.

[1] Una lode meritata va al traduttore che ha dato il meglio di sé, districandosi con maestria nel labirinto dei giochi di parole originali

Appendice E

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft," which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

E.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document," below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical, or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

E.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in Section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

E.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

E.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of Sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History," and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- In any section entitled "Acknowledgements" or "Dedications," preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements," provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

E.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in Section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements," and any sections entitled "Dedications." You must delete all sections entitled "Endorsements."

E.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

E.7 Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate," and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of Section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

E.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of Section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

E.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense, or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

E.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http:///www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License or any later version applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

E.11 Addendum: How to Use This License for Your Documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License."

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Indice analitico