

Pensare in Python

Come pensare da Informatico

Versione 2.0.16

Pensare in Python

Come pensare da Informatico

Versione 2.0.16

Allen Downey

Green Tea Press

Needham, Massachusetts

Copyright © 2012 Allen Downey.

Green Tea Press
9 Washburn Ave
Needham MA 02492

È concessa l'autorizzazione a copiare, distribuire e/o modificare questo documento sotto i termini della Creative Commons Attribution-NonCommercial 3.0 Unported License, che è scaricabile dal sito <http://creativecommons.org/licenses/by-nc/3.0/>.

La forma originale di questo libro è in codice sorgente \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 codice sorgente \LaTeX di questo libro è disponibile all'indirizzo <http://www.thinkpython.com>.

Titolo originale: *Think Python: How to Think Like a Computer Scientist*

Traduzione di Andrea Zanella (andreazanella@tiscali.it).

Prefazione

La strana storia di questo libro

Nel gennaio 1999 mi stavo preparando a tenere un corso introduttivo di programmazione in Java. Lo insegnavo per la terza volta, ma la cosa stava diventando per me frustrante. Il tasso di insuccesso nel corso era troppo elevato, e anche per gli studenti che venivano promossi, il livello globale di apprendimento era troppo basso.

Uno dei problemi che avevo individuato erano i libri. Troppo grandi, con troppi dettagli non necessari su Java e privi di una guida di alto livello su come programmare. E tutti soffrivano dell' "effetto botola": cominciavano in modo semplice, procedevano gradualmente e poi, verso il Capitolo 5, mancava il pavimento sotto i piedi. Gli studenti si trovavano con troppo nuovo materiale e troppo velocemente, e io passavo il resto del semestre a raccogliere i cocci.

Due settimane prima dell'inizio delle lezioni, decisi allora di scrivere un libro tutto mio. I miei obiettivi erano:

- Mantenerlo breve. Gli studenti preferiscono leggere 10 pagine piuttosto che 50.
- Prestare attenzione ai vocaboli. Cercai di ridurre al minimo i termini gergali e di spiegare ciascun termine la prima volta che veniva usato.
- Costruire gradualmente. Per evitare le "botole", presi gli argomenti più ostici suddividendoli in una serie di piccoli passi.
- Focalizzare sulla programmazione, non sul linguaggio di programmazione. Inclusi la minima parte necessaria di Java e tralasciai il resto.

Mi serviva un titolo, così d'istinto scelsi *Come pensare da Informatico*.

La prima versione era grezza, ma funzionò. Gli studenti lo lessero, e capirono abbastanza da permettermi di impiegare il tempo della lezione per gli argomenti più difficili, per quelli interessanti e (cosa più importante) per la parte pratica.

Pubblicai il libro sotto la GNU Free Documentation License, che permette ai fruitori di copiare, modificare, e distribuire il libro.

Ma il bello venne dopo. Jeff Elkner, insegnante di liceo in Virginia, utilizzò il mio libro adattandolo per Python. Mi mandò una copia della sua versione, e io ebbi la insolita esperienza di imparare Python leggendo il mio stesso libro. Con la Green Tea Press, pubblicai la prima versione Python nel 2001.

Nel 2003 cominciai a lavorare all'Olin College, ed ottenni di insegnare Python per la prima volta. Il contrasto con Java fu abissale. Gli studenti dovettero faticare meno, impararono di più, lavorarono su progetti più interessanti, e in generale si divertirono di più.

Negli ultimi nove anni, ho continuato a sviluppare il libro, correggendo errori, migliorando alcuni esempi e aggiungendo nuovo materiale, soprattutto esercizi.

Il risultato è questo libro, che ora ha il meno grandioso titolo *Pensare in Python*. Ecco alcune novità:

- Ho aggiunto un paragrafo sul debug alla fine di ciascun capitolo. Questi paragrafi presentano le tecniche generali per scovare ed evitare gli errori, e le avvertenze sui trabocchetti di Python.
- Ho aggiunto altri esercizi, da brevi test di apprendimento ad alcuni progetti sostanziosi, scrivendo per la maggior parte di essi la soluzione.
- Ho aggiunto una serie di esercitazioni - esempi più articolati con esercizi, soluzioni e discussione. Alcuni sono basati su Swampy, una raccolta di programmi in Python che ho scritto a supporto delle mie lezioni. Swampy, il codice degli esempi e alcune soluzioni sono disponibili sul sito <http://thinkpython.com>.
- Ho ampliato la trattazione sui metodi di sviluppo di un programma e sugli schemi fondamentali di progettazione.
- Ho aggiunto delle appendici sul debug, l'analisi degli algoritmi e i diagrammi ULM con Lumpy.

Spero che troviate piacevole utilizzare questo libro, e che vi aiuti ad imparare a programmare e a pensare, almeno un pochino, da informatici.

Allen B. Downey
Needham MA

Allen Downey è Professore di Informatica presso il Franklin W. Olin College of Engineering.

Ringraziamenti

Grazie infinite a Jeff Elkner, che ha adattato il mio libro su Java in Python, ha dato inizio a questo progetto e mi ha introdotto in quello che poi è diventato il mio linguaggio di programmazione preferito.

Grazie anche a Chris Meyers, che ha contribuito ad alcuni paragrafi di *How to Think Like a Computer Scientist*.

Grazie alla Free Software Foundation per aver sviluppato la GNU Free Documentation License, che ha aiutato a rendere possibile la mia collaborazione con Jeff e Chris, e a Creative Commons per la licenza che uso attualmente.

Grazie ai redattori di Lulu che hanno lavorato su *How to Think Like a Computer Scientist*.

Grazie a tutti gli studenti che hanno usato le versioni precedenti di questo libro e a tutti coloro (elencati di seguito) che hanno contribuito inviando correzioni e suggerimenti.

Elenco dei collaboratori

Più di 100 lettori premurosi e dalla vista aguzza hanno inviato suggerimenti e correzioni negli anni passati. Il loro contributo e l'entusiasmo per questo progetto, sono stati di enorme aiuto.

Se volete proporre suggerimenti o correzioni, inviate una email a feedback@thinkpython.com. Se farò delle modifiche in seguito al vostro contributo, sarete aggiunti all'elenco dei collaboratori (a meno che non chiediate di non comparire).

Se includete almeno parte della frase in cui si trova l'errore, mi faciliterete la ricerca. Vanno bene anche numeri di pagina e di paragrafo, ma sono meno agevoli da trattare. Grazie!

- Lloyd Hugh Allen ha inviato una correzione al Paragrafo 8.4.
- Yvon Boulianne ha inviato una correzione a un errore di semantica nel Capitolo 5.
- Fred Bremmer ha inviato una correzione al Paragrafo 2.1.
- Jonah Cohen ha scritto gli script Perl per convertire i sorgenti LaTeX di questo libro in un meraviglioso HTML.
- Michael Conlon ha inviato una correzione grammaticale nel Capitolo 2 e un miglioramento dello stile nel Capitolo 1, e ha iniziato la discussione sugli aspetti tecnici degli interpreti.
- Benoit Girard ha inviato una correzione ad un umoristico errore nel Paragrafo 5.6.
- Courtney Gleason e Katherine Smith hanno scritto `horsebet.py`, che veniva usato come esercitazione in una versione precedente del libro. Ora il loro programma si può trovare sul sito web.
- Lee Harr ha sottoposto più correzioni di quelle che è possibile elencare in questo spazio, e pertanto andrebbe considerato come uno dei principali revisori del testo.
- James Kaylin è uno studente che ha usato il libro. Ha sottoposto numerose correzioni.
- David Kershaw ha sistemato la funzione guasta `catTwice` nel Paragrafo 3.10.
- Eddie Lam ha mandato molte correzioni ai Capitoli 1, 2, e 3. Ha anche sistemato il Makefile in modo che crei un indice alla prima esecuzione e ha aiutato nell'impostazione dello schema delle versioni.
- Man-Yong Lee ha inviato una correzione al codice di esempio nel Paragrafo 2.4.
- David Mayo ha puntualizzato che la parola "inconsiamente" nel Capitolo 1 doveva essere cambiata in "subconsciamente".
- Chris McAloon ha inviato alcune correzioni ai Paragrafi 3.9 e 3.10.
- Matthew J. Moelter è un collaboratore di lunga data che ha inviato numerose correzioni e suggerimenti al libro.
- Simon Dicon Montford ha comunicato una definizione di funzione mancante e alcuni errori di battitura nel Capitolo 3. Ha anche trovato un errore nella funzione `incremento` nel Capitolo 13.
- John Ouzts ha corretto la definizione di "valore di ritorno" nel Capitolo 3.
- Kevin Parks ha inviato preziosi commenti e suggerimenti su come migliorare la distribuzione del libro.

- David Pool ha inviato un errore di battitura nel glossario del Capitolo 1, e gentili parole di incoraggiamento.
- Michael Schmitt ha inviato correzioni al capitolo sui file e le eccezioni.
- Robin Shaw ha evidenziato un errore nel Paragrafo 13.1 dove la funzione `printTime` veniva usata in un esempio senza essere definita.
- Paul Sleigh ha trovato un errore nel Capitolo 7 e un bug nello script Perl di Jonah Cohen che genera HTML a partire da LaTeX.
- Craig T. Snyder sta provando il testo in un corso presso la Drew University. Ha contribuito con alcuni preziosi consigli e correzioni.
- Ian Thomas e i suoi studenti stanno usando il testo in un corso di programmazione. Sono i primi a collaudare i capitoli della seconda metà del libro, e hanno apportato numerose correzioni e suggerimenti.
- Keith Verheyden ha inviato una correzione al Capitolo 3.
- Peter Winstanley ci ha portato a conoscenza di un annoso errore nel nostro carattere latin nel Capitolo 3.
- Chris Wrobel ha apportato correzioni al codice nel capitolo su file I/O ed eccezioni.
- Moshe Zadka ha dato un inestimabile contributo a questo progetto. Oltre a scrivere la prima bozza del capitolo sui Dizionari, è stato una continua fonte di indicazioni nei primi abbozzi di questo libro.
- Christoph Zwerschke ha inviato alcune correzioni e suggerimenti pedagogici, e ha spiegato la differenza tra *gleich* e *selbe*.
- James Mayer ci ha mandato correzioni a un sacco di errori di battitura e di dizione, compresi due nell'elenco dei collaboratori.
- Hayden McAfee ha colto una incongruenza, fonte di probabile confusione, tra due esempi.
- Angel Arnal fa parte del gruppo internazionale di traduttori e lavora sulla versione spagnola. Ha trovato anche alcuni errori nella versione inglese.
- Tauhidul Hoque e Lex Berezhny hanno creato le illustrazioni del Capitolo 1 e migliorato molte delle altre.
- Il Dr. Michele Alzetta ha colto un errore nel Capitolo 8 e inviato alcuni interessanti commenti pedagogici su Fibonacci e Old Maid.
- Andy Mitchell ha trovato un errore di battitura nel Capitolo 1 e un esempio non funzionante nel Capitolo 2.
- Kalin Harvey ha suggerito un chiarimento nel Capitolo 7 e ha trovato alcuni errori di battitura.
- Christopher P. Smith ha trovato alcuni errori di battitura e ci ha aiutato ad aggiornare il libro a Python 2.2.
- David Hutchins ha trovato un errore di battitura nella Premessa.
- Gregor Lingl insegna Python in un liceo di Vienna, in Austria. Sta lavorando alla traduzione tedesca del libro e ha trovato un paio di brutti errori nel Capitolo 5.
- Julie Peters ha trovato un errore di battitura nella Premessa.

-
- Florin Oprina ha inviato un miglioramento in `makeTime`, una correzione in `printTime`, e un simpatico errore di battitura.
 - D. J. Webre ha suggerito un chiarimento nel Capitolo 3.
 - Ken ha trovato una manciata di errori nei Capitoli 8, 9 e 11.
 - Ivo Wever ha trovato un errore di battitura nel Capitolo 5 e ha suggerito un chiarimento nel Capitolo 3.
 - Curtis Yanko ha suggerito un chiarimento nel Capitolo 2.
 - Ben Logan ha evidenziato alcuni errori di battitura e dei problemi nella trasposizione del libro in HTML.
 - Jason Armstrong ha notato una parola mancante nel Capitolo 2.
 - Louis Cordier ha notato un punto del Capitolo 16 dove il codice non corrispondeva al testo.
 - Brian Cain ha suggerito dei chiarimenti nei Capitoli 2 e 3.
 - Rob Black ha inviato un'ampia raccolta di correzioni, inclusi alcuni cambiamenti per Python 2.2.
 - Jean-Philippe Rey dell'Ecole Centrale di Parigi ha inviato un buon numero di correzioni, inclusi degli aggiornamenti per Python 2.2 e altri preziosi miglioramenti.
 - Jason Mader della George Washington University ha dato parecchi utili suggerimenti e correzioni.
 - Jan Gundtofte-Bruun ci ha ricordato che "a error" è un errore.
 - Abel David e Alexis Dinno ci hanno ricordato che il plurale di "matrix" è "matrices", non "matrixes". Questo errore è rimasto nel libro per anni, ma due lettori con le stesse iniziali lo hanno segnalato nello stesso giorno. Curioso.
 - Charles Thayer ci ha spronati a sbarazzarci dei due punti che avevamo messo alla fine di alcune istruzioni, e a fare un uso più appropriato di "argomenti" e "parametri".
 - Roger Sperberg ha indicato un brano dalla logica contorta nel Capitolo 3.
 - Sam Bull ha evidenziato un paragrafo confuso nel Capitolo 2.
 - Andrew Cheung ha evidenziato due istanze di "uso prima di def."
 - C. Corey Capel ha notato una parola mancante nel Terzo Teorema del Debugging e un errore di battitura nel Capitolo 4.
 - Alessandra ha aiutato a sistemare un po' di confusione nelle Tartarughe.
 - Wim Champagne ha trovato un errore in un esempio di dizionario.
 - Douglas Wright ha trovato un problema con la divisione intera in `arco`.
 - Jared Spindor ha trovato alcuni scarti alla fine di una frase.
 - Lin Peiheng ha inviato una serie di suggerimenti molto utili.
 - Ray Hagtvedt ha sottoposto due errori e un non-abbastanza-errore.
 - Torsten Hübsch ha evidenziato un'incongruenza in `Swampy`.
 - Inga Petuhhov ha corretto un esempio nel Capitolo 14.

- Arne Babenhauserheide ha inviato alcune utili correzioni.
- Mark E. Casida è bravo bravo a trovare parole ripetute.
- Scott Tyler ha inserito una che mancava. E ha poi inviato una pila di correzioni.
- Gordon Shephard ha inviato alcune correzioni, tutte in email separate.
- Andrew Turner ha trovato un errore nel Capitolo 8.
- Adam Hobart ha sistemato un problema con la divisione intera in arco.
- Daryl Hammond e Sarah Zimmerman hanno osservato che ho servito `math.pi` troppo presto. E Zim ha trovato un errore di battitura.
- George Sass ha trovato un bug in un Paragrafo sul Debug.
- Brian Bingham ha suggerito l'Esercizio 11.10.
- Leah Engelbert-Fenton ha osservato che avevo usato `tuple` come nome di variabile, contro le mie stesse affermazioni. E poi ha trovato una manciata di errori di battitura e un "uso prima di def."
- Joe Funke ha trovato un errore di battitura.
- Chao-chao Chen ha trovato un'incoerenza nell'esempio su Fibonacci.
- Jeff Paine conosce la differenza tra `space` e `spam`.
- Lubos Pintes ha corretto un errore di battitura.
- Gregg Lind e Abigail Heithoff hanno suggerito l'Esercizio 14.4.
- Max Hailperin ha inviato parecchie correzioni e suggerimenti. Max è uno degli autori dello straordinario *Concrete Abstractions*, che potreste prendere in considerazione dopo aver letto questo libro.
- Chotipat Pornavalai ha trovato un errore in un messaggio di errore.
- Stanislaw Antol ha mandato un elenco di suggerimenti molto utili.
- Eric Pashman ha inviato parecchie correzioni ai Capitoli 4–11.
- Miguel Azevedo ha trovato alcuni errori di battitura.
- Jianhua Liu ha inviato un lungo elenco di correzioni.
- Nick King ha trovato una parola mancante.
- Martin Zuther ha inviato un lungo elenco di suggerimenti.
- Adam Zimmerman ha trovato un'incongruenza nella mia istanza di un' "istanza" e qualche altro errore.
- Ratnakar Tiwari ha suggerito una nota a piè di pagina per spiegare i triangoli degeneri.
- Anurag Goel ha suggerito un'altra soluzione per `alfabetica` e alcune altre correzioni. E sa come si scrive Jane Austen.
- Kelli Kratzer ha evidenziato un errore di battitura.
- Mark Griffiths ha osservato un esempio poco chiaro nel Capitolo 3.

- Roydan Ongie ha trovato un errore nel mio metodo di Newton.
- Patryk Wolowiec mi ha aiutato a risolvere un problema con la versione HTML.
- Mark Chonofsky mi ha riferito di una nuova parola riservata in Python 3.
- Russell Coleman mi ha aiutato con la geometria.
- Wei Huang ha evidenziato alcuni errori tipografici.
- Karen Barber ha trovato il più vecchio errore di battitura del libro.
- Nam Nguyen ha trovato un errore di battitura e ha osservato che avevo usato uno schema di Decoratore senza farne menzione.
- Stéphane Morin ha inviato alcune correzioni e suggerimenti.
- Paul Stoop ha corretto un errore di battitura in `usa_solo`.
- Eric Bronner ha notato un po' di confusione nella discussione dell'ordine delle operazioni.
- Alexandros Gezerlis ha fissato un nuovo standard per il numero e la qualità dei suoi suggerimenti. Gli siamo profondamente grati!
- Gray Thomas distingue la sua destra dalla sua sinistra.
- Giovanni Escobar Sosa ha inviato un lungo elenco di correzioni e suggerimenti.
- Alix Etienne ha sistemato uno degli URL.
- Kuang He ha trovato un errore di battitura.
- Daniel Neilson ha corretto un errore nell'ordine delle operazioni.
- Will McGinnis ha evidenziato che `pol` il `inea` era definita in modo diverso in due punti.
- Swarup Sahoo ha notato un punto e virgola mancante.
- Frank Hecker ha osservato un esercizio poco spiegato e alcuni collegamenti non funzionanti.
- Animesh B mi ha aiutato a spiegare meglio un esempio poco chiaro.
- Martin Caspersen ha trovato due errori di arrotondamento.
- Gregor Ulm ha inviato alcune correzioni e suggerimenti.
- Dimitrios Tsirigkas ha suggerito di chiarire meglio un esercizio.
- Carlos Tafur ha inviato una pagina di correzioni e suggerimenti.
- Martin Nordsletten ha trovato un bug nella soluzione di un esercizio.
- Lars O.D. Christensen ha trovato un riferimento non funzionante.
- Victor Simeone ha trovato un errore di battitura.
- Sven Hoexter ha osservato che una variabile di nome `input` oscura una funzione predefinita.
- Viet Le ha trovato un errore di battitura.
- Stephen Gregory ha evidenziato il problema di `cmp` in Python 3.
- Matthew Shultz mi ha comunicato un collegamento non funzionante.

- Lokesh Kumar Makani mi ha comunicato alcuni collegamenti non funzionanti e dei cambiamenti nei messaggi di errore.
- Ishwar Bhat ha corretto la mia formulazione dell'ultimo teorema di Fermat.
- Brian McGhie ha suggerito un chiarimento.
- Andrea Zanella ha tradotto il libro in italiano e, strada facendo, ha inviato alcune correzioni.

Indice

Prefazione	v
1 Lo scopo del programma	1
1.1 Il linguaggio di programmazione Python	1
1.2 Che cos'è un programma?	3
1.3 Che cos'è il debug?	3
1.4 Linguaggi formali e linguaggi naturali	5
1.5 Il primo programma	7
1.6 Debug	7
1.7 Glossario	8
1.8 Esercizi	9
2 Variabili, espressioni ed istruzioni	11
2.1 Valori e tipi	11
2.2 Variabili	12
2.3 Nomi delle variabili e parole chiave riservate	12
2.4 Operatori e operandi	13
2.5 Espressioni e istruzioni	14
2.6 Modalità interattiva e modalità script	14
2.7 Ordine delle operazioni	15
2.8 Operazioni sulle stringhe	16
2.9 Commenti	16
2.10 Debug	17
2.11 Glossario	17
2.12 Esercizi	18

3	Funzioni	21
3.1	Chiamate di funzione	21
3.2	Funzioni di conversione di tipo	21
3.3	Funzioni matematiche	22
3.4	Composizione	23
3.5	Aggiungere nuove funzioni	23
3.6	Definizioni e utilizzi	24
3.7	Flusso di esecuzione	25
3.8	Parametri e argomenti	26
3.9	Variabili e parametri sono locali	27
3.10	Diagrammi di stack	27
3.11	Funzioni produttive e funzioni vuote	28
3.12	Perché le funzioni?	29
3.13	Importare con <code>from</code>	29
3.14	Debug	30
3.15	Glossario	30
3.16	Esercizi	31
4	Esercitazione: Progettazione dell'interfaccia	35
4.1	TurtleWorld	35
4.2	Ripetizione semplice	36
4.3	Esercizi	37
4.4	Incapsulamento	38
4.5	Generalizzazione	39
4.6	Progettazione dell'interfaccia	39
4.7	Refactoring	40
4.8	Tecnica di sviluppo	41
4.9	Stringa di documentazione	42
4.10	Debug	42
4.11	Glossario	42
4.12	Esercizi	43

Indice	xv
5 Istruzioni condizionali e ricorsione	45
5.1 L'operatore modulo	45
5.2 Espressioni booleane	45
5.3 Operatori logici	46
5.4 Esecuzione condizionale	46
5.5 Esecuzione alternativa	47
5.6 Condizioni in serie	47
5.7 Condizioni nidificate	48
5.8 Ricorsione	48
5.9 Diagrammi di stack delle funzioni ricorsive	50
5.10 Ricorsione infinita	50
5.11 Input da tastiera	51
5.12 Debug	52
5.13 Glossario	53
5.14 Esercizi	54
6 Funzioni produttive	57
6.1 Valori di ritorno	57
6.2 Sviluppo incrementale	58
6.3 Composizione	60
6.4 Funzioni booleane	61
6.5 Altro sulla ricorsione	61
6.6 Salto sulla fiducia	63
6.7 Un altro esempio	64
6.8 Controllo dei tipi	64
6.9 Debug	65
6.10 Glossario	66
6.11 Esercizi	67

7	Iterazione	69
7.1	Assegnazioni multiple	69
7.2	Aggiornare le variabili	70
7.3	L'istruzione while	70
7.4	break	71
7.5	Radici quadrate	72
7.6	Algoritmi	73
7.7	Debug	74
7.8	Glossario	75
7.9	Esercizi	75
8	Stringhe	77
8.1	Una stringa è una sequenza	77
8.2	len	77
8.3	Attraversamento con un ciclo for	78
8.4	Slicing	79
8.5	Le stringhe sono immutabili	80
8.6	Ricerca	80
8.7	Cicli e contatori	81
8.8	Metodi delle stringhe	81
8.9	L'operatore in	82
8.10	Confronto di stringhe	83
8.11	Debug	83
8.12	Glossario	85
8.13	Esercizi	86
9	Esercitazione: Giochi con le parole	89
9.1	Leggere elenchi di parole	89
9.2	Esercizi	90
9.3	Ricerca	91
9.4	Cicli con gli indici	92
9.5	Debug	93
9.6	Glossario	94
9.7	Esercizi	94

Indice	xvii
10 Liste	97
10.1 Una lista è una sequenza	97
10.2 Le liste sono mutabili	97
10.3 Attraversamento di una lista	99
10.4 Operazioni sulle liste	99
10.5 Slicing delle liste	99
10.6 Metodi delle liste	100
10.7 Mappare, filtrare e ridurre	101
10.8 Cancellare elementi	102
10.9 Liste e stringhe	103
10.10 Oggetti e valori	104
10.11 Alias	105
10.12 Liste come argomenti	105
10.13 Debug	106
10.14 Glossario	108
10.15 Esercizi	108
11 Dizionari	111
11.1 Il dizionario come gruppo di contatori	112
11.2 Cicli e dizionari	114
11.3 Lookup inverso	114
11.4 Dizionari e liste	115
11.5 Memoizzazione	117
11.6 Variabili globali	118
11.7 Interi lunghi	120
11.8 Debug	120
11.9 Glossario	121
11.10 Esercizi	122

12 Tuple	123
12.1 Le tuple sono immutabili	123
12.2 Assegnazione di tupla	124
12.3 Tuple come valori di ritorno	125
12.4 Tuple di argomenti a lunghezza variabile	125
12.5 Liste e tuple	126
12.6 Dizionari e tuple	127
12.7 Confrontare tuple	129
12.8 Sequenze di sequenze	130
12.9 Debug	130
12.10 Glossario	131
12.11 Esercizi	131
13 Esercitazione: Scelta della struttura di dati	135
13.1 Analisi di frequenza delle parole	135
13.2 Numeri casuali	136
13.3 Istogramma di parole	137
13.4 Parole più comuni	138
13.5 Parametri opzionali	138
13.6 Sottrazione di dizionari	139
13.7 Parole a caso	140
13.8 Analisi di Markov	140
13.9 Strutture di dati	142
13.10 Debug	143
13.11 Glossario	144
13.12 Esercizi	144
14 File	147
14.1 Persistenza	147
14.2 Lettura e scrittura	147
14.3 L'operatore di formato	148
14.4 Nomi di file e percorsi	149

Indice	xix
14.5 Gestire le eccezioni	150
14.6 Database	151
14.7 Pickling	152
14.8 Pipe	152
14.9 Scrivere moduli	154
14.10 Debug	155
14.11 Glossario	155
14.12 Esercizi	156
15 Classi e oggetti	157
15.1 Tipi definiti dall'utente	157
15.2 Attributi	158
15.3 Rettangoli	159
15.4 Istanze come valori di ritorno	160
15.5 Gli oggetti sono mutabili	160
15.6 Copia	161
15.7 Debug	163
15.8 Glossario	163
15.9 Esercizi	164
16 Classi e funzioni	165
16.1 Tempo	165
16.2 Funzioni pure	166
16.3 Modificatori	167
16.4 Sviluppo prototipale e Sviluppo pianificato	168
16.5 Debug	169
16.6 Glossario	170
16.7 Esercizi	170

17 Classi e metodi	173
17.1 Funzionalità orientate agli oggetti	173
17.2 Stampa di oggetti	174
17.3 Un altro esempio	175
17.4 Un esempio più complesso	176
17.5 Il metodo speciale <code>init</code>	176
17.6 Il metodo speciale <code>__str__</code>	177
17.7 Operator overloading	177
17.8 Smistamento in base al tipo	178
17.9 Polimorfismo	179
17.10 Debug	180
17.11 Interfaccia e implementazione	181
17.12 Glossario	182
17.13 Esercizi	182
18 Ereditarietà	185
18.1 Oggetti Carta	185
18.2 Attributi di classe	186
18.3 Confrontare le carte	187
18.4 Mazzi di carte	188
18.5 Stampare il mazzo	189
18.6 Aggiungere, togliere, mescolare e ordinare	189
18.7 Ereditarietà	190
18.8 Diagrammi di classe	192
18.9 Debug	193
18.10 Incapsulamento dei dati	193
18.11 Glossario	195
18.12 Esercizi	195

Indice	xxi
19 Esercitazione: Tkinter	199
19.1 Interfacce grafiche	199
19.2 Pulsanti e callback	200
19.3 Controlli Canvas	201
19.4 Sequenze di coordinate	202
19.5 Altri controlli	202
19.6 Packing dei controlli	203
19.7 Menu e oggetti Callable	206
19.8 Binding	206
19.9 Debug	208
19.10 Glossario	209
19.11 Esercizi	210
A Debug	213
A.1 Errori di sintassi	213
A.2 Errori di runtime	215
A.3 Errori di semantica	218
B Analisi degli Algoritmi	223
B.1 Ordine di complessità	224
B.2 Analisi delle operazioni fondamentali di Python	226
B.3 Analisi degli algoritmi di ricerca	228
B.4 Tabelle hash	228
C Lumpy	233
C.1 Diagramma di stato	234
C.2 Diagramma di stack	234
C.3 Diagrammi di oggetto	235
C.4 Oggetti funzione e classe	237
C.5 Diagrammi di classe	238

Capitolo 1

Lo scopo del programma

Lo scopo di questo libro è insegnarvi a pensare da informatici. 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 esprimere concetti (nella fattispecie, elaborazioni). Come gli ingegneri, progettano cose, assemblano singoli componenti in sistemi e valutano costi e benefici 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 riuscire a schematizzarli, pensare creativamente alle possibili soluzioni ed esprimerle in modo chiaro ed accurato. Ne deriva che imparare a programmare è un'eccellente opportunità di mettere in pratica l'abilità di risolvere problemi. Ecco perché questo capitolo è chiamato "Lo scopo del programma".

Da una parte, vi verrà insegnato a programmare, che già di per sé è un'utile capacità. Dall'altra, userete la programmazione come un mezzo per raggiungere uno scopo. Man mano che procederemo, quello scopo vi diverrà più chiaro.

1.1 Il linguaggio di programmazione Python

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

Esistono anche **linguaggi di basso livello**, talvolta chiamati "linguaggi macchina" o "linguaggi assembly". Detto in breve, 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 procedimento di elaborazione impiega del tempo e rappresenta un piccolo svantaggio dei linguaggi di alto livello.

I vantaggi d'altra parte sono 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



Figura 1.1: Un interprete elabora il programma un po' per volta, leggendo le righe ed eseguendo i calcoli alternativamente.

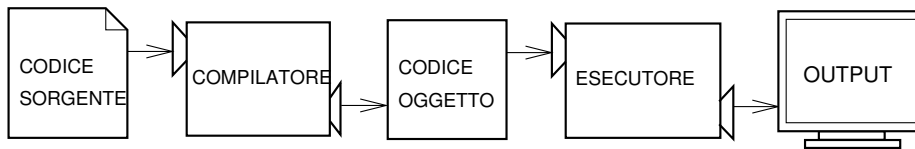


Figura 1.2: Un compilatore traduce il codice sorgente in codice oggetto, che viene avviato da un esecutore hardware.

di computer diversi con poche modifiche o addirittura nessuna. 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 talmente 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. La Figura 1.1 mostra la struttura di un interprete.

Un compilatore legge il programma di alto livello e lo traduce completamente in basso livello, prima che il programma venga 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 operazioni di traduzione. La Figura 1.2 mostra la struttura di un compilatore.

Python è considerato un linguaggio interpretato perché i programmi Python sono eseguiti da un interprete. Ci sono due modalità di usare l'interprete: la **modalità interattiva** e la **modalità script**. In modalità interattiva (detta anche "a riga di comando") si scrivono i programmi in Python e l'interprete elabora immediatamente il risultato:

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

La sequenza di caporali, `>>>`, è chiamata **prompt**, ed è l'indicazione con cui l'interprete comunica che è pronto ad accettare comandi. Se avete scritto `1 + 1`, l'interprete risponde con 2.

In alternativa alla riga di comando, si può scrivere e salvare un programma in un file di testo semplice, chiamato **script**, ed usare l'interprete per eseguire il contenuto del fi-

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

Per eseguire lo script, dobbiamo comunicare all'interprete il nome del file. Se avete uno script chiamato `dinsdale.py` e state lavorando in una finestra di comando UNIX, digitate `python dinsdale.py`. In altri ambienti di lavoro, i dettagli dell'esecuzione dei programmi variano. Potete trovare le istruzioni per il vostro sistema sul sito web di Python <http://python.org>.

Lavorare in modalità interattiva è conveniente per provare piccoli pezzi di codice, perché si possono inserire ed eseguire immediatamente. Ma per qualcosa di più di poche righe, è meglio salvare il codice in uno script, per poterlo eseguire o modificare in futuro senza doverlo riscrivere da capo ogni volta.

1.2 Che cos'è un programma?

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

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

input: ricezione di dati da tastiera, da un file o da un altro dispositivo.

output: scrittura di dati sullo schermo, su un file o trasmissione ad altro dispositivo.

matematiche: esecuzione di semplici operazioni matematiche, quali l'addizione e la moltiplicazione.

condizionali: controllo di certe condizioni ed esecuzione della sequenza di istruzioni appropriata.

ripetizione: ripetizione di un'azione, di solito con qualche variazione.

Che ci crediate o no, questo è più o meno tutto ciò che serve. Tutti i programmi che avete usato, non importa quanto complessi, sono fatti di istruzioni che assomigliano a queste. Possiamo affermare che la programmazione non è altro che la suddivisione di un compito grande e complesso in una serie di sotto-compiti via via più piccoli, finché non risultano 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 Che cos'è il debug?

La programmazione è soggetta ad errori. Per ragioni bizzarre, gli errori di programmazione sono chiamati **bug**, ed il procedimento della loro ricerca e correzione è chiamato **debug**.

Ci sono tre tipi di errori nei quali si incorre durante la programmazione: gli errori di sintassi, gli errori in esecuzione e gli errori di semantica. È utile analizzarli singolarmente per facilitarne l'individuazione.

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 restituisce un messaggio d'errore. La **sintassi** si riferisce alla struttura di un programma e alle regole che la governano. Ad esempio, le parentesi devono essere sempre presenti a coppie corrispondenti, così $(1 + 2)$ è corretto, ma $8)$ è un **errore di sintassi**.

Nelle lingue naturali, chi legge può tollerare la maggior parte degli errori di sintassi, tanto che gli inglesi possono leggere le poesie di e.e. cummings [prive di punteggiatura, NdT] senza emettere "messaggi d'errore". Ma Python non è così permissivo: se c'è un singolo errore di sintassi da qualche parte nel programma, Python visualizzerà un messaggio d'errore e ne interromperà l'esecuzione, rendendo impossibile proseguire. Durante le prime settimane della vostra carriera di programmatori, probabilmente passerete molto tempo a cercare errori di sintassi. Via via che acquisirete esperienza, questi si faranno meno numerosi e vi risulterà sempre più facile rintracciarli.

1.3.2 Errori in esecuzione

Il secondo tipo di errore è l'**errore in esecuzione** (o di *runtime*), così chiamato perché l'errore non appare finché il programma non viene eseguito. Questi errori sono anche chiamati **eccezioni** perché indicano che è accaduto qualcosa di eccezionale (e di spiacevole) nel corso dell'esecuzione.

Gli errori in esecuzione sono rari nei semplici programmi che vedrete nei primissimi capitoli, e potrebbe passare un po' di tempo prima di incontrarne uno.

1.3.3 Errori di semantica

Il terzo tipo di errore è l'**errore di semantica** (o di logica). In presenza di un errore di semantica, il programma verrà eseguito senza problemi, nel senso che il computer non genererà messaggi d'errore durante l'esecuzione; tuttavia il risultato non sarà quello che vi aspettate. Sarà qualcosa di diverso, ma questo qualcosa è esattamente ciò che voi avete detto di fare al computer.

Il problema sta nel fatto che il programma che avete scritto non è quello che volevate scrivere: il significato del programma (o la sua semantica) è sbagliato. L'identificazione degli errori di semantica può essere complicata perché richiede di lavorare a ritroso, partendo dai risultati dell'esecuzione e cercando di risalire a che cosa non sia andato per il verso giusto.

1.3.4 Debug sperimentale

Una delle più importanti abilità che acquisirete è la capacità di effettuare il debug (o "rimozione degli errori"). Sebbene questa possa essere un'operazione noiosa, 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. Siete messi di fronte a degli indizi e dovete ricostruire i processi e gli eventi che hanno portato ai risultati che avete ottenuto.

Il debug è come una scienza sperimentale: dopo aver ipotizzato quello che può essere andato storto, modificate il programma e riprovate. Se l'ipotesi era corretta, allora avete saputo predire il risultato della modifica e vi siete avvicinati di un ulteriore passo verso un programma funzionante. Se l'ipotesi era sbagliata, dovete formularne un'altra. Come disse Sherlock Holmes: "Quando hai eliminato l'impossibile, qualsiasi cosa rimanga, per quanto improbabile, deve essere la verità" (A. Conan Doyle, *Il segno dei quattro*)

Per alcuni, programmazione e debug sono la stessa cosa, intendendo con questo che la programmazione è un procedimento di graduale rimozione degli errori, fino a quando il programma non fa quello che vogliamo. L'idea è quella di 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 nacque come un semplice programma che Linus Torvalds usò per esplorare il chip Intel 80386. Secondo Larry Greenfields, "Uno dei progetti iniziali di Linus era un programma che doveva trasformare una sequenza di AAAA in BBBB e viceversa. Questo in seguito diventò Linux". (*The Linux Users' Guide Beta Version 1*).

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

1.4 Linguaggi formali e linguaggi 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; e, cosa più importante dal nostro punto di vista,

I linguaggi di programmazione sono linguaggi formali che sono stati progettati per esprimere delle elaborazioni.

I linguaggi formali tendono ad avere regole rigide per quanto riguarda la sintassi. Per esempio, $3 + 3 = 6$ è una espressione matematica sintatticamente corretta, mentre $3+ = 3\$6$ non lo è. H_2O è un simbolo chimico sintatticamente corretto, contrariamente a $_2Zz$.

Le regole sintattiche hanno due aspetti, che riguardano i **simboli** e la **struttura**. I simboli (in inglese *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 $3+ = 3\$6$ è che $\$$ non è un simbolo valido in matematica (almeno per quanto mi risulta). Allo stesso modo, $_2Zz$ non è valido perché nessun elemento chimico è identificato dal simbolo Zz .

Il secondo aspetto riguarda la struttura di un'espressione, cioè il modo in cui i simboli sono disposti. L'espressione $3+ = 3$ è strutturalmente non valida perché, anche se $+$ e $=$ sono dei simboli validi, non è possibile che uno segua immediatamente l'altro. Allo stesso modo, il pedice numerico nelle formule chimiche deve essere scritto dopo il simbolo dell'elemento chimico, e non prima.

Esercizio 1.1. *Scrivete una frase ben strutturata nella vostra lingua, contenente dei simboli non validi. Poi scrivete un'altra frase con tutti simboli validi ma con una struttura non valida.*

Quando leggete una frase in italiano o un'espressione in un linguaggio formale, dovete analizzare quale sia la struttura della frase (in un linguaggio naturale, questa operazione viene effettuata subconsciousamente). Questo processo di analisi è chiamato **parsing**.

Per esempio, quando sentite la frase "Mangiare la foglia", comprendete che "la foglia" è l'oggetto e "mangiare" è il predicato verbale. Una volta analizzata la frase, potete coglierne il significato (ovvero la semantica della frase). Partendo dal presupposto che sappiate cosa sia una "foglia" e cosa significhi "mangiare", riuscirete a comprendere il significato generale della frase.

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

ambiguità: i linguaggi naturali ne sono pieni, ed il significato viene ottenuto anche grazie ad altri indizi ricavati dal contesto. I linguaggi formali sono progettati per essere quasi o completamente privi di ambiguità, e ciò comporta che ciascuna dichiarazione ha un unico significato, indipendente dal contesto.

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

letteralità: i linguaggi naturali sono pieni di frasi idiomatiche e metafore. Se dico: "Mangiare la foglia", presumibilmente non c'è nessuna foglia e nessuno che la mangi (è un modo di dire di una persona che si rende conto di come stanno realmente le cose). I linguaggi formali invece esprimono esattamente ciò che dicono.

Poiché siamo tutti cresciuti parlando 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 voluta.

Prosa: il significato letterale delle parole è più 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, e può essere compreso nella sua totalità con l'analisi dei simboli e della struttura.

Ecco alcuni suggerimenti per la lettura dei programmi e degli altri linguaggi formali. Primo, ricordate che i linguaggi formali sono molto più ricchi di significato dei linguaggi naturali, per questo è necessario più tempo per leggerli e comprenderli. Poi, la struttura dei linguaggi formali è molto importante e di solito non è bene leggerli dall'alto in basso, da sinistra a destra, come avviene per un testo letterario: imparate ad analizzare il programma nella vostra testa, identificandone i simboli ed interpretandone la struttura. Infine, i dettagli sono importanti: piccoli errori di ortografia e punteggiatura sono spesso trascurabili nei linguaggi naturali, ma fanno una enorme differenza in quelli formali.

1.5 Il primo programma

Per tradizione, il primo programma scritto in un nuovo linguaggio è chiamato “Ciao, Mondo!”, perché tutto ciò che fa è scrivere a video le parole “Ciao, Mondo!”, e niente di più. In Python questo programma si scrive così:

```
print 'Ciao, Mondo!'
```

Questo è un esempio di **istruzione di stampa**, che a dispetto del nome non stampa nulla su carta, limitandosi invece a visualizzare un valore scrivendolo sullo schermo. In questo caso ciò che viene “stampato” sono le parole:

```
Ciao, Mondo!
```

Gli apici nell'istruzione segnano l'inizio e la fine del valore da stampare e non appaiono nel risultato.

Nell'ultima versione di Python, Python 3, la sintassi è leggermente cambiata:

```
print('Ciao, Mondo!')
```

Le parentesi indicano che, in questo caso, `print` è una funzione. Torneremo a parlare di funzioni nel Capitolo 3.

In questo libro, adotterò la classica istruzione `print`. Se voi utilizzate Python 3, dovrete adattarla come indicato sopra. Ma a parte questa, ci saranno pochissime altre differenze a cui dovrete prestare attenzione.

1.6 Debug

È opportuno leggere questo libro davanti al computer, in modo da poter provare gli esempi man mano che procedete nella lettura. Potete eseguire la maggior parte degli esempi in modalità interattiva, ma se scrivete il codice in uno script sarà più facile provare delle variazioni.

Ogni volta che sperimentate una nuova caratteristica, dovrete provare ad inserire degli errori. Ad esempio, nel programma “Ciao, mondo!”, cosa succede se dimenticate uno dei due apici? O entrambi? O se scrivete sbagliato `print`?

Esperimenti di questo tipo aiutano a ricordare quello che avete letto; aiutano anche nel debug, perché in questo modo imparate a conoscere il significato dei messaggi di errore. È meglio fare errori ora e di proposito, che più avanti e accidentalmente.

La programmazione, e specialmente il debug, a volte fanno emergere emozioni forti. Se siete alle prese con un bug difficile, vi può capitare di sentirvi arrabbiati, scoraggiati o in difficoltà.

Ci sono prove che le persone tendono naturalmente a rapportarsi con i computer come se fossero esseri umani. Se funzionano bene, li pensiamo come compagni di squadra, e quando sono ostinati o rudi, li trattiamo come trattiamo la gente rude o ostinata (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

Prepararsi a reazioni simili può aiutarvi ad affrontarle. Un possibile approccio è quello di pensare al computer come ad un impiegato con alcuni punti di forza, come velocità e precisione, e particolari debolezze, come mancanza di empatia e incapacità di cogliere il quadro generale.

Il vostro compito è di essere un buon manager: trovare il modo di trarre vantaggio dai pregi e mitigare i difetti. E trovare il modo di usare le vostre emozioni per affrontare i problemi, senza lasciare che le vostre reazioni interferiscano con la vostra capacità di lavorare in modo efficace.

Imparare a dare la caccia agli errori può essere noioso, ma è un'abilità preziosa, utile anche per tante altre attività oltre alla programmazione. Alla fine di ogni capitolo trovate un Paragrafo dedicato al debug, come questo, con le mie riflessioni in merito. Spero vi siano di aiuto!

1.7 Glossario

soluzione di problemi: Il procedimento di formulare un problema, trovare una soluzione ed esprimerla.

linguaggio di alto livello: Un linguaggio di programmazione come Python, progettato per essere facilmente leggibile e utilizzabile dagli uomini.

linguaggio di basso livello: Un linguaggio di programmazione che è progettato per essere facilmente eseguibile da un computer; è chiamato anche “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 una riga alla volta.

compilare: Tradurre tutto in una volta un programma scritto in un linguaggio di alto livello in un linguaggio di basso livello, preparandolo alla successiva esecuzione.

codice sorgente: Un programma in linguaggio 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.

prompt: Serie di caratteri mostrati dall'interprete per indicare che è pronto a ricevere input dall'utente.

script: Programma memorizzato in un file, di solito destinato ad essere interpretato.

modalità interattiva: Modo di usare l'interprete di Python che consiste nello scrivere comandi ed espressioni al prompt.

modalità script: Modo di usare l'interprete Python che consiste nel leggere ed eseguire le istruzioni contenute in uno script.

programma: Serie di istruzioni che specificano come effettuare un'elaborazione.

algoritmo: Procedimento generale usato per risolvere una particolare categoria di problemi.

bug: Un errore in un programma.

debug: Operazione di ricerca e di rimozione di ciascuno dei tre tipi di errori di programmazione.

sintassi: La struttura di un programma.

errore di sintassi: Errore in un programma che ne rende impossibile l'analisi (il programma non può quindi essere interpretato o compilato).

eccezione: Errore (detto anche di *runtime*) che si verifica mentre il programma viene eseguito.

semantica: Il significato logico di un programma.

errore di semantica: Errore nel programma tale per cui esso produce risultati diversi da quello che il programmatore intendeva.

linguaggio naturale: Qualunque linguaggio parlato che si è evoluto spontaneamente nel tempo.

linguaggio formale: Qualunque linguaggio progettato per scopi specifici, quali la rappresentazione di concetti matematici o programmi per computer (tutti i linguaggi di programmazione sono linguaggi formali).

simbolo o token: Uno degli elementi di base della struttura sintattica di un programma, analogo a una parola nei linguaggi naturali.

parsing: Esame e analisi della struttura sintattica di un programma.

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

1.8 Esercizi

Esercizio 1.2. Con un browser web, visitate il sito Internet di Python <http://python.org>. Queste pagine contengono informazioni su Python e collegamenti ad altre pagine correlate, e vi consentono di consultare la documentazione di Python.

Ad esempio, se inserite `print` nel campo di ricerca, il primo link che appare è la documentazione dell'istruzione `print`. In questo momento non vi sarà tutto quanto chiaro, ma è bene sapere che c'è.

Esercizio 1.3. Lanciate l'interprete di Python e scrivete `help()` per avviare la guida in linea. Oppure scrivete `help('print')` per avere informazioni sull'istruzione `print`.

Se qualcosa non funziona, probabilmente dovete installare della documentazione aggiuntiva o specificare una variabile di ambiente; i dettagli dipendono dal vostro sistema operativo e dalla versione di Python.

Esercizio 1.4. Avviate l'interprete di Python e utilizzatelo come calcolatrice. La sintassi di Python per le operazioni è quasi la stessa della notazione matematica consueta. Per esempio i simboli `+`, `-` e `/` denotano rispettivamente addizione, sottrazione, divisione. Il simbolo per la moltiplicazione è `*`.

Se correte una gara di 10 chilometri in 43 minuti e 30 secondi, qual è il vostro tempo medio sul miglio? E la vostra velocità media in miglia all'ora? (Suggerimento: un miglio equivale a 1,61 km)

Capitolo 2

Variabili, espressioni ed istruzioni

2.1 Valori e tipi

Un **valore** è uno degli elementi fondamentali con cui un programma lavora, come lo sono una lettera dell'alfabeto nella scrittura o un numero in matematica. I valori che abbiamo visto finora sono 1, 2, e 'Ciao, Mondo! '.

Questi valori appartengono a **tipi** diversi: 2 è un numero intero, e 'Ciao, Mondo!' è una **stringa**, così chiamata perché contiene una serie di caratteri. Voi (e l'interprete) potete identificare le stringhe dal fatto che sono racchiuse tra due apici.

Se non sapete a quale tipo appartenga un dato valore, l'interprete ve lo può dire:

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

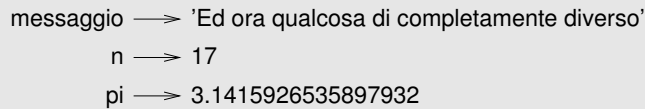
Ovviamente le stringhe sono di tipo `str` e gli interi di tipo `int`. Non è invece ovvio che i numeri con il punto decimale siano di tipo `float`: questi numeri sono rappresentati in un formato chiamato "a virgola mobile" o **floating-point**.

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

Cosa dire di valori come '17' e '3.2'? Sembrano a prima vista dei numeri, ma notate che sono racchiusi tra apici e questo sicuramente significa qualcosa. Infatti non sono numeri, ma stringhe:

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

Quando scrivete numeri grandi, potrebbe venirvi l'idea di usare delle virgole per delimitare i gruppi di tre cifre, come in 1,000,000. [Python utilizza la notazione anglosassone, per cui i separatori delle migliaia sono le virgole, mentre il punto è usato per separare le cifre decimali, NdT]. Questo non è un numero intero valido in Python, ma è comunque un qualcosa di consentito:



```
messaggio —> 'Ed ora qualcosa di completamente diverso'
n —> 17
pi —> 3.1415926535897932
```

Figura 2.1: Diagramma di stato.

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

Anche se non è quello che ci aspettavamo! Python in questo caso interpreta 1,000,000 come una sequenza di tre interi separati da virgole. Questo è il primo esempio di errore di semantica che abbiamo incontrato: il codice funziona senza produrre errori, ma fa qualcosa di diverso da quello che noi riteniamo “giusto”.

2.2 Variabili

Una delle caratteristiche più potenti in un linguaggio di programmazione è la capacità di lavorare con le **variabili**. Una variabile è un nome che fa riferimento ad un valore.

Un’**istruzione di assegnazione** crea nuove variabili e assegna loro un valore:

```
>>> messaggio = 'Ed ora qualcosa di completamente diverso'
>>> n = 17
>>> pi = 3.1415926535897932
```

Questo esempio effettua tre assegnazioni. La prima assegna una stringa ad una nuova variabile chiamata `messaggio`; la seconda assegna il numero intero 17 alla variabile `n`; la terza assegna il valore decimale approssimato di π 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 illustrazione è chiamato **diagramma di stato** perché mostra lo stato in cui si trova la variabile. La Figura 2.1 illustra il risultato delle istruzioni di assegnazione dell’esempio precedente.

Il tipo di una variabile è il tipo del valore a cui essa fa riferimento.

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

2.3 Nomi delle variabili e parole chiave riservate

Generalmente, i programmatori scelgono dei nomi significativi per le loro variabili, in modo da documentare a che cosa servono.

I nomi delle variabili possono essere lunghi a piacere e possono contenere sia lettere che numeri, ma devono sempre iniziare con una lettera. È possibile usare anche le lettere maiuscole, ma è bene che il nome cominci con una lettera minuscola (vedremo più avanti perché). Ricordate comunque che l'interprete le considera diverse, pertanto `Numero`, `NUmEro` e `numero` sono variabili diverse.

Il carattere di sottolineatura `_`, può far parte di un nome: è usato spesso in nomi di variabile composti da più parole (per esempio `il_mio_nome` o `monty_python`.)

Se assegnate un nome non valido alla variabile, otterrete un errore di sintassi:

```
>>> 76tromboni = 'grande banda'
SyntaxError: invalid syntax
>>> altro@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Zymurgy Teorico Avanzato'
SyntaxError: invalid syntax
```

`76tromboni` non è valido perché non inizia con una lettera. `altro@` non è valido perché contiene un carattere non ammesso (la chiocciola `@`). Ma cosa c'è di sbagliato in `class`?

Succede che `class` è una delle **parole chiave riservate** di Python. L'interprete utilizza queste parole per riconoscere la struttura del programma, pertanto non possono essere usate come nomi di variabili.

Python 2 ha 31 parole chiave riservate:

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

In Python 3, `exec` non è più una parola riservata, ma lo è `nonlocal`.

È consigliabile tenere questa lista a portata di mano: se l'interprete ha problemi con il nome che volete assegnare ad una variabile e non ne capite il motivo, provate a controllare se si trova in questa lista.

2.4 Operatori e operandi

Gli **operatori** sono simboli speciali usati per rappresentare calcoli come l'addizione e la moltiplicazione. I valori a cui si applicano gli operatori sono detti **operandi**.

Gli operatori `+`, `-`, `*`, `/` e `**` eseguono nell'ordine addizione, sottrazione, moltiplicazione, divisione ed elevamento a potenza, come negli esempi seguenti:

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

In altri linguaggi, viene usato `^` per le potenze, ma in Python è un operatore bitwise chiamato XOR. Non tratteremo questi operatori in questo libro, ma potete eventualmente approfondire l'argomento sul sito <http://wiki.python.org/moin/BitwiseOperators>.

In Python 2, l'operatore di divisione potrebbe comportarsi in modo inatteso:

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

Il valore di `minuti` è 59, e nell'aritmetica convenzionale 59 diviso 60 fa 0,98333 e non 0. La ragione di questa discrepanza è che Python compie una **divisione intera**. Quando entrambi gli operandi sono interi, anche il risultato è un intero; la divisione intera tronca la parte decimale, e in questo esempio arrotonda a 0.

In Python 3, invece, il risultato di questa divisione è un decimale `float`, e per eseguire una divisione intera si usa un nuovo operatore: `//`

Tuttavia, se almeno uno degli operandi è un numero decimale, Python esegue una divisione decimale e il risultato è un `float`:

```
>>> minuti/60.0
0.98333333333333328
```

2.5 Espressioni e istruzioni

Un'**espressione** è una combinazione di valori, variabili e operatori. Un valore è considerato già di per sé un'espressione, come pure una variabile, per cui quelle che seguono sono tutte delle espressioni valide (supponendo che alla variabile `x` sia già stato assegnato un valore):

```
17
x
x + 17
```

Un'**istruzione** è una porzione di codice che l'interprete Python può eseguire. Abbiamo già visto due tipi di istruzioni: istruzioni di stampa (`print`) e istruzioni di assegnazione.

Tecnicamente, un'espressione è anche un'istruzione, ma è forse più semplice considerarle come cose distinte. La differenza fondamentale è che l'espressione contiene un valore, l'istruzione no.

2.6 Modalità interattiva e modalità script

Uno dei vantaggi di lavorare con un linguaggio interpretato è quello di poter provare pezzi di codice in modalità interattiva prima di inserirli in uno script. Ma tra le due modalità, ci sono delle differenze che possono disorientare.

Per esempio, usando Python come una calcolatrice, potreste scrivere:

```
>>> miglia = 26.2
>>> miglia * 1.61
42.182
```

La prima riga assegna un valore a `miglia`, e non ha alcun effetto visibile. La seconda riga è un'espressione, e l'interprete la valuta e ne mostra il risultato. Vediamo così che una maratona misura circa 42 chilometri.

Ma se scrivete lo stesso codice in uno script e lo eseguite, non otterrete alcun riscontro. In modalità script, un'espressione, di per sé, non ha effetti visibili. In realtà Python valuta l'espressione, ma non ne mostra il risultato finché non gli dite esplicitamente di farlo:

```
miglia = 26.2  
print miglia * 1.61
```

Questo comportamento inizialmente può confondere.

Uno script di solito contiene una sequenza di istruzioni. Se ci sono più istruzioni, i risultati compaiono man mano che le istruzioni vengono eseguite.

Per esempio lo script:

```
print 1  
x = 2  
print x  
visualizza questo:
```

```
1  
2
```

mentre l'istruzione di assegnazione non produce alcun output sullo schermo.

Esercizio 2.1. *Scrivete le seguenti istruzioni nell'interprete Python per vedere quali effetti producono:*

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

Ora scrivete le stesse istruzioni in uno script ed eseguitelo. Qual è il risultato? Modificate lo script trasformando ciascuna espressione in un'istruzione di stampa, ed eseguitelo nuovamente.

2.7 Ordine delle operazioni

Quando più operatori compaiono in un'espressione, l'ordine in cui viene eseguito il calcolo dipende dalle **regole di precedenza**. Python segue le stesse regole di precedenza usate in matematica. L'acronimo **PEMDAS** è un modo utile per ricordare le regole:

- **Parentesi:** hanno il più alto livello di precedenza e possono essere usate per far valutare l'espressione in qualsiasi ordine volete. Dato che le espressioni tra parentesi sono valutate per prime, $2 * (3-1)$ fa 4, e $(1+1) ** (5-2)$ fa 8. Potete usare le parentesi per rendere più leggibile un'espressione come in $(\text{minuti} * 100) / 60$, 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 ad **Addizione e Sottrazione**, anch'esse aventi la stessa priorità. $2*3-1$ fa 5, e non 4, e $6+4/2$ fa 8, e non 5.
- **Gli operatori con la stessa priorità** sono valutati da sinistra verso destra (eccetto la potenza), così nell'espressione $\text{gradi} / 2 * \text{pi}$, la divisione viene calcolata per prima e il risultato viene moltiplicato per pi. Per dividere per 2π , dovete usare le parentesi o scrivere $\text{gradi} / 2 / \text{pi}$.

Personalmente, non mi sforzo molto di ricordare le regole di precedenza per gli altri operatori. Se non ne sono certo guardando un'espressione, inserisco le parentesi per fugare ogni dubbio.

2.8 Operazioni sulle stringhe

In genere non potete effettuare operazioni matematiche sulle stringhe, anche se il loro contenuto sembra essere un numero, quindi gli esempi che seguono non sono validi.

```
'2'-'1'      'uova'/'facili'      'terzo'*'una magia'
```

L'operatore `+` invece funziona con le stringhe, anche se non si comporta nel modo consueto: in questo caso esegue il **concatenamento**, cioè unisce le stringhe collegandole ai due estremi. Per esempio:

```
primo = 'bagno'
secondo = 'schiuma'
print primo + secondo
```

Il risultato a video di questo programma è bagnoschiuma.

Anche l'operatore `*` funziona sulle stringhe: ne esegue la ripetizione. Per esempio, `'Spam'*3` dà `'SpamSpamSpam'`. Uno degli operandi deve essere una stringa, l'altro un numero intero.

Questo utilizzo di `+` e `*` è coerente per analogia con l'addizione e la moltiplicazione in matematica. Così come $4*3$ è equivalente a $4+4+4$, ci aspettiamo che `'Spam'*3` sia lo stesso di `'Spam'+'Spam'+'Spam'`, ed effettivamente è così. D'altro canto, c'è un particolare sostanziale che rende diverse la somma e la moltiplicazione di numeri interi e di stringhe. Riuscite a pensare ad una proprietà che ha l'addizione ma che non vale per il concatenamento di stringhe?

2.9 Commenti

Man mano che il programma cresce di dimensioni e diventa più complesso, diventa anche 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 buona abitudine aggiungere delle note ai vostri programmi, per spiegare in linguaggio naturale cosa sta facendo il programma nelle sue varie parti. Queste note sono chiamate **commenti**, e sono demarcate dal simbolo `#`:

```
# calcola la percentuale di ora trascorsa
percentuale = (minuti * 100) / 60
```

In questo caso il commento appare su una riga a sé stante. Potete anche inserire un commento alla fine di una riga:

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

Qualsiasi cosa scritta dopo il simbolo `#` e fino alla fine della riga, viene trascurata nell'esecuzione del programma e non ha alcun effetto.

I commenti più utili sono quelli che documentano caratteristiche del codice di non immediata comprensione. È ragionevole supporre che chi legge il codice possa capire *cosa* esso faccia; è molto più utile spiegare *perché*.

Questo commento è ridondante e inutile:

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

Questo commento contiene un'informazione utile che non è contenuta nel codice:

```
v = 5      # velocita' in metri/secondo
```

Dei buoni nomi di variabile possono ridurre la necessità di commenti, ma nomi lunghi possono complicare la lettura, pertanto va trovato un giusto compromesso.

2.10 Debug

Giunti a questo punto, l'errore di sintassi che è più probabile commettere è di assegnare un nome non valido a una variabile, come `class` e `yield`, che sono parole chiave riservate, oppure `strano~lavoro` e `US$`, che contengono caratteri non ammessi.

Se inserite uno spazio nel nome di una variabile, Python li interpreta come due operandi senza un operatore:

```
>>> brutto nome = 5
SyntaxError: invalid syntax
```

Per gli errori di sintassi, il messaggio di errore non è di grande aiuto. I messaggi più comuni sono `SyntaxError: invalid syntax` e `SyntaxError: invalid token`, nessuno dei quali è molto informativo.

L'errore in esecuzione più frequente è un "uso prima di `def`" cioè cercare di utilizzare una variabile prima di averle assegnato un valore. Questo può succedere anche se scrivete il nome di una variabile sbagliato:

```
>>> capitale = 327.68
>>> interesse = capitlae * tasso
NameError: name 'capitlae' is not defined
```

I caratteri maiuscoli e minuscoli nei nomi delle variabili sono considerati diversi, per cui LaTeX non è la stessa cosa di latex.

La causa più probabile di errore di semantica è l'ordine delle operazioni. Per esempio, se per valutare l'operazione: $\frac{1}{2\pi}$, scrivete:

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

dato che la divisione viene calcolata per prima, ottenete invece $\pi/2$, che non è la stessa cosa! Python non ha modo di capire cosa volevate veramente scrivere, per cui in questo caso non vi risulta un messaggio di errore, ma una risposta sbagliata.

2.11 Glossario

valore: Una delle unità fondamentali di dati, come un numero o una stringa, che un programma può manipolare.

tipo: Una categoria di valori. I tipi che abbiamo visto finora sono i numeri interi (tipo `int`), i numeri in virgola mobile (tipo `float`), e le stringhe (tipo `str`).

intero: Tipo di dati che rappresenta i numeri interi.

virgola mobile: Tipo di dati che rappresenta i numeri con parte decimale; è detto anche *floating-point*.

stringa: Tipo di dati che rappresenta sequenze di caratteri.

variabile: Un nome che fa riferimento ad un valore.

istruzione: Parte di codice che rappresenta un comando o un'azione. Finora abbiamo visto istruzioni di assegnazione e istruzioni di stampa.

assegnazione: Istruzione che assegna un valore ad una variabile.

diagramma di stato: Rappresentazione grafica di una serie di variabili e dei valori ai quali esse si riferiscono.

parola chiave riservata: Parola chiave che il linguaggio usa per analizzare il programma e che non può essere usata come nome di variabile o di funzione, come `if`, `def`, e `while`.

operatore: Simbolo speciale che rappresenta un'operazione semplice tipo l'addizione, la moltiplicazione o il concatenamento di stringhe.

operando: Uno dei valori sui quali agisce un operatore.

divisione intera: Operazione che divide due numeri e tronca la parte decimale.

espressione: Combinazione di variabili, operatori e valori che rappresentano un unico valore risultante.

valutare: Semplificare un'espressione eseguendo una serie di operazioni per produrre un singolo valore.

regole di precedenza: Insieme di regole che determinano l'ordine dei calcoli di espressioni complesse in cui sono presenti più operandi ed operatori.

concatenare: Unire due stringhe tramite l'accodamento della seconda alla prima.

commento: Informazione inserita in un programma riguardante il significato di una sua parte; non ha alcun effetto sull'esecuzione del programma ma serve solo per facilitarne la comprensione.

2.12 Esercizi

Esercizio 2.2. *Supponiamo di eseguire le seguenti istruzioni di assegnazione:*

```
larghezza = 17
altezza = 12.0
delimitatore = '.'
```

Per ognuna delle seguenti espressioni, scrivetene il valore e il tipo (del valore risultante)

1. `larghezza/2`
2. `larghezza/2.0`
3. `altezza/3`
4. `1 + 2 * 5`

5. delimitatore * 5

Usate l'interprete Python per controllare le vostre risposte.

Esercizio 2.3. Fate un po' di pratica con l'interprete Python usandolo come calcolatrice:

1. Il volume di una sfera di raggio r è $\frac{4}{3}\pi r^3$. Che volume ha una sfera di raggio 5? Suggerimento: 392,7 è sbagliato!
2. Il prezzo di copertina di un libro è € 24,95, ma una libreria ottiene il 40% di sconto. I costi di spedizione sono € 3 per la prima copia e 75 centesimi per ogni copia aggiuntiva. Qual è il costo totale di 60 copie?
3. Se uscite di casa alle 6:52 di mattina e correte 1 miglio a ritmo blando (8:15 al miglio), poi 3 miglia a ritmo moderato (7:12 al miglio), quindi 1 altro miglio a ritmo blando, a che ora tornate a casa per colazione?

Capitolo 3

Funzioni

3.1 Chiamate di funzione

Nell’ambito della programmazione, una **funzione** è una sequenza di istruzioni che esegue un calcolo, alla quale viene assegnato un nome. Per definire una funzione, dovete specificarne il nome e scrivere la sequenza di istruzioni. In un secondo tempo, potete “chiamare” la funzione mediante il nome che le avete assegnato. Abbiamo già visto un esempio di una **chiamata di funzione**:

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

Il nome di questa funzione è `type`. L’espressione tra parentesi è chiamata **argomento** della funzione, e il risultato che produce è il tipo di valore dell’argomento che abbiamo inserito.

Si usa dire che una funzione “prende” o “riceve” un argomento e “ritorna” o “restituisce” un risultato. Il risultato è detto **valore di ritorno**.

3.2 Funzioni di conversione di tipo

Python fornisce una raccolta di funzioni predefinite o *built-in*, che convertono i valori da un tipo all’altro. La funzione `int` prende un dato valore e lo converte, se possibile, in intero. Se la conversione è impossibile compare un messaggio d’errore:

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

`int` può anche convertire valori in virgola mobile in interi, ma non arrotonda bensì tronca la parte decimale.

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

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

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

Infine, `str` converte l'argomento in una stringa:

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

3.3 Funzioni matematiche

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

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
```

Questa istruzione crea un **oggetto modulo** chiamato `math`. Se stampate l'oggetto modulo ottenete alcune informazioni su di esso:

```
>>> print math
<module 'math' (built-in)>
```

L'oggetto modulo contiene le funzioni e le variabili definite all'interno del modulo stesso. Per chiamare una funzione inclusa in un modulo, dobbiamo specificare nell'ordine il nome del modulo che la contiene e il nome della funzione, separati da un punto. Questo formato è chiamato **notazione a punto** o *dot notation*.

```
>>> rapporto = potenza_segnaile / potenza_rumore
>>> decibel = 10 * math.log10(rapporto)
```

```
>>> radianti = 0.7
>>> altezza = math.sin(radianti)
```

Il primo esempio utilizza `log10` per calcolare un rapporto segnale/rumore in decibel (a condizione che siano stati definiti i valori di `potenza_segnaile` e `potenza_rumore`). Il modulo `math` contiene anche `log`, che calcola i logaritmi naturali in base e .

Il secondo esempio calcola il seno della variabile `radianti`. Il nome della variabile spiega già che `sin` e le altre funzioni trigonometriche (`cos`, `tan`, ecc.) accettano argomenti espressi in radianti. Per convertire da gradi in radianti occorre dividere per 360 e moltiplicare per 2π :

```
>>> gradi = 45
>>> radianti = gradi / 360.0 * 2 * math.pi
>>> math.sin(radianti)
0.707106781187
```

L'espressione `math.pi` ricava la variabile `pi` dal modulo matematico. Il suo valore è un'approssimazione di π , accurata a circa 15 cifre.

Se ricordate la trigonometria, potete verificare il risultato precedente confrontandolo con la radice quadrata di due diviso due:

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

3.4 Composizione

Finora, abbiamo considerato gli elementi di un programma - variabili, espressioni e istruzioni - separatamente, senza parlare di come utilizzarli insieme.

Una delle caratteristiche più utili dei linguaggi di programmazione è la loro capacità di prendere dei piccoli pezzi e **comporli** tra loro. Per esempio, l'argomento di una funzione può essere un qualunque tipo di espressione, incluse operazioni aritmetiche:

```
x = math.sin(gradi / 360.0 * 2 * math.pi)
```

E anche chiamate di funzione:

```
x = math.exp(math.log(x+1))
```

Potete mettere quasi ovunque un valore o un'espressione a piacere, con una eccezione: il lato sinistro di una istruzione di assegnazione deve essere un nome di una variabile. Ogni altra espressione darebbe un errore di sintassi (vedremo più avanti le eccezioni a questa regola).

```
>>> minuti = ore * 60                # giusto
>>> ore * 60 = minuti                # sbagliato!
SyntaxError: can't assign to operator
```

3.5 Aggiungere nuove funzioni

Finora abbiamo soltanto usato solo funzioni che sono parte integrante di Python, ma è anche possibile aggiungerne di nuove. Una **definizione di funzione** specifica il nome di una nuova funzione e la sequenza di istruzioni che viene eseguita quando la funzione viene chiamata.

Ecco un esempio:

```
def stampa_brani():
    print "Terror di tutta la foresta egli e',"
    print "Con l'ascia in mano si sente un re."
```

`def` è una parola chiave riservata che indica la definizione di una nuova funzione. Il nome della funzione è `stampa_brani`. Le regole per i nomi delle funzioni sono le stesse dei nomi delle variabili: lettere, numeri e alcuni segni di interpunzione sono permessi, ma il primo carattere non può essere un numero. Non si possono usare parole riservate, e bisogna evitare di avere una funzione e una variabile con lo stesso nome.

Le parentesi vuote dopo il nome indicano che la funzione non accetta alcun argomento.

La prima riga della definizione di funzione è chiamata **intestazione**; il resto è detto **corpo**. L'intestazione deve terminare con i due punti e il corpo deve essere indentato, cioè deve avere un rientro rispetto all'intestazione. Per convenzione, l'indentazione è sempre di quattro spazi (vedi Paragrafo 3.14). Il corpo può contenere un qualsiasi numero di istruzioni.

Le stringhe nelle istruzioni print sono racchiuse tra virgolette (" "). Le virgolette e gli apici (' ') sono equivalenti; la maggioranza degli utenti usa gli apici, eccetto nei casi in cui nel testo da stampare sono contenuti degli apici (che possono essere usati anche come apostrofi o accenti). In questi casi, bisogna usare le virgolette.

Se scrivete una funzione in modalità interattiva, l'interprete mette tre puntini di sospensione (...) per indicare che la definizione non è completa:

```
>>> def stampa_brani():
...     print "Terror di tutta la foresta egli e',"
...     print "Con l'ascia in mano si sente un re."
... 
```

Per concludere la funzione, dovete inserire una riga vuota (questo non è necessario in uno script).

La definizione di una funzione crea una variabile con lo stesso nome:

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

Il valore di stampa_brani è un **oggetto funzione**, che è di tipo 'function'.

La sintassi per chiamare la nuova funzione è la stessa che abbiamo visto per le funzioni predefinite:

```
>>> stampa_brani()
Terror di tutta la foresta egli e',
Con l'ascia in mano si sente un re.
```

Una volta definita una funzione, si può utilizzarla all'interno di un'altra funzione. Per esempio, per ripetere due volte il brano precedente possiamo scrivere una funzione ripeti_brani:

```
def ripeti_brani():
    stampa_brani()
    stampa_brani()
```

E quindi chiamare ripeti_brani:

```
>>> ripeti_brani()
Terror di tutta la foresta egli e',
Con l'ascia in mano si sente un re.
Terror di tutta la foresta egli e',
Con l'ascia in mano si sente un re.
```

Ma a dire il vero, la canzone non fa così!

3.6 Definizioni e utilizzi

Raggruppando assieme i frammenti di codice del Paragrafo precedente il programma diventa:

```
def stampa_brani():
    print "Terror di tutta la foresta egli e',"

```

```
print "Con l'ascia in mano si sente un re."

def ripeti_brani():
    stampa_brani()
    stampa_brani()

ripeti_brani()
```

Questo programma contiene due definizioni di funzione: `stampa_brani` e `ripeti_brani`. Le definizioni di funzione sono eseguite come le altre istruzioni, ma il loro effetto è solo quello di creare una nuova funzione. Le istruzioni all'interno di una definizione non sono eseguite finché la funzione non viene chiamata, e la definizione di per sé non genera alcun risultato.

Come potete immaginare, una funzione deve essere definita prima di poterla usare: la definizione della funzione deve sempre precedere la sua chiamata.

Esercizio 3.1. *Spostate l'ultima riga del programma all'inizio, per fare in modo che la chiamata della funzione appaia prima della definizione. Eseguite il programma e guardate che tipo di messaggio d'errore ottenete.*

Esercizio 3.2. *Riportate la chiamata della funzione al suo posto, e spostate la definizione di `stampa_brani` dopo la definizione di `ripeti_brani`. Cosa succede quando eseguite il programma?*

3.7 Flusso di esecuzione

Per assicurarvi che una funzione sia definita prima del suo uso, dovete conoscere l'ordine in cui le istruzioni vengono eseguite, cioè il **flusso di esecuzione** del programma.

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

Le definizioni di funzione non alterano il flusso di esecuzione del programma ma va ricordato che le istruzioni all'interno delle funzioni non vengono eseguite fino a quando la funzione non viene chiamata.

Una chiamata di funzione è una sorta di 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.

Sinora è tutto abbastanza semplice, ma dovete tenere conto che una funzione può chiamarne un'altra al suo interno. Nel bel mezzo di una funzione, il programma può dover eseguire le istruzioni situate in un'altra funzione. Ma mentre esegue la nuova funzione, il programma può doverne eseguire un'altra ancora!

Fortunatamente, Python sa tener bene traccia di dove si trova, e ogni volta che una funzione viene completata il programma ritorna al punto che aveva lasciato. Giunto all'ultima istruzione, dopo averla eseguita, il programma termina.

Morale della favola? Quando leggete un programma non limitatevi sempre a farlo dall'alto in basso. Spesso ha più senso cercare di seguire il flusso di esecuzione.

3.8 Parametri e argomenti

Alcune delle funzioni che abbiamo visto richiedono degli argomenti. Per esempio, se volete trovare il seno di un numero chiamando la funzione `math.sin`, dovete passarle quel numero come argomento. Alcune funzioni ricevono più di un argomento: a `math.pow` ne servono due, che sono la base e l'esponente dell'operazione di elevamento a potenza.

All'interno della funzione, gli argomenti che le vengono passati sono assegnati ad altrettante variabili chiamate **parametri**. Ecco un esempio di una funzione che riceve un argomento:

```
def stampa2volte(bruce):
    print bruce
    print bruce
```

Questa funzione assegna l'argomento ricevuto ad un parametro chiamato `bruce`. Quando la funzione viene chiamata, stampa il valore del parametro (qualunque esso sia) due volte.

Questa funzione lavora con qualunque valore che possa essere stampato.

```
>>> stampa2volte('Spam')
Spam
Spam
>>> stampa2volte(17)
17
17
>>> stampa2volte(math.pi)
3.14159265359
3.14159265359
```

Le stesse regole di composizione che valgono per le funzioni predefinite si applicano anche alle funzioni definite dall'utente, pertanto possiamo usare come argomento per `stampa2volte` qualsiasi espressione:

```
>>> stampa2volte('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> stampa2volte(math.cos(math.pi))
-1.0
-1.0
```

L'argomento viene valutato prima della chiamata alla funzione, pertanto nell'esempio appena proposto le espressioni `'Spam '*4` e `math.cos(math.pi)` vengono valutate una volta sola.

Potete anche usare una variabile come argomento di una funzione:

```
>>> michael = 'Eric, the half a bee.'
>>> stampa2volte(michael)
Eric, the half a bee.
Eric, the half a bee.
```

Il nome della variabile che passiamo come argomento (`michael`) non ha niente a che fare con il nome del parametro nella definizione della funzione (`bruce`). Non ha importanza come era stato denominato il valore di partenza (nel codice chiamante); qui in `stampa2volte`, chiamiamo tutto quanto `bruce`.

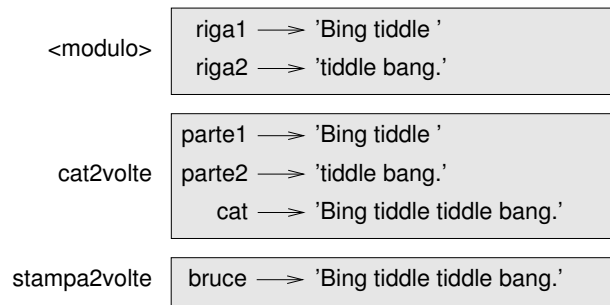


Figura 3.1: Diagramma di stack.

3.9 Variabili e parametri sono locali

Quando create una variabile in una funzione, essa è **locale**, cioè esiste solo all'interno della funzione. Per esempio:

```
def cat2volte(parte1, parte2):
    cat = parte1 + parte2
    stampa2volte(cat)
```

Questa funzione prende due argomenti, li concatena e poi ne stampa il risultato due volte. Ecco un esempio che la utilizza:

```
>>> riga1 = 'Bing tiddle '
>>> riga2 = 'tiddle bang.'
>>> cat2volte(riga1, riga2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

Quando `cat2volte` termina, la variabile `cat` viene distrutta. Se provassimo a stamparla, otterremmo infatti un messaggio d'errore:

```
>>> print cat
NameError: name 'cat' is not defined
```

Anche i parametri sono locali: al di fuori della funzione `stampa2volte`, non esiste alcuna cosa chiamata `bruce`.

3.10 Diagrammi di stack

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

Ogni funzione è rappresentata da un **frame**, un riquadro con il nome della funzione a fianco e la lista dei suoi parametri e delle sue variabili all'interno. Il diagramma di stack nel caso dell'esempio precedente, è illustrato in Figura 3.1.

I frame sono disposti in una pila che indica quale funzione ne ha chiamata un'altra e così via. Nell'esempio, `stampa2volte` è stata chiamata da `cat2volte`, e `cat2volte` è stata a sua volta chiamata da `__main__`, che è un nome speciale per il frame principale. Quando si crea una variabile che è esterna ad ogni funzione, essa appartiene a `__main__`.

Ogni parametro fa riferimento allo stesso valore del suo argomento corrispondente. Così, `parte1` ha lo stesso valore di `riga1`, `parte2` ha lo stesso valore di `riga2`, e `bruce` ha lo stesso valore di `cat`.

Se si verifica 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 a sua volta ha chiamato quest'ultima e così via, fino a raggiungere il primo livello che è sempre `__main__`.

Ad esempio se cercate di accedere a `cat` dall'interno di `stampa2volte`, ottenete un errore di tipo `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat2volte(riga1, riga2)
  File "test.py", line 5, in cat2volte
    stampa2volte(cat)
  File "test.py", line 9, in stampa2volte
    print cat
NameError: name 'cat' is not defined
```

Questo elenco di funzioni è detto **traceback**. Il traceback vi dice in quale file è avvenuto l'errore, e in quale riga, e quale funzione era in esecuzione in quel momento. Mostra anche la riga di codice che ha causato l'errore.

L'ordine delle funzioni nel traceback è lo stesso di quello dei frame nel diagramma di stack. La funzione attualmente in esecuzione si trova in fondo all'elenco.

3.11 Funzioni produttive e funzioni vuote

Alcune delle funzioni che abbiamo usato, tipo le funzioni matematiche, restituiscono dei risultati; in mancanza di definizioni migliori, personalmente le chiamo **funzioni produttive**. Altre funzioni, come `stampa2volte`, eseguono un'azione ma non restituiscono alcun valore. Le chiameremo **funzioni vuote**.

Quando chiamate una funzione produttiva, quasi sempre è per fare qualcosa di utile con il suo risultato, tipo assegnarlo a una variabile o usarlo come parte di un'espressione.

```
x = math.cos(radiani)
aureo = (math.sqrt(5) + 1) / 2
```

Se chiamate una funzione in modalità interattiva, Python ne mostra il risultato:

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

Ma in uno script, se chiamate una funzione produttiva così come è, il valore di ritorno è perso!

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

Questo script in effetti calcola la radice quadrata di 5, ma non conserva nè visualizza il risultato, per cui non è di grande utilità.

Le funzioni vuote possono visualizzare qualcosa sullo schermo o avere qualche altro effetto, ma non restituiscono un valore. Se provate comunque ad assegnare il risultato ad una variabile, ottenete un valore speciale chiamato `None` (nulla).

```
>>> risultato = stampa2volte('Bing')
Bing
Bing
>>> print risultato
None
```

Il valore `None` non è la stessa cosa della stringa `'None'`. È un valore speciale che appartiene ad un tipo tutto suo:

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

Le funzioni che abbiamo scritto finora, sono tutte vuote. Cominceremo a scriverne di produttive tra alcuni capitoli.

3.12 Perché le funzioni?

Potrebbe non esservi ancora ben chiaro perché valga la pena di suddividere il programma in funzioni. Ecco alcuni motivi:

- Creare una nuova funzione vi dà modo di dare un nome a un gruppo di istruzioni, rendendo il programma più facile da leggere e da correggere.
- Le funzioni possono rendere un programma più breve eliminando il codice ripetitivo. Se in un secondo tempo dovete fare una modifica, basterà farla in un posto solo.
- Dividere un programma lungo in funzioni vi permette di correggere le parti una per una, per poi assemblarle in un complesso funzionante.
- Funzioni ben fatte sono spesso utili per più programmi. Quando ne avete scritta e corretta una, la potete riutilizzare tale e quale.

3.13 Importare con `from`

Python fornisce due metodi per importare i moduli. Ne abbiamo già incontrato uno:

```
>>> import math
>>> print math
<module 'math' (built-in)>
>>> print math.pi
3.14159265359
```

Quando importate `math`, ottenete un oggetto modulo di nome `math` che contiene delle costanti come `pi` (il pi-greco) e funzioni come `sin` e `exp`.

Ma se tentate di accedere a `pi` direttamente, otterrete un errore.

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

Un'alternativa è di importare un oggetto da un modulo in questa maniera:

```
>>> from math import pi
```

Ora potete accedere a `pi` direttamente, senza usare la notazione a punto.

```
>>> print pi
3.14159265359
```

Oppure potete usare l'operatore asterisco per importare *tutto* da un modulo:

```
>>> from math import *
>>> cos(pi)
-1.0
```

Il vantaggio di importare tutto dal modulo matematico è che il vostro codice diventa più conciso. Per contro, potrebbero insorgere delle omonimie tra nomi definiti in moduli diversi, o tra un nome del modulo importato e una delle vostre variabili.

3.14 Debug

Se usate un editor di testo per scrivere gli script, potreste incontrare dei problemi nell'indentare il codice con spazi e tabulazioni. Il modo migliore per evitare problemi di questo tipo è usare esclusivamente gli spazi, non le tabulazioni. La maggior parte degli editor studiati per Python lo fanno in modo predefinito, ma alcuni no.

Tabulazioni e spazi di solito sono invisibili, rendendo difficoltoso il debug: cercate quindi di usare un editor che gestisca l'indentazione automaticamente.

Non dimenticate di salvare il programma prima di eseguirlo. Alcuni ambienti di sviluppo lo fanno automaticamente, ma altri no. In quest'ultimo caso, il programma che eseguite potrebbe non essere lo stesso che state guardando nell'editor.

Il debug può richiedere molto tempo se continuate a eseguire tutte le volte lo stesso programma non corretto!

Siate certi che il codice che state guardando sia lo stesso che eseguite. Se avete qualche dubbio, mettete qualcosa come `print 'ciao'` all'inizio del programma ed eseguitelo di nuovo. Se non vi compare ciao, allora non state eseguendo il programma giusto!

3.15 Glossario

funzione: Una sequenza di istruzioni dotata di un nome che esegue una certa operazione utile. Le funzioni possono o meno ricevere argomenti e possono o meno produrre un risultato.

definizione di funzione: Istruzione che crea una nuova funzione, specificandone il nome, i parametri, e le istruzioni che esegue.

oggetto funzione: Valore creato da una definizione di funzione. Il nome della funzione è una variabile che fa riferimento a un oggetto funzione.

intestazione: La prima riga di una definizione di funzione.

corpo: La sequenza di istruzioni all'interno di una definizione di funzione.

parametro: Un nome usato all'interno di una funzione che fa riferimento al valore passato come argomento.

chiamata di funzione: Istruzione che esegue una funzione. Consiste nel nome della funzione seguito da un elenco di argomenti.

argomento: Un valore fornito (passato) a una funzione quando viene chiamata. Questo valore viene assegnato al corrispondente parametro nella funzione.

variabile locale: Variabile definita all'interno di una funzione e che può essere usata solo all'interno della funzione.

valore di ritorno: Il risultato di una funzione. Se una chiamata di funzione viene usata come espressione, il valore di ritorno è il valore dell'espressione.

funzione produttiva: Una funzione che restituisce un valore.

funzione vuota: Una funzione che non restituisce un valore.

modulo: Un file che contiene una raccolta di funzioni correlate e altre definizioni.

istruzione import: Istruzione che legge un file modulo e crea un oggetto modulo utilizzabile.

oggetto modulo: Valore creato da un'istruzione import che fornisce l'accesso ai valori definiti in un modulo.

dot notation o notazione a punto: Sintassi per chiamare una funzione di un modulo diverso, specificando il nome del modulo seguito da un punto e dal nome della funzione.

composizione: Utilizzare un'espressione come parte di un'espressione più grande o un'istruzione come parte di un'istruzione più grande.

flusso di esecuzione: L'ordine in cui vengono eseguite le istruzioni nel corso di un programma.

diagramma di stack: Rappresentazione grafica di una serie di funzioni impilate, delle loro variabili e dei valori a cui fanno riferimento.

frame: Un riquadro in un diagramma di stack che rappresenta una chiamata di funzione. Contiene le variabili locali e i parametri della funzione.

traceback: Elenco delle funzioni in corso di esecuzione, stampato quando si verifica un errore.

3.16 Esercizi

Esercizio 3.3. *Python contiene una funzione predefinita chiamata `len` che restituisce la lunghezza di una stringa, ad esempio il valore di `len('allen')` è 5.*

Scrivete una funzione chiamata `giustif_destra` che richieda una stringa `s` come parametro e stampi la stringa con tanti spazi iniziali da far sì che l'ultima lettera della stringa cada nella colonna 70 del display.

```
>>> giustif_destra('allen')
                                allen
```

Esercizio 3.4. Un oggetto funzione è un valore che potete assegnare a una variabile o passare come argomento. Ad esempio, `fai2volte` è una funzione che accetta un oggetto funzione come argomento e la chiama per due volte.

```
def fai2volte(f):
    f()
    f()
```

Ecco un esempio che usa `fai2volte` per chiamare una funzione di nome `stampa_spam` due volte.

```
def stampa_spam():
    print 'spam'
```

```
fai2volte(stampa_spam)
```

1. Scrivete questo esempio in uno script e provatelo.
2. Modificate `fai2volte` in modo che accetti due argomenti, un oggetto funzione e un valore, e che chiami la funzione due volte passando il valore come argomento.
3. Scrivete una versione più generale di `stampa_spam`, di nome `stampa_2volte`, che richieda una stringa come parametro e la stampi due volte.
4. Usate la versione modificata di `fai2volte` per chiamare `stampa_2volte` per due volte, passando `'spam'` come argomento.
5. Definite una nuova funzione di nome `fai_quattro` che richieda un oggetto funzione e un valore e chiami la funzione per 4 volte, passando il valore come argomento. Dovrebbero esserci solo due istruzioni nel corpo di questa funzione, non quattro.

Soluzione: http://thinkpython.com/code/do_four.py.

Esercizio 3.5. Questo esercizio può essere svolto con le sole istruzioni e caratteristiche del linguaggio imparate finora.

1. Scrivete una funzione che disegni una griglia come questa:

```
+ - - - + - - - +
|       |       |
|       |       |
|       |       |
+ - - - + - - - +
|       |       |
|       |       |
|       |       |
+ - - - + - - - +
```

Suggerimento: per stampare più di un valore per riga, usate `print` con una sequenza di valori separati da virgole:

```
print '+', '-'
```

Se la sequenza termina con una virgola, Python lascia in sospeso la riga, e il successivo valore stampato compare sulla stessa riga.

```
print '+',  
print '-'
```

L'output di questa istruzione è '+ -'.

Un'istruzione `print` a sé stante, termina la riga e va a capo.

2. *Scrivete una funzione che disegni una griglia simile, con quattro righe e quattro colonne.*

Soluzione: <http://thinkpython.com/code/grid.py>. Fonte: Esercizio tratto da Oualline, Practical C Programming, Third Edition, O'Reilly Media, 1997.

Capitolo 4

Esercitazione: Progettazione dell'interfaccia

Il codice degli esempi di questo capitolo è scaricabile dal sito <http://thinkpython.com/code/polygon.py>.

4.1 TurtleWorld

Come ausilio a questo libro, ho preparato un pacchetto chiamato Swampy. Potete scaricarlo dal sito <http://thinkpython.com/swampy>; per installarlo nel vostro sistema, seguite le istruzioni pubblicate sul sito.

Un **pacchetto** è una raccolta di moduli; uno dei moduli di Swampy si chiama TurtleWorld (Mondo delle Tartarughe), e fornisce un gruppo di funzioni per disegnare delle linee guidando delle tartarughe per lo schermo.

Se il pacchetto Swampy è correttamente installato nel vostro sistema, potete importare TurtleWorld in questo modo:

```
from swampy.TurtleWorld import *
```

Se avete scaricato i moduli di Swampy ma non li avete installati come pacchetto, potete lavorare nella directory che contiene i file, oppure aggiungere quella directory ai percorsi di ricerca di Python. Poi potete importare TurtleWorld così:

```
from TurtleWorld import *
```

I dettagli del procedimento di installazione e dell'impostazione del percorso di ricerca dipendono dal vostro sistema, e per non aggiungere qui troppi dettagli cercherò di mantenere delle informazioni aggiornate per alcuni sistemi sul sito <http://thinkpython.com/swampy>

Create con un editor di testo un file di nome mieipoligoni.py e scriveteci il codice seguente:

```
from swampy.TurtleWorld import *
```

```
mondo = TurtleWorld()
```

```
bob = Turtle()  
print bob
```

```
wait_for_user()
```

La prima riga importa tutto il modulo `TurtleWorld` contenuto nel pacchetto `Swampy`.

La riga seguente crea un oggetto `TurtleWorld` assegnato alla variabile `mondo` e una tartaruga assegnata al nome `bob`. Se stampiamo `bob` otteniamo qualcosa del genere:

```
<TurtleWorld.Turtle instance at 0xb7bfbf4c>
```

Questo significa che `bob` fa riferimento a un'istanza di una tartaruga (`Turtle`) come definita nel modulo `TurtleWorld`. In questo contesto, "istanza" significa un elemento di un insieme; questa tartaruga di nome `bob` è una dell'insieme di tutte le possibili tartarughe.

`wait_for_user` dice a `TurtleWorld` di attendere che l'utente faccia qualcosa, sebbene in questo caso non ci sia molto che l'utente possa fare, se non chiudere la finestra.

`TurtleWorld` comprende alcune funzioni per guidare le tartarughe: `fd` e `bk` per avanti e indietro e `lt` e `rt` per girare a sinistra e a destra. Inoltre, ogni tartaruga regge una penna, che può essere appoggiata o sollevata; se la penna è appoggiata, la tartaruga lascia un segno dove passa. Le funzioni `pu` e `pd` stanno per "penna su (up)" e "penna giù (down)".

Per disegnare un angolo retto, aggiungete queste righe al programma (dopo aver creato `bob` e prima di chiamare `wait_for_user`):

```
fd(bob, 100)  
lt(bob)  
fd(bob, 100)
```

La prima riga dice a `bob` di fare 100 passi avanti. La seconda, di girare a sinistra.

Eseguendo il programma, dovrete vedere `bob` muoversi verso destra e poi su, lasciandosi dietro due segmenti.

Ora provate a modificare il programma per disegnare un quadrato. Non andate avanti finché non ci riuscite!

4.2 Ripetizione semplice

Probabilmente avete scritto qualcosa del genere (tralasciando il codice che crea `Turtleworld` e mette l'utente in attesa):

```
fd(bob, 100)  
lt(bob)
```

```
fd(bob, 100)  
lt(bob)
```

```
fd(bob, 100)  
lt(bob)
```

```
fd(bob, 100)
```

Possiamo ottenere lo stesso risultato in modo più conciso con un'istruzione `for`. Aggiungete questo esempio a `mieipoligoni.py` ed eseguitelo di nuovo:

```
for i in range(4):  
    print 'Ciao!'
```

Dovreste vedere qualcosa di simile:

```
Ciao!  
Ciao!  
Ciao!  
Ciao!
```

Questo è l'utilizzo più semplice dell'istruzione `for`; ne vedremo altri più avanti. Ma questo dovrebbe bastare per permettervi di riscrivere il vostro programma di disegno di quadrati. Proseguite solo dopo averlo fatto.

Ecco l'istruzione `for` che disegna un quadrato:

```
for i in range(4):  
    fd(bob, 100)  
    lt(bob)
```

La sintassi di un'istruzione `for` è simile a quella di una funzione. Ha un'intestazione che termina con i due punti e un corpo indentato che può contenere un numero qualunque di istruzioni.

Un'istruzione `for` è chiamata **ciclo** perché il flusso dell'esecuzione ne attraversa il corpo per poi ritornare indietro e ripeterlo da capo. In questo caso, il corpo viene eseguito per quattro volte.

Questa versione del disegno di quadrati è in realtà un pochino differente dalla precedente, in quanto provoca un'ultima svolta della tartaruga dopo aver disegnato l'ultimo lato. Ciò comporta un istante di tempo in più, ma il codice viene semplificato, inoltre lascia la tartaruga nella stessa posizione di partenza, rivolta nella direzione iniziale.

4.3 Esercizi

Quella che segue è una serie di esercizi che utilizzano TurtleWorld. Sono pensati per essere divertenti, ma hanno anche uno scopo. Mentre ci lavorate su, provate a pensare quale sia.

I paragrafi successivi contengono le soluzioni degli esercizi, per cui non guardatele finché non avete finito (o almeno provato).

1. Scrivete una funzione di nome `quadrato` che richieda un parametro di nome `t`, che è una tartaruga. La funzione deve usare la tartaruga per disegnare un quadrato.

Scrivete una chiamata alla funzione `quadrato` che passi `bob` come argomento, ed eseguite nuovamente il programma.

2. Aggiungete a `quadrato` un nuovo parametro di nome `lunghezza`. Modificate il corpo in modo che la lunghezza dei lati sia `lunghezza`, quindi modificate la chiamata alla funzione in modo da fornire un secondo argomento. Eseguite di nuovo il programma e provatelo con vari valori di `lunghezza`.

3. Le funzioni `lt` e `rt` eseguono in modo predefinito dei cambi di direzione di 90 gradi, ma è possibile passare un secondo argomento che specifichi il numero dei gradi. Per esempio, `lt(bob, 45)` ruota bob di 45 gradi a sinistra.

Fate una copia di `quadrato` e cambiate il nome in `poligono`. Aggiungete un altro parametro di nome `n` e modificate il corpo in modo che sia disegnato un poligono regolare di `n` lati. Suggerimento: gli angoli esterni di un poligono regolare di `n` lati misurano $360/n$ gradi.

4. Scrivete una funzione di nome `cerchio` che prenda come parametri una tartaruga, `t`, e un raggio, `r`, e che disegni un cerchio approssimato utilizzando `poligono` con una appropriata lunghezza e numero di lati. Provate la funzione con diversi valori di `r`.

Suggerimento: pensate alla circonferenza del cerchio e accertatevi che `lunghezza * n = circonferenza`.

Altro suggerimento: se bob è troppo lento potete accelerarlo cambiando `bob.delay`, che regola il tempo tra due movimenti, in secondi. `bob.delay = 0.01` dovrebbe dargli una mossa.

5. Create una versione più generica della funzione `cerchio` di nome `arco`, che richieda un parametro aggiuntivo `angolo`, il quale determina la porzione di cerchio da disegnare. `angolo` è espresso in gradi, quindi se `angolo=360`, `arco` dovrebbe disegnare un cerchio completo.

4.4 Incapsulamento

Il primo esercizio chiede di inserire il codice per disegnare un quadrato in una definizione di funzione, passando la tartaruga come argomento. Ecco una soluzione:

```
def quadrato(t):
    for i in range(4):
        fd(t, 100)
        lt(t)
```

```
quadrato(bob)
```

Le istruzioni più interne, `fd` e `lt` sono doppiamente indentate per significare che si trovano all'interno del ciclo `for`, che a sua volta è all'interno della funzione. L'ultima riga, `quadrato(bob)`, è a livello del margine sinistro, pertanto segna la fine sia del ciclo `for` che della definizione di funzione.

Dentro la funzione, `t` si riferisce alla stessa tartaruga a cui si riferisce `bob`, per cui `lt(t)` ha lo stesso effetto di `lt(bob)`. Ma allora perché non chiamare `bob` il parametro? Il motivo è che `t` può essere qualunque tartaruga, non solo `bob`, e in questa maniera è possibile anche creare una seconda tartaruga e passarla come parametro a `quadrato`:

```
ray = Turtle()
quadrato(ray)
```

L'inglobare un pezzo di codice in una funzione è chiamato **incapsulamento**. Uno dei benefici dell'incapsulamento è che appiccica un nome al codice, il che può servire come una sorta di documentazione. Un altro vantaggio è il riuso del codice: è più conciso chiamare una funzione due volte che copiare e incollare il corpo!

4.5 Generalizzazione

Il passo successivo è aggiungere a quadrato un parametro lunghezza. Ecco una soluzione:

```
def quadrato(t, lunghezza):
    for i in range(4):
        fd(t, lunghezza)
        lt(t)
```

```
quadrato(bob, 100)
```

L'aggiunta di un parametro a una funzione è chiamata **generalizzazione** poiché rende la funzione più generale: nella versione precedente, il quadrato aveva sempre la stessa dimensione, ora può essere grande a piacere.

Anche il passo seguente è una generalizzazione. Invece di disegnare solo quadrati, poligono disegna poligoni regolari di un qualunque numero di lati. Ecco una soluzione:

```
def poligono(t, n, lunghezza):
    angolo = 360.0 / n
    for i in range(n):
        fd(t, lunghezza)
        lt(t, angolo)
```

```
poligono(bob, 7, 70)
```

Questo codice disegna un ettagono regolare con lati di lunghezza 70. Quando in una funzione avete più di qualche argomento numerico, è facile dimenticare a cosa si riferiscono o in quale ordine sono disposti. È consentito, e a volte opportuno, includere i nomi dei parametri nell'elenco degli argomenti:

```
poligono(bob, n=7, lunghezza=70)
```

Questi sono detti **argomenti con nome** perché includono il nome del parametro a cui vengono passati, quale "parola chiave" (da non confondere con le parole chiave riservate come `while` e `def`).

Questa sintassi rende il programma più leggibile. È anche un appunto di come funzionano argomenti e parametri: quando chiamate una funzione, gli argomenti vengono assegnati a quei parametri.

4.6 Progettazione dell'interfaccia

Il prossimo passaggio è scrivere `cerchio`, che richiede come parametro il raggio, `r`. Ecco una semplice soluzione che usa `poligono` per disegnare un poligono di 50 lati:

```
def cerchio(t, r):
    circonferenza = 2 * math.pi * r
    n = 50
    lunghezza = circonferenza / n
    poligono(t, n, lunghezza)
```

La prima riga calcola la circonferenza di un cerchio di raggio `r` usando la nota formula $2\pi r$. Dato che usiamo `math.pi`, vi ricordo che dovete prima importare il modulo `math`. Per convenzione, l'istruzione `import` si scrive all'inizio dello script.

n è il numero di segmenti del nostro cerchio approssimato, e *lunghezza* è la lunghezza di ciascun segmento. Così facendo, *poligono* disegna un poligono di 50 lati che approssima un cerchio di raggio r .

Un limite di questa soluzione è che n è costante, il che comporta che per cerchi molto grandi i segmenti sono troppo lunghi, e per cerchi piccoli perdiamo tempo a disegnare minuscoli segmenti. Una soluzione sarebbe di generalizzare la funzione tramite un parametro n , dando all'utente (chiunque chiami la funzione *cerchio*) più controllo, ma rendendo così l'interfaccia meno chiara.

L'**interfaccia** è un riassunto di come è usata la funzione: quali sono i parametri? Che cosa fa la funzione? Qual è il valore restituito? Un'interfaccia è considerata "pulita" se è "più semplice possibile, ma non necessariamente più semplice. (Einstein)"

In questo esempio, r appartiene all'interfaccia perché specifica il cerchio da disegnare. n è meno pertinente perché riguarda i dettagli di *come* il cerchio viene reso.

Piuttosto di ingombrare l'interfaccia di parametri, è meglio scegliere un valore appropriato di n che dipenda da circonferenza:

```
def cerchio(t, r):
    circonferenza = 2 * math.pi * r
    n = int(circonferenza / 3) + 1
    lunghezza = circonferenza / n
    poligono(t, n, lunghezza)
```

Ora il numero di segmenti è (circa) $\text{circonferenza}/3$, e la lunghezza dei segmenti è (circa) 3, che è abbastanza piccolo da dare un cerchio di bell'aspetto, ma abbastanza grande da essere efficiente e appropriato per qualsiasi dimensione del cerchio.

4.7 Refactoring

Nello scrivere *cerchio*, ho potuto riusare *poligono* perché un poligono con molti lati è una buona approssimazione di un cerchio. Ma la funzione *arco* non è così collaborativa; non possiamo usare *poligono* o *cerchio* per disegnare un arco.

Un'alternativa è partire da una copia di *poligono* e trasformarla in *arco*. Il risultato può essere qualcosa del genere:

```
def arco(t, r, angolo):
    arco_lunghezza = 2 * math.pi * r * angolo / 360
    n = int(arco_lunghezza / 3) + 1
    passo_lunghezza = arco_lunghezza / n
    passo_angolo = float(angolo) / n

    for i in range(n):
        fd(t, passo_lunghezza)
        lt(t, passo_angolo)
```

La seconda metà di questa funzione somiglia a *poligono*, ma non possiamo riusare questa funzione senza cambiarne l'interfaccia. Potremmo generalizzare *poligono* in modo che riceva un angolo come terzo argomento, ma allora *poligono* non sarebbe più un nome appropriato! Invece, creiamo una funzione più generale chiamata *polilinea*:

```
def polilinea(t, n, lunghezza, angolo):
    for i in range(n):
        fd(t, lunghezza)
        lt(t, angolo)
```

Ora possiamo riscrivere poligono e arco in modo che usino polilinea:

```
def poligono(t, n, lunghezza):
    angolo = 360.0 / n
    polilinea(t, n, lunghezza, angolo)

def arco(t, r, angolo):
    arco_lunghezza = 2 * math.pi * r * angolo / 360
    n = int(arco_lunghezza / 3) + 1
    passo_lunghezza = arco_lunghezza / n
    passo_angolo = float(angolo) / n
    polilinea(t, n, passo_lunghezza, passo_angolo)
```

Infine, riscriviamo cerchio in modo che usi arco:

```
def cerchio(t, r):
    arco(t, r, 360)
```

Questo procedimento di riarrangiare una programma per migliorare le interfacce delle funzioni e facilitare il riuso del codice, è chiamato **refactoring**. In questo caso, abbiamo notato che in arco e in poligono c’era del codice simile, allora abbiamo semplificato il tutto in polilinea.

Avendoci pensato prima, avremmo potuto scrivere polilinea direttamente, evitando il refactoring, ma spesso all’inizio di un lavoro non si hanno le idee abbastanza chiare per progettare al meglio tutte le interfacce. Una volta cominciato a scrivere il codice, si colgono meglio i problemi. A volte il refactoring è segno che avete imparato qualcosa.

4.8 Tecnica di sviluppo

Una **tecnica di sviluppo** è una procedura di scrittura dei programmi. Quello che abbiamo usato in questa esercitazione si chiama “incapsulamento e generalizzazione”. I passi del procedimento sono:

1. Iniziare scrivendo un piccolo programma senza definire funzioni.
2. Una volta ottenuto un programma funzionante, incapsularlo in una funzione e dargli un nome.
3. Generalizzare la funzione aggiungendo i parametri appropriati.
4. Ripetere i passi da 1 a 3 fino ad avere un insieme di funzioni. Copiate e incollate il codice funzionante per evitare di riscriverlo (e ricorreggerlo).
5. Cercare le occasioni per migliorare il programma con il refactoring. Ad esempio, se avete del codice simile in più punti, valutate di semplificare rielaborandolo in una funzione più generale.

Questa procedura ha alcuni inconvenienti—vedremo più avanti alcune alternative—ma può essere di aiuto se in principio non sapete bene come suddividere il vostro programma in funzioni. È un approccio che vi permette di progettare man mano che andate avanti.

4.9 Stringa di documentazione

Una **stringa di documentazione**, o *docstring*, è una stringa posta all'inizio di una funzione che ne illustra l'interfaccia. Ecco un esempio:

```
def polilinea(t, n, lunghezza, angolo):  
    """Disegna n segmenti di data lunghezza e angolo  
       (in gradi) tra di loro. t e' una tartaruga.  
    """  
    for i in range(n):  
        fd(t, lunghezza)  
        lt(t, angolo)
```

Questa stringa ha triple virgolette, che le consentono di essere divisa su più righe (stringa a righe multiple).

È breve, ma contiene le informazioni essenziali di cui qualcuno potrebbe aver bisogno per usare la funzione. Spiega in modo conciso cosa fa la funzione (senza entrare nei dettagli di come lo fa). Spiega che effetti ha ciascun parametro sul comportamento della funzione e di che tipo devono essere i parametri stessi (se non è ovvio).

Scrivere questo tipo di documentazione è una parte importante della progettazione dell'interfaccia. Un'interfaccia ben studiata dovrebbe essere semplice da spiegare; se fate fatica a spiegare una delle vostre funzioni, può essere segno che l'interfaccia dovrebbe essere migliorata.

4.10 Debug

Un'interfaccia è simile ad un contratto tra la funzione e il suo chiamante. Il chiamante si impegna a fornire certi parametri e la funzione si impegna a svolgere un dato lavoro.

Ad esempio, a `polilinea` devono essere passati quattro argomenti: `t` deve essere una tartaruga; `n` il numero dei segmenti, quindi un numero intero; `lunghezza` deve essere un numero positivo; e `angolo` un numero che si intende espresso in gradi.

Questi requisiti sono detti **precondizioni** perché si suppone siano verificati prima che la funzione sia eseguita. Per contro, le condizioni che si devono verificare al termine della funzione sono dette **postcondizioni**, e comprendono l'effetto che deve avere la funzione (come il disegnare segmenti) e ogni altro effetto minore (come muovere la tartaruga o fare altri cambiamenti nel suo Mondo).

Le precondizioni sono responsabilità del chiamante. Se questi viola una precondizione (documentata in modo appropriato!) e la funzione non fa correttamente ciò che deve, l'errore sta nel chiamante e non nella funzione.

4.11 Glossario

istanza: Un elemento di un insieme. Il Mondo delle Tartarughe di questo capitolo è un elemento dei possibili Mondi delle Tartarughe.

ciclo: Una porzione di programma che può essere eseguita ripetutamente.

incapsulamento: Il procedimento di trasformare una sequenza di istruzioni in una funzione.

generalizzazione: Il procedimento di sostituire qualcosa di inutilmente specifico (come un numero) con qualcosa di più generale ed appropriato (come una variabile o un parametro).

argomento con nome: Un argomento che include il nome del parametro a cui è destinato come “parola chiave”.

interfaccia: Una descrizione di come usare una funzione, incluso il nome, la descrizione degli argomenti e il valore di ritorno.

refactoring: Il procedimento di modifica di un programma funzionante per migliorare le interfacce delle funzioni e altre qualità del codice.

tecnica di sviluppo: Procedura di scrittura dei programmi.

stringa di documentazione o docstring: Una stringa che compare in una definizione di una funzione per documentarne l’interfaccia.

precondizione: Un requisito che deve essere soddisfatto dal chiamante prima di eseguire una funzione.

postcondizione: Un requisito che deve essere soddisfatto dalla funzione prima di terminare.

4.12 Esercizi

Esercizio 4.1. Scaricate il codice in questo capitolo dal sito <http://thinkpython.com/code/polygon.py>.

1. Scrivete delle appropriate stringhe di documentazione per poligono, arco e cerchio.
2. Disegnate un diagramma di stack che illustri lo stato del programma mentre esegue `cerchio(bob, raggio)`. Potete fare i conti a mano o aggiungere istruzioni `print` al codice.
3. La versione di arco nel Paragrafo 4.7 non è molto accurata, perché l’approssimazione lineare del cerchio è sempre esterna al cerchio vero. Ne deriva che la tartaruga finisce ad alcune unità di distanza dal traguardo corretto. La mia soluzione mostra un modo per ridurre questo errore. Leggete il codice e cercate di capirlo. Disegnare un diagramma può aiutarvi a comprendere il funzionamento.

Esercizio 4.2. Scrivete un insieme di funzioni, generali in modo appropriato, che disegni dei fiori stilizzati come in Figura 4.1.

Soluzione: <http://thinkpython.com/code/flower.py>, richiede anche <http://thinkpython.com/code/polygon.py>.

Esercizio 4.3. Scrivete un insieme di funzioni, generali in modo appropriato, che disegni delle forme a torta come in Figura 4.2.

Soluzione: <http://thinkpython.com/code/pie.py>.

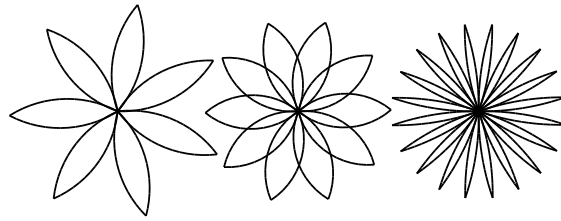


Figura 4.1: Fiori delle tartarughe.

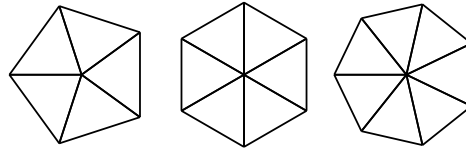


Figura 4.2: Torte delle tartarughe.

Esercizio 4.4. Le lettere dell'alfabeto possono essere costruite con un moderato numero di elementi di base, come linee orizzontali e verticali e alcune curve. Progettate un carattere che possa essere disegnato con un numero minimo di elementi di base e poi scrivete delle funzioni che disegnano le lettere dell'alfabeto.

Dovreste scrivere una funzione per ogni lettera, con nomi tipo `disegna_a`, `disegna_b`, ecc., e inserirle in un file di nome `lettere.py`. Potete scaricare una “macchina da scrivere a tartaruga” da <http://thinkpython.com/code/typewriter.py> per aiutarvi a provare il vostro codice.

Soluzione: <http://thinkpython.com/code/letters.py>, richiede anche <http://thinkpython.com/code/polygon.py>.

Esercizio 4.5. Documentatevi sulle spirali sul sito <http://it.wikipedia.org/wiki/Spirale>; quindi scrivete un programma che disegni una spirale di Archimede (o di qualche altro tipo). Soluzione: <http://thinkpython.com/code/spiral.py>.

Capitolo 5

Istruzioni condizionali e ricorsione

5.1 L'operatore modulo

L'**operatore modulo** si applica a due numeri interi e fornisce il resto della divisione del primo operando 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
```

Pertanto 7 diviso 3 fa 2, con il resto di 1.

L'operatore modulo può rivelarsi sorprendentemente utile. Per esempio, permette di controllare se un numero è divisibile per un altro—se $x \% y$ è zero, significa che 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 del numero x (in base 10). Allo stesso modo, $x \% 100$ restituisce le ultime due cifre.

5.2 Espressioni booleane

Un'**espressione booleana** è un'espressione che può essere o vera o falsa. Gli esempi che seguono usano l'operatore `==`, confrontano due valori e restituiscono `True` (vero) se sono uguali, `False` (falso) altrimenti:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` e `False` sono valori speciali che sono di tipo `bool`; non sono delle stringhe:

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

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

<code>x != y</code>	# x e' diverso da y
<code>x > y</code>	# x e' maggiore di y
<code>x < y</code>	# x e' minore di y
<code>x >= y</code>	# x e' maggiore o uguale a y
<code>x <= y</code>	# x e' minore o uguale a y

Queste operazioni vi saranno familiari, tuttavia i simboli in Python sono diversi da quelli usati comunemente in matematica. Un errore frequente è quello di usare il simbolo di uguale(=) invece del doppio uguale(==). Ricordate che = è un operatore di assegnazione, mentre == è un operatore di confronto. Inoltre in Python non esistono simboli del tipo `=<` o `=>`.

5.3 Operatori logici

Ci sono tre **operatori logici**: `and`, `or`, e `not`. Il significato di questi operatori è simile al loro significato comune (e, o, non): per esempio, l'espressione `x > 0 and x < 10` è vera solo se `x` è più grande di 0 e più piccolo di 10.

L'espressione `n%2 == 0 or n%3 == 0` invece 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, `not (x > y)` è vera se `x > y` è falsa, cioè se `x` è minore o uguale a `y`.

In senso stretto, gli operandi degli operatori logici dovrebbero essere delle espressioni booleane, ma da questo punto di vista Python non è troppo fiscale: infatti ogni numero diverso da zero viene considerato "vero", e lo zero è considerato "falso".

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

Questa flessibilità può essere utile, ma ci sono alcune sottigliezze che potrebbero confondere. È preferibile evitarla (a meno che non sappiate quello che state facendo).

5.4 Esecuzione condizionale

Per poter scrivere programmi utili, abbiamo quasi sempre la necessità di valutare delle condizioni e di cambiare il comportamento del programma a seconda del risultato della valutazione. Le **istruzioni condizionali** ci offrono questa possibilità. La forma più semplice è l'istruzione `if` ("se" in inglese):

```
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 nulla.

Come nel caso della definizione di funzione, la struttura dell'istruzione `if` è costituita da un'intestazione seguita da un corpo indentato. Le istruzioni come questa vengono chiamate **istruzioni composte**.

Non c'è limite al numero di istruzioni che possono comparire nel corpo, 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 provare il programma. In questo caso si può usare l'istruzione `pass`, che serve solo da segnaposto temporaneo ed è innocua:

```
if x < 0:
    pass          # scrivere cosa fare con i valori negativi!
```

5.5 Esecuzione alternativa

Una seconda forma di istruzione `if` è l'**esecuzione alternativa**, nella quale ci sono due azioni possibili, e il valore della condizione determina quale delle due debba essere eseguita e quale no. La sintassi è:

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

Se il resto della divisione di `x` per 2 è zero significa che `x` è un numero pari, e il programma mostra il messaggio corrispondente. Se invece la condizione è falsa, viene eseguita la serie di istruzioni descritta dopo la riga `else` (che in inglese significa "altrimenti"). In ogni caso, una delle due alternative sarà sempre eseguita. Le due alternative sono chiamate **ramificazioni**, perché rappresentano dei bivi nel flusso di esecuzione del programma.

5.6 Condizioni in serie

Talvolta occorre tenere conto di più di due possibili sviluppi, pertanto possiamo aver bisogno di più di due ramificazioni. Un modo per esprimere questo tipo di calcolo 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 dell'esito del confronto tra `x` e `y`. Non c'è alcun limite al numero di istruzioni `elif`. Se esiste una clausola `else`, deve essere scritta per ultima, ma non è obbligatoria.

```
if scelta == 'a':
    disegna_a()
elif scelta == 'b':
    disegna_b()
elif scelta == 'c':
    disegna_c()
```

Le condizioni sono controllate nell'ordine in cui sono state scritte: se la prima è falsa viene controllata la seconda e così via. Non appena una condizione risulta vera, viene eseguito il ramo corrispondente e l'intera istruzione `if` si conclude. In ogni caso, anche se risultassero vere altre condizioni successive, dopo l'esecuzione della prima queste verranno trascurate.

5.7 Condizioni nidificate

Un'espressione condizionale può anche essere inserita nel corpo di un'altra espressione condizionale. Possiamo scrivere il precedente esempio a tre scelte anche in questo modo:

```
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 esterna (`if x == y`) contiene due rami: il primo contiene un'istruzione semplice, il secondo un'altra istruzione `if` che a sua volta prevede un'ulteriore ramificazione. Entrambi i rami del secondo `if` sono istruzioni di stampa, ma potrebbero anche contenere a loro volta ulteriori istruzioni condizionali.

Sebbene l'indentazione delle istruzioni aiuti a rendere evidente la struttura, le **condizioni nidificate** diventano rapidamente difficili da leggere, quindi è meglio usarle con moderazione.

Qualche volta, gli operatori logici permettono di semplificare le espressioni condizionali nidificate:

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

L'istruzione `print` è eseguita solo se entrambe le condizioni si verificano. Possiamo allora usare l'operatore booleano `and` per combinarle:

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

5.8 Ricorsione

Abbiamo visto che è del tutto normale che una funzione ne chiami un'altra, ma è anche consentito ad una funzione di chiamare se stessa. L'utilità può non essere immediatamente evidente, ma questa è una delle cose più magiche che un programma possa fare. Per fare un esempio, diamo un'occhiata a questa funzione:

```
def contoallaroveschia(n):  
    if n <= 0:  
        print 'Via!'  
    else:  
        print n  
        contoallaroveschia(n-1)
```

Se n vale 0 o è negativo, scrive la parola "Via!". Altrimenti scrive il numero n e poi chiama la funzione `contoallaroveschia` (cioè se stessa) passando un argomento che vale $n-1$.

Cosa succede quando chiamiamo la funzione in questo modo?

```
>>> contoallaroveschia(3)
```

L'esecuzione di `contoallaroveschia` inizia da $n=3$, e dato che n è maggiore di 0, stampa il valore 3, poi chiama se stessa...

L'esecuzione di `contoallaroveschia` inizia da $n=2$, e dato che n è maggiore di 0, stampa il valore 2, poi chiama se stessa...

L'esecuzione di `contoallaroveschia` inizia da $n=1$, e dato che n è maggiore di 0, stampa il valore 1, poi chiama se stessa...

L'esecuzione di `contoallaroveschia` inizia da $n=0$, e dato che n è uguale a 0, stampa la parola "Via!" e poi ritorna.

La funzione `contoallaroveschia` che aveva dato $n=1$ ritorna.

La funzione `contoallaroveschia` che aveva dato $n=2$ ritorna.

La funzione `contoallaroveschia` che aveva dato $n=3$ ritorna.

E infine ritorniamo in `__main__`. Il risultato finale è questo:

```
3  
2  
1  
Via!
```

Una funzione che chiama se stessa si dice **ricorsiva** e la procedura è detta **ricorsione**.

Come secondo esempio, scriviamo una funzione che stampi una data stringa per n volte.

```
def stampa_n(s, n):  
    if n <= 0:  
        return  
    print s  
    stampa_n(s, n-1)
```

Se $n \leq 0$ l'istruzione `return` provoca l'uscita dalla funzione. Il flusso dell'esecuzione torna immediatamente al chiamante, e le righe rimanenti della funzione non vengono eseguite.

Il resto della funzione è simile a `contoallaroveschia`: se n è maggiore di zero, visualizza la stringa s e chiama se stessa per $n - 1$ altre volte. Il numero di righe risultanti sarà $1 + (n - 1)$, che corrisponde a n .

Per esempi semplici come questi, è forse più facile usare un ciclo `for`. Vedremo però più avanti degli esempi difficili da scrivere con un ciclo `for` ma facili con la ricorsione; meglio quindi cominciare subito a prendere mano.

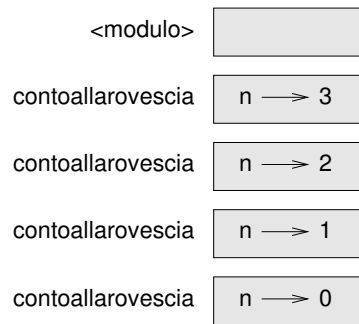


Figura 5.1: Diagramma di stack.

5.9 Diagrammi di stack delle funzioni ricorsive

Nel Paragrafo 3.10, abbiamo usato un diagramma di stack per rappresentare lo stato di un programma durante una chiamata di funzione. Lo stesso tipo di diagramma può servire 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 contemporaneamente più frame riguardanti una stessa funzione.

La Figura 5.1 mostra il diagramma di stack della funzione `contoallarovescia` chiamata con `n = 3`.

Come al solito, il livello superiore dello stack è il frame di `__main__`. Questo frame è vuoto, perché in questo caso non vi abbiamo creato alcuna variabile né abbiamo passato alcun parametro.

I quattro frame di `contoallarovescia` hanno valori diversi del parametro `n`. Il livello inferiore dello stack, dove `n=0`, è chiamato **caso base**. Esso non effettua ulteriori chiamate ricorsive, quindi non ci sono ulteriori frame.

Esercizio 5.1. *Disegnate il diagramma di stack della funzione `stampa_n` chiamata con `s='Ciao'` e `n=2`.*

Esercizio 5.2. *Scrivete una funzione di nome `fai_n` che accetti come argomenti un oggetto funzione e un numero `n`, e che chiami per `n` volte la funzione data.*

5.10 Ricorsione infinita

Se una ricorsione non raggiunge mai un caso 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 di solito non è considerata una buona cosa. Questo è un programma minimo che genera una ricorsione infinita:

```
def ricorsiva():
    ricorsiva()
```


Nella maggior parte degli ambienti, un programma con una ricorsione infinita non viene eseguito davvero all'infinito. Python stampa un messaggio di errore quando è stato raggiunto il massimo livello di ricorsione possibile:

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

Questo traceback è un po' più lungo di quello che abbiamo visto nel capitolo precedente. Quando si verifica l'errore, nello stack ci sono oltre 1000 frame di chiamata di ricorsiva!

5.11 Input da tastiera

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

Python 2 comprende una funzione predefinita chiamata `raw_input` che permette di inserire dati da tastiera. In Python 3, la funzione si chiama `input`. Quando questa funzione viene chiamata, il programma si ferma ed attende che l'operatore scriva qualcosa e confermi poi l'inserimento premendo il tasto Invio o Enter. A quel punto il programma riprende e `raw_input` restituisce ciò che l'operatore ha inserito sotto forma di stringa:

```
>>> testo = raw_input()
Cosa stai aspettando?
>>> print testo
Cosa stai aspettando?
```

Prima di chiamare la funzione, è buona norma stampare un messaggio che informa l'utente di ciò che deve inserire. Questo messaggio è chiamato *prompt*, e può essere passato come argomento a `raw_input`:

```
>>> nome = raw_input('Come ti chiami?\n')
Come ti chiami?
Artu', Re dei Bretoni!
>>> print nome
Artu', Re dei Bretoni!
```

La sequenza `\n` alla fine del prompt rappresenta un **ritorno a capo**, un carattere speciale che provoca un'interruzione di riga. Ecco perché l'input dell'utente compare sulla riga successiva sotto al prompt.

Se il valore da inserire è un intero possiamo provare a convertire il valore inserito in `int`:

```
>>> prompt = "Qual e' la velocita' in volo di una rondine?\n"
>>> velocita = raw_input(prompt)
Qual e' la velocita' in volo di una rondine?
17
>>> int(velocita)
17
```

Ma se la stringa inserita contiene qualcosa di diverso da dei numeri, si verifica un errore:

```
>>> velocita = raw_input(prompt)
Qual e' la velocita' in volo di una rondine?
Cosa intendi, una rondine europea o africana?
>>> int(velocita)
ValueError: invalid literal for int() with base 10
```

Vedremo più avanti come trattare questo tipo di errori.

5.12 Debug

Il traceback che Python mostra quando si verifica un errore contiene molte informazioni, ma può essere sovrabbondante quando ci sono molti frame nello stack. Di solito le parti più utili sono:

- Che tipo di errore era, e
- Dove si è verificato.

Gli errori di sintassi di solito sono facili da trovare, tranne qualcuno. Gli spaziatori possono essere insidiosi, perché spazi e tabulazioni non sono visibili e siamo abituati a non tenerne conto.

```
>>> x = 5
>>> y = 6
File "<stdin>", line 1
    y = 6
    ^
```

IndentationError: unexpected indent

In questo esempio, il problema è che la seconda riga è indentata di uno spazio. Ma il messaggio di errore punta su `y`, portando fuori strada. In genere, i messaggi di errore indicano dove il problema è venuto a galla, ma l'errore vero potrebbe essere in un punto precedente del codice, a volte anche nella riga precedente.

Lo stesso vale per gli errori di runtime.

Supponiamo di voler calcolare un rapporto segnale/rumore in decibel. La formula è $SNR_{db} = 10 \log_{10}(P_{segnale} / P_{rumore})$. In Python si può scrivere:

```
import math
potenza_segnale = 9
potenza_rumore = 10
rapporto = potenza_segnale / potenza_rumore
decibel = 10 * math.log10(rapporto)
print decibel
```

Ma se lo eseguite in Python 2, compare un messaggio di errore.

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibel = 10 * math.log10(rapporto)
ValueError: math domain error
```

Il messaggio punta alla riga 5, ma lì non c'è niente di sbagliato. Per trovare il vero errore, può essere utile stampare il valore di rapporto, che risulta essere 0. Il problema è nella riga 4, perché la divisione di due interi dà una divisione intera. La soluzione è rappresentare i valori del segnale e del rumore in numeri decimali a virgola mobile (9.0 e 10.0).

In genere, i messaggi di errore dicono dove è emerso il problema, ma spesso la causa è altrove.

In Python 3, questo esempio funziona senza causare errori: infatti, nella nuova versione l'operatore di divisione esegue una divisione decimale anche con operandi interi.

5.13 Glossario

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

espressione booleana: Espressione il cui valore è o vero (True) o falso (False).

operatore di confronto: Un operatore che confronta due valori detti operandi: ==, !=, >, <, >=, e <=.

operatore logico: Un operatore che combina due espressioni booleane: and, or, e not.

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

condizione: Espressione booleana in un'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.

ramificazione: Una delle sequenze di istruzioni alternative scritte in una istruzione condizionale.

condizioni in serie: Istruzione condizionale con una serie di ramificazioni alternative.

condizione nidificata o annidata: Un'istruzione condizionale inserita in una ramificazione di un'altra istruzione condizionale.

ricorsione: Chiamata della stessa funzione attualmente in esecuzione.

caso base: Ramificazione di un'istruzione condizionale, posta in una funzione ricorsiva, che non esegue a sua volta una chiamata ricorsiva.

ricorsione infinita: Una ricorsione priva di un caso base, oppure che non lo raggiunge mai. Nell'evenienza, causa un errore in esecuzione.

5.14 Esercizi

Esercizio 5.3. L'ultimo teorema di Fermat afferma che non esistono interi positivi a , b , e c tali che

$$a^n + b^n = c^n$$

per qualsiasi valore di n maggiore di 2.

1. Scrivete una funzione di nome `verifica_fermat` che richieda quattro parametri— a , b , c e n —e controlli se il teorema regge. Se n è maggiore di 2 e capitasse che

$$a^n + b^n = c^n$$

il programma deve visualizzare: "Santi Numi, Fermat si è sbagliato!", altrimenti: "No, questo non è vero."

2. Scrivete una funzione che chieda all'utente di inserire valori di a , b , c e n , li converta in interi e usi `verifica_fermat` per controllare se violano il teorema di Fermat.

Esercizio 5.4. Dati tre bastoncini, può essere possibile o meno riuscire a sistamarli in modo da formare un triangolo. Per esempio, se uno dei bastoncini misura 12 centimetri e gli altri due 1 centimetro, è chiaro che non riuscireste a far toccare le estremità di tutti e tre i bastoncini. Date tre lunghezze, c'è una semplice regola per controllare se è possibile formare un triangolo:

Se una qualsiasi delle tre lunghezze è maggiore della somma delle altre due, non potete formare un triangolo. (Se la somma di due lunghezze è uguale alla terza, si ha un triangolo "degenere".)

1. Scrivete una funzione di nome `triangolo` che riceva tre interi come argomenti e che mostri "Si" o "No", a seconda che si possa o meno formare un triangolo con dei bastoncini delle tre lunghezze date.
2. Scrivete una funzione che chieda all'utente di inserire tre lunghezze, le converta in interi, e le passi a `triangolo` per verificare se si possa o meno formare un triangolo.

Gli esercizi seguenti utilizzano TurtleWorld del Capitolo 4:

Esercizio 5.5. Leggete la seguente funzione e cercate di capire cosa fa. Quindi eseguirla (vedere gli esempi nel Capitolo 4).

```
def disegna(t, lunghezza, n):
    if n == 0:
        return
    angolo = 50
    fd(t, lunghezza*n)
    lt(t, angolo)
    disegna(t, lunghezza, n-1)
    rt(t, 2*angolo)
    disegna(t, lunghezza, n-1)
    lt(t, angolo)
    bk(t, lunghezza*n)
```

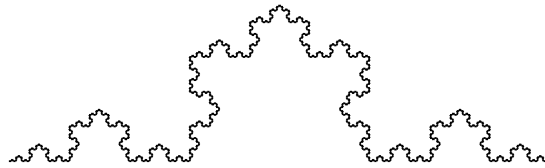


Figura 5.2: Una curva di Koch.

Esercizio 5.6. *La curva di Koch è un frattale che somiglia a quello in Figura 5.2. Per disegnare una curva di Koch di lunghezza x , dovete:*

1. *Disegnare una curva di Koch di lunghezza $x/3$.*
2. *Girare a sinistra di 60 gradi.*
3. *Disegnare una curva di Koch di lunghezza $x/3$.*
4. *Girare a destra di 120 gradi.*
5. *Disegnare una curva di Koch di lunghezza $x/3$.*
6. *Girare a sinistra di 60 gradi.*
7. *Disegnare una curva di Koch di lunghezza $x/3$.*

Ad eccezione di quando x è minore di 3: in questo caso si disegna una linea dritta lunga x .

1. *Scrivete una funzione di nome `koch` che preveda una tartaruga e una lunghezza come parametri, e che usi la tartaruga per disegnare una curva di Koch della data lunghezza.*
2. *Scrivete una funzione chiamata `fioccodineve` che disegni tre curve di Koch per ottenere il contorno di un fiocco di neve.*

Soluzione: <http://thinkpython.com/code/koch.py>.

3. *La curva di Koch può essere generalizzata in alcuni modi. Consultate http://it.wikipedia.org/wiki/Curva_di_Koch per degli esempi e implementate quello che preferite.*

Capitolo 6

Funzioni produttive

6.1 Valori di ritorno

Alcune delle funzioni predefinite che abbiamo usato finora, come quelle matematiche, producono dei risultati: la chiamata della funzione genera un nuovo valore, che di solito viene associato ad una variabile o viene usato come parte di un'espressione.

```
e = math.exp(1.0)
altezza = raggio * math.sin(radiani)
```

Tutte le funzioni che abbiamo scritto finora sono “vuote”: visualizzano qualcosa o muovono tartarughe, ma restituiscono come valore `None`.

In questo capitolo scriveremo finalmente delle funzioni che restituiscono un valore e che chiameremo funzioni “produttive”. Il primo esempio è `area`, che fornisce l'area di un cerchio di dato raggio:

```
def area(raggio):
    temp = math.pi * raggio**2
    return temp
```

Abbiamo già visto l'istruzione `return`, ma nel caso di una funzione produttiva questa istruzione include un'espressione. Il suo significato è: “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, e allora l'esempio precedente può essere riscritto in modo più compatto:

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

Dato che queste istruzioni `return` si trovano in rami diversi di una condizione alternativa, solo una di esse verrà effettivamente eseguita.

Non appena viene eseguita un'istruzione `return`, la funzione termina senza eseguire ulteriori istruzioni. Il codice che viene a trovarsi dopo l'istruzione `return` o in ogni altro punto che non può essere raggiunto dal flusso di esecuzione, è denominato **codice morto**.

In una funzione produttiva è bene assicurarsi che ogni possibile flusso di esecuzione del programma porti ad un'uscita dalla funzione con un'istruzione `return`. Per esempio:

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

Questa funzione non è corretta, in quanto se x è uguale a 0, nessuna delle due condizioni è vera e la funzione termina senza incontrare un'istruzione `return`. Se il flusso di esecuzione arriva alla fine della funzione, il valore di ritorno è `None`, che non è certo il valore assoluto di 0.

```
>>> print valore_assoluto(0)  
None
```

A proposito: Python comprende già la funzione `abs` che calcola il valore assoluto.

Esercizio 6.1. *Scrivete una funzione di nome `compares` che restituisce 1 se $x > y$, 0 se $x == y$, e -1 se $x < y$.*

6.2 Sviluppo incrementale

A mano a mano che scriverete funzioni di complessità maggiore, vi troverete a impiegare più tempo per il debug.

Per fare fronte a programmi via via più complessi, 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 esempio, supponiamo che vogliate trovare la distanza tra due punti, note le coordinate (x_1, y_1) e (x_2, y_2) . Per il teorema di Pitagora, la distanza è

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

La prima cosa da considerare è l'aspetto che la funzione `distanza` deve avere in Python, chiarendo subito quali siano i parametri che deve avere la funzione e quale sia il valore di ritorno da ottenere.

Nel nostro caso i dati di partenza (o di *input*) sono i due punti, rappresentabili attraverso le loro coordinate (due coppie di numeri); il risultato (o *output*) è la distanza, che è un valore decimale.

Possiamo così scrivere un primo abbozzo di funzione:


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

Ovviamente questa prima versione non calcola ancora la distanza, ma restituisce sempre 0. Però è già una funzione sintatticamente corretta e può essere eseguita: potete quindi provarla prima di procedere a renderla più complessa.

Proviamo allora la nuova funzione, chiamandola con dei valori di esempio:

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

Ho scelto questi valori in modo che la loro distanza orizzontale sia 3 e quella verticale 4. In tal modo, il risultato è pari a 5 (è l'ipotenusa di un triangolo rettangolo i cui cateti sono lunghi 3 e 4). Quando collaudiamo una funzione è sempre utile sapere prima il risultato.

A questo punto, abbiamo verificato che la funzione è sintatticamente corretta e possiamo cominciare ad aggiungere righe di codice nel corpo. Un passo successivo plausibile è quello di calcolare le differenze $x_2 - x_1$ e $y_2 - y_1$. Memorizzeremo queste differenze in variabili temporanee che chiameremo dx e dy , e le mostreremo a video.

```
def distanza(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    print "dx e' ", dx  
    print "dy e' ", dy  
    return 0.0
```

Se la funzione è giusta, usando i valori di prima dovrebbe mostrare 'dx e' 3' e 'dy e' 4'. Se i risultati coincidono, siamo sicuri che la funzione riceve correttamente i parametri ed elabora altrettanto correttamente i primi calcoli. Altrimenti, dovremo controllare solo poche righe.

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

```
def distanza(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    dsquadr = dx**2 + dy**2  
    print "dsquadr e': ", dsquadr  
    return 0.0
```

Di nuovo, eseguite il programma in questa fase e controllate il risultato, che nel nostro caso dovrebbe essere 25. Infine, usate la funzione radice quadrata `math.sqrt` per calcolare e restituire il risultato:

```
def distanza(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    dsquadr = dx**2 + dy**2  
    risultato = math.sqrt(dsquadr)  
    return risultato
```

Se tutto funziona, avete finito. Altrimenti, potete stampare per verifica il valore di `risultato` prima dell'istruzione `return`.

La versione definitiva della funzione non mostra nulla quando viene eseguita; restituisce solo un valore. Le istruzioni `print` che avevamo inserito erano utili per il debug, ma una

volta verificato che tutto funziona vanno rimosse. Pezzi di codice temporanei come questi sono detti **impalcature**, perché sono di aiuto nella fase di costruzione del programma ma non fanno parte del prodotto finale.

Soprattutto agli inizi, non dovrete mai aggiungere più di qualche riga di codice alla volta. Con l'esperienza, potrete scrivere e fare il debug di pezzi di codice sempre più corposi. In ogni caso, nelle prime fasi il processo di sviluppo incrementale potrà farvi risparmiare un bel po' di tempo di debug.

Ecco i punti chiave di questa procedura:

1. Iniziate con un programma funzionante e fate piccoli cambiamenti: questo permetterà di scoprire facilmente dove sono localizzati gli eventuali errori.
2. Usate variabili temporanee per memorizzare i valori intermedi, così da poterli stampare e controllare.
3. Quando il programma funziona perfettamente, rimuovete le istruzioni temporanee e consolidate le istruzioni multiple in espressioni composte, sempre che questo non renda il programma troppo difficile da leggere.

Esercizio 6.2. *Usate lo sviluppo incrementale per scrivere una funzione chiamata `ipotenusa`, che restituisca la lunghezza dell'ipotenusa di un triangolo rettangolo, dati i due cateti come parametri. Registrare ogni passo del processo di sviluppo man mano che procedete.*

6.3 Composizione

Come potete ormai immaginare, è possibile chiamare una funzione dall'interno di un'altra funzione. Questa proprietà è chiamata **composizione**.

Scriveremo come esempio una funzione che prende due punti geometrici, il centro di un cerchio ed un punto sulla sua circonferenza, e calcola l'area del cerchio.

Supponiamo che le coordinate del centro del cerchio siano memorizzate nelle variabili `xc` e `yc`, e quelle del punto sulla circonferenza in `xp` e `yp`. Innanzitutto, bisogna trovare il raggio del cerchio, che è pari alla distanza tra i due punti. La funzione `distanza` che abbiamo appena scritto, ci torna utile:

```
raggio = distanza(xc, yc, xp, yp)
```

Il secondo passo è trovare l'area del cerchio di quel raggio; anche questa funzione l'abbiamo già scritta:

```
risultato = area(raggio)
```

Incapsulando il tutto in una funzione otteniamo:

```
def area_cerchio(xc, yc, xp, yp):
    raggio = distanza(xc, yc, xp, yp)
    risultato = area(raggio)
    return risultato
```

Le variabili temporanee `raggio` e `risultato` sono utili per lo sviluppo e il debug ma, una volta constatato che il programma funziona, possiamo riscrivere la funzione in modo più conciso componendo le chiamate alle funzioni:

```
def area_cerchio(xc, yc, xp, yp):
    return area(distanza(xc, yc, xp, yp))
```

6.4 Funzioni booleane

Le funzioni possono anche restituire valori booleani (vero o falso), cosa che è spesso utile per includere al loro interno dei test anche complessi. Per esempio:

```
def divisibile(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

È prassi assegnare come nomi alle funzioni booleane dei predicati che, con accezione interrogativa, attendono una risposta sì/no; `divisibile` restituisce `True` o `False` per rispondere alla domanda se è vero o no che `x` è divisibile per `y`.

Facciamo un esempio:

```
>>> divisibile(6, 4)
False
>>> divisibile(6, 3)
True
```

Possiamo scrivere la funzione in modo ancora più conciso, visto che il risultato dell'operatore di confronto `==` è anch'esso un booleano, restituendolo direttamente:

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

Le funzioni booleane sono usate spesso nelle istruzioni condizionali:

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

Potreste pensare di scrivere in questo modo:

```
if divisibile(x, y) == True:
    print "x e' divisibile per y"
```

ma il confronto supplementare è superfluo.

Esercizio 6.3. *Scrivete una funzione `compreso_tra(x, y, z)` che restituisca `True` se $x \leq y \leq z$ o `False` altrimenti.*

6.5 Altro sulla ricorsione

Abbiamo trattato solo una piccola parte di Python, ma è interessante sapere che questo sottoinsieme è già di per sé un linguaggio di programmazione *completo*: questo significa che con gli elementi che già conoscete potete esprimere qualsiasi tipo di elaborazione. Qualsiasi tipo di programma esistente potrebbe essere scritto usando solo le caratteristiche del linguaggio che avete appreso finora (aggiungendo solo alcuni comandi di controllo per gestire dispositivi come tastiera, mouse, dischi, ecc.)

La prova di questa affermazione è un esercizio tutt'altro che banale affrontato per la prima volta da Alan Turing, uno dei pionieri dell'informatica (qualcuno potrebbe obiettare che in realtà era un matematico, ma molti dei primi informatici erano dei matematici). Di conseguenza la dimostrazione è chiamata Tesi di Turing. Per una trattazione più completa

(ed accurata) della Tesi di Turing, consiglio il libro di Michael Sipser, *Introduction to the Theory of Computation*.

Per darvi un'idea di cosa potete fare con gli strumenti imparati finora, proveremo a valutare delle 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 si sta definendo. Una vera definizione circolare non è propriamente utile:

vorpale: aggettivo usato per descrivere qualcosa di vorpale.

Sarebbe fastidioso trovare una definizione del genere in un vocabolario. D'altra parte, considerate la definizione della funzione matematica fattoriale (indicata da un numero seguito da un punto esclamativo, !), cioè:

$$\begin{aligned} 0! &= 1 \\ n! &= n(n-1)! \end{aligned}$$

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

Pertanto, $3!$ 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. Riassumendo il tutto, $3!$ è uguale a 3 per 2 per 1 per 1, che fa 6.

Se potete scrivere una definizione ricorsiva di qualcosa, di solito potete anche scrivere un programma Python per valutarla. Il primo passo è quello di decidere quali siano i parametri della funzione. Il fattoriale ha evidentemente un solo parametro, un intero:

```
def fattoriale(n):
```

Se l'argomento è 0, dobbiamo solo restituire 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:
        ricors = fattoriale(n-1)
        risultato = n * ricors
        return risultato
```

Il flusso di esecuzione del programma è simile a quello di contoallarovescia del Paragrafo 5.8. Se chiamiamo `fattoriale` con il valore 3:

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

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

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

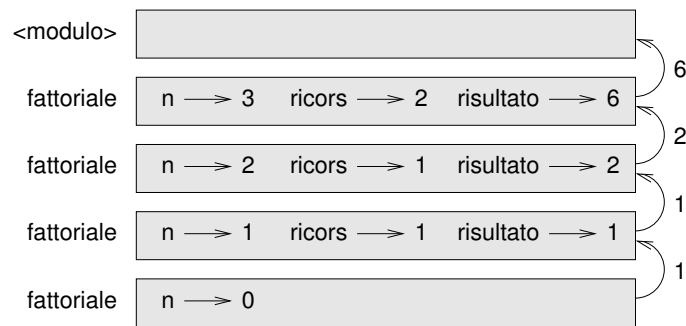


Figura 6.1: Diagramma di stack .

Dato che $0 \neq 0$, seguiamo il primo ramo e ritorniamo 1 senza fare altre chiamate ricorsive.

Il valore di ritorno (1) è moltiplicato per n , che è 1, e il risultato ritorna al chiamante.

Il valore di ritorno (1) è moltiplicato per n , che è 2, e il risultato ritorna al chiamante.

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

La Figura 6.1 mostra il diagramma di stack per l'intera sequenza di chiamate di funzione:

I valori di ritorno sono illustrati mentre vengono passati all'indietro verso l'alto della pila. In ciascun frame, il valore di ritorno è quello di `risultato`, che è il prodotto di n e `ricors`.

Notate che nell'ultimo frame le variabili locali `ricors` e `risultato` non esistono, perché il ramo che le crea non viene eseguito.

6.6 Salto sulla fiducia

Seguire il flusso di esecuzione è il modo giusto di leggere i programmi, ma può diventare rapidamente labirintico se le dimensioni del codice aumentano. Un metodo alternativo è quello che io chiamo "salto sulla fiducia". Quando arrivate ad una chiamata di funzione, invece di seguire il flusso di esecuzione, *date per scontato* che la funzione chiamata si comporti correttamente e che restituisca il valore esatto.

Nei fatti, già praticate questo atto di fede quando utilizzate le funzioni predefinite: se chiamate `math.cos` o `math.exp`, non andate a controllare l'implementazione delle funzioni, ma date per scontato che funzionano a dovere perché le hanno scritte dei validi programmatori.

Lo stesso si può dire per le funzioni che scrivete voi: quando, nel Paragrafo 6.4, abbiamo scritto la funzione `divisibile` che controlla se un numero è divisibile per un altro, e abbiamo verificato che la funzione è corretta,—controllando e provando il codice—possiamo poi usarla senza doverla ricontrollare di nuovo.

Idem quando abbiamo chiamate ricorsive: invece di seguire il flusso di esecuzione, potete partire dal presupposto che la chiamata ricorsiva funzioni (dando il risultato corretto), per

poi chiedervi: “Supponendo che io trovi il fattoriale di $n - 1$, posso calcolare il fattoriale di n ?”. In questo caso è chiaro che potete farlo, moltiplicando per n .

Certo, è strano partire dal presupposto che una funzione sia giusta quando non avete ancora finito di scriverla, ma non per nulla si chiama salto sulla fiducia!

6.7 Un altro esempio

Dopo il fattoriale, l'esempio più noto di funzione matematica definita ricorsivamente è la funzione fibonacci, che ha la seguente definizione: (vedere http://it.wikipedia.org/wiki/Successione_di_Fibonacci):

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

Che tradotta in Python è:

```
def fibonacci (n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Con una funzione simile, provare a seguire il flusso di esecuzione vi farebbe scoppiare la testa anche con valori di n piuttosto piccoli. Ma in virtù del “salto sulla fiducia”, dando per scontato che le due chiamate ricorsive funzionino correttamente, è chiaro che la somma dei loro valori di ritorno sarà corretta.

6.8 Controllo dei tipi

Cosa succede se chiamiamo fattoriale passando 1.5 come argomento?

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

Parrebbe una ricorsione infinita. Ma come mai? Esiste un caso base—quando $n == 0$. Ma se n non è intero, *manchiamo* il caso base e la ricorsione non si ferma più.

Alla prima chiamata ricorsiva, infatti, il valore di n è 0.5. Alla successiva diventa -0.5. Da lì in poi, il valore passato alla funzione diventa ogni volta più piccolo di una unità (cioè più negativo) e non potrà mai essere 0.

Abbiamo due scelte: possiamo tentare di generalizzare la funzione fattoriale per farla funzionare anche nel caso di numeri a virgola mobile, o possiamo far controllare alla funzione se il parametro passato è del tipo corretto. La prima possibilità è chiamata in matematica funzione gamma, ma è un po' oltre gli scopi di questo libro; quindi sceglieremo la seconda alternativa.

Possiamo usare la funzione predefinita `isinstance` per verificare il tipo di argomento. E già che ci siamo, ci accerteremo anche che il numero sia positivo:

```
def fattoriale (n):
    if not isinstance(n, int):
        print "Il fattoriale e' definito solo per numeri interi."
        return None
    elif n < 0:
        print "Il fattoriale non e' definito per interi negativi."
        return None
    elif n == 0:
        return 1
    else:
        return n * fattoriale(n-1)
```

Il primo caso base gestisce i numeri non interi; il secondo cattura gli interi negativi. In entrambi i casi, il programma mostra un messaggio di errore e restituisce il valore `None` per indicare che qualcosa non ha funzionato:

```
>>> fattoriale('alfredo')
Il fattoriale e' definito solo per numeri interi.
None
>>> fattoriale(-2)
Il fattoriale non e' definito per interi negativi.
None
```

Se superiamo entrambi i controlli, possiamo essere certi che n è un intero positivo oppure zero, e che la ricorsione avrà termine.

Questo programma mostra lo schema di funzionamento di una **condizione di guardia**. I primi due controlli agiscono da “guardiani”, difendendo il codice che segue da valori che potrebbero causare errori. Le condizioni di guardia rendono possibile provare la correttezza del codice.

Nel Paragrafo 11.3 vedremo un’alternativa più flessibile alla stampa di messaggi di errore: sollevare un’eccezione.

6.9 Debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole, crea dei naturali punti di controllo per il debug. Se una funzione non va, ci sono tre possibilità da prendere in esame:

- C’è qualcosa di sbagliato negli argomenti che la funzione sta accettando: è violata una *precondizione*.
- C’è qualcosa di sbagliato nella funzione: è violata una *postcondizione*.
- C’è qualcosa di sbagliato nel valore di ritorno o nel modo in cui viene usato.

Per escludere la prima possibilità, potete aggiungere un’istruzione `print` all’inizio della funzione per visualizzare i valori dei parametri (e magari i loro tipi). O potete scrivere del codice che controlla esplicitamente le *precondizioni*.

Se i parametri sembrano corretti, aggiungete un’istruzione `print` prima di ogni istruzione `return` in modo che sia mostrato il valore di ritorno. Se possibile, controllate i risultati

calcolandoveli a parte. Valutate di chiamare la funzione con dei valori che permettono un controllo agevole del risultato (come nel Paragrafo 6.2).

Se la funzione sembra a posto, controllate la chiamata per essere sicuri che il valore di ritorno venga usato correttamente (e soprattutto, venga usato!).

Aggiungere istruzioni di stampa all’inizio e alla fine di una funzione può aiutare a rendere più chiaro il flusso di esecuzione. Ecco una versione di fattoriale con delle istruzioni di stampa:

```
def fattoriale(n):
    spazi = ' ' * (4 * n)
    print spazi, 'fattoriale', n
    if n == 0:
        print spazi, 'ritorno 1'
        return 1
    else:
        ricors = fattoriale(n-1)
        risultato = n * ricors
        print spazi, 'ritorno ', risultato
        return risultato
```

spazi è una stringa di caratteri di spaziatura che controlla l’indentazione dell’output. Ecco il risultato di fattoriale(5) :

```

                fattoriale 5
            fattoriale 4
        fattoriale 3
    fattoriale 2
    fattoriale 1
fattoriale 0
ritorno 1
    ritorno 1
        ritorno 2
            ritorno 6
                ritorno 24
                    ritorno 120
```

Se il flusso di esecuzione vi confonde, questo tipo di output può aiutarvi. Ci vuole un po’ di tempo per sviluppare delle “impalcature” efficaci, ma queste possono far risparmiare molto tempo di debug.

6.10 Glossario

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 return.

None: Valore speciale restituito da una funzione che non ha un’istruzione return o ha un’istruzione return priva di argomento.

sviluppo incrementale: Tecnica di sviluppo del programma inteso ad evitare lunghe sessioni di debug, aggiungendo e provando piccole porzioni di codice alla volta.

impalcatura: Codice temporaneo inserito solo nella fase di sviluppo del programma e che non fa parte della versione finale.

condizione di guardia: Schema di programmazione che usa una condizione per controllare e gestire le circostanze che possono causare un errore.

6.11 Esercizi

Esercizio 6.4. *Disegnate un diagramma di stack del seguente programma. Che cosa stampa?*

Soluzione: http://thinkpython.com/code/stack_diagram.py.

```
def b(z):
    prod = a(z, z)
    print z, prod
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square

x = 1
y = x + 1
print c(x, y+3, x+y)
```

Esercizio 6.5. *La funzione di Ackermann, $A(m, n)$, è così definita:*

$$A(m, n) = \begin{cases} n + 1 & \text{se } m = 0 \\ A(m - 1, 1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{se } m > 0 \text{ e } n > 0. \end{cases}$$

Vedere anche http://it.wikipedia.org/wiki/Funzione_di_Ackermann. Scrivete una funzione di nome `ack` che valuti la funzione di Ackermann. Usate la vostra funzione per calcolare `ack(3, 4)`, vi dovrebbe risultare 125. Cosa succede per valori maggiori di m e n ? Soluzione: <http://thinkpython.com/code/ackermann.py>.

Esercizio 6.6. *Un palindromo è una parola che si legge nello stesso modo sia da sinistra verso destra che viceversa, come “ottetto” e “radar”. In termini ricorsivi, una parola è un palindromo se la prima e l’ultima lettera sono uguali e ciò che resta in mezzo è un palindromo.*

Quelle che seguono sono funzioni che hanno una stringa come parametro e restituiscono rispettivamente la prima lettera, l’ultima lettera, e quelle in mezzo:

```
def prima(parola):  
    return parola[0]  
  
def ultima(parola):  
    return parola[-1]  
  
def mezzo(parola):  
    return parola[1:-1]
```

Vedremo meglio come funzionano nel Capitolo 8.

1. *Scrivete queste funzioni in un file script `palindromo.py` e provatele. Cosa succede se chiamate `mezzo` con una stringa di due lettere? E di una lettera? E con la stringa vuota, che si scrive `' '` e non contiene caratteri?*
2. *Scrivete una funzione di nome `palindromo` che riceva una stringa come argomento e restituisca `True` se è un palindromo e `False` altrimenti. Ricordate che potete usare la funzione predefinita `len` per controllare la lunghezza di una stringa.*

Soluzione: http://thinkpython.com/code/palindrome_soln.py.

Esercizio 6.7. *Un numero, a , è una potenza di b se è divisibile per b e a/b è una potenza di b . Scrivete una funzione di nome `potenza` che prenda come parametri a e b e che restituisca `True` se a è una potenza di b . Nota: dovete pensare bene al caso base.*

Esercizio 6.8. *Il massimo comun divisore (MCD) di due interi a e b è il numero intero più grande che divide entrambi senza dare resto.*

Un modo per trovare il MCD di due numeri si basa sull'osservazione che, se r è il resto della divisione tra a e b , allora $\text{mcd}(a, b) = \text{mcd}(b, r)$. Come caso base, possiamo usare $\text{mcd}(a, 0) = a$.

Scrivete una funzione di nome `mcd` che abbia come parametri a e b e restituisca il loro massimo comun divisore.

Fonte: Questo esercizio è basato su un esempio in Structure and Interpretation of Computer Programs di Abelson e Sussman.

Capitolo 7

Iterazione

7.1 Assegnazioni multiple

Vi sarete probabilmente già accorti che è possibile effettuare più assegnazioni ad una stessa variabile. Una nuova assegnazione fa sì che la variabile faccia riferimento ad un nuovo valore (cessando di riferirsi a quello vecchio).

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

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

La Figura 7.1 illustra il diagramma di stato per questa **assegnazione multipla**.

Nel caso di assegnazioni multiple, è particolarmente importante distinguere tra operazioni di assegnazione e controlli di uguaglianza. Dato che Python usa (=) per le assegnazioni, potreste interpretare l'istruzione `a = b` come un'espressione di uguaglianza, ma non lo è!

In primo luogo l'equivalenza è simmetrica, cioè vale in entrambi i sensi, mentre l'assegnazione non lo è: in matematica se $a = 7$ allora è anche $7 = a$. Ma in Python l'istruzione `a = 7` è valida mentre `7 = a` non lo è.

Inoltre, in matematica un'uguaglianza è o vera o falsa, e rimane tale: se ora $a = b$ allora a sarà sempre uguale a b . In Python, un'assegnazione può rendere due variabili temporaneamente uguali, ma non è affatto detto che l'uguaglianza permanga:

```
a = 5
b = a    # a e b ora sono uguali
a = 3    # a e b non sono piu' uguali
```

La terza riga cambia il valore di `a` ma non cambia quello di `b`, quindi `a` e `b` non sono più uguali.

Anche se le assegnazioni multiple sono spesso utili, vanno usate con cautela. Se il valore di una variabile cambia di frequente, può rendere il codice difficile da leggere e correggere.

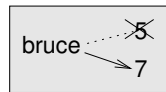


Figura 7.1: Diagramma di stato.

7.2 Aggiornare le variabili

Una delle forme più comuni di assegnazione multipla è l'**aggiornamento**, dove il nuovo valore della variabile dipende da quello precedente.

```
x = x+1
```

Questo vuol dire: “prendi il valore attuale di *x*, aggiungi uno, e aggiorna *x* al nuovo valore.”

Se tentate di aggiornare una variabile inesistente, si verifica un errore perché Python valuta il lato destro prima di assegnare un valore a *x*:

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

Prima di aggiornare una variabile occorre quindi **inizializzarla**, di solito con una comune assegnazione:

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

L'aggiornamento di una variabile aggiungendo 1 è detto **incremento**; sottrarre 1 è detto invece **decremento**.

7.3 L'istruzione while

Spesso i computer sono usati per automatizzare dei compiti ripetitivi: ripetere operazioni identiche o simili un gran numero di volte senza fare errori, è qualcosa che i computer fanno molto bene e le persone piuttosto male.

Abbiamo visto due programmi, *contoallarovescia* e *stampa_n*, che usano la ricorsione per eseguire una ripetizione, che è anche chiamata **iterazione**. Dato che l'iterazione è un'operazione molto frequente, Python fornisce varie caratteristiche del linguaggio per renderla più semplice da implementare. Una è l'istruzione *for*, che abbiamo già visto nel Paragrafo 4.2 e sulla quale torneremo.

Un'altra istruzione è *while*. Ecco una variante di *contoallarovescia* che usa l'istruzione *while*:

```
def contoallarovescia(n):
    while n > 0:
        print n
        n = n-1
    print 'Via!'
```

Si può quasi leggere il programma con l'istruzione *while* come fosse scritto in inglese: significa “Finché (while) *n* è maggiore di 0, stampa il valore di *n* e poi decrementa *n* di 1. Quando arrivi a 0, stampa la stringa *Via!*”

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

1. Valuta la condizione, controllando se è vera (True) o falsa (False).
2. Se la condizione è falsa, esce dal ciclo `while` e continua l'esecuzione dalla prima istruzione che lo segue.
3. Se la condizione è vera, esegue tutte le istruzioni nel corpo del ciclo `while` e vi rimane, ritornando al punto 1.

Questo tipo di flusso è chiamato **ciclo** (in inglese *loop*), perché il terzo punto ritorna ciclicamente da capo.

Il corpo del ciclo deve cambiare il valore di una o più variabili in modo che la condizione possa prima o poi diventare falsa e fare in modo che il ciclo abbia termine. In caso contrario il ciclo si ripeterebbe continuamente, determinando un **ciclo infinito**. Una fonte inesauribile di divertimento per gli informatici, è osservare che le istruzioni dello shampoo: “lava, risciacqua, ripeti” sono un ciclo infinito.

Nel caso di contoallaroveschia, possiamo essere certi che il ciclo terminerà, visto che il valore di `n` diventa via via più piccolo ad ogni ripetizione del ciclo stesso, fino a diventare, prima o poi, zero. In altri casi può non essere così facile stabilirlo:

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

La condizione di questo ciclo è `n != 1`, per cui il ciclo si ripeterà fino a quando `n` non sarà uguale a 1, cosa che rende falsa la condizione.

Ad ogni ripetizione del 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 è dispari, `n` è moltiplicato per 3 e al risultato viene aggiunto 1. Se per esempio il valore passato a `sequenza` è 3, la sequenza risultante sarà 3, 10, 5, 16, 8, 4, 2, 1.

Dato che `n` a volte sale e a volte scende, non c'è prova evidente che `n` raggiungerà 1 in modo da terminare il ciclo. Per qualche particolare valore di `n`, possiamo dimostrarlo: ad esempio, se il valore di partenza è una potenza di 2, il valore di `n` sarà per forza un numero pari per ogni ciclo, fino a raggiungere 1. L'esempio precedente finisce proprio con una sequenza simile, a partire dal numero 16.

La domanda difficile è se il programma giunga a termine per *qualsiasi valore positivo* di `n`. Sinora, nessuno è riuscito a dimostrarlo né a smentirlo! (Vedere http://it.wikipedia.org/wiki/Congettura_di_Collatz.)

Esercizio 7.1. *Riscrivete la funzione `stampa_n` del Paragrafo 5.8 usando l'iterazione al posto della ricorsione.*

7.4 break

Vi può capitare di poter stabilire il momento in cui è necessario terminare un ciclo solo mentre il flusso di esecuzione si trova nel bel mezzo del corpo. In questi casi potete usare l'istruzione `break` per saltare fuori dal ciclo.

Per esempio, supponiamo che vogliate ricevere delle risposte dall'utente, fino a quando non viene digitata la parola fatto. Potete scrivere:

```
while True:
    riga = raw_input('> ')
    if riga == 'fatto':
        break
    print riga

print 'Fatto!'
```

La condizione del ciclo è True, che è sempre vera per definizione, quindi il ciclo è destinato a continuare, a meno che non incontri l'istruzione break.

Ad ogni ripetizione, il programma mostra come prompt il simbolo >. Se l'utente scrive fatto, l'istruzione break interrompe il ciclo, altrimenti ripete quello che l'utente ha scritto e ritorna da capo. Ecco un esempio di esecuzione:

```
> non fatto
non fatto
> fatto
Fatto!
```

Questo modo di scrivere i cicli while è frequente, perché vi permette di controllare la condizione ovunque all'interno del ciclo (e non solo all'inizio) e di esprimere la condizione di stop in modo affermativo ("fermati quando succede questo") piuttosto che negativo ("continua fino a quando non succede questo").

7.5 Radici quadrate

Spesso si usano i cicli per calcolare risultati numerici, partendo da un valore approssimativo che viene migliorato iterativamente con approssimazioni successive.

Per esempio, un modo di calcolare le radici quadrate è il metodo di Newton. Supponiamo di voler calcolare la radice quadrata di a . A partire da una qualunque stima, x , possiamo calcolare una stima migliore con la formula seguente:

$$y = \frac{x + a/x}{2}$$

Supponiamo per esempio che a sia 4 e x sia 3:

```
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print y
2.16666666667
```

Che è più vicino al valore vero ($\sqrt{4} = 2$). Se ripetiamo il procedimento usando la nuova stima, ci avviciniamo ulteriormente:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00641025641
```

Dopo qualche ulteriore passaggio, la stima diventa quasi esatta:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00000000003
```

In generale, non possiamo sapere *a priori* quanti passaggi ci vorranno per ottenere la risposta esatta, ma sapremo che ci saremo arrivati quando la stima non cambierà più:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.0
```

Possiamo fermarci quando $y == x$. Ecco quindi un ciclo che parte da una stima iniziale, x , e la migliora fino a quando non cambia più:

```
while True:
    print x
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

Per la maggior parte dei valori di a , questo codice funziona bene, ma in genere è pericoloso testare l'uguaglianza su valori decimali di tipo `float`, perché sono solo approssimativamente esatti: la maggior parte dei numeri razionali come $1/3$, e irrazionali, come $\sqrt{2}$, non possono essere rappresentati in modo preciso con un `float`.

Piuttosto di controllare se x e y sono identici, è meglio usare la funzione predefinita `abs` per calcolare il valore assoluto della loro differenza:

```
if abs(y-x) < epsilon:
    break
```

Dove `epsilon` è un valore molto piccolo, come `0.0000001`, che determina quando i due numeri confrontati sono abbastanza vicini da poter essere considerati praticamente uguali.

Esercizio 7.2. Incapsulate questo ciclo in una funzione di nome `radice_quad` che richieda `a` come parametro, scelga un valore ragionevole di x , e restituisca una stima della radice quadrata di a .

7.6 Algoritmi

Il metodo di Newton è un esempio di **algoritmo**: è un'operazione meccanica per risolvere un tipo di problema (in questo caso, calcolare la radice quadrata).

Non è facile definire un algoritmo. Può essere utile iniziare a vedere cosa non è un algoritmo. Quando avete imparato a fare le moltiplicazioni dei numeri a una cifra, probabilmente avete imparato a memoria le tabelline, che significa ricordare 100 specifiche soluzioni. Una conoscenza di questo tipo non è algoritmica.

Ma se eravate dei bambini un po' pigri, probabilmente avete cercato di imparare qualche truccetto. Per esempio, per trovare il prodotto tra n e 9, si scrive $n - 1$ come prima cifra e $10 - n$ come seconda cifra. Questo trucco è una soluzione generica per moltiplicare per nove qualunque numero a una cifra. Questo è un algoritmo!

Similmente, le tecniche che avete imparato per l'addizione con riporto, la sottrazione con prestito e le divisioni lunghe sono tutte algoritmi. Una caratteristica degli algoritmi è che non richiedono intelligenza per essere eseguiti. Sono procedimenti meccanici in cui ad ogni passo ne segue un altro, secondo delle semplici regole.

A mio parere, è piuttosto imbarazzante che le persone impieghino tanto tempo a scuola per imparare ad eseguire degli algoritmi che, letteralmente, non richiedono alcuna intelligenza.

D'altra parte, la procedura di realizzazione di un algoritmo è interessante, intellettualmente stimolante, e una parte cruciale di quella che chiamiamo programmazione.

Alcune delle cose che le persone fanno in modo naturale senza difficoltà o senza nemmeno pensarci, sono le più difficili da esprimere con algoritmi. Capire il linguaggio naturale è un esempio calzante. Lo facciamo tutti, ma finora nessuno è stato in grado di spiegare *come* lo facciamo, almeno non sotto forma di un algoritmo.

7.7 Debug

Quando inizierete a scrivere programmi di grande dimensioni, aumenterà il tempo da dedicare al debug. Più codice significa più probabilità di commettere un errore e più posti in cui gli errori possono annidarsi.

Un metodo per ridurre il tempo di debug è il "debug binario". Se nel vostro programma ci sono 100 righe e le controllate una ad una, ci vorranno 100 passaggi.

Provate invece a dividere il problema in due. Cercate verso la metà del programma un valore intermedio che potete controllare. Aggiungete un'istruzione `print` (o qualcos'altro di controllabile) ed eseguite il programma.

Se il controllo nel punto mediano non è corretto, deve esserci un problema nella prima metà del programma. Se invece è corretto, l'errore sarà nella seconda metà.

Per ogni controllo eseguito in questa maniera, dimezzate le righe da controllare. Dopo 6 passaggi (che sono meno di 100), dovreste teoricamente arrivare a una o due righe di codice.

In pratica, non è sempre chiaro quale sia la "metà del programma" e non è sempre possibile controllare. Non ha neanche molto senso contare le righe e trovare la metà esatta. Meglio considerare i punti del programma dove è più probabile che vi siano errori e quelli dove è facile posizionare dei controlli. Quindi, scegliere un punto dove stimate che le probabilità che l'errore sia prima o dopo quel punto siano circa le stesse.

7.8 Glossario

assegnazione multipla: Assegnazione alla stessa variabile di valori diversi nel corso del programma.

aggiornamento: Assegnazione in cui il nuovo valore della variabile dipende da quello precedente.

inizializzazione: Assegnazione che fornisce un valore iniziale ad una variabile da aggiornare successivamente.

incremento: Aggiornamento che aumenta il valore di una variabile (spesso di una unità).

decremento: Aggiornamento che riduce il valore di una variabile.

iterazione: Ripetizione di una serie di istruzioni utilizzando una funzione ricorsiva oppure un ciclo.

ciclo infinito: Ciclo nel quale la condizione che ne determina la fine non è mai soddisfatta.

7.9 Esercizi

Esercizio 7.3. *Per verificare l'algoritmo di calcolo della radice quadrata illustrato in questo capitolo, potete confrontarlo con la funzione `math.sqrt`. Scrivete una funzione di nome `test_radice_quad` che stampi una tabella come questa:*

1.0	1.0	1.0	0.0
2.0	1.41421356237	1.41421356237	2.22044604925e-16
3.0	1.73205080757	1.73205080757	0.0
4.0	2.0	2.0	0.0
5.0	2.2360679775	2.2360679775	0.0
6.0	2.44948974278	2.44948974278	0.0
7.0	2.64575131106	2.64575131106	0.0
8.0	2.82842712475	2.82842712475	4.4408920985e-16
9.0	3.0	3.0	0.0

La prima colonna è un numero, a ; la seconda è la radice quadrata di a calcolata con la funzione del Paragrafo 7.5; la terza è quella calcolata con `math.sqrt`; la quarta è il valore assoluto della differenza tra le due stime.

Esercizio 7.4. *La funzione predefinita `eval` valuta un'espressione sotto forma di stringa, usando l'interprete Python. Ad esempio:*

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<type 'float'>
```

Scrivete una funzione di nome `eval_ciclo` che chieda iterativamente all'utente di inserire un dato, prenda il dato inserito e lo valuti con `eval`, infine visualizzi il risultato.

Deve continuare fino a quando l'utente non scrive `'fatto'`, e poi restituire il valore dell'ultima espressione che ha valutato.

Esercizio 7.5. Il matematico Srinivasa Ramanujan scoprì una serie infinita che può essere usata per generare un'approssimazione di $1/\pi$:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Scrivete una funzione di nome `stima_pi` che utilizzi questa formula per calcolare e restituire una stima di π . Deve usare un ciclo `while` per calcolare gli elementi della sommatoria, fino a quando l'ultimo termine è più piccolo di $1e-15$ (che è la notazione di Python per 10^{-15}). Controllate il risultato confrontandolo con `math.pi`.

Soluzione: <http://thinkpython.com/code/pi.py>.

Capitolo 8

Stringhe

8.1 Una stringa è una sequenza

Una stringa è una **sequenza** di caratteri. Potete accedere ai singoli caratteri usando gli operatori parentesi quadre:

```
>>> frutto = 'banana'
>>> lettera = frutto[1]
```

La seconda istruzione seleziona il carattere numero 1 della variabile `frutto` e lo assegna a `lettera`.

L'espressione all'interno delle parentesi quadre è chiamato **indice**. L'indice indica (di qui il nome) quale carattere della sequenza desiderate estrarre.

Ma il risultato potrebbe lasciarvi perplessi:

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

Per la maggior parte delle persone, la prima lettera di `'banana'` è `b`, non `a`. Ma per gli informatici, premesso che l'indice è la posizione a partire dall'inizio della stringa, la posizione della prima lettera è considerata la numero zero, non uno.

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

Quindi `b` è la “zero-esima” lettera di `'banana'`, `a` è la prima lettera (“1-esima”), e `n` è la seconda (“2-esima”) lettera.

Potete usare come indice qualsiasi espressione, compresi variabili e operatori, ma il valore risultante deve essere un intero. Altrimenti succede questo:

```
>>> lettera = frutto[1.5]
TypeError: string indices must be integers, not float
```

8.2 `len`

`len` è una funzione predefinita che restituisce il numero di caratteri di una stringa:

```
>>> frutto = 'banana'
>>> len(frutto)
6
```

Per estrarre l'ultimo carattere di una stringa, potreste pensare di scrivere qualcosa del genere:

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

La ragione dell'`IndexError` è che non c'è nessuna lettera in 'banana' con indice 6. Siccome partiamo a contare da zero, le sei lettere sono numerate da 0 a 5. Per estrarre l'ultimo carattere, dobbiamo perciò sottrarre 1 da lunghezza:

```
>>> ultimo = frutto[lunghezza-1]
>>> print ultimo
a
```

In alternativa, possiamo usare utilmente gli indici negativi, che contano a ritroso dalla fine della stringa: l'espressione `frutto[-1]` ricava l'ultimo carattere della stringa, `frutto[-2]` il penultimo carattere, e così via.

8.3 Attraversamento con un ciclo `for`

Molti tipi di calcolo comportano l'elaborazione di una stringa, un carattere per volta. Spesso iniziano dal primo carattere, selezionano un carattere per volta, eseguono una certa operazione e continuano fino al completamento della stringa. Questo tipo di elaborazione è definita **attraversamento**. Un modo per scrivere un attraversamento è quello di usare un ciclo `while`:

```
indice = 0
while indice < len(frutto):
    lettera = frutto[indice]
    print lettera
    indice = indice + 1
```

Questo ciclo attraversa tutta la stringa e ne mostra una lettera alla volta, una per riga. La condizione del ciclo è `indice < len(frutto)`, pertanto quando `indice` è uguale alla lunghezza della stringa la condizione diventa falsa, il corpo del ciclo non viene più 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 8.1. *Scrivete una funzione che riceva una stringa come argomento e ne stampi i singoli caratteri, uno per riga, partendo dall'ultimo a ritroso.*

Un altro modo di scrivere un attraversamento è usare un ciclo `for`:

```
for lettera in frutto:
    print lettera
```

Ad ogni ciclo, il successivo carattere della stringa viene assegnato alla variabile `lettera`. Il ciclo continua finché non rimangono più caratteri da analizzare.

L'esempio seguente mostra come usare il concatenamento (addizione di stringhe) e un ciclo `for` per generare una serie alfabetica, cioè una lista di valori in cui gli elementi appaiono

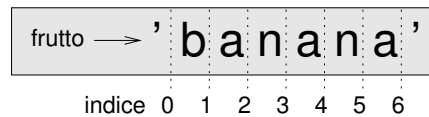


Figura 8.1: Indici di slicing.

in ordine alfabetico. Per esempio, nel libro *Make Way for Ducklings* di Robert McCloskey, ci sono degli anatroccoli che si chiamano 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
Quack
```

È evidente che non è del tutto giusto, dato che “Ouack” e “Quack” sono scritti in modo errato.

Esercizio 8.2. *Modificate il programma per correggere questo errore.*

8.4 Slicing

Un segmento o porzione di stringa è chiamato **slice**. L’operazione di selezione di una porzione di stringa è simile alla selezione di un carattere, ed è detta **slicing**:

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:12]
Python
```

L’operatore `[n:m]` restituisce la porzione di stringa nell’intervallo compreso tra l’“n-esimo” carattere incluso fino all’“m-esimo” escluso. Questo comportamento è poco intuitivo, e per tenerlo a mente può essere d’aiuto immaginare gli indici puntare *tra* i caratteri, come spiegato nella Figura 8.1.

Se non è specificato il primo indice (quello prima dei due punti :), la porzione parte dall’inizio della stringa. Se manca il secondo indice, la porzione arriva fino in fondo alla stringa:

```
>>> frutto = 'banana'
>>> frutto[:3]
'ban'
>>> frutto[3:]
'ana'
```

Se il primo indice è maggiore o uguale al secondo, il risultato è una **stringa vuota**, rappresentata da due apici consecutivi.

```
>>> frutto = 'banana'
>>> frutto[3:3]
''
```

Una stringa vuota non contiene caratteri e ha lunghezza 0, ma a parte questo è a tutti gli effetti una stringa come le altre.

Esercizio 8.3. *Data una stringa di nome `frutto`, che cosa significa `frutto[:]`?*

8.5 Le stringhe sono immutabili

Potreste pensare di utilizzare l'operatore `[]` sul lato sinistro di un'assegnazione per cambiare un carattere all'interno di una stringa. Per esempio così:

```
>>> saluto = 'Ciao, mondo!'
>>> saluto[0] = 'M'
TypeError: 'str' object does not support item assignment
```

L'"oggetto" (*object*) in questo caso è la stringa, e l'"elemento" (*item*) è il carattere che avete tentato di assegnare. Per ora, consideriamo un **oggetto** come la stessa cosa di un valore, ma più avanti puntualizzeremo meglio questa definizione. Un **elemento** è uno dei valori in una sequenza.

La ragione dell'errore è che le stringhe sono **immutabili**, in altre parole, non potete cambiare una stringa esistente. La miglior cosa da fare è creare una nuova stringa, variante dell'originale:

```
>>> saluto = 'Ciao, mondo!'
>>> nuovo_saluto = 'M' + saluto[1:]
>>> print nuovo_saluto
Miao, mondo!
```

Questo esempio concatena una nuova prima lettera con la restante porzione di saluto. Non ha alcun effetto sulla stringa di origine, che resta invariata.

8.6 Ricerca

Cosa fa la funzione seguente?

```
def trova(parola, lettera):
    indice = 0
    while indice < len(parola):
        if parola[indice] == lettera:
            return indice
        indice = indice + 1
    return -1
```

In un certo senso, questa funzione trova è l'opposto dell'operatore []. Invece di prendere un indice ed estrarre il carattere corrispondente, prende un carattere e trova l'indice in corrispondenza del quale appare il carattere. Se non trova il carattere indicato nella parola, la funzione restituisce -1.

Per la prima volta incontriamo l'istruzione `return` all'interno di un ciclo. Se `parola[indice] == lettera`, la funzione interrompe il ciclo e ritorna immediatamente, restituendo `indice`.

Se il carattere non compare nella stringa data, il programma termina il ciclo normalmente e restituisce -1.

Questo schema di calcolo—attraversare una sequenza e ritornare quando trova quello che sta cercando—è chiamato **ricerca**.

Esercizio 8.4. *Modificate la funzione `trova` in modo che richieda un terzo parametro, che rappresenta la posizione da cui si deve cominciare la ricerca all'interno della stringa `parola`.*

8.7 Cicli e contatori

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

```
parola = 'banana'
conta = 0
for lettera in parola:
    if lettera == 'a':
        conta = conta + 1
print conta
```

Si tratta di un altro schema di calcolo chiamato **contatore**. La variabile `conta` è inizializzata a 0, quindi incrementata di uno per ogni volta che viene trovata una `a`. Al termine del ciclo, `conta` contiene il risultato: il numero totale di lettere `a` nella stringa.

Esercizio 8.5. *Incapsulate questo codice in una funzione di nome `conta`, e generalizzatela in modo che accetti come argomenti sia la stringa che la lettera da cercare.*

Esercizio 8.6. *Riscrivete questa funzione in modo che, invece di attraversare completamente la stringa, faccia uso della versione a tre parametri di `trova`, vista nel precedente paragrafo.*

8.8 Metodi delle stringhe

Un **metodo** è simile a una funzione—riceve argomenti e restituisce un valore—ma la sintassi è diversa. Ad esempio, il metodo `upper` prende una stringa e crea una nuova stringa di tutte lettere maiuscole.

Al posto della sintassi della funzione, `upper(parola)`, usa la sintassi del metodo, `parola.upper()`.

```
>>> parola = 'banana'
>>> nuova_parola = parola.upper()
>>> print nuova_parola
BANANA
```

Questa forma di notazione a punto, in inglese *dot notation*, specifica il nome del metodo, upper, e il nome della stringa a cui va applicato il metodo, parola. Le parentesi vuote indicano che il metodo non ha argomenti.

La chiamata di un metodo è detta **invocazione**; nel nostro caso, diciamo che stiamo invocando upper su parola.

Visto che ci siamo, esiste un metodo delle stringhe chiamato find che è molto simile alla funzione che abbiamo scritto prima:

```
>>> parola = 'banana'
>>> indice = parola.find('a')
>>> print indice
1
```

In questo esempio, abbiamo invocato find su parola e abbiamo passato come parametro la lettera che stiamo cercando.

In realtà, il metodo find è più generale della nostra funzione: può ricercare anche sottostringhe e non solo singoli caratteri:

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

Può inoltre ricevere come secondo argomento l'indice da cui partire:

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

E come terzo argomento l'indice in corrispondenza del quale fermarsi:

```
>>> nome = 'bob'
>>> nome.find('b', 1, 2)
-1
```

In quest'ultimo caso la ricerca fallisce, perché b non è compreso nell'intervallo da 1 a 2 (2 si considera escluso).

Esercizio 8.7. Esiste un metodo delle stringhe di nome count che è simile alla funzione dell'esercizio precedente. Leggete la documentazione del metodo e scrivete un'invocazione che conti il numero di a in 'banana'.

Esercizio 8.8. Leggete la documentazione dei metodi delle stringhe sul sito <http://docs.python.org/2/library/stdtypes.html#string-methods>. Fate degli esperimenti con alcuni metodi per assicurarvi di avere capito come funzionano. strip e replace sono particolarmente utili.

La documentazione utilizza una sintassi che può risultare poco chiara. Per esempio, in find(sub[, start[, end]]), le parentesi quadre indicano dei parametri opzionali (non vanno digitate). Quindi sub è obbligatorio, ma start è opzionale, e se indicate start, allora end è a sua volta opzionale.

8.9 L'operatore in

La parola in è un operatore booleano che confronta due stringhe e restituisce True se la prima è una sottostringa della seconda:


```
>>> 'a' in 'banana'
True
>>> 'seme' in 'banana'
False
```

Ad esempio, la funzione che segue stampa tutte le lettere di parola1 che compaiono anche in parola2:

```
def in_entrambe(parola1, parola2):
    for lettera in parola1:
        if lettera in parola2:
            print lettera
```

Con qualche nome di variabile scelto bene, Python a volte si legge quasi come fosse un misto di inglese e italiano: “per (ogni) lettera in parola1, se (la) lettera (è) in parola2, stampa (la) lettera.”

Ecco cosa succede se paragonate carote e patate:

```
>>> in_entrambe('carote', 'patate')
a
t
e
```

8.10 Confronto di stringhe

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

```
if parola == 'banana':
    print 'Tutto ok, banane.'
```

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 'Tutto ok, banane.'
```

Attenzione che Python non gestisce le parole maiuscole e minuscole come siamo abituati: in un confronto, le lettere maiuscole vengono sempre prima di tutte le minuscole, così che:

La tua parola, Papaya, viene prima di banana.

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

8.11 Debug

Quando usate gli indici per l’attraversamento dei valori di una sequenza, non è facile determinare bene l’inizio e la fine. Ecco una funzione che vorrebbe confrontare due parole e restituire True quando una parola è scritta al contrario dell’altra, ma contiene due errori:

```
def al_contrario(parola1, parola2):
    if len(parola1) != len(parola2):
        return False

    i = 0
    j = len(parola2)

    while j > 0:
        if parola1[i] != parola2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

La prima istruzione `if` controlla se le parole sono della stessa lunghezza. Se non è così, possiamo restituire immediatamente `False`, altrimenti, per il resto della funzione, possiamo presupporre che le parole hanno pari lunghezza. È un altro esempio di condizione di guardia, vista nel Paragrafo 6.8.

`i` e `j` sono indici: `i` attraversa `parola1` in avanti, mentre `j` attraversa `parola2` a ritroso. Se troviamo due lettere che non coincidono, possiamo restituire subito `False`. Se continuiamo per tutto il ciclo e tutte le lettere coincidono, il valore di ritorno è `True`.

Se proviamo la funzione con i valori “pots” e “stop”, ci aspetteremmo di ricevere di ritorno `True`, invece risulta un `IndexError`:

```
>>> al_contrario('pots', 'stop')
...
File "reverse.py", line 15, in al_contrario
    if parola1[i] != parola2[j]:
IndexError: string index out of range
```

Per fare il debug, la mia prima mossa è di stampare il valore degli indici appena prima della riga dove è comparso l’errore.

```
while j > 0:
    print i, j          # stampare qui

    if parola1[i] != parola2[j]:
        return False
    i = i+1
    j = j-1
```

Ora, eseguendo di nuovo il programma, ho qualche informazione in più:

```
>>> al_contrario('pots', 'stop')
0 4
...
IndexError: string index out of range
```

Alla prima esecuzione del ciclo, il valore di `j` è 4, che è fuori intervallo della stringa ‘pots’. Infatti l’indice dell’ultimo carattere è 3, e il valore iniziale di `j` va corretto in `len(parola2)-1`.

Se correggo l’errore e rieseguo ancora il programma:

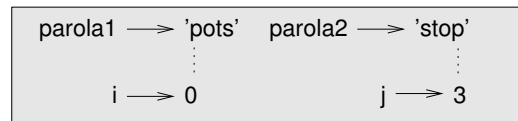


Figura 8.2: Diagramma di Stato.

```
>>> al_contrario('pots', 'stop')
0 3
1 2
2 1
True
```

Stavolta il risultato è giusto, ma pare che il ciclo sia stato eseguito solo per tre volte, il che è sospetto. Per avere un'idea di cosa stia succedendo, è utile disegnare un diagramma di stato. Durante la prima iterazione, il frame di `al_contrario` è illustrato in Figura 8.2.

Ho fatto una piccola eccezione, disponendo le variabili nel frame e aggiungendo delle linee tratteggiate per evidenziare che i valori di `i` e `j` indicano i caratteri in `parola1` e `parola2`.

Esercizio 8.9. *Partendo da questo diagramma, sviluppate il programma su carta cambiando i valori di `i` e `j` ad ogni iterazione. Trovate e correggete il secondo errore in questa funzione.*

8.12 Glossario

oggetto: Qualcosa a cui una variabile può fare riferimento. Per ora, potete utilizzare “oggetto” e “valore” indifferentemente.

sequenza: Un insieme ordinato di valori, ovvero un gruppo di valori in cui ciascuno è identificato da un numero intero.

elemento: Uno dei valori di una sequenza.

indice: Un valore intero usato per selezionare un elemento di una sequenza, come un carattere in una stringa.

slice: Porzione di una stringa identificata tramite un intervallo di indici.

stringa vuota: Una stringa priva di caratteri e di lunghezza 0, rappresentata da due apici o virgolette successive.

immutabile: Detto di una sequenza i cui elementi non possono essere assegnati.

attraversare: Iterare attraverso gli elementi di una sequenza, effettuando su ciascuno un'operazione simile.

ricerca: Schema di attraversamento che si ferma quando trova ciò che si sta cercando.

contatore: Variabile utilizzata per contare qualcosa, solitamente inizializzata a zero e poi incrementata.

metodo: Funzione associata ad un oggetto e invocata utilizzando la notazione a punto.

invocazione: Istruzione che chiama un metodo.

8.13 Esercizi

Esercizio 8.10. Nello slicing, si può specificare un terzo indice che stabilisce lo step o “passo”, cioè il numero di elementi da saltare tra un carattere estratto e il successivo. Uno step di 2 significa estrarre un carattere ogni 2 (uno sì, uno no), 3 significa uno ogni 3 (uno sì, due no), ecc.

```
>>> frutto = 'banana'
>>> frutto[0:5:2]
'bnn'
```

Uno step di -1 fa scorrere all’indietro nella parola, per cui lo slice `[: :-1]` genera una stringa scritta al contrario.

Usate questo costrutto per scrivere una variante di una sola riga della funzione `palindromo` dell’Esercizio 6.6.

Esercizio 8.11. Tutte le funzioni che seguono dovrebbero controllare se una stringa contiene almeno una lettera minuscola, ma qualcuna di esse è sbagliata. Per ogni funzione, descrivete cosa fa in realtà (supponendo che il parametro sia una stringa).

```
def una_minuscola1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def una_minuscola2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def una_minuscola3(s):
    for c in s:
        flag = c.islower()
    return flag

def una_minuscola4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def una_minuscola5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Esercizio 8.12. ROT13 è un metodo di criptazione debole che consiste nel “ruotare” ogni lettera di una parola di 13 posti seguendo la sequenza alfabetica, ricominciando da capo quando necessario. Ad

esempio 'A' ruotata di 3 posti diventa 'D', 'Z' ruotata di 1 posto diventa 'A'. Scrivete una funzione di nome `ruota_parola` che richieda una stringa e un intero come parametri, e che restituisca una nuova stringa che contiene le lettere della stringa di partenza ruotate della quantità indicata. Per esempio, "cheer" ruotata di 7 dà "jolly" e "melon" ruotata di -10 dà "cubed".

Potete usare le funzioni predefinite `ord`, che converte un carattere in un codice numerico, e `chr`, che converte i codici numerici in caratteri.

Su Internet, talvolta, vengono codificate in ROT13 delle barzellette potenzialmente offensive. Se non siete suscettibili, cercatene qualcuna e decodificatela. Soluzione: <http://thinkpython.com/code/rotate.py>.

Capitolo 9

Esercitazione: Giochi con le parole

9.1 Leggere elenchi di parole

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

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

```
>>> fin = open('words.txt')
>>> print fin
<open file 'words.txt', mode 'r' at 0xb7f4b380>
```

`fin` è un nome comunemente usato per un oggetto file usato per operazioni di input. La modalità `'r'` indica che il file è aperto in lettura (per contro, `'w'` sta per scrittura).

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

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

La prima parola di questa speciale lista è “aa”, che è un tipo di lava vulcanica. La sequenza `\r\n` rappresenta due caratteri spaziatori, un ritorno di carrello e un a capo, che separano questa parola dalla successiva.

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

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

La parola successiva è “aah”, che è perfettamente valida per cui non fate quella faccia! Oppure, se gli spaziatori vi danno fastidio, potete sbarazzarvene con il metodo delle stringhe `strip`:

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

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

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

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

9.2 Esercizi

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

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

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

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

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

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

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

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

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

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

9.3 Ricerca

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.4 Cicli con gli indici

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

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

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

Un'alternativa è usare la ricorsione:

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

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

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

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

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

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

Ecco una variante di palindromo (vedere l'Esercizio 6.6) che usa due indici; uno parte dall'inizio e aumenta, uno parte dalla fine e diminuisce.

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

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

    return True
```

Oppure, se vi siete accorti che questa è un'istanza di un problema risolto precedentemente, potreste avere scritto:

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

Si suppone che abbiate svolto l'Esercizio 8.9.

9.5 Debug

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

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

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

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

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

Secondo un leggendario informatico:

Il collaudo di un programma può essere usato per dimostrare la presenza di bug, ma mai per dimostrarne l'assenza!

— Edsger W. Dijkstra

9.6 Glossario

oggetto file: Un valore che rappresenta un file aperto.

riconoscimento del problema: Modo di risolvere un problema esprimendolo come un'istanza di un problema precedentemente risolto.

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

9.7 Esercizi

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

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

Scrivete un programma per trovare la parola. Soluzione: <http://thinkpython.com/code/cartalk1.py>.

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

“L’altro giorno stavo guidando in autostrada e guardai il mio contachilometri. È a sei cifre, come la maggior parte dei contachilometri, e mostra solo chilometri interi. Se la mia macchina, per esempio, avesse 300.000 km, vedrei 3-0-0-0-0-0.”

“Quello che vidi quel giorno era interessante. Notai che le ultime 4 cifre erano palindromo, cioè si potevano leggere in modo identico sia da sinistra a destra che viceversa. Per esempio 5-4-4-5 è palindromo, per cui il contachilometri avrebbe potuto essere 3-1-5-4-4-5”

“Un chilometro dopo, gli ultimi 5 numeri erano palindromi. Per esempio potrei aver letto 3-6-5-4-5-6. Un altro chilometro dopo, le 4 cifre di mezzo erano palindromo. E tenetevi forte: un altro chilometro dopo tutte e 6 erano palindromo!”

“La domanda è: quanto segnava il contachilometri la prima volta che guardai?”

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

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

“Di recente ho fatto visita a mia madre, e ci siamo accorti che le due cifre che compongono la mia età, invertite, formano la sua. Per esempio, se lei avesse 73 anni, io ne avrei 37. Ci siamo domandati quanto spesso succedesse questo negli anni, ma poi abbiamo divagato su altri discorsi senza darci una risposta.”

“Tornato a casa, ho calcolato che le cifre delle nostre età sono state sinora invertibili per sei volte. Ho calcolato anche che se fossimo fortunati succederebbe ancora tra pochi anni, e se fossimo veramente fortunati succederebbe un'altra volta ancora. In altre parole, potrebbe succedere per 8 volte in tutto. La domanda è: quanti anni ho io in questo momento?”

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

Soluzione: <http://thinkpython.com/code/cartalk3.py>.

Capitolo 10

Liste

10.1 Una lista è una sequenza

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

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

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

Il primo esempio è una lista di quattro interi, il secondo una lista di tre stringhe. Gli elementi di una stessa lista non devono necessariamente essere tutti dello stesso tipo. La lista che segue contiene una stringa, un numero in virgola mobile, un intero e (meraviglia!) un'altra lista:

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

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

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

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

```
>>> formaggi = ['Cheddar', 'Edam', 'Gouda']
>>> numeri = [17, 123]
>>> vuota = []
>>> print formaggi, numeri, vuota
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

10.2 Le liste sono mutabili

La sintassi per l'accesso agli elementi di una lista è la stessa che abbiamo già visto per i caratteri di una stringa: le parentesi quadre, con un'espressione tra parentesi che specifica l'indice dell'elemento (non dimenticate che gli indici partono da 0!):

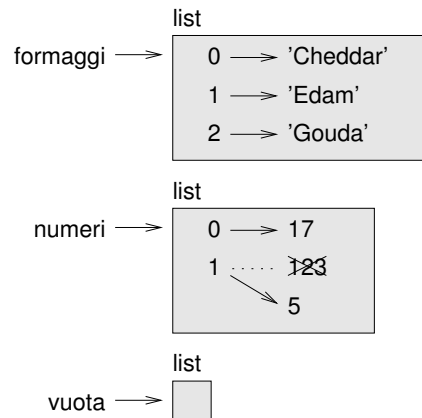


Figura 10.1: Diagramma di stato

```
>>> print formaggi[0]
```

```
Cheddar
```

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

```
>>> numeri = [17, 123]
```

```
>>> numeri[1] = 5
```

```
>>> print numeri
```

```
[17, 5]
```

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

Potete pensare a una lista come una relazione di corrispondenza tra indici ed elementi. Questa relazione è chiamata **mappatura**; ogni indice corrisponde a uno degli elementi. La Figura 10.1 mostra il diagramma di `formaggi`, `numeri` e `vuota`:

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

Gli indici di una lista funzionano nello stesso modo già visto per le stringhe:

- Come indice possiamo usare qualsiasi espressione che produca un intero.
- Se tentate di leggere o modificare un elemento che non esiste, ottenete un messaggio d'errore `IndexError`.
- Se un indice ha valore negativo, il conteggio parte dalla fine della lista.

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

```
>>> formaggi = ['Cheddar', 'Edam', 'Gouda']
```

```
>>> 'Edam' in formaggi
```

```
True
```

```
>>> 'Brie' in formaggi
```

```
False
```


10.3 Attraversamento di una lista

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

```
for formaggio in formaggi:
    print formaggio
```

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

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

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

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

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

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

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

10.4 Operazioni sulle liste

L'operatore `+` concatena delle liste:

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

Similmente, l'operatore `*` ripete una lista per un dato numero di volte:

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

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

10.5 Slicing delle liste

Anche l'operazione di slicing funziona sulle liste:

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

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

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

Dato che le liste sono mutabili, spesso è utile farne una copia prima di eseguire operazioni che le pieghino, le stirino o le mutilino.

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

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

10.6 Metodi delle liste

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

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

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

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

Questo esempio lascia immutata la lista `t2`.

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

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

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

10.7 Mappare, filtrare e ridurre

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

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

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

```
totale += x
```

è equivalente a:

```
totale = totale + x
```

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

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

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

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

Esercizio 10.1. *Scrivete una funzione chiamata somma_nidificata che richieda come parametro una lista nidificata di numeri interi e sommi gli elementi di tutte le liste nidificate.*

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

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

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

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

Esercizio 10.2. *Usate tutte_maiuscole per scrivere una funzione di nome maiuscole_nidif che prenda una lista nidificata di stringhe e restituisca una nuova lista nidificata di stringhe, in lettere maiuscole.*

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

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

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

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

La maggior parte delle operazioni sulle liste possono essere espresse come combinazioni di mappa, filtro e riduzione. Poiché queste operazioni sono frequenti, Python contiene alcune caratteristiche del linguaggio per supportarle, come la funzione predefinita `map` e un costrutto chiamato *list comprehension*.

Esercizio 10.3. *Scrivete una funzione che prenda una lista di numeri e restituisca le somme cumulative. Ovvero, una nuova lista dove l'*i*-esimo elemento è la somma dei primi $i + 1$ elementi della lista di origine. Per esempio, la somma cumulativa di `[1, 2, 3]` è `[1, 3, 6]`.*

10.8 Cancellare elementi

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

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

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

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

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

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

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

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

Per cancellare più di un elemento potete usare `del` con uno slice:

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

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

Esercizio 10.4. *Scrivete una funzione di nome `mediani` che prenda una lista e restituisca una nuova lista che contenga tutti gli elementi esclusi il primo e l'ultimo. Quindi `mediani([1,2,3,4])` deve restituire `[2,3]`.*

Esercizio 10.5. *Scrivete una funzione di nome `tronca` che prenda una lista, la modifichi togliendo il primo e l'ultimo elemento, e restituisca `None`.*

10.9 Liste e stringhe

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

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

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

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

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

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

```
>>> s = 'spam-spam-spam'
>>> delimita = '-'
>>> s.split(delimita)
['spam', 'spam', 'spam']
```

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

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

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

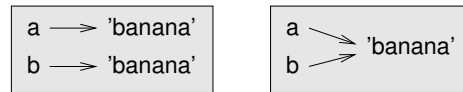


Figura 10.2: Diagramma di stato.

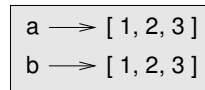


Figura 10.3: Diagramma di stato.

10.10 Oggetti e valori

Se eseguiamo queste istruzioni di assegnazione:

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

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

In un caso, a e b si riferiscono a due oggetti diversi che hanno lo stesso valore. Nel secondo, si riferiscono allo stesso oggetto.

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

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

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

Ma se create due liste, ottenete due oggetti distinti:

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

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

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

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

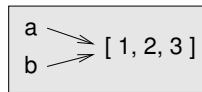


Figura 10.4: Diagramma di stato.

10.11 Alias

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

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

Il diagramma di stato è quello in Figura 10.4.

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

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

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

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

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

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

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

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

10.12 Liste come argomenti

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

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

Vediamo come si usa:

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

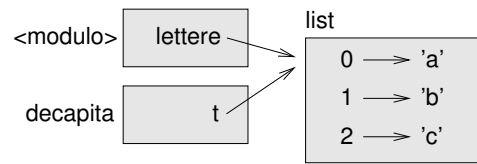


Figura 10.5: Diagramma di stack.

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

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

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

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

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

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

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

L'operatore di slicing crea una nuova lista e l'assegnazione fa in modo che `t` si riferisca ad essa, ma nulla di tutto ciò ha effetti sulla lista passata come argomento.

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

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

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

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

10.13 Debug

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

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

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

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

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

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

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

Prima di usare i metodi delle liste e gli operatori, leggete attentamente la documentazione e fate una prova in modalità interattiva. Metodi e operatori condivisi tra liste e altre sequenze (come le stringhe) sono descritti sul sito <http://docs.python.org/2/library/stdtypes.html#typeseq>. Quelli applicabili solo alle sequenze mutabili si trovano invece all'indirizzo <http://docs.python.org/2/library/stdtypes.html#typeseq-mutable>.

2. Scegliete un costrutto e usate sempre quello.

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

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

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

Mentre queste sono sbagliate:

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

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

3. Fate copie per evitare gli alias.

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

```
orig = t[:]
t.sort()
```

In questo esempio potete anche usare la funzione predefinita `sorted`, che restituisce una nuova lista ordinata e lascia intatta quella di origine. Ma in questo caso evitate di usare `sorted` come nome di variabile!

10.14 Glossario

lista: Una sequenza di valori.

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

indice: Valore intero che indica un elemento all'interno di una lista.

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

attraversamento di una lista: Accesso in sequenza di tutti gli elementi di una lista.

mappatura: Una relazione in cui ogni elemento di un insieme corrisponde ad un elemento di un altro insieme. Ad esempio, una lista è una mappatura da indici a elementi.

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

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

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

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

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

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

equivalente: Avente lo stesso valore.

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

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

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

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

10.15 Esercizi

Esercizio 10.6. *Scrivete una funzione di nome `ordinata` che prenda una lista come parametro e restituisca `True` se la lista è ordinata in senso crescente, `False` altrimenti. Supponete (come precondizione) che gli elementi della lista siano confrontabili con gli operatori relazionali `<`, `>`, ecc.*

Per esempio, `ordinata([1,2,2])` deve restituire `True` e `ordinata(['b','a'])` invece `False`.

Esercizio 10.7. *Due parole sono anagrammi se potete ottenerle riordinando le lettere di cui sono composte. Scrivete una funzione di nome `anagramma` che riceva due stringhe e restituisca `True` se sono anagrammi.*

Esercizio 10.8. Il cosiddetto “Paradosso del compleanno”:

1. Scrivete una funzione di nome `ha_duplicati` che richieda una lista e restituisca `True` se contiene elementi che compaiono più di una volta. Non deve modificare la lista di origine.
2. Se in una classe ci sono 23 studenti, quante probabilità ci sono che due di loro compiano gli anni lo stesso giorno? Potete stimare questa probabilità generando alcuni campioni a caso di 23 date e controllando le corrispondenze. Suggerimento: per generare date in modo casuale usate la funzione `randint` nel modulo `random`.

Potete saperne di più su questo problema sul sito http://it.wikipedia.org/wiki/Paradosso_del_compleanno, e scaricare la mia soluzione da <http://thinkpython.com/code/birthday.py>.

Esercizio 10.9. Scrivete una funzione di nome `rimuovi_duplicati` che richieda una lista come parametro e restituisca una nuova lista che contiene solo valori non ripetuti. Suggerimento: non è necessario che siano nello stesso ordine.

Esercizio 10.10. Scrivete una funzione che legga il file `words.txt` e crei una lista in cui ogni parola è un elemento. Scrivete due versioni della funzione, una che usi il metodo `append` e una il costrutto `t = t + [x]`. Quale richiede più tempo di esecuzione? Perché?

Suggerimento: Usate il modulo `time` per misurare il tempo impiegato. Soluzione: <http://thinkpython.com/code/wordlist.py>.

Esercizio 10.11. Per controllare se una parola è contenuta in un elenco, è possibile usare l’operatore `in`, ma è un metodo lento, perché ricerca le parole seguendo il loro ordine.

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

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

Scrivete una funzione di nome `bisezione` che richieda una lista ordinata e un valore da ricercare, e restituisca l’indice di quel valore, se fa parte della lista, oppure `None` se non esiste.

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

Esercizio 10.12. Una coppia di parole è “bifronte” se l’una si legge nel verso opposto dell’altra. Scrivete un programma che trovi tutte le parole bifronti nella lista di parole. Soluzione: http://thinkpython.com/code/reverse_pair.py.

Esercizio 10.13. Due parole si “incastrano” se, prendendo le loro lettere alternativamente dall’una e dall’altra, si forma una nuova parola. Per esempio, le parole inglesi “shoe” and “cold” incastrandosi formano “schooled”.

1. Scrivete un programma che trovi tutte le coppie di parole che possono incastrarsi. Suggerimento: non elaborate tutte le coppie!

2. Riuscite a trovare dei gruppi di tre parole che possono incastrarsi tra loro? Cioè, tre parole da cui, prendendo le lettere una ad una alternativamente, nell'ordine, si formi una nuova parola? (Es. "ace", "bus" e "as" danno "abacuses")

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

Capitolo 11

Dizionari

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

Si può pensare ad un dizionario come ad una relazione di corrispondenza (mappatura) da un insieme di indici, chiamati **chiavi**, a un insieme di valori. Ad ogni chiave corrisponde un valore. L'associazione tra una chiave e un valore è detta **coppia chiave-valore** o anche **elemento**.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Per controllare invece se qualcosa compare come valore, potete usare il metodo `values`, che restituisce i valori sotto forma di lista, e quindi usare l'operatore `in`:

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

L'operatore `in` utilizza algoritmi diversi per liste e dizionari. Per le prime, usa un algoritmo di ricerca, come nel Paragrafo 8.6. Se la lista si allunga, anche il tempo di ricerca si allunga in proporzione. Per i secondi, Python usa un algoritmo chiamato **tabella hash** che ha notevoli proprietà: l'operatore `in` impiega sempre circa lo stesso tempo, indipendentemente da quanti elementi contiene il dizionario. Non mi dilungo a spiegare come ciò sia possibile, ma chi lo desidera può saperne di più sul sito http://it.wikipedia.org/wiki/Hash_table.

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

Se avete svolto l'Esercizio 10.11, potete confrontare la velocità di questa implementazione con l'operatore `in` applicato alla lista e la ricerca binaria.

11.1 Il dizionario come gruppo di contatori

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

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

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

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

Ecco come potrebbe essere scritto il codice:

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

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

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

Vediamo come funziona:

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

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

Esercizio 11.2. *I dizionari hanno un metodo `get` che richiede una chiave e un valore predefinito. Se la chiave è presente nel dizionario, `get` restituisce il suo valore corrispondente, altrimenti restituisce il valore predefinito. Per esempio:*

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

```
>>> h.get('b', 0)
0
```

Usate `get` per scrivere istogramma in modo più compatto. Dovreste riuscire a fare a meno dell'istruzione `if`.

11.2 Cicli e dizionari

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

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

Ecco come risulta l'output:

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

Di nuovo, le chiavi sono alla rinfusa.

Esercizio 11.3. *I dizionari hanno un metodo `keys` che restituisce le chiavi, senza seguire un ordine particolare, sotto forma di lista.*

Modificate `stampa_isto` in modo da stampare le chiavi, con i corrispondenti valori, in ordine alfabetico.

11.3 Lookup inverso

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

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

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

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


Questa funzione è un altro esempio di schema di ricerca, ma usa un'istruzione che non abbiamo mai visto prima, `raise`. L'istruzione `raise` solleva un'eccezione; in questo caso genera un errore `ValueError`, che di solito indica che c'è qualcosa di sbagliato nel valore di un parametro.

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

Ecco un esempio di lookup inverso riuscito:

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

E di uno fallito:

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

Quando generate un errore, il risultato è lo stesso di quando lo genera Python: viene stampato un traceback con un messaggio di errore.

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

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

In Python 3, la sintassi è cambiata:

```
>>> raise ValueError('il valore non compare nel dizionario')
```

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

Esercizio 11.4. *Modificate `inverso_lookup` in modo che crei e restituisca una lista di tutte le chiavi che corrispondono a `v`, oppure una lista vuota se non ce ne sono.*

11.4 Dizionari e liste

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

Ecco una funzione che inverte un dizionario:

```
def inverti_diz(d):
    inverso = dict()
```

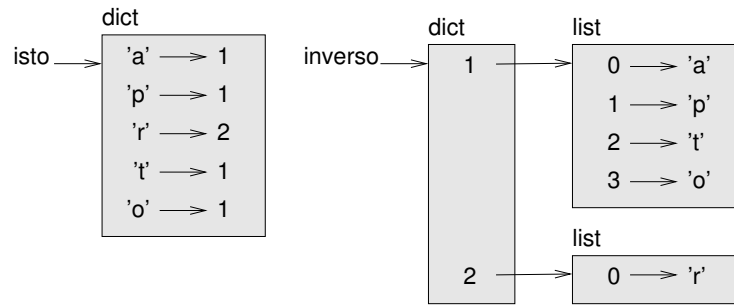


Figura 11.1: Diagramma di stato.

```

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

```

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

Ecco un esempio:

```

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

```

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

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

```

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

```

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



Figura 11.2: Grafico di chiamata.

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

Questo sistema funziona se le chiavi sono immutabili; ma se sono mutabili, come le liste, succedono disastri. Per esempio, nel creare una coppia chiave-valore, Python fa l'hash della chiave e la immagazzina nello spazio corrispondente. Se modificate la chiave e quindi viene nuovamente calcolato l'hash, si collocherebbe in un altro spazio. In quel caso potreste avere due voci della stessa chiave, oppure non riuscire a trovare una chiave. In ogni caso il dizionario non funzionerà correttamente.

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

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

Esercizio 11.5. Leggete la documentazione del metodo dei dizionari `setdefault` e usatelo per scrivere una versione più concisa di `inverti_diz`. Soluzione: http://thinkpython.com/code/invert_dict.py.

11.5 Memoizzazione

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

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

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

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

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

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

def fibonacci(n):
    if n in memo:
        return memo[n]

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

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

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

Esercizio 11.6. *Eseguite questa versione di `fibonacci` e l'originale con un ventaglio di parametri, e confrontate i loro tempi di esecuzione.*

Esercizio 11.7. *Applicate la memoizzazione alla funzione di Ackermann dell'Esercizio 6.5 e provate a vedere se questa tecnica rende possibile il calcolo della funzione con argomenti più grandi. Suggerimento: no. Soluzione: http://thinkpython.com/code/ackermann_memo.py.*

11.6 Variabili globali

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

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

```
verbose = True

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

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

```
stata_chiamata = False

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

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

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

```
stata_chiamata = False

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

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

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

```
conta = 0

def example3():
    conta = conta + 1          # SBAGLIATO
```

Se lo eseguite, ottenete:

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

Python presume che `conta` nella funzione sia locale, il che comporta che state usando la variabile prima di averla inizializzata. La soluzione è ancora quella di dichiarare `count` globale.

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

Se il valore globale è mutabile, potete modificarlo anche senza dichiararlo:

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

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

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

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

11.7 Interi lunghi

Se calcolate `fibonacci(50)`, ottenete:

```
>>> fibonacci(50)
12586269025L
```

La `L` finale indica che il risultato è un intero lungo, o di tipo `long`. In Python 3, `long` non esiste più; tutti gli interi sono di tipo `int`, anche quelli molto grandi.

I valori di tipo `int` hanno un intervallo limitato; gli interi `long` possono essere grandi a piacere, ma più grandi sono più richiedono spazio in memoria e tempo.

Gli operatori matematici funzionano con gli interi lunghi, come pure le funzioni del modulo `math` e in genere tutto il codice che funziona con gli `int` funzionerà anche con i `long`.

Tutte le volte che il risultato di un calcolo è troppo grande per essere rappresentato da un intero, Python lo converte in un intero lungo:

```
>>> 1000 * 1000
1000000
>>> 100000 * 100000
100000000000L
```

Nel primo caso il risultato è di tipo `int`; nel secondo è `long`.

Esercizio 11.8. *L'elevamento a potenza di grandi interi è alla base dei noti algoritmi di crittografia a chiave pubblica. Leggete la pagina di Wikipedia sull'algoritmo RSA (<http://it.wikipedia.org/wiki/RSA>) e scrivete una funzione per codificare e decodificare i messaggi.*

11.8 Debug

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

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

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

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

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

Scrivere controlli automatici: Talvolta è utile scrivere del codice per controllare automaticamente gli errori. Per esempio, se dovete calcolare la media di una lista di numeri,

potete controllare che il risultato non sia maggiore dell'elemento più grande della lista e non sia minore del più piccolo. Questo è detto "controllo di congruenza" perché mira a trovare i risultati "incongruenti".

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

Stampare gli output in bella copia: Una buona presentazione dei risultati di debug rende più facile trovare un errore. Abbiamo visto un esempio nel Paragrafo 6.9. Il modulo `pprint` contiene la funzione `pprint` che mostra i tipi predefiniti in un formato più leggibile.

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

11.9 Glossario

dizionario: Una mappatura da un insieme di chiavi ai loro valori corrispondenti.

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

elemento: Altro nome della coppia chiave-valore.

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

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

implementazione: Un modo per effettuare un'elaborazione.

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

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

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

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

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

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

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

istogramma: Gruppo di contatori.

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

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

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

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

11.10 Esercizi

Esercizio 11.9. Se avete svolto l'Esercizio 10.8, avete già una funzione di nome `ha_duplicati` che richiede come parametro una lista e restituisce `True` se ci sono oggetti ripetuti all'interno della lista.

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

Esercizio 11.10. Due parole sono "ruotabili" se potete far ruotare le lettere dell'una per ottenere l'altra (vedere `ruota_parola` nell'Esercizio 8.12).

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

Esercizio 11.11. Ecco un altro quesito tratto da Car Talk (<http://www.cartalk.com/content/puzzlers>):

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

"Facciamo un esempio che non funziona del tutto. Prendiamo la parola 'wrack'; togliendo la prima lettera resta 'rack', che è un'altra parola ma è un perfetto omofono. Se però rimettete la prima lettera e togliete la seconda, ottenete 'wack' che pure esiste ma non è un omofono delle altre due parole."

"Esiste comunque almeno una parola, che Dan e noi conosciamo, che dà due parole omofone di quattro lettere, sia che togliate la prima oppure la seconda lettera."

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

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

Scrivete un programma che elenchi tutte le parole che risolvono il quesito. Soluzione: <http://thinkpython.com/code/homophone.py>.

Capitolo 12

Tuple

12.1 Le tuple sono immutabili

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

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

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

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

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

Per creare una tupla con un singolo elemento dobbiamo aggiungere la virgola finale dopo l'elemento:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Anche se non potete modificare direttamente gli elementi di una tupla, potete sempre sostituirla con una sua copia modificata:

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

12.2 Assegnazione di tupla

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

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

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

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

La parte sinistra dell'assegnazione è una tupla di variabili; la parte destra una tupla di valori o espressioni. Ogni valore è assegnato alla rispettiva variabile. Tutte le espressioni sulla destra vengono valutate prima della rispettiva assegnazione.

Ovviamente il numero di variabili sulla sinistra deve corrispondere al numero di valori sulla destra:

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

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

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

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

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

12.3 Tuple come valori di ritorno

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

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

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

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

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

Ecco un esempio di funzione che restituisce una tupla:

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

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

12.4 Tuple di argomenti a lunghezza variabile

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

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

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

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

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

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

Ma se esplodete la tupla, funziona:

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

Esercizio 12.1. Molte funzioni predefinite possono usare le tuple di argomenti a lunghezza variabile. Ad esempio, `max` e `min` ricevono un numero qualunque di argomenti:

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

Ma con `sum` non funziona.

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

Scrivete una funzione di nome `sommatutto` che riceva un numero di argomenti a piacere e ne restituisca la somma.

12.5 Liste e tuple

`zip` è una funzione predefinita che riceve due o più sequenze e le abbina in una lista di tuple, dove ciascuna tupla contiene un elemento di ciascuna sequenza. In Python 3, `zip` restituisce un iteratore di tuple, ma per la maggior parte degli scopi un iteratore si comporta come una lista.

Questo esempio abbina una stringa e una lista:

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

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

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

```
>>> zip('Anna', 'Edi')
[('A', 'E'), ('n', 'd'), ('n', 'i')]
```

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

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

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

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

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

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

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

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

Il risultato del ciclo è ancora:

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

12.6 Dizionari e tuple

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

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

Come di consueto nei dizionari, gli elementi non sono in un ordine particolare. In Python 3, `items` restituisce un iteratore, ma per la maggior parte degli scopi gli iteratori si comportano come le liste.

Per altro verso, potete usare una lista di tuple per inizializzare un nuovo dizionario:

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

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

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

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

Mettendo assieme `items`, assegnazione di tupla e `for`, ottenete un costrutto per attraversare chiavi e valori di un dizionario:

```
for chiave, valore in d.items():
    print valore, chiave
```

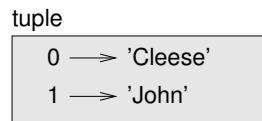


Figura 12.1: Diagramma di stato.

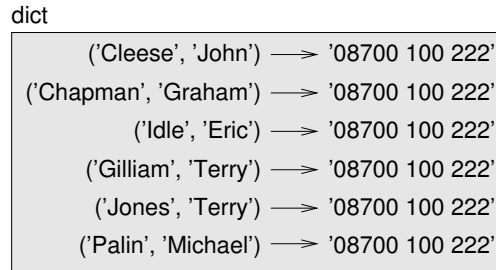


Figura 12.2: Diagramma di stato.

L'output di questo ciclo è nuovamente:

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

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

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

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

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

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

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

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

Qui le tuple sono mostrate usando la sintassi di Python come abbreviazione grafica.

Il numero di telefono nel diagramma è quello dei reclami della BBC, per cui vi prego, non chiamatelo.

12.7 Confrontare tuple

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

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

Il metodo `sort` funziona nello stesso modo. Ordina innanzitutto basandosi sul primo elemento, ma in caso di uguaglianza ordina in base al secondo elemento, e così via.

Questa caratteristica conduce a uno schema di calcolo detto **DSU** [acronimo dell'inglese *Decorate-Sort-Undecorate*, NdT] che:

Decora una sequenza, creando una lista di tuple contenenti una o più chiavi di ordinamento che precedono gli elementi della sequenza,

Ordina la lista di tuple e

Rimuove la decorazione, estraendo gli elementi ordinati della sequenza.

Per esempio, supponiamo di avere una lista di parole e di volerle ordinare dalla più lunga alla più corta:

```
def ordina_lungh(parole):
    t = []
    for parola in parole:
        t.append((len(parola), parola))

    t.sort(reverse=True)

    res = []
    for lunghezza, parola in t:
        res.append(parola)
    return res
```

Il primo ciclo crea una lista di tuple, dove ogni tupla contiene una parola preceduta dalla sua lunghezza.

`sort` confronta dapprima il primo elemento, la lunghezza, e prende in considerazione il secondo, la parola, solo per risolvere i casi di parità. L'argomento con nome `reverse=True` dice a `sort` di procedere in ordine decrescente.

Il secondo ciclo attraversa la lista di tuple ordinata, e crea una nuova lista delle parole in ordine decrescente.

Esercizio 12.2. *In questo esempio, i casi di pari lunghezza vengono risolti confrontando l'ordine delle parole, per cui parole di uguale lunghezza compaiono in ordine alfabetico inverso. Per altre applicazioni, si può preferire un criterio casuale. Modificate questo esempio in modo che le parole di uguale lunghezza compaiano in ordine casuale. Suggerimento: vedere la funzione `random` nel modulo `random`. Soluzione: http://thinkpython.com/code/unstable_sort.py.*

12.8 Sequenze di sequenze

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

In molti casi, i diversi tipi di sequenze (stringhe, liste, tuple) possono essere intercambiabili. E allora, come e perché scegliere di usarne una piuttosto di un'altra?

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

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

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

Siccome le tuple sono immutabili, non possiedono metodi come `sort` e `reverse`, che modificano delle liste esistenti. Però Python contiene le funzioni `sorted` e `reversed`, che richiedono come parametro qualsiasi sequenza e restituiscono una nuova lista con gli stessi elementi in ordine diverso.

12.9 Debug

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

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

Questo è il risultato per una lista semplice:

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> print structshape(t)
list of 3 int
```


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

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

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

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

Ecco una lista di tuple:

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

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

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

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

12.10 Glossario

tupla: Una sequenza di elementi immutabile.

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

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

esplosione: L’operazione di trattare una sequenza come una lista di argomenti.

DSU: Acronimo di “Decorate-Sort-Undecorate”, uno schema che comporta la costruzione di una lista di tuple, il suo ordinamento, e l’estrazione di parte dei risultati.

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

formato (di una struttura di dati): Un riassunto di tipo, dimensione e composizione di una struttura di dati.

12.11 Esercizi

Esercizio 12.3. *Scrivete una funzione di nome piu_frequente che riceva una stringa e stampi le lettere in ordine di frequenza decrescente. Trovate delle frasi di esempio in diverse lingue e osservate come varia la frequenza delle lettere. Confrontate i vostri risultati con le tabelle del sito http://en.wikipedia.org/wiki/Letter_frequencies. Soluzione: http://thinkpython.com/code/most_frequent.py.*

Esercizio 12.4. Ancora anagrammi!

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

Un esempio di come si può presentare il risultato:

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

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

2. Modificate il programma in modo che stampi l'insieme di anagrammi più numeroso per primo, seguito dal secondo, e così via.
3. Nel gioco Scarabeo, fate un "en-plein" quando giocate tutte le sette lettere sul vostro leggio formando, insieme a una lettera sul tavolo, una parola di otto lettere. Con quale gruppo di 8 lettere si può fare un "en-plein" con maggior probabilità? Suggerimento: il gruppo dà sette combinazioni.

Soluzione: http://thinkpython.com/code/anagram_sets.py.

Esercizio 12.5. Si ha una metatesi quando una parola si può ottenere scambiando due lettere di un'altra parola, per esempio, "conversa" e "conserva". Scrivete un programma che trova tutte le coppie con metatesi nel dizionario. Suggerimento: non provate tutte le possibili coppie di parole e non provate tutti i possibili scambi. Soluzione: <http://thinkpython.com/code/metathesis.py>. Fonte: Esercizio suggerito da un esempio nel sito <http://puzzlers.org>.

Esercizio 12.6. Ed ecco un altro quesito di Car Talk: (<http://www.cartalk.com/content/puzzlers>):

Qual è la più lunga parola inglese che rimane una parola valida se le togliete una lettera alla volta? Le lettere possono essere rimosse sia agli estremi o in mezzo, ma senza spostare le lettere rimanenti. Ogni volta che togliete una lettera, ottenete un'altra parola inglese. Se andate avanti, ottenete un'altra parola. Ora, voglio sapere qual è la parola più lunga possibile e quante lettere ha.

Vi faccio un piccolo esempio: Sprite. Partite da sprite, togliete una lettera, una interna, come la r e resta la parola spite, poi togliete la e finale e avete spit, togliamo la s e resta pit, poi it, infine I.

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

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

1. Potete scrivere una funzione che prenda una parola e calcoli una lista di tutte le parole che si possono formare togliendo una lettera. Queste sono le "figlie" della parola.
2. Ricorsivamente, una parola è riducibile se qualcuna delle sue figlie è a sua volta riducibile. Come caso base, potete considerare riducibile la stringa vuota.

3. *L'elenco di parole che ho fornito, `words.txt`, non contiene parole di una lettera. Potreste quindi aggiungere "I", "a", e la stringa vuota.*
4. *Per migliorare le prestazioni del programma, potete memoizzare le parole che sono risultate riducibili.*

Soluzione: <http://thinkpython.com/code/reducible.py>.

Capitolo 13

Esercitazione: Scelta della struttura di dati

13.1 Analisi di frequenza delle parole

Come al solito, tentate almeno di risolvere i seguenti esercizi prima di leggere le mie risoluzioni.

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

Suggerimento: il modulo `string` fornisce delle stringhe chiamate `whitespace`, che contiene i caratteri spaziatori come spazio, tabulazione, a capo ecc., e `punctuation` che contiene i caratteri di punteggiatura. Vediamo se Python ce lo conferma:

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

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

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

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

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

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

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

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

13.2 Numeri casuali

A parità di dati di partenza, la maggior parte dei programmi fanno la stessa cosa ogni volta che vengono eseguiti, e per questo motivo sono detti deterministici. Di solito il determinismo è una buona cosa, in quanto dagli stessi dati in ingresso è logico aspettarsi sempre lo stesso risultato. Per alcune applicazioni, invece, serve che l'esecuzione sia imprevedibile: i videogiochi sono un esempio lampante, ma ce ne sono tanti altri.

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

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

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

```
import random

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

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

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

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

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

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

Esercizio 13.5. *Scrivete una funzione di nome `estrai_da_isto` che prenda un istogramma come definito nel Paragrafo 11.1 e restituisca un valore casuale dall'istogramma, scelto in modo che la probabilità sia proporzionale alla frequenza. Per esempio, dato questo istogramma:*

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

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

13.3 Istogramma di parole

Provate a risolvere gli esercizi precedenti prima di procedere oltre. Le soluzioni sono scaricabili da http://thinkpython.com/code/analyze_book.py. Vi servirà anche <http://thinkpython.com/code/emma.txt>.

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

```
import string

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

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

    for parola in riga.split():
        parola = parola.strip(string.punctuation + string.whitespace)
        parola = parola.lower()

        isto[parola] = isto.get(parola, 0) + 1

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

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

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

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

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

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

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

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

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

Ed ecco del codice per stampare i risultati:

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

E i relativi risultati:

Numero totale di parole: 161080

Numero di parole diverse: 7214

13.4 Parole più comuni

Per trovare le parole più comuni, possiamo applicare lo schema DSU; `piu_comuni` prende un istogramma e restituisce una lista di tuple parola-frequenza, ordinate in ordine di frequenza decrescente:

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

Ecco un ciclo che stampa le dieci parole più comuni:

```
t = piu_comuni(hist)  
print "Le parole piu' comuni sono:"  
for freq, parola in t[0:10]:  
    print parola, '\t', freq
```

E questi sono i risultati nel caso di *Emma*:

```
Le parole piu' comuni sono:  
to 5242  
the 5205  
and 4897  
of 4295  
i 3191  
a 3130  
it 2529  
her 2483  
was 2400  
she 2364
```

13.5 Parametri opzionali

Abbiamo visto funzioni predefinite e metodi che ricevono un numero variabile di argomenti. È possibile anche scrivere funzioni definite dall'utente con degli argomenti opzionali. Ad esempio, questa è una funzione che stampa le parole più comuni in un istogramma:


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

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

Se passate un solo argomento:

```
stampa_piu_comuni(isto)
```

num assume il valore predefinito. Se ne passate due:

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

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

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

13.6 Sottrazione di dizionari

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

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

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

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

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

```
print "Le parole del libro che non si trovano nell'elenco sono:"
for parola in diff.keys():
    print parola,
```

Ecco alcuni dei risultati per *Emma*:

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

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

Esercizio 13.6. *Python ha una struttura di dati chiamata `set`, o insieme, che fornisce molte operazioni comuni sugli insiemi. Leggetene la documentazione sul sito <http://docs.python.org/2/library/stdtypes.html#types-set> e scrivete un programma che usi la sottrazione di insiemi per trovare le parole del libro che non sono nell'elenco. Soluzione: http://thinkpython.com/code/analyze_book2.py.*

13.7 Parole a caso

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

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

    return random.choice(t)
```

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

Esercizio 13.7. *Questo algoritmo funziona, ma non è molto efficiente: ogni volta che estraete una parola, ricostruisce la lista, che è grande come il libro originale. Un ovvio miglioramento è di costruire la lista una sola volta e poi fare estrazioni multiple, ma la lista è ancora grande.*

Un'alternativa è:

1. Usare `keys` per ottenere una lista delle parole del libro.
2. Costruire una lista che contiene la somma cumulativa delle frequenze delle parole (vedere l'Esercizio 10.3). L'ultimo elemento della lista è il numero totale delle parole nel libro, `n`.
3. Scegliere un numero a caso da 1 a `n`. Usare una ricerca binaria (vedere l'Esercizio 10.11) per trovare l'indice dove il numero casuale si inserirebbe nella somma cumulativa.
4. Usare l'indice per trovare la parola corrispondente nella lista di parole.

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

13.8 Analisi di Markov

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

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

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

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

Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D'you see?

But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?

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

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

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

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

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

Esercizio 13.8. Analisi di Markov:

1. Scrivete un programma che legga un testo da un file ed esegua l'analisi di Markov. Il risultato dovrebbe essere un dizionario che fa corrispondere i prefissi a una raccolta di possibili suffissi. La raccolta può essere una lista, tupla o dizionario: a voi valutare la scelta più appropriata. Potete testare il vostro programma con una lunghezza del prefisso di due parole, ma dovrete scrivere il programma in modo da poter provare facilmente anche lunghezze superiori.
2. Aggiungete una funzione al programma precedente per generare un testo casuale basato sull'analisi di Markov. Ecco un esempio tratto da *Emma* con prefisso di lunghezza 2:

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

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

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

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

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

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

13.9 Strutture di dati

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

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

Ok, l’ultima è facile: il solo tipo in grado di mappare che abbiamo visto è il dizionario, quindi la scelta è scontata.

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

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

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

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

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

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

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

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

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

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

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

13.10 Debug

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

leggere: Esaminate il vostro codice, rileggetevelo e controllate che dica esattamente quello che voi intendete dire.

eseguire: Sperimentate facendo modifiche ed eseguendo le diverse versioni. Spesso, se nel programma visualizzate la cosa giusta al posto giusto, il problema diventa evidente, anche se talvolta dovrete spendere un po' di tempo per inserire qualche "impalcatura".

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

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

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

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

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

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

Prendersi una pausa aiuta a pensare. Come pure parlarne. Spiegando il problema a qualcun altro (o anche a voi stessi), talvolta si trova la risposta ancora prima di finire la domanda.

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

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

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

13.11 Glossario

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

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

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

sovrascrivere: Sostituire un valore di default con un argomento.

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

13.12 Esercizi

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

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

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

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

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

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

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

Soluzione: <http://thinkpython.com/code/zipf.py>. Per i grafici potete eventualmente installare matplotlib (vedere <http://matplotlib.sourceforge.net/>).

Capitolo 14

File

14.1 Persistenza

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

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

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

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

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

14.2 Lettura e scrittura

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

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

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

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

Il metodo `write` inserisce i dati nel file.

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

Come sempre, l'oggetto `file` tiene traccia di dove si trova, e se invocate ancora il metodo `write` aggiunge i dati in coda al file.

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

Quando avete finito di scrivere, dovete chiudere il file.

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

14.3 L'operatore di formato

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

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

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

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

Per esempio, la sequenza di formato `'%d'` significa che il secondo operando dovrebbe essere nel formato di numero intero (`d` sta per "decimale", nel senso di sistema di numerazione):

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

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

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

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

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

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

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

Il numero degli elementi nella tupla deve naturalmente essere corrispondente a quello delle sequenze di formato nella stringa, ed i tipi degli elementi devono corrispondere a quelli delle sequenze di formato:

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

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

L'operatore di formato è potente, ma può essere difficile da usare. Per saperne di più: <http://docs.python.org/2/library/stdtypes.html#string-formatting>.

14.4 Nomi di file e percorsi

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

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

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

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

Una stringa come `cwd` che localizza un file è chiamata **percorso**. Si distinguono un **percorso relativo** che parte dalla directory attuale e un **percorso assoluto** che parte dalla directory principale del file system.

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

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

`os.path.exists` controlla se un file o una cartella esistono:

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

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

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('musica')
True
```

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

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

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

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

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

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

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

Esercizio 14.1. Il modulo `os` contiene una funzione di nome `walk` che è simile a questa ma più versatile. Leggetene la documentazione e usatela per stampare i nomi dei file di una data directory e delle sue sottodirectory.

Soluzione: <http://thinkpython.com/code/walk.py>.

14.5 Gestire le eccezioni

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

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

Se non avete il permesso di accedere al file:

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

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

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

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

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

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

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

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

Esercizio 14.2. *Scrivete una funzione di nome `sed` che richieda come argomenti una stringa modello, una stringa di sostituzione, e due nomi di file. La funzione deve leggere il primo file e scriverne il contenuto nel secondo file (creandolo se necessario). Se la stringa modello compare da qualche parte nel testo del file, deve sostituirla con la seconda stringa.*

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

14.6 Database

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

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

Un database si apre in modo simile agli altri file:

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

La modalità `'c'` significa che il database deve essere creato se non esiste già. Il risultato è un oggetto database che può essere utilizzato (per la maggior parte delle operazioni) come un dizionario. Se create un nuovo elemento, `anydbm` aggiorna il file di database.

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

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

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

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

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

Molti metodi dei dizionari, come `keys` e `items`, funzionano anche con gli oggetti database, come pure l'iterazione con un'istruzione `for`.

```
for chiave in db:
    print chiave
```

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

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

14.7 Pickling

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

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

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

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
'(lp0\nI1\naI2\naI3\na.'
```

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

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

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

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

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

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

Esercizio 14.3. *Se avete scaricato la mia soluzione dell'Esercizio 12.4 dal sito http://thinkpython.com/code/anagram_sets.py, avrete visto che crea un dizionario che fa corrispondere una stringa ordinata di lettere alla lista di parole che possono essere scritte con quelle lettere. Per esempio, 'opst' corrisponde alla lista ['opts', 'post', 'pots', 'spot', 'stop', 'tops'].*

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

14.8 Pipe

Molti sistemi operativi forniscono un'interfaccia a riga di comando, nota anche come **shell**. Le shell sono dotate di comandi per spostarsi nel file system e per lanciare le applicazio-

ni. Per esempio, in UNIX potete cambiare directory con il comando `cd`, visualizzarne il contenuto con `ls`, e lanciare un web browser scrivendone il nome, per esempio `firefox`.

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

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

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

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

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

Quando avete finito, chiudete il pipe come fosse un file:

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

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

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

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

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

Esercizio 14.4. *In una grande raccolta di file MP3 possono esserci più copie della stessa canzone, messe in cartelle diverse o con nomi di file differenti. Scopo di questo esercizio è di ricercare i duplicati.*

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

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

2. Per riconoscere i duplicati, potete usare `md5sum` per calcolare la “checksum” di ogni file. Se due file hanno la stessa checksum, significa che con ogni probabilità hanno lo stesso contenuto.
3. Per effettuare un doppio controllo, usate il comando Unix `diff`.

Soluzione: http://thinkpython.com/code/find_duplicates.py.

14.9 Scrivere moduli

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

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

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

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

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

Ora avete un oggetto modulo `wc`:

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

Che fornisce una funzione di nome `contarighe`:

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

Ecco come scrivere moduli in Python.

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

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

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

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

Esercizio 14.5. Scrivete questo esempio in un file di nome `wc.py` ed eseguitelo come script. Poi avviate l'interprete e scrivete `import wc`. Che valore ha `__name__` quando il modulo viene importato?

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

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

14.10 Debug

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

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

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

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

Questa funzione può quindi aiutare nel debug.

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

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

14.11 Glossario

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

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

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

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

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

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

percorso: Stringa che localizza un file.

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

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

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

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

14.12 Esercizi

Esercizio 14.6. Il modulo `urllib` contiene dei metodi per manipolare indirizzi web e scaricare informazioni da Internet. L'esempio seguente scarica e stampa un messaggio segreto da `thinkpython.com`:

```
import urllib

conn = urllib.urlopen('http://thinkpython.com/secret.html')
for riga in conn:
    print riga.strip()
```

Eseguite questo codice e seguite le istruzioni che vi verranno date. Soluzione: `http://thinkpython.com/code/zip_code.py`.

Capitolo 15

Classi e oggetti

Il codice degli esempi di questo capitolo è scaricabile dal sito <http://thinkpython.com/code/Point1.py>; le soluzioni degli esercizi da http://thinkpython.com/code/Point1_soln.py.

15.1 Tipi definiti dall'utente

Abbiamo usato molti dei tipi predefiniti in Python, e ora siamo pronti per crearne uno nuovo: come esempio, creeremo un tipo che chiameremo `Punto`, che rappresenta un punto in un piano cartesiano bidimensionale.

Nella notazione matematica, il punto è denotato da una coppia ordinata di numeri, dette coordinate; le coordinate dei punti sono spesso scritte tra parentesi con una virgola che separa i due valori. Per esempio, $(0,0)$ rappresenta l'origine e (x,y) il punto che si trova a x unità a destra e y unità in alto rispetto all'origine.

Ci sono alcuni modi per rappresentare i punti in Python:

- Memorizzare le coordinate in due variabili separate, x e y .
- Memorizzare le coordinate come elementi di una lista o di una tupla.
- Creare un nuovo tipo che rappresenti i punti come degli oggetti.

L'ultima opzione è un pochino più complicata delle altre, ma ha dei vantaggi che saranno presto chiariti.

Un tipo definito dall'utente è chiamato anche **classe**. Una definizione di classe ha questa sintassi:

```
class Punto(object):  
    """Rappresenta un punto in un piano."""
```

L'intestazione indica che la nuova classe è un `Punto`, che è un tipo di oggetto, `object`, che a sua volta è un tipo predefinito.

Il corpo è una stringa di documentazione che spiega cosa fa la classe. Al suo interno si possono poi definire funzioni e variabili, ma ci arriveremo tra poco.

La definizione di una classe di nome `Punto` crea un oggetto classe.

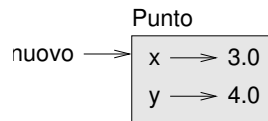


Figura 15.1: Diagramma di oggetto

```
>>> print Punto
<class '__main__.Punto'>
```

Poiché la classe `Punto` è stata definita al livello principale, il suo “cognome e nome” è `__main__.Punto`.

L’oggetto classe è simile ad uno stampo che ci permette di fabbricare degli oggetti. Per creare un nuovo oggetto `Punto`, basta chiamare `Punto` come se fosse una funzione.

```
>>> nuovo = Punto()
>>> print nuovo
<__main__.Punto instance at 0xb7e9d3ac>
```

Il valore di ritorno è un riferimento ad un oggetto `Punto`, che qui abbiamo assegnato alla variabile `nuovo`. La creazione di un nuovo oggetto è detta **istanziamento**, e l’oggetto è un’**istanza** della classe.

Quando stampate un’istanza, Python informa a quale classe appartiene e in quale posizione di memoria è collocata (il prefisso `0x` significa che il numero che segue è in formato esadecimale).

15.2 Attributi

Potete assegnare dei valori ad un’istanza usando la notazione a punto:

```
>>> nuovo.x = 3.0
>>> nuovo.y = 4.0
```

Questa sintassi è simile a quella usata per la selezione di una variabile appartenente ad un modulo, tipo `math.pi` o `string.whitespace`. In questo caso però, stiamo assegnando dei valori a degli elementi di un oggetto, ai quali è stato attribuito un nome (`x` e `y`). Questi elementi sono detti **attributi**.

Il diagramma di stato in Figura 15.1 mostra il risultato delle assegnazioni. Un diagramma di stato che illustra un oggetto e i suoi attributi è detto **diagramma di oggetto**.

La variabile `nuovo` fa riferimento ad un oggetto `Punto` che contiene due attributi, ed ogni attributo fa riferimento ad un numero in virgola mobile.

Potete leggere il valore di un attributo usando la stessa sintassi:

```
>>> print nuovo.y
4.0
>>> x = nuovo.x
>>> print x
3.0
```

L'espressione `nuovo.x` significa: "Vai all'oggetto a cui `nuovo` fa riferimento e prendi il valore di `x`". In questo caso, assegniamo il valore ad una variabile di nome `x`. Non c'è conflitto tra la variabile locale `x` e l'attributo `x`.

Potete usare la notazione a punto all'interno di qualunque espressione, per esempio:

```
>>> print '(%g, %g)' % (nuovo.x, nuovo.y)
(3.0, 4.0)
>>> distanza = math.sqrt(nuovo.x**2 + nuovo.y**2)
>>> print distanza
5.0
```

Potete anche passare un'istanza come argomento, nel modo consueto:

```
def stampa_punto(p):
    print '(%g, %g)' % (p.x, p.y)
```

La funzione `stampa_punto` riceve come argomento un `Punto` e lo visualizza in notazione matematica. Per invocarla, passate `nuovo` come argomento:

```
>>> stampa_punto(nuovo)
(3.0, 4.0)
```

Dentro alla funzione, il parametro `p` è un alias di `nuovo`, quindi se la funzione modifica `p`, anche `nuovo` viene modificato di conseguenza.

Esercizio 15.1. *Scrivete una funzione di nome `distanza_tra_punti` che riceva due `Punti` come argomenti e ne restituisca la distanza.*

15.3 Rettangoli

A volte è abbastanza ovvio stabilire gli attributi necessari ad un oggetto, ma in altre occasioni occorre fare delle scelte. Immaginate di progettare una classe che rappresenti un rettangolo: quali attributi dovete usare per specificarne le dimensioni e la collocazione nel piano? Per semplicità, ignorate l'inclinazione e supponete che il rettangolo sia allineato in orizzontale o verticale.

Ci sono almeno due possibili scelte:

- Definire il centro del rettangolo oppure un angolo, e le sue dimensioni (altezza e larghezza);
- Definire due angoli opposti.

È difficile stabilire quale delle due opzioni sia la migliore, ma giusto per fare un esempio implementeremo la prima.

Definiamo la nuova classe:

```
class Rettangolo(object):
    """Rappresenta un rettangolo.

    attributi: larghezza, altezza, angolo.
    """
```

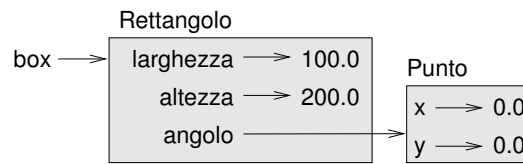


Figura 15.2: Diagramma di oggetto.

La docstring elenca gli attributi: `larghezza` e `altezza` sono numeri; `angolo` è un oggetto `Punto` che identifica l'angolo in basso a sinistra.

Per ottenere una rappresentazione di un rettangolo, dovete istanziare un oggetto `Rettangolo` e assegnare dei valori ai suoi attributi:

```

box = Rettangolo()
box.larghezza = 100.0
box.altezza = 200.0
box.angolo = Punto()
box.angolo.x = 0.0
box.angolo.y = 0.0

```

L'espressione `box.angolo.x` significa: "Vai all'oggetto a cui `box` fa riferimento e seleziona l'attributo chiamato `angolo`; poi vai a quell'oggetto e seleziona l'attributo chiamato `x`."

La Figura 15.2 mostra lo stato di questo oggetto. Un oggetto che è un attributo di un altro oggetto è detto **oggetto contenuto** (embedded).

15.4 Istanze come valori di ritorno

Le funzioni possono restituire istanze. Per esempio, `trova_centro` prende un oggetto `Rettangolo` come argomento e restituisce un oggetto `Punto` che contiene le coordinate del centro di `Rettangolo`:

```

def trova_centro(rett):
    p = Punto()
    p.x = rett.angolo.x + rett.larghezza/2.0
    p.y = rett.angolo.y + rett.altezza/2.0
    return p

```

Ecco un esempio che passa `box` come argomento e assegna il `Punto` risultante a `centro`:

```

>>> centro = trova_centro(box)
>>> stampa_punto(centro)
(50.0, 100.0)

```

15.5 Gli oggetti sono mutabili

Potete cambiare lo stato di un oggetto con un'assegnazione ad uno dei suoi attributi. Per esempio, per cambiare le dimensioni di un rettangolo senza cambiarne la posizione, potete modificare i valori di `larghezza` e `altezza`:

```
box.larghezza = box.larghezza + 50
box.altezza = box.altezza + 100
```

Potete anche scrivere delle funzioni che modificano oggetti. Per esempio, `accresci Rettangolo` prende un oggetto `Rettangolo` e due numeri, `dlargh` e `dalt`, e li aggiunge alla larghezza e all'altezza del rettangolo:

```
def accresci Rettangolo(rett, dlargh, dalt):
    rett.larghezza += dlargh
    rett.altezza += dalt
```

Ecco un esempio dell'effetto della funzione:

```
>>> print box.larghezza
100.0
>>> print box.altezza
200.0
>>> accresci Rettangolo(box, 50, 100)
>>> print box.larghezza
150.0
>>> print box.altezza
300.0
```

Dentro la funzione, `rett` è un alias di `box`, pertanto se la funzione modifica `rett`, anche `box` cambia.

Esercizio 15.2. *Scrivete una funzione di nome `sposta Rettangolo` che prenda come parametri un `Rettangolo` e due valori `dx` e `dy`. La funzione deve spostare il rettangolo nel piano, aggiungendo `dx` alla coordinata `x` di angolo, e aggiungendo `dy` alla coordinata `y` di angolo.*

15.6 Copia

Abbiamo già visto che gli alias possono rendere il programma difficile da leggere, perché una modifica in un punto del programma può dare degli effetti inattesi in un altro punto. Non è semplice tenere traccia di tutte le variabili che potrebbero fare riferimento ad un dato oggetto.

La copia di un oggetto è spesso una comoda alternativa all'alias. Il modulo `copy` contiene una funzione, anch'essa di nome `copy`, che permette di duplicare qualsiasi oggetto:

```
>>> p1 = Punto()
>>> p1.x = 3.0
>>> p1.y = 4.0
```

```
>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` e `p2` contengono gli stessi dati, ma non sono lo stesso `Punto`.

```
>>> stampa_punto(p1)
(3.0, 4.0)
>>> stampa_punto(p2)
(3.0, 4.0)
>>> p1 is p2
False
```

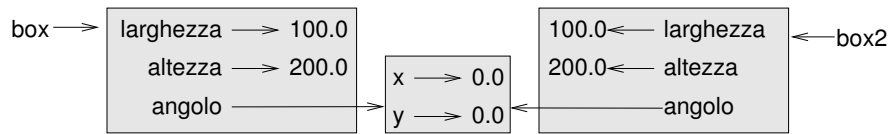


Figura 15.3: Diagramma di oggetto.

```
>>> p1 == p2
False
```

L'operatore `is` indica che `p1` e `p2` non sono lo stesso oggetto, come volevasi dimostrare. Ma forse avevate previsto che l'operatore `==` desse `True`, perché i due punti contengono gli stessi dati. In tal caso, non vi farà piacere di apprendere che nel caso di istanze, il comportamento predefinito dell'operatore `==` è lo stesso dell'operatore `is`: controlla l'identità dell'oggetto e non l'equivalenza. Questo comportamento però può essere cambiato—più avanti vedremo come.

Nell'usare `copy.copy` per duplicare un Rettangolo, noterete che copia l'oggetto Rettangolo ma non l'oggetto Punto contenuto.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.angolo is box.angolo
True
```

La Figura 15.3 mostra la situazione del diagramma di oggetto.

Questa operazione è chiamata **copia shallow** (o copia superficiale) perché copia l'oggetto ed ogni riferimento che contiene, ma non gli oggetti contenuti.

Nella maggior parte dei casi, questo non è desiderabile. Nel nostro esempio, invocare `accresci Rettangolo` su uno dei Rettangoli non influenzerebbe l'altro, ma invocare `sposta Rettangolo` su uno dei due, influenzerebbe entrambi! Questo comportamento genera confusione ed è foriero di errori.

Fortunatamente, il modulo `copy` contiene anche un altro metodo chiamato `deepcopy` che non solo copia l'oggetto, ma anche gli oggetti a cui si riferisce, e gli oggetti a cui questi ultimi a loro volta si riferiscono, e così via. Non vi sorprenderà che questa si chiami **copia profonda**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.angolo is box.angolo
False
```

`box3` e `box` sono oggetti completamente diversi.

Esercizio 15.3. Scrivete una versione di `sposta Rettangolo` che crei e restituisca un nuovo Rettangolo anziché modificare quello di origine.

15.7 Debug

Iniziando a lavorare con gli oggetti, è facile imbattersi in alcuni nuovi tipi di eccezioni. Se cercate di accedere ad un attributo che non esiste, si verifica un `AttributeError`:

```
>>> p = Punto()
>>> print p.z
AttributeError: Punto instance has no attribute 'z'
```

Se non siete sicuri di che tipo sia un oggetto, potete chiederlo:

```
>>> type(p)
<type '__main__.Punto'>
```

Se volete sapere se un oggetto ha un certo attributo, usate la funzione predefinita `hasattr`:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

Il primo argomento può essere un qualunque oggetto, il secondo è una *stringa* che contiene il nome dell'attributo.

15.8 Glossario

classe: Tipo di dato definito dall'utente. Una definizione di classe crea un nuovo oggetto classe.

oggetto classe: Oggetto che contiene le informazioni su un tipo definito dall'utente e che può essere usato per creare istanze del tipo.

istanza: Oggetto che appartiene ad una classe.

attributo: Uno dei valori associati ad un oggetto, dotato di un nome.

contenuto (oggetto): Oggetto che è contenuto come attributo di un altro oggetto (detto contenitore).

copia shallow: copia "superficiale" dei contenuti di un oggetto, senza includere alcun riferimento ad eventuali oggetti contenuti; è implementata grazie alla funzione `copy` del modulo `copy`.

copia profonda: Copia del contenuto di un oggetto e anche degli eventuali oggetti interni e degli oggetti a loro volta contenuti in essi; è implementata grazie alla funzione `deepcopy` del modulo `copy`.

diagramma di oggetto: Diagramma che mostra gli oggetti, i loro attributi e i valori di questi ultimi.

15.9 Esercizi

Esercizio 15.4. *Swampy (vedere Capitolo 4) contiene un modulo di nome `World`, che definisce un nuovo tipo chiamato anch'esso `World`. Lo potete importare in questo modo:*

```
from swampy.World import World
```

Oppure così, a seconda di come avete installato Swampy:

```
from World import World
```

Il codice seguente crea un oggetto `World` e chiama il metodo `mainloop` che resta in attesa dell'azione dell'utente.

```
world = World()
world.mainloop()
```

Dovrebbe apparire una finestra con una barra del titolo e un riquadro vuoto. Useremo questa finestra per disegnare Punti, Rettangoli e altre figure. Aggiungete le righe seguenti prima della chiamata a `mainloop` ed eseguite di nuovo il programma.

```
canvas = world.ca(width=500, height=500, background='white')
bbox = [[-150,-100], [150, 100]]
canvas.rectangle(bbox, outline='black', width=2, fill='green4')
```

Ora dovrete vedere un rettangolo verde con contorno nero. La prima riga crea un'Area di disegno (Canvas), che appare nella finestra come un riquadro bianco. L'oggetto Canvas contiene dei metodi come `rectangle` che disegnano varie forme.

`bbox` è una lista di liste che rappresenta il "contenitore" del rettangolo. La prima coppia di coordinate è l'angolo in basso a sinistra del rettangolo, la seconda è quello in alto a destra.

Potete disegnare un cerchio in questo modo:

```
canvas.circle([-25,0], 70, outline=None, fill='red')
```

Il primo parametro sono le coordinate del centro del cerchio, il secondo è il raggio.

Se aggiungete queste righe, il risultato somiglierà alla bandiera dello stato del Bangladesh. (vedi http://en.wikipedia.org/wiki/Gallery_of_sovereign-state_flags).

1. *Scrivete una funzione di nome `disegna_rettangolo` che richieda un'Area di disegno (Canvas) e un Rettangolo come argomenti, e disegni una rappresentazione del rettangolo nel Canvas.*
2. *Aggiungete un attributo di nome `color` all'oggetto Rettangolo e modificate `disegna_rettangolo` in modo che usi il nuovo attributo come colore di riempimento.*
3. *Scrivete una funzione di nome `disegna_punto` che richieda un'Area di disegno e un Punto come argomenti e disegni una rappresentazione del punto nel Canvas.*
4. *Definite una nuova classe di nome `Cerchio` con degli appropriati attributi e istanziate alcuni oggetti `Cerchio`. Scrivete una funzione `disegna_cerchio` che disegni cerchi nel Canvas.*
5. *Scrivete un programma che disegni la bandiera della Repubblica Ceca. Suggerimento: potete disegnare un poligono in questo modo:*

```
points = [[-150,-100], [150, 100], [150, -100]]
canvas.polygon(points, fill='blue')
```

Ho scritto un programmino che elenca i colori disponibili, scaricabile da http://thinkpython.com/code/color_list.py.

Capitolo 16

Classi e funzioni

Il codice degli esempi di questo capitolo è scaricabile dal sito <http://thinkpython.com/code/Time1.py>.

16.1 Tempo

Facciamo un altro esempio di tipo definito dall'utente, e definiamo una classe chiamata `Tempo` che permette di rappresentare un'ora del giorno:

```
class Tempo(object):
    """Rappresenta un'ora del giorno.

    attributi: ora, minuto, secondo
    """
```

Possiamo creare un nuovo oggetto `Tempo`, assegnandogli tre attributi per le ore, i minuti e i secondi:

```
tempo = Tempo()
tempo.ora = 11
tempo.minuto = 59
tempo.secondo = 30
```

Il diagramma di stato dell'oggetto `Tempo` è riportato in Figura 16.1.

Esercizio 16.1. *Scrivete una funzione di nome `stampa_tempo` che accetti un oggetto `Tempo` come argomento e ne stampi il risultato nel formato `ore:minuti:secondi`. Suggerimento: la sequenza di formato `'%.2d'` stampa un intero usando almeno due cifre, compreso uno zero iniziale dove necessario.*

Esercizio 16.2. *Scrivete una funzione booleana `viene_dopo` che riceva come argomenti due oggetti `Tempo`, `t1` e `t2`, e restituisca `True` se `t1` è temporalmente successivo a `t2` e `False` in caso contrario. Sfida: non usate un'istruzione `if`.*

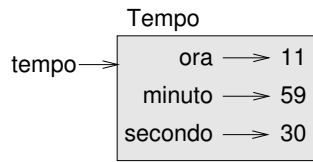


Figura 16.1: Diagramma di oggetto.

16.2 Funzioni pure

Nei prossimi paragrafi scriveremo due funzioni che aggiungono dei valori, espressi in termini temporali. Illusteremo così due tipi di funzioni: le funzioni pure e i modificatori. Dimostreremo anche una tecnica di sviluppo che chiameremo **prototipo ed evoluzioni**, che è un modo di affrontare un problema complesso partendo da un prototipo semplice e trattando poi in maniera incrementale gli aspetti di maggior complessità.

Ecco un semplice prototipo della funzione `somma_tempo`:

```
def somma_tempo(t1, t2):
    somma = Tempo()
    somma.ora = t1.ora + t2.ora
    somma.minuto = t1.minuto + t2.minuto
    somma.secondo = t1.secondo + t2.secondo
    return somma
```

La funzione crea un nuovo oggetto `Tempo`, ne inizializza gli attributi, e restituisce un riferimento al nuovo oggetto. Questa è detta **funzione pura**, perché non modifica alcuno degli oggetti che le vengono passati come argomento e, oltre a restituire un valore, non ha effetti visibili come visualizzare valori o chiedere input all'utente.

Per provare questa funzione, creiamo due oggetti `Tempo`: `inizio` che contiene l'ora di inizio di un film, come *I Monty Python e il Sacro Graal*, e `durata` che contiene la durata del film, che è un'ora e 35 minuti.

`somma_tempo` ci dirà a che ora finisce il film.

```
>>> inizio = Tempo()
>>> inizio.ora = 9
>>> inizio.minuto = 45
>>> inizio.secondo = 0

>>> durata = Tempo()
>>> durata.ora = 1
>>> durata.minuto = 35
>>> durata.secondo = 0

>>> fine = somma_tempo(inizio, durata)
>>> stampa_tempo(fine)
10:80:00
```

Il risultato, 10:80:00 non è soddisfacente. Il problema è che questa funzione non gestisce correttamente i casi in cui la somma dei minuti e dei secondi equivale o supera sessanta. Quando questo accade, dobbiamo "riportare" i 60 secondi come minuto ulteriore, o i 60 minuti come ora ulteriore.

Ecco allora una versione migliorata della funzione:

```
def somma_tempo(t1, t2):
    somma = Tempo()
    somma.ora = t1.ora + t2.ora
    somma.minuto = t1.minuto + t2.minuto
    somma.secondo = t1.secondo + t2.secondo

    if somma.secondo >= 60:
        somma.secondo -= 60
        somma.minuto += 1

    if somma.minuto >= 60:
        somma.minuto -= 60
        somma.ora += 1

    return somma
```

Sebbene questa funzione sia corretta, comincia ad essere lunga. Tra poco vedremo un'alternativa più concisa.

16.3 Modificatori

Ci sono casi in cui è utile che una funzione possa modificare gli oggetti che assume come parametri. I cambiamenti risulteranno visibili anche al chiamante. Funzioni che si comportano in questo modo sono dette **modificatori**.

`incremento`, che aggiunge un dato numero di secondi ad un oggetto `Tempo`, può essere scritta intuitivamente come modificatore. Ecco un primo abbozzo della funzione:

```
def incremento(tempo, secondi):
    tempo.secondo += secondi

    if tempo.secondo >= 60:
        tempo.secondo -= 60
        tempo.minuto += 1

    if tempo.minuto >= 60:
        tempo.minuto -= 60
        tempo.ora += 1
```

La prima riga esegue l'operazione di addizione fondamentale, mentre le successive controllano i casi particolari che abbiamo già visto prima.

Questa funzione è corretta? Cosa succede se il parametro `secondi` è molto più grande di 60?

In questo caso non è più sufficiente un unico riporto tra secondi e minuti: dobbiamo fare in modo di ripetere il controllo più volte, finché `tempo.secondo` diventa minore di 60. Allora, una possibile soluzione è quella di sostituire le istruzioni `if` con delle istruzioni `while`. Questo renderebbe la funzione corretta, ma non molto efficiente.

Esercizio 16.3. *Scrivete una versione corretta di incremento che non contenga alcun ciclo.*

Tutto quello che può essere fatto con i modificatori può anche essere fatto con le funzioni pure. Tanto è vero che alcuni linguaggi di programmazione prevedono unicamente l'uso di funzioni pure. Si può affermare che i programmi che utilizzano funzioni pure sono più veloci da sviluppare e meno soggetti ad errori rispetto a quelli che fanno uso dei modificatori. Ma in qualche caso i modificatori convengono, perché i programmi funzionali risultano meno efficienti.

In linea generale, raccomando di usare funzioni pure quando possibile e usare i modificatori solo se c'è un evidente vantaggio nel farlo. Questo tipo di approccio può essere definito **stile di programmazione funzionale**.

Esercizio 16.4. *Scrivete una versione “pura” di incremento che crei e restituisca un nuovo oggetto Tempo anziché modificare il parametro.*

16.4 Sviluppo prototipale e Sviluppo pianificato

La tecnica di sviluppo del programma che sto illustrando in questo Capitolo è detta “prototipo ed evoluzioni”: per ogni funzione, si inizia scrivendo una versione grezza (prototipo) che effettui solo i calcoli fondamentali, provandola e via via migliorandola e correggendo gli errori.

Sebbene questo approccio possa essere abbastanza efficace, specie se non avete una adeguata conoscenza del problema, può condurre a scrivere del codice inutilmente complesso (perché deve affrontare molti casi particolari) e poco affidabile (dato che è difficile essere certi che tutti gli errori siano stati rimossi).

Un'alternativa è lo **sviluppo pianificato**, nel quale una conoscenza approfondita degli aspetti del problema da affrontare rende la programmazione molto più semplice. Nel nostro caso, questa conoscenza sta nel fatto che l'oggetto Tempo è rappresentabile da un numero a tre cifre in base numerica 60! (vedere http://it.wikipedia.org/wiki/Sistema_sessagesimale.) L'attributo secondo è la “colonna delle unità”, l'attributo minuto è la “colonna delle sessantine”, e l'attributo ora quella della “trecentosessantine”.

Quando abbiamo scritto `somma_tempo` e `incremento`, stavamo a tutti gli effetti calcolando una addizione in base 60, e questo è il motivo per cui dovevamo gestire i riporti tra secondi e minuti e tra minuti e ore.

Questa osservazione ci suggerisce un altro tipo di approccio al problema: possiamo convertire l'oggetto Tempo in un numero intero e approfittare della capacità del computer di effettuare operazioni sui numeri interi.

Questa funzione converte Tempo in interi:

```
def tempo_in_int(tempo):  
    minuti = tempo.ora * 60 + tempo.minuto  
    secondi = minuti * 60 + tempo.secondo  
    return secondi
```

E questa è la funzione inversa, che converte gli interi in Tempi (ricordate che `divmod` divide il primo argomento per il secondo e restituisce una tupla che contiene il quoziente e il resto).

```
def int_in_tempo(secondi):
    tempo = Tempo()
    minuti, tempo.secondo = divmod(secondi, 60)
    tempo.ora, tempo.minuto = divmod(minuti, 60)
    return tempo
```

Per convincervi della esattezza di queste funzioni, pensateci un po' su e fate qualche prova. Una maniera di collaudarle è controllare che `tempo_in_int(int_in_tempo(x)) == x` per vari valori di `x`. Questo è un esempio di controllo di coerenza.

Quando vi siete convinti, potete usarle per riscrivere `somma_tempo`:

```
def somma_tempo(t1, t2):
    secondi = tempo_in_int(t1) + tempo_in_int(t2)
    return int_in_tempo(secondi)
```

Questa versione è più concisa dell'originale e più facile da verificare.

Esercizio 16.5. *Riscrivete* `incremento` usando `tempo_in_int` e `int_in_tempo`.

Sicuramente, la conversione numerica da base 60 a base 10 e viceversa è più astratta e meno immediata rispetto al lavoro diretto con i tempi, che è istintivamente migliore.

Ma avendo l'intuizione di trattare i tempi come numeri in base 60, e investendo il tempo necessario per scrivere le funzioni di conversione (`tempo_in_int` e `int_in_tempo`), abbiamo ottenuto un programma molto più corto, facile da leggere e correggere, e più affidabile.

Risulta anche più semplice aggiungere nuove caratteristiche, in un secondo tempo. Ad esempio, immaginate di dover sottrarre due Tempi per determinare l'intervallo trascorso. L'approccio iniziale avrebbe reso necessaria l'implementazione di una sottrazione con il prestito. Invece, con le funzioni di conversione, è molto più facile e rapido avere un programma corretto.

Paradossalmente, qualche volta rendere un problema più difficile (o più generale) lo rende più semplice, perché ci sono meno casi particolari da gestire e minori possibilità di errore.

16.5 Debug

Un oggetto `Tempo` è ben impostato se i valori di `minuto` e `secondo` sono compresi tra 0 e 60 (zero incluso ma 60 escluso) e se `ora` è positiva. `ora` e `minuto` devono essere interi, ma potremmo anche permettere a `secondo` di avere una parte decimale.

Requisiti come questi sono detti **invarianti** perché devono essere sempre soddisfatti. In altre parole, se non sono soddisfatti significa che qualcosa non è andato per il verso giusto.

Scrivere del codice per controllare le invarianti può servire a trovare errori e a identificarne le cause. Per esempio, potete scrivere una funzione `tempo_valido` che prende un oggetto `Tempo` e restituisce `False` se viola un'invariante:

```
def tempo_valido(tempo):
    if tempo.ora < 0 or tempo.minuto < 0 or tempo.secondo < 0:
        return False
    if tempo.minuto >= 60 or tempo.secondo >= 60:
        return False
    return True
```

Poi, all'inizio di ogni funzione, potete controllare l'argomento per assicurarvi della sua validità:

```
def somma_tempo(t1, t2):
    if not tempo_valido(t1) or not tempo_valido(t2):
        raise ValueError, 'oggetto Tempo non valido in somma_tempo'
    secondi = tempo_in_int(t1) + tempo_in_int(t2)
    return int_in_tempo(secondi)
```

Oppure potete usare un'istruzione `assert`, che controlla una data invariante e solleva un'eccezione in caso di difetti:

```
def somma_tempo(t1, t2):
    assert tempo_valido(t1) and tempo_valido(t2)
    secondi = tempo_in_int(t1) + tempo_in_int(t2)
    return int_in_tempo(secondi)
```

Le istruzioni `assert` sono utili perché permettono di distinguere il codice che tratta le condizioni normali da quello che controlla gli errori.

16.6 Glossario

prototipo ed evoluzioni: Tecnica di sviluppo del programma a partire da un prototipo che viene gradualmente provato, esteso e migliorato.

sviluppo pianificato: Tecnica di sviluppo che coinvolge profonde conoscenze del problema e maggiore pianificazione rispetto allo sviluppo incrementale o per prototipo

funzione pura: Funzione che non modifica gli oggetti ricevuti come argomenti. La maggior parte delle funzioni pure sono produttive.

modificatore: Funzione che cambia uno o più oggetti ricevuti come argomenti. La maggior parte dei modificatori non restituisce valori.

stile di programmazione funzionale: Stile di programmazione in cui la maggior parte delle funzioni è pura.

invariante: Condizione che deve sempre essere vera durante l'esecuzione del programma.

16.7 Esercizi

Il codice degli esempi di questo capitolo è scaricabile dal sito <http://thinkpython.com/code/Time1.py>; le soluzioni di questi esercizi si trovano in http://thinkpython.com/code/Time1_soln.py.

Esercizio 16.6. *Scrivete una funzione di nome `moltiplica_tempo` che accetti un oggetto `Tempo` e un numero, e restituisca un nuovo oggetto `Tempo` che contiene il prodotto del `Tempo` iniziale per il numero.*

Usate poi `moltiplica_tempo` per scrivere una funzione che prenda un oggetto `Tempo` che rappresenta il tempo finale di una gara, e un numero che rappresenta la distanza percorsa, e restituisca un oggetto `Tempo` che rappresenta la media di gara (tempo al chilometro).

Esercizio 16.7. Il modulo `datetime` fornisce gli oggetti `date` e `time`, simili agli oggetti `Data` e `Tempo` di questo capitolo, ma che contengono un ricco insieme di metodi e operatori. Leggetene la documentazione sul sito <http://docs.python.org/2/library/datetime.html>.

1. Usate il modulo `datetime` per scrivere un programma che ricavi la data odierna e visualizzi il giorno della settimana.
2. Scrivete un programma che riceva una data di nascita come input e stampi l'età dell'utente e il numero di giorni, ore, minuti e secondi che mancano al prossimo compleanno.
3. Date due persone nate in giorni diversi, esiste un giorno in cui uno ha un'età doppia dell'altro. Questo è il loro "Giorno del Doppio". Scrivete un programma che prenda due date di nascita e calcoli quando si verifica il "Giorno del Doppio".
4. Un po' più difficile: scrivetene una versione più generale che calcoli il giorno in cui una persona ha n volte l'età di un'altra.

Capitolo 17

Classi e metodi

Il codice degli esempi di questo capitolo è scaricabile dal sito <http://thinkpython.com/code/Time2.py>.

17.1 Funzionalità orientate agli oggetti

Python è un **linguaggio di programmazione orientato agli oggetti**, in altre parole contiene delle caratteristiche a supporto della programmazione orientata agli oggetti.

Non è facile definire la programmazione orientata agli oggetti, tuttavia abbiamo già visto alcune sue particolarità:

- I programmi sono costituiti da definizioni di oggetti e definizioni di funzioni, e buona 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 sugli oggetti corrispondono al modo in cui gli oggetti interagiscono tra loro nella realtà quotidiana.

Per esempio, la classe `Tempo` definita nel Capitolo 16 corrisponde al modo in cui le persone pensano alle ore del giorno, e le funzioni che abbiamo definite corrispondono al tipo di operazioni che le persone fanno con il tempo. Allo stesso modo, le classi `Punto` e `Rettangolo` corrispondono ai rispettivi concetti matematici.

Finora, non abbiamo tratto vantaggio dalle capacità di supporto della programmazione orientata agli oggetti fornite da Python. A dire il vero, queste funzionalità non sono indispensabili; piuttosto, forniscono una sintassi alternativa per fare le cose che abbiamo già fatto. Ma in molti casi questa alternativa è più concisa e si adatta in modo più accurato alla struttura del programma.

Ad esempio, nel programma `Time` non c'è una chiara connessione tra la definizione della classe e le definizioni di funzione che seguono. A un esame più attento, è però evidente che tutte queste funzioni ricevono almeno un oggetto `Tempo` come argomento.

Questa osservazione giustifica l'esistenza dei **metodi**; un metodo è una funzione associata ad una particolare classe. Abbiamo già visto qualche metodo per le stringhe, le liste, i dizionari e le tuple. In questo capitolo, definiremo metodi per i tipi definiti dall'utente.

Da un punto di vista logico, i metodi sono la stessa cosa delle funzioni, ma con due differenze sintattiche:

- I metodi sono definiti all'interno di una definizione di classe, per rendere esplicita la relazione tra la classe stessa ed il metodo.
- La sintassi per invocare un metodo è diversa da quella usata per chiamare una funzione.

Nei prossimi paragrafi prenderemo le funzioni scritte nei due capitoli precedenti e le trasformeremo in metodi. Questa trasformazione è puramente meccanica e si fa semplicemente seguendo una serie di passi: se siete in grado di convertire da funzione a metodo e viceversa, riuscirete anche a scegliere la forma migliore, qualsiasi cosa dobbiate fare.

17.2 Stampa di oggetti

Nel Capitolo 16, abbiamo definito una classe chiamata `Tempo`, e nell'Esercizio 16.1, avete scritto una funzione di nome `stampa_tempo`:

```
class Tempo(object):  
    """Rappresenta un'ora del giorno."""
```

```
def stampa_tempo(tempo):  
    print '%.2d:%.2d:%.2d' % (tempo.ora, tempo.minuto, tempo.secondo)
```

Per chiamare questa funzione occorre passare un oggetto `Tempo` come argomento:

```
>>> inizio = Tempo()  
>>> inizio.ora = 9  
>>> inizio.minuto = 45  
>>> inizio.secondo = 00  
>>> stampa_tempo(inizio)  
09:45:00
```

Per trasformare `stampa_tempo` in un metodo, tutto quello che dobbiamo fare è spostare la definizione della funzione all'interno della definizione della classe. Notate bene la modifica nell'indentazione.

```
class Tempo(object):  
    def stampa_tempo(tempo):  
        print '%.2d:%.2d:%.2d' % (tempo.ora, tempo.minuto, tempo.secondo)
```

Ora ci sono due modi di chiamare `stampa_tempo`. Il primo (e meno usato) è utilizzare la sintassi delle funzioni:

```
>>> Tempo.stampa_tempo(inizio)  
09:45:00
```

In questo uso della notazione a punto, `Tempo` è il nome della classe e `stampa_tempo` è il nome del metodo. `inizio` è passato come parametro.

Il secondo modo, più conciso, è usare la sintassi dei metodi:

```
>>> inizio.stampa_tempo()  
09:45:00
```

Sempre usando la *dot notation*, `stampa_tempo` è ancora il nome del metodo, mentre `inizio` è l'oggetto sul quale il metodo è invocato, che è chiamato il **soggetto**. Come il soggetto di una frase è ciò a cui si riferisce la frase, il soggetto del metodo è ciò a cui si applica l'invocazione del metodo.

All'interno del metodo, il soggetto viene assegnato al primo dei parametri: in questo caso, `inizio` viene assegnato a `tempo`.

Per convenzione, il primo parametro di un metodo viene chiamato `self`, di conseguenza è bene riscrivere `stampa_tempo` così:

```
class Tempo(object):
    def stampa_tempo(self):
        print '%.2d:%.2d:%.2d' % (self.ora, self.minuto, self.secondo)
```

La ragione di questa convenzione è una metafora implicita:

- La sintassi di una chiamata di funzione, `stampa_tempo(inizio)`, suggerisce che la funzione è la parte attiva, che dice qualcosa del tipo: "Ehi, `stampa_tempo`! Ti passo un oggetto da stampare!"
- Nella programmazione orientata agli oggetti, la parte attiva sono gli oggetti. L'invocazione di un metodo come `inizio.stampa_tempo()` dice: "Ehi, `inizio`! Stampa te stesso!"

Questo cambio di prospettiva sarà anche più elegante, ma cogliere la sua utilità non è immediato. Nei semplici esempi che abbiamo visto finora, può non esserlo. Ma in altri casi, spostare la responsabilità dalle funzioni agli oggetti rende possibile scrivere funzioni più versatili e rende più facile mantenere e riusare il codice.

Esercizio 17.1. *Riscrivete `tempo_in_int` (vedere Paragrafo 16.4) come metodo. Probabilmente è improprio riscrivere `int_in_tempo` come metodo: su quale oggetto lo invochereste?*

17.3 Un altro esempio

Ecco una versione di incremento (vedere Paragrafo 16.3), riscritto come metodo:

```
# all'interno della classe Tempo:
```

```
def incremento(self, secondi):
    secondi += self.tempo_in_int()
    return int_in_tempo(secondi)
```

Questa versione presuppone che pure `tempo_in_int` sia stato scritto come metodo, come nell'Esercizio 17.1. Notate anche che si tratta di una funzione pura e non un modificatore.

Ecco come invocare incremento:

```
>>> inizio.stampa_tempo()
09:45:00
>>> fine = inizio.incremento(1337)
>>> fine.stampa_tempo()
10:07:17
```

Il soggetto, `inizio`, viene assegnato quale primo parametro, a `self`. L'argomento, `1337`, viene assegnato quale secondo parametro, a `secondi`.

Questo meccanismo può confondere le idee, specie se commettete qualche errore. Per esempio, se invocate `incremento` con due argomenti ottenete:

```
>>> fine = inizio.incremento(1337, 460)
TypeError: incremento() takes exactly 2 arguments (3 given)
```

Il messaggio di errore a prima vista non è chiaro, perché ci sono solo due argomenti tra parentesi. Ma bisogna tener conto che anche il soggetto è considerato un argomento, ecco perché in totale fanno tre.

17.4 Un esempio più complesso

`viene_dopo` (vedere Esercizio 16.2) è leggermente più complesso da scrivere come metodo, perché richiede come parametri due oggetti `Tempo`. In questo caso, la convenzione prevede di denominare il primo parametro `self` e il secondo `other`:

```
# all'interno della classe Tempo:
```

```
def viene_dopo(self, other):
    return self.tempo_in_int() > other.tempo_in_int()
```

Per usare questo metodo, lo dovete invocare su un oggetto e passare l'altro come argomento:

```
>>> fine.viene_dopo(inizio)
True
```

Una particolarità di questa sintassi è che si legge quasi come in italiano: “fine viene dopo inizio?”

17.5 Il metodo speciale `init`

Il metodo `init` (abbreviazione di *initialization*, ovvero inizializzazione) è un metodo speciale che viene invocato quando un oggetto viene istanziato. Il suo nome completo è `__init__` (due caratteri underscore, seguiti da `init`, e da altri due underscore). Un metodo `init` per la classe `Tempo` può essere il seguente:

```
# all'interno della classe Tempo:
```

```
def __init__(self, ora=0, minuto=0, secondo=0):
    self.ora = ora
    self.minuto = minuto
    self.secondo = secondo
```

È prassi che i parametri di `__init__` abbiano gli stessi nomi degli attributi. L'istruzione

```
self.ora = ora
```

memorizza il valore del parametro `ora` come attributo di `self`.

I parametri sono opzionali, quindi se chiamate `Tempo` senza argomenti, ottenete i valori di default.

```
>>> tempo = Tempo()
>>> tempo.stampa_tempo()
00:00:00
```

Se fornite un argomento, esso va a sovrascrivere ora:

```
>>> tempo = Tempo (9)
>>> tempo.stampa_tempo()
09:00:00
```

Se ne fornite due, sovrascrivono ora e minuto.

```
>>> tempo = Tempo(9, 45)
>>> tempo.stampa_tempo()
09:45:00
```

E se ne fornite tre, sovrascrivono tutti e tre i valori di default.

Esercizio 17.2. *Scrivete un metodo `init` per la classe `Punto` che prenda `x` e `y` come parametri opzionali e li assegni agli attributi corrispondenti.*

17.6 Il metodo speciale `__str__`

`__str__` è un altro metodo speciale, come `__init__`, che ha lo scopo di restituire una rappresentazione di un oggetto in forma di stringa.

Ecco ad esempio un metodo `str` per un oggetto `Tempo`:

```
# all'interno della classe Tempo:
```

```
def __str__(self):
    return '%.2d:%.2d:%.2d' % (self.ora, self.minuto, self.secondo)
```

Quando stampate un oggetto con l'istruzione `print`, Python invoca il metodo `str`:

```
>>> tempo = Tempo(9, 45)
>>> print tempo
09:45:00
```

Personalmente, quando scrivo una nuova classe, quasi sempre inizio con lo scrivere `__init__`, che rende più facile istanziare un oggetto, e `__str__`, che è utile per il debugging.

Esercizio 17.3. *Scrivete un metodo `str` per la classe `Punto`. Create un oggetto `Punto` e stampatelo.*

17.7 Operator overloading

Nelle classi definite dall'utente, avete la possibilità di adattare il comportamento degli operatori attraverso la definizione di altri appositi metodi speciali. Per esempio se definite il metodo speciale di nome `__add__` per la classe `Tempo`, potete poi usare l'operatore `+` sugli oggetti `Tempo`.

Ecco come potrebbe essere scritta la definizione:

```
# all'interno della classe Tempo:

def __add__(self, other):
    secondi = self.tempo_in_int() + other.tempo_in_int()
    return int_in_tempo(secondi)
```

Ed ecco come può essere usata:

```
>>> inizio = Tempo(9, 45)
>>> durata = Tempo(1, 35)
>>> print inizio + durata
11:20:00
```

Quando applicate l'operatore + agli oggetti Tempo, Python invoca `__add__`. Quando stampa il risultato, Python invoca `__str__`. Accadono parecchie cose, dietro le quinte!

Cambiare il comportamento degli operatori in modo che funzionino con i tipi definiti dall'utente è chiamato **operator overloading** (letteralmente, sovraccarico degli operatori). In Python, per ogni operatore esiste un corrispondente metodo speciale, come `__add__`. Per ulteriori dettagli consultate <http://docs.python.org/2/reference/datamodel.html#specialnames>.

Esercizio 17.4. *Scrivete un metodo add per la classe Punto.*

17.8 Smistamento in base al tipo

Nel Paragrafo precedente abbiamo sommato due oggetti Tempo, ma potrebbe anche capitare di voler aggiungere un numero intero a un oggetto Tempo. Quella che segue è una versione di `__add__` che controlla il tipo di `other` e, a seconda dei casi, invoca o `somma_tempo` o `incremento`:

```
# all'interno della classe Tempo:

def __add__(self, other):
    if isinstance(other, Tempo):
        return self.somma_tempo(other)
    else:
        return self.incremento(other)

def somma_tempo(self, other):
    secondi = self.tempo_in_int() + other.tempo_in_int()
    return int_in_tempo(secondi)

def incremento(self, secondi):
    secondi += self.tempo_in_int()
    return int_in_tempo(secondi)
```

La funzione predefinita `isinstance` prende un valore e un oggetto classe, e restituisce `True` se il valore è un'istanza della classe.

Quindi, se `other` è un oggetto Tempo, `__add__` invoca `somma_tempo`. Altrimenti, considera che il parametro sia un numero, e invoca `incremento`. Questa operazione è detta

smistamento in base al tipo, perché invia il calcolo a metodi diversi a seconda del tipo di argomento.

Ecco degli esempi che usano l'operatore + con tipi diversi:

```
>>> inizio = Tempo(9, 45)
>>> durata = Tempo(1, 35)
>>> print inizio + durata
11:20:00
>>> print inizio + 1337
10:07:17
```

Sfortunatamente, questa implementazione di addizione non è commutativa. Se l'intero è il primo operando vi risulterà infatti:

```
>>> print 1337 + inizio
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

Il problema è che, invece di chiedere all'oggetto Tempo di aggiungere un intero, Python chiede all'intero di aggiungere un oggetto Tempo, ma l'intero non ha la minima idea di come farlo. Ma a questo c'è una soluzione intelligente: il metodo speciale `__radd__`, che sta per *right-side add* ("addizione lato destro"). Questo metodo viene invocato quando un oggetto Tempo compare sul lato destro dell'operatore +. Eccone la definizione:

```
# all'interno della classe Tempo:

    def __radd__(self, other):
        return self.__add__(other)
```

Ed eccolo in azione:

```
>>> print 1337 + inizio
10:07:17
```

Esercizio 17.5. *Scrivete un metodo add per i Punti che possa funzionare sia con un oggetto Punto che con una tupla:*

- Se il secondo operando è un Punto, il metodo deve restituire un nuovo Punto la cui coordinata x sia la somma delle coordinate x cdegli operandi, e lo stesso per le coordinate y.
- Se il secondo operando è una tupla, il metodo deve aggiungere il primo elemento della tupla alla coordinata x e il secondo elemento alla coordinata y, e restituire un nuovo Punto con le coordinate risultanti.

17.9 Polimorfismo

Lo smistamento in base al tipo è utile all'occorrenza, ma (fortunatamente) non è sempre necessario. Spesso potete evitarlo scrivendo le funzioni in modo che operino correttamente con argomenti di tipo diverso.

Molte delle funzioni che abbiamo scritto per le stringhe, in realtà funzionano con qualsiasi tipo di sequenza. Per esempio, nel Paragrafo 11.1 abbiamo usato istogramma per contare quante volte ciascuna lettera appare in una parola.

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

Questa funzione è applicabile anche a liste, tuple e perfino dizionari, a condizione che agli elementi di *s* sia applicabile un hash in modo che possano essere usati come chiavi in *d*.

```
>>> t = ['spam', 'uovo', 'spam', 'spam', 'bacon', 'spam']
>>> istogramma(t)
{'bacon': 1, 'uovo': 1, 'spam': 4}
```

Le funzioni che sono in grado di operare con tipi diversi sono dette **polimorfiche**. Il polimorfismo facilita il riuso del codice. Ad esempio, la funzione predefinita *sum*, che addiziona gli elementi di una sequenza, funziona alla sola condizione che gli elementi della sequenza siano addizionabili.

Dato che agli oggetti *Tempo* abbiamo fornito un metodo *add*, funzionano con *sum*:

```
>>> t1 = Tempo(7, 43)
>>> t2 = Tempo(7, 41)
>>> t3 = Tempo(7, 37)
>>> totale = sum([t1, t2, t3])
>>> print totale
23:01:00
```

In linea generale, se tutte le operazioni all'interno di una funzione si possono applicare ad un dato tipo, allora la funzione può operare con quel tipo.

Il miglior genere di polimorfismo è quello involontario, quando scoprite che una funzione che avete già scritto può essere applicata anche ad un tipo che non avevate previsto.

17.10 Debug

È consentito aggiungere attributi in qualsiasi momento dell'esecuzione di un programma, ma per i puristi della teoria dei tipi è una prassi discutibile avere oggetti dello stesso tipo con differenti gruppi di attributi. Di solito, inizializzare tutti gli attributi di un oggetto nel metodo *init* è una prassi migliore.

Se non siete certi che un oggetto abbia un particolare attributo, potete usare la funzione predefinita *hasattr* (vedere Paragrafo 15.7).

Un altro modo di accedere agli attributi di un oggetto è tramite l'attributo speciale *__dict__*, che è un dizionario che fa corrispondere nomi degli attributi (come stringhe) e valori:

```
>>> p = Punto(3, 4)
>>> print p.__dict__
{'y': 4, 'x': 3}
```

Per gli scopi del debug, può essere utile tenere questa funzione a portata di mano:

```
def stampa_attributi(obj):  
    for attr in obj.__dict__:
        print attr, getattr(obj, attr)
```

`stampa_attributi` attraversa gli elementi nel dizionario relativo all'oggetto e stampa ciascun nome di attributo con il suo valore.

La funzione predefinita `getattr` prende un oggetto e un nome di attributo (come stringa) e restituisce il valore dell'attributo.

17.11 Interfaccia e implementazione

Uno degli scopi della progettazione orientata agli oggetti è di rendere più agevole la manutenzione del software, che significa poter mantenere il programma funzionante quando altre parti del sistema vengono cambiate e poter modificare il programma per adeguarlo a dei nuovi requisiti.

Un principio di progettazione che aiuta a raggiungere questo obiettivo è di tenere le interfacce separate dalle implementazioni. Per gli oggetti, significa che i metodi forniti da una classe non devono dipendere da come vengono rappresentati gli attributi.

Per esempio, in questo capitolo abbiamo sviluppato una classe che rappresenta un'ora del giorno. I metodi forniti da questa classe comprendono `tempo_in_int`, `viene_dopo`, e `somma_tempo`.

Quei metodi possono essere implementati in diversi modi. I dettagli dell'implementazione dipendono da come rappresentiamo il tempo. In questo capitolo, gli attributi di un oggetto `Tempo` sono `ora`, `minuto`, e `secondo`.

Come alternativa, avremmo potuto sostituire quegli attributi con un singolo numero intero, come secondi trascorsi dalla mezzanotte. Con questa implementazione, alcuni metodi come `viene_dopo`, sarebbero diventati più facili da scrivere, ma altri più difficili.

Dopo aver sviluppato una nuova classe, potreste scoprire una implementazione migliore. Se altre parti del programma usano quella classe, cambiare l'interfaccia può essere dispendioso in termini di tempo e fonte di errori.

Ma se avete progettato l'interfaccia accuratamente, potete cambiare l'implementazione senza cambiare l'interfaccia, che significa che non occorre cambiare altre parti del programma.

Mantenere l'interfaccia separata dall'implementazione comporta il dover nascondere gli attributi. Il codice in altre parti del programma (esterne alla definizione di classe) dovrebbe usare metodi per leggere e modificare lo stato dell'oggetto e non accedere direttamente agli attributi. Questo principio è detto **information hiding** (occultamento delle informazioni); vedere http://en.wikipedia.org/wiki/Information_hiding e http://it.wikipedia.org/wiki/Incapsulamento_%28informatica%29.

Esercizio 17.6. Scaricate il codice degli esempi di questo capitolo (<http://thinkpython.com/code/Time2.py>). Cambiate gli attributi di `Tempo` con un singolo intero che rappresenta i secondi dalla mezzanotte. Quindi modificate i metodi (e la funzione `int_in_tempo`) in modo che funzionino con la nuova implementazione. Non dovete cambiare il codice di prova in `main`. Quando avete finito, l'output dovrebbe essere lo stesso di prima. Soluzione: http://thinkpython.com/code/Time2_soln.py

17.12 Glossario

linguaggio orientato agli oggetti: Linguaggio che possiede delle caratteristiche, come classi definite dall'utente e sintassi dei metodi, che facilitano la programmazione orientata agli oggetti.

programmazione orientata agli oggetti: Paradigma di programmazione in cui i dati e le operazioni sui dati vengono organizzati in classi e metodi.

metodo: Funzione definita all'interno di una definizione di classe e che viene invocata su istanze di quella classe.

soggetto: L'oggetto sul quale viene invocato un metodo.

operator overloading: Cambiare il comportamento di un operatore come + in modo che funzioni con un tipo definito dall'utente.

smistamento in base al tipo: Schema di programmazione che controlla il tipo di un operando e invoca funzioni diverse in base ai diversi tipi.

polimorfico: Di una funzione che può operare con più di un tipo di dati.

information hiding: Principio per cui l'interfaccia di un oggetto non deve dipendere dalla sua implementazione, con particolare riferimento alla rappresentazione dei suoi attributi.

17.13 Esercizi

Esercizio 17.7. *Questo esercizio è un aneddoto monitorio su uno degli errori più comuni e difficili da trovare in Python. Scrivete una definizione di una classe di nome `Canguro` con i metodi seguenti:*

1. Un metodo `__init__` che inizializza un attributo di nome `contenuto_tasca` ad una lista vuota.
2. Un metodo di nome `intasca` che prende un oggetto di qualsiasi tipo e lo inserisce in `contenuto_tasca`.
3. Un metodo `__str__` che restituisce una stringa di rappresentazione dell'oggetto `Canguro` e dei contenuti della tasca.

Provate il codice creando due oggetti `Canguro`, assegnandoli a variabili di nome `can` e `guro`, e aggiungendo poi `guro` al contenuto della tasca di `can`.

Scaricate <http://thinkpython.com/code/BadKangaroo.py>. Contiene una soluzione al problema precedente, ma con un grande e serio errore. Trovatelo e sistematelo.

Se vi bloccate, potete scaricare <http://thinkpython.com/code/GoodKangaroo.py>, che spiega il problema e illustra una soluzione.

Esercizio 17.8. *Visual è un modulo Python che offre supporto alla grafica 3-D. Dato che non è sempre incluso in un'installazione standard di Python, potrebbe essere necessario installarlo dal vostro repository software o scaricandolo da <http://vpython.org>.*

L'esempio che segue crea uno spazio 3-D ampio 256 unità, in altezza, larghezza e profondità, e imposta il "centro" nel punto (128, 128, 128). Quindi, disegna una sfera di colore blu.

```
from visual import *

scene.range = (256, 256, 256)
scene.center = (128, 128, 128)

color = (0.1, 0.1, 0.9)          # mostly blue
sphere(pos=scene.center, radius=128, color=color)

color è una tupla che esprime il colore in formato RGB; cioè gli elementi sono i livelli di Rosso-Verde-Blu, compresi tra 0.0 e 1.0 (vedere http://it.wikipedia.org/wiki/RGB).
```

Se eseguite questo codice, dovreste vedere una finestra con sfondo nero e una sfera blu. Se trascinate premendo il pulsante centrale verso l'alto e il basso, potete zoomare la scena, mentre trascinando con il pulsante destro si può ruotarla, anche se con una sola sfera blu non si colgono bene le differenze.

Il ciclo seguente crea un cubo di sfere:

```
t = range(0, 256, 51)
for x in t:
    for y in t:
        for z in t:
            pos = x, y, z
            sphere(pos=pos, radius=10, color=color)
```

1. Inserite questo codice in uno script e accertatevi che funzioni.
2. Modificate il programma in modo che ogni sfera nel cubo abbia un colore che corrisponda alla sua posizione nello spazio RGB. Notate che la coordinate spaziali vanno da 0 a 255, mentre i valori delle tuple sono compresi nell'intervallo 0.0-1.0 .
3. Scaricate http://thinkpython.com/code/color_list.py e usate la funzione `read_colors` per generare una lista dei colori disponibili nel vostro sistema, i loro nomi e valori RGB. Per ogni colore con nome, disegnate una sfera nella posizione che corrisponde ai suoi valori RGB.

Confrontate la mia soluzione sul sito http://thinkpython.com/code/color_space.py.

Capitolo 18

Ereditarietà

In questo capitolo illustrerò delle classi che rappresentano carte da gioco, mazzi di carte e mani di poker. Se non giocate a poker, potete leggere qualcosa in proposito sul sito <http://it.wikipedia.org/wiki/Poker>, ma non è un obbligo: vi spiegherò quello che serve. Gli esempi di codice di questo capitolo si trovano all'indirizzo <http://thinkpython.com/code/Card.py>.

Se non conoscete bene le carte da gioco francesi, potete consultare il sito http://it.wikipedia.org/wiki/Carte_da_gioco.

18.1 Oggetti Carta

In un mazzo ci sono 52 carte, e ciascuna appartiene a uno tra quattro semi e a uno tra tredici valori. I semi sono Picche, Cuori, Quadri e Fiori (in ordine decrescente nel gioco del bridge). I valori sono Asso, 2, 3, 4, 5, 6, 7, 8, 9, 10, Fante, Regina e Re. A seconda del gioco, l'Asso può essere superiore al Re o inferiore al 2.

Se vogliamo definire un nuovo oggetto che rappresenti una carta da gioco, è evidente quali attributi dovrebbe avere: **valore** e **seme**. È meno evidente stabilire di che tipo devono essere questi attributi. Una possibilità è usare stringhe contenenti parole come 'Picche' per i semi e 'Regina' per i valori. Ma un problema di questa implementazione è che non è facile confrontare le carte per vedere quale abbia un seme o un valore superiore.

Un'alternativa è usare degli interi per **codificare** valori e semi. In questo contesto, “codificare” significa determinare una corrispondenza tra numeri e semi o numeri e valori. Non significa che debba essere un segreto (quello è “criptare”).

Per esempio, questa tabella mostra i semi e i corrispondenti codici interi:

Picche	↦	3
Cuori	↦	2
Quadri	↦	1
Fiori	↦	0

In questo modo, diventa facile confrontare le carte: siccome ai semi più alti corrispondono numeri più alti, si possono confrontare i semi confrontando i loro codici corrispondenti.

Nel caso dei valori, la corrispondenza è abbastanza immediata: ogni valore numerico corrisponde al rispettivo intero, mentre per le figure:

```
Fante    ↦    11
Regina   ↦    12
Re       ↦    13
```

Uso il simbolo \mapsto per chiarire che queste corrispondenze non fanno parte del programma Python. Fanno parte del progetto del programma, ma non compaiono esplicitamente nel codice.

Ecco come si può presentare la definizione di classe per Carta:

```
class Carta(object):
    """Rappresenta una carta da gioco standard."""

    def __init__(self, seme=0, valore=2):
        self.seme = seme
        self.valore = valore
```

Come al solito, il metodo `init` prevede un parametro opzionale per ciascun attributo. La carta di default è il 2 di fiori.

Per creare una carta, si chiama la classe `Carta` con il seme e il valore desiderati.

```
regina_di_quadri = Carta(1, 12)
```

18.2 Attributi di classe

Per stampare gli oggetti `Carta` in un modo comprensibile agli utenti, occorre stabilire una corrispondenza dai codici interi ai relativi semi e valori. Un modo naturale per farlo è usare delle liste di stringhe, che assegneremo a degli **attributi di classe**:

all'interno della classe `Carta`:

```
nomi_semi = ['Fiori', 'Quadri', 'Cuori', 'Picche']
nomi_valori = [None, 'Asso', '2', '3', '4', '5', '6', '7',
               '8', '9', '10', 'Fante', 'Regina', 'Re']

def __str__(self):
    return '%s di %s' % (Carta.nomi_valori[self.valore],
                        Carta.nomi_semi[self.seme])
```

Variabili come `nomi_semi` e `nomi_valori`, che sono definite dentro la classe ma esternamente a ogni metodo, sono chiamate attributi di classe perché sono associati all'oggetto classe `Carta`.

Questo termine li distingue da variabili come `seme` e `valore`, che sono chiamati **attributi di istanza** perché sono associati ad una specifica istanza.

Ad entrambi i tipi si accede usando la notazione a punto. Per esempio in `__str__`, `self` è un oggetto carta e `self.valore` è il suo valore. Allo stesso modo, `Carta` è un oggetto classe, e `Carta.nomi_valori` è una lista di stringhe associata alla classe.

Ogni carta ha i suoi propri `seme` e `valore`, ma esiste una sola copia di `nomi_semi` e `nomi_valori`.

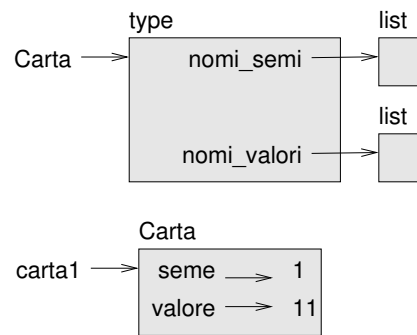


Figura 18.1: Diagramma di oggetto.

Mettendo insieme il tutto, l'espressione `Carta.nomi_valori[self.valore]` significa “usa l'attributo `valore` dell'oggetto `self` come indice nella lista `nomi_valori` dalla classe `Carta`, e seleziona la stringa corrispondente.”

Il primo elemento della lista `nomi_valori` è `None` perché non esiste una carta di valore zero. Includendo `None` come segnaposto, otteniamo una corrispondenza corretta per cui all'indice 2 corrisponde la stringa `'2'`, e così via. Per evitare questo trucco, avremmo potuto usare un dizionario al posto di una lista.

Con i metodi che abbiamo visto fin qui, possiamo creare e stampare i nomi delle carte:

```
>>> carta1 = Carta(2, 11)
>>> print carta1
Fante di Cuori
```

La Figura 18.1 è un diagramma dell'oggetto classe `Carta` e di una `Carta`, sua istanza. `Carta` è un oggetto classe, quindi è di tipo `type`. `carta1` invece è di tipo `Carta`. (Per motivi di spazio ho ommesso i contenuti di `nomi_semi` e `nomi_valori`).

18.3 Confrontare le carte

Per i tipi predefiniti, esistono gli operatori relazionali (`<`, `>`, `==`, etc.) che permettono di confrontare i valori e determinare quale è maggiore, minore o uguale a un altro. Per i tipi definiti dall'utente, possiamo sovrascrivere il comportamento degli operatori predefiniti grazie a un metodo speciale chiamato `__cmp__`.

`__cmp__` richiede due parametri, `self` e `other`, e restituisce un numero positivo se il primo oggetto è più grande del secondo, un numero negativo se il secondo oggetto è maggiore, e 0 se sono equivalenti.

L'ordinamento corretto delle carte da gioco non è immediato. Per esempio, tra il 3 di Fiori e il 2 di Quadri, quale è più grande? Una carta ha un valore maggiore, ma l'altra ha un seme superiore. Per confrontare le carte, bisogna prima stabilire se è più importante il seme oppure il valore.

La risposta dipenderà dalle regole del gioco a cui stiamo giocando, ma per semplificare supponiamo che sia più importante il seme, per cui le carte di Picche sovrastano tutte quelle di Quadri, e così via.

Deciso questo, possiamo scrivere `__cmp__`:

```
# all'interno della classe Carta:

def __cmp__(self, other):
    # controlla i semi
    if self.seme > other.seme: return 1
    if self.seme < other.seme: return -1

    # semi uguali... controlla i valori
    if self.valore > other.valore: return 1
    if self.valore < other.valore: return -1

    # valori uguali... parita'
    return 0
```

Potete scriverlo anche in modo più compatto, usando un confronto di tuple:

```
# all'interno della classe Carta:

def __cmp__(self, other):
    t1 = self.seme, self.valore
    t2 = other.seme, other.valore
    return cmp(t1, t2)
```

La funzione predefinita `cmp` ha la stessa interfaccia del metodo `__cmp__`: prende due valori e restituisce un numero positivo se il primo è maggiore, uno negativo se il secondo è maggiore e 0 se sono uguali.

In Python 3, la funzione `cmp` non esiste più e il metodo `__cmp__` non è più supportato. In sostituzione si trova `__lt__`, che restituisce `True` se `self` è minore di `other`. Potete implementare `__lt__` usando delle tuple e l'operatore `<`.

Esercizio 18.1. *Scrivete un metodo `__cmp__` per gli oggetti `Tempo`. Suggerimento: potete usare un confronto di tuple, ma anche prendere in considerazione una sottrazione intera.*

18.4 Mazzi di carte

Ora che abbiamo le carte, il prossimo passo è definire i Mazzi. Dato che un mazzo è composto di carte, è ovvio che ogni Mazzo contenga una lista di carte come attributo.

Quella che segue è una definizione di classe di Mazzo. Il metodo `init` crea l'attributo `carte` e genera l'insieme standard di 52 carte:

```
class Mazzo(object):

    def __init__(self):
        self.carte = []
        for seme in range(4):
            for valore in range(1, 14):
                carta = Carta(seme, valore)
                self.carte.append(carta)
```

Il modo più facile di popolare il mazzo è quello di usare un ciclo nidificato. Il ciclo più esterno enumera i semi da 0 a 3; quello interno enumera i valori da 1 a 13. Ogni iterazione crea una nuova carta del seme e valore correnti e la accoda nella lista `self.carte`.

18.5 Stampare il mazzo

Ecco un metodo `__str__` per `Mazzo`:

#all'interno della classe `Mazzo`:

```
def __str__(self):
    res = []
    for carta in self.carte:
        res.append(str(carta))
    return '\n'.join(res)
```

Questo metodo illustra un modo efficiente di accumulare una stringa lunga: costruire una lista di stringhe e poi usare `join`. La funzione predefinita `str` invoca il metodo `__str__` su ciascuna carta e restituisce la rappresentazione della stringa.

Dato che invochiamo `join` su un carattere di ritorno a capo, le carte sono stampate su righe separate. Ed ecco quello che risulta:

```
>>> mazzo = Mazzo()
>>> print mazzo
Asso di Fiori
2 di Fiori
3 di Fiori
...
10 di Picche
Fante di Picche
Regina di Picche
Re di Picche
```

Anche se il risultato viene visualizzato su 52 righe, si tratta di un'unica lunga stringa che contiene caratteri di ritorno a capo.

18.6 Aggiungere, togliere, mescolare e ordinare

Per distribuire le carte, ci serve un metodo che tolga una carta dal mazzo e la restituisca. Il metodo delle liste `pop` è adatto allo scopo:

#all'interno della classe `Mazzo`:

```
def toglì_carta(self):
    return self.carte.pop()
```

Siccome `pop` rimuove l'ultima carta della lista, è come se distribuissimo le carte dal fondo del mazzo. Nella realtà, questa sarebbe una cosa orribile, ma in questo contesto non ha importanza.

Per aggiungere una carta, usiamo il metodo delle liste `append`:

#all'interno della classe `Mazzo`:

```
def aggiungi_carta(self, carta):
    self.carte.append(carta)
```

Un metodo come questo, che usa in realtà un'altra funzione senza fare molto di più, da alcuni viene chiamato **impiallacciatura**. Questa metafora deriva dall'industria del legno: l'impiallacciatura consiste nell'incollare un sottile strato di legno di buona qualità sulla superficie di un pannello economico.

In questo caso abbiamo definito un metodo “sottile” che esprime un'operazione su una lista, in una forma appropriata per i mazzi di carte.

Per fare un altro esempio, scriviamo anche un metodo per un Mazzo di nome *mescola*, usando la funzione *shuffle* contenuta nel modulo *random*:

```
# all'interno della classe Mazzo:
```

```
def mescola(self):
    random.shuffle(self.carte)
```

Non scordate di importare *random*.

Esercizio 18.2. *Scrivete un metodo per Mazzo di nome *ordina* che usi il metodo delle liste *sort* per ordinare le carte in un Mazzo. Per determinare il criterio di ordinamento, *sort* utilizza il metodo *__cmp__* che abbiamo definito.*

18.7 Ereditarietà

La caratteristica più frequentemente associata alla programmazione orientata agli oggetti è l'**ereditarietà**, che è la capacità di definire una nuova classe come versione modificata di una classe già esistente.

È chiamata “ereditarietà” perché la nuova classe eredita tutti i metodi della classe originale. Estendendo questa metafora, la classe originale è spesso definita **madre** (o superclasse) e la classe derivata **figlia** (o sottoclasse).

Come esempio, supponiamo di voler creare una classe che rappresenti una “mano” di carte, vale a dire un gruppo di carte distribuite a un giocatore. Una mano è simile a un mazzo: entrambi sono fatti di carte, ed entrambi richiedono operazioni come l'aggiunta e la rimozione di carte.

D'altra parte, ci sono altre operazioni che servono per la mano ma che non hanno senso per il mazzo. Nel poker, ad esempio, dobbiamo confrontare due mani per vedere quale vince. Nel bridge, è utile calcolare il punteggio della mano per decidere la dichiarazione.

Questo tipo di relazione tra classi—simili, ma non uguali—porta all'ereditarietà.

La definizione di una classe figlia è uguale alle altre, a parte il fatto che compare fra parentesi il nome della classe madre:

```
class Mano(Mazzo):
    """Rappresenta una mano di carte da gioco."""
```

Questa definizione indica che *Mano* eredita da *Mazzo*; ciò comporta che per *Mano* possiamo utilizzare i metodi di *Mazzo* come *togli_carta* e *aggiungi_carta*.

Mano eredita anche *__init__* da *Mazzo*, ma in questo caso il metodo non fa la cosa giusta: invece di popolare la mano con 52 nuove carte, il metodo *init* di *Mano* dovrebbe inizializzare *carte* con una lista vuota.

Ma se noi specifichiamo un nuovo metodo `init` nella classe `Mano`, esso andrà a sovrascrivere quello della classe madre `Mazzo`:

```
# all'interno della classe Mano:
```

```
def __init__(self, label=''):
    self.carte = []
    self.label = label
```

Allora, quando si crea una `Mano`, Python invoca questo metodo `init` specifico:

```
>>> mano = Mano('nuova mano')
>>> print mano.carte
[]
>>> print mano.label
nuova mano
```

Ma gli altri metodi vengono ereditati da `Mazzo`, pertanto possiamo usare `togli_carta` e `aggiungi_carta` per distribuire una carta:

```
>>> mazzo = Mazzo()
>>> carta = mazzo.togli_carta()
>>> mano.aggiungi_carta(carta)
>>> print mano
Re di Picche
```

Viene poi spontaneo incapsulare questo codice in un metodo di nome `sposta_carta`:

```
# all'interno della classe Mazzo:
```

```
def sposta_carta(self, mano, num):
    for i in range(num):
        mano.aggiungi_carta(self.togli_carta())
```

`sposta_carta` prende come argomenti un oggetto `Mano` e il numero di carte da distribuire. Modifica sia `self` che `mano`, e restituisce `None`.

In alcuni giochi, le carte si spostano da una mano all'altra, o da una mano di nuovo al mazzo. Potete usare `sposta_carta` per qualsiasi di queste operazioni: `self` può essere sia un `Mazzo` che una `Mano`, e `mano`, a dispetto del nome, può anche essere un `Mazzo`.

Esercizio 18.3. *Scrivete un metodo per `Mazzo` di nome `dai_mani` che prenda come parametri il numero di mani e il numero di carte da dare a ciascuna mano, e crei nuovi oggetti `Mano`, distribuisca il numero prefissato di carte a ogni mano e restituisca una lista di oggetti `Mano`.*

L'ereditarietà è una caratteristica utile. Certi programmi che sarebbero ripetitivi senza ereditarietà, possono invece essere scritti in modo più elegante. Facilita il riuso del codice, poiché potete personalizzare il comportamento delle superclassi senza doverle modificare. In certi casi, la struttura dell'ereditarietà rispecchia quella del problema, il che rende il programma più facile da capire.

D'altra parte, l'ereditarietà può rendere il programma difficile da leggere. Quando viene invocato un metodo, a volte non è chiaro dove trovare la sua definizione. Il codice rilevante può essere sparso tra moduli diversi. Inoltre, molte cose che possono essere fatte usando l'ereditarietà si possono fare anche, o talvolta pure meglio, senza di essa.

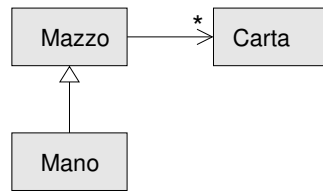


Figura 18.2: Diagramma di classe.

18.8 Diagrammi di classe

Sinora abbiamo visto i diagrammi di stack, che illustrano lo stato del programma, e i diagrammi di oggetto, che mostrano gli attributi di un oggetto e i loro valori. Questi diagrammi rappresentano una istantanea nell'esecuzione del programma, e quindi cambiano nel corso del programma.

Sono anche molto dettagliati, per alcuni scopi anche troppo. Un diagramma di classe è una rappresentazione più astratta della struttura di un programma. Invece di mostrare singoli oggetti, mostra le classi e le relazioni che sussistono tra le classi.

Ci sono alcuni tipi diversi di relazioni tra classi:

- Oggetti in una classe possono contenere riferimenti a oggetti in un'altra classe. Per esempio, ogni Rettangolo contiene un riferimento a un Punto, e ogni Mazzo contiene riferimenti a molte Carte. Questo tipo di relazione è chiamata **HAS-A** (ha-un), come in: "un Rettangolo ha un Punto".
- Una classe può ereditare da un'altra. Questa relazione è detta **IS-A** (è-un), come in: "una Mano è un tipo di Mazzo".
- Una classe può dipendere da un'altra, nel senso che modifiche in una classe richiedono modifiche anche nell'altra.

Un **diagramma di classe** è una rappresentazione grafica di queste relazioni. Per esempio, la Figura 18.2 mostra le relazioni tra Carta, Mazzo e Mano.

La freccia con un triangolo vuoto rappresenta la relazione IS-A: in questo caso indica che Mano eredita da Mazzo.

La freccia standard rappresenta la relazione HAS-A; in questo caso un Mazzo ha riferimenti agli oggetti Carta.

L'asterisco (*) vicino alla testa della freccia indica una **molteplicità**, cioè quante Carte ha un Mazzo. Una molteplicità può essere un numero semplice, come 52, un intervallo come 5..7, o un asterisco che indica che un Mazzo può contenere un numero qualsiasi di Carte.

Un diagramma più dettagliato dovrebbe evidenziare che un Mazzo contiene in realtà una *lista* di Carte, ma i tipi predefiniti come liste e dizionari di solito non vengono inclusi in questi diagrammi.

Esercizio 18.4. Leggete `TurtleWorld.py`, `World.py` e `Gui.py` e disegnatte un diagramma di classe che illustri le relazioni tra le classi ivi definite.

18.9 Debug

L'ereditarietà può rendere il debug difficoltoso, perché quando invocate un metodo su un oggetto, può capitare di non sapere esattamente quale sia il metodo che viene invocato.

Supponiamo che stiate scrivendo una funzione che lavori su oggetti Mano. Vorreste che fosse valida per Mani di tutti i tipi come ManiDiPoker, ManiDiBridge ecc. Se invocate un metodo come `mescola`, potrebbe essere quello definito in `Mazzo`, ma se qualcuna delle sottoclassi sovrascrive il metodo, avrete invece quella diversa versione.

Quando siete incerti sul flusso di esecuzione del vostro programma, la soluzione più semplice è aggiungere istruzioni di stampa all'inizio di ogni metodo importante. Se `Mazzo.mescola` stampa un messaggio come `Sto eseguendo Mazzo.mescola`, allora il programma traccia il flusso di esecuzione mentre viene eseguito.

In alternativa, potete usare la funzione seguente, che richiede un oggetto e un nome di metodo (come stringa) e restituisce la classe che contiene la definizione del metodo:

```
def trova_classe_def(obj, nome_metodo):
    for ty in type(obj).mro():
        if nome_metodo in ty.__dict__:
            return ty
```

Ecco un esempio:

```
>>> mano = Mano()
>>> print trova_classe_def(mano, 'mescola')
<class 'Carta.Mazzo'>
```

Quindi il metodo `mescola` di questa `Mano` è quello definito in `Mazzo`.

`trova_classe_def` usa il metodo `mro` per ricavare la lista degli oggetti classe (tipi) in cui verrà effettuata la ricerca dei metodi. “MRO” sta per *Method Resolution Order* (ordine di risoluzione dei metodi).

Un consiglio per la progettazione di un programma: ogni volta che sovrascrivete un metodo, l'interfaccia del nuovo metodo dovrebbe essere la stessa di quello sostituito: deve richiedere gli stessi parametri, restituire lo stesso tipo, rispettare le stesse precondizioni e postcondizioni. Se rispettate questa regola, vedrete che ogni funzione progettata per un'istanza di una superclasse, come `Mazzo`, funzionerà anche con le istanze delle sottoclassi come `Mano` o `ManoDiPoker`.

Se la violate, il vostro codice crollerà come (perdonatemi) un castello di carte.

18.10 Incapsulamento dei dati

Il capitolo 16 ha illustrato una tecnica di sviluppo detta “progettazione orientata agli oggetti”. Abbiamo identificato gli oggetti che ci servivano—`Tempo`, `Punto` e `Rettangolo`—e definito le classi per rappresentarli. Per ciascuno c'è un'evidente corrispondenza tra l'oggetto e una qualche entità del mondo reale (o per lo meno del mondo della matematica).

Ma altre volte la scelta degli oggetti e del modo in cui interagiscono è meno ovvia. In questo caso serve una tecnica di sviluppo diversa. Nella stessa maniera in cui abbiamo scoperto le interfacce delle funzioni per mezzo dell'incapsulamento e della generalizzazione, scopriamo ora le interfacce delle classi tramite l'**incapsulamento dei dati**.

L'analisi di Markov, vista nel Paragrafo 13.8, è un buon esempio. Se scaricate il mio codice dal sito <http://thinkpython.com/code/markov.py>, vi accorgerete che usa due variabili globali—`suffix_map` e `prefix`—che vengono lette e scritte da più funzioni.

```
suffix_map = {}
prefix = ()
```

Siccome queste variabili sono globali, possiamo eseguire una sola analisi alla volta. Se leggessimo due testi contemporaneamente, i loro prefissi e suffissi verrebbero aggiunti nella stessa struttura di dati (il che produce comunque alcuni interessanti testi generati).

Per eseguire analisi multiple mantenendole separate, possiamo incapsulare lo stato di ciascuna analisi in un oggetto. Ecco come si presenta:

```
class Markov(object):
```

```
    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

Poi, trasformiamo le funzioni in metodi. Ecco per esempio `elabora_parola`:

```
    def elabora_parola(self, parola, ordine=2):
        if len(self.prefix) < ordine:
            self.prefix += (parola,)
            return

        try:
            self.suffix_map[self.prefix].append(parola)
        except KeyError:
            # se non c'è una voce per questo prefisso, creane una
            self.suffix_map[self.prefix] = [parola]

        self.prefix = shift(self.prefix, parola)
```

Questa trasformazione di un programma—cambiarne la forma senza cambiarne le funzioni—è un altro esempio di refactoring (vedi Paragrafo 4.7).

L'esempio suggerisce una tecnica di sviluppo per progettare oggetti e metodi:

1. Cominciare scrivendo funzioni che leggono e scrivono variabili globali (dove necessario)
2. Una volta ottenuto un programma funzionante, cercare le associazioni tra le variabili globali e le funzioni che le usano.
3. Incapsulare le variabili correlate come attributi di un oggetto.
4. Trasformare le funzioni associate in metodi della nuova classe.

Esercizio 18.5. Scaricate il mio codice riferito al Paragrafo 13.8 (<http://thinkpython.com/code/markov.py>), e seguite i passi appena descritti per incapsulare le variabili globali come attributi di una nuova classe chiamata `Markov`. Soluzione: <http://thinkpython.com/code/Markov.py> (notare la *M* maiuscola).

18.11 Glossario

codificare: Rappresentare un insieme di valori usando un altro insieme di valori e costruendo una mappatura tra di essi.

attributo di classe: Attributo associato ad un oggetto classe. Gli attributi di classe sono definiti all'interno di una definizione di classe ma esternamente ad ogni metodo.

attributo di istanza: Attributo associato ad un'istanza di una classe.

impiallacciatura: Metodo o funzione che fornisce un'interfaccia diversa a un'altra funzione, senza tanti calcoli aggiuntivi.

ereditarietà: Capacità di definire una classe come versione modificata di una classe già definita in precedenza.

classe madre o superclasse: Classe dalla quale una classe figlia eredita.

classe figlia o sottoclasse: Nuova classe creata ereditando da una classe esistente.

relazione IS-A: Relazione tra una classe figlia e la sua classe madre.

relazione HAS-A: Relazione tra due classi dove le istanze di una classe contengono riferimenti alle istanze dell'altra classe.

diagramma di classe: Diagramma che illustra le classi di un programma e le relazioni tra di esse.

molteplicità: Notazione in un diagramma di classe che mostra, per una relazione HAS-A, quanti riferimenti ad istanze di un'altra classe ci sono.

18.12 Esercizi

Esercizio 18.6. *Quelle che seguono sono le possibili combinazioni nel gioco del poker, in ordine crescente di valore (e decrescente di probabilità):*

coppia: *due carte dello stesso valore*

doppia coppia: *due coppie di carte dello stesso valore*

tris: *tre carte dello stesso valore*

scala: *cinque carte con valori in sequenza (gli assi possono essere sia la carta di valore inferiore che quella di valore superiore, per cui Asso-2-3-4-5 è una scala, e anche 10-Fante-Regina-Re-Asso, ma non Regina-Re-Asso-2-3).*

colore: *cinque carte dello stesso seme*

full: *tre carte dello stesso valore più una coppia di carte dello stesso valore*

poker: *quattro carte dello stesso valore*

scala reale: *cinque carte dello stesso seme in scala (definita come sopra)*

Scopo di questo esercizio è stimare la probabilità di avere servita una di queste combinazioni.

1. Scaricate i file seguenti da <http://thinkpython.com/code/>:

Card.py : Versione completa delle classi Carta, Mazza e Mano di questo capitolo.

`PokerHand.py` : Implementazione incompleta di una classe che rappresenta una mano di poker con del codice di prova.

2. Se eseguite `PokerHand.py`, serve delle mani di sette carte e controlla se qualcuna contenga un colore. Leggete attentamente il codice prima di proseguire.
3. Aggiungete dei metodi a `PokerHand.py` di nome `ha_coppia`, `ha_doppiacoppia`, ecc. che restituiscano `True` o `False` a seconda che le mani soddisfino o meno il rispettivo criterio. Il codice deve funzionare indipendentemente dal numero di carte che contiene la mano (5 e 7 carte sono i casi più comuni).
4. Scrivete un metodo di nome `classifica` che riconosca la combinazione più elevata in una mano e imposta di conseguenza l'attributo `label`. Per esempio, una mano di 7 carte può contenere un colore e una coppia; deve essere etichettata "colore".
5. Quando siete sicuri che i vostri metodi di classificazione funzionano, il passo successivo è stimare la probabilità delle varie mani. Scrivete una funzione in `PokerHand.py` che mescoli un mazzo di carte, lo divida in mani, le classifichi e conti quante volte compare ciascuna combinazione.
6. Stampate una tabella delle combinazioni con le rispettive probabilità. Eseguite il vostro programma con numeri sempre più grandi di mani finché i valori ottenuti convergono ad un ragionevole grado di accuratezza. Confrontate i vostri risultati con i valori pubblicati su http://en.wikipedia.org/wiki/Hand_rankings.

Soluzione: <http://thinkpython.com/code/PokerHandSoln.py>.

Esercizio 18.7. Questo esercizio utilizza `Turtleworld` del Capitolo 4. Scriverete del codice per far giocare le Tartarughe ad Acchiapparella. Se non ne conoscete le regole, vedete http://it.wikipedia.org/wiki/Ce_l%27hai.

1. Scaricate <http://thinkpython.com/code/Wobbler.py> ed eseguitelo. Dovreste vedere un Mondo di Tartarughe con tre Tartarughe. Se premete il pulsante Run le Tartarughe vagano a caso.
2. Leggete il codice e accertatevi di capire come funziona. La classe `Wobbler` eredita da `Turtle`, il che significa che i metodi di `Turtle`: `lt`, `rt`, `fd` e `bk` funzionano in `Wobblers`.
Il metodo `step` viene invocato da `TurtleWorld`. Esso invoca `steer`, che gira la tartaruga nella direzione voluta, `wobble`, che provoca una rotazione casuale in proporzione alla goffaggine della tartaruga e `move`, che sposta la tartaruga di alcuni pixel in avanti a seconda della sua velocità.
3. Create un file di nome `Tagger.py`. Importate tutto da `Wobbler`, quindi definite una classe di nome `Tagger` che eredita da `Wobbler`. Chiamate `make_world` passando l'oggetto classe `Tagger` come argomento.
4. Aggiungete un metodo `steer` a `Tagger` per sovrascrivere quello in `Wobbler`. Per iniziare, scrivete una versione che faccia puntare sempre la tartaruga verso l'origine. Suggerimento: usate la funzione matematica `atan2` e gli attributi della Tartaruga `x`, `y` e `heading`.
5. Modificate `steer` in modo che le tartarughe stiano ai margini. Per il debug provate a usare il pulsante Step che invoca `step` una volta su ciascuna Tartaruga.

6. *Modify `steer` in modo che ciascuna Tartaruga punti verso quella più vicina. Suggerimento: le tartarughe hanno un attributo, `world`, che è un riferimento al mondo `TurtleWorld` in cui vivono, e `TurtleWorld` ha un attributo, `animals`, che è una lista di tutte le Tartarughe nel mondo.*
7. *Modify `steer` in modo che le tartarughe giochino ad Acchiapparella. Potete aggiungere dei metodi a `Tagger` e sovrascrivere `steer` e `__init__`, ma senza modificare o sovrascrivere `step`, `wobble` o `move`. Inoltre, `steer` può cambiare la direzione di moto della tartaruga ma non la posizione.*

Aggiustate le regole e il vostro metodo `steer` per una buona qualità del gioco: per esempio deve essere possibile che le tartarughe più lente tocchino quelle più veloci, all'occorrenza.

Soluzione: <http://thinkpython.com/code/Tagger.py>.

Capitolo 19

Esercitazione: Tkinter

19.1 Interfacce grafiche

I programmi che abbiamo visto sinora funzionano in modalità testuale, ma molti di quelli in circolazione usano delle **interfacce grafiche**, dette anche **GUI** (*Graphical User Interface*).

Per Python esistono alcune alternative per la scrittura di programmi basati su interfaccia grafica, come wxPython, Tkinter, e Qt. Ciascuna ha argomenti sia pro che contro, e per questo motivo Python non ha espresso uno standard.

Quello che prenderemo in considerazione è Tkinter, perché è il più semplice con cui iniziare. La maggior parte dei concetti che vedremo si potranno applicare anche agli altri moduli di GUI.

Esistono alcuni libri e pagine web dedicati a Tkinter. Una delle migliori risorse online è *An Introduction to Tkinter* di Fredrik Lundh.

Ho scritto un modulo chiamato `Gui.py` che fa parte di Swampy e contiene un'interfaccia semplificata alle funzioni e alle classi di Tkinter. Gli esempi di questo capitolo sono basati su questo modulo.

Ecco un semplice esempio che crea e visualizza un'interfaccia grafica: per crearla dovete importare `Gui` da Swampy:

```
from swampy.Gui import *
```

Oppure, a seconda di come avete installato Swampy, così:

```
from Gui import *
```

Quindi istanziate un oggetto `Gui`:

```
g = Gui()
g.title('Gui')
g.mainloop()
```

Eseguendo questo codice, dovrebbe comparire una finestra con un quadrato grigio vuoto e con il titolo `Gui`. `mainloop` avvia l'**evento ciclico**, che resta in attesa di una risposta da parte

dell'utente e quindi agisce di conseguenza. È una specie di ciclo infinito: viene eseguito finché l'utente non fa qualcosa o preme Control-C o provoca la chiusura del programma.

Questa Gui non fa gran che, perché non contiene alcun **controllo**. I controlli, o widget, sono gli elementi che caratterizzano un'interfaccia grafica e che includono:

Pulsante (Button): Un controllo, contenente testo o un'immagine, che esegue un'azione quando viene premuto.

Area (Canvas): Una regione in cui possono essere disegnate linee, figure geometriche e altre forme.

Casella di testo (Entry): Un'area in cui l'utente può inserire del testo.

Barra di scorrimento (Scrollbar): Un controllo che permette di scorrere la parte visibile di un altro controllo.

Riquadro (Frame): Un contenitore, spesso non visibile, che contiene altri controlli.

Il riquadro grigio vuoto che vedete quando create una Gui è un Frame. Quando create un nuovo controllo, verrà aggiunto dentro questo Frame.

19.2 Pulsanti e callback

Il metodo `bu` crea un controllo Button:

```
button = g.bu(text='Premi qui.')
```

Il valore di ritorno di `bu` è un oggetto Button. Il pulsante che compare nel Frame è una rappresentazione grafica di questo oggetto; potete controllare il comportamento del pulsante invocando dei metodi su di esso.

`bu` riceve fino a 32 parametri che ne controllano l'aspetto e il funzionamento. Questi parametri sono detti **opzioni**. Invece di dare valori a tutti i 32 parametri, è meglio usare gli argomenti con nome, come `text='Premi qui.'`, specificando così solo quelli che vi interessano e lasciando il valore predefinito per tutti gli altri.

Quando aggiungete un controllo al Frame, quest'ultimo gli si adatta, cioè si restringe alle dimensioni del pulsante. Se poi aggiungete altri controlli, il Frame crescerà in modo da sistemarli al suo interno.

Il metodo `la` crea un controllo etichetta (Label):

```
label = g.la(text='Premere il pulsante.')
```

Di default, Tkinter impila i controlli dall'alto verso il basso e centrati. Vedremo tra poco come variare questo comportamento.

Se premete il pulsante, noterete che non fa nulla. Questo perché non lo avete ancora "collegato", cioè non gli avete spiegato cosa fare!

L'opzione che controlla il comportamento di un pulsante è `command`. Il valore di `command` è una funzione che viene eseguita quando il pulsante viene premuto. Per esempio, ecco una funzione che crea una nuova etichetta:

```
def crea_label():
    g.la(text='Grazie.')
```

Ora possiamo creare un pulsante che ha questa funzione come suo command:

```
button2 = g.bu(text='No, premi qui!', command=crea_label)
```

Quando premete questo pulsante, viene eseguita `crea_label`, e compare una nuova etichetta.

Il valore dell'opzione `command` è un oggetto funzione che viene chiamato **callback** perché dopo aver chiamato `bu` per creare il pulsante, il flusso di esecuzione “chiama all'indietro” quando l'utente preme il pulsante.

Questo tipo di flusso è caratteristico della **programmazione orientata agli eventi**. Le azioni dell'utente, come premere pulsanti o tasti, sono dette **eventi**. Nella programmazione orientata agli eventi, il flusso di esecuzione è determinato dalle azioni di chi usa il programma piuttosto che dal programmatore.

La difficoltà di questo tipo di programmazione è costruire un insieme di controlli e callback che funzionino correttamente (o almeno generino appropriati messaggi di errore) per qualsiasi possibile sequenza di azioni dell'utente.

Esercizio 19.1. *Scrivete un programma che crea una GUI con un singolo pulsante. Se si preme il pulsante, deve creare un secondo pulsante. Quando si preme il nuovo pulsante, deve creare un'etichetta con scritto “Bel lavoro!”.*

Cosa succede se premete i pulsanti più di una volta? Soluzione: http://thinkpython.com/code/button_demo.py

19.3 Controlli Canvas

Il controllo Canvas è uno dei più versatili: crea un'area in cui è possibile disegnare linee, cerchi e altre forme. Se avete svolto l'Esercizio 15.4 dovrete già avere un po' di familiarità con controlli di questo tipo.

Per creare una nuova area usiamo il metodo `ca`:

```
canvas = g.ca(width=500, height=500)
```

`width` e `height` sono la larghezza e l'altezza dell'area espressi in pixel.

Dopo aver creato il controllo, potete ancora cambiarne i valori delle opzioni mediante il metodo `config`. Per esempio, l'opzione `bg` cambia il colore dello sfondo:

```
canvas.config(bg='white')
```

Il valore di `bg` è una stringa che è il nome di un colore. L'insieme dei valori di colore validi è diverso tra le varie implementazioni di Python, ma tutte comprendono almeno questi:

```
white    black
red      green    blue
cyan     yellow   magenta
```

Le forme in un controllo Canvas sono chiamate **elementi o item**. Per esempio il metodo `circle` disegna (avete già indovinato) un cerchio:

```
item = canvas.circle([0,0], 100, fill='red')
```

Il primo argomento è una coppia di coordinate che specifica il centro del cerchio; il secondo è il raggio.

Gui.py contiene un sistema standard di coordinate cartesiane con l'origine al centro dell'area e l'asse *y* orientato verso l'alto (valori crescenti dal basso verso l'alto). In altri sistemi grafici è diverso: l'origine è l'angolo in alto a sinistra e l'asse *y* è orientato verso il basso.

L'opzione `fill` specifica che il cerchio va riempito di rosso.

Il valore di ritorno di `circle` è un oggetto `Item` che contiene metodi per modificare l'elemento nell'area. Potete per esempio usare `config` per cambiare qualunque opzione del cerchio:

```
item.config(fill='yellow', outline='orange', width=10)
```

`width` è lo spessore del contorno in pixel; `outline` è il suo colore.

Esercizio 19.2. *Scrivete un programma che crei un'area di disegno e un pulsante; premendo il pulsante, deve essere disegnato un cerchio nell'area.*

19.4 Sequenze di coordinate

Il metodo `rectangle` ha come parametro una sequenza di coordinate che specificano gli angoli opposti del rettangolo. Questo esempio disegna un rettangolo blu con l'angolo in basso a sinistra sull'origine e quello in alto a destra nel punto (200, 100):

```
canvas.rectangle([[0, 0], [200, 100]],
                 fill='blue', outline='orange', width=10)
```

Questo modo di specificare gli angoli è chiamato **contenitore** perché i due punti determinano i limiti del rettangolo.

`oval` richiede come parametro un contenitore e disegna un ovale interno al rettangolo specificato:

```
canvas.oval([[0, 0], [200, 100]], outline='orange', width=10)
```

`line` prende una sequenza di coordinate e disegna una spezzata che collega tutti i punti. Questo esempio disegna due lati di un triangolo:

```
canvas.line([0, 100], [100, 200], [200, 100], width=10)
```

`polygon` prende gli stessi parametri, ma disegna anche l'ultimo lato di chiusura del poligono (se necessario) e lo riempie:

```
canvas.polygon([0, 100], [100, 200], [200, 100],
               fill='red', outline='orange', width=10)
```

19.5 Altri controlli

Tkinter comprende due controlli che permettono all'utente di inserire del testo: una casella di testo semplice a riga singola (controllo `Entry`), e una casella di testo a righe multiple (controllo `Text`).

`en` crea un nuovo controllo `Entry`:


```
entry = g.en(text='Testo di prova.')
```

L'opzione `text` permette di inserire nella casella un testo predefinito, non appena viene creata. Il metodo `get` legge e restituisce il contenuto della casella (che può essere stato cambiato dall'utente):

```
>>> entry.get()
'Testo di prova.'
```

`te` crea un controllo `Text`:

```
text = g.te(width=100, height=5)
```

`width` e `height` sono le dimensioni della casella, espresse rispettivamente in caratteri e righe.

`insert` inserisce del testo nel controllo `Text`:

```
text.insert(END, 'Una riga di testo.')
```

`END` è un indice speciale che indica l'ultimo carattere presente nella casella di testo.

Potete anche specificare una posizione usando un indice puntato, come `1.1`, dove il numero prima del punto è la riga e quello dopo il punto è la colonna. L'esempio che segue aggiunge le lettere `'altr'` dopo il secondo carattere della prima riga.

```
>>> text.insert(1.2, "'altr'")
```

Il metodo `get` legge il testo nel controllo; prende come argomenti un indice di inizio e uno di fine. L'esempio seguente restituisce tutto il testo nel controllo, incluso il carattere di ritorno a capo:

```
>>> text.get(0.0, END)
'Un'altra riga di testo.\n'
```

Il metodo `delete` cancella il testo dal controllo; questo esempio cancella tutto, tranne i primi due caratteri del testo:

```
>>> text.delete(1.2, END)
>>> text.get(0.0, END)
'Un\n'
```

Esercizio 19.3. *Modificate la soluzione dell'Esercizio 19.2 aggiungendo un controllo `Entry` e un secondo pulsante. Quando l'utente preme il secondo pulsante, il programma deve leggere il nome di un colore dalla casella di testo e usarlo per cambiare il colore di riempimento del cerchio. Usate `config` per modificare il cerchio esistente, senza crearne uno nuovo.*

Il programma deve gestire i casi in cui l'utente cerca di cambiare il colore di un cerchio che non è ancora stato creato, oppure il nome del colore non è valido.

Potete scaricare la mia soluzione dal sito http://thinkpython.com/code/circle_demo.py.

19.6 Packing dei controlli

Sinora abbiamo impilato i controlli uno sull'altro, su una sola colonna, ma di norma l'aspetto delle interfacce grafiche è più articolato. Per esempio, la Figura 19.1 mostra una versione semplificata di `TurtleWorld` (vedere Capitolo 4).

In questo Paragrafo illustriamo il codice che crea questa GUI, passo dopo passo. Potete scaricare l'esempio completo da <http://thinkpython.com/code/SimpleTurtleWorld.py>.

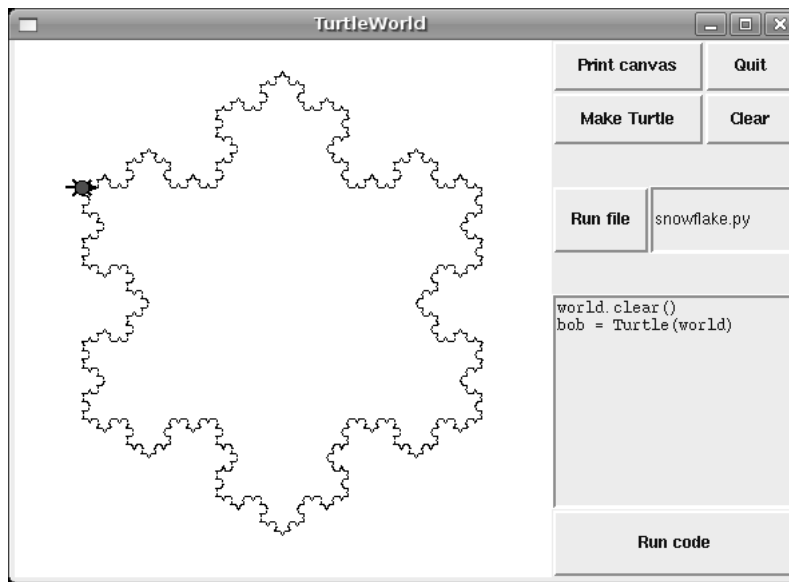


Figura 19.1: TurtleWorld dopo l'esecuzione del codice che disegna un fiocco di neve.

Al livello superiore, questa GUI contiene due controlli—un Canvas e un Frame—disposti affiancati su una riga. Il primo passo è quindi creare la riga.

```
class SimpleTurtleWorld(TurtleWorld):
    """Questa classe è identica a TurtleWorld, ma il codice che
       compone la GUI è semplificato per motivi didattici."""

    def setup(self):
        self.row()
        ...
```

setup è la funzione che crea e dispone i controlli. La disposizione dei controlli in una GUI è chiamata **packing**.

row crea un Frame riga e lo rende il "Frame attivo". Finché questo frame non viene chiuso o un altro Frame non viene creato, tutti i controlli successivi verranno disposti su una riga.

Ecco il codice che crea il controllo Canvas e il Frame colonna che contiene gli altri controlli:

```
self.canvas = self.ca(width=400, height=400, bg='white')
self.col()
```

Il primo controllo nella colonna è un Frame griglia che contiene quattro pulsanti disposti a coppie:

```
self.gr(cols=2)
self.bu(text='Print canvas', command=self.canvas.dump)
self.bu(text='Quit', command=self.quit)
self.bu(text='Make Turtle', command=self.make_turtle)
self.bu(text='Clear', command=self.clear)
self.endgr()
```

gr crea la griglia; l'argomento è il numero delle colonne. Nella griglia, i controlli vengono disposti nell'ordine da sinistra verso destra e dall'alto verso il basso.

Il primo pulsante usa `self.canvas.dump` come callback; il secondo usa `self.quit`. Questi sono **metodi collegati**, cioè metodi associati ad uno specifico oggetto. Quando vengono invocati, lo sono sull'oggetto.

Il metodo successivo nella colonna è un Frame riga che contiene un pulsante e una casella di testo semplice:

```
self.row([0,1], pady=30)
self.bu(text='Run file', command=self.run_file)
self.en_file = self.en(text='snowflake.py', width=5)
self.endrow()
```

Il primo argomento di `row` è una lista di pesi che determina come viene ripartito lo spazio sovrabbondante tra i controlli presenti. La lista `[0,1]` significa che tutto lo spazio extra viene assegnato al secondo controllo, cioè alla casella di testo. Se eseguite il codice e massimizzate la finestra, vedrete che la casella di testo si espande ma il pulsante no.

L'opzione `pady` allarga la riga in direzione dell'asse *y*, aggiungendo 30 pixel di spazio sopra e sotto.

`endrow` chiude questa riga di controlli, pertanto quelli seguenti verranno inseriti nel Frame colonna. `Gui.py` mantiene una coda dei Frame:

- Quando usate `row`, `col` o `gr` per creare un Frame, questo va in cima alla coda e diventa il Frame attivo.
- Quando usate `endrow`, `endcol` o `endgr` per chiudere un Frame, questo viene eliminato dalla coda, e il Frame che precede nella coda diventa il Frame attivo.

Il metodo `run_file` legge il contenuto della casella di testo, lo usa come nome di file, legge i contenuti del file e li passa a `run_code`. `self.inter` è un oggetto Interprete capace di prendere una stringa ed eseguirla come codice Python.

```
def run_file(self):
    filename = self.en_file.get()
    fp = open(filename)
    source = fp.read()
    self.inter.run_code(source, filename)
```

Gli ultimi due controlli sono una casella di testo `Text` e un pulsante:

```
self.te_code = self.te(width=25, height=10)
self.te_code.insert(END, 'world.clear()\n')
self.te_code.insert(END, 'bob = Turtle(world)\n')

self.bu(text='Run code', command=self.run_text)
```

`run_text` è simile a `run_file` a parte il fatto che legge il codice da un controllo `Text` anziché da un file:

```
def run_text(self):
    source = self.te_code.get(1.0, END)
    self.inter.run_code(source, '<user-provided code>')
```

Sfortunatamente, i dettagli della disposizione dei controlli sono diversi in altri linguaggi e tra i diversi moduli di Python. Tkinter stesso ha tre meccanismi diversi per disporre i controlli, detti **gestori di geometria**. Quello che ho illustrato in questo Capitolo è il gestore di geometria “a griglia”; gli altri sono chiamati “a pacchetto” e “a posizione”.

In compenso, molti dei concetti di questo Capitolo si applicano anche ad altri moduli GUI e ad altri linguaggi.

19.7 Menu e oggetti Callable

Un Menubutton è un controllo che ha l'aspetto di un pulsante, ma che quando viene premuto visualizza un menu. Dopo che l'utente ha scelto un elemento, il menu scompare.

Questo codice (scaricabile da http://thinkpython.com/code/menubutton_demo.py) crea un Menubutton che propone una scelta di colori:

```
g = Gui()
g.la('Scegliere un colore:')
colors = ['red', 'green', 'blue']
mb = g.mb(text=colors[0])
```

mb crea il Menubutton. In prima battuta, il testo del pulsante è il nome del colore predefinito. Il ciclo seguente crea un elemento di menu per ciascun colore:

```
for color in colors:
    g.mi(mb, text=color, command=Callable(set_color, color))
```

Il primo argomento di mi è il Menubutton al quale sono associati gli elementi di menu.

L'opzione `command` è un oggetto Callable, che è un qualcosa di nuovo. Finora abbiamo visto funzioni e metodi collegati usati come callback, che funzionano bene, a patto di non dover passare argomenti alla funzione. Altrimenti, dovete costruire un oggetto Callable che contiene una funzione, come `set_color`, e i suoi argomenti, come `color`.

L'oggetto Callable mantiene un riferimento alla funzione e agli argomenti sotto forma di attributi. Poi, quando l'utente fa click su un elemento del menu, il callback chiama la funzione e le passa gli argomenti così archiviati.

Ecco come si presenta `set_color`:

```
def set_color(color):
    mb.config(text=color)
    print color
```

Quando l'utente seleziona una voce di menu e viene chiamato `set_color`, questo configura il Menubutton per visualizzare il colore scelto. Stampa inoltre il nome del colore; se provate questo esempio, avrete la conferma che `set_color` viene chiamato quando selezionate una voce (e *non* quando create l'oggetto Callable).

19.8 Binding

Un **binding** è un'associazione tra un controllo, un evento e un callback: quando un evento (come la pressione di un pulsante) avviene su un controllo, viene invocato il callback.

Molti controlli hanno dei binding di default. Ad esempio, premendo un pulsante, il binding di default cambia l'effetto in rilievo del pulsante facendolo sembrare incassato. Quando si rilascia il pulsante, il binding ripristina l'aspetto del pulsante e chiama il callback specificato con l'opzione `command`.

Potete usare il metodo `bind` per sovrascrivere i binding di default o per aggiungerne di nuovi. Per esempio, il codice seguente crea un binding per un oggetto Canvas (potete scaricare il codice di questo Paragrafo da http://thinkpython.com/code/draggable_demo.py):

```
ca.bind('<ButtonPress-1>', make_circle)
```

Il primo argomento è una stringa evento; questo evento è scatenato quando l'utente preme il pulsante sinistro del mouse. Altri eventi legati al mouse sono `ButtonMotion`, `ButtonRelease` e `Double-Button`.

Il secondo argomento è un gestore di evento, che è una funzione o un metodo collegato, come un callback, ma una differenza importante è che un gestore di evento prende come parametro un oggetto Evento. Un esempio:

```
def make_circle(event):
    pos = ca.canvas_coords([event.x, event.y])
    item = ca.circle(pos, 5, fill='red')
```

L'oggetto Evento contiene informazioni sul tipo di evento e dettagli come le coordinate del puntatore del Mouse. In questo esempio, le informazioni di cui abbiamo bisogno è l'ubicazione del click del mouse. Questi valori sono in "coordinate pixel", che sono definite dal sistema grafico sottostante. Il metodo `canvas_coords` le trasforma in "coordinate del Canvas", che sono compatibili con i metodi dell'oggetto Canvas come `circle`.

I controlli Entry sono frequentemente collegati all'evento `<Return>` che si verifica quando l'utente preme il tasto Invio o Return. Ad esempio il codice seguente crea un Button e un Entry.

```
bu = g.bu('Make text item:', make_text)
en = g.en()
en.bind('<Return>', make_text)
```

`make_text` viene chiamato quando viene premuto il pulsante o quando l'utente preme Invio dopo aver scritto qualcosa nella casella di testo. Per fare funzionare il tutto, ci serve una funzione che possa essere chiamata come command (senza argomenti) o gestore di eventi (con un Evento come argomento):

```
def make_text(event=None):
    text = en.get()
    item = ca.text([0,0], text)
```

`make_text` prende il contenuto della casella di testo Entry e lo visualizza come elemento di testo nel controllo Canvas.

È anche possibile creare dei binding per gli elementi (Item) di un Canvas. Quella che segue è una definizione di classe per `Draggable`, una sottoclasse di `Item` che fornisce binding che implementano capacità di drag-and-drop.

```
class Draggable(Item):

    def __init__(self, item):
        self.canvas = item.canvas
        self.tag = item.tag
        self.bind('<Button-3>', self.select)
        self.bind('<B3-Motion>', self.drag)
        self.bind('<Release-3>', self.drop)
```

Il metodo `init` prende un oggetto Item come parametro. Copia gli attributi di Item e quindi crea dei binding per tre eventi: la pressione di un pulsante, lo spostamento del pulsante e il rilascio del pulsante.

Il gestore di eventi `select` registra le coordinate dell'evento corrente e il colore iniziale dell'elemento, quindi cambia il colore in giallo:

```
def select(self, event):
    self.dragx = event.x
    self.dragy = event.y

    self.fill = self.cget('fill')
    self.config(fill='yellow')
```

cget sta per “get configuration”; prende il nome di un’opzione come stringa e restituisce il valore attuale di quell’opzione.

drag calcola di quanto l’oggetto è stato spostato rispetto al punto iniziale, aggiorna le coordinate, quindi sposta l’elemento.

```
def drag(self, event):
    dx = event.x - self.dragx
    dy = event.y - self.dragy

    self.dragx = event.x
    self.dragy = event.y

    self.move(dx, dy)
```

Questo calcolo è fatto in coordinate pixel; non è necessaria la conversione in coordinate del Canvas.

Infine, drop ripristina il colore iniziale dell’elemento:

```
def drop(self, event):
    self.config(fill=self.fill)
```

Potete usare la classe Draggable per aggiungere funzionalità drag-and-drop ad un elemento esistente. Per esempio, questa è una versione modificata di `make_circle` che usa `circle` per creare un Item e `Draggable` per renderlo trascinabile:

```
def make_circle(event):
    pos = ca.canvas_coords([event.x, event.y])
    item = ca.circle(pos, 5, fill='red')
    item = Draggable(item)
```

Questo esempio dimostra uno dei benefici dell’ereditarietà: potete modificare le proprietà di una classe madre senza modificare la sua definizione. Cosa particolarmente utile se volete cambiare il comportamento definito in un modulo che non avete scritto voi.

19.9 Debug

Una delle problematiche della programmazione di GUI è tenere traccia di ciò che succede mentre la GUI viene costruita e di ciò che succede poi in risposta alle azioni dell’utente.

Per esempio, quando impostate un callback, un errore frequente è chiamare la funzione invece di passarle un riferimento:

```
def il_callback():
    print 'Chiamato.'
```

```
g.bu(text='Sbagliato!', command=il_callback())
```

Se eseguite questo codice, vedrete che chiama `il_callback` immediatamente, e *dopo* crea il pulsante. Quando premete il pulsante, non succede nulla perché il valore di ritorno di `il_callback` è `None`. In genere, non dovete invocare un callback mentre state impostando la GUI; va invocato solo in risposta ad un evento generato dall'utente.

Un altro problema della programmazione di GUI è che non avete controllo sul flusso di esecuzione. Sono le azioni dell'utente a determinare quali parti del programma vengono eseguite e in che ordine. Questo significa che dovete progettare il programma in modo che funzioni correttamente per tutte le possibili sequenze di eventi.

Ad esempio, la GUI nell'Esercizio 19.3 ha due controlli: uno crea un elemento `Circle` e l'altro ne cambia il colore. Se l'utente crea il cerchio e poi cambia il colore va tutto bene. Ma cosa succede se l'utente cambia il colore di un cerchio che non esiste ancora? O crea più di un cerchio?

Più cresce il numero di controlli, più risulta difficile prevedere tutte le possibili sequenze di eventi. Un modo di gestire questa complessità è incapsulare la situazione del sistema in un oggetto e quindi considerare:

- Quali sono le possibili situazioni? Nell'esempio del cerchio, dobbiamo considerarne due: prima e dopo che l'utente ha creato il primo cerchio.
- In ciascuna situazione, che eventi possono verificarsi? Nell'esempio, l'utente può premere uno dei due pulsanti o uscire dal programma.
- Per ogni coppia situazione-evento, qual è il risultato desiderato? Siccome ci sono due situazioni e due pulsanti, ci sono quattro coppie situazione-evento da considerare.
- Cosa può causare una transizione da una situazione a un'altra? In questo caso, c'è una transizione quando l'utente crea il primo cerchio.

Può essere anche utile definire e controllare delle invarianti che dovrebbero valere indipendentemente dalla sequenza di eventi.

Questo approccio alla programmazione di GUI vi può aiutare a scrivere del codice corretto, senza perdere tempo a provare ogni possibile sequenza di eventi generati dall'utente!

19.10 Glossario

GUI: Un'interfaccia grafica utente.

controllo o widget: Uno degli elementi che costituiscono una GUI, come pulsanti, menu, caselle di testo, ecc.

opzione: Un valore da cui dipende l'aspetto o la funzione di un controllo.

argomento con nome: Argomento che indica il nome del parametro che fa parte della chiamata di una funzione.

callback: Funzione associata ad un controllo che viene chiamata quando l'utente esegue una determinata azione.

metodo collegato: Metodo associato ad una particolare istanza.

programmazione ad eventi: Stile di programmazione in cui il flusso di esecuzione è determinato dalle azioni dell'utente.

evento: Un'azione dell'utente, come un click del mouse o la pressione di un tasto, che causa una risposta della GUI.

evento ciclico: Ciclo infinito che attende un'azione dell'utente e risponde.

elemento o item: Elemento grafico su un controllo Canvas.

contenitore: Rettangolo che include un insieme di elementi, di solito identificato mediante due angoli opposti.

packing: Disporre gli elementi su una GUI.

gestore di geometria: Sistema di packing dei controlli in una GUI.

binding: Associazione tra un controllo, un evento e un gestore di evento. Il gestore di evento viene chiamato quando si verifica l'evento nel controllo.

19.11 Esercizi

Esercizio 19.4. *In questo esercizio, scriverete un visualizzatore di immagini. Ecco un semplice esempio:*

```
g = Gui()
canvas = g.ca(width=300)
photo = PhotoImage(file='danger.gif')
canvas.image([0,0], image=photo)
g.mainloop()
```

PhotoImage legge un file e restituisce un oggetto PhotoImage che Tkinter è in grado di visualizzare. Canvas.image dispone l'immagine nell'area di lavoro, centrata sulle coordinate date. Potete mettere immagini anche su etichette, pulsanti, e qualche altro controllo:

```
g.la(image=photo)
g.bu(image=photo)
```

PhotoImage può gestire solo pochi formati di immagine, come GIF e PPM, ma possiamo far uso della Python Imaging Library (PIL) per leggerne altri.

Il nome del modulo PIL è Image, ma Tkinter definisce già un oggetto con lo stesso nome. Per evitare conflitti, dovete usare import...as in questo modo:

```
import Image as PIL
import ImageTk
```

La prima riga importa Image assegnandogli il nome locale PIL. La seconda importa ImageTk, che è in grado di trasformare un'immagine PIL in un PhotoImage di Tkinter. Ecco un esempio:

```
image = PIL.open('allen.png')
photo2 = ImageTk.PhotoImage(image)
g.la(image=photo2)
```

1. Scaricate `image_demo.py`, `danger.gif` e `allen.png` da <http://thinkpython.com/code>. Eseguite `image_demo.py`. Probabilmente dovrete installare PIL e ImageTk, che dovrebbero essere presenti nel vostro repository software, altrimenti li potete scaricare da <http://pythonware.com/products/pil>.

2. In `image_demo.py` cambiate il nome del secondo `PhotoImage` da `photo2` a `photo` ed eseguite di nuovo il programma. Dovreste riuscire a vedere il secondo `PhotoImage` ma non il primo.

Il problema sta nel fatto che quando riassegnate `photo`, sovrascrive il riferimento al primo `PhotoImage`, che così sparisce. Lo stesso capita se assegnate un `PhotoImage` a una variabile locale: sparisce quando la funzione termina.

Per evitare questo problema dovete registrare un riferimento ad ogni `PhotoImage` che volete mantenere. Potete usare una variabile globale, o conservarli in una struttura di dati o come attributi di un oggetto.

Questo comportamento può dare dei fastidi, ed è il motivo per cui vi sto mettendo sull'avviso (e l'immagine di esempio dice "Pericolo!").

3. Partendo da questo esempio, scrivete un programma che prenda il nome di una directory e analizzi con un ciclo tutti i file, mostrando quelli che PIL riconosce come immagini. Potete usare un'istruzione `try` per gestire i file non riconosciuti da PIL.

Quando l'utente fa clic sull'immagine, il programma deve mostrare quella successiva.

4. PIL fornisce svariati metodi per trattare le immagini. Potete saperne di più leggendo `http://pythonware.com/library/pil/handbook`. Per i più bravi, scegliete alcuni di questi metodi e create una GUI per applicarli alle immagini.

Soluzione: `http://thinkpython.com/code/ImageBrowser.py`.

Esercizio 19.5. Un editor di grafica vettoriale è un programma che consente di disegnare e modificare forme sullo schermo e generare file di output in formati grafici vettoriali come Postscript e SVG.

Scrivete un semplice editor di grafica vettoriale usando Tkinter. Per lo meno, dovrebbe consentire all'utente di disegnare linee, cerchi e rettangoli, e usare `Canvas.dump` per generare una descrizione Postscript dei contenuti del `Canvas`.

I più coraggiosi possono anche consentire all'utente di selezionare e ridimensionare gli elementi sul `Canvas`.

Esercizio 19.6. Usate Tkinter per scrivere un web browser basilare. Deve avere un controllo `Text` dove l'utente può scrivere l'indirizzo web e un `Canvas` per mostrare il contenuto della pagina.

Per scaricare i file usate il modulo `urllib` (vedere Esercizio 14.6) e per analizzare i tag HTML il modulo `HTMLParser` (vedere `http://docs.python.org/2/library/htmlparser.html`).

Come minimo il vostro web browser dovrebbe gestire il testo semplice e i collegamenti. Se volete fare di più, anche i colori di sfondo, i tag di formattazione del testo e le immagini.

Appendice A

Debug

In un programma si possono verificare diversi tipi di errore, ed è utile distinguerli in modo da poterli rintracciare più velocemente:

- Gli errori di sintassi sono prodotti da Python quando traduce il codice sorgente in codice macchina. Di solito, indicano che c'è qualcosa di sbagliato nella sintassi del programma. Esempio: omettere i due punti alla fine di una istruzione `def` provoca il messaggio, un po' ridondante, `SyntaxError: invalid syntax`.
- Gli errori di runtime sono prodotti dall'interprete se qualcosa va storto durante l'esecuzione del programma. Nella maggior parte dei casi, il messaggio di errore specifica dove si è verificato l'errore e quali funzioni erano in esecuzione. Esempio: una ricorsione infinita causa un errore di runtime `"maximum recursion depth exceeded"`.
- Gli errori di semantica sono dei problemi con un programma che viene eseguito senza produrre messaggi di errore, ma che non fa le cose nel modo giusto. Esempio: un'espressione può essere valutata secondo un ordine diverso da quello che si intendeva, generando un risultato non corretto.

La prima cosa da fare nel debug è capire con che tipo di errore abbiamo a che fare. Anche se i paragrafi che seguono sono organizzati per tipo di errore, alcune tecniche sono applicabili in più di una situazione.

A.1 Errori di sintassi

Gli errori di sintassi sono in genere facili da sistemare, una volta capito in cosa consistono. Purtroppo, il messaggio di errore spesso è poco utile. Quelli più comuni sono: `SyntaxError: invalid syntax` e `SyntaxError: invalid token`, nessuno dei quali è molto esplicativo.

In compenso, il messaggio comunica il punto del programma dove si è verificato il problema. Più precisamente, dice dove Python ha notato il problema, che non è necessariamente il punto in cui si trova l'errore. A volte l'errore è prima del punto indicato dal messaggio, spesso nella riga precedente.

Se state costruendo il programma in modo incrementale, è molto probabile che l'errore sia nell'ultima riga che avete aggiunto.

Se state copiando il codice da un libro, cominciate confrontando attentamente il vostro codice con quello del libro. Controllate ogni carattere. Ricordate però che anche il libro può essere sbagliato, e se vedete qualcosa che somiglia a un errore di sintassi, potrebbe esserlo.

Ecco alcuni modi di evitare i più comuni errori di sintassi:

1. Accertatevi di non usare parole chiave di Python come nomi di variabile.
2. Controllate che ci siano i due punti alla fine di ogni intestazione di ogni istruzione composta, inclusi `for`, `while`, `if`, e le istruzioni `def`.
3. Accertatevi che ogni stringa nel codice sia racchiusa da una coppia di virgolette o apici.
4. Se avete stringhe a righe multiple con triple virgolette, accertatevi di averle chiuse in modo appropriato. Una stringa non chiusa può causare un errore di `invalid token` al termine del programma, oppure può trattare la parte che segue del programma come fosse una stringa, finché non incontra la stringa successiva. Nel secondo caso, potrebbe anche non produrre affatto un messaggio di errore!
5. Un operatore di apertura non chiuso—`(`, `{`, o `[`—fa sì che Python consideri la riga successiva come parte dell'istruzione corrente. In genere, si verifica un errore nella riga immediatamente successiva.
6. Controllate che non ci sia il classico `=` al posto di `==` all'interno di un'istruzione condizionale.
7. Controllate l'indentazione per assicurarvi che allinei il codice nel modo corretto. Python può gestire sia spazi che tabulazioni, ma se li mescolate possono esserci problemi. Il modo migliore di evitarli è usare un editor di testo appositamente realizzato per Python, che gestisca l'indentazione in modo coerente.

Se nulla di tutto questo funziona, continuate con il paragrafo seguente...

A.1.1 Continuo a fare modifiche ma non cambia nulla.

Se l'interprete dice che c'è un errore ma non lo trovate, può essere che voi e l'interprete non stiate guardando lo stesso codice. Controllate il vostro ambiente di programmazione per assicurarvi che il programma che state modificando sia proprio quello che Python sta tentando di eseguire.

Se non ne siete certi, provate a mettere deliberatamente un evidente errore di sintassi all'inizio del programma e rieseguitelo. Se l'interprete non trova l'errore, non state eseguendo il nuovo codice.

Alcune cause possibili:

- Avete modificato il file e dimenticato di salvare le modifiche prima dell'esecuzione. Alcuni ambienti di programmazione lo fanno automaticamente, ma altri no.

- Avete cambiato il nome del file, ma state eseguendo ancora quello con il vecchio nome.
- Qualcosa nel vostro ambiente di sviluppo non è configurato correttamente.
- Se state scrivendo un modulo usando `import`, accertatevi di non dare al vostro modulo lo stesso nome di uno dei moduli standard di Python.
- Se state usando `import` per leggere un modulo, ricordatevi che dovete riavviare l'interprete o usare `reload` per leggere un file modificato. Se importate nuovamente il modulo, non succederà nulla.

Se vi bloccate e non riuscite a capire cosa sta succedendo, un'alternativa è ripartire con un nuovo programma come "Ciao, mondo!", e accertarvi di avere un programma ben conosciuto da eseguire. Quindi, aggiungete gradualmente i pezzi del programma originale a quello nuovo.

A.2 Errori di runtime

Una volta che il programma è sintatticamente corretto, Python può compilarlo e quantomeno cominciare ad eseguirlo. Cosa può succedere di spiacevole?

A.2.1 Il programma non fa assolutamente nulla.

È il problema più frequente se il vostro file consiste di funzioni e classi, ma in realtà non invoca nulla per avviare l'esecuzione. Può essere una cosa intenzionale, se intendete utilizzarlo solo come modulo da importare allo scopo di fornire le classi e le funzioni.

Se non è questo il caso, assicuratevi che venga invocata una funzione per avviare l'esecuzione, o eseguitene una dal prompt interattivo. Vedete anche il paragrafo "Flusso di esecuzione" più avanti.

A.2.2 Il programma si blocca.

Se un programma si blocca e pare che non stia succedendo nulla, spesso vuol dire che è incappato in un ciclo infinito o in una ricorsione infinita.

- Se c'è un ciclo particolare dove sospettate che stia il problema, aggiungete un'istruzione di stampa immediatamente prima del ciclo che dice: "Sto entrando nel ciclo" e una immediatamente dopo che dice: "Sto uscendo dal ciclo".

Eseguite il programma. Se viene visualizzato il primo messaggio ma non il secondo, c'è un ciclo infinito. Proseguite con il Paragrafo "Ciclo infinito" più sotto.

- Il più delle volte, in presenza di una ricorsione infinita, il programma funziona per qualche tempo per poi produrre un errore "RuntimeError: Maximum recursion depth exceeded". Se succede questo, andate al Paragrafo "Ricorsione infinita".

Se non ottenete questo errore ma sospettate che ci sia un problema con un metodo o funzione ricorsivi, potete usare ugualmente le tecniche illustrate nel Paragrafo "Ricorsione infinita".

- Se nessuno di questi punti funziona, fate delle prove su altri cicli o funzioni e metodi ricorsivi.
- Se ancora non funziona, forse non avete ben chiaro il flusso di esecuzione del vostro programma. Andate al relativo Paragrafo.

Ciclo infinito

Se sospettate che vi sia un ciclo infinito e di sapere quale ciclo in particolare stia causando il problema, aggiungete un'istruzione di stampa alla fine del ciclo in modo da visualizzare il valore delle variabili nella condizione e il valore della condizione.

Per esempio:

```
while x > 0 and y < 0 :  
    # fa qualcosa con x  
    # fa qualcosa con y  
  
    print "x: ", x  
    print "y: ", y  
    print "condizione: ", (x > 0 and y < 0)
```

Ora, eseguendo il programma, vedrete tre righe di output per ogni ripetizione del ciclo. All'ultimo passaggio, la condizione dovrebbe risultare `False`. Se il ciclo continua, vedrete comunque i valori di `x` e `y`, e potrete capire meglio il motivo per cui non vengono aggiornati correttamente.

Ricorsione infinita

Il più delle volte, una ricorsione infinita provocherà un errore di `Maximum recursion depth exceeded`.

Se sospettate che una funzione o un metodo stia provocando una ricorsione infinita, controllate innanzitutto che esista un caso base, vale a dire un qualche tipo di condizione che provoca il ritorno della funzione o del metodo senza generare un'altra chiamata ricorsiva. Se no, occorre ripensare l'algoritmo e stabilire un caso base.

Se il caso base c'è ma sembra che il programma non lo raggiunga mai, aggiungete un'istruzione di stampa all'inizio della funzione/metodo in modo da visualizzare i parametri. Ora, eseguendo il programma vedrete alcune righe di output con i parametri per ogni chiamata alla funzione o al metodo. Se i parametri non tendono verso il caso base, avrete qualche spunto per capire il perché.

Flusso di esecuzione

Se non siete sicuri di come il flusso di esecuzione si muova dentro il vostro programma, aggiungete delle istruzioni di stampa all'inizio di ogni funzione con un messaggio del tipo "sto eseguendo la funzione pippo", dove pippo è il nome della funzione.

Ora, eseguendo il programma, verrà stampata una traccia di ogni funzione che viene invocata.

A.2.3 Quando eseguo il programma è sollevata un'eccezione.

Se qualcosa non va durante l'esecuzione, Python stampa un messaggio che include il nome dell'eccezione, la riga del programma dove il problema si è verificato, ed un traceback.

Il traceback identifica la funzione che era in esecuzione, quella che l'aveva chiamata, quella che *a sua volta* l'aveva chiamata e così via. In altre parole, traccia la sequenza di chiamate di funzione che hanno condotto alla situazione attuale. Include anche il numero di riga del file dove è avvenuta ciascuna chiamata.

Innanzitutto bisogna esaminare il punto del programma dove è emerso l'errore e vedere se si riesce a capire cosa è successo. Questi sono alcuni dei più comuni errori in esecuzione:

NameError: State cercando di usare una variabile che non esiste nell'ambiente attuale. Ricordate che le variabili locali sono, per l'appunto, locali: non potete fare loro riferimento dall'esterno della funzione in cui sono definite.

TypeError: Ci sono alcune possibili cause:

- State cercando di usare un valore in modo improprio. Esempio: indicizzare una stringa, lista o tupla con qualcosa di diverso da un numero intero.
- C'è una mancata corrispondenza tra gli elementi in una stringa di formato e gli elementi passati per la conversione. Succede se il numero degli elementi non corrisponde o se viene tentata una conversione non valida.
- State passando a una funzione o a un metodo un numero sbagliato di argomenti. Per i metodi, guardatene la definizione e controllate che il primo parametro sia `self`. Quindi guardate l'invocazione: assicuratevi di invocare il metodo su un oggetto con il giusto tipo e di fornire correttamente gli altri argomenti.

KeyError: State tentando di accedere a un elemento di un dizionario usando una chiave che nel dizionario non esiste.

AttributeError: State tentando di accedere a un attributo o a un metodo che non esiste. Controllate se è scritto giusto! Potete usare `dir` per elencare gli attributi esistenti.

Se un `AttributeError` indica che un oggetto è di tipo `NoneType`, vuol dire che è `None`. Una causa frequente è dimenticare di ritornare un valore da una funzione: se arrivate in fondo a una funzione senza intercettare un'istruzione `return`, questa restituisce `None`. Un'altra causa frequente è usare il risultato di un metodo di una lista, come `sort`, che restituisce `None`.

IndexError: L'indice che state usando per accedere a una lista, stringa o tupla è maggiore della lunghezza della sequenza meno uno. Immediatamente prima dell'ubicazione dell'errore aggiungete un'istruzione di stampa che mostri il valore dell'indice e la lunghezza della struttura. Quest'ultima è della lunghezza esatta? E l'indice ha il valore corretto?

Il debugger Python (`pdb`) è utile per catturare le Eccezioni perché vi permette di esaminare lo stato del programma immediatamente prima dell'errore. Potete leggere approfondimenti su `pdb` sul sito <http://docs.python.org/2/library/pdb.html>.

A.2.4 Ho aggiunto talmente tante istruzioni di stampa che sono sommerso di output.

Una controindicazione delle istruzioni di stampa nel debugging è che si rischia di essere inondati di messaggi. Ci sono due modi di procedere: o semplificate l'output o semplificate il programma.

Per semplificare l'output, potete rimuovere o commentare le istruzioni `print` superflue, o accorparle, o dare all'output un formato più leggibile.

Per semplificare il programma, ci sono diverse opzioni. Primo, riducete il problema che il programma sta affrontando. Per esempio, se state cercando una lista, cercatene una *piccola*. Se il programma riceve input dall'utente, dategli il dato più semplice che causa il problema.

Secondo, ripulite il programma. Togliete il codice inutile e riorganizzate il programma in modo da renderlo il più facile possibile da leggere. Per esempio, se sospettate che l'errore sia in una parte profondamente nidificata del programma, cercate di riscrivere quella parte con una struttura più semplice. Se sospettate di una corposa funzione, provate a suddividerla in funzioni più piccole e a testarle separatamente.

Spesso, il procedimento di ricercare il caso di prova più circoscritto porta a trovare l'errore. Se riscontrate che il programma funziona in un caso ma non in un altro, questo vi dà una chiave di lettura di quello che sta succedendo.

Allo stesso modo, riscrivere un pezzo di codice vi può aiutare a trovare errori insidiosi. Una modifica che pensavate influente sul programma, e che invece influisce, può farvi trovare il bandolo della matassa.

A.3 Errori di semantica

Gli errori di semantica sono i più difficili da affrontare, perché l'interprete non fornisce informazioni su ciò che non va. Sapete solo quello che il programma dovrebbe fare.

Innanzitutto occorre stabilire una connessione logica tra il testo del programma e il comportamento che vedete. Vi serve un'ipotesi di cosa sta facendo in realtà il programma. Quello che rende difficili le cose è che i computer eseguono le operazioni in tempi rapidissimi.

Vi capiterà di desiderare di poter rallentare il programma ad una velocità umana, e in effetti con alcuni strumenti di debug potete farlo. Ma il tempo che ci vuole per inserire alcune istruzioni di stampa ben calibrate è spesso più breve di quello necessario a impostare il debugger, inserire e togliere i punti di interruzione, ed eseguire i passi per portare il programma dove l'errore si verifica.

A.3.1 Il mio programma non funziona.

Dovreste porvi queste domande:

- C'è qualcosa che il programma dovrebbe fare ma che non sembra accadere? Trovate la parte del codice che esegue quella funzione e assicuratevi che sia effettivamente eseguita quando ritenete che dovrebbe esserlo.

- Sta succedendo qualcosa che non dovrebbe succedere? Trovate il codice che genera quella funzione e controllate se viene eseguita quando invece non dovrebbe esserlo.
- Una porzione di codice sta producendo effetti inattesi? Assicuratevi di capire il codice in questione, specie se coinvolge invocazioni a funzioni o metodi in altri moduli Python. Leggete la documentazione delle funzioni che invocate. Provatele scrivendo semplici test e controllando i risultati.

Per programmare, dovete avere un modello mentale di come funziona il programma. Se scrivete un programma che non fa quello che volete, molto spesso il problema non è nel programma ma nel vostro modello mentale.

Il modo migliore per correggere il vostro modello mentale è spezzare il programma nei suoi componenti (di solito funzioni e metodi) e provare indipendentemente ogni singolo componente. Quando avrete trovato la discrepanza tra il vostro modello e la realtà, potrete risolvere il problema.

Naturalmente, dovrete costruire e provare i componenti mentre state sviluppando il programma. Così, se vi imbattete in un problema, dovrebbe esserci solo una piccola quantità di codice di cui occorre verificare l'esattezza.

A.3.2 Ho una grande e complicata espressione che non fa quello che voglio.

Scrivere espressioni complesse va bene fino a quando restano leggibili, ma possono essere difficili da correggere. Un buon consiglio è di spezzare un'espressione complessa in una serie di assegnazioni a variabili temporanee.

Per esempio:

```
self.mani[i].aggiungiCarta(self.mani[self.trovaVicino(i)].togliCarta())
```

Può essere riscritta così:

```
vicino = self.trovaVicino(i)
cartaScelta = self.mani[vicino].togliCarta()
self.mani[i].aggiungiCarta(cartaScelta)
```

La versione esplicita è più leggibile perché i nomi delle variabili aggiungono informazione, ed è più facile da correggere perché potete controllare i tipi delle variabili intermedie e visualizzare il loro valore.

Un altro problema che si verifica con le grandi espressioni è che l'ordine di valutazione delle operazioni può essere diverso da quello che pensate. Per esempio, nel tradurre in Python l'espressione $\frac{x}{2\pi}$, potreste scrivere:

```
y = x / 2 * math.pi
```

È sbagliato, perché moltiplicazione e divisione hanno la stessa priorità e vengono calcolate da sinistra verso destra; quindi quell'espressione calcola $x\pi/2$.

Un buon modo di fare il debug delle espressioni è aggiungere delle parentesi per rendere esplicito l'ordine delle operazioni.

```
y = x / (2 * math.pi)
```

Usate le parentesi ogni volta che non siete certi dell'ordine delle operazioni. Non solo il programma sarà corretto (nel senso che farà quello che volete), sarà anche più leggibile da altre persone che non hanno imparato a memoria le regole di precedenza.

A.3.3 Ho una funzione o metodo che non restituisce quello che voglio.

Se avete un'istruzione `return` associata ad un'espressione complessa, non avete modo di stampare il valore prima del ritorno. Di nuovo, usate una variabile temporanea. Per esempio, anziché:

```
return self.mani[i].togliUguali()
```

potete scrivere:

```
conta = self.mani[i].togliUguali()
return conta
```

Ora potete stampare il valore di `conta` prima che sia restituito.

A.3.4 Sono proprio bloccato e mi serve aiuto.

Per prima cosa, staccatevi dal computer per qualche minuto. I computer emettono onde che influenzano il cervello, causando questi sintomi:

- Frustrazione e rabbia.
- Credenze superstiziose ("il computer mi odia") e influssi magici ("il programma funziona solo quando indosso il berretto all'indietro").
- Programmazione a tentoni (il tentativo di programmare scrivendo ogni possibile programma e prendendo quello che funziona).

Se accusate qualcuno di questi sintomi, alzatevi e andate a fare una passeggiata. Quando vi siete calmati, ripensate al programma. Cosa sta facendo? Quali sono le possibili cause del suo comportamento? Quand'era l'ultima volta che avete avuto un programma funzionante, e cosa avete fatto dopo?

A volte per trovare un bug è richiesto solo del tempo. Io trovo spesso bug mentre non sono al computer e distraigo la mente. Tra i posti migliori per trovare bug: in treno, sotto la doccia, a letto appena prima di addormentarsi.

A.3.5 No, ho davvero bisogno di aiuto.

Capita. Anche i migliori programmatori a volte si bloccano. Magari avete lavorato talmente a lungo sul programma da non riuscire a vedere un errore. Un paio di occhi freschi sono quello che ci vuole.

Prima di rivolgervi a qualcun altro, dovete fare dei preparativi. Il vostro programma dovrebbe essere il più semplice possibile, e dovete fare in modo di lavorare sul più circoscritto input che causa l'errore. Dovete posizionare delle istruzioni di stampa nei posti adatti (e l'output che producono deve essere comprensibile). Il problema va compreso abbastanza bene da poterlo descrivere in poche parole.

Quando portate qualcuno ad aiutarvi, assicuratevi di dargli tutte le informazioni che servono:

- Se c'è un messaggio di errore, di cosa si tratta e quale parte del programma indica?

- Qual è l'ultima cosa che avete fatto prima della comparsa dell'errore? Quali erano le ultime righe di codice che avevate scritto, oppure il nuovo caso di prova che non è riuscito?
- Cosa avete provato a fare finora, e cosa avete appreso dai tentativi?

Quando trovate l'errore, prendetevi un attimo di tempo per pensare cosa avreste potuto fare per trovarlo più velocemente: la prossima volta che incontrerete qualcosa di simile, vi sarà più facile scoprire l'errore.

Ricordate che lo scopo non è solo far funzionare il programma, ma imparare a farlo funzionare.

Appendice B

Analisi degli Algoritmi

Questa Appendice è un estratto adattato da *Think Complexity*, di Allen B. Downey, pure pubblicato da O'Reilly Media (2011). Quando avete finito questo libro, vi invito a prenderlo in considerazione.

L'**analisi degli algoritmi** è una branca dell'informatica che studia le prestazioni degli algoritmi, in particolare il tempo di esecuzione e i requisiti di memoria. Vedere anche http://en.wikipedia.org/wiki/Analysis_of_algorithms.

L'obiettivo pratico dell'analisi degli algoritmi è predire le prestazioni di algoritmi diversi in modo da orientare le scelte di progettazione.

Durante la campagna elettorale per le Presidenziali degli Stati Uniti del 2008, al candidato Barack Obama fu chiesto di fare un'analisi estemporanea in occasione della sua visita a Google. Il direttore esecutivo Eric Schmidt gli chiese scherzosamente "il modo più efficiente di ordinare un milione di interi a 32-bit". Obama era stato presumibilmente messo sull'avviso, poiché replicò subito: "Credo che un ordinamento a bolle sarebbe il modo sbagliato di procedere". Vedere http://www.youtube.com/watch?v=k4RRi_ntQc8.

È vero: l'ordinamento a bolle, o "bubble sort", è concettualmente semplice ma è lento per grandi insiemi di dati. La risposta che Schmidt probabilmente si aspettava era "radix sort" (http://it.wikipedia.org/wiki/Radix_sort)¹.

Scopo dell'analisi degli algoritmi è fare dei confronti significativi tra algoritmi, ma occorre tener conto di alcuni problemi:

- L'efficienza relativa degli algoritmi può dipendere dalle caratteristiche dell'hardware, per cui un algoritmo può essere più veloce sulla Macchina A, un altro sulla Macchina B. La soluzione in genere è specificare un **modello di macchina** e quindi analizzare il numero di passi, o operazioni, che un algoritmo richiede su quel dato modello.

¹Ma se vi capita una domanda come questa in un'intervista, ritengo che una risposta migliore sarebbe: "Il modo più rapido di ordinare un milione di interi è usare una qualsiasi funzione di ordinamento di cui dispone il linguaggio di programmazione che uso. Il suo rendimento sarà abbastanza buono per la maggior parte delle applicazioni, ma se proprio capitasse che la mia fosse troppo lenta, userei un profiler per controllare dove viene impiegato il tempo. Se risultasse che un algoritmo di ordinamento più rapido avrebbe un impatto significativo sulle prestazioni, cercherei una buona implementazione del radix sort".

- L'efficienza relativa può dipendere da alcuni dettagli dell'insieme di dati. Per esempio, alcuni algoritmi di ordinamento sono più veloci se i dati sono già parzialmente ordinati; altri in casi simili sono più lenti. Un modo di affrontare il problema è di analizzare lo scenario del **caso peggiore**. Talvolta è utile anche analizzare le prestazioni del caso medio, ma questo comporta più difficoltà e può non essere facile stabilire quale insieme di dati mediare.
- L'efficienza relativa dipende anche dalle dimensioni del problema. Un algoritmo di ordinamento che è veloce per liste corte può diventare lento su liste lunghe. La soluzione più comune è di esprimere il tempo di esecuzione (o il numero di operazioni) in funzione delle dimensioni, e confrontare le funzioni **asintoticamente**, al crescere delle dimensioni del problema.

Il lato buono di questo tipo di confronto è che conduce a una semplice classificazione degli algoritmi. Ad esempio, se sappiamo che il tempo di esecuzione dell'algoritmo A tende ad essere proporzionale alle dimensioni dell'input, n , e l'algoritmo B tende ad essere proporzionale a n^2 , allora possiamo attenderci che A sia più veloce di B al crescere di n .

Questo tipo di analisi ha alcune avvertenze, ma ci torneremo più avanti.

B.1 Ordine di complessità

Supponiamo che abbiate analizzato due algoritmi esprimendo i loro tempi di esecuzione in funzione delle dimensioni dell'input: l'Algoritmo A impiega $100n + 1$ operazioni per risolvere un problema di dimensione n ; l'Algoritmo B impiega $n^2 + n + 1$ operazioni.

La tabella seguente mostra il tempo di esecuzione di questi algoritmi per diverse dimensioni del problema:

Dimensione dell'input	Tempo Algoritmo A	Tempo Algoritmo B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

Per $n = 10$, l'Algoritmo A si comporta piuttosto male: impiega quasi 10 volte il tempo dell'Algoritmo B. Ma per $n = 100$ sono circa equivalenti, e per grandi valori A è molto migliore.

Il motivo fondamentale è che, per grandi valori di n , ogni funzione che contiene un termine n^2 crescerà più rapidamente di una che ha come termine dominante n . Il **termine dominante** è quello con il più alto esponente.

Per l'algoritmo A, il termine dominante ha un grande coefficiente, 100, ed è per questo che B è migliore di A per piccoli valori di n . Ma indipendentemente dal coefficiente, esisterà un valore di n a partire dal quale $an^2 > bn$.

Stesso discorso vale per i termini secondari. Anche se il tempo di esecuzione dell'Algoritmo A fosse $n + 1000000$, sarebbe sempre migliore di B per valori sufficientemente grandi di n .

In genere, possiamo aspettarci che un algoritmo con piccolo termine dominante sia migliore per problemi di dimensione maggiore, ma per quelli di minori dimensioni può esistere un **punto di intersezione** dove un altro algoritmo diventa migliore. Questo punto dipende dai dettagli dell'algoritmo, dai dati di input e dall'hardware, quindi di solito è trascurato per gli scopi dell'analisi degli algoritmi. Ma non significa che dobbiate scordarvene.

Se due algoritmi hanno il termine dominante dello stesso ordine, è difficile stabilire quale sia migliore; ancora, la risposta dipende dai dettagli. Per l'analisi degli algoritmi, le funzioni dello stesso ordine sono considerate equivalenti, anche se hanno coefficienti diversi.

Un **ordine di complessità** è dato da un insieme di funzioni il cui comportamento di crescita asintotica è considerato equivalente. Per esempio, $2n$, $100n$ e $n + 1$ appartengono allo stesso ordine di complessità, che si scrive $O(n)$ nella **notazione O-grande** e viene chiamato **lineare** perché tutte le funzioni dell'insieme crescono in maniera lineare al crescere di n .

Tutte le funzioni con termine dominante n^2 appartengono a $O(n^2)$; sono dette **quadratiche**, che è una parolona per denotare le funzioni di termine n^2 .

La tabella seguente mostra alcuni degli ordini di complessità più comuni nell'analisi degli algoritmi, in ordine crescente di inefficienza.

Ordine di complessità	Nome
$O(1)$	costante
$O(\log_b n)$	logaritmico (per qualunque b)
$O(n)$	lineare
$O(n \log_b n)$	"enne log enne"
$O(n^2)$	quadratico
$O(n^3)$	cubico
$O(c^n)$	esponenziale (per qualunque c)

Per i termini logaritmici, la base del logaritmo non ha importanza; cambiare base equivale a moltiplicare per una costante, il che non modifica l'ordine di complessità. Allo stesso modo, tutte le funzioni esponenziali appartengono allo stesso ordine indipendentemente dall'esponente. Dato che le funzioni esponenziali crescono molto velocemente, gli algoritmi esponenziali sono utili solo per problemi di piccole dimensioni.

Esercizio B.1. Leggete la pagina di Wikipedia sulla notazione O-grande: <http://it.wikipedia.org/wiki/O-grande> e rispondete alle seguenti domande:

1. Qual è l'ordine di complessità di $n^3 + n^2$? E di $1000000n^3 + n^2$? E di $n^3 + 1000000n^2$?
2. Qual è l'ordine di complessità $(n^2 + n) \cdot (n + 1)$? Prima di iniziare a moltiplicare, ricordate che vi interessa solo il termine dominante.
3. Se f appartiene a $O(g)$, per una non specificata funzione g , cosa possiamo dire di $af + b$?
4. Se f_1 e f_2 appartengono a $O(g)$, cosa possiamo dire di $f_1 + f_2$?
5. Se f_1 appartiene a $O(g)$ e f_2 appartiene a $O(h)$, cosa possiamo dire di $f_1 + f_2$?
6. Se f_1 appartiene a $O(g)$ e f_2 appartiene a $O(h)$, cosa possiamo dire di $f_1 \cdot f_2$?

I programmatori attenti alle prestazioni sono spesso critici su questo tipo di analisi. Ne hanno un motivo: a volte i coefficienti e i termini secondari fanno davvero differenza. E a volte i dettagli dell'hardware, del linguaggio di programmazione, e delle caratteristiche dell'input fanno una grande differenza. E per i piccoli problemi, l'analisi asintotica è irrilevante.

Ma tenute presenti queste avvertenze, l'analisi degli algoritmi è uno strumento utile. Almeno per i grandi problemi, l'algoritmo "migliore" è effettivamente migliore, e a volte *molto* migliore. La differenza tra due algoritmi dello stesso ordine di solito è un fattore costante, ma la differenza tra un buon algoritmo e uno cattivo è illimitata!

B.2 Analisi delle operazioni fondamentali di Python

Molte operazioni aritmetiche richiedono un tempo costante: di solito la moltiplicazione impiega più tempo di addizione e sottrazione, e la divisione impiega ancora di più, ma i tempi di esecuzione sono indipendenti dalla grandezza degli operandi. Fanno eccezione gli interi molto grandi: in tal caso il tempo di elaborazione cresce al crescere del numero delle cifre.

Le operazioni di indicizzazione—lettura e scrittura di elementi di una sequenza o dizionario—sono anch'esse a tempo costante, indipendentemente dalle dimensioni della struttura di dati.

Un ciclo `for` che attraversa una sequenza o un dizionario è di solito lineare, a patto che tutte le operazioni nel corpo del ciclo siano a tempo costante. Per esempio, la sommatoria degli elementi di una lista è lineare:

```
totale = 0
for x in t:
    totale += x
```

La funzione predefinita `sum` è pure lineare, visto che fa la stessa cosa, ma tende ad essere più rapida perché è un'implementazione più efficiente: nel linguaggio dell'analisi degli algoritmi, ha un coefficiente del termine dominante più piccolo.

Se usate lo stesso ciclo per "sommare" una lista di stringhe, il tempo di esecuzione è quadratico perché il concatenamento di stringhe è lineare.

Il metodo delle stringhe `join` di solito è più veloce, perché è lineare rispetto alla lunghezza totale delle stringhe.

Come regola di massima, se il corpo di un ciclo appartiene a $O(n^a)$ allora il ciclo nel suo complesso appartiene a $O(n^{a+1})$. Fa eccezione il caso in cui il ciclo esce dopo un numero di iterazioni costante. Se un ciclo viene eseguito per k volte indipendentemente da n , allora il ciclo appartiene a $O(n^a)$, anche per grandi valori di k .

Moltiplicare per k non cambia l'ordine di complessità, ma nemmeno dividere. Pertanto se il corpo del ciclo appartiene $O(n^a)$ e viene eseguito n/k volte, il ciclo appartiene a $O(n^{a+1})$, anche per grandi valori di k .

La maggioranza delle operazioni su stringhe e tuple sono lineari, eccetto l'indicizzazione e `len`, che sono a tempo costante. Le funzioni predefinite `min` e `max` sono lineari. Il tempo di esecuzione dello slicing è proporzionale alla lunghezza del risultato, ma indipendente dalle dimensioni del dato di partenza.

Tutti i metodi delle stringhe sono lineari, ma se le lunghezze delle stringhe sono limitate da una costante—ad esempio operazioni su singoli caratteri—sono considerati a tempo costante.

La maggior parte dei metodi delle liste sono lineari, con alcune eccezioni:

- L'aggiunta di un elemento alla fine di una lista è mediamente a tempo costante; quando si supera lo spazio disponibile, occasionalmente la lista viene copiata in uno spazio più ampio, ma il tempo totale per n operazioni è $O(n)$, quindi possiamo considerare che il tempo “spalmato” su una operazione è $O(1)$.
- La rimozione di un elemento dalla fine della lista è a tempo costante.
- L'ordinamento appartiene a $O(n \log n)$.

La maggior parte delle operazioni e dei metodi dei dizionari sono lineari, con alcune eccezioni:

- Il tempo di esecuzione di copy è proporzionale al numero di elementi, ma non alle loro dimensioni (copia i riferimenti e non gli elementi in sé).
- Il tempo di esecuzione di update è proporzionale alle dimensioni del dizionario passato come parametro, non del dizionario che viene aggiornato.
- keys, values e items sono lineari perché restituiscono nuove liste; iterkeys, intervalues e iteritems sono costanti perché restituiscono iteratori. Ma se fate un ciclo su un iteratore, il ciclo sarà lineare. Usare le funzioni “iter” risparmia qualche spesa, ma non cambia l'ordine di incremento, a meno che il numero di elementi a cui accedete non abbia un limite.

Le prestazioni dei dizionari sono uno dei piccoli miracoli dell'informatica. Vedremo come funzionano nel Paragrafo B.4.

Esercizio B.2. Leggete la pagina di Wikipedia sugli algoritmi di ordinamento: http://it.wikipedia.org/wiki/Algoritmo_di_ordinamento e rispondete alle seguenti domande:

1. Che cos'è un ordinamento per confronto (“comparison sort”)? Qual è l'ordine di complessità minimo, nel peggiore dei casi, per un ordinamento per confronto? Qual è l'ordine di complessità minimo, nel peggiore dei casi, per qualsiasi algoritmo di ordinamento?
2. Qual è l'ordine di complessità dell'ordinamento a bolle (o bubblesort), e perché Barack Obama pensa che sia “il modo sbagliato di procedere”?
3. Qual è l'ordine di complessità del radix sort? Quali precondizioni devono essere soddisfatte per poterlo usare?
4. Che cos'è un ordinamento stabile (“stable sort”) e perché è interessante in pratica?
5. Qual è il peggior algoritmo di ordinamento (tra quelli che hanno un nome)?
6. Quale algoritmo di ordinamento usa la libreria C? Quale usa Python? Questi algoritmi sono stabili? Eventualmente fate alcune ricerche sul web per trovare le risposte.
7. Molti degli ordinamenti che non operano per confronto sono lineari, allora perché Python usa un ordinamento per confronto di tipo $O(n \log n)$?

B.3 Analisi degli algoritmi di ricerca

Una **ricerca** è un algoritmo che, data una raccolta e un elemento, determina se l'elemento appartiene alla raccolta, restituendo di solito il suo indice.

Il più semplice algoritmo di ricerca è una “ricerca lineare”, che attraversa gli elementi della raccolta nel loro ordine, fermandosi se trova quello che cerca. Nel caso peggiore, dovrà attraversare tutta la raccolta, quindi il tempo di esecuzione è lineare.

L'operatore `in` delle sequenze usa una ricerca lineare, come pure i metodi delle stringhe `find` e `count`.

Se gli elementi della sequenza sono ordinati, potete usare una **ricerca binaria**, che appartiene a $O(\log n)$. È simile all'algoritmo che usate per cercare una voce in un vocabolario. Invece di partire dall'inizio e controllare ogni voce nell'ordine, cominciate con un elemento nel mezzo e controllate se quello che cercate viene prima o dopo. Se viene prima, cercate nella prima metà della sequenza, altrimenti nella seconda metà. In ogni caso, dimezzate il numero di elementi rimanenti.

Se una sequenza ha 1.000.000 di elementi, ci vorranno al massimo una ventina di passaggi per trovare la parola o concludere che non esiste. Quindi è circa 50.000 volte più veloce di una ricerca lineare.

Esercizio B.3. *Scrivete una funzione di nome `bisezione` che prenda una lista ordinata e un valore da cercare, e restituisca l'indice di quel valore nella lista, se esiste, oppure `None` se non esiste.*

Oppure leggete la documentazione del metodo `bisect` e usate quello!

La ricerca binaria può essere molto più veloce di quella lineare, ma richiede che la sequenza sia ordinata, il che può comportare del lavoro supplementare.

Esiste un'altra struttura di dati, chiamata **tabella hash**, che è ancora più veloce—è in grado di effettuare una ricerca a tempo costante—e non richiede che gli elementi siano ordinati. I dizionari di Python sono implementati usando tabelle hash, e questo è il motivo per cui la maggior parte delle operazioni sui dizionari, incluso l'operatore `in`, sono a tempo costante.

B.4 Tabelle hash

Per spiegare come funzionano le tabelle hash e perché le loro prestazioni sono così buone, inizierò con un'implementazione semplice di una mappatura e la migliorerò gradualmente fino a farla diventare una tabella hash.

Per illustrare questa implementazione userò Python, ma in pratica non scriverete mai codice del genere in Python: userete semplicemente un dizionario! Pertanto, per il resto di questo capitolo immaginate che i dizionari non esistano, e di voler implementare una struttura di dati che fa corrispondere delle chiavi a dei valori. Le operazioni che bisogna implementare sono:

`add(k, v)`: Aggiunge un nuovo elemento che fa corrispondere la chiave `k` al valore `v`. Con un dizionario Python, `d`, questa operazione si scrive `d[k] = v`.

`get(target)`: Cerca e restituisce il valore corrispondente alla chiave `target`. Con un dizionario Python, `d`, questa operazione si scrive `d[target]` oppure `d.get(target)`.

Per ora, supponiamo che ogni chiave compaia solo una volta. L'implementazione più semplice di questa interfaccia usa una lista di tuple, dove ogni tupla è una coppia chiave-valore.

```
class MappaLineare(object):

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for chiave, valore in self.items:
            if chiave == k:
                return valore
        raise KeyError
```

add accoda una tupla chiave-valore alla lista di elementi, operazione che è a tempo costante.

get usa un ciclo for per ricercare nella lista: se trova la chiave target restituisce il corrispondente valore, altrimenti solleva un `KeyError`. Quindi get è lineare.

Un'alternativa è mantenere la lista ordinata per chiavi. Allora, get potrebbe usare una ricerca binaria, che appartiene a $O(\log n)$. Ma l'inserimento di un nuovo elemento in mezzo a una lista è lineare, quindi questa potrebbe non essere l'opzione migliore. Esistono altre strutture di dati (vedere <http://it.wikipedia.org/wiki/RB-Albero>) in grado di implementare add e get in tempo logaritmico, ma non va ancora così bene come il tempo costante, quindi andiamo avanti.

Un modo di migliorare MappaLineare è di spezzare la lista di coppie chiave-valore in liste più piccole. Ecco un'implementazione chiamata MappaMigliore, che è una lista di 100 MappedLineari. Vedremo in un istante che l'ordine di complessità di get è sempre lineare, ma MappaMigliore è un passo in direzione delle tabelle hash:

```
class MappaMigliore(object):

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(MappaLineare())

    def trova_mappa(self, k):
        indice = hash(k) % len(self.maps)
        return self.maps[indice]

    def add(self, k, v):
        m = self.trova_mappa(k)
        m.add(k, v)

    def get(self, k):
        m = self.trova_mappa(k)
        return m.get(k)
```

`__init__` crea una lista di n `MappaLineari`.

`trova_mappa` è usata da `add` e `get` per capire in quale mappatura inserire il nuovo elemento o in quale mappatura ricercare.

`trova_mappa` usa la funzione predefinita `hash`, che accetta pressoché qualunque oggetto Python e restituisce un intero. Un limite di questa implementazione è che funziona solo con chiavi a cui è applicabile un hash, e i tipi mutabili come liste e dizionari non lo sono.

Gli oggetti hash-abili che vengono considerati uguali restituiscono lo stesso valore hash, ma l'inverso non è necessariamente vero: due oggetti diversi possono restituire lo stesso valore hash.

`trova_mappa` usa l'operatore modulo per inglobare i valori hash nell'intervallo da 0 a `len(self.maps)`, in modo che il risultato sia un indice valido per la lista. Naturalmente, ciò significa che molti valori hash diversi saranno inglobati nello stesso indice. Ma se la funzione hash distribuisce le cose abbastanza uniformemente (che è quello per cui le funzioni hash sono progettate), possiamo attenderci $n/100$ elementi per `MappaLineare`.

Siccome il tempo di esecuzione di `MappaLineare.get` è proporzionale al numero di elementi, possiamo attenderci che `MappaMigliore` sia circa 100 volte più veloce di `MappaLineare`. L'ordine di complessità è sempre lineare, ma il coefficiente del termine dominante è più piccolo. Risultato discreto, ma non ancora come una tabella hash.

Ed ecco (finalmente) il punto cruciale che rende veloci le tabelle hash: se riuscite a mantenere limitata la lunghezza massima delle `MappaLineari`, `MappaLineare.get` diventa a tempo costante. Quello che bisogna fare è tenere conto del numero di elementi, e quando questo numero per `MappaLineare` eccede una soglia, ridimensionare la tabella hash aggiungendo altre `MappaLineari`.

Ecco un'implementazione di una tabella hash:

```
class MappaHash(object):

    def __init__(self):
        self.maps = MappaMigliore(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def ridimensiona(self):
        new_maps = MappaMigliore(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
```

```
new_maps.add(k, v)
```

```
self.maps = new_maps
```

Ogni MappaHash contiene una MappaMigliore; `__init__` comincia con sole 2 MappeLineari e inizializza `num`, che tiene il conto del numero di elementi.

`get` rinvia semplicemente a `MappaMigliore`. Il lavoro vero si svolge in `add`, che controlla il numero di elementi e le dimensioni di `MappaMigliore`: se sono uguali, il numero medio di elementi per `MappaLineare` è 1, quindi chiama `ridimensiona`.

`ridimensiona` crea una nuova `MappaMigliore`, di capienza doppia della precedente, e ricalcola l'hash degli elementi dalla vecchia mappatura alla nuova.

Il ricalcolo è necessario perché cambiare il numero di MappeLineari cambia il denominatore dell'operatore modulo in `trova_mappa`. Ciò significa che alcuni oggetti che erano inglobati nella stessa `MappaLineare` saranno separati (che era quello che volevamo, no?).

Il ricalcolo dell'hash è lineare, quindi `ridimensiona` è lineare, il che può sembrare negativo dato che mi ripromettevo che `add` diventasse a tempo costante. Ma ricordate che non dobbiamo ridimensionare ogni volta, quindi `add` è di norma costante e solo qualche volta lineare. Il lavoro complessivo per eseguire `add` n volte è proporzionale a n , quindi il tempo medio di ogni `add` è costante!

Per capire come funziona, supponiamo di iniziare con una tabella hash vuota e aggiungere una sequenza di elementi. Iniziamo con 2 MappeLineari, quindi le prime 2 aggiunte saranno veloci (nessun ridimensionamento richiesto). Diciamo che richiedono una unità lavoro ciascuna. L'aggiunta successiva richiede il ridimensionamento, e dobbiamo ricalcolare l'hash dei primi due elementi (diciamo 2 unità lavoro in più), quindi aggiungere il terzo elemento (1 altra unità). Aggiungere l'elemento successivo costa 1 unità, e in totale fanno 6 unità lavoro per 4 elementi.

Il successivo `add` costa 5 unità, ma i tre successivi solo 1 unità ciascuno, in totale 14 unità per 8 aggiunte.

L'aggiunta successiva costa 9 unità, ma poi possiamo aggiungerne altre 7 prima del ridimensionamento successivo, per un totale di 30 unità lavoro per le prime 16 aggiunte.

Dopo 32 aggiunte, il costo totale è 62 unità, e spero stiate cominciando ad avere chiaro lo schema. Dopo n aggiunte, con n potenza di 2, il costo totale è $2n - 2$ unità, per cui il lavoro medio per aggiunta è poco meno di 2 unità. Con n potenza di 2 si ha il caso migliore; per altri valori di n il lavoro medio è leggermente più alto, ma non in modo importante. La cosa importante è che sia $O(1)$.

La Figura B.1 illustra graficamente il funzionamento. Ogni quadrato è una unità di lavoro. Le colonne mostrano il lavoro totale per ogni aggiunta nell'ordine da sinistra verso destra: le prime due aggiunte costano 1 unità, la terza 3, ecc.

Il lavoro supplementare di ricalcolo appare come una sequenza di torri sempre più alte e con spazi sempre più ampi tra due torri successive. Ora, se abbattete le torri, mediando il costo del ridimensionamento su tutte le aggiunte, potete vedere graficamente che il costo del lavoro totale dopo n aggiunte è $2n - 2$.

Una caratteristica importante di questo algoritmo è che quando ridimensioniamo la tabella hash, cresce geometricamente, cioè moltiplichiamo la dimensione per una costante. Se

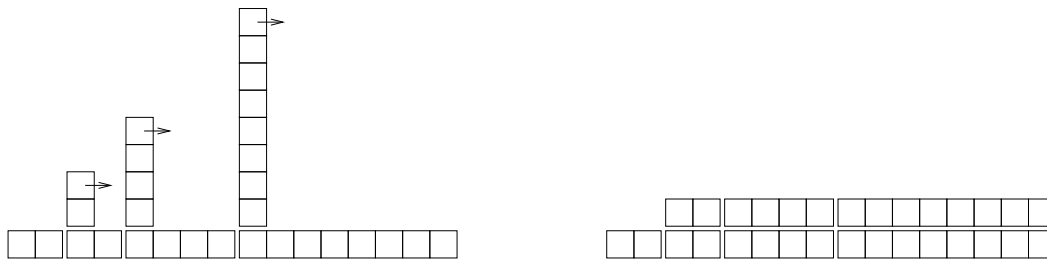


Figura B.1: Costo delle aggiunte a una Tabella Hash.

incrementate le dimensioni aritmeticamente, aggiungendo ogni volta un numero fisso, il tempo medio per aggiunta sarebbe lineare.

Potete scaricare la mia implementazione di MappaHash da <http://thinkpython/code/Map.py>, ma ricordate che non c'è alcuna buona ragione per usarla. Piuttosto, se dovete fare una mappatura, usate un dizionario di Python.

Appendice C

Lumpy

Nel corso del libro, ho utilizzato dei diagrammi per rappresentare la situazione di un programma in esecuzione.

Nel Paragrafo 2.2, abbiamo usato un diagramma di stato per mostrare nomi e valori delle variabili. Nel Paragrafo 3.10 ho presentato il diagramma di stack, che mostra un frame per ciascuna chiamata di funzione. Ogni frame contiene i parametri e le variabili locali della funzione o del metodo. Diagrammi di stack per le funzioni ricorsive compaiono nei Paragrafi 5.9 e 6.5.

Il Paragrafo 10.2 illustra come appare una lista in un diagramma di stato, nel Paragrafo 11.4 come appare un dizionario, e nel Paragrafo 12.6 vi sono due modi di raffigurare le tuple.

Il Paragrafo 15.2 presenta i diagrammi di oggetto, che mostrano lo stato degli attributi di un oggetto, e dei loro attributi, e così via. Il Paragrafo 15.3 contiene diagrammi di oggetto dei Rettangoli e dei loro Punti contenuti. Nel Paragrafo 16.1 viene illustrato lo stato di un oggetto Tempo. Nel Paragrafo 18.2 c'è un diagramma che comprende un oggetto classe e un'istanza, ciascuna con i relativi attributi.

Infine, il Paragrafo 18.8 presenta i diagrammi di classe, che schematizzano le classi che costituiscono un programma e le relazioni tra di esse.

Questi diagrammi sono basati sull'*Unified Modeling Language* (UML), che è un linguaggio grafico standardizzato usato dai programmatori per scambiarsi informazioni sul progetto dei programmi, in particolare per quelli orientati agli oggetti.

L'UML è un linguaggio ricco con molti tipi di diagrammi che rappresentano molti tipi di relazioni tra oggetti e classi. Quella che ho mostrato in questo libro è solo una piccola parte del linguaggio, ma è quella più comunemente usata nella pratica.

Lo scopo di questa Appendice è di riassumere i diagrammi illustrati nei capitoli precedenti e di presentare Lumpy. Lumpy sta per "UML in Python", con qualche lettera scambiata, ed è parte di Swampy, che avete già installato per lavorare sulle Esercitazioni dei Capitoli 4 e 19, oppure se avete svolto l'Esercizio 15.4.

Lumpy usa il modulo di Python `inspect` per esaminare lo stato di un programma in esecuzione e generare diagrammi di oggetto (inclusi diagrammi di stack) e diagrammi di classe.

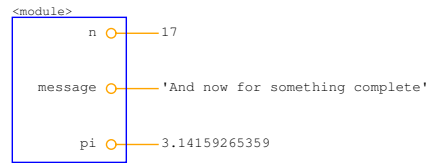


Figura C.1: Diagramma di stato generato da Lumpy.

C.1 Diagramma di stato

Ecco un esempio che utilizza Lumpy per generare un diagramma di stato.

```
from swampy.Lumpy import Lumpy
```

```
lumpy = Lumpy()
lumpy.make_reference()
```

```
messaggio = 'Ed ora qualcosa di completamente diverso'
n = 17
pi = 3.1415926535897932
```

```
lumpy.object_diagram()
```

La prima riga importa la classe Lumpy da `swampy.Lumpy`. Se non avete installato Swampy come pacchetto, assicuratevi che i file di Swampy siano nel percorso di ricerca di Python e usate invece questa istruzione `import`:

```
from Lumpy import Lumpy
```

La riga successiva crea un oggetto Lumpy e crea un punto di “riferimento”, cioè Lumpy registra gli oggetti che vengono definiti da questo punto in poi.

Definiamo poi delle nuove variabili, per poi invocare `object_diagram`, che disegna gli oggetti che sono stati definiti a partire dal punto di riferimento, in questo caso `messaggio`, `n` e `pi`.

La Figura C.1 mostra il risultato. Lo stile grafico è diverso da quello visto sinora, ad esempio ogni riferimento è rappresentato da un cerchietto vicino al nome della variabile e da una linea con il valore. E le stringhe troppo lunghe vengono troncate. Ma l’informazione contenuta nel diagramma è la stessa.

I nomi delle variabili si trovano in un riquadro etichettato `<module>`, che indica che si tratta di variabili a livello di modulo, dette anche globali.

Potete scaricare questo esempio da http://thinkpython.com/code/lumpy_demo1.py. Provate ad aggiungere qualche altra assegnazione e a vedere come si presenta il diagramma.

C.2 Diagramma di stack

Ecco un esempio che utilizza Lumpy per generare un diagramma di stack. Lo potete scaricare da http://thinkpython.com/code/lumpy_demo2.py.



Figura C.2: Diagramma di stack.

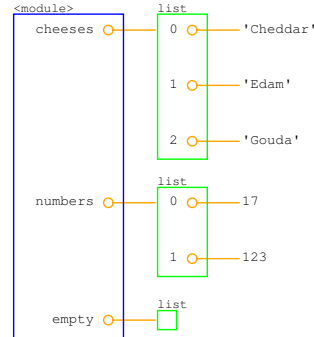


Figura C.3: Diagramma di oggetto.

```
from swampy.Lumpy import Lumpy
```

```
def contoallarovescia(n):
    if n <= 0:
        print 'Via!'
        lumpy.object_diagram()
    else:
        print n
        contoallarovescia(n-1)
```

```
lumpy = Lumpy()
lumpy.make_reference()
contoallarovescia(3)
```

La Figura C.2 mostra il risultato. Ogni frame è rappresentato da un riquadro che reca all'esterno il nome della funzione, e le relative variabili all'interno. Dato che questa funzione è ricorsiva, c'è un riquadro per ogni livello di ricorsione.

Ricordate che un diagramma di stack mostra lo stato del programma in un ben determinato punto della sua esecuzione. Per ottenere il diagramma che volete, dovrete pensare a dove invocare `object_diagram`.

In questo caso ho invocato `object_diagram` dopo l'esecuzione del caso base della ricorsione: in questo modo il diagramma mostra ciascun livello. Potete anche chiamare `object_diagram` più volte per ottenere una sequenza di istantanee dell'esecuzione del programma.

C.3 Diagrammi di oggetto

Questo esempio genera un diagramma di oggetto che illustra le liste del Paragrafo 10.1. Potete scaricarlo da http://thinkpython.com/code/lumpy_demo3.py.

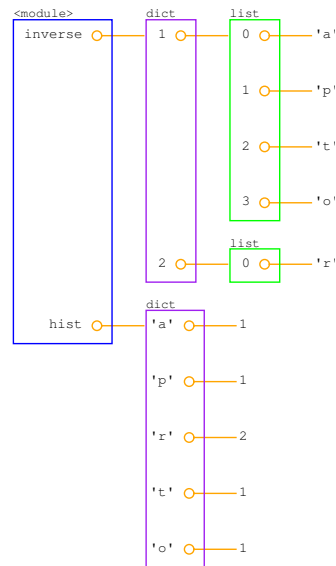


Figura C.4: Diagramma di oggetto.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

formaggi = ['Cheddar', 'Edam', 'Gouda']
numeri = [17, 123]
vuota = []

lumpy.object_diagram()
```

Il risultato è riportato in Figura C.3. Le liste sono rappresentate da un riquadro che contiene gli indici corrispondenti agli elementi. Questa rappresentazione è un po' impropria, perché gli indici in realtà non fanno parte della lista, ma ritengo che renda più leggibile il diagramma. La lista vuota è rappresentata da un riquadro vuoto.

Ecco ora un esempio che mostra i dizionari del Paragrafo 11.4, scaricabile da http://thinkpython.com/code/lumpy_demo4.py.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

isto = istogramma('parrot')
inverso = inverti_diz(isto)

lumpy.object_diagram()
```

In Figura C.4 il risultato. `isto` è un dizionario che fa corrispondere caratteri (le singole lettere di una stringa) ad interi; `inverso` fa una mappatura da interi a liste di stringhe.

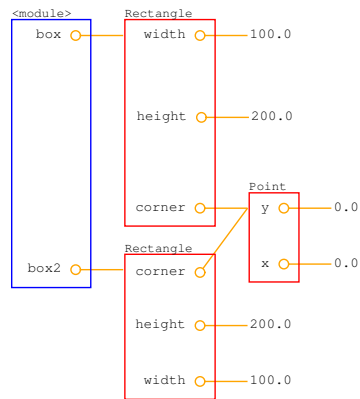


Figura C.5: Diagramma di oggetto.

Quest'altro esempio genera un diagramma di oggetto per gli oggetti Punto e Rettangolo visti nel Paragrafo 15.6. Potete scaricarlo da http://thinkpython.com/code/lumpy_demo5.py.

```
import copy
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

box = Rettangolo()
box.larghezza = 100.0
box.altezza = 200.0
box.angolo = Punto()
box.angolo.x = 0.0
box.angolo.y = 0.0

box2 = copy.copy(box)

lumpy.object_diagram()
```

La Figura C.5 mostra il risultato. `copy.copy` produce una copia shallow, quindi `box` e `box2` hanno le loro proprie larghezza e altezza, ma condividono lo stesso oggetto contenuto `Punto`. Questo tipo di condivisione di solito va bene se gli oggetti sono immutabili, ma con tipi mutabili è altamente soggetta ad errori.

C.4 Oggetti funzione e classe

Quando uso Lumpy per generare diagrammi di oggetto, di solito definisco funzioni e classi prima di fissare il punto di riferimento. In questo modo, gli oggetti funzione e classe non compaiono nel diagramma.

Ma se state passando funzioni e classi come parametri, può essere preferibile che compaiano. Questo esempio mostra come si presenta il diagramma; lo potete scaricare da http://thinkpython.com/code/lumpy_demo6.py.

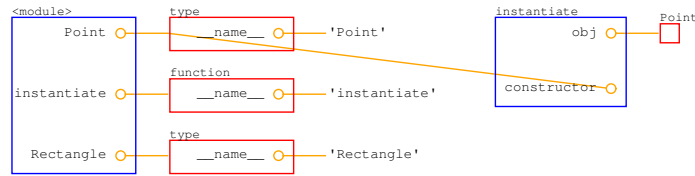


Figura C.6: Diagramma di oggetto.

```
import copy
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

class Punto(object):
    """Rappresenta un punto nel piano cartesiano."""

class Rettangolo(object):
    """Rappresenta un rettangolo."""

def istanzia(constructor):
    """Istanzia un nuovo oggetto."""
    obj = constructor()
    lumpy.object_diagram()
    return obj

punto = istanzia(Punto)
```

La Figura C.6 mostra il risultato. Dato che invochiamo `object_diagram` dentro una funzione, otteniamo un diagramma di stack con un riquadro per le variabili a livello di modulo e uno per l'invocazione di `istanzia`.

A livello di modulo, `Punto` e `Rettangolo` si riferiscono ad oggetti classe (che sono di tipo `type`); `istanzia` si riferisce a un oggetto funzione.

Questo diagramma aiuta a chiarire due punti che di solito generano malintesi: (1) la differenza tra l'oggetto classe, `Punto`, e l'istanza di `Punto`, `obj`, e (2) la differenza tra l'oggetto funzione creato quando `istanzia` viene definita e il frame creato quando viene chiamata.

C.5 Diagrammi di classe

Sebbene io faccia distinzione tra diagrammi di stato, diagrammi di stack e diagrammi di oggetto, in fondo sono più o meno la stessa cosa: mostrano la situazione di un programma in esecuzione, in un certo momento.

I diagrammi di classe sono diversi. Mostrano le classi che costituiscono un programma e le relazioni tra di esse. Sono indipendenti dal tempo, nel senso che descrivono globalmente il programma, non in un particolare momento. Per esempio, se un'istanza della Classe A contiene in generale un riferimento ad un'istanza della Classe B, diciamo che c'è una "relazione HAS-A" tra queste due classi.

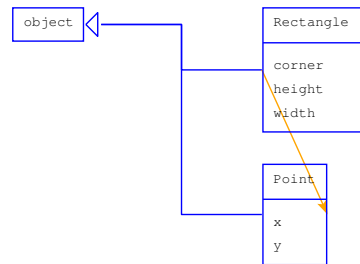


Figura C.7: Diagramma di classe.

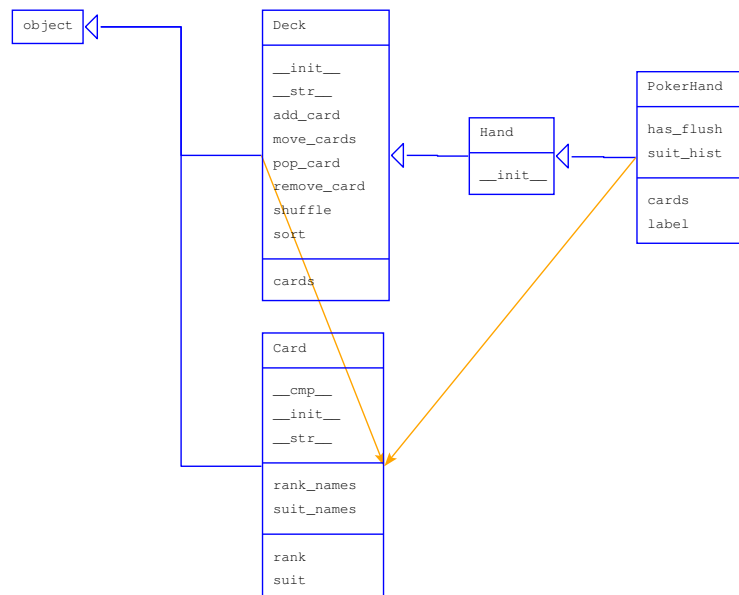


Figura C.8: Diagramma di classe.

Ecco un esempio che mostra una relazione HAS-A. Lo potete scaricare da http://thinkpython.com/code/lumpy_demo7.py.

```
from swampy.Lumpy import Lumpy
```

```
lumpy = Lumpy()
lumpy.make_reference()
```

```
box = Rettangolo()
box.larghezza = 100.0
box.altezza = 200.0
box.angolo = Punto()
box.angolo.x = 0.0
box.angolo.y = 0.0
```

```
lumpy.class_diagram()
```

La Figura C.7 mostra il risultato. Ogni classe viene rappresentata da un riquadro che contiene il nome, tutti i metodi che la classe fornisce, tutte le variabili di classe e tutte le variabili di istanza. In questo esempio, Rettangolo e Punto hanno variabili di istanza ma nessun metodo né variabili di classe.

La freccia da Rettangolo a Punto mostra che i Rettangoli contengono un Punto come oggetto contenuto. Inoltre, sia Rettangolo che Punto ereditano da object, che è rappresentato nel diagramma da una freccia con la testa a triangolo.

Ecco ora un esempio più complesso che utilizza la soluzione dell'Esercizio 18.6. Il codice è scaricabile da http://thinkpython.com/code/lumpy_demo8.py; vi servirà anche <http://thinkpython.com/code/PokerHand.py>.

```
from swampy.Lumpy import Lumpy
```

```
from ManoDiPoker import *
```

```
lumpy = Lumpy()
lumpy.make_reference()
```

```
mazzo = Mazzo()
mano = ManoDiPoker()
mazzo.move_cards(mano, 7)
```

```
lumpy.class_diagram()
```

La Figura C.8 mostra il risultato. ManoDiPoker eredita da Mano, che a sua volta eredita da Mazzo. Sia Mazzo che ManoDiPoker contengono Carte.

Il diagramma non evidenzia che anche Mano contiene Carte, perché nel programma non ci sono istanze di Mano. Questo esempio dimostra un limite di Lumpy: viene a conoscenza solo degli attributi e delle relazioni HAS-A di oggetti che sono istanziati.

Indice analitico

- abs, funzione, 58
- accesso, 97
- accumulatore, 108
 - istogramma, 137
 - lista, 101
 - somma, 101
 - stringa, 189
- Ackermann, funzione di, 67, 118
- add, metodo, 177
- addizione
 - con riporto, 74, 166, 168
- aggiornamento, 70, 73, 75
 - di coordinate, 208
 - di database, 151
 - elemento, 99
 - istogramma, 137
 - operatore di, 101
 - slicing, 100
 - variabile globale, 119
- alfabetico, 78, 91
- alfabeto, 44
- algoritmi
 - analisi pratica, 226
- algoritmo, 3, 9, 73, 140, 223
 - md5, 153
 - radice quadrata, 75
 - RSA, 120
- alias, 104, 105, 108, 159, 161, 182
 - copiare per evitare, 107
- ambiguità, 6
- anagramma, 108
- anagramma, insieme, 132, 152
- analisi asintotica, 224
- analisi degli algoritmi, 223
- analisi delle primitive, 226
- and, operatore, 46
- anydbm, modulo, 151
- apici, 7, 11, 80
- appartenenza
 - dizionario, 112
 - insieme di, 112
 - lista, 98
 - ricerca binaria, 109
- append, metodo, 100, 106, 109, 188, 189
- arco, funzione, 38
- argomento, 21, 23, 26, 31, 105
 - con nome, 39, 43, 129, 200, 209
 - esplosione, 125
 - lista, 105
 - opzionale, 82, 103, 115
 - raccolta, 125
 - tupla di lunghezza variabile, 125
- assegnazione, 18, 69, 97
 - elemento, 80, 98, 124
 - multipla, 69, 75, 119
 - potenziata, 101, 108
 - tupla, 124–126, 131
- assert, istruzione, 170
- attraversamento, 78, 81, 83, 85, 91, 101, 108, 113, 114, 126, 127, 129, 137
 - dizionario, 127, 181
 - lista, 99
- AttributeError, 163, 217
- attributo, 181
 - __dict__, 180
 - di classe, 186, 195
 - di istanza, 158, 186, 195
 - inizializzazione, 180
 - istanza, 163
- Austin, Jane, 137
- Bangladesh, bandiera, 164
- bella copia, 121
- benchmarking, 143, 144
- bifronte, parola, 109
- binding, 206, 210
- bisect, modulo, 109, 228
- blocco, 215
- bool, tipo, 45
- booleana, funzione, 165
- booleano, operatore, 82
- break, istruzione, 71

- bubble sort, 223
- bug, 3, 9
 - il peggiore, 182, 211
- Button, controllo, 200
- cadenza di gara, 10
- calcolatrice, 10, 19
- Callable, oggetto, 206
- callback, 201, 204, 206, 208, 209
- cancellare, elementi di lista, 102
- Canguro, classe, 182
- Canvas
 - controllo, 201
 - coordinate, 202, 208
 - elemento, 201
 - oggetto, 164
- Car Talk, 94, 122, 132
- carattere, 77
- Carta, classe, 186, 240
- carte da gioco, francesi, 185
- cartella, 149
- case-sensitivity, nomi di variabile, 17
- caso
 - medio, 224
 - particolare, 93, 94, 167
 - peggiore, 224
- caso base, 50, 53, 235
- caso di prova minimo, 218
- cerchio, funzione, 38
- checksum, 153
- chiamata, grafico di, 121
- chiave, 111, 121
- chiave-valore, coppia, 111, 121, 127
- choice, funzione, 136
- Ciao, Mondo, 7
- cicli e contatori, 81
- ciclo, 37, 42, 71, 126
 - attraversamento, 78
 - con dizionario, 114
 - con indici, 92, 99
 - con stringhe, 81
 - condizione, 216
 - for, 37, 78, 99
 - infinito, 71, 75, 200, 215, 216
 - nidificato, 188
 - while, 70
- classe, 157, 163
 - attributo di, 186, 195
 - Canguro, 182
 - Carta, 186
 - definizione di, 157
 - figlia, 190, 195
 - madre, 190, 195
 - Mano, 190
 - Mazzo, 188
 - Punto, 157, 177, 237
 - Rettangolo, 159, 237
 - SimpleTurtleWorld, 204
 - Tempo, 165
- close, metodo, 148, 151, 153
- __cmp__, metodo, 187
- cmp, funzione, 188
- codice morto, 58, 66, 218
- codice oggetto, 2, 8
- codice sorgente, 2, 8
- codificare, 185, 195
- coefficiente dominante, 224
- coerenza, controllo di, 121, 169
- collaboratori, vii
- colori disponibili, 164, 183
- comando Unix
 - ls, 153
- commento, 16, 18
- commutatività, 16, 179
- compara, funzione, 58
- comparison sort, 227
- compilare, 2, 8
- compleanno, 171
- compleanno, paradosso del, 109
- complessità
 - esponenziale, 225
 - lineare, 225
 - logaritmica, 225
 - ordine di, 224
 - quadratica, 225
- composizione, 23, 26, 31, 60, 188
- concatenamento, 16, 18, 27, 78, 80, 103
 - lista, 99, 106, 109
- condizione, 47, 53, 71, 216
 - di guardia, 65, 67
 - in serie, 47, 53
 - nidificata, 48, 53
- config, metodo, 201
- confronto
 - stringa, 83
 - tupla, 129, 188
- confronto di algoritmi, 223
- confronto, operatore di, 46
- congettura di Collatz, 71
- congruenza, controllo di, 121

- contachilometri, 94
- contatore, 81, 85, 112, 119
- contatori e cicli, 81
- conteggio, 154
- contenitore, 164, 202, 210
- contenuto, oggetto, 182
- controlli, packing, 204
- controllo, 200, 209
 - Button, 200
 - Canvas, 201
 - Entry, 202
 - Frame, 204
 - Label, 200
 - Menubutton, 206
 - Text, 202
- controllo dei tipi, 64
- controllo errore, 64
- conversione
 - di tipo, 21
- coordinate
 - Canvas, 202, 208
 - pixel, 208
- coordinate, sequenze di, 202
- copia, 237
 - di un oggetto, 161
 - per evitare alias, 107
 - profonda, 162, 163
 - shallow, 162, 163, 237
 - slicing, 80, 100
- copy, modulo, 161
- corpo, 23, 30, 71
- corrispondenza, 185
- costo medio, 231
- costruttore, 238
- count, metodo, 82
- Creative Commons, vi
- criptare, 185
- crittografia, 120
- cruciverba, 89
- cummings, e. e., 4
- cumulativa, somma, 102
- curva di Koch, 55
- database, 151, 156
- datetime, modulo, 171
- debug, 3, 7, 9, 17, 30, 42, 52, 65, 83, 93, 106, 120, 130, 143, 155, 163, 169, 180, 193, 208, 213
 - binario, 74
 - risposta emotiva, 8, 220
 - sperimentale, 4, 144
 - superstizione, 220
- debugger (pdb), 217
- decorate-sort-undecorate, schema, 129
- decremento, 70, 75
- deepcopy, funzione, 162
- def, parola chiave, 23
- default, valore di, 144
- definizione
 - circolare, 62
 - di classe, 157
 - di funzione, 23
 - ricorsiva, 62, 132
- del, operatore, 102
- delimitatore, 103, 108
- deterministico, 136, 144
- diagramma
 - di classe, 192, 195, 233, 238
 - di oggetto, 158, 160, 162, 163, 165, 187, 233, 235
 - di stack, 27, 31, 43, 50, 63, 106, 233, 234
 - di stato, 12, 18, 69, 85, 98, 104, 105, 116, 128, 158, 160, 162, 165, 187, 233, 234
 - grafico di chiamata, 121
- diagramma di stack, 67
- dichiarazione, 119, 122
- __dict__ attributo, 180
- dict, funzione, 111
- diff, 154
- Dijkstra, Edsger, 93
- directory, 149, 155
 - di lavoro, 149
 - esplorazione, 150
- divisibilità, 45
- divisione
 - a virgola mobile, 14
 - intera, 14, 53
- divisione intera, 18
- divmod, 125, 168
- dizionario, 111, 121, 127, 217, 236
 - attraversamento, 127, 181
 - ciclo con, 114
 - inizializzazione, 127
 - inverso, 115
 - lookup, 114
 - lookup inverso, 114
 - metodi, 227
 - sottrazione, 139
- dizionario, metodi
 - anydbm, modulo, 151

- docstring, 42, 43, 157
- documentazione, 10
- dot notation, 22, 30, 31, 158, 174, 186
- Doyle, Arthur Conan, 5
- drag-and-drop, 207
- DSU, schema, 129, 131, 138
- due punti, 23, 214
- duplicato, 109, 122, 153
- eccezione, 4, 9, 17, 213, 217
 - AttributeError, 163, 217
 - IndexError, 78, 84, 98, 217
 - IOError, 150
 - KeyError, 112, 217
 - NameError, 27, 217
 - OverflowError, 52
 - RuntimeError, 51
 - SyntaxError, 23
 - TypeError, 77, 80, 116, 124, 125, 149, 176, 217
 - UnboundLocalError, 119
 - ValueError, 52, 115, 124
- eccezione, gestione, 150
- Einstein, Albert, 40
- elemento, 85, 97, 108
 - aggiornamento, 99
 - assegnazione, 80, 124
 - cancellazione, 102
 - Canvas, 201, 210
 - dizionario, 121
- elemento, assegnazione, 98
- elif, parola chiave, 47
- Elkner, Jeff, v, vi
- ellissi, 24
- else, parola chiave, 47
- email, indirizzo, 124
- emotività, nel debug, 8, 220
- Entry, controllo, 202
- enumerate, funzione, 127
- epsilon, 73
- equivalente, 104, 108
- ereditarietà, 190, 195
- errore
 - di battitura, 143
 - di formato, 130
 - di runtime, 4, 17, 213
 - di semantica, 4, 12, 17, 84, 213, 218
 - di sintassi, 4, 17, 213
 - in esecuzione, 4, 17, 50, 52, 213, 217
 - messaggio di, 4, 7, 12, 17, 213
 - esadecimale, 158
 - esecuzione alternativa, 47
 - esecuzione condizionale, 46
 - eseguibile, 2, 8
 - esercizio, segreto, 156
 - esplosione, 125, 131
 - esponente, 224
 - espressione, 13, 14, 18
 - booleana, 45, 53
 - grande e complicata, 219
 - eval, funzione, 75
 - evento, 210
 - ciclico, 200, 210
 - gestore di, 207
 - stringa, 207
 - Evento, oggetto, 207
 - exists, funzione, 149
 - extend, metodo, 100
 - False, valore speciale, 45
 - fattoriale, funzione, 62, 64
 - Fermat, ultimo teorema di, 54
 - fibonacci, funzione, 64, 117
 - figlia, classe, 190
 - file, 147
 - di testo, 155
 - lettura e scrittura, 147
 - nome, 149
 - permesso, 150
 - filtro, schema, 101, 108
 - fine riga, carattere, 155
 - fiore, 43
 - flag, 118, 122
 - float, funzione, 21
 - float, tipo, 11
 - floating-point, 17
 - flusso di esecuzione, 25, 31, 64, 66, 70, 193, 209, 216
 - for, ciclo, 37, 78, 99, 127
 - formato, 131
 - operatore di, 148, 217
 - sequenza di, 148
 - stringa di, 148
 - formato, errore di, 130
 - frame, 27, 31, 50, 63, 117, 235
 - Frame, controllo, 204
 - Free Documentation License, GNU, v, vi
 - frequenza, 113
 - di lettere, 131
 - di parole, 135, 144

- frustrazione, 220
- funzione, 7, 23, 30, 173
 - abs, 58
 - ack, 67, 118
 - arco, 38
 - argomento di, 26
 - booleana, 61
 - cerchio, 38
 - chiamata di, 21, 31
 - choice, 136
 - cmp, 188
 - compara, 58
 - composizione di, 60
 - deepcopy, 162
 - definizione di, 23, 24, 30
 - dict, 111
 - enumerate, 127
 - eval, 75
 - exists, 149
 - fattoriale, 62
 - fibonacci, 64, 117
 - float, 21
 - frame di, 27, 31, 50, 63, 117, 235
 - gamma, 64
 - getattr, 181
 - getcwd, 149
 - hasattr, 163, 180
 - int, 21
 - isinstance, 64, 178
 - len, 31, 77, 112
 - list, 103
 - log, 22
 - max, 125, 126
 - min, 125, 126
 - modificatore, 167
 - motivi, 29
 - open, 89, 90, 147, 150, 151
 - parametro di, 26
 - poligono, 38
 - popen, 153
 - produttiva, 28, 31
 - pura, 166, 170
 - randint, 109, 136
 - random, 129, 136
 - raw_input, 51
 - reload, 154, 215
 - repr, 155
 - reversed, 130
 - ricorsiva, 49
 - shuffle, 190
 - sintassi, 174
 - sorted, 130
 - sqrt, 22, 59
 - str, 22
 - sum, 126
 - trigonometrica, 22
 - trova, 80
 - tupla come valore di ritorno, 125
 - tuple, 123
 - type, 163
 - vuota, 28, 31
 - zip, 126
- funzioni matematiche, 22
- generalizzazione, 39, 43, 91, 169
- gestire, 156
- gestore di evento, 207
- gestore di geometria, 205, 210
- get, metodo, 113
- getattr, funzione, 181
- getcwd, funzione, 149
- Giorno del Doppio, 171
- global, istruzione, 119
- globale, variabile, 121
- GNU Free Documentation License, v, vi
- graffe, parentesi, 111
- grafica vettoriale, 211
- grafico di chiamata, 117
- griglia, 32
- guardia, condizione di, 65, 67, 84
- GUI, 199, 209
- Gui, modulo, 199
- guida in linea, 10
- HAS-A, relazione, 192, 195, 238
- hasattr, funzione, 163, 180
- hash, funzione, 116, 121, 230
- hash-abile, 116, 121, 128
- Holmes, Sherlock, 5
- HTMLParser, modulo, 211
- hyperlink, 211
- identico, 108
- identità, 104
- if, istruzione, 46
- Image, modulo, 210
- immagini, visualizzatore di, 210
- immutabilità, 80, 85, 105, 117, 123, 130
- impalcatura, 60, 67, 121
- impiallacciatura, 190, 195
- implementazione, 113, 121, 142, 181

- import, istruzione, 31, 36, 154
- in, operatore, 82, 98, 112, 228
- incapsulamento, 38, 43, 60, 73, 81, 191
 - dei dati, 193
- incastro, parola, 109
- incremento, 70, 75, 167, 175
- indentazione, 23, 174, 214
- IndexError, 78, 84, 98, 217
- indice, 77, 84, 85, 97, 108, 111, 217
 - inizio da zero, 77, 98
 - negativo, 78
 - nei cicli, 92, 99
 - slicing, 79, 99
- indicizzazione, 226
- inefficienza, 225
- information hiding, 181, 182
- init, metodo, 176, 180, 186, 188, 190
- inizializzazione
 - prima di aggiornare, 70
 - variabile, 75
- Input da tastiera, 51
- insieme, 140
 - anagramma, 132, 152
- insieme di appartenenza, 112
- int, funzione, 21
- int, tipo, 11
- interfaccia, 40, 42, 43, 181, 193
- interfaccia grafica, 199
- intero, 17
 - long, 120
- interpretare, 2, 8
- intestazione, 23, 30, 214
- invariante, 169, 170, 209
- invocazione, 82, 85
- IOError, 150
- ipotenusa, 60
- is, operatore, 104, 162
- IS-A, relazione, 192, 195, 240
- isinstance, funzione, 64, 178
- istanza, 36, 42, 158, 163
 - attributo di, 158, 163, 186, 195
 - come argomento, 159
 - come valore di ritorno, 160
- istanziare, 158, 238
- istogramma, 113, 121
 - frequenza delle parole, 137
 - scelta casuale, 136, 140
- istruzione, 18
 - assegnazione, 69
 - assert, 170
 - break, 71
 - composta, 47, 53
 - condizionale, 46, 53, 61, 214
 - di assegnazione, 12
 - di stampa, 7, 9
 - for, 37, 78, 99
 - global, 119
 - if, 46
 - import, 31, 36, 154
 - pass, 47
 - print, 7, 177, 218
 - raise, 115, 170
 - return, 49, 57, 220
 - try, 150
 - while, 70
- items, metodo, 127
- iterazione, 70, 75
- join, 226
- join, metodo, 103, 189
- KeyError, 112, 217, 229
- keys, metodo, 114
- Label, controllo, 200
- lavoro, directory di, 149
- len, funzione, 31, 77, 112
- letteralità, 6
- lettere
 - doppie, 94
 - frequenza, 131
 - rotazione, 122
- linguaggio
 - completo di Turing, 61
 - di alto livello, 1, 8
 - di basso livello, 1, 8
 - di programmazione, 1
 - formale, 5, 9
 - naturale, 5, 9
 - orientato agli oggetti, 182
 - sicuro, 4
- Linux, 5
- lipogramma, 90
- list comprehension, 102
- list, funzione, 103
- lista, 97, 103, 108, 130
 - appartenenza, 98
 - attraversamento, 99, 108
 - come argomento, 105
 - concatenamento, 99, 106, 109
 - copia, 100

- di oggetti, 188
- di tuple, 126
- elemento, 97
- indice, 98, 236
- metodi, 100, 227
- nidificata, 97, 99, 108
- operazione, 99
- ripetizione, 99
- slicing, 99
- vuota, 97
- lista dei colori, 164, 183
- log, funzione, 22
- logaritmo, 145
- logico, operatore, 46
- long, intero, 120
- lookup, 121
 - dizionario, 114
- lookup inverso, dizionario, 114, 121
- ls (comando Unix), 153
- Lumpy, 233
- lunghezza variabile, tupla di argomenti, 125
- macchina da scrivere a tartaruga, 44
- macchina, modello di, 223
- madre, classe, 190
- Mano, classe, 190, 240
- manutenzione, 181
- mappa, schema, 101, 108
- MappaHash, 230
- MappaLineare, 229
- MappaMigliore, 229
- mappatura, 98, 108, 141
- Markov, analisi di, 140
- massimo comun divisore (MCD), 68
- Matplotlib, 145
- max, funzione, 125, 126
- mazzo, carte da gioco, 188
- Mazzo, classe, 188, 240
- McCloskey, Robert, 78
- MCD (massimo comun divisore), 68
- md5, 153
- md5, algoritmo, 153
- md5sum, 154
- memoizzazione, 118, 121
- Menubutton, controllo, 206
- metafora, invocazione di metodo, 175
- metatesi, 132
- metodi delle liste, 100
- metodi delle stringhe, 82
- metodo, 81, 85, 173, 182
 - __cmp__, 187
 - __str__, 177, 189
 - add, 177
 - append, 100, 106, 109, 188, 189
 - close, 148, 151, 153
 - config, 201
 - count, 82
 - extend, 100
 - get, 113
 - init, 176, 186, 188, 190
 - items, 127
 - join, 103, 189
 - keys, 114
 - mro, 193
 - pop, 102, 189
 - radd, 179
 - read, 153
 - readline, 89, 153
 - remove, 102
 - replace, 135
 - setdefault, 117
 - sintassi, 174
 - sort, 100, 107, 129, 190
 - split, 103, 124
 - strip, 90, 135
 - translate, 135
 - update, 127
 - values, 112
 - vuoto, 100
- metodo collegato, 204, 209
- metodo di Newton, 72
- Meyers, Chris, vi
- min, funzione, 125, 126
- minestrone, 141
- Moby Project, 89
- modalità interattiva, 2, 9, 14, 28
- modalità script, 2, 9, 14, 28
- modello mentale, 219
- modificatore, 167, 170
- modulo, 22, 31
 - anydbm, 151
 - bisect, 109, 228
 - copy, 161
 - datetime, 171
 - Gui, 199
 - HTMLParser, 211
 - Image, 210
 - os, 149
 - pickle, 147, 152
 - pprint, 121

- profile, 143
- random, 109, 129, 136, 190
- reload, 154, 215
- shelve, 152
- string, 135
- structshape, 130
- time, 109
- urllib, 156, 211
- Visual, 182
- vpython, 182
- World, 164
- modulo, operatore, 45, 53
- modulo, scrittura, 154
- molteplicità (nel diagramma di classe), 192, 195
- Monty Python e il Sacro Graal, 166
- MP3, 153
- mro, metodo, 193
- mutabilità, 80, 98, 100, 105, 119, 123, 130, 160
- NameError, 27, 217
- nidificata, lista, 99
- nome, argomento con, 200
- None, valore speciale, 28, 58, 66, 100, 102
- not, operatore, 46
- notazione a punto, 22, 31, 81
- numero casuale, 136
- O-grande, notazione, 225
- Obama, Barack, 223
- oggetto, 80, 85, 104, 108, 157
 - Callable, 206
 - Canvas, 164
 - classe, 158, 163, 237
 - contenuto, 182
 - copia, 161
 - Evento, 207
 - file, 89, 94
 - funzione, 24, 30, 32, 237
 - modulo, 22, 154
 - mutabile, 160
 - stampa, 174
- oggetto contenuto (embedded), 160, 163
 - copia, 162
- oggetto mutabile, come valore di default, 182
- Olin College, vi
- omofono, 122
- open, funzione, 89, 90, 147, 150, 151
- operando, 13, 18
- operator overloading, 178, 182, 187
- operatore, 18
 - and, 46
 - aritmetico, 13
 - bitwise, 13
 - booleano, 82
 - confronto, 46
 - del, 102
 - di aggiornamento, 101
 - di formato, 148, 155, 217
 - di slicing, 79, 86, 99, 106, 124
 - in, 82, 98, 112
 - is, 104, 162
 - logico, 45, 46
 - modulo, 45, 53
 - not, 46
 - or, 46
 - parentesi quadre, 77, 97, 124
 - relazionale, 187
 - stringa, 16
- opzione, 200, 209
- or, operatore, 46
- ordinamento, 227
- ordine delle operazioni, 15, 17, 219
- ordine di risoluzione dei metodi, 193
- os, modulo, 149
- other (nome di parametro), 176
- OverflowError, 52
- overloading, 182
- pacchetto, 35
- packing dei controlli, 204, 210
- palindromo, 67, 86, 92, 94
- parametri, 27
- parametro, 26, 31, 105
 - opzionale, 138, 176
 - other, 176
 - raccolta, 125
 - self, 175
- parentesi
 - argomento in, 21
 - classe madre in, 190
 - corrispondenza, 4
 - graffe, 111
 - parametri in, 26, 27
 - priorità nella precedenza, 15
 - tuple in, 123
 - vuote, 23, 82
- parentesi quadre, operatore, 77, 97, 124
- parola chiave, 12, 13, 18, 214

- def, 23
- elif, 47
- else, 47
- parola, frequenza, 135
- parola, riducibile, 122, 132
- parsing, 6, 9
- pass, istruzione, 47
- passo di corsa, 19, 170
- pdb (Python debugger), 217
- peggior bug, 182, 211
- PEMDAS, 15
- percorso, 149, 155
 - assoluto, 149, 156
 - relativo, 149, 156
- permesso, accesso a file, 150
- persistenza, 147, 155
- pi, 22, 76
- pickle, modulo, 147, 152
- pickling, 152
- PIL (Python Imaging Library), 210
- pipe, 152
- pixel, coordinate, 208
- poesia, 6
- poker, 185, 195
- poligono, funzione, 38
- polimorfismo, 180, 182, 193
- pop, metodo, 102, 189
- popen, funzione, 153
- portabilità, 1, 8
- postcondizione, 42, 43, 65, 193
- potenziata, assegnazione, 101
- pprint, modulo, 121
- precedenza, 18, 219
- precondizione, 42, 43, 65, 108, 193
- prefisso, 141
- print, funzione, 7
- print, istruzione, 7, 177, 218
- profile, modulo, 143
- profonda, copia, 162
- progettazione
 - orientata agli oggetti, 181
- Progetto Gutenberg, 135
- programma, 3, 9
 - test, 93
- programmazione
 - a tentoni, 220
 - linguaggio di, 1
 - orientata agli eventi, 201, 209, 210
 - orientata agli oggetti, 173, 182, 190
- prompt, 2, 9, 51
- prosa, 6
- prototipo ed evoluzioni, 166, 168, 170
- pseudocasuale, 136, 144
- punto di intersezione, 225
- Punto, classe, 157, 177, 237
- punto, matematico, 157
- Puzzler, 94, 122, 132
- Python 3, 7, 14, 51, 120, 126
- Python debugger (pdb), 217
- Python Imaging Library (PIL), 210
- python.org, 10
- quesito, 94
- rabbia, 220
- raccolta, 125, 131
- radd, metodo, 179
- radiante, 22
- radice quadrata, 72
- radix sort, 223
- raise, istruzione, 115, 170
- Ramanujan, Srinivasa, 76
- ramificazione, 47, 53
- randint, funzione, 109, 136
- random, funzione, 129, 136
- random, modulo, 109, 129, 136, 190
- rappresentazione, 157, 159, 185
 - di stringa, 177
- raw_input, funzione, 51
- RB-albero, 229
- read, metodo, 153
- readline, metodo, 89, 153
- refactoring, 40, 41, 43, 194
- regole di precedenza, 15, 18
- rehashing, 231
- reload, funzione, 154, 215
- remove, metodo, 102
- replace, metodo, 135
- repr, funzione, 155
- Repubblica Ceca, bandiera, 164
- Rettangolo, classe, 159, 237
- return, istruzione, 49, 57, 220
- reversed, funzione, 130
- ricerca, 115, 228
 - binaria, 109, 228
 - lineare, 228
 - schema di, 81, 85, 91
- riconoscimento del problema, 92–94
- ricorsione, 48, 49, 53, 61, 63, 235
 - caso base, 50

- infinita, 50, 53, 64, 215, 216
- ridimensionamento geometrico, 232
- ridondanza, 6
- riducibile, parola, 122
- riduzione, schema, 101, 108
- referimento, 105, 108
 - alias, 105
- ripetizione, 36
 - lista, 99
- risoluzione di problemi, 1
- ritorno a capo, 51, 69, 189
- rotazione
 - lettere, 122
- rotazione di lettere, 86
- RSA, algoritmo, 120
- RuntimeError, 51, 64
- salto sulla fiducia, 63
- scambio, schema di, 124
- Scarabeo, 132
- schema
 - decorate-sort-undecorate, 129
 - di ricerca, 81, 85, 91, 115
 - di scambio, 124
 - DSU, 129, 138
 - filtro, 101, 108
 - guardiani, 65, 84
 - mappa, 101, 108
 - riduzione, 101, 108
- Schmidt, Eric, 223
- script, 3, 9
- self (nome di parametro), 175
- semantica, 4, 9, 173
 - errore di, 4, 9, 12, 17, 213, 218
- semantica, errore di, 84
- seme, 185
- seno, funzione, 22
- sequenza, 77, 85, 97, 103, 123, 130
 - di formato, 148, 155
- sequenze di coordinate, 202
- serie, condizioni in, 47
- sessagesimale, 168
- setdefault, metodo, 117
- shallow, copia, 162
- shell, 152
- shelve, modulo, 152
- shuffle, funzione, 190
- simbolo, 5, 9
- SimpleTurtleWorld, classe, 204
- singleton, 116, 121, 123
- sintassi, 4, 9, 173, 214
 - errore di, 4, 9, 17, 213
- slice, 85
- slicing
 - aggiornamento, 100
 - copia, 80, 100
 - lista, 99
 - stringa, 79
 - tupla, 124
- slicing, operatore di, 79, 86, 99, 106, 124
- smistamento in base al tipo, 179, 182
- soggetto, 175, 182, 204
- soluzione di problemi, 8
- sort, metodo, 100, 107, 129, 190
- sorted, funzione, 130
- sottoclasse, 190
- sottrazione
 - con prestito, 74, 169
 - dizionario, 139
- sovrascrittura, 139, 144, 176, 187, 190, 193
- spazi, 30, 214
- spaziatore, 52, 90, 155
- spirale, 44
- spirale di Archimede, 44
- split, metodo, 103, 124
- sqrt, funzione, 22, 59
- stable sort, 227
- stampa, istruzione di, 7, 9
- step, ampiezza, 86
- stile di programmazione funzionale, 168, 170
- __str__, metodo, 177, 189
- str, funzione, 22
- string, modulo, 135
- string, tipo, 11
- stringa, 11, 18, 103, 130
 - a righe multiple, 42, 214
 - accumulatore, 189
 - concatenamento, 226
 - confronto, 83
 - di documentazione, 42
 - di formato, 148, 155
 - documentazione, 43
 - immutabile, 80
 - metodi, 81, 82, 226
 - operazioni, 16
 - rappresentazione, 155
 - slicing, 79
 - triple virgolette, 42
 - vuota, 85, 103

- stringa evento, 207
- strip, metodo, 90, 135
- structshape, modulo, 130
- struttura, 5
- struttura di dati, 130, 131, 142
- suffisso, 141
- sum, funzione, 126
- superclasse, 190
- SVG, 211
- sviluppo incrementale, 67, 213
- sviluppo pianificato, 168, 170
- Swampy, 35, 164, 196, 199, 233
- SyntaxError, 23
- tabella hash, 112, 121, 228
- Tagger, 196
- tecnica di sviluppo, 43, 193
 - incapsulamento e generalizzazione, 41
 - incrementale, 58, 213
 - pianificato, 168
 - programmazione a tentoni, 144, 220
 - prototipo ed evoluzioni, 166, 168
 - riconoscimento del problema, 92, 93
- tempo costante, 231
- Tempo, classe, 165
- teorema di Pitagora, 58
- termine dominante, 224
- Tesi di Turing, 61
- test
 - caso di prova minimo, 218
 - difficoltà, 93
 - e assenza di bug, 93
 - modalità interattiva, 3
 - salto sulla fiducia, 63
 - sapere il risultato, 59
 - sviluppo incrementale, 58
- testo
 - casuale, 141
 - semplice, 89, 135, 211
- Text, controllo, 202
- time, modulo, 109
- tipo, 11, 17
 - bool, 45
 - conversione di, 21
 - definito dall'utente, 157, 165
 - dict, 111
 - file, 147
 - float, 11
 - int, 11
 - list, 97
 - long, 120
 - set, 140
 - str, 11
 - tuple, 123
- Tkinter, 199
- token, 5, 9
- torta, 43
- traceback, 28, 31, 50, 52, 115, 217
- translate, metodo, 135
- triangolo, 54
- trova, funzione, 80
- True, valore speciale, 45
- try, istruzione, 150
- tupla, 123, 125, 130, 131
 - assegnazione, 124–126, 131
 - come chiave di dizionario, 128, 142
 - confronto, 129, 188
 - in parentesi quadre, 128
 - metodi, 226
 - singleton, 123
 - slicing, 124
- tuple, funzione, 123
- Turing, Alan, 61
- TurtleWorld, 35, 54, 196
- type, funzione, 163
- TypeError, 77, 80, 116, 124, 125, 149, 176, 217
- uguaglianza e assegnazione, 69
- UML, 233, 238
- UnboundLocalError, 119
- underscore, carattere, 13
- unicità, 109
- Unified Modeling Language, 233
- update, metodo, 127
- URL, 156, 211
- urllib, modulo, 156, 211
- uso prima di def, 17, 25
- valore, 11, 17, 104, 121
 - tupla, 125
- valore (delle carte da gioco), 185
- valore di default, 139, 176
 - evitare i mutabili, 182
- valore di ritorno, 21, 31, 57, 160
 - tupla, 125
- valore speciale
 - False, 45
 - None, 28, 58, 66, 100, 102
 - True, 45
- ValueError, 52, 115, 124

- values, metodo, 112
- valutazione, 14, 18
- variabile, 12, 18
 - a livello di modulo, 234
 - aggiornamento, 70
 - globale, 118, 234
 - aggiornamento, 119
 - locale, 27, 31
 - temporanea, 57, 66, 219
- virgola mobile, 17, 73
- virgolette, 42, 214
- Visual, modulo, 182
- vorpale, 62
- vpython, modulo, 182

- widget, 200, 209
- World, modulo, 164

- zero, indice iniziale, 77, 98
- zip, funzione, 126
 - uso con dict, 127
- Zipf, legge di, 144