

Pensare in Python

Come pensare da Informatico

Seconda Edizione, Versione 2.2.23

Pensare in Python

Come pensare da Informatico

Seconda Edizione, Versione 2.2.23

Allen Downey

Green Tea Press

Needham, Massachusetts

Copyright © 2015 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.thinkpython2.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 avevo tenuto già tre volte, ma ne stavo diventando sempre più insoddisfatto. 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.

Da allora, 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. Per la maggior parte di essi, c'è un collegamento web alla soluzione.
- Ho aggiunto una serie di esercitazioni - esempi più articolati con esercizi, soluzioni e discussione.
- Ho ampliato la trattazione sui metodi di sviluppo di un programma e sugli schemi fondamentali di progettazione.
- Ho aggiunto delle appendici sul debug e l'analisi degli algoritmi.

Novità di questa seconda edizione:

- Il libro e tutto il codice di supporto sono stati aggiornati a Python 3.
- Ho aggiunto alcuni paragrafi, con ulteriori dettagli sul web, per aiutare i meno esperti ad iniziare a usare Python in un browser, in modo da non doverne affrontare l'installazione fino a quando non si sentiranno pronti.
- Per il capitolo 4, al posto del mio pacchetto grafico basato su *turtle graphics*, chiamato Swampy, ho adottato il modulo *turtle* di Python, più standard, potente e facile da installare.
- Ho inserito un nuovo capitolo chiamato "Ulteriori strumenti", che presenta alcune funzionalità aggiuntive di Python non indispensabili, ma che possono tornare utili.

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

Allen B. Downey

Olin College

Ringraziamenti

Grazie infinite a Jeff Elkner, che ha adattato a Python il mio libro su Java, 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 a *How to Think Like a Computer Scientist*.

Grazie ai redattori di O'Reilly Media che hanno lavorato a *Think Python*.

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 errata `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 Hagtvædt 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 11.5.
- 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 14.3.
- 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.
- 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.
- Daniel Neilson ha corretto un errore nell'ordine delle operazioni.
- Will McGinnis ha evidenziato che `polilinea` era definita in modo diverso in due punti.
- Frank Hecker ha osservato un esercizio non ben 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.
- Sven Hoexter ha osservato che una variabile di nome `input` oscura una funzione predefinita.
- Stephen Gregory ha evidenziato il problema di `cmp` in Python 3.

- Ishwar Bhat ha corretto la mia formulazione dell'ultimo teorema di Fermat.
- Andrea Zanella ha tradotto il libro in italiano e, strada facendo, ha inviato alcune correzioni.
- Mille grazie a Melissa Lewis e a Luciano Ramalho per gli eccellenti commenti e suggerimenti alla seconda edizione.
- Grazie a Harry Percival di PythonAnywhere per il suo aiuto a chi vuole iniziare ad usare Python in un web browser.
- Xavier Van Aubel ha prodotto alcune utili correzioni alla seconda edizione.
- William Murray ha puntualizzato la mia definizione di divisione intera.
- Per Starbäck mi ha aggiornato sui ritorni a capo universali in Python 3.

Hanno inoltre segnalato errori di stampa o indicato correzioni: Czeslaw Czapla, Richard Fursa, Brian McGhie, Lokesh Kumar Makani, Matthew Shultz, Viet Le, Victor Simeone, Lars O.D. Christensen, Swarup Sahoo, Alix Etienne, Kuang He, Wei Huang, Karen Barber, e Eric Ransom.

Indice

Prefazione	v
1 Lo scopo del programma	1
1.1 Che cos'è un programma?	1
1.2 Avviare Python	2
1.3 Il primo programma	3
1.4 Operatori aritmetici	3
1.5 Valori e tipi	4
1.6 Linguaggi formali e linguaggi naturali	5
1.7 Debug	6
1.8 Glossario	7
1.9 Esercizi	8
2 Variabili, espressioni ed istruzioni	9
2.1 Istruzioni di assegnazione	9
2.2 Nomi delle variabili	9
2.3 Espressioni e istruzioni	10
2.4 Modalità script	11
2.5 Ordine delle operazioni	12
2.6 Operazioni sulle stringhe	12
2.7 Commenti	13
2.8 Debug	14
2.9 Glossario	14
2.10 Esercizi	15

3	Funzioni	17
3.1	Chiamate di funzione	17
3.2	Funzioni matematiche	18
3.3	Composizione	19
3.4	Aggiungere nuove funzioni	19
3.5	Definizioni e loro utilizzo	20
3.6	Flusso di esecuzione	21
3.7	Parametri e argomenti	22
3.8	Variabili e parametri sono locali	23
3.9	Diagrammi di stack	23
3.10	Funzioni “produttive” e funzioni “vuote”	24
3.11	Perché le funzioni?	25
3.12	Debug	25
3.13	Glossario	26
3.14	Esercizi	27
4	Esercitazione: Progettazione dell’interfaccia	29
4.1	Il modulo turtle	29
4.2	Ripetizione semplice	30
4.3	Esercizi	31
4.4	Incapsulamento	32
4.5	Generalizzazione	32
4.6	Progettazione dell’interfaccia	33
4.7	Refactoring	34
4.8	Tecnica di sviluppo	35
4.9	Stringa di documentazione	36
4.10	Debug	36
4.11	Glossario	37
4.12	Esercizi	37

Indice	xv
5 Istruzioni condizionali e ricorsione	39
5.1 Divisione intera e modulo	39
5.2 Espressioni booleane	40
5.3 Operatori logici	40
5.4 Esecuzione condizionale	41
5.5 Esecuzione alternativa	41
5.6 Condizioni in serie	42
5.7 Condizioni nidificate	42
5.8 Ricorsione	43
5.9 Diagrammi di stack delle funzioni ricorsive	44
5.10 Ricorsione infinita	45
5.11 Input da tastiera	45
5.12 Debug	46
5.13 Glossario	47
5.14 Esercizi	48
6 Funzioni produttive	51
6.1 Valori di ritorno	51
6.2 Sviluppo incrementale	52
6.3 Composizione	54
6.4 Funzioni booleane	55
6.5 Altro sulla ricorsione	55
6.6 Salto sulla fiducia	57
6.7 Un altro esempio	58
6.8 Controllo dei tipi	58
6.9 Debug	59
6.10 Glossario	61
6.11 Esercizi	61

7	Iterazione	63
7.1	Riassegnazione	63
7.2	Aggiornare le variabili	64
7.3	L'istruzione while	64
7.4	break	66
7.5	Radici quadrate	66
7.6	Algoritmi	68
7.7	Debug	68
7.8	Glossario	69
7.9	Esercizi	69
8	Stringhe	71
8.1	Una stringa è una sequenza	71
8.2	len	72
8.3	Attraversamento con un ciclo for	72
8.4	Slicing	73
8.5	Le stringhe sono immutabili	74
8.6	Ricerca	75
8.7	Cicli e contatori	75
8.8	Metodi delle stringhe	76
8.9	L'operatore in	77
8.10	Confronto di stringhe	77
8.11	Debug	78
8.12	Glossario	79
8.13	Esercizi	80
9	Esercitazione: Giochi con le parole	83
9.1	Leggere elenchi di parole	83
9.2	Esercizi	84
9.3	Ricerca	85
9.4	Cicli con gli indici	86
9.5	Debug	87
9.6	Glossario	88
9.7	Esercizi	88

Indice	xvii
10 Liste	91
10.1 Una lista è una sequenza	91
10.2 Le liste sono mutabili	92
10.3 Attraversamento di una lista	93
10.4 Operazioni sulle liste	93
10.5 Slicing delle liste	94
10.6 Metodi delle liste	94
10.7 Mappare, filtrare e ridurre	95
10.8 Cancellare elementi	96
10.9 Liste e stringhe	96
10.10 Oggetti e valori	97
10.11 Alias	98
10.12 Liste come argomenti	99
10.13 Debug	100
10.14 Glossario	102
10.15 Esercizi	102
11 Dizionari	105
11.1 Un dizionario è una mappatura	105
11.2 Il dizionario come raccolta di contatori	106
11.3 Cicli e dizionari	108
11.4 Lookup inverso	108
11.5 Dizionari e liste	109
11.6 Memoizzazione	111
11.7 Variabili globali	112
11.8 Debug	113
11.9 Glossario	114
11.10 Esercizi	115

12 Tuple	117
12.1 Le tuple sono immutabili	117
12.2 Assegnazione di tupla	118
12.3 Tuple come valori di ritorno	119
12.4 Tuple di argomenti a lunghezza variabile	120
12.5 Liste e tuple	120
12.6 Dizionari e tuple	122
12.7 Sequenze di sequenze	123
12.8 Debug	124
12.9 Glossario	124
12.10 Esercizi	125
13 Esercitazione: Scelta della struttura di dati	127
13.1 Analisi di frequenza delle parole	127
13.2 Numeri casuali	128
13.3 Istogramma di parole	129
13.4 Parole più comuni	130
13.5 Parametri opzionali	131
13.6 Sottrazione di dizionari	131
13.7 Parole a caso	132
13.8 Analisi di Markov	133
13.9 Strutture di dati	134
13.10 Debug	135
13.11 Glossario	137
13.12 Esercizi	137
14 File	139
14.1 Persistenza	139
14.2 Lettura e scrittura	139
14.3 L'operatore di formato	140
14.4 Nomi di file e percorsi	141
14.5 Gestire le eccezioni	142

Indice	xix
14.6 Database	143
14.7 Pickling	144
14.8 Pipe	145
14.9 Scrivere moduli	145
14.10 Debug	146
14.11 Glossario	147
14.12 Esercizi	148
15 Classi e oggetti	149
15.1 Tipi personalizzati	149
15.2 Attributi	150
15.3 Rettangoli	151
15.4 Istanze come valori di ritorno	152
15.5 Gli oggetti sono mutabili	153
15.6 Copia	153
15.7 Debug	155
15.8 Glossario	155
15.9 Esercizi	156
16 Classi e funzioni	157
16.1 Tempo	157
16.2 Funzioni pure	158
16.3 Modificatori	159
16.4 Sviluppo prototipale e Sviluppo pianificato	160
16.5 Debug	161
16.6 Glossario	162
16.7 Esercizi	162
17 Classi e metodi	165
17.1 Funzionalità orientate agli oggetti	165
17.2 Stampa di oggetti	166
17.3 Un altro esempio	167

17.4	Un esempio più complesso	168
17.5	Il metodo speciale <code>init</code>	168
17.6	Il metodo speciale <code>__str__</code>	169
17.7	Operator overloading	170
17.8	Smistamento in base al tipo	170
17.9	Polimorfismo	171
17.10	Debug	172
17.11	Interfaccia e implementazione	173
17.12	Glossario	173
17.13	Esercizi	174
18	Ereditarietà	175
18.1	Oggetti Carta	175
18.2	Attributi di classe	176
18.3	Confrontare le carte	177
18.4	Mazzi di carte	178
18.5	Stampare il mazzo	178
18.6	Aggiungere, togliere, mescolare e ordinare	179
18.7	Ereditarietà	180
18.8	Diagrammi di classe	181
18.9	Debug	182
18.10	Incapsulamento dei dati	183
18.11	Glossario	184
18.12	Esercizi	186
19	Ulteriori strumenti	189
19.1	Espressioni condizionali	189
19.2	List comprehension	190
19.3	Generator expression	191
19.4	<code>any</code> e <code>all</code>	192
19.5	Insiemi (<code>set</code>)	192
19.6	Contatori	193

Indice	xxi
19.7 defaultdict	194
19.8 Tuple con nome (namedtuple)	196
19.9 Raccolta di argomenti con nome	197
19.10 Glossario	197
19.11 Esercizi	198
A Debug	199
A.1 Errori di sintassi	199
A.2 Errori di runtime	201
A.3 Errori di semantica	204
B Analisi degli Algoritmi	209
B.1 Ordine di complessità	210
B.2 Analisi delle operazioni fondamentali di Python	212
B.3 Analisi degli algoritmi di ricerca	214
B.4 Tabelle hash	214
B.5 Glossario	218

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

1.1 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, o ancora operazioni grafiche come elaborare un'immagine o riprodurre un filmato.

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, dalla rete o da un altro dispositivo.

output: scrittura di dati sullo schermo, salvataggio su un file o trasmissione verso la rete, ecc.

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.

1.2 Avviare Python

Un possibile scoglio nell'iniziare ad usare Python è quello di doverlo installare, con il software correlato, nel vostro computer. Se siete già pratici del vostro sistema operativo, e soprattutto se ve la cavate con l'interfaccia a riga di comando, non avrete nessun problema ad installare Python. Ma per i meno esperti, può risultare faticoso dover imparare contemporaneamente l'amministrazione del sistema e la programmazione.

A chi dovesse trovare difficoltà, suggerisco per il momento di avviare Python all'interno di un browser web. Più avanti, una volta presa confidenza con Python, fornirò dei suggerimenti per l'installazione.

Esistono alcuni siti web che permettono di usare Python senza doverlo installare. Se avete già dimestichezza con uno di questi siti, usate pure quello. Altrimenti, vi consiglio PythonAnywhere. Trovate le istruzioni dettagliate per iniziare all'indirizzo <http://tinyurl.com/thinkpython2e>.

Ci sono due versioni di Python, chiamate Python 2 e Python 3. Sono molto simili, pertanto imparandone una non è difficile passare poi all'altra. Di fatto, ai primi livelli di apprendimento le differenze tra le due versioni sono poche. Questo libro fa riferimento alla più recente versione Python 3, ma troverete anche alcune annotazioni su Python 2.

L'**interprete** di Python è un programma che legge ed esegue il codice Python. A seconda del vostro ambiente di lavoro, lo potete avviare facendo click su un'icona, oppure digitando `python` in una riga di comando. In alcune installazioni di Python, è compreso anche un ambiente di sviluppo di base chiamato IDLE. All'avvio, dovrete vedere un output simile a questo:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Le prime tre righe contengono informazioni sull'interprete e il sistema operativo in cui viene eseguito, per cui nel vostro caso concreto potrebbero essere diverse. Ma occhio al numero di versione, che in questo esempio è 3.4.0: comincia con 3, il che significa che state usando Python 3. Se cominciasse con 2, vorrebbe dire che state usando (avete indovinato!) Python 2.

L'ultima riga è un **prompt**, che comunica che l'interprete è pronto a ricevere il codice che inserirete. Se scrivete una riga di codice e poi premete Invio, l'interprete elabora immediatamente il risultato:

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

Ora siete pronti per iniziare. D'ora in poi, darò per scontato che sappiate come avviare l'interprete di Python ed eseguire del codice.

1.3 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.

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

In Python 2, l'istruzione di stampa è leggermente diversa: non è una funzione, per cui non si usano le parentesi.

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

La differenza sarà presto chiarita meglio, ma questo ci basta per cominciare.

1.4 Operatori aritmetici

Dopo "Ciao, Mondo!", passiamo all'aritmetica. Python dispone di **operatori**, che sono simboli speciali che rappresentano i calcoli fondamentali, come l'addizione e la moltiplicazione.

Gli operatori `+`, `-`, e `*` eseguono nell'ordine addizione, sottrazione e moltiplicazione, come negli esempi seguenti:

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

L'operatore `/` esegue la divisione:

```
>>> 84 / 2
42.0
```

Vi chiederete come mai il risultato è 42.0 anziché 42. Lo vedremo nel prossimo paragrafo.

Infine, l'operatore `**` esegue l'elevamento a potenza; ovvero, calcola la potenza di un numero:

```
>>> 6**2 + 6
42
```

In altri linguaggi viene usato il simbolo \wedge per le potenze, ma in Python questo è un operatore bitwise chiamato XOR. Se non siete pratici di questi operatori, il risultato vi lascerà sorpresi:

```
>>> 6 ^ 2
4
```

Non tratteremo gli operatori bitwise in questo libro, ma se volete approfondire l'argomento andate sul sito <http://wiki.python.org/moin/BitwiseOperators>.

1.5 Valori e tipi

Un **valore** è l'elemento fondamentale con cui un programma lavora, come lo è una lettera dell'alfabeto nella scrittura o un numero in matematica. I valori che abbiamo visto finora sono 2, 42.0, e 'Ciao, Mondo!'.

Questi valori appartengono a **tipi** diversi: 2 è un numero **intero**, 42.0 è un numero decimale, detto anche "a virgola mobile" o **floating-point**, e 'Ciao, Mondo!' è una **stringa**, in quanto contiene una serie di caratteri racchiusi tra due apici.

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

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Ciao, Mondo!')
<class 'str'>
```

Nei responsi, la parola "class" (classe) viene usata nel senso di categoria; un tipo è una categoria di valori.

Ovviamente le stringhe sono di tipo `str`, gli interi di tipo `int`, i numeri con il punto decimale di tipo `float`.

Cosa dire di valori come '2' e '42.0'? Sembrano a prima vista dei numeri, ma notate che sono racchiusi tra apici e questo sicuramente significa qualcosa.

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

Infatti non sono numeri, ma stringhe.

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:

```
>>> 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. Approfondiremo meglio questo tipo di sequenza più avanti.

1.6 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** che governa ciò che devono esprimere. 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.

Questa è un'@ frase ben \$strutturata in italiano che conti&ne simb*li non validi.
Frase questa simboli validi tutti ha, ma struttura non valida con.

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**.

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 dei linguaggi naturali, 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.

I linguaggi formali sono molto più ricchi di significato dei linguaggi naturali, per questo è necessario più tempo per leggerli e comprenderli. Inoltre, 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: dovete invece imparare 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.

1.7 Debug

I programmatori inevitabilmente commettono errori. Per ragioni bizzarre, gli errori di programmazione sono chiamati **bug**, ed il procedimento della loro ricerca e correzione è chiamato **debug**.

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!

1.8 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.

interprete: Un programma che legge un altro programma e lo esegue.

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

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

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

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

valore: Una unità fondamentale di dati, come un numero o una stringa, che un programma elabora.

tipo: Una categoria di valori. I tipi visti finora sono gli interi (tipo `int`), numeri a virgola mobile o floating-point (tipo `float`), e stringhe (tipo `str`).

intero: Tipo che rappresenta i numeri interi.

floating-point: Tipo che rappresenta i numeri con parte decimale.

stringa: Tipo che rappresenta sequenze di caratteri.

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 di 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.

sintassi: Le regole che governano la struttura di un programma.

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

bug: Un errore in un programma.

debug: L'operazione di ricerca e di rimozione degli errori di programmazione.

1.9 Esercizi

Esercizio 1.1. È opportuno leggere questo libro davanti al computer, in modo da poter provare gli esempi man mano che procedete nella lettura.

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 nella programmazione, 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.

1. In un'istruzione di stampa, cosa succede se dimenticate una delle parentesi, o entrambe?
2. Se state cercando di stampare una stringa, cosa succede se dimenticate uno degli apici, o entrambi?
3. Per rendere negativo un numero, gli si antepone un segno meno, come in `-2`. Cosa succede se antepone un segno più ad un numero? E nel caso di `2++2`?
4. Nella notazione matematica, gli zeri iniziali sono ammessi, come in `02`. In questo caso, cosa succede in Python?
5. Cosa succede se scrivete due valori, senza inserire in mezzo un operatore?

Esercizio 1.2. Avviate l'interprete di Python e utilizzatelo come calcolatrice.

1. Quanti secondi ci sono in 42 minuti e 42 secondi?
2. Quante miglia ci sono in 10 chilometri? Suggerimento: un miglio equivale a 1,61 km.
3. Se correte una gara di 10 chilometri in 42 minuti e 42 secondi, qual è la vostra cadenza media (tempo per miglio in minuti e secondi)? Qual è la vostra velocità media, in miglia all'ora?

Capitolo 2

Variabili, espressioni ed istruzioni

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.

2.1 Istruzioni di assegnazione

Un'**istruzione di assegnazione** serve a creare una nuova variabile, specificandone il nome, e ad assegnarle un valore:

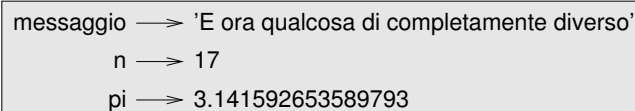
```
>>> messaggio = 'E ora qualcosa di completamente diverso'  
>>> n = 17  
>>> pi = 3.141592653589793
```

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 loro valore. 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.

2.2 Nomi delle variabili

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



```
messaggio —> 'E ora qualcosa di completamente diverso'  
n —> 17  
pi —> 3.141592653589793
```

Figura 2.1: Diagramma di stato.

I nomi delle variabili possono essere lunghi a piacere e possono contenere sia lettere che numeri, ma non possono iniziare con un numero. È possibile usare anche le lettere maiuscole, ma per i nomi di variabile è convenzione utilizzare solo lettere minuscole. Ricordate comunque che, per l'interprete, maiuscole e minuscole sono diverse, pertanto spam, Spam e SPAM sono variabili diverse.

Il trattino basso o *underscore*, `_`, 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 3 ha queste parole chiave:

<code>False</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
<code>None</code>	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
<code>True</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>
<code>and</code>	<code>del</code>	<code>global</code>	<code>not</code>	<code>with</code>
<code>as</code>	<code>elif</code>	<code>if</code>	<code>or</code>	<code>yield</code>
<code>assert</code>	<code>else</code>	<code>import</code>	<code>pass</code>	
<code>break</code>	<code>except</code>	<code>in</code>	<code>raise</code>	

Non occorre imparare a memoria questo elenco. Nella maggior parte degli ambienti di sviluppo, le parole chiave vengono evidenziate con un diverso colore; se cercate di usarne una come nome di variabile, ve ne accorgete subito.

2.3 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 `n` sia già stato assegnato un valore):

```
>>> 42
42
>>> n
17
>>> n + 25
42
```


Quando scrivete un'espressione al prompt dei comandi, l'interprete la **valuta**, cioè trova il valore dell'espressione. Nell'esempio di prima, `n` ha valore 17 e `n + 25` ha valore 42.

Un'**istruzione** è una porzione di codice che l'interprete Python può eseguire e che ha un qualche effetto, come creare una variabile o mostrare un valore.

```
>>> n = 17
>>> print(n)
```

La prima riga è un'istruzione di assegnazione che dà un valore alla variabile `n`. La seconda è un'istruzione di stampa che mostra a video il valore di `n`.

Quando scrivete un'istruzione, l'interprete la **esegue**, cioè fa quello che l'istruzione dice di fare. In linea generale, le istruzioni, a differenza delle espressioni, non contengono valori.

2.4 Modalità script

Finora abbiamo avviato Python in **modalità interattiva**, detta anche “a riga di comando”, che vuol dire interagire direttamente con l'interprete. La modalità interattiva è un buon modo per iniziare e fare esperimenti, ma se si deve lavorare con più di qualche riga di codice, può diventare in breve tempo un impiccio.

In alternativa alla riga di comando, si può scrivere e salvare un programma in un file di testo semplice, chiamato **script**, ed usare poi l'interprete in **modalità script** per eseguirlo. Per convenzione, i file contenenti programmi Python hanno nomi che terminano con l'estensione `.py`.

Se già sapete come creare e avviare uno script nel vostro computer, siete a cavallo. Altrimenti vi consiglio di nuovo di usare PythonAnywhere. Le istruzioni per l'avvio in modalità script sono pubblicate all'indirizzo <http://tinyurl.com/thinkpython2e>.

Poiché Python consente entrambe queste modalità, potete provare dei 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, si può 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 avviate, 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 uno alla volta, 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.

Per controllare se avete capito tutto, 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 avviatelo. Qual è il risultato? Modificate lo script trasformando ciascuna espressione in un'istruzione di stampa, ed avviatelo nuovamente.

2.5 Ordine delle operazioni

Quando un'espressione contiene più operatori, l'ordine in cui viene eseguito il calcolo dipende dall'**ordine delle operazioni**. 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. Si possono usare le parentesi anche solo per rendere più leggibile un'espressione, come in $(minuti * 100) / 60$; in questo caso non influiscono sul risultato.
- **Elevamento a potenza**: ha la priorità successiva, così $1 + 2**3$ fa 9, e non 27, e $2 * 3**2$ fa 18, e non 36.
- **Moltiplicazione e Divisione** hanno priorità superiore ad **Addizione** e **Sottrazione**. Per cui $2*3-1$ 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 $gradi / 2 * pi$, la divisione viene calcolata per prima e il risultato viene moltiplicato per pi . Per dividere per 2π , dovete usare le parentesi o scrivere $gradi / 2 / pi$.

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

2.6 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'
```

Ma ci sono due eccezioni: + e *.

L'operatore + esegue il **concatenamento**, cioè unisce le stringhe collegandole ai due estremi. Per esempio:

```
primo = 'bagno'
secondo = 'schiuma'
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?

2.7 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 e non ha alcun effetto sull'esecuzione del programma.

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 più utile spiegare *perché*.

Questo commento è ridondante e inutile:

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

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

```
v = 5      # velocità 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.

2.8 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.

2.8.1 Errori di sintassi

Il termine **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**.

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.

2.8.2 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.

2.8.3 Errori di semantica

Il terzo tipo di errore è l'**errore di semantica** (o di logica), che è correlato al significato del programma. In presenza di un errore di semantica, il programma verrà eseguito senza che compaia alcun messaggio di errore, ma non farà la cosa giusta: farà qualcosa di diverso. Nello specifico, farà esattamente ciò che voi gli avete detto di fare, esprimendovi in modo 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.

2.9 Glossario

variabile: Un nome che fa riferimento ad un valore.

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 usata per analizzare il programma e che non può essere usata come nome di variabile o di funzione, come `if`, `def`, e `while`.

operando: Uno dei valori sui quali agisce un operatore.

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.

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

eseguire: Dare effetto a un'istruzione e fare ciò che dice.

modalità interattiva: Un modo di usare l'interprete Python, scrivendo del codice al prompt.

modalità script: Un modo di usare l'interprete Python, leggendo del codice da uno script ed eseguendolo.

script: Un programma memorizzato in un file.

ordine delle operazioni: 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.

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 è in esecuzione.

semantica: Il significato logico di un programma.

errore di semantica: Errore nel programma tale per cui esso produce risultati diversi da quelli che il programmatore si aspettava.

2.10 Esercizi

Esercizio 2.1. *Rinnovo la raccomandazione del capitolo precedente: ogni volta che apprendete qualcosa di nuovo, provatelo in modalità interattiva e fate degli errori di proposito per vedere cosa non funziona.*

- Abbiamo visto che $n = 42$ è valido. E $42 = n$?
- E se scrivete $x = y = 1$?

- In alcuni linguaggi, ogni istruzione termina con un punto e virgola, ;. Cosa succede se mettete un punto e virgola alla fine di un'istruzione in Python?
- E se mettete un punto alla fine di un'istruzione?
- Nella notazione matematica potete indicare la moltiplicazione di x per y scrivendo: xy . Cosa succede scrivendo questo in Python?

Esercizio 2.2. 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?
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

Nell'ambito della programmazione, una **funzione** è una serie 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.

3.1 Chiamate di funzione

Abbiamo già visto un esempio di una **chiamata di funzione**:

```
>>> type(42)
<class '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, una volta eseguita l'elaborazione, “ritorna” o “restituisce” un risultato. Il risultato è detto **valore di ritorno**.

Python fornisce una raccolta di funzioni 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'
```

3.2 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 contenute in un modulo, dobbiamo dire all'interprete di caricare il modulo in memoria con un'**istruzione di importazione**:

```
>>> import math
```

Questa istruzione crea un **oggetto modulo** chiamato `math`. Se visualizzate l'oggetto modulo, ottenete alcune informazioni a riguardo:

```
>>> 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_segnaled / potenza_rumored
>>> decibel = 10 * math.log10(rapporto)
```

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

Il primo esempio utilizza la funzione `math.log10` per calcolare un rapporto segnale/rumore in decibel (a condizione che siano stati definiti i valori di `potenza_segnaled` e `potenza_rumored`). 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 180 e moltiplicare per π :

```
>>> gradi = 45
>>> radianti = gradi / 180.0 * math.pi
>>> math.sin(radianti)
0.707106781187
```

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

Se ricordate la trigonometria, potete verificare il risultato precedente confrontandolo con la radice quadrata di 2 diviso 2:


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

3.3 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 mattoni 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
```

3.4 Aggiungere nuove funzioni

Finora abbiamo usato solo funzioni predefinite o "built-in", che sono parte integrante di Python, ma è anche possibile crearne 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 è,")
    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 underscore (`_`) 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 obbligatoriamente indentato, cioè deve avere un rientro rispetto all'intestazione. Per convenzione, l'indentazione è sempre di quattro spazi. Il corpo può contenere un qualsiasi numero di istruzioni.

Le stringhe nelle istruzioni di stampa sono racchiuse 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, frequenti con l'italiano, bisogna usare le virgolette.

Virgolette e apici devono essere alti e di tipo indifferenziato, quelli che trovate tra i simboli in alto sulla vostra tastiera. Altre virgolette "tipografiche", come quelle in questa frase, non sono valide in Python.

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 è,")  
...     print("Con l'ascia in mano si sente un re.")  
...
```

Per concludere la funzione, dovete inserire una riga vuota.

La definizione di una funzione crea un **oggetto funzione** che è di tipo `function`:

```
>>> print(stampa_brani)  
<function stampa_brani at 0xb7e99e9c>  
>>> type(stampa_brani)  
<class '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 è,  
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 è,  
Con l'ascia in mano si sente un re.  
Terror di tutta la foresta egli è,  
Con l'ascia in mano si sente un re.
```

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

3.5 Definizioni e loro utilizzo

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

```
def stampa_brani():  
    print("Terror di tutta la foresta egli è,")  
    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 vengono eseguite fino a quando la funzione non viene chiamata, e la definizione di per sé non genera alcun risultato.

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

Come esercizio, 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.

Ora riportate la chiamata della funzione al suo posto, e spostate la definizione di `stampa_brani` dopo la definizione di `ripeti_brani`. Cosa succede quando avviate il programma?

3.6 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.

In conclusione, quando leggete un programma non limitatevi sempre a farlo dall'alto in basso. Spesso ha più senso cercare di seguire il flusso di esecuzione.

3.7 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 definizione 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(42)
42
42
>>> stampa2volte(math.pi)
3.14159265359
3.14159265359
```

Le stesse regole di composizione che valgono per le funzioni predefinite si applicano anche alle funzioni definite da un programmatore, 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`.

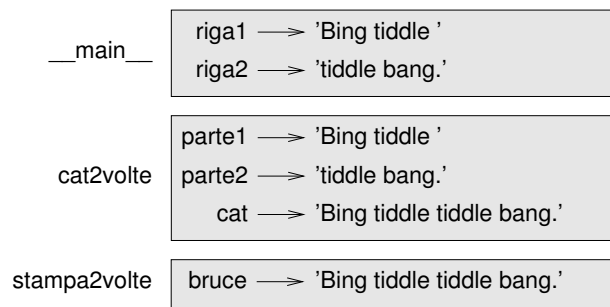


Figura 3.1: Diagramma di stack.

3.8 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`.

3.9 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.

3.10 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:

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

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

3.11 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.

3.12 Debug

Una delle più importanti abilità che acquisirete è quella 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 riprova. 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 funzionante, e fare via via piccole modifiche con rimozione degli errori.

Linux, per fare un esempio, è un sistema operativo che contiene milioni 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 scambiare una sequenza di AAAA in BBBB e viceversa. Questo in seguito diventò Linux". (*The Linux Users' Guide Beta Version 1*).

3.13 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 contiene.

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 restituisce sempre `None`.

None: Valore speciale restituito dalle funzioni vuote.

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, visualizzato quando si verifica un errore.

3.14 Esercizi

Esercizio 3.1. *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('monty')  
  
monty
```

Suggerimento: usate concatenamento delle stringhe e ripetizione. Inoltre, Python contiene una funzione predefinita chiamata `len` che restituisce la lunghezza di una stringa, ad esempio il valore di `len('monty')` è 5.

Esercizio 3.2. *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. *Copiate nel vostro script la definizione di `stampa_2volte` che abbiamo visto nel corso di questo capitolo.*

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://thinkpython2.com/code/do_four.py.

Esercizio 3.3. Nota: questo esercizio dovrebbe 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, stampate una sequenza di valori separati da virgole:

```
print('+', '-')
```

Di default, `print` va a capo; si può però variare questo comportamento e restare sulla stessa riga, inserendo uno spazio, in questo modo:

```
print('+', end=' ')
print('-')
```

L'output di queste istruzioni è `'+ -'`.

Una funzione `print` priva di argomento, termina la riga e va a capo.

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

Soluzione: <http://thinkpython2.com/code/grid.py>. Fonte: Esercizio tratto da Oualline, *Practical C Programming*, Third Edition, O'Reilly Media, 1997.

Capitolo 4

Esercitazione: Progettazione dell'interfaccia

Questo capitolo vi propone un'esercitazione che dimostra una procedura per progettare delle funzioni che collaborano tra loro.

Viene illustrato il modulo grafico `turtle` che vi permette di creare immagini utilizzando *turtle graphics*. Si tratta di un modulo già compreso nella maggior parte delle installazioni di Python; tuttavia, se usate PythonAnywhere, non sarete in grado di visualizzare gli esempi basati su `turtle` (almeno non nel momento in cui scrivo).

Se avete già installato Python sul vostro computer, gli esempi dovrebbero funzionare. Se non lo avete ancora installato, questo è il momento buono per farlo. Ho pubblicato delle istruzioni all'indirizzo <http://tinyurl.com/thinkpython2e>.

Il codice degli esempi di questo capitolo è scaricabile dal sito <http://thinkpython2.com/code/polygon.py>

4.1 Il modulo turtle

Per controllare se il modulo `turtle` è installato, aprite l'interprete Python e scrivete:

```
>>> import turtle
>>> bob = turtle.Turtle()
```

Eseguendo questo codice, dovrebbe comparire una nuova finestra con un cursore a forma di freccetta che rappresenta un'ideale tartaruga. Ora chiudete pure la finestra.

Create un file di nome `miopoligono.py` e scriveteci il seguente codice:

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

Il modulo `turtle` (con la 't' minuscola) contiene una funzione di nome `Turtle` (con la 'T' maiuscola) che crea un oggetto `Turtle` (una "tartaruga"); questo oggetto viene assegnato a una variabile di nome `bob`. Stampando `bob` viene visualizzato qualcosa di questo genere:

```
<turtle.Turtle object at 0xb7bfbf4c>
```

Ciò significa che bob fa riferimento ad un oggetto Turtle, come definito nel modulo turtle.

mainloop dice alla finestra di attendere che l'utente faccia qualcosa, sebbene in questo caso non ci sia molto da fare, se non chiudere la finestra.

Una volta creata una tartaruga, potete chiamare uno dei suoi **metodi** per spostarla in giro per la finestra. Un metodo è simile ad una funzione, ma usa una sintassi leggermente diversa. Ad esempio, per spostare la tartaruga in avanti:

```
bob.fd(100)
```

Il metodo, fd, è associato all'oggetto Turtle che abbiamo chiamato bob. Chiamare un metodo è come effettuare una richiesta: in questo caso state chiedendo a bob di muoversi in avanti [fd sta per *forward*, NdT]. L'argomento di fd è una distanza espressa in pixel, per cui l'effettivo spostamento dipenderà dalle caratteristiche del vostro schermo.

Altri metodi che potete chiamare su una tartaruga sono: bk per muoversi indietro (*backward*) e lt e rt per girare a sinistra (*left*) e a destra (*right*). Per questi ultimi due, l'argomento è un angolo espresso in gradi.

Inoltre, ogni tartaruga regge una penna, che può essere appoggiata o sollevata; se la penna è appoggiata, la tartaruga lascia un segno dove passa. I metodi 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 mainloop):

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

Avviando il programma, dovreste vedere bob muoversi verso destra e poi in alto, lasciandosi dietro due segmenti.

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

4.2 Ripetizione semplice

Probabilmente avete scritto qualcosa del genere:

```
bob.fd(100)
bob.lt(90)
```

```
bob.fd(100)
bob.lt(90)
```

```
bob.fd(100)
bob.lt(90)
```

```
bob.fd(100)
```

Si può ottenere lo stesso risultato in modo più conciso con un'istruzione `for`. Aggiungete questo esempio a `miopoligono.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 nella lettura solo dopo averlo fatto.

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

```
for i in range(4):  
    bob.fd(100)  
    bob.lt(90)
```

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 anche **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 dopo aver disegnato l'ultimo lato. Ciò comporta del 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 `turtle`. 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 continuate la lettura 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 pari a `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. 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 chiamando `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`.
5. Create una versione più generale 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` deve 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):
        t.fd(100)
        t.lt(90)
```

```
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 indica 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 `t.lt(90)` ha lo stesso effetto di `bob.lt(90)`. 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`:

```
alice = turtle.Turtle()
quadrato(alice)
```

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):
        t.fd(lunghezza)
        t.lt(90)
```

```
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 / n
    for i in range(n):
        t.fd(lunghezza)
        t.lt(angolo)
```

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

Questo esempio disegna un ettagono regolare con lati di lunghezza 70.

Se usate Python 2, il valore di `angolo` può risultare impreciso, per il fatto che la divisione di due interi dà come risultato un intero ("divisione intera", che vedremo meglio nel prossimo Capitolo). Una semplice soluzione è calcolare `angolo = 360.0 / n`. Dato che il numeratore ora è un numero floating-point, anche il risultato sarà un floating-point.

Quando in una chiamata di funzione avete più di qualche argomento numerico, è facile dimenticare a cosa si riferiscono o in quale ordine vanno disposti. In questi casi, è bene 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 dati 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:

```
import math

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 permette al chiamante di fare ciò che deve, senza avere a che fare con dettagli non necessari.

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) + 3
    lunghezza = circonferenza / n
    poligono(t, n, lunghezza)
```

Ora il numero di segmenti è un numero intero vicino a $\text{circonferenza}/3$, 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.

Aggiungere 3 a n garantisce che il poligono abbia come minimo 3 lati.

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 = angolo / n

    for i in range(n):
        t.fd(passo_lunghezza)
        t.lt(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):
        t.fd(lunghezza)
        t.lt(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 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 della procedura sono:

1. Iniziare scrivendo un piccolo programma senza definire funzioni.
2. Una volta ottenuto un programma funzionante, identificare una sua porzione che sia in sé coerente e autonoma, incapsularla 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 correggerlo).
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):  
        t.fd(lunghezza)  
        t.lt(angolo)
```

Per convenzione, la docstring è racchiusa tra triple virgolette, che le consentono di essere divisibile 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 darsi che la sua interfaccia sia migliorabile.

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` deve essere 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).

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.

Se le precondizioni sono soddisfatte e le postcondizioni no, l'errore sta nella funzione. E il fatto che le vostre pre- e postcondizioni siano chiare, è di aiuto nel debug.

4.11 Glossario

metodo: Una funzione associata ad un oggetto che viene chiamata utilizzando la notazione a punto.

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 si usa 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 all’inizio di 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://thinkpython2.com/code/polygon.py>.

1. 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 di stampa al codice.
2. 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 alcuni pixel 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://thinkpython2.com/code/flower.py>, richiede anche <http://thinkpython2.com/code/polygon.py>.

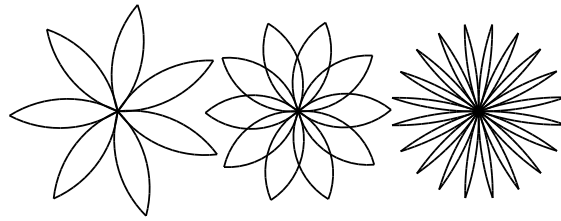


Figura 4.1: Fiori disegnati con turtle.

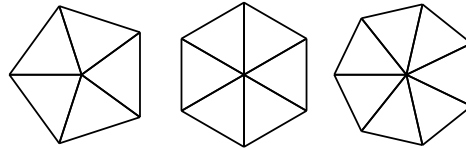


Figura 4.2: Torte disegnate con turtle.

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://thinkpython2.com/code/pie.py>.

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 alfabeto che possa essere disegnato con un numero minimo di elementi di base e poi scrivete delle funzioni che disegnano le lettere.

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://thinkpython2.com/code/typewriter.py> per aiutarvi a provare il vostro codice.

Soluzione: <http://thinkpython2.com/code/letters.py>, richiede anche <http://thinkpython2.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://thinkpython2.com/code/spiral.py>.

Capitolo 5

Istruzioni condizionali e ricorsione

L'argomento principale di questo capitolo è l'istruzione `if`, che permette di eseguire codice diverso a seconda dello stato del programma. Prima di tutto, vediamo però due nuovi operatori: divisione intera e modulo.

5.1 Divisione intera e modulo

L'operatore di **divisione intera**, `//`, divide due numeri e arrotonda il risultato all'intero inferiore. Ad esempio, supponiamo che la durata di un film sia di 105 minuti, e di volerla esprimere in ore. La normale divisione restituisce un numero decimale:

```
>>> minuti = 105
>>> minuti / 60
1.75
```

Ma di solito non si esprimono le ore con un numero decimale. La divisione intera dà invece come risultato il numero di ore, arrotondando per difetto:

```
>>> minuti = 105
>>> ore = minuti // 60
>>> ore
1
```

Per ottenere il resto, potete sottrarre dai minuti l'equivalente delle ore:

```
>>> resto = minuti - ore * 60
>>> resto
45
```

Un'alternativa è utilizzare l'**operatore modulo**, `%`, che restituisce il resto dell'operazione di divisione tra due numeri interi.

```
>>> resto = minuti % 60
>>> resto
45
```

L'operatore modulo è più utile di quel che sembra. Per esempio, permette di controllare se un numero intero 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.

Per chi usa Python 2, la divisione funziona in modo diverso. L'operatore di divisione intera non esiste, e quello di divisione, `/`, esegue una divisione intera se entrambi gli operandi sono interi, mentre il risultato è un decimale a virgola mobile se almeno uno degli operandi è un decimale.

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

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

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

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 sono vere *entrambe* le condizioni, cioè `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 (o per entrambi).

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 `True`, e lo zero è considerato `False`.

```
>>> 42 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, c'è molto spesso la necessità di valutare delle condizioni e di variare 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 è 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 e nulla più:

```
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 è pari")
else:
    print("x è 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 appropriato. 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 è minore di y")
elif x > y:
    print("x è maggiore di y")
else:
    print("x e y sono uguali")
```

`elif` è l'abbreviazione di *else if*, 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; il ramo corrispondente viene eseguito solo quando tutte le condizioni precedenti sono false.

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

Le condizioni vengono 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 ultime verranno trascurate.

5.7 Condizioni nidificate

Si può anche inserire un'istruzione condizionale nel corpo di un'altra istruzione condizionale. Possiamo scrivere l'esempio del paragrafo precedente anche in questo modo:

```
if x == y:
    print("x e y sono uguali")
else:
    if x < y:
        print("x è minore di y")
    else:
        print("x è 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 istruzioni condizionali nidificate:

```
if 0 < x:
    if x < 10:
        print("x è un numero positivo a una cifra.")
```

L'istruzione di stampa 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 è un numero positivo a una cifra.")
```

Per una condizione di questo tipo, Python consente anche un'opzione sintattica più concisa:

```
if 0 < x < 10:
    print("x è 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 che la esegue 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 di ritorno** `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 `contoallarovescia`: 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.

5.9 Diagrammi di stack delle funzioni ricorsive

Nel Paragrafo 3.9, 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 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.

Come esercizio, disegnate il diagramma di stack della funzione `stampa_n` chiamata con `s='Ciao'` e $n=2$. Poi, 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.

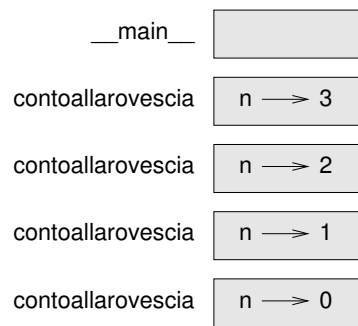


Figura 5.1: Diagramma di stack.

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!

Se vi imbattete accidentalmente in una ricorsione infinita, rivedete la vostra funzione per accertare che esista un caso base che non genera una chiamata ricorsiva. E se c'è un caso base, controllate che venga sicuramente raggiunto.

5.11 Input da tastiera

I programmi che abbiamo scritto finora non accettano inserimenti di dati da parte dell'operatore, e si limitano a eseguire sempre le stesse operazioni.

Python comprende una funzione predefinita chiamata `input` che sospende il programma 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 `input` restituisce ciò che l'operatore ha inserito sotto forma di stringa. In Python 2, la funzione si chiama `raw_input`.

```
>>> testo = input()
Cosa stai aspettando?
>>> 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 `input`:

```
>>> nome = input('Come...ti chiami?\n')
Come...ti chiami?
Artu', Re dei Bretoni!
>>> 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 è la velocità in volo di una rondine?\n"
>>> velocita = input(prompt)
Qual è la velocità in volo di una rondine?
42
>>> int(velocita)
42
```

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

```
>>> velocita = input(prompt)
Qual è la velocità 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

Quando si verifica un errore di sintassi o di runtime, il messaggio d'errore contiene molte informazioni, ma può essere sovrabbondante. 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, con qualche eccezione. Gli spaziatori possono essere insidiosi, perché spazi e tabulazioni non sono visibili e non siamo abituati a 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 erroneamente indentata di uno spazio, mentre dovrebbe stare al margine sinistro. 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 il vero errore 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_segnaile = 9
potenza_rumore = 10
rapporto = potenza_segnaile // potenza_rumore
decibel = 10 * math.log10(rapporto)
print(decibel)
```

Se avviate questo programma, 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 sta nella riga 4, perché calcola una divisione intera anziché una normale divisione.

Prendetevi la briga di leggere attentamente i messaggi di errore, ma non date per scontato che tutto quello che dicono sia esatto.

5.13 Glossario

divisione intera: Operatore, avente simbolo `//`, che divide due numeri e arrotonda il risultato all'intero inferiore (ovvero, verso l'infinito negativo).

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.

istruzione di ritorno: Un'istruzione che termina immediatamente una funzione e ritorna al chiamante.

ricorsione: Procedura che chiama la 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 5.1. Il modulo `time` contiene una funzione, anch'essa di nome `time`, che restituisce l'attuale GMT (Tempo Medio di Greenwich) riferito ad un "tempo zero", che è un momento arbitrario usato come punto di riferimento. Nei sistemi UNIX, questo "tempo zero" è il 1 gennaio 1970.

```
>>> import time
>>> time.time()
1437746094.5735958
```

Realizzate uno script che acquisisca il tempo attuale e lo converta in un tempo in ore, minuti e secondi, più i giorni trascorsi dal "tempo zero".

Esercizio 5.2. 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 fosse

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

il programma dovrebbe 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 5.3. 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, non riuscirete 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.*

Esercizio 5.4. *Qual è l’output del seguente programma? Disegnate un diagramma di stack che illustri lo stato del programma nel momento in cui stampa il risultato.*

```
def ricorsione(n, s):
    if n == 0:
        print(s)
    else:
        ricorsione(n-1, n+s)

ricorsione(3, 0)
```

1. *Cosa succede se chiamate la funzione in questo modo: `ricorsione(-1, 0)`?*
2. *Scrivete una stringa di documentazione che spieghi tutto quello che serve per usare questa funzione (e niente di più).*

Gli esercizi seguenti utilizzano il modulo `turtle`, descritto nel Capitolo 4:

Esercizio 5.5. *Leggete la seguente funzione e cercate di capire cosa fa (vedere gli esempi nel Capitolo 4). Quindi eseguirla per controllare se avevate indovinato.*

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

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 $x/3$.*

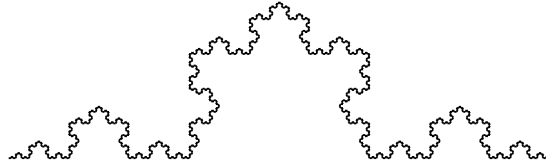


Figura 5.2: Una curva di Koch.

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://thinkpython2.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

Molte tra le funzioni di Python che abbiamo usato, come quelle matematiche, producono dei valori di ritorno. Ma quelle che abbiamo scritto noi finora sono tutte “vuote”: hanno un qualche effetto, come visualizzare un testo o muovere tartarughe, ma non hanno un valore di ritorno. In questo capitolo vedremo come si scrivono le funzioni che chiameremo “produttive”.

6.1 Valori di ritorno

La chiamata di una funzione genera un nuovo valore, che di solito viene associato ad una variabile o si usa come parte di un’espressione.

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

Le funzioni che abbiamo scritto finora sono “vuote”. Detto in modo semplicistico, non hanno valore di ritorno; ma a voler essere precisi, il loro valore di ritorno è `None`.

In questo capitolo scriveremo finalmente delle funzioni che restituiscono un valore e che chiameremo funzioni “produttive”. Facciamo un primo esempio con `area`, che calcola l’area di un cerchio di dato raggio:

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

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 `a` può rendere il programma più leggibile e semplificarne 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 detto **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 ha un difetto, 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 sarà `None`, che non è di certo il valore assoluto di 0.

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

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

Per esercitarvi, scrivete una funzione di nome `compara` che prenda due valori, x e y , e restituisca 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, espressa con un valore decimale.

Si può subito 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: 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 due 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 è ", dx)  
    print("dy è ", dy)  
    return 0.0
```

Se la funzione è giusta, usando i valori di prima dovrebbe mostrare *dx* è 3 e *dy* è 4. Se i risultati coincidono, siamo sicuri che la funzione riceve correttamente i parametri ed elabora altrettanto correttamente i primi calcoli. Altrimenti, dovremo comunque 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 è: ", 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 di stampa che avevamo inserito erano utili per il debug, ma una volta verificato che tutto funziona vanno rimosse. Pezzi di codice temporaneo 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 si dovrebbe mai aggiungere più di qualche riga di codice alla volta. Con l'esperienza, potrete scrivere e fare il debug di blocchi 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 delle variabili 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.

Come esercizio, 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. 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 è divisibile per y")
```

Potreste pensare di scrivere in questo modo:

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

ma il confronto supplementare è superfluo.

Scrivete ora, per esercizio, 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 sottinsieme 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 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, 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
```

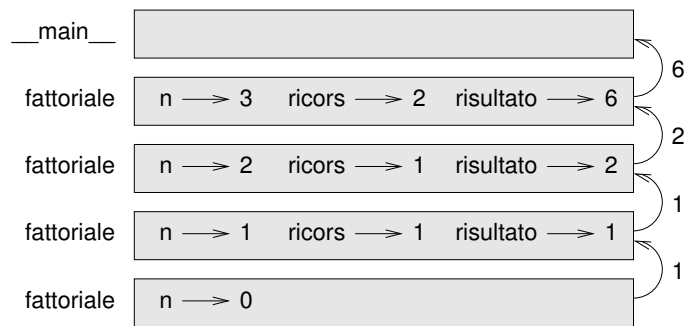


Figura 6.1: Diagramma di stack .

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

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

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

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

Dato che 0 è uguale a 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 funzionino 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 chiamato ricorsive: invece di seguire il flusso di esecuzione, potete partire dal presupposto che la chiamata ricorsiva funzioni (restituendo il risultato corretto), per poi chiedervi: “Supponendo che io trovi il fattoriale di $n - 1$, posso calcolare il fattoriale di n ?”. 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. Come mai? La funzione ha 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 è definito solo per numeri interi.")
        return None
    elif n < 0:
        print("Il fattoriale non è 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, 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:

```
>>> print(fattoriale('alfredo'))
Il fattoriale è definito solo per numeri interi.
None
>>> print(fattoriale(-2))
Il fattoriale non è 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 11.4 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 preconditione.
- 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 di stampa all'inizio della funzione per visualizzare i valori dei parametri (e magari i loro tipi). O potete scrivere del codice che controlla esplicitamente le preconditioni.

Se i parametri sembrano corretti, aggiungete un'istruzione di stampa prima di ogni istruzione `return` e visualizzate il valore di ritorno. Se possibile, controllate i risultati calcolandovi a parte. Cercate di chiamare la funzione fornendole dei valori che permettono un agevole controllo 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(4)`:

```

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

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 in compenso 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`.

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 6.1. *Disegnate un diagramma di stack del seguente programma. Che cosa visualizza?*

```
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):
    totale = x + y + z
    quadrato = b(totale)**2
    return quadrato

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

Esercizio 6.2. *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://thinkpython2.com/code/ackermann.py>.

Esercizio 6.3. 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://thinkpython2.com/code/palindrome_soln.py.

Esercizio 6.4. Un numero, a , è una potenza di b se è divisibile per b e a/b è a sua volta 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 6.5. 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

In questo capitolo parleremo dell'iterazione, che è la capacità di eseguire ripetutamente uno stesso blocco di istruzioni. Abbiamo visto una sorta di iterazione nel Paragrafo 5.8, usando la ricorsione. Ne abbiamo visto un tipo nel Paragrafo 4.2, dove abbiamo utilizzato un ciclo `for`. Qui ne vedremo un tipo ulteriore, che usa l'istruzione `while`. Ma prima, qualche altro dettaglio sull'assegnazione delle variabili.

7.1 Riassegnazione

Vi sarete forse 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).

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

La prima volta che visualizziamo `x`, il suo valore è 5; la seconda volta è 7.

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

Ora, è bene chiarire un punto che è frequente motivo di confusione. Dato che Python usa `(=)` per le assegnazioni, potreste interpretare l'istruzione `a = b` come un'espressione matematica di uguaglianza, cioè una proposizione per cui `a` e `b` sono uguali. Questo non è corretto.

In primo luogo, l'equivalenza è una relazione 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:

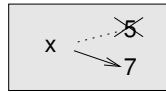


Figura 7.1: Diagramma di stato.

```
>>> a = 5
>>> b = a    # a e b ora sono uguali
>>> a = 3    # a e b non sono più uguali
>>> b
5
```

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 riassegnazioni di variabile 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.

7.2 Aggiornare le variabili

Una delle forme più comuni di riassegnazione 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. Nella programmazione, la ripetizione è chiamata anche **iterazione**.

Abbiamo già visto due funzioni, `contoallarovescia` e `stampa_n`, che iterano usando la ricorsione. 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 `contoallaroveschia` che usa l'istruzione `while`:

```
def contoallaroveschia(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. Determina se la condizione 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: se `n` è zero o negativo, il ciclo non viene mai eseguito. Altrimenti, `n` diventa via via più piccolo ad ogni ripetizione del ciclo stesso, fino a diventare, prima o poi, zero.

In altri cicli, può non essere così facile stabilirlo. Per esempio:

```
def sequenza(n):
    while n != 1:
        print(n)
        if n % 2 == 0:          # n è pari
            n = n / 2
        else:                  # n è 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, i valori risultanti di `n` saranno 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, n sarà per forza un numero pari ad 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.)

Come esercizio, 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 interrompere il ciclo e saltarne fuori.

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

```
while True:
    riga = 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
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

Il risultato 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
>>> y
2.00641025641
```

Dopo qualche ulteriore passaggio, la stima diventa quasi esatta:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> 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
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> 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.

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).

Per capire cosa sia un algoritmo, può essere utile iniziare a vedere cosa non è un algoritmo. Quando a scuola vi insegnarono a fare le moltiplicazioni dei numeri a una cifra, probabilmente avevate 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 avevate imparato 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.

L'esecuzione di un algoritmo, in sé, è una cosa noiosa e ripetitiva. 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 di stampa (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), dovrete 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

riassegnazione: Assegnazione di un nuovo valore ad una variabile che esiste già.

aggiornamento: Riassegnazione 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 in cui la condizione che ne determina la fine non è mai soddisfatta.

algoritmo: Una procedura generica per risolvere una categoria di problemi.

7.9 Esercizi

Esercizio 7.1. Copiate il ciclo del Paragrafo 7.5 e incapsulatelo in una funzione di nome `mia_radq` che prenda `a` come parametro, scelga un valore appropriato di `x`, e restituisca una stima del valore della radice quadrata di `a`.

Quale verifica, scrivete una funzione di nome `test_radq` che stampi una tabella come questa:

a	<code>mia_radq(a)</code>	<code>math.sqrt(a)</code>	diff
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 `math.sqrt`; la terza è la radice quadrata calcolata con `math.sqrt`; la quarta è il valore assoluto della differenza tra le due stime.

Esercizio 7.2. 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)')
<class '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 7.3. 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://thinkpython2.com/code/pi.py>.

Capitolo 8

Stringhe

Le stringhe non sono valori come gli interi, i float e i booleani. Una stringa è una **sequenza**, vale a dire un insieme ordinato di valori di altra natura. In questo capitolo vedrete come si accede ai caratteri che compongono una stringa e imparerete alcuni metodi che le stringhe espongono.

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 alla variabile `lettera`.

L'espressione all'interno delle parentesi quadre è chiamato **indice**. L'indice è un numero intero che indica (di qui il nome) il carattere della sequenza che desiderate estrarre.

Ma il risultato potrebbe lasciarvi perplessi:

```
>>> 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]
>>> 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:

```
>>> i = 1
>>> frutto[i]
'a'
>>> frutto[i+1]
'n'
```

Tuttavia, il valore risultante deve essere un intero. Altrimenti succede questo:

```
>>> lettera = frutto[1.5]
TypeError: string indices must be integers
```

8.2 len

`len` è una funzione predefinita che restituisce il numero di caratteri contenuti in una stringa:

```
>>> frutto = 'banana'
>>> len(frutto)
6
```

Per estrarre l'ultimo carattere di una stringa, si potrebbe 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]
>>> ultimo
'a'
```

Oppure, 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 a cui si accede è quello con indice `len(frutto)-1`, che è l'ultimo carattere della stringa.

Come esercizio, 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 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
Ouack
Pack
Quack
```

È evidente che non è del tutto giusto, dato che "Ouack" e "Quack" sono scritti in modo errato.

Provate a modificare 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'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

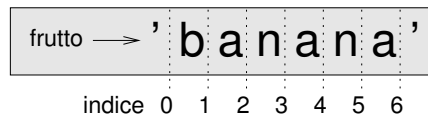


Figura 8.1: Indici di slicing.

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.

Proseguendo con l'esempio, data una stringa di nome `frutto`, secondo voi che cosa significa `frutto[:]`? Provate a vedere.

8.5 Le stringhe sono immutabili

Potrete 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 (Paragrafo 10.10) puntualizzeremo meglio questa definizione.

La ragione dell'errore è che le stringhe sono **immutabili**, in altre parole, non è consentito 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:]
>>> 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'inverso 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 si trova ciò che si sta cercando—è chiamato **ricerca**.

Come esercizio, 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.

Come esercizio, 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. Quindi, 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

Le stringhe espongono dei metodi che permettono di effettuare molte utili operazioni. Un **metodo** è simile a una funzione—riceve argomenti e restituisce un valore—ma la sintassi è diversa. Prendiamo ad esempio il metodo `upper`, che prende una stringa e crea una nuova stringa di tutte lettere maiuscole.

Al posto della sintassi delle funzioni, `upper(parola)`, si usa la sintassi dei metodi, `parola.upper()`.

```
>>> parola = 'banana'
>>> nuova_parola = parola.upper()
>>> nuova_parola
BANANA
```

Questa forma di notazione a punto, in inglese *dot notation*, specifica il nome del metodo, `upper`, preceduto dal 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')
>>> 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
```

Di default, `find` parte dall'inizio della stringa, ma può ricevere come secondo argomento l'indice da cui partire:

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

Questo è un esempio di **argomento opzionale**; `find` può anche avere un terzo argomento opzionale, 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, in quanto 2 si considera escluso. Questo comportamento rende `find` coerente con l'operatore di slicing.

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 dovrebbe 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 abbiano 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
```

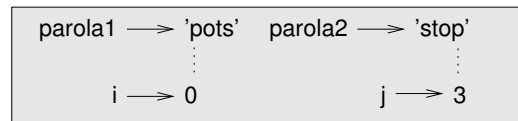


Figura 8.2: Diagramma di Stato.

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:

```
>>> 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.

Mi sono preso la libertà di disporre le variabili nel frame e di aggiungere delle linee tratteggiate per evidenziare che i valori di `i` e `j` indicano i caratteri in `parola1` e `parola2`.

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: Una raccolta ordinata di valori, in cui ciascun valore 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. In Python gli indici partono da 0.

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 cambiati.

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.

invocazione: Istruzione che chiama un metodo.

argomento opzionale: Un argomento di una funzione o di un metodo che non è obbligatorio.

8.13 Esercizi

Esercizio 8.1. Leggete la documentazione dei metodi delle stringhe sul sito <http://docs.python.org/3/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.

Esercizio 8.2. Esiste un metodo delle stringhe di nome `count` che è simile alla funzione del Paragrafo 8.7. Leggete la documentazione del metodo e scrivete un'invocazione che conti il numero di `a` in `'banana'`.

Esercizio 8.3. 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 slicing `[::-1]` genera una stringa scritta al contrario.

Usate questo costrutto per scrivere una variante di una sola riga della funzione `palindromo` dell'Esercizio 6.3.

Esercizio 8.4. 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 8.5. Un cifrario di Cesare è un metodo di criptazione debole che consiste nel “ruotare” ogni lettera di una parola di un dato numero di 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’.

Per ruotare una parola, si ruota ciascuna delle sue lettere dello stesso numero di posti prefissato. Per esempio, “cheer” ruotata di 7 dà “jolly” e “melon” ruotata di -10 dà “cubed”. Nel film 2001: Odissea nello Spazio, il computer di bordo si chiama HAL, che non è altro che IBM ruotato di -1.

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.

Potete usare le funzioni predefinite `ord`, che converte un carattere in un codice numerico, e `chr`, che converte i codici numerici in caratteri. Le lettere sono codificate con il loro numero di ordine alfabetico, per esempio:

```
>>> ord('c') - ord('a')
2
```

Dato che ‘c’ è la “2-esima” lettera dell’alfabeto. Ma attenzione: i codici numerici delle lettere maiuscole sono diversi.

Su Internet, talvolta, vengono codificate in ROT13 (un cifrario di Cesare con rotazione 13) delle barzellette potenzialmente offensive. Se non siete suscettibili, cercatene qualcuna e decodificatela. Soluzione: <http://thinkpython2.com/code/rotate.py>.

Capitolo 9

Esercitazione: Giochi con le parole

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

9.1 Leggere elenchi di parole

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

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

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

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

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

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

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

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

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

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

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

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

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

9.2 Esercizi

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

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

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

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

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

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

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

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

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

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

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

9.3 Ricerca

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.4 Cicli con gli indici

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

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

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

Un'alternativa è usare la ricorsione:

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

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

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

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

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

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

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

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

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

    return True
```

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

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

Usando `al_contrario` del Paragrafo 8.11.

9.5 Debug

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

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

Per ciascun caso ci sono alcuni sottocasi meno ovvi. Tra le parole che contengono "e", 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 test di un programma può essere usato per dimostrare la presenza di bug, ma mai per dimostrarne l'assenza!

— Edsger W. Dijkstra

9.6 Glossario

oggetto file: Un valore che rappresenta un file aperto.

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

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

9.7 Esercizi

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

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

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

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

“L’altro giorno stavo guidando in autostrada e guardai il mio contachilometri. È a sei cifre, come la maggior parte dei contachilometri, e mostra solo chilometri interi. Se la mia macchina, per esempio, avesse 300.000 km, vedrei 3-0-0-0-0-0.”

“Quello che vidi quel giorno era interessante. Notai che le ultime 4 cifre erano 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 palindrome. E tenetevi forte: un altro chilometro dopo tutte e 6 erano palindrome!”

“La domanda è: quanto segnava il contachilometri la prima volta che guardai?”

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

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

“Di recente ho fatto visita a mia madre, e ci siamo accorti che le due cifre che compongono la mia età, invertite, formano la sua. Per esempio, se lei avesse 73 anni, io ne avrei 37. Ci siamo domandati quanto spesso succedesse questo negli anni, ma poi abbiamo divagato su altri discorsi senza darci una risposta.”

“Tornato a casa, ho calcolato che le cifre delle nostre età sono state sinora invertibili per sei volte. Ho calcolato anche che se fossimo fortunati succederebbe ancora tra pochi anni, e se fossimo veramente fortunati succederebbe un'altra volta ancora. In altre parole, potrebbe succedere per 8 volte in tutto. La domanda è: quanti anni ho io in questo momento?”

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

Soluzione: <http://thinkpython2.com/code/cartalk3.py>.

Capitolo 10

Liste

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

10.1 Una lista è una sequenza

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

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

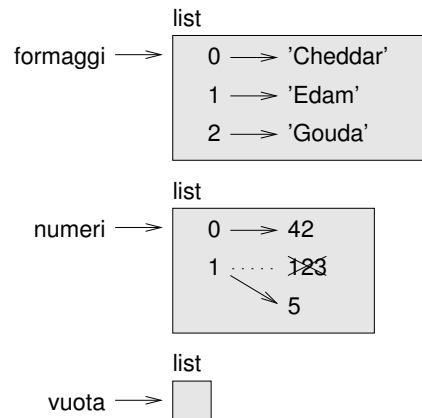


Figura 10.1: Diagramma di stato

10.2 Le liste sono mutabili

La sintassi per 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!):

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

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

La Figura 10.1 mostra il diagramma di `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 predefinite `range` e `len`:

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

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

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

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

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

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

10.4 Operazioni sulle liste

L'operatore `+` concatena delle liste:

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

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

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

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

10.5 Slicing delle liste

Anche l'operazione di slicing funziona sulle liste:

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

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

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

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

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

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

10.6 Metodi delle liste

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

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

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

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

Questo esempio lascia immutata la lista `t2`.

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

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

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

10.7 Mappare, filtrare e ridurre

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

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

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

```
totale += x
```

è equivalente a

```
totale = totale + x
```

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

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

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

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

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

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

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

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

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

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

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

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

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

10.8 Cancellare elementi

Ci sono 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)
>>> t
['a', 'c']
>>> x
'b'
```

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

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

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

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

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

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

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

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

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

10.9 Liste e stringhe

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

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

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

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

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

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

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

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

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

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

10.10 Oggetti e valori

Se eseguiamo queste istruzioni di assegnazione:

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

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

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

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

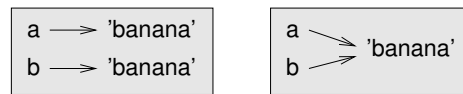


Figura 10.2: Diagramma di stato.

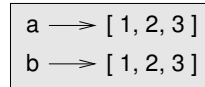


Figura 10.3: Diagramma di stato.

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

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

Ma se create due liste, ottenete due oggetti distinti:

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

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

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

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

10.11 Alias

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

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

Il diagramma di stato è quello in Figura 10.4.

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

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

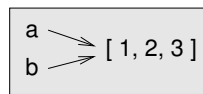


Figura 10.4: Diagramma di stato.

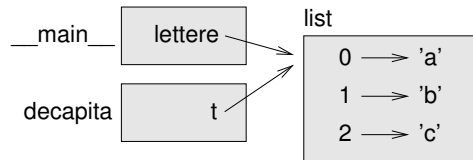


Figura 10.5: Diagramma di stack.

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

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

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

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

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

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

10.12 Liste come argomenti

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

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

Vediamo come si usa:

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

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

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

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

Ecco un esempio che usa `append`:

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

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

Un esempio di utilizzo dell'operatore `+`:

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

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

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

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

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

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

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

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

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

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

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

10.13 Debug

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

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

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

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

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

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

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

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

2. Scegliete un costrutto e usate sempre quello.

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

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

```
t.append(x)
```

```
t = t + [x]
```

Mentre queste sono sbagliate:

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

```
t = t.append(x)         # SBAGLIATO!
```

```
t + [x]                 # SBAGLIATO!
```

```
t = t + x               # SBAGLIATO!
```

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

3. Fate copie per evitare gli alias.

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

```
>>> t = [3, 1, 2]
```

```
>>> t2 = t[:]
```

```
>>> t2.sort()
```

```
>>> t
```

```
[3, 1, 2]
```

```
>>> t2
```

```
[1, 2, 3]
```

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

```
>>> t2 = sorted(t)
```

```
>>> t
```

```
[3, 1, 2]
```

```
>>> t2
```

```
[1, 2, 3]
```

10.14 Glossario

lista: Una sequenza di valori.

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

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

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

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

riduzione: Schema di 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Potete scaricare la mia soluzione da <http://thinkpython2.com/code/birthday.py>.

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

Soluzione: <http://thinkpython2.com/code/wordlist.py>.

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

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

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

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

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

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

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

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

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

Capitolo 11

Dizionari

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

11.1 Un dizionario è una mappatura

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

11.2 Il dizionario come raccolta di contatori

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

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

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

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

Ecco come potrebbe essere scritto il codice:

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

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

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

Vediamo come funziona:

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

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

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

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

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

11.3 Cicli e dizionari

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

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

Ecco come risulta l'output:

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

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

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

11.4 Lookup inverso

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

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

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

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

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

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

Ecco un esempio di lookup inverso riuscito:

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

E di uno fallito:

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

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

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

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

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

11.5 Dizionari e liste

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

Ecco una funzione che inverte un dizionario:

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

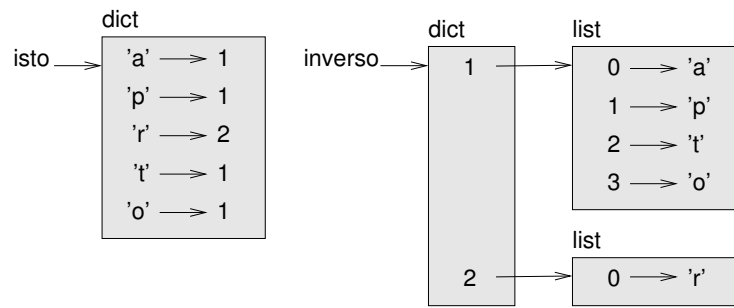


Figura 11.1: Diagramma di stato.

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

Ecco un esempio:

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

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

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

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

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

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

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

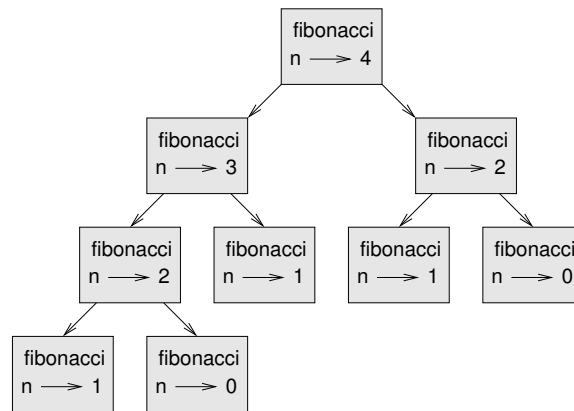


Figura 11.2: Grafico di chiamata.

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

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

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

11.6 Memoizzazione

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

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

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

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

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

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

```
def fibonacci(n):
```

```

    if n in memo:
        return memo[n]

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

```

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

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

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

11.7 Variabili globali

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

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

```
verbose = True
```

```

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

```

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

```
stata_chiamata = False
```

```

def esempio2():
    stata_chiamata = True          # SBAGLIATO

```

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

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

```
stata_chiamata = False
```

```

def esempio2():
    global stata_chiamata
    stata_chiamata = True

```

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

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

```
conta = 0
```

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

Se lo eseguite, ottenete:

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

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

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

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

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

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

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

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

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

11.8 Debug

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

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

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

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

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

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

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

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

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

11.9 Glossario

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

dizionario: Una mappatura da chiavi nei loro valori corrispondenti.

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

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

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

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

implementazione: Un modo per effettuare un'elaborazione.

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

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

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

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

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

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

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

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

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

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

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

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

11.10 Esercizi

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

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

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

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

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

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

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

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

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

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

“Facciamo un esempio che non funziona del tutto. Prendiamo la parola 'wrack'; togliendo la prima lettera resta 'rack', che è un'altra parola ma è un perfetto omofono. Se però rimettete la prima lettera e togliete la seconda, ottenete 'wack' che pure esiste ma non è un omofono delle altre due parole.”

“Esiste comunque almeno una parola, che Dan e noi conosciamo, che dà due parole omofone di quattro lettere, sia che togliate la prima oppure la seconda lettera.”

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

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

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

Capitolo 12

Tuple

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

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)  
<class 'tuple'>
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

12.2 Assegnazione di tupla

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

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

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

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

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

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

12.3 Tuple come valori di ritorno

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

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

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

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

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

Ecco un esempio di funzione che restituisce una tupla:

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

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

12.4 Tuple di argomenti a lunghezza variabile

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

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

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

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

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

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

Ma se spacchettate la tupla, funziona:

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

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

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

Ma con `sum` non funziona.

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

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

12.5 Liste e tuple

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

Questo esempio abbina una stringa e una lista:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

12.6 Dizionari e tuple

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

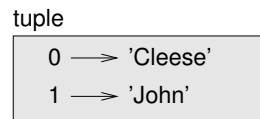


Figura 12.1: Diagramma di stato.

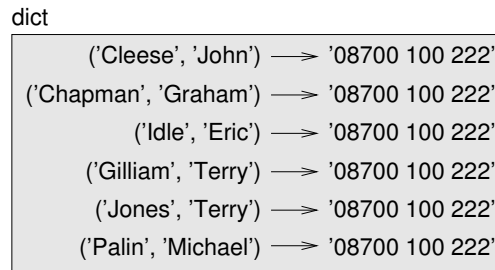


Figura 12.2: Diagramma di stato.

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

12.7 Sequenze di sequenze

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

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

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

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

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

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

12.8 Debug

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

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

Questo è il risultato per una lista semplice:

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

Un programma più aggraziato avrebbe scritto “list of 3 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]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

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

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

Ecco una lista di tuple:

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

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

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

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

12.9 Glossario

tupla: Una sequenza di elementi immutabile.

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

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

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

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

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

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

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

12.10 Esercizi

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

Esercizio 12.2. *Ancora anagrammi!*

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

Un esempio di come si può presentare il risultato:

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

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

2. Modificate il programma in modo che stampi la lista di anagrammi più lunga per prima, seguita dalla seconda più lunga, e così via.
3. Nel gioco da tavolo Scarabeo, fate un "en-plein" quando giocate tutte le sette lettere sul vostro leggio formando, insieme a una lettera sul tavolo, una parola di otto lettere. Con quale gruppo di 8 lettere si può fare un "en-plein" con maggior probabilità? Suggerimento: il gruppo dà sette combinazioni.

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

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

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

Qual è la più lunga parola inglese che rimane una parola valida se le togliete una lettera alla volta? Le lettere possono essere rimosse sia agli estremi o in mezzo, ma senza spostare le lettere rimanenti. Ogni volta che togliete una lettera, ottenete un'altra parola inglese. Se andate avanti, ottenete un'altra parola. Ora, voglio sapere qual è la parola più lunga possibile e quante lettere ha.

Vi faccio un piccolo esempio: Sprite. Partite da sprite, togliete una lettera, una interna, come la r e resta la parola spite, poi togliete la e finale e avete spit, togliamo la s e resta pit, poi it, infine I.

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

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

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

Soluzione: <http://thinkpython2.com/code/reducible.py>.

Capitolo 13

Esercitazione: Scelta della struttura di dati

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

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

13.1 Analisi di frequenza delle parole

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

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

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

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

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

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

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

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

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

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

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

13.2 Numeri casuali

A parità di dati di partenza, i programmi, per la maggior parte, 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 11.2 e restituisca un valore casuale dall'istogramma, scelto in modo che la probabilità sia proporzionale alla frequenza. Per esempio, dato questo istogramma:*

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

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

13.3 Istogramma di parole

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

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

```
import string

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

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

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

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

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

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

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

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

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

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

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

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

Ed ecco del codice per stampare i risultati:

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

E i relativi risultati:

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

13.4 Parole più comuni

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

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

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

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

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

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

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

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



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

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

13.5 Parametri opzionali

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

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

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

Se passate un solo argomento:

```
stampa_piu_comuni(isto)
```

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

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

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

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

13.6 Sottrazione di dizionari

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

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

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

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

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

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

Ecco alcuni risultati per *Emma*:

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

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

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

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

13.7 Parole a caso

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

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

    return random.choice(t)
```

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

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

Un’alternativa è:

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

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

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

13.8 Analisi di Markov

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

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

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

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

Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D'you see?

But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?

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

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

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

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

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

Esercizio 13.8. *Analisi di Markov:*

1. Scrivete un programma che legga un testo da un file ed esegua l'analisi di Markov. Il risultato dovrebbe essere un dizionario che fa corrispondere i prefissi a una raccolta di possibili suffissi. La raccolta può essere una lista, tupla o dizionario: a voi valutare la scelta più appropriata. Potete testare il vostro programma con una lunghezza del prefisso di due parole, ma 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 combinate testi presi da due o più libri, il testo generato mescolerà il vocabolario e le frasi dei sorgenti in modi interessanti.

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

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

13.9 Strutture di dati

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

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

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

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

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

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

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

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

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

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

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

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

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

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

13.10 Debug

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

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

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

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

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

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

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

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

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

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

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

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

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

13.11 Glossario

deterministico: Qualità di un programma di fare le stesse cose ogni 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.

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

13.12 Esercizi

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

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

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

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

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

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

Scrivete un programma che legga un testo da un file, conti le 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://thinkpython2.com/code/zipf.py>. Per avviare la mia risoluzione serve il modulo di plotting `matplotlib`. Se avete installato Anaconda, avete già `matplotlib`; altrimenti potrebbe essere necessario installarlo.

Capitolo 14

File

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

14.1 Persistenza

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

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

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

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

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

14.2 Lettura e scrittura

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

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

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

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

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

Il metodo `write` inserisce i dati nel file.

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

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

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

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

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

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

14.3 L'operatore di formato

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14.4 Nomi di file e percorsi

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14.5 Gestire le eccezioni

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

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

Se non avete il permesso di accedere al file:

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

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

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

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

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

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

Python comincia con l’eseguire la clausola `try`. Se tutto va bene, 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.

14.6 Database

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

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

Un database si apre in modo simile agli altri file:

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

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

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

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

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

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

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

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

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

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

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

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

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

14.7 Pickling

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

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

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

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03q\x00(K\x01K\x02K\x03e.'
```

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

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

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

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

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

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

14.8 Pipe

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

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

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

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

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

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

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

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

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

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

Potete allora usare 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
```

14.9 Scrivere moduli

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

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

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

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

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

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

Ora avete un oggetto modulo `wc`:

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

L'oggetto modulo fornisce `contarighe`:

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

Ecco come scrivere moduli in Python.

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

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

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

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

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

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

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

14.10 Debug

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


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

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

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

Questa funzione può quindi aiutare nel debug.

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

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

14.11 Glossario

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

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

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

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

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

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

percorso: Stringa che localizza un file.

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

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

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

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

oggetto bytes: Un oggetto simile ad una stringa.

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

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

14.12 Esercizi

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

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

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

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

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

1. Scrivete un programma che cerchi in una cartella e, ricorsivamente, nelle sue sottocartelle, e restituisca un elenco dei percorsi completi di tutti i file con una stessa estensione (come `.mp3`). Suggestimento: `os.path` contiene alcune funzioni utili per trattare nomi di file e percorsi.
2. Per riconoscere i duplicati, potete usare `md5sum` per calcolare la "checksum" di ogni file. Se due file hanno la stessa checksum, significa che con ogni probabilità hanno lo stesso contenuto.
3. Per effettuare un doppio controllo, usate il comando Unix `diff`.

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

Capitolo 15

Classi e oggetti

A questo punto, sapete come usare le funzioni per organizzare il codice, e i tipi predefiniti per organizzare i dati. Il passo successivo è imparare la programmazione orientata agli oggetti, che usa tipi personalizzati in modo da organizzare sia il codice che i dati. La programmazione orientata agli oggetti è un argomento vasto, per addentrarsi nel quale occorrono alcuni capitoli.

Il codice degli esempi di questo capitolo è scaricabile dal sito <http://thinkpython2.com/code/Point1.py>; le soluzioni degli esercizi da http://thinkpython2.com/code/Point1_soln.py.

15.1 Tipi personalizzati

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 più complessa delle altre, ma ha dei vantaggi che saranno presto chiariti.

Un tipo personalizzato, definito dal programmatore, è chiamato anche **classe**. Una definizione di classe ha questa sintassi:

```
class Punto:
    """Rappresenta un punto in un piano."""
```

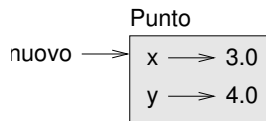


Figura 15.1: Diagramma di oggetto

L'intestazione indica che la nuova classe si chiama Punto. Il corpo è una stringa di documentazione che spiega cosa fa la classe. Al suo interno si possono poi definire metodi e variabili, ma ci arriveremo tra poco.

La definizione di una classe di nome Punto crea un **oggetto classe**.

```
>>> 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()
>>> nuovo
<__main__.Punto object 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).

Ogni oggetto è un'istanza di una qualche classe, per cui i termini “oggetto” ed “istanza” sono equivalenti. In questa sede, utilizzerò “istanza” per indicare che sto parlando di un tipo personalizzato.

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:

```
>>> nuovo.y
4.0
>>> x = nuovo.x
>>> x
3.0
```

L'espressione `nuovo.x` significa: "Vai all'oggetto a cui `nuovo` fa riferimento e prendi il valore di `x`". In questo esempio, 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)
>>> 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.

Per esercizio, 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:

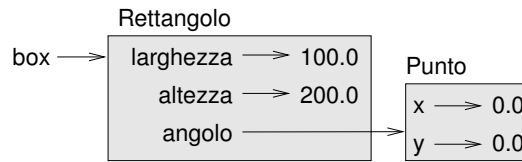


Figura 15.2: Diagramma di oggetto.

```

class Rettangolo:
    """Rappresenta un rettangolo.

    attributi: larghezza, altezza, angolo.
    """

```

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
    p.y = rett.angolo.y + rett.altezza/2
    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, 100)

```

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:

```
>>> box.larghezza, box.altezza
(150.0, 300.0)
>>> accresci_rettangolo(box, 50, 100)
>>> box.larghezza, box.altezza
(200.0, 400.0)
```

Dentro la funzione, `rett` è un alias di `box`, pertanto quando la funzione modifica `rett`, anche `box` cambia.

Come esercizio, 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, 4)
>>> stampa_punto(p2)
(3, 4)
>>> p1 is p2
```

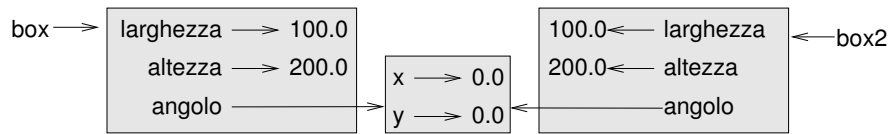


Figura 15.3: Diagramma di oggetto.

```
False
>>> p1 == p2
False
```

L'operatore `is` indica che `p1` e `p2` non sono lo stesso oggetto, come volevasi dimostrare. Ma forse vi aspettavate che l'operatore `==` desse `True`, perché i due punti contengono gli stessi dati. Invece, dovete sapere 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 perché, per i tipi personalizzati, Python non sa cosa debba essere considerato equivalente. O almeno, non lo sa ancora.

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 il comportamento ideale. Nel nostro esempio, invocare `accresci Rettangolo` su uno dei Rettangoli non influenzerebbe l'altro, ma invocare `sposta Rettangolo` su uno dei due, influenzerebbe entrambi! Tutto ciò genera confusione ed è foriero di errori.

Fortunatamente, il modulo `copy` è dotato anche di 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.

Come esercizio, 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()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Punto instance has no attribute 'z'
```

Se non siete sicuri di che tipo sia un oggetto, potete chiederlo:

```
>>> type(p)
<class '__main__.Punto'>
```

Si può usare anche `isinstance` per controllare se un oggetto è un'istanza di una classe:

```
>>> isinstance(p, Punto)
True
```

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.

Si può anche usare un'istruzione `try` per controllare che l'oggetto contenga gli attributi che servono:

```
try:
    x = p.x
except AttributeError:
    x = 0
```

Questa tecnica può facilitare la scrittura di funzioni che trattano tipi di dati differenti; vedremo altro su questo tema nel Paragrafo 17.9.

15.8 Glossario

classe: Tipo di dato personalizzato definito dal programmatore. Una definizione di classe crea un nuovo oggetto classe.

oggetto classe: Oggetto che contiene le informazioni su un tipo personalizzato e che può essere usato per creare istanze del tipo.

istanza: Oggetto che appartiene ad una classe.

istanziare: Creare un nuovo oggetto.

attributo: Uno dei valori associati ad un oggetto, dotato di un nome.

oggetto contenuto (embedded): 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 15.1. *Scrivete una definizione di classe di nome `Cerchio`, avente gli attributi `centro` e `raggio`, dove `centro` è un oggetto `Punto` e `raggio` è un numero.*

Istanziare un oggetto `Cerchio` che rappresenti un cerchio con il centro nel punto (150,100) e di raggio 75.

Scrivete una funzione di nome `punto_nel_cerchio`, che prenda un `Cerchio` e un `Punto` e restituisca `True` se il punto giace dentro il cerchio, circonferenza compresa.

Scrivete una funzione di nome `rett_nel_cerchio`, che prenda un `Cerchio` e un `Rettangolo` e restituisca `True` se il rettangolo giace interamente all'interno del cerchio, circonferenza compresa.

Scrivete una funzione di nome `rett_cerchio_sovrapp`, che prenda un `Cerchio` e un `Rettangolo` e restituisca `True` se almeno uno degli angoli del Rettangolo ricade all'interno del cerchio. Oppure, più difficile, se una qualunque porzione del Rettangolo ricade all'interno del cerchio.

Soluzione: <http://thinkpython2.com/code/Circle.py>.

Esercizio 15.2. *Scrivete una funzione di nome `disegna_rett` che prenda un oggetto `Turtle` e un `Rettangolo` e usi la Tartaruga per disegnare il Rettangolo. Vedere il Capitolo 4 per esempi di uso degli oggetti `Turtle`.*

Scrivete una funzione di nome `disegna_cerchio` che prenda un oggetto `Turtle` e un `Cerchio`, e disegni il Cerchio.

Soluzione: <http://thinkpython2.com/code/draw.py>.

Capitolo 16

Classi e funzioni

Ora che sappiamo come creare dei nuovi tipi, il passo successivo è scrivere delle funzioni che prendano i tipi personalizzati come parametri e restituiscano dei risultati. In questo capitolo presenterò anche lo “stile di programmazione funzionale” e due nuove tecniche di sviluppo.

Il codice degli esempi di questo capitolo è scaricabile dal sito <http://thinkpython2.com/code/Time1.py>. Le soluzioni degli esercizi si trovano qui: http://thinkpython2.com/code/Time1_soln.py.

16.1 Tempo

Facciamo un altro esempio di tipo personalizzato, creato dal programmatore, e definiamo una classe chiamata `Tempo` che permette di rappresentare un'ora del giorno:

```
class Tempo:
    """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.

Provate ora a scrivere 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.

Scrivete poi 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. Opzione più difficile: non usate un'istruzione `if`.

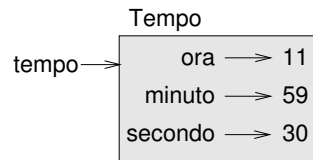


Figura 16.1: Diagramma di oggetto.

16.2 Funzioni pure

Nei prossimi paragrafi scriveremo due funzioni che sommano dei valori, espressi in termini temporali. Illustreremo 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 `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.

Come esercizio, 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**.

Per esercizio, 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 un intero:

```
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 un intero in un Tempo (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.

Come esercizio, 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
```

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 comporta 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 sono vuoti, ovvero restituiscono `None`.

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.

istruzione `assert`: Istruzione che controlla una condizione e solleva un'eccezione se fallisce.

16.7 Esercizi

Il codice degli esempi di questo capitolo è scaricabile dal sito <http://thinkpython2.com/code/Time1.py>; le soluzioni degli esercizi si trovano in http://thinkpython2.com/code/Time1_soln.py.

Esercizio 16.1. *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 16.2. *Il modulo `datetime` fornisce l'oggetto `time`, simile all'oggetto `Tempo` di questo capitolo, ma che contiene un ricco insieme di metodi e operatori. Leggetene la documentazione sul sito <http://docs.python.org/3/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 visualizzi 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.*

Soluzione: <http://thinkpython2.com/code/double.py>

Capitolo 17

Classi e metodi

Anche se abbiamo usato alcune delle caratteristiche *object-oriented* di Python, i programmi degli ultimi due capitoli non sono del tutto orientati agli oggetti, perché non mettono in evidenza le relazioni che esistono tra i tipi personalizzati e le funzioni che operano su di essi. Il passo successivo è di trasformare queste funzioni in metodi, in modo da rendere esplicite queste relazioni.

Il codice degli esempi di questo capitolo è scaricabile dal sito <http://thinkpython2.com/code/Time2.py>, e le soluzioni degli esercizi da http://thinkpython2.com/code/Point2_soln.py.

17.1 Funzionalità orientate agli oggetti

Python è un **linguaggio di programmazione orientato agli oggetti**, in altre parole contiene delle funzionalità a supporto della programmazione orientata agli oggetti, che ha le seguenti caratteristiche distintive:

- I programmi includono definizioni di classi e metodi.
- Buona parte dell'elaborazione è espressa in termini di operazioni sugli oggetti.
- Gli oggetti corrispondono spesso ad un oggetto o concetto del mondo reale, mentre i metodi che operano sugli oggetti corrispondono spesso 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` nel Capitolo 15 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 `Time1.py` 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 dei metodi per i tipi personalizzati.

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 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 nel Paragrafo 16.1, avete scritto una funzione di nome `stampa_tempo`:

```
class Tempo:
    """Rappresenta un'ora del giorno."""

def stampa_tempo(tempo):
    print('%02d:%02d:%02d' % (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:
    def stampa_tempo(tempo):
        print('%02d:%02d:%02d' % (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:
    def stampa_tempo(self):
        print('%02d:%02d:%02d' % (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 (o metodi) più versatili e rende più facile mantenere e riusare il codice.

Come esercizio, riscrivete `tempo_in_int` (vedere Paragrafo 16.4) come metodo. Potreste pensare di riscrivere anche `int_in_tempo` come metodo, ma non avrebbe molto senso: non vi sarebbe alcun oggetto sul quale invocarlo.

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 `tempo_in_int` sia stato scritto come metodo. 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 2 positional arguments but 3 were 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.

Tra parentesi, un **argomento posizionale** è un argomento privo di nome di un parametro; cioè, non è un argomento con nome. In questa chiamata di funzione:

```
sketch(pappagallo, gabbia, morto=True)
```

`pappagallo` e `gabbia` sono argomenti posizionali, e `morto` è un argomento con nome.

17.4 Un esempio più complesso

`viene_dopo` (vedere Paragrafo 16.1) è leggermente più complesso da riscrivere 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.

Per esercizio, 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 di stampa, 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.

Come esercizio, scrivete un metodo `str` per la classe `Punto`. Create un oggetto `Punto` e stampatelo.

17.7 Operator overloading

Nei tipi personalizzati, 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 personalizzati 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>.

Esercitatevi scrivendo 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
```

Come esercizio, 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 degli 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, funzioneranno anche con qualsiasi altro tipo di sequenza. Per esempio, nel Paragrafo 11.2 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 gli elementi di `s` siano idonei all'hashing, in modo da poter 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, 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 se avete oggetti dello stesso tipo che non hanno gli stessi attributi, è facile generare errori. Inizializzare tutti gli attributi di un oggetto nel metodo `init` è considerata 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 è la funzione predefinita `vars`, che prende un oggetto e restituisce un dizionario che fa corrispondere nomi degli attributi (come stringhe) e i relativi valori:

```
>>> p = Punto(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

Per gli scopi del debug, può essere utile tenere questa funzione a portata di mano:

```
def stampa_attributi(oggetto):
    for attr in vars(oggetto):
        print(attr, getattr(oggetto, attr))
```

`stampa_attributi` attraversa il dizionario 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 esposti 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 esposti 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.

17.12 Glossario

linguaggio orientato agli oggetti: Linguaggio che possiede delle caratteristiche, come tipi personalizzati e 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.

argomento posizionale: Un argomento che non include il nome di un parametro, ovvero non è un argomento con nome.

operator overloading: Cambiare il comportamento di un operatore come + in modo che funzioni con un tipo personalizzato.

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 17.1. Scaricate il codice degli esempi di questo capitolo (<http://thinkpython2.com/code/Time2.py>). Cambiate gli attributi di `Tempo` con un singolo intero che rappresenti 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://thinkpython2.com/code/Time2_soln.py.

Esercizio 17.2. 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://thinkpython2.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://thinkpython2.com/code/GoodKangaroo.py>, che spiega il problema e illustra una soluzione.

Capitolo 18

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. In questo capitolo illustrerò l'ereditarietà usando 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.

Il codice degli esempi di questo capitolo è scaricabile da <http://thinkpython2.com/code/Card.py>.

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:
    """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.

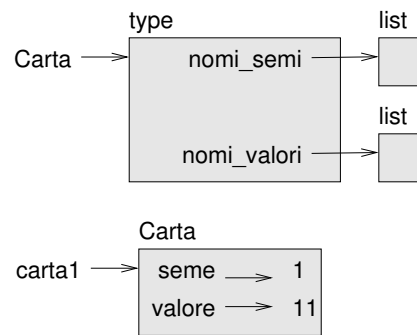


Figura 18.1: Diagramma di oggetto.

Ogni carta ha i suoi propri `seme` e `valore`, ma esiste una sola copia di `nomi_semi` e `nomi_valori`.

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 personalizzati, possiamo sovrascrivere il comportamento degli operatori predefiniti grazie a un metodo speciale chiamato `__lt__`, che sta per “less than”.

`__lt__` richiede due parametri, `self` e `other`, e restituisce `True` se `self` è minore di `other`.

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 `__lt__`:

```
# all'interno della classe Carta:

def __lt__(self, other):
    # controlla i semi
    if self.seme < other.seme: return True
    if self.seme > other.seme: return False

    # semi uguali... controlla i valori
    return self.valore < other.valore
```

Potete scriverlo anche in modo più compatto, usando un confronto di tuple:

```
# all'interno della classe Carta:

def __lt__(self, other):
    t1 = self.seme, self.valore
    t2 = other.seme, other.valore
    return t1 < t2
```

Come esercizio, scrivete un metodo `__lt__` per gli oggetti `Tempo`. Potete usare un confronto di tuple, ma anche prendere in considerazione di confrontare degli interi.

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:

    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 il metodo delle stringhe `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.

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 altro metodo 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, per migliorarne l'aspetto.

In questo caso, `aggiungi_carta` è un metodo "sottile" che esprime un'operazione su una lista, in una forma appropriata per i mazzi di carte. Esso migliora l'aspetto, ovvero l'interfaccia, dell'implementazione.

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`.

Come esercizio, 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 `__lt__` che abbiamo definito.

18.7 Ereditarietà

L'ereditarietà è la capacità di definire una nuova classe come versione modificata di una classe già esistente.

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.

Per definire una nuova classe che eredita da una classe esistente, basta scrivere tra parentesi il nome della classe esistente:

```
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`.

Quando una nuova classe eredita da una esistente, quest'ultima è chiamata **madre** (o superclasse) e quella nuova è chiamata **figlia** (o sottoclasse).

In questo esempio, `Mano` eredita `__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 e non quello di *Mazzo*:

```
>>> mano = Mano('nuova mano')
>>> mano.carte
[]
>>> mano.label
'nuova mano'
```

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*.

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.

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:

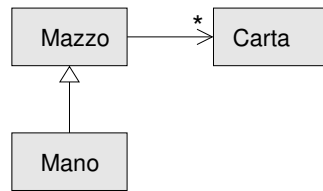


Figura 18.2: Diagramma di classe.

- 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 oggetti di una classe possono prendere come parametri oggetti di una seconda classe oppure usarli per svolgere parte delle elaborazioni. Una relazione di questo tipo è detta **dipendenza**.

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.

In questo diagramma non vi sono dipendenze. In genere, verrebbero illustrate con delle frecce tratteggiate. Se vi sono parecchie dipendenze, talvolta vengono omesse.

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.

18.9 Debug

L'ereditarietà può rendere il debug difficoltoso, perché quando invocate un metodo su un oggetto, può risultare laborioso capire 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 sotto-classi sovrascrive il metodo, avrete invece quella diversa versione. Questo comportamento è appropriato, ma a volte può confondere.

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()
>>> 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), che è la sequenza di classi che Python ricerca per “risolvere” un nome di metodo.

Un consiglio per la progettazione di un programma: quando 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` e `ManoDiPoker`.

Se violate questa regola, conosciuta come “principio di sostituzione di Liskov”, il vostro codice crollerà come (perdonatemi) un castello di carte.

18.10 Incapsulamento dei dati

Il capitolo precedente ha illustrato una tecnica di sviluppo detta “progettazione orientata agli oggetti”. Abbiamo identificato gli oggetti che ci servivano—come `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://thinkpython2.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:
```

```
    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 il comportamento—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.

Come esercizio, scaricate il mio codice da (<http://thinkpython2.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://thinkpython2.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 effettuare ulteriori calcoli.

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.

dipendenza: Relazione tra due classi dove istanze di una classe utilizzano istanze dell'altra classe, ma senza conservarle sotto forma di attributi.

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.

incapsulamento dei dati: Tecnica di sviluppo che prevede un prototipo che usa variabili globali e una versione finale in cui le variabili globali vengono trasformate in attributi di istanza.

18.12 Esercizi

Esercizio 18.1. *Dato il seguente programma, disegnate un diagramma di classe UML (Unified Modeling Language) che illustri queste classi e le relazioni che intercorrono tra esse.*

```
class PingPongMadre:
    pass

class Ping(PingPongMadre):
    def __init__(self, pong):
        self.pong = pong

class Pong(PingPongMadre):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

Esercizio 18.2. *Scrivete un metodo per Mazza di nome `dai_mani` che prenda come parametri il numero di mani e il numero di carte da dare a ciascuna mano, e crei il numero stabilito di oggetti Mano, distribuisca il numero prefissato di carte a ogni mano e restituisca una lista delle Mani.*

Esercizio 18.3. *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://thinkpython2.com/code/>:

`Card.py` : Versione completa delle classi `Carta`, `Mazzo` 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://thinkpython2.com/code/PokerHandSoln.py>.

Capitolo 19

Ulteriori strumenti

Uno degli obiettivi di questo libro è di illustrarvi il minimo indispensabile di Python. Quando esistono due modi diversi di fare qualcosa, preferisco sceglierne uno ed evitare di citare l'altro, oppure inserire il secondo all'interno di un esercizio.

Ora vorrei tornare a recuperare alcune chicche che avevo tralasciato. Python è dotato di parecchie funzionalità che non sono indispensabili—potete scrivere del buon codice anche senza usarle—ma che in certi casi vi permettono di scrivere del codice più conciso, leggibile, efficiente, o anche tutte e tre le cose insieme.

19.1 Espressioni condizionali

Abbiamo visto nel Paragrafo 5.4 le istruzioni condizionali, che vengono usate di frequente per scegliere uno tra due valori alternativi, per esempio:

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

Questa istruzione controlla se x è positivo. Se lo è, calcola `math.log`, altrimenti `math.log` scatenerrebbe un `ValueError`. Per evitare che il programma si arresti, generiamo un “NaN”, che è un valore a virgola mobile speciale che rappresenta “Not a Number” (Non è un Numero).

Possiamo scrivere questa condizione in modo più conciso utilizzando un'**espressione condizionale**:

```
y = math.log(x) if x > 0 else float('nan')
```

Si può quasi leggere questa riga come fosse: “ y diventa $\log x$ se x è maggiore di 0; altrimenti diventa NaN”.

A volte, le funzioni ricorsive possono essere riscritte utilizzando le espressioni condizionali. Prendiamo ad esempio una versione ricorsiva di fattoriale:

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

Si può riscrivere così:

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

Un altro utilizzo delle espressioni condizionali è la gestione degli argomenti opzionali. Per esempio, ecco il metodo `init` di `GoodKangaroo` (vedere Esercizio 17.2):

```
def __init__(self, nome, contenuti=None):
    self.nome = nome
    if contenuti == None:
        contenuti = []
    self.contenuto_tasca = contenuti
```

Si può riscrivere così:

```
def __init__(self, nome, contenuti=None):
    self.nome = nome
    self.contenuto_tasca = [] if contenuti == None else contenuti
```

In generale, si può sostituire un'istruzione condizionale con un'espressione condizionale se entrambe le ramificazioni contengono semplici espressioni che vengono o ritornate o assegnate alla stessa variabile.

19.2 List comprehension

Nel Paragrafo 10.7 abbiamo visto gli schemi di mappa e filtro. Per esempio, questa funzione prende una lista di stringhe, mappa il metodo delle stringhe `capitalize` negli elementi e restituisce una nuova lista di stringhe:

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

Si può scrivere in modo più conciso utilizzando una **list comprehension**:

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

Gli operatori parentesi quadre indicano che stiamo costruendo una nuova lista. L'espressione all'interno delle parentesi specifica gli elementi della lista, e il costrutto `for` specifica la sequenza che stiamo attraversando.

La sintassi di una list comprehension è un po' sgraziata, perché la variabile del ciclo, `s` in questo esempio, compare nell'espressione prima di ottenerne la definizione.

Si può usare la list comprehension anche per filtrare. Per esempio, questa funzione seleziona solo gli elementi di `t` che sono composti di lettere maiuscole, e restituisce una nuova lista:

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

Riscriviamola usando una list comprehension:

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

Le list comprehension sono concise e leggibili, almeno per le espressioni semplici. E di solito sono più veloci dei cicli for equivalenti, a volte molto più veloci. Capisco quindi se mi state biasimando per non averne parlato prima.

La giustificazione è che il debug delle list comprehension è più difficile, perché non potete inserire delle istruzioni di stampa nel ciclo. Vi consiglio di usarle solo se i calcoli sono abbastanza semplici da avere buone probabilità di azzeccarci al primo colpo. Che per un principiante, vuol dire quasi mai.

19.3 Generator expression

Le **generator expression** assomigliano sintatticamente a delle list comprehension, ma con parentesi tonde anziché quadre:

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

Il risultato è un oggetto generatore che è in grado di iterare attraverso una sequenza di valori. Ma a differenza di una list comprehension, non calcola i valori tutti in una volta: attende che gli venga chiesto di farlo. Con la funzione predefinita `next`, si ottiene dal generatore il valore successivo:

```
>>> next(g)
0
>>> next(g)
1
```

Arrivati alla fine della sequenza, `next` solleva un'eccezione `StopIteration`. Si può anche usare un ciclo `for` per iterare attraverso i valori:

```
>>> for val in g:
...     print(val)
4
9
16
```

L'oggetto generatore mantiene traccia del punto in cui si trova all'interno della sequenza, quindi il ciclo `for` riprende da dove `next` era rimasto. Una volta che il generatore è esaurito, continua sollevando delle `StopIteration`:

```
>>> next(g)
StopIteration
```

Le generator expression vengono usate spesso con funzioni come `sum`, `max`, e `min`:

```
>>> sum(x**2 for x in range(5))
30
```

19.4 any e all

Python dispone di una funzione predefinita di nome `any`, che prende una sequenza di valori booleani e restituisce `True` se almeno uno dei valori è `True`. Funziona sulle liste:

```
>>> any([False, False, True])
True
```

Ma viene usata spesso con le generator expression:

```
>>> any(lettera == 't' for lettera in 'monty')
True
```

Questo esempio non è granché utile, perché fa la stessa cosa dell'operatore `in`. Ma possiamo usare `any` per riscrivere alcune delle funzioni di ricerca che avevamo scritto nel Paragrafo 9.3. Per esempio, avremmo potuto scrivere `evita` così:

```
def evita(parola, vietate):
    return not any(lettera in vietate for lettera in parola)
```

La funzione si legge quasi come fosse: “la parola evita le vietate se non c'è alcuna lettera in vietate per ogni lettera in parola.”

L'uso di `any` con una generator expression è efficiente, perché si ferma immediatamente se trova un valore `True`, senza dover necessariamente verificare tutta la sequenza.

Python contiene poi un'altra funzione predefinita, `all`, che restituisce `True` se ogni elemento di una sequenza è `True`. Come esercizio, usate `all` per riscrivere la funzione `usa_tutte` del Paragrafo 9.3.

19.5 Insiemi (set)

Nel Paragrafo 13.6 avevo utilizzato dei dizionari per trovare le parole che comparivano in un testo, ma non in un elenco di parole. La funzione che avevo scritto prendeva `d1`, contenente le parole del testo come chiavi, e `d2`, contenente l'elenco di parole. Essa restituiva un dizionario contenente le chiavi di `d1` che non comparivano in `d2`.

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

In tutti questi dizionari, i valori sono `None` perché non sono necessari e non vengono usati. Ma questo spreca dello spazio di memoria.

Python dispone di un altro tipo predefinito chiamato **insieme** o `set`, che si comporta come una raccolta di chiavi di dizionario prive di valori. Aggiungere elementi ad un insieme è rapido, come pure controllare se un elemento appartiene all'insieme. Vengono poi forniti metodi e operatori per eseguire le comuni operazioni sugli insiemi.

Per esempio, la sottrazione di insiemi è disponibile sotto forma di metodo chiamato `difference` oppure come operatore, `-`. Possiamo allora riscrivere `sottrai` in questo modo:

```
def sottrai(d1, d2):
    return set(d1) - set(d2)
```

Il risultato è un insieme anziché un dizionario, ma per operazioni come l'iterazione il comportamento è identico.

Alcuni esercizi di questo libro possono essere svolti in modo conciso ed efficiente usando gli insiemi. Per esempio, questa è una soluzione di `ha_duplicati`, dall'Esercizio 10.7, che utilizza un dizionario:

```
def ha_duplicati(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

Quando un elemento compare per la prima volta, viene aggiunto al dizionario. Se il medesimo elemento ricompare, la funzione restituisce `True`.

Usando gli insiemi, si può riscrivere la funzione così:

```
def ha_duplicati(t):
    return len(set(t)) < len(t)
```

Siccome un elemento può comparire in un insieme solo una volta, se in `t` esiste qualche elemento che compare più volte, l'insieme risulterà più piccolo di `t`. Se invece non ci sono duplicati, l'insieme avrà la stessa dimensione di `t`.

Si possono usare gli insiemi anche per risolvere alcuni esercizi del Capitolo 9. Per esempio, questa è la versione di `usa_solo` con un ciclo:

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

`usa_solo` controlla se tutte le lettere in una `word` sono tra quelle valide. La possiamo riscrivere così:

```
def usa_solo(parola, valide):
    return set(parola) <= set(valide)
```

L'operatore `<=` controlla se un insieme è un sottoinsieme di un altro, compresa la possibilità che siano uguali, il che è vero se tutte le lettere nella `parola` fanno parte delle `valide`.

Per esercizio, riscrivete la funzione `evita` usando gli insiemi.

19.6 Contatori

Un contatore è una specie di insieme, tranne per il fatto che se un elemento compare più di una volta, il contatore prende nota di quante volte compare. Se vi è noto il concetto matematico di **multiinsieme**, un contatore è un modo immediato per rappresentarlo.

Il contatore è definito all'interno di un modulo standard chiamato `collections`, quindi dovete innanzitutto importarlo. Potete inizializzare un contatore con una stringa, lista o qualsiasi altro oggetto che sia iterabile:

```
>>> from collections import Counter
>>> conta = Counter('parrot')
>>> conta
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

I contatori si comportano per molti versi come i dizionari: fanno corrispondere ciascuna chiave al numero di volte in cui essa compare. Come per i dizionari, le chiavi devono essere idonee all'hashing.

A differenza dei dizionari, i contatori non sollevano eccezioni in caso di accesso a un elemento che non esiste; invece, restituiscono 0:

```
>>> conta['d']
0
```

Possiamo usare i contatori per riscrivere anagramma dell'Esercizio 10.6:

```
def anagramma(parola1, parola2):
    return Counter(parola1) == Counter(parola2)
```

Se due parole sono anagrammi, contengono le stesse lettere, lo stesso numero di volte: pertanto i loro contatori sono equivalenti.

Anche i contatori sono dotati dei metodi e degli operatori per eseguire le operazioni tipiche degli insiemi, incluse addizione, sottrazione e intersezione. Ed espongono un metodo molto utile, `most_common`, che restituisce una lista di coppie valore-frequenza, in ordine di frequenza decrescente:

```
>>> conta = Counter('parrot')
>>> for valore, frequenza in count.most_common(3):
...     print(valore, frequenza)
r 2
p 1
a 1
```

19.7 defaultdict

Il modulo `collections` contiene anche `defaultdict`, che è simile a un dizionario, con la differenza che quando si tenta di accedere ad una chiave inesistente, può generare al volo un nuovo valore.

Nel creare un `defaultdict`, dovete fornire una funzione che viene usata per creare i nuovi valori. Una funzione usata per creare oggetti viene detta da alcuni **factory**. Le funzioni predefinite che creano liste, insiemi e altri tipi, possono essere usate come factory:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

Notate che l'argomento è `list`, che è un oggetto classe, non `list()`, che è una nuova lista. La funzione che avete creato viene chiamata soltanto quando tentate di accedere ad una chiave che non esiste.

```
>>> t = d['nuova chiave']
>>> t
[]
```


La nuova lista, che chiamiamo `t`, viene aggiunta al dizionario. Quindi se modifichiamo `t`, i cambiamenti compaiono in `d`:

```
>>> t.append('nuovo valore')
>>> d
defaultdict(<class 'list'>, {'nuova chiave': ['nuovo valore']})
```

Se state creando un dizionario di liste, usare `defaultdict` permette spesso di scrivere codice più semplice. Nella mia risoluzione dell'Esercizio 12.2, che potete scaricare da http://thinkpython2.com/code/anagram_sets.py, ho creato un dizionario che mappa da una stringa ordinata di lettere nella lista di parole che si possono comporre con quelle lettere. Per esempio, `'opst'` corrisponde alla lista `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Questo è il codice di partenza:

```
def tutti_anagrammi(nomefile):
    d = {}
    for riga in open(nomefile):
        parola = riga.strip().lower()
        t = signature(parola)
        if t not in d:
            d[t] = [parola]
        else:
            d[t].append(parola)
    return d
```

Si può semplificare usando `setdefault`, che potreste avere usato nell'Esercizio 11.2:

```
def tutti_anagrammi(nomefile):
    d = {}
    for riga in open(nomefile):
        parola = riga.strip().lower()
        t = signature(parola)
        d.setdefault(t, []).append(parola)
    return d
```

Questa soluzione ha il difetto di creare ogni volta una nuova lista, anche se non è necessario. Non è un grande problema se si tratta di liste, ma se la funzione `factory` è complessa, può diventarlo.

Ma si può evitare il problema e semplificare il codice con un `defaultdict`:

```
def tutti_anagrammi(nomefile):
    d = defaultdict(list)
    for riga in open(nomefile):
        parola = riga.strip().lower()
        t = signature(parola)
        d[t].append(parola)
    return d
```

La mia risoluzione dell'Esercizio 18.3, scaricabile da <http://thinkpython2.com/code/PokerHandSoln.py>, usa `setdefault` nella funzione `has_straightflush`. Ha il difetto di creare un oggetto `Mano` ad ogni ripetizione del ciclo, anche se non serve. Come esercizio, riscrivetela usando un `defaultdict`.

19.8 Tuple con nome (namedtuple)

Molti oggetti semplici sono, fondamentalmente, delle raccolte di valori tra loro correlati. Ad esempio, l'oggetto `Punto` che abbiamo definito nel Capitolo 15 contiene due numeri, `x` e `y`. Nella definizione di una classe come questa, si comincia di solito con un metodo `init` e un metodo `str`:

```
class Punto:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

Qui serve molto codice per trasportare una piccola quantità di informazione. Python dispone di un modo più conciso per dire la stessa cosa:

```
from collections import namedtuple
Punto = namedtuple('Punto', ['x', 'y'])
```

Il primo argomento è il nome della classe che volete creare. Il secondo è una lista degli attributi, come stringhe, che l'oggetto `Punto` deve avere. Il valore di ritorno da `namedtuple` è un oggetto classe:

```
>>> Punto
<class '__main__.Punto'>
```

`Punto` dispone automaticamente dei metodi `__init__` e `__str__`, pertanto non occorre scriverli.

Per creare un oggetto `Punto`, usate la classe `Punto` come fosse una funzione:

```
>>> p = Punto(1, 2)
>>> p
Punto(x=1, y=2)
```

Il metodo `init` assegna gli argomenti agli attributi usando i nomi che avete specificato. Il metodo `str` mostra una rappresentazione dell'oggetto `Punto` e dei suoi attributi.

Potete accedere agli elementi della `namedtuple` usando il loro nome:

```
>>> p.x, p.y
(1, 2)
```

Ma potete anche trattare una `namedtuple` come una tupla, e usare gli indici:

```
>>> p[0], p[1]
(1, 2)
```

```
>>> x, y = p
>>> x, y
(1, 2)
```

Le tuple con nome offrono un modo rapido per definire delle classi semplici. Per contro, le classi semplici non sempre rimangono tali. Si potrebbe decidere in un secondo tempo di voler aggiungere dei metodi a una tupla con nome. In quel caso, occorrerebbe definire una nuova classe che erediti dalla tupla con nome:

```
class IperPunto(Punto):
    # aggiungere qui altri metodi
```

Oppure si può passare ad una definizione di classe tradizionale.

19.9 Raccolta di argomenti con nome

Nel Paragrafo 12.4, avevamo visto come scrivere una funzione che raccoglie in una tupla i suoi argomenti:

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

Questa funzione può essere chiamata con un numero qualunque di argomenti posizionali (cioè, argomenti che non hanno nome):

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

Tuttavia, l'operatore di raccolta `*` non funziona con gli argomenti con nome:

```
>>> stampatutti(1, 2.0, terzo='3')
TypeError: stampatutti() got an unexpected keyword argument 'terzo'
```

Per raccogliere gli argomenti con nome, si usa invece l'operatore `**`:

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

Si può chiamare il parametro di raccolta come si vuole, ma per prassi si usa `kwargs`. Il risultato è un dizionario che mappa i nomi nei valori:

```
>>> stampatutti(1, 2.0, terzo='3')
(1, 2.0) {'terzo': '3'}
```

Se disponete di un dizionario di nomi e valori, potete usare l'operatore di spaccettamento, `**`, per chiamare una funzione:

```
>>> d = dict(x=1, y=2)
>>> Punto(**d)
Punto(x=1, y=2)
```

Senza l'operatore di spaccettamento, la funzione interpreterebbe `d` come un singolo argomento posizionale, assegnandolo a `x` e lamentando l'assenza di qualcosa da assegnare a `y`:

```
>>> d = dict(x=1, y=2)
>>> Punto(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

Quando si lavora con funzioni che hanno parecchi parametri, è una buona idea creare e passare loro dei dizionari che contengono le opzioni di uso più frequente.

19.10 Glossario

espressione condizionale: Un'espressione che contiene uno tra due alternativi valori, a seconda di una data condizione.

list comprehension: Espressione con un ciclo `for` tra parentesi quadre che genera una nuova lista.

generator expression: Espressione con un ciclo `for` tra parentesi tonde che produce un oggetto generatore.

multiinsieme: Ente matematico che rappresenta una corrispondenza tra gli elementi di un insieme e il numero di volte in cui compaiono.

factory: Funzione, di solito passata come argomento, usata per creare oggetti.

19.11 Esercizi

Esercizio 19.1. *La seguente funzione calcola ricorsivamente il coefficiente binomiale:*

```
def coeff_binomiale(n, k):  
    """Calcola il coefficiente binomiale "n sopra k".  
  
    n: numero di prove  
    k: numero di successi  
  
    ritorna: int  
    """  
    if k == 0:  
        return 1  
    if n == 0:  
        return 0  
  
    res = coeff_binomiale(n-1, k) + coeff_binomiale(n-1, k-1)  
    return res
```

Riscrivete il corpo della funzione usando delle espressioni condizionali nidificate.

Nota: questa funzione non è molto efficiente, perché finisce per calcolare continuamente gli stessi valori. Potreste renderla più efficiente con la memoizzazione (vedere Paragrafo 11.6). Riscontrerete però che è difficile farlo, scrivendola con le espressioni condizionali.

Appendice A

Debug

Nell'operazione di rimozione degli errori da un programma, è opportuno distinguere i vari tipi di errore in modo da poterli rintracciare più velocemente:

- Gli errori di sintassi vengono scoperti dall'interprete quando traduce il codice sorgente in codice macchina. Indicano che c'è qualcosa di sbagliato nella struttura del programma. Esempio: omettere i due punti alla fine di una istruzione `def` genera 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 ricorrenza 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 tutte le istruzioni composte, inclusi `for`, `while`, `if`, e le istruzioni `def`.
3. Accertatevi che ogni stringa nel codice sia racchiusa da una coppia di virgolette o apici, e che questi siano di tipo indifferenziato e non "tipografici".
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 il codice sia allineato 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.
8. Se nel codice (compresi stringhe e commenti) ci sono caratteri non-ASCII, tipo le lettere accentate, potrebbero causare problemi, anche se Python 3 gestisce abbastanza bene questi caratteri. Fate attenzione, se copiate e incollate del testo da una pagina web o da altre fonti.

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 leggerlo 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 alcuna funzione 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 nel programma ci sia una chiamata di funzione e che il flusso di esecuzione la raggiunga (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".

Avviate 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 provoca un errore di `Maximum recursion depth exceeded`, dopo che il programma è stato in esecuzione per qualche istante.

Se sospettate che una funzione stia provocando una ricorsione infinita, controllate innanzitutto che esista un caso base. Deve esistere un qualche tipo di condizione che provoca il ritorno della funzione 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, in modo da visualizzare i parametri. Ora, eseguendo il programma vedrete alcune righe di output con i valori dei parametri per ogni chiamata della funzione. 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. Controllate che il nome sia scritto esattamente e che sia coerente. Ricordate anche 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 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. Se le chiavi sono delle stringhe, ricordate che c'è differenza tra lettere maiuscole e minuscole.

AttributeError: State tentando di accedere a un attributo o a un metodo che non esiste. Controllate se è scritto giusto! Potete usare la funzione predefinita `vars` per elencare gli attributi esistenti.

Se un `AttributeError` indica che un oggetto è di tipo `NoneType`, vuol dire che è `None`. Il problema non è il nome dell'attributo, ma l'oggetto. la ragione per cui un oggetto è `None` può essere 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 <https://docs.python.org/3/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 di stampa 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 per certo solo quello che il programma dovrebbe fare, ma non fa.

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 si verifica l'errore .

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 funzioni o metodi in altri moduli Python. Leggete la documentazione delle funzioni che chiamate. Provatele scrivendo semplici test e controllando i risultati.

Per programmare, vi serve un modello mentale di come funziona il programma. Se scrivete un programma che non fa quello che volete, 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, dovreste 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 poi possono diventare 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 l'ordine delle operazioni

A.3.3 Ho una funzione che non restituisce quello che voglio.

Se avete un'istruzione `return` associata ad un'espressione complessa, non avete modo di stampare il risultato 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 (2012). 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 del problema, e raggruppare le funzioni in categorie a seconda di quanto velocemente crescono 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, almeno per grandi valori 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 . L'**operazione dominante** è quella relativa al termine con il più alto esponente.

Per l'algoritmo A, l'operazione 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 $an^2 > bn$, qualunque siano i valori di a e b .

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 piccola operazione 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 l'operazione 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 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 operazione dominante n^2 appartengono a $O(n^2)$ e sono dette **quadratiche**.

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)
$O(n)$	
$O(n \log_b n)$	linearitmico
$O(n^2)$	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 l'operazione 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

In Python, molte operazioni aritmetiche sono a 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 dell'operazione dominante più piccolo.

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.

Il concatenamento di stringhe è lineare. Il tempo di esecuzione dipende dalla somma delle lunghezze degli operandi.

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. Il metodo delle stringhe `join` è lineare, e il tempo di esecuzione dipende dalla lunghezza totale delle stringhe.

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 il tempo medio di ciascuna 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 `update` è proporzionale alle dimensioni del dizionario passato come parametro, non del dizionario che viene aggiornato.
- `keys`, `values` e `items` sono a tempo costante perché restituiscono iteratori. Ma se attraversate con un ciclo un iteratore, il ciclo sarà lineare.

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 quando consultate 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.

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(k): Cerca e restituisce il valore corrispondente alla chiave `k`. Con un dizionario Python, `d`, questa operazione si scrive `d[k]` oppure `d.get(k)`.

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:

    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 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 MappaLineari. 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:

    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 equivalenti restituiscono lo stesso valore hash, ma l'inverso non è necessariamente vero: due oggetti con valori 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 dell'operazione 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 `MappeLineari`, `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 `MappeLineari`.

Ecco un'implementazione di una tabella hash:

```
class MappaHash:

    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.

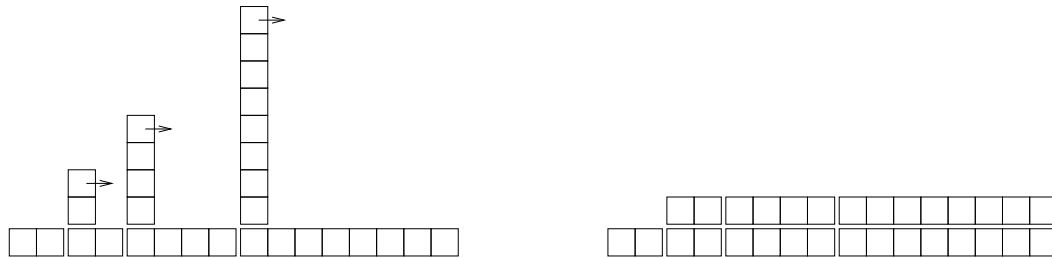


Figura B.1: Costo delle aggiunte a una Tabella Hash.

`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 `MappaLineari` cambia il denominatore dell'operatore modulo in `trova_mappa`. Ciò significa che alcuni oggetti che erano hashati 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 `MappaLineari`, 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, spalmando 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 incrementaste 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://thinkpython2/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.

B.5 Glossario

analisi degli algoritmi: Un modo di confrontare gli algoritmi in termini di tempo di esecuzione e/o requisiti di memoria.

modello di macchina: Rappresentazione semplificata di un computer usato per analizzare il comportamento degli algoritmi.

caso peggiore: L'input che, con riferimento ad un algoritmo, lo rende più lento nell'esecuzione o richiede più spazio di memoria.

operazione dominante: In un polinomio, il termine con il più alto esponente.

punto di intersezione: La dimensione del problema per cui due diversi algoritmi richiedono lo stesso tempo di esecuzione o lo stesso spazio di memoria.

ordine di complessità: Insieme di funzioni che, per gli scopi dell'analisi degli algoritmi, crescono in maniera equivalente. Per esempio, tutte le funzioni con incremento lineare appartengono allo stesso ordine di complessità.

Notazione O-grande: Notazione per rappresentare l'ordine di complessità; ad esempio, $O(n)$ rappresenta l'insieme delle funzioni con incremento lineare.

lineare: Un algoritmo il cui tempo di esecuzione è proporzionale alle dimensioni del problema, almeno per i problemi di grandi dimensioni.

quadratico: Un algoritmo il cui tempo di esecuzione è proporzionale a n^2 , dove n è una misura delle dimensioni del problema.

ricerca: Il problema di rintracciare un elemento in un insieme (come una lista o un dizionario) oppure stabilire che non è presente.

tabella hash: Struttura di dati che rappresenta una raccolta di coppie chiave-valore e ne esegue la ricerca a tempo costante.

Indice analitico

- abs, funzione, 52
- accesso, 92
- accumulatore, 102
 - istogramma, 129
 - lista, 95
 - somma, 95
 - stringa, 179
- Ackermann, funzione di, 61, 115
- add, metodo, 170
- addizione
 - con riporto, 68, 158, 160
- aggiornamento, 64, 67, 69
 - di database, 143
 - elemento, 93
 - istogramma, 130
 - operatore di, 95
 - slicing, 94
 - variabile globale, 113
- alfabetico, 73, 85
- alfabeto, 38
- algoritmo, 2, 68, 69, 133, 209
 - md5, 148
 - radice quadrata, 69
- alias, 97, 98, 102, 151, 153, 174
 - copiare per evitare, 101
- all, funzione, 192
- ambiguità, 5
- anagramma, 103
- anagramma, insieme, 125, 148
- analisi degli algoritmi, 209, 218
- analisi delle primitive, 212
- and, operatore, 40
- any, funzione, 192
- apici, 3, 4, 74
- appartenenza
 - dizionario, 106
 - insieme di, 115
 - lista, 92
 - ricerca binaria, 103
- append, metodo, 94, 100, 103, 178, 179
- arco, funzione, 32
- argomento, 17, 19, 22, 26, 99
 - con nome, 33, 37, 197
 - lista, 99
 - opzionale, 76, 80, 97, 109, 190
 - posizionale, 168, 174, 197
 - raccolta, 120
 - spacchettamento, 120
 - tupla di lunghezza variabile, 120
- aritmetico
 - operatore, 3
- assegnazione, 14, 63, 91
 - elemento, 74, 92, 118
 - potenziata, 95, 102
 - tupla, 118, 119, 121, 125
- assegnazione, istruzione di, 9
- assert, istruzione, 162
- attraversamento, 72, 75, 78, 79, 85, 86, 95,
102, 107, 108, 121, 129
 - dizionario, 122, 173
 - lista, 93
- AttributeError, 155, 203
- attributo, 155, 173
 - __dict__, 172
 - di classe, 176, 184
 - di istanza, 150, 176, 185
 - inizializzazione, 172
 - istanza, 155
- Austin, Jane, 129
- avviare Python, 2
- bella copia, 114
- benchmarking, 135, 137
- bifronte, parola, 104
- Big-Oh notation, 218
- bisect, modulo, 104
- bitwise, operatore, 4
- blocco, 201
- bool, tipo, 40
- booleano, operatore, 77
- break, istruzione, 66
- bubble sort, 209

- bug, 6, 8
 - il peggiore, 174
- bytes, oggetto, 143, 147
- cadenza di corsa, 8, 16, 163
- calcolatrice, 8, 16
- cancellare, elementi di lista, 96
- Canguro, classe, 174
- Car Talk, 88, 89, 115, 126
- carattere, 71
- Carta, classe, 176
- carte da gioco, 175
- cartella, 141
- caso
 - medio, 210
 - particolare, 87, 88, 159
 - peggiore, 210
- caso base, 44, 48
- caso di prova minimo, 204
- caso peggiore, 218
- cerchio, funzione, 32
- checksum, 145, 148
- chiamata, grafico di, 114
- chiave, 105, 114
- chiave-valore, coppia, 105, 114, 122
- choice, funzione, 128
- Ciao, Mondo, 3
- cicli e contatori, 75
- ciclo, 31, 37, 65, 121
 - attraversamento, 72
 - con dizionario, 108
 - con indici, 86, 93
 - con stringhe, 75
 - condizione, 202
 - for, 31, 44, 72, 93
 - infinito, 65, 69, 201, 202
 - nidificato, 178
 - while, 64
- classe, 4, 149, 155
 - attributo di, 176, 184
 - Canguro, 174
 - Carta, 176
 - definizione di, 149
 - figlia, 180, 185
 - madre, 180, 185
 - Mano, 180
 - Mazzo, 178
 - Punto, 150, 169
 - Rettangolo, 151
 - Tempo, 157
 - classe, oggetto, 196
 - close, metodo, 140, 144, 145
 - __lt__, metodo, 177
 - codice morto, 52, 61, 204
 - codificare, 175, 184
 - coefficiente dominante, 210
 - coerenza, controllo di, 114, 161
 - collaboratori, vii
 - collections, 193, 194, 196
 - comando Unix
 - ls, 145
 - commento, 13, 15
 - commutatività, 13, 171
 - comparsa, funzione, 52
 - comparison sort, 213
 - compleanno, 163
 - compleanno, paradosso del, 103
 - complessità
 - esponenziale, 211
 - lineare, 211
 - logaritmica, 211
 - ordine di, 210
 - quadratica, 211
 - complessità, ordine di, 218
 - composizione, 19, 22, 27, 54, 178
 - concatenamento, 13, 15, 23, 73, 75, 97
 - lista, 93, 100, 103
 - condizionale, espressione, 189, 197
 - condizionale, istruzione, 190
 - condizione, 41, 47, 65, 202
 - di guardia, 59, 61
 - in serie, 42, 48
 - nidificata, 42, 48
 - confronto
 - stringa, 77
 - tupla, 178
 - confronto di algoritmi, 209
 - confronto, operatore di, 40
 - congettura di Collatz, 66
 - congruenza, controllo di, 114
 - contachilometri, 88
 - contatore, 75, 80, 106, 113, 193
 - contatori e cicli, 75
 - conteggio, 145
 - contenuto, oggetto, 174
 - controllo dei tipi, 58
 - controllo errore, 58
 - conversione
 - di tipo, 17
 - copia

- di un oggetto, 153
- per evitare alias, 101
- profonda, 154, 156
- shallow, 154, 156
- slicing, 74, 94
- copy, modulo, 153
- corpo, 19, 26, 65
- corrispondenza, 175
- costo medio, 217
- count, metodo, 80
- Creative Commons, vii
- criptare, 175
- cruciverba, 83
- cumulativa, somma, 102
- curva di Koch, 49
- database, 143, 147
- database, oggetto, 143
- datetime, modulo, 163
- dbm, modulo, 143
- debug, 6, 8, 14, 25, 36, 46, 59, 78, 87, 100, 113, 124, 135, 146, 155, 161, 172, 182, 191, 199
 - binario, 68
 - con la papera di gomma, 137
 - risposta emotiva, 7, 206
 - sperimentale, 25, 136
 - superstizione, 206
- debugger (pdb), 204
- decremento, 64, 69
- deepcopy, funzione, 154
- def, parola chiave, 19
- default, valore di, 137
- defaultdict, 194
- definizione
 - circolare, 56
 - di classe, 149
 - di funzione, 19
 - ricorsiva, 56, 126
- del, operatore, 96
- delimitatore, 97, 102
- deterministico, 128, 137
- diagramma
 - di classe, 182, 185
 - di oggetto, 150, 152, 154, 156, 157, 177
 - di stack, 23, 27, 37, 44, 57, 61, 99
 - di stato, 9, 14, 63, 79, 92, 98, 110, 122, 150, 152, 154, 157, 177
 - grafico di chiamata, 114
- dichiarazione, 112, 115
- __dict__ attributo, 172
- dict, funzione, 105
- diff, 148
- Dijkstra, Edsger, 88
- dipendenza, relazione, 185
- directory, 141, 147
 - di lavoro, 141
 - esplorazione, 142
- divisibilità, 39
- divisione
 - decimale, 39
 - intera, 39, 47
- divmod, 119, 160
- dizionario, 105, 114, 122, 203
 - attraversamento, 122, 173
 - ciclo con, 108
 - inizializzazione, 122
 - inverso, 109
 - lookup, 108
 - lookup inverso, 108
 - metodi, 213
 - sottrazione, 131
- dizionario, metodi
 - anydbm, modulo, 144
- docstring, 36, 37, 150
- dot notation, 18, 27, 150, 166, 176
- Doyle, Arthur Conan, 25
- due punti, 19, 200
- duplicato, 103, 115, 148, 193
- eccezione, 14, 15, 199, 203
 - AttributeError, 155, 203
 - IndexError, 72, 78, 92, 204
 - IOError, 142
 - KeyError, 106, 203
 - LookupError, 109
 - NameError, 23, 203
 - OverflowError, 47
 - RuntimeError, 45
 - StopIteration, 191
 - SyntaxError, 19
 - TypeError, 72, 74, 110, 118, 120, 141, 168, 203
 - UnboundLocalError, 113
 - ValueError, 46, 119
- eccezione, gestione, 143
- elemento, 74, 79, 91, 102, 105
 - aggiornamento, 93
 - assegnazione, 74, 118
 - cancellazione, 96

- dizionario, 114
- elemento, assegnazione, 92
- elif, parola chiave, 42
- Elkner, Jeff, v, vi
- ellissi, 20
- else, parola chiave, 41
- email, indirizzo, 119
- emotività, nel debug, 7, 206
- enumerate, funzione, 121
- enumerate, oggetto, 121
- epsilon, 67
- equivalente, 98, 102
- equivalenza, 154
- ereditarietà, 180, 182, 185, 196
- errore
 - di battitura, 136
 - di formato, 124
 - di runtime, 14, 199
 - di semantica, 14, 199, 204
 - di sintassi, 14, 199
 - in esecuzione, 14, 45, 47, 199, 203
 - messaggio di, 8, 14, 199
- esadecimale, 150
- esecuzione, 11, 15
- esecuzione alternativa, 41
- esecuzione condizionale, 41
- esponente, 210
- espressione, 10, 15
 - booleana, 40, 47
 - condizionale, 189, 197
 - grande e complicata, 205
- eval, funzione, 70
- exists, funzione, 142
- extend, metodo, 94
- factory, funzione, 194, 195, 198
- False, valore speciale, 40
- fattoriale, 189
- fattoriale, funzione, 56, 58
- Fermat, ultimo teorema di, 48
- fibonacci, funzione, 58, 111
- figlia, classe, 180
- file, 139
 - di testo, 147
 - lettura e scrittura, 139
 - nome, 141
 - permesso, 142
- filtro, schema, 95, 102, 190
- fine riga, carattere, 147
- fiore, 37
- flag, 112, 115
- float, funzione, 17
- float, tipo, 4
- floating-point, 4, 8, 189
- flusso di esecuzione, 21, 27, 58, 60, 65, 183, 203
- for, ciclo, 31, 44, 72, 93, 121, 190
- formato, 125
 - operatore di, 140, 203
 - sequenza di, 140
 - stringa di, 140
- formato, errore di, 124
- frame, 23, 27, 44, 57, 111
- Free Documentation License, GNU, v, vii
- frequenza, 107
 - di lettere, 125
 - di parole, 127, 137
- frustrazione, 206
- funzione, 3, 17, 19, 26, 166
 - abs, 52
 - ack, 61, 115
 - all, 192
 - any, 192
 - arco, 32
 - argomento di, 22
 - booleana, 55
 - cerchio, 32
 - chiamata di, 17, 26
 - choice, 128
 - comparsa, 52
 - composizione di, 54
 - deepcopy, 154
 - definizione di, 19, 20, 26
 - dict, 105
 - enumerate, 121
 - eval, 70
 - exists, 142
 - fattoriale, 56, 189
 - fibonacci, 58, 111
 - float, 17
 - frame di, 23, 27, 44, 57, 111
 - gamma, 59
 - getattr, 173
 - getcwd, 141
 - hasattr, 155, 172
 - input, 45
 - int, 17
 - instance, 59, 155, 171
 - len, 27, 72, 106
 - list, 96

- log, 18
- max, 119, 120
- min, 119, 120
- modificatore, 159
- motivi, 25
- open, 83, 84, 139, 142, 143
- parametro di, 22
- personalizzata, 22, 131
- poligono, 32
- popen, 145
- produttiva, 24, 26
- pura, 158, 162
- randint, 103, 128
- random, 128
- reload, 146, 201
- repr, 147
- reversed, 123
- ricorsiva, 44
- shuffle, 180
- sintassi, 166
- sorted, 101, 108, 123
- sqrt, 18, 53
- str, 18
- sum, 120, 191
- trigonometrica, 18
- trova, 75
- tupla come valore di ritorno, 119
- tuple, 117
- type, 155
- vuota, 24, 26
- zip, 120
- funzioni matematiche, 18
- generalizzazione, 33, 37, 85, 161
- generator expression, 191, 192, 198
- generatore, oggetto, 191
- gestire, 147
- get, metodo, 107
- getattr, funzione, 173
- getcwd, funzione, 141
- Giorno del Doppio, 163
- global, istruzione, 112, 115
- globale, variabile, 115
- GNU Free Documentation License, v, vii
- graffe, parentesi, 105
- grafico di chiamata, 111
- griglia, 28
- guardia, condizione di, 59, 61, 78
- HAS-A, relazione, 182, 185
- hasattr, funzione, 155, 172
- hash, funzione, 110, 114, 216
- hashing, 110, 114, 122
- Holmes, Sherlock, 25
- identico, 102
- identità, 98, 154
- if, istruzione, 41
- immutabilità, 74, 79, 99, 110, 117, 123
- impalcatura, 54, 61, 114
- impiallacciatura, 179, 185
- implementazione, 107, 114, 134, 173
- import, istruzione, 26, 146
- in, operatore, 77, 85, 92, 106, 214
- incapsulamento, 32, 37, 54, 69, 75, 181
 - dei dati, 183, 185
- incastro, parola, 104
- incremento, 64, 69, 159, 167
- indentazione, 19, 166, 200
- IndexError, 72, 78, 92, 204
- indice, 71, 72, 78, 79, 92, 105, 203
 - inizio da zero, 71, 92
 - negativo, 72
 - nei cicli, 86, 93
 - slicing, 73, 94
- indicizzazione, 212
- inefficienza, 211
- information hiding, 174
- init, metodo, 168, 172, 176, 178, 180
- inizializzazione
 - prima di aggiornare, 64
 - variabile, 69
- Input da tastiera, 45
- input, funzione, 45
- insieme, 132, 192
 - anagramma, 125, 148
- insieme di appartenenza, 115
- int, funzione, 17
- int, tipo, 4
- interfaccia, 34, 36, 37, 173, 183
- intero, 4, 7
- interprete, 2, 7
- intersezione, punto di, 218
- intestazione, 19, 26, 200
- invariante, 161, 162
- invocazione, 76, 80
- IOError, 142
- ipotenusa, 54
- is, operatore, 97, 154
- IS-A, relazione, 182, 185

- isinstance, funzione, 59, 155, 171
- istanza, 150, 155
 - attributo di, 150, 155, 176, 185
 - come argomento, 151
 - come valore di ritorno, 152
- istanziare, 150, 155
- istogramma, 107
 - frequenza delle parole, 129
 - scelta casuale, 129, 132
- istruzione, 15
 - assegnazione, 63
 - assert, 162
 - break, 66
 - composta, 41, 47
 - condizionale, 41, 47, 55, 190, 200
 - di assegnazione, 9
 - di stampa, 3, 7, 169, 204
 - for, 31, 72, 93
 - global, 112, 115
 - if, 41
 - import, 26, 146
 - pass, 41
 - raise, 109, 162
 - return, 44, 51, 206
 - try, 143, 155
 - while, 64
- items, metodo, 122
- iteratore, 121–123, 125, 213
- iterazione, 64, 69
- join, 213
- join, metodo, 97, 179
- KeyError, 106, 203, 215
- lavoro, directory di, 141
- len, funzione, 27, 72, 106
- letteralità, 5
- lettere
 - doppie, 88
 - frequenza, 125
 - rotazione, 115
- lineare, 218
- linguaggio
 - completo di Turing, 55
 - di alto livello, 7
 - di basso livello, 7
 - formale, 5, 8
 - naturale, 5, 8
 - orientato agli oggetti, 173
 - sicuro, 14
- Linux, 26
- lipogramma, 84
- Liskov, principio di sostituzione, 183
- list comprehension, 190, 198
- list, funzione, 96
- lista, 91, 96, 102, 123, 190
 - appartenenza, 92
 - attraversamento, 93
 - come argomento, 99
 - concatenamento, 93, 100, 103
 - copia, 94
 - di oggetti, 178
 - di tuple, 121
 - elemento, 92
 - indice, 92
 - metodi, 94, 213
 - nidificata, 91, 93, 102
 - operazione, 93
 - ripetizione, 93
 - slicing, 94
 - vuota, 91
- log, funzione, 18
- logaritmo, 137
- logico, operatore, 40
- lookup, 114
 - dizionario, 108
- lookup inverso, 114
- lookup inverso, dizionario, 108
- LookupError, 109
- loop variable, 190
- ls (comando Unix), 145
- lunghezza variabile, tupla di argomenti, 120
- macchina da scrivere a tartaruga, 38
- madre, classe, 180
- main, 23, 43, 112, 146
- Mano, classe, 180
- manutenzione, 173
- mappa, schema, 95, 102
- MappaHash, 216
- MappaLineare, 214
- MappaMigliore, 215
- mappatura, 114, 133
- Markov, analisi di, 133
- massimo comun divisore (MCD), 62
- Matplotlib, 137
- max, funzione, 119, 120
- mazzo, 175
- mazzo, carte da gioco, 178
- Mazzo, classe, 178

- McCloskey, Robert, 73
- MCD (massimo comun divisore), 62
- md5, 145
- md5, algoritmo, 148
- md5sum, 148
- memoizzazione, 111, 115
- metafora, invocazione di metodo, 167
- metatesi, 126
- metodi delle liste, 94
- metodo, 37, 76, 166, 174
 - `__lt__`, 177
 - `__str__`, 169, 178
 - `add`, 170
 - `append`, 94, 100, 103, 178, 179
 - `close`, 140, 144, 145
 - `count`, 80
 - `extend`, 94
 - `get`, 107
 - `init`, 168, 176, 178, 180
 - `items`, 122
 - `join`, 97, 179
 - `mro`, 183
 - `pop`, 96, 179
 - `radd`, 171
 - `read`, 145
 - `readline`, 83, 145
 - `remove`, 96
 - `replace`, 127
 - `setdefault`, 115
 - `sintassi`, 167
 - `sort`, 94, 101, 180
 - `split`, 97, 119
 - `stringa`, 80
 - `strip`, 84, 127
 - `translate`, 127
 - `update`, 122
 - `values`, 106
 - `vuoto`, 94
- metodo di Newton, 66
- Meyers, Chris, vii
- min, funzione, 119, 120
- minestrone, 134
- Moby Project, 83
- modalità interattiva, 11, 15, 24
- modalità script, 11, 15, 24
- modello di macchina, 209, 218
- modello mentale, 205
- modificatore, 159, 162
- modulo, 18, 26
 - bisect, 104
 - collections, 193, 194, 196
 - copy, 153
 - datetime, 163
 - dbm, 143
 - os, 141
 - pickle, 139, 144
 - pprint, 114
 - profile, 135
 - random, 103, 128, 180
 - reload, 146, 201
 - shelve, 144
 - string, 127
 - structshape, 124
- modulo, operatore, 39, 47
- modulo, scrittura, 145
- molteplicità (nel diagramma di classe), 182, 185
- Monty Python e il Sacro Graal, 158
- MP3, 148
- mro, metodo, 183
- multiinsieme, 193
- mutabilità, 74, 92, 94, 98, 113, 117, 123, 153
- name, variabile predefinita, 146
- namedtuple, 196
- NameError, 23, 203
- NaN, 189
- nidificata, lista, 93
- None, valore speciale, 24, 26, 52, 94, 96
- not, operatore, 40
- notazione a punto, 18, 27, 76
- numero casuale, 128
- O-grande, notazione, 211
- Obama, Barack, 209
- oggetto, 74, 79, 97, 98, 102
 - bytes, 143, 147
 - classe, 149, 150, 155, 196
 - contenuto, 174
 - copia, 153
 - Counter, 193
 - database, 143
 - defaultdict, 194
 - enumerate, 121
 - file, 83, 88
 - funzione, 20, 26, 27
 - generatore, 191
 - modulo, 18, 146
 - mutabile, 153
 - namedtuple, 196

- pipe, 148
- set, 192
- stampa, 166
- zip, 125
- oggetto contenuto (embedded), 152, 155
 - copia, 154
- oggetto mutabile, come valore di default, 174
- Olin College, vi
- omofono, 116
- open, funzione, 83, 84, 139, 142, 143
- operando, 15
- operator overloading, 170, 174, 177
- operatore, 7
 - and, 40
 - aritmetico, 3
 - bitwise, 4
 - booleano, 77
 - confronto, 40
 - del, 96
 - di aggiornamento, 95
 - di formato, 140, 147, 203
 - di slicing, 73, 80, 94, 100, 118
 - in, 77, 85, 92, 106
 - is, 97, 154
 - logico, 40
 - modulo, 39, 47
 - not, 40
 - or, 40
 - parentesi quadre, 71, 92, 118
 - relazionale, 177
 - stringa, 12
- operazione dominante, 210, 218
- opzionale, argomento, 80, 190
- or, operatore, 40
- ordinamento, 213
- ordine delle operazioni, 12, 15, 206
- ordine di risoluzione dei metodi, 183
- os, modulo, 141
- other (nome di parametro), 168
- OverflowError, 47
- overloading, 174
- palindromo, 62, 80, 87–89
- parametri, 23
- parametro, 22, 26, 99
 - opzionale, 131, 169
 - other, 168
 - raccolta, 120
 - self, 167
- parentesi
 - argomento in, 17
 - classe madre in, 180
 - corrispondenza, 14
 - graffe, 105
 - parametri in, 22, 23
 - tuple in, 117
 - vuote, 19, 76
- parentesi quadre, operatore, 71, 92, 118
- parola chiave, 10, 15, 200
 - def, 19
 - elif, 42
 - else, 41
- parola, frequenza, 127
- parola, riducibile, 116, 126
- parsing, 5, 8
- pass, istruzione, 41
- pdb (Python debugger), 204
- peggior bug, 174
- PEMDAS, 12
- percorso, 141, 147
 - assoluto, 141, 147
 - relativo, 141, 147
- permesso, accesso a file, 142
- persistenza, 139, 147
- pi, 18, 70
- pickle, modulo, 139, 144
- pickling, 144
- pipe, 145
- pipe, oggetto, 148
- poesia, 6
- poker, 175, 186
- poligono, funzione, 32
- polimorfismo, 172, 174
- pop, metodo, 96, 179
- popen, funzione, 145
- portabilità, 7
- posizionale, argomento, 168, 174, 197
- postcondizione, 36, 37, 60, 183
- potenziata, assegnazione, 95
- pprint, modulo, 114
- precedenza, 206
- precondizione, 36, 37, 60, 183
- prefisso, 133
- print, funzione, 3
- profile, modulo, 135
- profonda, copia, 154
- progettazione
 - orientata agli oggetti, 173
- Progetto Gutenberg, 127

- programma, 1, 7
 - test, 87
- programmazione
 - a tentoni, 206
 - orientata agli oggetti, 149, 165, 174, 180
- prompt, 3, 7, 46
- prosa, 6
- prototipo ed evoluzioni, 158, 160, 162
- pseudocasuale, 128, 137
- punto di intersezione, 211
- Punto, classe, 150, 169
- punto, matematico, 149
- Puzzler, 88, 89, 115, 126
- Python
 - avvio, 2
- Python 2, 2, 3, 33, 40, 45
- Python in un browser, 2
- PythonAnywhere, 2
- quadratico, 218
- quesito, 88
- rabbia, 206
- raccolta, 120, 125, 197
- radd, metodo, 171
- radiante, 18
- radice quadrata, 66
- radix sort, 209
- raise, istruzione, 109, 162
- Ramanujan, Srinivasa, 70
- ramificazione, 41, 48
- randint, funzione, 103, 128
- random, funzione, 128
- random, modulo, 103, 128, 180
- rappresentazione, 149, 151, 175
 - di stringa, 169
- RB-albero, 215
- read, metodo, 145
- readline, metodo, 83, 145
- refactoring, 34, 35, 37, 184
- rehashing, 217
- reload, funzione, 146, 201
- remove, metodo, 96
- replace, metodo, 127
- repr, funzione, 147
- Rettangolo, classe, 151
- return, istruzione, 44, 51, 206
- reversed, funzione, 123
- riassegnazione, 63, 69, 92, 112
- ricerca, 109, 214, 218
 - binaria, 103, 214
 - lineare, 214
 - schema, 192
 - schema di, 75, 79, 85
- ricorsione, 43, 44, 48, 55, 57
 - caso base, 44
 - infinita, 45, 48, 58, 201, 202
- ridimensionamento geometrico, 218
- ridondanza, 5
- riducibile, parola, 116
- riduzione ad un problema già risolto, 86–88
- riduzione, schema, 95, 102
- riferimento, 98, 99, 102
 - alias, 98
- ripetizione, 30
 - lista, 93
- risoluzione di problemi, 1
- ritorno a capo, 46, 179
- rotazione
 - lettere, 115
- rotazione di lettere, 81
- runtime
 - errore di, 199
- RuntimeError, 45, 58
- salto sulla fiducia, 57
- scambio, schema di, 118
- Scarabeo, 125
- schema
 - di ricerca, 75, 79, 85, 109
 - di scambio, 118
 - filtro, 95, 102, 190
 - guardiani, 59, 78
 - mappa, 95, 102
 - riduzione, 95, 102
- schema di ricerca, 192
- Schmidt, Eric, 209
- script, 11, 15
- self (nome di parametro), 167
- semantica, 14, 15, 166
 - errore di, 14, 15, 199, 204
- seme, 175
- seno, funzione, 18
- sequenza, 5, 71, 79, 91, 96, 117, 123
 - di formato, 140, 147
- serie, condizioni in, 42
- sessagesimale, 160
- setdefault, 195
- setdefault, metodo, 115
- shallow, copia, 154

- shell, 145, 148
- shelve, modulo, 144
- shuffle, funzione, 180
- simbolo, 5, 8
- singleton, 110, 114, 117
- sintassi, 8, 14, 166, 200
 - errore di, 14, 15, 199
- slice, 79
- slicing
 - aggiornamento, 94
 - copia, 74, 94
 - lista, 94
 - operatore di, 73, 80, 94, 100, 118
 - stringa, 73
 - tupla, 118
- smistamento in base al tipo, 171, 174
- soggetto, 167, 174
- soluzione di problemi, 7
- sort, metodo, 94, 101, 180
- sorted
 - funzione, 101
- sorted, funzione, 108, 123
- sottoinsieme, 193
- sottrazione
 - con prestito, 68, 161
 - di dizionari, 192
 - di insiemi, 192
 - dizionario, 131
- sovrascrittura, 131, 137, 169, 177, 180, 183
- spacchettamento, 120, 125, 197
- spazi, 200
- spaziatore, 46, 84, 146
- spirale, 38
- spirale di Archimede, 38
- split, metodo, 97, 119
- sqrt, funzione, 18, 53
- stable sort, 213
- stampa, istruzione di, 3, 7, 169, 204
- step, ampiezza, 80
- stile di programmazione funzionale, 160, 162
- StopIteration, 191
- __str__, metodo, 169, 178
- str, funzione, 18
- string, modulo, 127
- string, tipo, 4
- stringa, 4, 8, 96, 123
 - a righe multiple, 36, 200
 - accumulatore, 179
 - concatenamento, 212
 - confronto, 77
 - di documentazione, 36
 - di formato, 140, 147
 - documentazione, 37
 - immutabile, 74
 - metodi, 76, 80, 212
 - operazioni, 12
 - rappresentazione, 147
 - slicing, 73
 - triple virgolette, 36
 - vuota, 79, 97
- strip, metodo, 84, 127
- structshape, modulo, 124
- struttura, 5
- struttura di dati, 124, 125, 134
- suffisso, 133
- sum, funzione, 120, 191
- sviluppo incrementale, 61, 199
- sviluppo pianificato, 160, 162
- SyntaxError, 19
- tabella hash, 106, 114, 214, 218
- tecnica di sviluppo, 37
 - incapsulamento dei dati, 183, 185
 - incapsulamento e generalizzazione, 35
 - incrementale, 52, 199
 - pianificato, 160
 - programmazione a tentoni, 136, 206
 - prototipo ed evoluzioni, 158, 160
 - riduzione, 86–88
- tempo costante, 217
- Tempo, classe, 157
- teorema di Pitagora, 52
- Tesi di Turing, 55
- test
 - caso di prova minimo, 204
 - difficoltà, 87
 - e assenza di bug, 88
 - salto sulla fiducia, 58
 - sapere il risultato, 53
 - sviluppo incrementale, 52
- testo
 - casuale, 133
 - semplice, 83, 127
- tipo, 4
 - bool, 40
 - dict, 105
 - file, 139
 - float, 4
 - function, 20

- int, 4
- list, 91
- personalizzato, 149, 155, 157, 166, 170, 177
- set, 132
- str, 4
- tuple, 117
- token, 5, 8
- torta, 38
- traceback, 24, 27, 45, 46, 109, 203
- translate, metodo, 127
- triangolo, 48
- trova, funzione, 75
- True, valore speciale, 40
- try, istruzione, 143, 155
- tupla, 117, 119, 123, 124
 - assegnazione, 118, 119, 121, 125
 - come chiave di dizionario, 122, 134
 - con nome, 196
 - confronto, 178
 - in parentesi quadre, 122
 - metodi, 212
 - singleton, 117
 - slicing, 118
- tuple, funzione, 117
- Turing, Alan, 55
- turtle, 49
- type, 7
- type, funzione, 155
- TypeError, 72, 74, 110, 118, 120, 141, 168, 203
- uguaglianza e assegnazione, 63
- UnboundLocalError, 113
- underscore, carattere, 10
- unicità, 103
- update, metodo, 122
- uso prima di def, 21
- valore, 4, 7, 97, 98, 114
 - tupla, 119
- valore (delle carte da gioco), 175
- valore di default, 131, 169
 - evitare i mutabili, 174
- valore di ritorno, 17, 26, 51, 152
 - tupla, 119
- valore speciale
 - False, 40
 - None, 24, 26, 52, 94, 96
 - True, 40
- ValueError, 46, 119
- values, metodo, 106
- valutazione, 10, 11, 15
- variabile, 9, 14
 - aggiornamento, 64
 - globale, 112
 - aggiornamento, 113
 - locale, 23, 26
 - temporanea, 51, 61, 206
- virgola mobile, 67
- virgolette, 36, 200
- vorpale, 56
- zero, indice iniziale, 71, 92
- zip, funzione, 120
 - uso con dict, 122
- zip, oggetto, 125
- Zipf, legge di, 137

