Think Python

How to Think Like a Computer Scientist

Έκδοση 1.0.5
Δεκέμβριος 2014

Think Python

Πώς να Σκέφτεσαι σαν Επιστήμονας της Πληροφορικής

Version 1.0.5

Δεκέμβριος 2014

Allen Downey
Μετάφραση-Επιμέλεια: Ποικιλίδης Ζαχαρίας
Επιβλέπων: Δρ. Βλάχος Βασίλειος

TEI of Larisa

Πρόλογος

Λίγα λόγια για το βιβλίο

Το βιβλίο αυτό αποτελεί μετάφραση του αγγλικού βιβλίου Think Python του καθηγητή Allen B. Downey από το κολέγιο εφαρμοσμένης μηχανικής Franklin W. Olin. Η μετάφραση διεκπεραιώθηκε στα πλαίσια της πτυχιακής μου εργασίας στο Τμήμα Μηχανικών Πληροφορικής Τ.Ε. του ΑΤΕΙ Λάρισας.

Η εργασία αυτή μου έδωσε την ευκαιρία να εργαστώ με το LaTeX και να μάθω τα βασικά για τη δημιουργία ενός εγγράφου. Για την επεξεργασία του πηγαίου κώδικα του βιβλίου χρησιμοποιήθηκε το εργαλείο texmaker τόσο σε περιβάλλον Linux (ubuntu 13.04/13.10) όσο και σε Windows 7. Αρχικά χρησιμοποίησα το πακέτο babel για να την παράλληλη χρήση ελληνικών και λατινικών χαρακτήρων αλλά στην πορεία άλλαξα την μηχανή σε XeLaTeX γιατί το XeLaTeX έχει εγγενή υποστήριξη για unicode.

Αυτή είναι η πρώτη έκδοση του βιβλίου στα ελληνικά και είναι βασισμένη στην 2.0.13 του αρχικού βιβλίου. Ο πηγαίος κώδικας του ελληνικού βιβλίου είναι διαθέσιμος στο σύνδεσμο `....` Κάθε διόρθωση η παρατήρηση πάνω σε αυτόν τον κώδικα είναι ευπρόσδεκτη και θα εκτιμηθεί ιδιαίτερα.

Το βιβλίο πραγματεύεται τη γλώσσα προγραμματισμού **Python** αλλά επικεντρώνεται περισσότερο στον προγραμματισμό και την επίλυση προβλημάτων παρά στη γλώσσα αυτή κάθε αυτή.

Μερικά από τα χαρακτηριστικά στοιχεία αυτού του βιβλίου είναι:

- Στο τέλος κάθε κεφαλαίου υπάρχει μία ενότητα σχετικά με την αποσφαλμάτωση. Αυτές οι ενότητες παρουσιάζουν κάποιες γενικές τεχνικές για την εύρεση και την αποφυγή σφαλμάτων σχετικά με τις παγίδες της **Python**.
- Κάθε κεφάλαιο έχει ασκήσεις η οποίες κυμαίνονται από μικρά τεστ κατανόησης μέχρι και κάποια ουσιαστικά προγράμματα και υπάρχουν λύσεις για τις περισσότερες από αυτές.
- Υπάρχει μία σειρά από μελέτες περιπτώσεων, οι οποίες είναι μεγαλύτερα παραδείγματα με ασκήσεις, λύσεις και συζήτηση. Μερικές είναι βασισμένες στη Swampy, μία σουίτα από προγράμματα Python την επιμέλεια της οποίας έχει ο καθηγητής Allen B. Downey. Η Swampy, παραδείγματα κώδικα και κάποιες λύσεις είναι διαθέσιμες στον σύνδεσμο `http://thinkpython.com`.
- Κατά την έκταση του βιβλίου λαμβάνει χώρα μία εκτενής συζήτηση σχετικά με τα σχέδια ανάπτυξης ενός προγράμματος και τα βασικά σχεδιαστικά πρότυπα.
- Στο τέλος του βιβλίου υπάρχουν παραρτήματα σχετικά με την αποσφαλμάτωση, την ανάλυση αλγορίθμων και και τα διαγράμματα UML με το πακέτο Lumpy.

Εύχομαι να σας αρέσει το αποτέλεσμα και να σας βοηθήσει να μάθετε πως να σκέφτεστε και πως να προγραμματίζετε σαν επιστήμονες της πληροφορικής.

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον καθηγητή Allen Downey για την επιλογή του να κυκλοφορήσει αυτό το βιβλίο κάτω από την Άδεια Ελεύθερης Τεκμηρίωσης GNU με αποτέλεσμα να μου δώσει την δυνατότητα να μεταφράσω αυτό το βιβλίο στα Ελληνικά.

Επίσης θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή μου Βλάχο Βασίλειο γιατί μου έδωσε την ευκαιρία να ασχοληθώ με αυτό το βιβλίο στα πλαίσια της πτυχιακής μου εργασίας. Ελπίζω αυτό το βιβλίο να αποτελέσει ένα χρήσιμο εργαλείο εκμάθησης προγραμματισμού στο Τμήμα Μηχανικών Πληροφορικής του ΤΕΙ της Λάρισας και γιατί όχι και αλλού.

Τέλος θα ήθελα να ευχαριστήσω τους γονείς μου Βασίλη και Φωτεινή, που με στήριξαν όλα αυτά τα χρόνια καθώς επίσης και την κοπέλα μου Ελεωνόρα για την πολύτιμη βοήθειά της στην μετάφραση αρκετών δυσνόητων σημείων του βιβλίου.

Λίστα συνεισφερόντων του αρχικού βιβλίου

Περισσότεροι από από 100 προσεκτικοί αναγνώστες έχουν στείλει προτάσεις και διορθώσεις τα περασμένα χρόνια για το αρχικό βιβλίο στα Αγγλικά. Αυτή είναι μια λίστα με σχεδόν όλους όσους βοήθησαν στην ανάπτυξη και τελειοποίηση του αγγλικού βιβλίου.

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken `catTwice` function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word “unconsciously” in Chapter 1 needed to be changed to “subconsciously”.
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.

- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the `increment` function in Chapter 13.
- John Ouzts corrected the definition of "return value" in Chapter 3.
- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the printTime function was used in an example without being defined.
- Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX.
- Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have made numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.
- Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.
- Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between *gleich* and *selbe*.
- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.
- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.
- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and helped us update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.
- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
- Julie Peters caught a typo in the Preface.
- Florin Oprina sent in an improvement in `makeTime`, a correction in `printTime`, and a nice typo.
- D. J. Webre suggested a clarification in Chapter 3.
- Ken found a fistful of errors in Chapters 8, 9 and 11.
- Ivo Wever caught a typo in Chapter 5 and suggested a clarification in Chapter 3.

- Curtis Yanko suggested a clarification in Chapter 2.
- Ben Logan sent in a number of typos and problems with translating the book into HTML.
- Jason Armstrong saw the missing word in Chapter 2.
- Louis Cordier noticed a spot in Chapter 16 where the code didn't match the text.
- Brian Cain suggested several clarifications in Chapters 2 and 3.
- Rob Black sent in a passel of corrections, including some changes for Python 2.2.
- Jean-Philippe Rey at Ecole Centrale Paris sent a number of patches, including some updates for Python 2.2 and other thoughtful improvements.
- Jason Mader at George Washington University made a number of useful suggestions and corrections.
- Jan Gundtofte-Bruun reminded us that "a error" is an error.
- Abel David and Alexis Dinno reminded us that the plural of "matrix" is "matrices", not "matrixes". This error was in the book for years, but two readers with the same initials reported it on the same day. Weird.
- Charles Thayer encouraged us to get rid of the semi-colons we had put at the ends of some statements and to clean up our use of "argument" and "parameter".
- Roger Sperberg pointed out a twisted piece of logic in Chapter 3.
- Sam Bull pointed out a confusing paragraph in Chapter 2.
- Andrew Cheung pointed out two instances of "use before def."
- C. Corey Capel spotted the missing word in the Third Theorem of Debugging and a typo in Chapter 4.
- Alessandra helped clear up some Turtle confusion.
- Wim Champagne found a brain-o in a dictionary example.
- Douglas Wright pointed out a problem with floor division in `arc`.
- Jared Spindor found some jetsam at the end of a sentence.
- Lin Peiheng sent a number of very helpful suggestions.
- Ray Hagtvedt sent in two errors and a not-quite-error.
- Torsten Hübsch pointed out an inconsistency in Swampy.
- Inga Petuhhov corrected an example in Chapter 14.
- Arne Babenhauserheide sent several helpful corrections.
- Mark E. Casida is is good at spotting repeated words.
- Scott Tyler filled in a that was missing. And then sent in a heap of corrections.
- Gordon Shephard sent in several corrections, all in separate emails.
- Andrew Turner `spot`ted an error in Chapter 8.
- Adam Hobart fixed a problem with floor division in `arc`.
- Daryl Hammond and Sarah Zimmerman pointed out that I served up `math.pi` too early. And Zim spotted a typo.
- George Sass found a bug in a Debugging section.
- Brian Bingham suggested Exercise 11.10.
- Leah Engelbert-Fenton pointed out that I used `tuple` as a variable name, contrary to my own advice. And then found a bunch of typos and a "use before def."
- Joe Funke spotted a typo.
- Chao-chao Chen found an inconsistency in the Fibonacci example.

- Jeff Paine knows the difference between space and spam.
- Lubos Pintes sent in a typo.
- Gregg Lind and Abigail Heithoff suggested Exercise 14.4.
- Max Hailperin has sent in a number of corrections and suggestions. Max is one of the authors of the extraordinary *Concrete Abstractions*, which you might want to read when you are done with this book.
- Chotipat Pornavalai found an error in an error message.
- Stanislaw Antol sent a list of very helpful suggestions.
- Eric Pashman sent a number of corrections for Chapters 4–11.
- Miguel Azevedo found some typos.
- Jianhua Liu sent in a long list of corrections.
- Nick King found a missing word.
- Martin Zuther sent a long list of suggestions.
- Adam Zimmerman found an inconsistency in my instance of an "instance" and several other errors.
- Ratnakar Tiwari suggested a footnote explaining degenerate triangles.
- Anurag Goel suggested another solution for `is_abecedarian` and sent some additional corrections. And he knows how to spell Jane Austen.
- Kelli Kratzer spotted one of the typos.
- Mark Griffiths pointed out a confusing example in Chapter 3.
- Roydan Ongie found an error in my Newton's method.
- Patryk Wolowiec helped me with a problem in the HTML version.
- Mark Chonofsky told me about a new keyword in Python 3.
- Russell Coleman helped me with my geometry.
- Wei Huang spotted several typographical errors.
- Karen Barber spotted the the oldest typo in the book.
- Nam Nguyen found a typo and pointed out that I used the Decorator pattern but didn't mention it by name.
- Stéphane Morin sent in several corrections and suggestions.
- Paul Stoop corrected a typo in `uses_only`.
- Eric Bronner pointed out a confusion in the discussion of the order of operations.
- Alexandros Gezerlis set a new standard for the number and quality of suggestions he submitted. We are deeply grateful!
- Gray Thomas knows his right from his left.
- Giovanni Escobar Sosa sent a long list of corrections and suggestions.
- Alix Etienne fixed one of the URLs.
- Kuang He found a typo.
- Daniel Neilson corrected an error about the order of operations.
- Will McGinnis pointed out that `polyline` was defined differently in two places.
- Swarup Sahoo spotted a missing semi-colon.
- Frank Hecker pointed out an exercise that was under-specified, and some broken links.
- Animesh B helped me clean up a confusing example.

- Martin Caspersen found two round-off errors.
- Gregor Ulm sent several corrections and suggestions.
- Dimitrios Tsirigkas suggested I clarify an exercise.
- Carlos Tafur sent a page of corrections and suggestions.
- Martin Nordsletten found a bug in an exercise solution.
- Lars O.D. Christensen found a broken reference.
- Victor Simeone found a typo.
- Sven Hoexter pointed out that a variable named `input` shadows a build-in function.
- Viet Le found a typo.
- Stephen Gregory pointed out the problem with `cmp` in Python 3.
- Matthew Shultz let me know about a broken link.
- Lokesh Kumar Makani let me know about some broken links and some changes in error messages.
- Ishwar Bhat corrected my statement of Fermat's last theorem.

Περιεχόμενα

Πρόλογος **v**

1 Ο τρόπος του προγράμματος **1**
1.1 Η γλώσσα προγραμματισμού Python . . . 1
1.2 Τι είναι ένα πρόγραμμα . . . 3
1.3 Τι είναι η αποσφαλμάτωση; . . . 3
1.3.1 Συντακτικά λάθη . . . 4
1.3.2 Λάθη χρόνου εκτέλεσης . . . 4
1.3.3 Λογικά λάθη . . . 4
1.3.4 Πειραματική αποσφαλμάτωση . . . 4
1.4 Φυσικές και τυπικές γλώσσες . . . 5
1.5 Το πρώτο πρόγραμμα . . . 7
1.6 Αποσφαλμάτωση . . . 7
1.7 Ορολογία . . . 8
1.8 Ασκήσεις . . . 9

2 Μεταβλητές, εκφράσεις και δηλώσεις **11**
2.1 Τιμές και τύποι . . . 11
2.2 Μεταβλητές . . . 12
2.3 Ονόματα μεταβλητών και λέξεις κλειδιά . . . 13
2.4 Τελεστές και τελεστέοι . . . 13
2.5 Εκφράσεις και δηλώσεις . . . 14
2.6 Διαδραστική λειτουργία και λειτουργία σεναρίων . . . 14
2.7 Η σειρά των πράξεων . . . 15
2.8 Πράξεις συμβολοσειρών . . . 16
2.9 Σχόλια . . . 16
2.10 Αποσφαλμάτωση . . . 17
2.11 Ορολογία . . . 18
2.12 Ασκήσεις . . . 19

3 Συναρτήσεις **21**
3.1 Κλήσεις συναρτήσεων . . . 21
3.2 Συναρτήσεις μετατροπής τύπων . . . 21
3.3 Μαθηματικές Συναρτήσεις . . . 22
3.4 Σύνθεση . . . 23
3.5 Προσθέτοντας νέες συναρτήσεις . . . 23
3.6 Ορισμοί και χρήσεις . . . 25
3.7 Ροή εκτέλεσης . . . 25
3.8 Παράμετροι και ορίσματα . . . 26

3.9 Οι μεταβλητές και οι παράμετροι είναι τοπικές . . . 27
3.10 Διαγράμματα στοίβας . . . 27
3.11 Γόνιμες και κενές συναρτήσεις . . . 28
3.12 Γιατί συναρτήσεις . . . 29
3.13 Εισαγωγή από μονάδα λογισμικού με from . . . 29
3.14 Αποσφαλμάτωση . . . 30
3.15 Ορολογία . . . 31
3.16 Ασκήσεις . . . 32

4 Μελέτη περίπτωσης: σχεδίαση διεπαφής 35
4.1 TurtleWorld . . . 35
4.2 Απλή επανάληψη . . . 36
4.3 Ασκήσεις . . . 37
4.4 Ενθυλάκωση . . . 38
4.5 Γενίκευση . . . 39
4.6 Σχεδίαση διεπαφής . . . 39
4.7 Ανακατασκευή κώδικα . . . 40
4.8 Πλάνο ανάπτυξης . . . 41
4.9 Συμβολοσειρά τεκμηρίωσης . . . 42
4.10 Αποσφαλμάτωση . . . 42
4.11 Ορολογία . . . 43
4.12 Ασκήσεις . . . 43

5 Δηλώσεις υπό συνθήκη και αναδρομή 45
5.1 Τελεστής υπολογισμού υπολοίπου ακέραιας διαίρεσης . . . 45
5.2 Λογικές εκφράσεις . . . 45
5.3 Λογικοί τελεστές . . . 46
5.4 Εκτέλεση υπό συνθήκη . . . 46
5.5 Εναλλακτική εκτέλεση . . . 47
5.6 Αλυσιδωτές συνθήκες . . . 47
5.7 Εμφωλευμένες συνθήκες . . . 48
5.8 Αναδρομή . . . 48
5.9 Διαγράμματα στοίβας για αναδρομικές συναρτήσεις . . . 50
5.10 Άπειρη αναδρομή . . . 50
5.11 Είσοδος από το πληκτρολόγιο . . . 51
5.12 Αποσφαλμάτωση . . . 52
5.13 Ορολογία . . . 53
5.14 Ασκήσεις . . . 53

6 Γόνιμες Συναρτήσεις 57
6.1 Επιστρεφόμενες τιμές . . . 57
6.2 Σταδιακή ανάπτυξη . . . 58
6.3 Σύνθεση . . . 60
6.4 Λογικές συναρτήσεις . . . 61
6.5 Περισσότερη αναδρομή . . . 61
6.6 Άλμα πίστης . . . 63
6.7 Ένα ακόμα παράδειγμα . . . 64
6.8 Έλεγχος τύπων . . . 64
6.9 Αποσφαλμάτωση . . . 65
6.10 Ορολογία . . . 67
6.11 Ασκήσεις . . . 67

7 Επανάληψη **69**
7.1 Πολλαπλή εκχώρηση . . . 69
7.2 Ενημέρωση μεταβλητών . . . 70
7.3 Η δήλωση `while` . . . 70
7.4 Η δήλωση `break` . . . 72
7.5 Τετραγωνικές ρίζες . . . 72
7.6 Αλγόριθμοι . . . 74
7.7 Αποσφαλμάτωση . . . 74
7.8 Ορολογία . . . 75
7.9 Ασκήσεις . . . 75

8 Συμβολοσειρές **77**
8.1 Μία συμβολοσειρά είναι μία ακολουθία . . . 77
8.2 Η δήλωση `len` . . . 78
8.3 Διάσχιση με `for` . . . 78
8.4 Τεμάχια συμβολοσειράς . . . 79
8.5 Οι συμβολοσειρές είναι αμετάβλητες . . . 80
8.6 Αναζήτηση . . . 80
8.7 Επανάληψη και καταμέτρηση . . . 81
8.8 Μέθοδοι συμβολοσειρών . . . 81
8.9 Τελεστής `in` . . . 83
8.10 Σύγκριση συμβολοσειρών . . . 83
8.11 Αποσφαλμάτωση . . . 84
8.12 Ορολογία . . . 85
8.13 Ασκήσεις . . . 86

9 Μελέτη περίπτωσης: λογοπαίγνια **89**
9.1 Διαβάζοντας λίστες λέξεων . . . 89
9.2 Ασκήσεις . . . 90
9.3 Αναζήτηση . . . 91
9.4 Βρόχοι επανάληψης με δείκτες . . . 92
9.5 Αποσφαλμάτωση . . . 93
9.6 Ορολογία . . . 94
9.7 Ασκήσεις . . . 94

10 Λίστες **97**
10.1 Η λίστα είναι μία ακολουθία . . . 97
10.2 Οι λίστες είναι μεταβλητές . . . 97
10.3 Διασχίζοντας μία λίστα . . . 99
10.4 Πράξεις με λίστες . . . 99
10.5 Λίστες και τεμάχια . . . 99
10.6 Μέθοδοι λιστών . . . 100
10.7 Map, filter και reduce . . . 101
10.8 Διαγραφή στοιχείων . . . 102
10.9 Λίστες και συμβολοσειρές . . . 103
10.10 Αντικείμενα και τιμές . . . 104
10.11 Ψευδώνυμα . . . 105
10.12 Ορίσματα λίστας . . . 106
10.13 Αποσφαλμάτωση . . . 107
10.14 Ορολογία . . . 108
10.15 Ασκήσεις . . . 109

11 Λεξικά **111**
11.1 Το λεξικό ως ένα σύνολο από μετρητές . . . 112
11.2 Λεξικά και βρόχοι . . . 114
11.3 Αντίστροφη αναζήτηση . . . 114
11.4 Λεξικά και λίστες . . . 115
11.5 Σημείωμα . . . 117
11.6 Καθολικές μεταβλητές . . . 118
11.7 Ακέραιοι μεγάλου μήκους . . . 119
11.8 Αποσφαλμάτωση . . . 120
11.9 Ορολογία . . . 121
11.10 Ασκήσεις . . . 121

12 Πλειάδες **123**
12.1 Οι πλειάδες είναι αμετάβλητες . . . 123
12.2 Εκχώρηση πλειάδων . . . 124
12.3 Οι πλειάδες σαν επιστρεφόμενες τιμές . . . 125
12.4 Οι πλειάδες σαν ορίσματα μεταβλητού μήκους . . . 125
12.5 Λίστες και πλειάδες . . . 126
12.6 Λεξικά και πλειάδες . . . 127
12.7 Συγκρίνοντας πλειάδες . . . 129
12.8 Ακολουθίες ακολουθιών . . . 130
12.9 Αποσφαλμάτωση . . . 130
12.10 Ορολογία . . . 131
12.11 Ασκήσεις . . . 132

13 Μελέτη περίπτωσης: επιλογή δομής δεδομένων **135**
13.1 Ανάλυση συχνότητας λέξεων . . . 135
13.2 Τυχαίοι αριθμοί . . . 136
13.3 Ιστόγραμμα λέξεων . . . 137
13.4 Οι πιο συνηθέστερες λέξεις . . . 138
13.5 Προαιρετικές παράμετροι . . . 139
13.6 Αφαίρεση λεξικών . . . 139
13.7 Τυχαίες λέξεις . . . 140
13.8 Ανάλυση Μαρκόφ . . . 141
13.9 Δομές δεδομένων . . . 142
13.10 Αποσφαλμάτωση . . . 143
13.11 Ορολογία . . . 144
13.12 Ασκήσεις . . . 145

14 Αρχεία **147**
14.1 Διάρκεια . . . 147
14.2 Διάβασμα και γράψιμο . . . 147
14.3 Τελεστής διαμόρφωσης . . . 148
14.4 Ονόματα αρχείων και διαδρομές . . . 149
14.5 Πιάσιμο εξαιρέσεων . . . 150
14.6 Βάσεις δεδομένων . . . 151
14.7 Σειριοποίηση . . . 152
14.8 Σωληνώσεις . . . 153
14.9 Γράψιμο αρθρωμάτων . . . 154
14.10 Αποσφαλμάτωση . . . 155
14.11 Ορολογία . . . 156

14.12 Ασκήσεις 156

15 Κλάσεις και αντικείμενα **157**
15.1 Τύποι ορισμένοι από το χρήστη 157
15.2 Ιδιότητες 158
15.3 Ορθογώνια παραλληλόγραμμα 159
15.4 Τα στιγμιότυπα σαν επιστρεφόμενες τιμές 160
15.5 Τα αντικείμενα είναι μεταβλητά 160
15.6 Αντιγραφή 161
15.7 Αποσφαλμάτωση 163
15.8 Ορολογία 163
15.9 Ασκήσεις 164

16 Κλάσεις και συναρτήσεις **167**
16.1 Ώρα 167
16.2 Αγνές συναρτήσεις 168
16.3 Συναρτήσεις τροποποίησης 169
16.4 Πρωτοτυποποίηση εναντίον σχεδιασμού 170
16.5 Αποσφαλμάτωση 171
16.6 Ορολογία 172
16.7 Ασκήσεις 172

17 Κλάσεις και μέθοδοι **175**
17.1 Αντικειμενοστραφή χαρακτηριστικά 175
17.2 Εκτύπωση αντικειμένων 176
17.3 Ένα ακόμη παράδειγμα 177
17.4 Ένα πιο σύνθετο παράδειγμα 178
17.5 Η μέθοδος init 178
17.6 Η μέθοδος `__str__` 179
17.7 Υπερφόρτωση τελεστών 179
17.8 Αποστολή βάση τύπου 180
17.9 Πολυμορφισμός 181
17.10 Αποσφαλμάτωση 182
17.11 Διεπαφή και υλοποίηση 183
17.12 Ορολογία 183
17.13 Ασκήσεις 184

18 Κληρονομικότητα **187**
18.1 Αντικείμενα τραπουλόχαρτων 187
18.2 Ιδιότητες κλάσεων 188
18.3 Συγκρίνοντας τραπουλόχαρτα 189
18.4 Τράπουλες 190
18.5 Τύπωση τράπουλας 191
18.6 Προσθήκη, αφαίρεση, ανακάτεμα και ταξινόμηση 191
18.7 Κληρονομικότητα 192
18.8 Διαγράμματα κλάσεων 194
18.9 Αποσφαλμάτωση 195
18.10 Ενθυλάκωση δεδομένων 196
18.11 Ορολογία 197
18.12 Ασκήσεις 197

19 Tkinter **201**

19.1 Γραφική διασύνδεση χρήστη . . . 201
19.2 Κουμπιά και επιστροφές κλήσεων . . . 202
19.3 Γραφικά στοιχεία Καμβά . . . 203
19.4 Ακολουθίες συντεταγμένων . . . 204
19.5 Περισσότερα γραφικά στοιχεία . . . 204
19.6 Πακετάρισμα γραφικών στοιχείων . . . 205
19.7 Μενού και αντικείμενα με δυνατότητα κλήσης . . . 208
19.8 Δεσμοί . . . 209
19.9 Αποσφαλμάτωση . . . 211
19.10 Ορολογία . . . 212
19.11 Ασκήσεις . . . 212

Α΄ Αποσφαλμάτωση **215**
Α΄.1 Συντακτικά λάθη . . . 215
Α΄.1.1 Συνεχίζω να κάνω αλλαγές αλλά δεν υπάρχει διαφορά. . . . 216
Α΄.2 Λάθη χρόνου εκτέλεσης . . . 217
Α΄.2.1 Το πρόγραμμά μου δεν κάνει απολύτως τίποτα. . . . 217
Α΄.2.2 Το πρόγραμμά μου κρεμάει. . . . 217
Α΄.2.3 Παίρνω μία εξαίρεση όταν τρέχω το πρόγραμμα. . . . 218
Α΄.2.4 Πρόσθεσα πολλές δηλώσεις `print` και πελάγωσα με την έξοδο. . . . 219
Α΄.3 Σημασιολογικά λάθη . . . 220
Α΄.3.1 Το πρόγραμμά μου δεν δουλεύει. . . . 220
Α΄.3.2 Έχω μία μεγάλη έκφραση η οποία δεν κάνει αυτό που θα περίμενα. . . . 221
Α΄.3.3 Έχω μία μέθοδο ή μία συνάρτηση η οποία δεν επιστρέφει αυτό που θα περίμενα. . . . 222
Α΄.3.4 Έχω κολλήσει πραγματικά και χρειάζομαι βοήθεια. . . . 222
Α΄.3.5 Όχι, χρειάζομαι πραγματικά βοήθεια. . . . 222

Β΄ Ανάλυση Αλγορίθμων **225**
Β΄.1 Τάξη αύξησης . . . 226
Β΄.2 Ανάλυση των βασικών πράξεων της γλώσσας . . . 228
Β΄.3 Ανάλυση των αλγόριθμων αναζήτησης . . . 229
Β΄.4 Πίνακες κατακερματισμού . . . 230

Γ΄ Lumpy **235**
Γ΄.1 Διάγραμμα κατάστασης . . . 236
Γ΄.2 Διάγραμμα στοίβας . . . 237
Γ΄.3 Διαγράμματα αντικειμένων . . . 237
Γ΄.4 Αντικείμενα συναρτήσεων και κλάσεων . . . 240
Γ΄.5 Διαγράμματα Κλάσεων . . . 241

Κεφάλαιο 1

Ο τρόπος του προγράμματος

Στόχος αυτού του βιβλίου είναι να σας διδάξει πώς να σκέφτεστε σαν επιστήμονες της πληροφορικής. Αυτός ο τρόπος σκέψης συνδυάζει κάποια από τα καλύτερα χαρακτηριστικά των μαθηματικών, της μηχανικής και της φυσικής επιστήμης. Όπως οι μαθηματικοί, έτσι και οι επιστήμονες της πληροφορικής χρησιμοποιούν τυπικές γλώσσες (formal) για να υποδηλώσουν ιδέες (και ειδικότερα υπολογισμούς). Όπως οι μηχανικοί, σχεδιάζουν πράγματα, συνθέτουν επιμέρους μέρη σε συστήματα και αξιολογούν τις συμβιβαστικές λύσεις σε σχέση με τις εναλλακτικές. Όπως οι επιστήμονες, παρατηρούν την συμπεριφορά πολύπλοκων συστημάτων, διαμορφώνουν τις υποθέσεις, και εξετάζουν τις προβλέψεις.

Η πιο σημαντική ικανότητα για έναν επιστήμονα της πληροφορικής είναι η **επίλυση προβλημάτων**. Επίλυση προβλημάτων είναι η δυνατότητα να διατυπώνεις προβλήματα, να σκέφτεσαι δημιουργικά όσον αφορά τις λύσεις και να εκφράζεις μια λύση με σαφήνεια και ακρίβεια. Όπως προκύπτει, η διαδικασία του να μαθαίνεις να προγραμματίζεις είναι μία εξαιρετική ευκαιρία για να εξασκήσεις τις ικανότητές σου πάνω στην επίλυση προβλημάτων. Γι' αυτό το λόγο αυτό το κεφάλαιο ονομάζεται, "Ο τρόπος του προγράμματος".

Ως ένα βαθμό, θα μάθετε να προγραμματίζετε, η οποία είναι μία χρήσιμη δεξιότητα από μόνη της. Ως έναν άλλο βαθμό, θα χρησιμοποιήσετε τον προγραμματισμό ως μέσο για ένα τέλος. Όσο προχωράμε, αυτό το τέλος θα γίνεται πιο ξεκάθαρο.

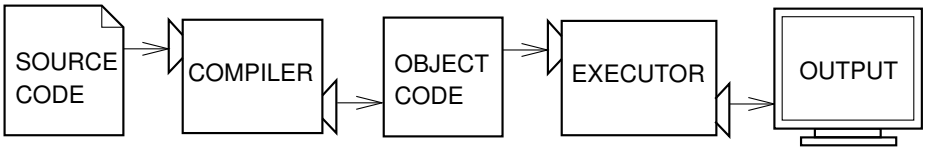
1.1 Η γλώσσα προγραμματισμού Python

Η γλώσσα προγραμματισμού που θα μάθετε είναι η Python. Η Python είναι ένα παράδειγμα **γλώσσας υψηλού επιπέδου**. Άλλες γλώσσες υψηλού επιπέδου που ενδεχομένως να έχετε ακούσει είναι η C, η C++ , η Perl και η Java.

Επίσης υπάρχουν και **γλώσσες χαμηλού επιπέδου**, οι οποίες μερικές φορές αναφέρονται και ως "γλώσσες μηχανής" ή "συμβολικές γλώσσες". Μιλώντας γενικά, οι υπολογιστές μπορούν να τρέξουν μόνο προγράμματα τα οποία είναι γραμμένα σε γλώσσες χαμηλού επιπέδου. Έτσι, προγράμματα τα οποία είναι γραμμένα σε μία γλώσσα υψηλού επιπέδου πρέπει πρώτα να επεξεργαστούν για να μπορούν να τρέξουν. Αυτή η επιπλέον επεξεργασία παίρνει κάποιο χρόνο, το οποίο είναι ένα μικρό μειονέκτημα των γλωσσών υψηλού επιπέδου.

Τα πλεονεκτήματα είναι τεράστια. Πρώτον, είναι πολύ ευκολότερο να προγραμματίζεις σε μία γλώσσα υψηλού επιπέδου. Προγράμματα γραμμένα σε μία γλώσσα υψηλού επιπέδου χρειάζονται λιγότερο

Σχήμα 1.1: Ένας διερμηνέας επεξεργάζεται το πρόγραμμα κομμάτι κομμάτι ή αλλιώς διαβάζει γραμμές και εκτελεί υπολογισμούς.



Σχήμα 1.2: Ένας μεταγλωττιστής μετατρέπει τον πηγαίο κώδικα σε αντικειμενικό κώδικα, ο οποίος τρέχει από έναν εκτελεστή υλικού.

χρόνο για να γραφτούν, είναι μικρότερα και διαβάζονται ευκολότερα, και είναι πιο πιθανό να είναι σωστά. Δεύτερον, οι γλώσσες υψηλού επιπέδου είναι **φορητές**, που σημαίνει ότι μπορούν να τρέξουν σε διαφορετικά είδη υπολογιστών με μερικές ή καθόλου τροποποιήσεις. Προγράμματα χαμηλού επιπέδου μπορούν να τρέξουν μόνο σε ένα είδος υπολογιστή και πρέπει να ξαναγραφτούν για να τρέξουν σε κάποιον άλλο.

Λόγω αυτών των πλεονεκτημάτων, σχεδόν όλα τα προγράμματα γράφονται σε γλώσσες υψηλού επιπέδου. Οι γλώσσες χαμηλού επιπέδου χρησιμοποιούνται μόνο σε μερικές εξειδικευμένες εφαρμογές.

Δύο είδη προγραμμάτων μετατρέπουν μία γλώσσα υψηλού επιπέδου σε μία γλώσσα χαμηλού επιπέδου: οι **διερμηνείς** (interpreters) και οι **μεταγλωττιστές** (compilers). Ένας διερμηνέας διαβάζει ένα πρόγραμμα υψηλού επιπέδου και το εκτελεί, αυτό σημαίνει ότι κάνει ό,τι λέει το πρόγραμμα. Επεξεργάζεται το πρόγραμμα κομμάτι κομμάτι ή αλλιώς διαβάζει γραμμές και εκτελεί υπολογισμούς.

Εικόνα 1.1 δείχνει τη δομή ενός διερμηνέα.

Ένας μεταγλωττιστής διαβάζει το πρόγραμμα και το μεταφράζει ολόκληρο πριν ξεκινήσει να τρέχει το πρόγραμμα. Σε αυτό το πλαίσιο, το πρόγραμμα υψηλού επιπέδου ονομάζεται **πηγαίος κώδικας** (source code), και το μεταφρασμένο πρόγραμμα ονομάζεται **αντικειμενικός κώδικας** (object code) ή **εκτελέσιμο** (executable). Όταν ένα πρόγραμμα μεταγλωττιστεί, μπορείτε να το εκτελέσετε επανειλημμένα χωρίς περαιτέρω μετάφραση.

Εικόνα 1.2 δείχνει τη δομή ενός μεταγλωττιστή.

Η Python θεωρείτε μία διερμηνευμένη γλώσσα επειδή τα προγράμματά της εκτελούνται από έναν διερμηνέα. Υπάρχουν δύο τρόποι χρήσης του διερμηνέα: **διαδραστική λειτουργία** (interactive mode) και **σεναριακή λειτουργία** (script mode). Στην διαδραστική λειτουργία, πληκτρολογούμε προγράμματα σε Python και ο διερμηνέας εμφανίζει το αποτέλεσμα:

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

Το σύμβολο, >>>, είναι ο **προτροπέας** (prompt) που χρησιμοποιεί ο διερμηνέας για να υποδείξει ότι είναι έτοιμος. Αν πληκτρολογήσετε `1 + 1`, ο διερμηνέας απαντάει 2.

Εναλλακτικά, μπορείτε να αποθηκεύσετε κώδικα σε ένα αρχείο και να χρησιμοποιήσετε το διερμηνέα για να εκτελέσει τα περιεχόμενα του αρχείου, το οποίο ονομάζεται **σενάριο** (script). Κατά παράδοση, τα σενάρια στην Python έχουν ονόματα με κατάληξη `.py`.

Για να εκτελεστεί το σενάριο πρέπει να δώσετε στο διερμηνέα το όνομα του φακέλου. Εάν έχετε ένα σενάριο με όνομα `dinsdale.py` και δουλεύετε σε ένα παράθυρο εντολών UNIX θα πρέπει πληκτρολογήσετε `python dinsdale.py`. Σε άλλα περιβάλλοντα ανάπτυξης, οι λεπτομέρειες εκτέλεσης των σεναρίων είναι διαφορετικές. Μπορείτε να βρείτε οδηγίες για το περιβάλλον σας στην ιστοσελίδα της Python `http://python.org`.

Όταν δουλεύετε στην διαδραστική λειτουργία σας βοηθάει να εξετάζετε μικρά κομμάτια κώδικα επειδή μπορείτε να τα πληκτρολογήσετε και να εκτελεστούν άμεσα. Αλλά για κάτι παραπάνω από λίγες γραμμές, θα πρέπει να αποθηκεύσετε τον κώδικά σας σαν σενάριο ώστε να μπορείτε να τον τροποποιήσετε και να το εκτελέσετε στο μέλλον.

1.2 Τι είναι ένα πρόγραμμα

Ένα **πρόγραμμα** είναι μία ακολουθία εντολών η οποία προσδιορίζει πως θα εκτελεστεί ένας υπολογισμός. Αυτός ο υπολογισμός μπορεί να είναι κάτι μαθηματικό, όπως το να λύνεις ένα σύστημα εξισώσεων ή το να βρίσκεις τις ρίζες ενός πολυωνύμου, αλλά επίσης μπορεί να είναι ένας συμβολικός υπολογισμός, όπως το να ψάχνεις και να αντικαθιστάς κείμενο μέσα σε ένα έγγραφο ή (περιέργως) να μεταγλωττίζεις ένα πρόγραμμα.

Οι λεπτομέρειες είναι διαφορετικές από γλώσσα σε γλώσσα αλλά μερικές βασικές εντολές εμφανίζονται σχεδόν σε όλες:

είσοδος: Εισαγωγή δεδομένων από το πληκτρολόγιο, ένα αρχείο, ή οποιαδήποτε άλλη συσκευή.

έξοδος: Εμφάνιση δεδομένων στην οθόνη ή αποστολή σε κάποιο αρχείο ή συσκευή.

μαθηματικά: Εκτέλεση βασικών μαθηματικών πράξεων όπως πρόσθεση και πολλαπλασιασμός.

εκτέλεση υπό συνθήκη: Έλεγχος συγκεκριμένων συνθηκών και εκτέλεση Ελέγχονται κατάλληλου κώδικα.

επανάληψη: Εκτέλεση κάποιας ενέργειας κατ'επανάληψη με κάποια παραλλαγή.

Είτε το πιστεύετε είτε όχι, λίγο πολύ αυτό είναι όλο. Κάθε πρόγραμμα που έχετε χρησιμοποιήσει, ανεξάρτητα από το πόσο περίπλοκο ήταν, απαρτίζεται από εντολές που μοιάζουν λίγο πολύ όπως αυτές. Έτσι μπορείτε να φανταστείτε τον προγραμματισμό σαν μία διαδικασία κατά την οποία σπάμε μία μεγάλη και πολύπλοκη εργασία σε όλο και μικρότερες υποδιεργασίες μέχρις ότου οι υποδιεργασίες να είναι αρκετά απλές για να εκτελεστούν με μία από αυτές τις βασικές εντολές.

Αυτό μπορεί να είναι λίγο ασαφές αλλά θα επανέλθουμε σε αυτό το θέμα όταν θα μιλήσουμε για αλγόριθμους.

1.3 Τι είναι η αποσφαλμάτωση;

Ο προγραμματισμός είναι επιρρεπής σε λάθη. Για ανεξήγητους λόγους τα λάθη στον προγραμματισμό ονομάζονται στα αγγλικά **bugs** ενώ στα ελληνικά σφάλματα και η διαδικασία εντοπισμού τους ονομάζεται **debugging** ή αποσφαλμάτωση στα ελληνικά.

Τρία είδη λαθών μπορεί να συμβούν σε ένα πρόγραμμα: συντακτικά λάθη, λάθη χρόνου εκτέλεσης και λογικά λάθη. Είναι χρήσιμο να γίνει διάκριση μεταξύ τους προκειμένου να εντοπίζονται γρηγορότερα.

1.3.1 Συντακτικά λάθη

Η Python μπορεί να εκτελέσει ένα πρόγραμμα μόνο εάν έχει σωστή σύνταξη, διαφορετικά ο διερμηνέας εμφανίζει μήνυμα λάθους. Η **σύνταξη** αφορά τη δομή ενός προγράμματος και τους κανόνες αυτής της δομής. Για παράδειγμα, οι παρενθέσεις πρέπει να είναι πάντα ζεύγη, έτσι το `(1 + 2)` είναι σωστό, αλλά το `8)` είναι ένα **συντακτικό λάθος** (syntax error).

Στις γλώσσες που χρησιμοποιούμε για επικοινωνία (Ελληνικά, Αγγλικά, Γαλλικά) οι αναγνώστες δέχονται τα περισσότερα συντακτικά λάθη και για αυτό μπορούμε να διαβάζουμε την ποίηση του e. e. cummings χωρίς να αραδιάζουμε μηνύματα λάθους. Η Python όμως δεν είναι τόσο επιεικής. Εάν υπάρχει έστω και ένα συντακτικό λάθος οπουδήποτε μέσα στο πρόγραμμα, η Python θα εμφανίσει ένα μήνυμα λάθους και θα σταματήσει χωρίς να μπορείτε να τρέξετε το πρόγραμμα. Κατά τη διάρκεια των πρώτων εβδομάδων της προγραμματιστικής σας καριέρας, πιθανότατα θα ξοδέψετε πολύ χρόνο στον εντοπισμό συντακτικών λαθών. Όσο αποκτάτε εμπειρία θα κάνετε λιγότερα λάθη και θα τα εντοπίζετε γρηγορότερα.

1.3.2 Λάθη χρόνου εκτέλεσης

Ο δεύτερος τύπος λαθών είναι τα **λάθη χρόνου εκτέλεσης** (runtime errors), ονομάζονται έτσι επειδή τα λάθη δεν εμφανίζονται μέχρις ότου αρχίσει να τρέχει το πρόγραμμα. Αυτά τα λάθη ονομάζονται επίσης **εξαιρέσεις** (exceptions) επειδή συνήθως υποδεικνύουν ότι κάτι σημαντικό (και κακό) έχει συμβεί.

Τα λάθη χρόνου εκτέλεσης είναι σπάνια στα απλά προγράμματα που θα δείτε στα πρώτα κεφάλαια, έτσι ίσως πάρει λίγο χρόνο μέχρι να συναντήσετε ένα.

1.3.3 Λογικά λάθη

Ο τρίτος τύπος λάθους είναι τα **λογικά λάθη** (semantic errors). Εάν υπάρχει ένα λογικό λάθος στο πρόγραμμά σας θα τρέξει επιτυχώς από την άποψη ότι ο υπολογιστής δεν θα παράξει κανένα μήνυμα λάθους, αλλά δεν θα κάνει το σωστό. Θα κάνει κάτι διαφορετικό. Συγκεκριμένα, θα κάνει αυτό που του είπατε να κάνει.

Το πρόβλημα είναι ότι το πρόγραμμα που γράψατε δεν είναι το πρόγραμμα που θέλατε να γράψετε. Το νόημα του προγράμματος (η σημασιολογία του) είναι λάθος. Η αναγνώριση λογικών λαθών μπορεί να είναι δύσκολη γιατί απαιτεί να δουλέψετε προς τα πίσω κοιτάζοντας την έξοδο του προγράμματος προσπαθώντας να καταλάβετε τι συμβαίνει.

1.3.4 Πειραματική αποσφαλμάτωση

Μία από τις πιο σημαντικές ικανότητες που θα αποκτήσετε είναι η αποσφαλμάτωση. Παρόλο που μπορεί να είναι μια επίπονη διαδικασία, η αποσφαλμάτωση είναι ένα από τα πιο πνευματικώς πλούσια, προκλητικά και ενδιαφέροντα μέρη του προγραμματισμού.

Υπό μία έννοια, η αποσφαλμάτωση είναι σαν την δουλειά του ντετέκτιβ. Έρχεστε αντιμέτωποι με ενδείξεις, και πρέπει να συμπεράνετε από ποιες διαδικασίες και συμβάντα προκύπτουν τα αποτελέσματα που βλέπετε.

Η αποσφαλμάτωση μοιάζει επίσης σαν μία πειραματική επιστήμη. Από τη στιγμή που έχετε μία ιδέα για το τι πηγαίνει λάθος, τροποποιείτε το πρόγραμμα και ξαναδοκιμάζετε. Εάν η υπόθεσή σας ήταν σωστή, τότε μπορείτε να προβλέψετε το αποτέλεσμα της τροποποίησης και να είστε ένα βήμα πιο κοντά σε ένα λειτουργικό πρόγραμμα. Εάν η υπόθεσή σας ήταν λανθασμένη, πρέπει να κάνετε μία νέα υπόθεση. Όπως έχει τονίσει ο Sherlock Holmes, ”Όταν έχετε αποκλείσει το αδύνατο, οτιδήποτε μένει, όσο απίθανο και αν είναι, πρέπει να είναι η αλήθεια ”. (A. Conan Doyle, *The Sign of Four*)

Για μερικούς ανθρώπους, ο προγραμματισμός και η αποσφαλμάτωση είναι το ίδιο πράγμα. Δηλαδή, ο προγραμματισμός είναι η διαδικασία της σταδιακής αποσφαλμάτωσης ενός προγράμματος έως ότου κάνει αυτό που θέλετε. Η γενική ιδέα είναι ότι θα πρέπει να ξεκινάτε με ένα πρόγραμμα το οποίο κάνει ”κάτι” και να κάνετε μικρές τροποποιήσεις, αποσφαλματώνοντάς τες προχωρώντας, έτσι ώστε να έχετε πάντα ένα λειτουργικό πρόγραμμα.

Για παράδειγμα, το Linux είναι ένα λειτουργικό σύστημα το οποίο περιέχει χιλιάδες γραμμές κώδικα, αλλά ξεκίνησε σαν ένα απλό πρόγραμμα το οποίο ο Linus Torvalds χρησιμοποιούσε για να εξερευνήσει το ολοκληρωμένο Intel 80386. Σύμφωνα με τον Larry Greenfield, ”Μία από τις πρώτες εργασίες του Linus ήταν ένα πρόγραμμα το οποίο θα ανέστρεφε την εκτύπωση από AAAA σε BBBB. Αυτό αργότερα εξελίχθηκε στο Linux.” (*The Linux Users' Guide* Beta Version 1).

Τα επόμενα κεφάλαια θα κάνουν περισσότερες υποδείξεις σχετικά με την αποσφαλμάτωση και άλλες προγραμματιστικές πρακτικές.

1.4 Φυσικές και τυπικές γλώσσες

Οι **φυσικές γλώσσες** είναι οι γλώσσες που μιλούν οι άνθρωποι, όπως τα Αγγλικά, τα Ισπανικά και τα Γαλλικά. Δεν έχουν σχεδιαστεί από τους ανθρώπους (παρόλο που οι άνθρωποι προσπαθούν να επιβάλουν κάποια τάξη σε αυτές), έχουν εξελιχθεί φυσικά.

Οι **τυπικές γλώσσες** (formal) είναι γλώσσες που έχουν σχεδιαστεί από ανθρώπους για συγκεκριμένες εφαρμογές. Για παράδειγμα, η σημειογραφία που χρησιμοποιούν οι μαθηματικοί είναι μια τυπική γλώσσα η οποία είναι ιδιαίτερα καλή στο να δείχνει τις σχέσεις μεταξύ αριθμών και συμβόλων. Οι χημικοί χρησιμοποιούν μία τυπική γλώσσα για να αναπαραστήσουν τη χημική δομή των μορίων. Και το πιο σημαντικό:

> **Οι προγραμματιστικές γλώσσες είναι τυπικές γλώσσες οι οποίες έχουν σχεδιαστεί για να εκφράζουν υπολογισμούς.**

Οι τυπικές γλώσσες τείνουν να έχουν αυστηρούς κανόνες σύνταξης. Για παράδειγμα, $3 + 3 = 6$ είναι μία συντακτικά σωστή μαθηματική έκφραση, αλλά αυτή $3+ = 3\$6$ δεν είναι. H_2O είναι ένας συντακτικά σωστός χημικός τύπος, αλλά αυτός ${}_2Zz$ δεν είναι.

Οι κανόνες σύνταξης είναι δύο τύπων, αυτοί που αφορούν τα **σύμβολα** (tokens) και αυτοί που αφορούν τη **δομή** (structure). Τα σύμβολα είναι τα βασικά στοιχεία της γλώσσας όπως λέξεις, αριθμοί και χημικά στοιχεία. Ένα από τα προβλήματα με το $3+ = 3\$6$ είναι ότι το \$ δεν είναι ένα έγκυρο σύμβολο στα μαθηματικά (τουλάχιστον απ' όσο γνωρίζω). Παρομοίως, το ${}_2Zz$ δεν είναι έγκυρο επειδή δεν υπάρχει στοιχείο με την συντομογραφία Zz.

Ο δεύτερος τύπος συντακτικού κανόνα αφορά τη δομή μίας έκφρασης, δηλαδή τον τρόπο με τον οποίο έχουν διαταχθεί τα σύμβολα. Η έκφραση $3+ = 3$ είναι λάθος γιατί παρόλο που το $+$ και το

= είναι έγκυρα σύμβολα, δεν μπορείτε να έχετε το ένα ακριβώς μετά το άλλο. Παρομοίως, σε ένα χημικό τύπο ο δείκτης μπαίνει μετά το όνομα του στοιχείου, όχι πριν.

Όταν διαβάζετε μία πρόταση στα Αγγλικά ή μία έκφραση σε μία τυπική γλώσσα, πρέπει να καταλάβετε ποια είναι η δομή της πρότασης (παρόλο που σε μια φυσική γλώσσα το κάνετε υποσυνείδητα). Αυτή η διαδικασία ονομάζεται συντακτική ανάλυση.

Για παράδειγμα, όταν ακούτε την πρόταση, "Το νόμισμα έπεσε," καταλαβαίνετε ότι "το νόμισμα" είναι το αντικείμενο και το "έπεσε" είναι το κατηγορούμενο. Μόλις αναλύσετε μία πρόταση, μπορείτε να καταλάβετε τι σημαίνει ή αλλιώς τη σημασιολογία της πρότασης. Υποθέτοντας ότι γνωρίζετε τι είναι το νόμισμα και τι σημαίνει το ότι πέφτει, θα καταλάβετε τον υπαινιγμό της πρότασης.

Παρόλο που οι επίσημες και οι τυπικές γλώσσες έχουν πολλά κοινά χαρακτηριστικά—σύμβολα, δομή, συντακτικό και σημασιολογία—υπάρχουν κάποιες διαφορές:

ασάφεια: Οι φυσικές γλώσσες είναι γεμάτες ασάφεια την οποία οι άνθρωποι αντιμετωπίζουν με βάση τα συμφραζόμενα και άλλες πληροφορίες. Οι τυπικές γλώσσες έχουν σχεδιαστεί για να είναι σχεδόν ή πλήρως σαφείς, το οποίο σημαίνει ότι οποιαδήποτε έκφραση έχει ακριβώς μία ερμηνεία, ανεξαρτήτως περιεχομένου.

πλεονασμός: Προκειμένου να επιτύχουμε σαφήνεια και να μειώσουμε τις παρεξηγήσεις, οι φυσικές γλώσσες χρησιμοποιούν πολλούς πλεονασμούς. Σαν αποτέλεσμα, είναι συχνά φλύαρες. Οι τυπικές γλώσσες είναι λιγότερο πλεονάζουσες και περισσότερο συνοπτικές.

μεταφορά: Οι φυσικές γλώσσες είναι γεμάτες από ιδιωματισμούς και μεταφορές. Εάν πω, "Το νόμισμα έπεσε" πιθανώς δεν υπάρχει νόμισμα και τίποτα δεν έχει πέσει (αυτός ο ιδιωματισμός σημαίνει ότι κάποιος συνειδητοποίησε κάτι μετά από μία περίοδο σύγχυσης). Οι τυπικές γλώσσες εννοούν αυτό ακριβώς που λένε.

Οι άνθρωποι που μεγάλωσαν μιλώντας μία φυσική γλώσσα (όλοι) είναι συχνά δύσκολο να εξοικειωθούν με τις τυπικές γλώσσες. Κατά κάποιον τρόπο, η διαφορά ανάμεσα στην τυπική και στη φυσική γλώσσα είναι σαν τη διαφορά ανάμεσα στην ποίηση και στον πεζό λόγο, αλλά ειδικότερα:

Ποίηση: Οι λέξεις χρησιμοποιούνται τόσο για τους ήχους τους όσο και για τη σημασία τους, και ολόκληρο το ποίημα δημιουργεί μία συναισθηματική αντίδραση. Η ασάφεια δεν είναι μόνο σύνηθες φαινόμενο αλλά συχνά σκόπιμη.

Πεζός λόγος: Η κυριολεκτική έννοια των λέξεων είναι περισσότερο σημαντική, και η δομή συμβάλλει περισσότερο στο νόημα. Ο πεζός λόγος επιδέχεται περισσότερη ανάλυση από την ποίηση αλλά είναι συχνά διφορούμενος.

Προγράμματα: Το νόημα ενός υπολογιστικού προγράμματος είναι σαφές και κυριολεκτικό, και μπορεί να κατανοηθεί πλήρως μέσω της ανάλυσης των συμβόλων και της δομής.

Αυτές είναι κάποιες υποδείξεις για το πώς να διαβάζετε προγράμματα (και άλλες τυπικές γλώσσες). Πρώτον, να θυμάστε ότι οι τυπικές γλώσσες είναι περισσότερο πυκνογραμμένες από τις φυσικές γλώσσες, γι' αυτό παίρνει περισσότερο να τις διαβάσουμε. Επίσης η δομή είναι πολύ σημαντική, οπότε συνήθως δεν είναι καλή ιδέα να τις διαβάζουμε από την αρχή προς το τέλος, από αριστερά στα δεξιά. Αντί αυτού, πρέπει να μάθετε να αναλύετε το πρόγραμμα στο μυαλό σας, αναγνωρίζοντας τα σύμβολα και ερμηνεύοντας τη δομή. Τελικά, οι λεπτομέρειες μετράνε. Μικρά λάθη στο συλλαβισμό και στη στίξη, τα οποία δεν δημιουργούν σοβαρό πρόβλημα στις φυσικές γλώσσες, μπορούν να κάνουν μεγάλη διαφορά σε μία τυπική γλώσσα.

1.5 Το πρώτο πρόγραμμα

Κατά παράδοση, το πρώτο πρόγραμμα που γράφετε σε μία νέα γλώσσα ονομάζεται ”Hello, World!” επειδή το μόνο που κάνει είναι να εμφανίζει τις λέξεις ”Hello, World”. Στην Python είναι κάπως έτσι:

```
print 'Hello, World!'
```

Αυτό είναι ένα παράδειγμα μίας **δήλωσης εκτύπωσης**, η οποία στην πραγματικότητα δεν εκτυπώνει κάτι στο χαρτί αλλά εμφανίζει μία τιμή στην οθόνη. Σε αυτή την περίπτωση το αποτέλεσμα είναι οι λέξεις:

```
Hello, World!
```

Τα εισαγωγικά μέσα στο πρόγραμμα σηματοδοτούν την αρχή και το τέλος του κειμένου που θα εμφανιστεί και δεν φαίνονται στο αποτέλεσμα.

Στην Python 3, η σύνταξη για εκτύπωση είναι λίγο διαφορετική:

```
print('Hello, World!')
```

Οι παρενθέσεις υποδηλώνουν ότι η `print` είναι μία συνάρτηση. Θα δούμε τις συναρτήσεις στο Κεφάλαιο 3.

Στη συνέχεια αυτού του βιβλίου θα χρησιμοποιήσουμε την έκφραση print της Python2. Εάν χρησιμοποιείτε την Python 3 θα πρέπει να την προσαρμόζετε. Αλλά εκτός αυτού δεν υπάρχουν πολλές διαφορές για τις οποίες θα πρέπει να ανησυχούμε.

1.6 Αποσφαλμάτωση

Είναι προτιμότερο να διαβάζετε αυτό το βιβλίο μπροστά από έναν υπολογιστή έτσι ώστε να μπορείτε να δοκιμάζετε τα παραδείγματα όσο προχωράτε. Μπορείτε να τρέξετε τα περισσότερα από τα παραδείγματα σε διαδραστική λειτουργία αλλά αν βάζατε τον κώδικα μέσα σε ένα σενάρια τότε θα ήταν ευκολότερο να πειραματιστείτε.

Κάθε φορά που πειραματίζεστε με ένα νέο χαρακτηριστικό της γλώσσας, θα πρέπει να δοκιμάζετε να κάνετε λάθη. Για παράδειγμα, τι θα συμβεί στο πρόγραμμα ”Hello, world!” αν δεν βάλετε ένα από τα εισαγωγικά; Εάν δεν βάλετε κανένα από τα δύο; Εάν γράψετε λάθος την `print`;

Αυτό το είδος πειραματισμού σας βοηθάει να θυμάστε τι διαβάζετε και επίσης σας βοηθάει στην αποσφαλμάτωση, επειδή θα μάθετε τι σημαίνουν τα μηνύματα λάθους. Είναι προτιμότερο να κάνετε εσκεμμένα λάθη τώρα παρά αργότερα κατά λάθος.

Μερικές φορές, ο προγραμματισμός και ιδιαίτερα η αποσφαλμάτωση μπορεί να προκαλέσουν έντονα συναισθήματα. Εάν ταλαιπωρείστε με ένα δύσκολο σφάλμα τότε μπορεί να νιώσετε Θυμό, αποθάρρυνση ή ακόμα και ντροπή.

Έχει αποδειχθεί ότι οι άνθρωποι αντιμετωπίζουν τους υπολογιστές σαν να ήταν άνθρωποι. Όταν λειτουργούν σωστά τους βλέπουμε σαν συνεργάτες αλλά όταν είναι ”πεισματάρηδες” ή ”αγενείς”, τότε τους συμπεριφερόμαστε με τον ίδιο τρόπο που θα συμπεριφερόμασταν σε αγενείς και πεισματάρηδες ανθρώπους (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

Αν είστε προετοιμασμένοι για αυτές τις αντιδράσεις μπορείτε να τις αντιμετωπίσετε. Μία καλή λύση είναι να σκέφτεστε τον υπολογιστή σαν ένα υπάλληλο με συγκεκριμένες δυνατότητες όπως η ταχύτητα και η ακρίβεια και συγκεκριμένες αδυναμίες όπως η έλλειψη συναίσθησης και η ανικανότητά τους να κατανοήσουν το γενικό νόημα.

Η δουλειά σας είναι να είστε ένας καλός διευθυντής: βρείτε τρόπους να εκμεταλλευτείτε τις δυνατότητες και να μετριάσετε τις αδυναμίες. Επίσης, βρείτε τρόπους να χρησιμοποιήσετε τα συναισθήματά σας ώστε να ασχοληθείτε με το πρόβλημα χωρίς να αφήσετε τις αντιδράσεις σας να επηρεάσουν την ικανότητά σας να δουλεύετε αποδοτικά.

Το να μάθετε να αποσφαλματώνετε μπορεί να είναι απογοητευτικό αλλά είναι μία σημαντική ικανότητα η οποία είναι χρήσιμη και για πολλές δραστηριότητες πέραν του προγραμματισμού. Στο τέλος κάθε κεφαλαίου υπάρχει μία ενότητα αποσφαλμάτωσης όπως αυτή με τις σκέψεις μου σχετικά με την αποσφαλμάτωση. Ελπίζω να βοηθήσουν!

1.7 Ορολογία

επίλυση προβλημάτων: Η διαδικασία τυποποίησης ενός προβλήματος, η εύρεση λύσης και η έκφραση της λύσης.

γλώσσα υψηλού επιπέδου: Μία γλώσσα προγραμματισμού όπως η Python που έχει σχεδιαστεί για να είναι εύκολη για τους ανθρώπους στην ανάγνωση και στη γραφή.

γλώσσα χαμηλού επιπέδου: Μία γλώσσα που έχει σχεδιαστεί για να είναι εύκολο να εκτελεστεί από έναν υπολογιστή, ονομάζεται επίσης ”γλώσσα μηχανής” ή ”συμβολική γλώσσα.”

φορητότητα: Η ιδιότητα ενός προγράμματος να μπορεί να τρέξει σε σε διαφορετικούς υπολογιστές.

διερμηνεία: Να εκτελείς ένα πρόγραμμα γλώσσας υψηλού επιπέδου μεταφράζοντάς μία γραμμή κάθε φορά.

μεταγλώττιση: Να μεταφράζεις ένα πρόγραμμα γραμμένο σε γλώσσα υψηλού επιπέδου σε γλώσσα χαμηλού επιπέδου κατευθείαν, προετοιμάζοντάς το για μετέπειτα εκτέλεση.

πηγαίος κώδικας: Ένα πρόγραμμα σε μία γλώσσα υψηλού επιπέδου προτού μεταγλωττιστεί.

αντικείμενος κώδικας: Η έξοδος του μεταγλωττιστή αφού μετατρέψει το πρόγραμμα.

εκτελέσιμο: Μία άλλη ονομασία του αντικείμενου κώδικα ο οποίος είναι έτοιμος για εκτέλεση.

προτροπέας: Οι χαρακτήρες που εμφανίζονται από το διερμηνέα για να υποδηλώσουν ότι είναι έτοιμος να δεχτεί την είσοδο από το χρήστη.

σενάριο: Ένα πρόγραμμα αποθηκευμένο σε ένα αρχείο (συνήθως προς διερμηνεία).

διαδραστική λειτουργία: Ένας τρόπος χρήσης του διερμηνέα της Python πληκτρολογώντας εντολές και εκφράσεις στον προτροπέα.

σεναριακή λειτουργία: Ένας τρόπος χρήσης του διερμηνέα της Python που διαβάζει και εκτελεί εκφράσεις από ένα σενάριο.

πρόγραμμα: Ένα σύνολο εντολών που ορίζει έναν υπολογισμό.

αλγόριθμος: Μία γενική διαδικασία για την επίλυση μιας κατηγορίας προβλημάτων.

σφάλμα: Ένα λάθος σε κάποιο πρόγραμμα.

αποσφαλμάτωση: Η διαδικασία εύρεσης και απομάκρυνσης οποιουδήποτε εκ των τριών τύπων προγραμματιστικών σφαλμάτων.

συντακτικό: Η δομή ενός προγράμματος.

συντακτικό λάθος: Ένα λάθος σε ένα πρόγραμμα το οποίο το καθιστά αδύνατο να αναλυθεί (και επομένως αδύνατον να διερμηνευτεί).

εξαίρεση: Ένα λάθος το οποίο ανακύπτει κατά την εκτέλεση του προγράμματος.

σημασιολογία: Το νόημα ενός προγράμματος.

σημασιολογικό λάθος: Ένα λάθος σε ένα πρόγραμμα όταν αυτό κάνει κάτι διαφορετικό από αυτό που είχε σκοπό ο προγραμματιστής.

φυσική γλώσσα: Οποιαδήποτε από τις γλώσσες που μιλούν οι άνθρωποι η οποία έχει εξελιχθεί φυσικά.

τυπική γλώσσα: Οποιαδήποτε από τις γλώσσες που έχουν σχεδιάσει οι άνθρωποι για συγκεκριμένους σκοπούς, όπως αναπαράσταση μαθηματικών ιδεών ή προγραμμάτων υπολογιστών. Όλες οι γλώσσες προγραμματισμού είναι τυπικές γλώσσες.

σύμβολο: Ένα από τα βασικά στοιχεία της συντακτικής δομής ενός προγράμματος, αντίστοιχο με τη λέξη σε μια φυσική γλώσσα.

ανάλυση: Να εξετάζεις ένα πρόγραμμα και να αναλύεις την συντακτική του δομή.

δήλωση εκτύπωσης: Μία εντολή που προκαλεί τον διερμηνέα της Python να εμφανίσει μία τιμή στην οθόνη.

1.8 Ασκήσεις

Άσκηση 1.1. *Χρησιμοποιήστε ένα φυλλομετρητή για να επισκεφθείτε την ιστοσελίδα της Python* `http://python.org`*. Αυτή η σελίδα περιέχει πληροφορίες σχετικά με την Python και συνδέσμους οι οποίοι σχετίζονται με την Python, και σας δίνει την δυνατότητα να αναζητήσετε την τεκμηρίωση της Python.*

Για παράδειγμα, εάν πληκτρολογήσετε `print` *στο πεδίο της αναζήτησης, ο πρώτος σύνδεσμος που εμφανίζεται είναι η τεκμηρίωση για αυτήν την έκφραση. Σε αυτό το σημείο ενδεχομένως να μην βγάζει νόημα όλο αυτό το κατεβατό, αλλά είναι καλό να ξέρετε που υπάρχει.*

Άσκηση 1.2. *Εκκινήστε τον διερμηνέα της Python πληκτρολογήστε* `help()` *για να ξεκινήσει το διαδικτυακό εργαλείο βοήθειας. Ή μπορείτε να πληκτρολογήσετε* `help('print')` *για να πάρετε πληροφορίες σχετικές με την έκφραση* `print`*.*

Εάν αυτό το παράδειγμα δεν δουλεύει, ίσως χρειαστεί να εγκαταστήσετε επιπρόσθετη τεκμηρίωση της Python ή να θέσετε μία μεταβλητή περιβάλλοντος, οι λεπτομέρειες εξαρτώνται από το λειτουργικό σας σύστημα και την έκδοση της Python.

Άσκηση 1.3. *Εκκινήστε τον διερμηνέα της Python και χρησιμοποιήστε τον σαν αριθμομηχανή. Το συντακτικό της Python για μαθηματικές πράξεις είναι σχεδόν το ίδιο με την τυπική μαθηματική σημειογραφία. Για παράδειγμα, τα σύμβολα* `+`*,* `-` *και* `/` *δηλώνουν πρόσθεση, αφαίρεση και διαίρεση, όπως θα περιμένατε. Το σύμβολο του πολλαπλασιασμού είναι* `*`*.*

Εάν τρέξετε έναν αγώνα 10 χιλιομέτρων σε 43 λεπτά και 30 δευτερόλεπτα, ποιος είναι ο μέσος χρόνος σας ανά μίλι ; Ποια είναι η μέση σας ταχύτητα σε μίλια ανά ώρα; (Σημείωση: ένα μίλι είναι 1,61 χιλιόμετρα).

Κεφάλαιο 2

Μεταβλητές, εκφράσεις και δηλώσεις

2.1 Τιμές και τύποι

Μία **τιμή** είναι ένα από τα βασικά στοιχεία που κάνουν λειτουργικό ένα πρόγραμμα, όπως για παράδειγμα ένα γράμμα ή ένας αριθμός. Οι τιμές που έχουμε δει μέχρι στιγμής είναι οι `1`, `2`, και `'Hello, World!'`.

Αυτές οι τιμές υπόκεινται σε διαφορετικούς τύπους: το `2` είναι ακέραιος και το `'Hello, World!'` είναι μία συμβολοσειρά (ονομάζεται έτσι επειδή περιέχει μια "σειρά" από σύμβολα (γράμματα). Μπορείτε (το ίδιο και ο διερμηνέας) να αναγνωρίσετε συμβολοσειρές, επειδή περικλείονται σε εισαγωγικά.

Αν δεν είστε σίγουροι για τον τύπο μιας τιμής, τότε μπορεί να σας τον υποδείξει ο διερμηνέας:

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

Προφανώς, οι συμβολοσειρές ανήκουν στον τύπο `str` και οι ακέραιοι ανήκουν στον τύπο `int`. Λιγότερο προφανές είναι όμως ότι οι αριθμοί με δεκαδικά ψηφία ανήκουν σε ένα τύπο που ονομάζεται `float`, επειδή αυτοί οι αριθμοί παριστάνονται σε μία μορφή ονομαζόμενη **floating-point**.

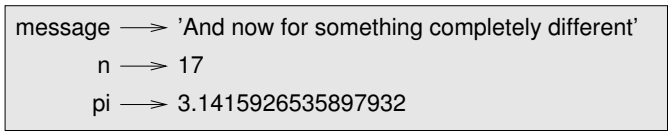
```
>>> type(3.2)
<type 'float'>
```

Τι γίνεται με τιμές όπως η `'17'` και η `'3.2'`; Μοιάζουν με αριθμούς αλλά περικλείονται σε εισαγωγικά όπως οι συμβολοσειρές.

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

Είναι συμβολοσειρές.

Εάν πληκτρολογήσετε έναν μεγάλο ακέραιο τότε ίσως μπείτε στον πειρασμό να χρησιμοποιήσετε κόμματα (ή τελείες) ανά τρία ψηφία, όπως για παράδειγμα `1,000,000`. Αυτός δεν είναι έγκυρος



Σχήμα 2.1: Διάγραμμα Κατάστασης.

ακέραιος για την Python, αλλά γενικά είναι σωστό:

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

Αυτό προφανώς δεν έχει καμία σχέση με αυτό που περιμέναμε! Η Python ερμηνεύει το `1,000,000` σαν μία ακολουθία ακεραίων χωρισμένη με κόμματα. Αυτό είναι το πρώτο παράδειγμα σημασιολογικού (λογικού) λάθους που βλέπουμε: ο κώδικας τρέχει χωρίς να παράγει κανένα μήνυμα λάθους αλλά δεν κάνει αυτό που θέλαμε.

2.2 Μεταβλητές

Ένα από τα πιο δυνατά χαρακτηριστικά μιας γλώσσας προγραμματισμού είναι η ικανότητα να διαχειρίζεται **μεταβλητές** (variables). Μία μεταβλητή είναι ένα όνομα που αναφέρεται σε μία τιμή.

Μία **δήλωση εκχώρησης** (assignment statement) δημιουργεί νέες μεταβλητές και δίνει τιμές σε αυτές:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897932
```

Σε αυτό το παράδειγμα κάνουμε τρεις εκχωρήσεις τιμών. Η πρώτη εκχωρεί μία συμβολοσειρά σε μία νέα μεταβλητή με όνομα `message`, η δεύτερη δίνει στη `n` την ακέραια τιμή `17` και η τρίτη εκχωρεί την (κατά προσέγγιση) τιμή του π στην `pi`.

Ο πιο συνηθισμένος τρόπος αναπαράστασης των μεταβλητών σε χαρτί είναι να γράψετε το όνομα της μεταβλητής με ένα βελάκι το οποίο θα δείχνει την τιμή της. Αυτός ο τρόπος απεικόνισης ονομάζεται **διάγραμμα κατάστασης** (state diagram) γιατί δείχνει την κατάσταση της κάθε μεταβλητής. Το σχήμα 2.1 δείχνει το αποτέλεσμα του προηγούμενου παραδείγματος.

Ο τύπος της μεταβλητής είναι ίδιος με τον τύπο της τιμής στην οποία αναφέρεται.

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

Άσκηση 2.1. *Εάν πληκτρολογήσετε έναν ακέραιο που έχει στην αρχή μηδέν, ενδεχομένως να προκύψει σφάλμα λόγω σύγχυσης στο διερμηνέα:*

```
>>> zipcode = 02492
                  ^
SyntaxError: invalid token
```

Άλλα νούμερα φαίνεται να δουλεύουν, αλλά τα αποτελέσματα είναι παράξενα:

```
>>> zipcode = 02132
>>> zipcode
1114
```

Μπορείτε να καταλάβετε τι συμβαίνει ; Σημείωση: εμφανίστε τις τιμές `01`*,* `010`*,* `0100` *και* `01000`*.*

2.3 Ονόματα μεταβλητών και λέξεις κλειδιά

Γενικά, οι προγραμματιστές επιλέγουν ονόματα για τις μεταβλητές τους τα οποία έχουν κάποιο νόημα (τεκμηριώνουν για ποιο λόγο χρησιμοποιείται η μεταβλητή).

Τα ονόματα των μεταβλητών μπορούν να είναι αυθαίρετα μεγάλα και μπορούν να περιέχουν και γράμματα και νούμερα αλλά πρέπει να ξεκινάνε με ένα γράμμα. Είναι έγκυρο να χρησιμοποιούμε κεφαλαία γράμματα αλλά είναι προτιμότερο να ξεκινάτε τα ονόματα των μεταβλητών με πεζά (θα δείτε αργότερα γιατί).

Η κάτω παύλα (`_`) μπορεί να χρησιμοποιηθεί σε κάποιο όνομα και είναι ιδιαίτερα χρήσιμη σε ονόματα με πολλές λέξεις όπως το `my_name` ή το `airspeed_of_unladen_swallow`.

Εάν δώσετε σε μία μεταβλητή λάθος όνομα τότε θα παραχθεί συντακτικό λάθος:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

Το `76trombones` είναι λάθος γιατί δεν ξεκινάει με γράμμα. Το `more@` είναι λάθος γιατί περιέχει ένα μη έγκυρο χαρακτήρα (`@`). Αλλά γιατί το `class` είναι λάθος;

Αποδεικνύεται ότι το `class` είναι μία από τις **λέξεις κλειδιά** (keywords) της Python. Ο διερμηνέας χρησιμοποιεί λέξεις κλειδιά για να αναγνωρίσει τη δομή ενός προγράμματος και για αυτόν το λόγο δεν μπορούν χρησιμοποιηθούν για ονόματα μεταβλητών.

Η Python 2 έχει 31 λέξεις κλειδιά.

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

Στην Python 3 το `exec` δεν είναι πλέον λέξη κλειδί αλλά είναι το `nonlocal`.

Καλό θα ήταν να έχετε αυτή τη λίστα εύχερη σε περίπτωση που ο διερμηνέας παραπονεθεί για μία από τις μεταβλητές σας και δεν ξέρετε γιατί, κοιτάξτε αν βρίσκεται σε αυτή τη λίστα.

2.4 Τελεστές και τελεστέοι

Οι **τελεστές** είναι ειδικά σύμβολα τα οποία αναπαριστούν υπολογισμούς όπως η πρόσθεση και ο πολλαπλασιασμός. Οι τιμές στις οποίες εφαρμόζεται ο τελεστής ονομάζονται **τελεστέοι**.

Οι τελεστές +, -, *, / και ** εκτελούν πρόσθεση, αφαίρεση, πολλαπλασιασμό, διαίρεση και ύψωση σε δύναμη:

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

Σε κάποιες άλλες γλώσσες, για την ύψωση σε δύναμη χρησιμοποιείται το σύμβολο ^ αλλά στην Python είναι δυαδικός τελεστής και ονομάζεται XOR. Σε αυτό το βιβλίο δεν θα επεκταθούμε στους δυαδικούς τελεστές αλλά μπορείτε να διαβάσετε για αυτούς στην διεύθυνση `http://wiki.python.org/moin/BitwiseOperators`.

Στην Python, ο τελεστής της διαίρεσης μπορεί να μην κάνει αυτό που θα περιμένατε:

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

Η τιμή του `minute` είναι 59 και στα συμβατικά μαθηματικά το 59 διαιρούμενο με το 60 μας δίνει 0.98333 και όχι 0. Ο λόγος που συμβαίνει αυτό είναι ότι η Python εκτελεί ακέραια διαίρεση. Όταν και οι δύο τελεστέοι είναι ακέραιοι αριθμοί τότε το αποτέλεσμα θα είναι επίσης ακέραιος αριθμός γιατί η ακέραια διαίρεση κόβει το κλασματικό τμήμα. Έτσι, σε αυτό το παράδειγμα το αποτέλεσμα στρογγυλοποιείται στο μηδέν.

Στην Python 3 το αποτέλεσμα της διαίρεσης είναι `float` και ο νέος τελεστής για την ακέραια διαίρεση είναι `//`.

Εάν κάποιος από τους τελεστέους είναι αριθμός κινητής υποδιαστολής τότε η Python εκτελεί κλασματική διαίρεση και το αποτέλεσμα είναι `float`:

```
>>> minute/60.0
0.98333333333333328
```

2.5 Εκφράσεις και δηλώσεις

Μία **έκφραση** είναι ένας συνδυασμός από τιμές, μεταβλητές και τελεστές. Μία τιμή από μόνη της θεωρείται σαν μία έκφραση, το ίδιο και μία μεταβλητή. Έτσι όλες οι ακόλουθες εκφράσεις είναι έγκυρες (υποθέτοντας ότι έχει ανατεθεί τιμή στη μεταβλητή `x`):

```
17
x
x + 17
```

Μία δήλωση είναι μία μονάδα κώδικα την οποία ο διερμηνέας της Python μπορεί να εκτελέσει. Έχουμε δει δύο τύπους δηλώσεων: την εκτύπωση και την εκχώρηση.

Στην πράξη μία έκφραση είναι ταυτόχρονα και δήλωση αλλά ίσως είναι πιο απλό να τα σκέφτεστε σαν δύο διαφορετικά πράγματα. Η σημαντική διαφορά είναι ότι μία έκφραση έχει μία τιμή ενώ μία δήλωση όχι.

2.6 Διαδραστική λειτουργία και λειτουργία σεναρίων

Ένα από τα πλεονεκτήματα του να δουλεύεις με μια διερμηνευόμενη γλώσσα είναι ότι μπορείτε να ελέγχετε μικρά κομμάτια κώδικα στην διαδραστική λειτουργία προτού τα τοποθετήσετε σε ένα σενάριο (script). Υπάρχουν όμως κάποιες διαφορές μεταξύ της διαδραστικής λειτουργίας και της λειτουργίας σεναρίων οι οποίες μπορεί να προκαλέσουν σύγχυση.

Για παράδειγμα, εάν χρησιμοποιείτε την Python σαν αριθμομηχανή τότε μπορείτε να πληκτρολογήσετε:

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

Στην πρώτη γραμμή εκχωρείται μία τιμή στην `miles` αλλά δεν έχει κάποια ορατή επίδραση. Η δεύτερη γραμμή είναι μία έκφραση και επομένως ο διερμηνέας την αποτιμά και εμφανίζει το αποτέλεσμα. Προκύπτει λοιπόν ότι ένας μαραθώνιος είναι περίπου 42 χιλιόμετρα.

Αν πληκτρολογήσετε όμως τον ίδιο κώδικα μέσα σε ένα σενάριο και το τρέξετε δεν θα έχετε καμία έξοδο. Σε λειτουργία σεναρίων μία έκφραση από μόνη της δεν έχει καμία ορατή επίδραση. Η Python αποτιμά την έκφραση αλλά δεν εμφανίζει την τιμή εκτός και αν της πείτε να το κάνει:

```
miles = 26.2
print miles * 1.61
```

Αυτή η συμπεριφορά μπορεί να σας μπερδεύει στην αρχή.

Ένα σενάριο συνήθως περιέχει μία ακολουθία από δηλώσεις. Εάν υπάρχουν περισσότερες από μία δηλώσεις τότε τα αποτελέσματα εμφανίζονται ένα ένα με βάση τη σειρά εκτέλεσης των δηλώσεων.

Για παράδειγμα, το σενάριο:

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

παράγει την έξοδο:

```
1
2
```

Η δήλωση εκχώρησης δεν παράγει καμία έξοδο.

Άσκηση 2.2. *Πληκτρολογήστε τις ακόλουθες δηλώσεις στον διερμηνέα της Python για να δείτε τι κάνουν:*

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

Τώρα τοποθετήστε τις ίδιες δηλώσεις μέσα σε ένα σενάριο και τρέξτε το. Ποια είναι η έξοδος; Τροποποιήστε το σενάριο μετατρέποντας καθεμία από τις εκφράσεις σε μία δήλωση εκτύπωσης και μετά ξανατρέξτε το.

2.7 Η σειρά των πράξεων

Όταν υπάρχουν περισσότεροι από έναν τελεστή μέσα σε μία έκφραση τότε η σειρά των πράξεων εξαρτάται από τους κανόνες προτεραιότητας. Για μαθηματικούς τελεστές, η Python ακολουθεί την πρότυπη μαθηματική προτεραιότητα. Ένας χρήσιμος τρόπος για να θυμάστε τους κανόνες είναι το ακρωνύμιο **PEMDAS**:

- Οι παρενθέσεις (**P**arentheses) έχουν την υψηλότερη προτεραιότητα και μπορούν να χρησιμοποιηθούν για να εξαναγκάσουν μια έκφραση να αποτιμηθεί με τη σειρά που θέλετε. Από τη στιγμή που εκφράσεις στις παρενθέσεις αξιολογούνται πρώτες, το `2 * (3-1)` μας κάνει 4,

και το `(1+1)**(5-2)` μας κάνει 8. Μπορείτε επίσης να χρησιμοποιήσετε τις παρενθέσεις για να κάνετε μία έκφραση πιο ευανάγνωστη, όπως την `(minute * 100) / 60`, ακόμα και αν δεν αλλάζει το αποτέλεσμα.

- Η ύψωση σε δύναμη (**E**xponentiation) έχει την αμέσως υψηλότερη προτεραιότητα. Έτσι η έκφραση `2**1+1` δίνει 3 αντί για 4 και η `3*1**3` δίνει 3 αντί για 27.
- Ο πολλαπλασιασμός (**M**ultiplication) και η διαίρεση **D**ivision) έχουν την ίδια προτεραιότητα, η οποία είναι υψηλότερη από την πρόσθεση (**A**ddition) και την αφαίρεση (**S**ubtraction), οι οποίες έχουν επίσης την ίδια προτεραιότητα. Έτσι το `2*3-1` μας κάνει 5 αντί για 4 και το `6+4/2` μας κάνει 8 αντί για 5.
- Οι τελεστές με την ίδια προτεραιότητα αξιολογούνται από τα αριστερά στα δεξιά (εκτός της ύψωσης σε δύναμη). Άρα στην έκφραση `degrees / 2 * pi` θα εκτελεστεί πρώτα η διαίρεση και το αποτέλεσμά της θα πολλαπλασιαστεί με `pi`. Για να διαιρέσετε με 2π μπορείτε να χρησιμοποιήσετε παρενθέσεις ή να γράψετε `degrees / 2 / pi`.

Δεν προσπαθώ ιδιαίτερα να θυμάμαι τους κανόνες προτεραιότητας για άλλους τελεστές. Εάν δεν μπορώ να καταλάβω κοιτώντας την έκφραση τότε χρησιμοποιώ παρενθέσεις για να το κάνω ξεκάθαρο.

2.8 Πράξεις συμβολοσειρών

Γενικά δεν μπορείτε να εκτελέσετε μαθηματικές πράξεις στις συμβολοσειρές (strings) ακόμα και αν οι συμβολοσειρές μοιάζουν με αριθμούς. Επομένως όλα τα ακόλουθα είναι λάθος:

```
'2'-'1'    'eggs'/'easy'    'third'*'a charm'
```

Ο τελεστής `+` μπορεί να εφαρμοστεί σε συμβολοσειρές και να εκτελέσει **συνένωση** (concatenation), το οποίο σημαίνει ότι ενώνει τις συμβολοσειρές τοποθετώντας στο τέλος της πρώτης την αρχή της δεύτερης. Για παράδειγμα:

```
first = 'throat'
second = 'warbler'
print first + second
```

Η έξοδος αυτού του προγράμματος είναι `throatwarbler`.

Ο τελεστής `*` λειτουργεί επίσης στις συμβολοσειρές και εκτελεί επανάληψη. Για παράδειγμα, το `'Spam'*3` δίνει `'SpamSpamSpam'`. Εάν ένας από τους τελεστέους είναι συμβολοσειρά τότε ο άλλος πρέπει να είναι ακέραιος.

Αυτή η χρήση του `+` και του `*` έχει ανάλογη σημασία με την πρόσθεση και τον πολλαπλασιασμό. Όπως το `4*3` είναι ισοδύναμο με το `4+4+4`, έτσι και το `'Spam'*3` να είναι το ίδιο με το `'Spam'+'Spam'+'Spam'`. Από την άλλη, υπάρχει μία συγκεκριμένη περίπτωση στην οποία η συνένωση και η επανάληψη συμβολοσειρών είναι διαφορετικές από την πρόσθεση και τον πολλαπλασιασμό ακεραίων. Μπορείτε να σκεφτείτε μία ιδιότητα την οποία έχει η πρόσθεση αλλά όχι η συνένωση συμβολοσειρών;

2.9 Σχόλια

Όσο τα προγράμματα θα μεγαλώνουν και θα περιπλέκονται τόσο θα γίνονται και πιο δυσανάγνωστα. Οι τυπικές (formal) γλώσσες είναι πυκνογραμμένες και συχνά είναι δύσκολο να καταλάβει κανείς τι κάνει ή γιατί το κάνει αυτό ένα κομμάτι του κώδικα.

Για αυτό το λόγο είναι προτιμότερο να προσθέτετε σημειώσεις στα προγράμματά σας στις οποίες να εξηγείτε σε φυσική γλώσσα τι κάνει το πρόγραμμα. Αυτές οι σημειώσεις ονομάζονται **σχόλια** και ξεκινάνε με το σύμβολο # :

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

Σε αυτή τη περίπτωση το σχόλιο εμφανίζεται σε μία γραμμή μόνο του. Μπορείτε να βάζετε σχόλια και στο τέλος μίας γραμμής μετά τη δήλωση:

```
percentage = (minute * 100) / 60     # percentage of an hour
```

Οτιδήποτε από το # μέχρι το τέλος της γραμμής αγνοείται και δεν έχει καμία επίδραση στο πρόγραμμα.

Τα σχόλια είναι περισσότερο χρήσιμα όταν τεκμηριώνουν τα μη προφανή χαρακτηριστικά του κώδικα. Είναι λογικό να υποθέσετε ότι ο αναγνώστης μπορεί να καταλάβει τι κάνει το πρόγραμμα αλλά είναι πολύ πιο χρήσιμο να εξηγήσετε το γιατί.

Αυτό το σχόλιο είναι περιττό για τον κώδικα και άχρηστο:

```
v = 5     # assign 5 to v
```

Αυτό το σχόλιο περιέχει χρήσιμη πληροφορία η οποία δεν είναι μέσα στον κώδικα:

```
v = 5     # velocity in meters/second.
```

Καλά ονόματα μεταβλητών μπορούν να μειώσουν την ανάγκη για σχόλια αλλά μεγάλα ονόματα μπορεί να κάνουν δυσανάγνωστες τις σύνθετες εκφράσεις. Επομένως πρέπει να υπάρχει κάποιου είδους συμβιβασμός.

2.10 Αποσφαλμάτωση

Σε αυτό το σημείο, το πιο πιθανό συντακτικό λάθος που ενδέχεται να κάνετε είναι ένα μη έγκυρο όνομα μεταβλητής όπως το `class` ή το `yield`, τα οποία είναι λέξεις-κλειδιά, ή το `odd~job` και το `US$`, τα οποία περιέχουν μη έγκυρους χαρακτήρες.

Εάν βάλετε ένα κενό μέσα στο όνομα κάποιας μεταβλητής η Python νομίζει ότι είναι δύο τελεστέοι χωρίς τελεστή:

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

Στα συντακτικά λάθη τα μηνύματα λάθους δεν βοηθούν και πολύ. Τα συνηθέστερα μηνύματα είναι `SyntaxError: invalid syntax` και `SyntaxError: invalid token`, από τα οποία κανένα δεν παρέχει αρκετή πληροφορία.

Το λάθος χρόνου εκτέλεσης που είναι πιθανότερο να κάνετε είναι ένα "use before def", δηλαδή να προσπαθείτε να χρησιμοποιείτε μία μεταβλητή προτού να της εκχωρήσετε κάποια τιμή. Αυτό μπορεί να συμβεί εάν πληκτρολογήσετε ένα όνομα μεταβλητής λάθος:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Τα ονόματα μεταβλητών είναι ευαίσθητα όσον αφορά το διαχωρισμό μικρών και κεφαλαίων γραμμάτων, έτσι το `LaTeX` είναι διαφορετικό από το `latex`.

Σε αυτό το σημείο, η πιθανότερη αιτία σημασιολογικού λάθους είναι η σειρά των πράξεων. Για παράδειγμα, αν θέλατε να υπολογίσετε το $\frac{1}{2\pi}$ ίσως μπαίνατε στον πειρασμό να γράψετε:

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

Αλλά η διαίρεση συμβαίνει πρώτη και άρα θα πάρετε $\pi/2$, το οποίο δεν είναι το ίδιο πράγμα! Δεν είναι δυνατόν να γνωρίζει η Python τι θέλατε να γράψετε, έτσι σε αυτή την περίπτωση δεν θα παραχθεί κάποιο μήνυμα λάθους, απλώς θα πάρετε λάθος απάντηση.

2.11 Ορολογία

τιμή: Μία από τις βασικές μονάδες δεδομένων όπως ένας αριθμός ή μία συμβολοσειρά, που διαχειρίζεται ένα πρόγραμμα.

τύπος: Μία κατηγορία τιμών. Οι τύποι οι οποίοι έχουμε συναντήσει μέχρι στιγμής είναι οι ακέραιοι (τύπος `int`), οι δεκαδικοί αριθμοί (τύπος `float`) και οι συμβολοσειρές (τύπος `str`).

ακέραιος: Ένας τύπος που αναπαριστά ακέραιους αριθμούς.

δεκαδικός: Ένας τύπος ο οποίος αναπαριστά αριθμούς με κλασματικό μέρος (δεκαδικά ψηφία).

συμβολοσειρά: Ένας τύπος ο οποίος αναπαριστά μία ακολουθία χαρακτήρων.

μεταβλητή: Ένα όνομα το οποίο αναφέρεται σε μία τιμή.

δήλωση: Ένα τμήμα κώδικα το οποίο αναπαριστά μία εντολή ή πράξη. Μέχρι στιγμής, οι δηλώσεις που έχουμε δει είναι εκχωρήσεις (ανάθεση τιμής) και δηλώσεις εκτύπωσης.

εκχώρηση: Μία δήλωση η οποία εκχωρεί μία τιμή σε μια μεταβλητή.

διάγραμμα κατάστασης: Μία γραφική αναπαράσταση ενός συνόλου μεταβλητών και των τιμών στις οποίες αναφέρονται.

λέξη κλειδί: Μια δεσμευμένη λέξη η οποία χρησιμοποιείται από τον μεταγλωττιστή ή τον διερμηνέα για να αναλύσει ένα πρόγραμμα. Δεν μπορείτε να χρησιμοποιήσετε λέξεις κλειδιά όπως το `if`, το `def` και το `while` σαν ονόματα μεταβλητών.

τελεστής: Ένα ειδικό σύμβολο το οποίο αναπαριστά ένα απλό υπολογισμό όπως η πρόσθεση, ο πολλαπλασιασμός ή συνένωση συμβολοσειρών.

τελεστέος: Μία από τις τιμές στις οποίες ενεργεί ο τελεστής.

ακέραια διαίρεση: Η πράξη η οποία διαίρει δύο αριθμούς και κόβει το δεκαδικό μέρος.

έκφραση: Ένας συνδυασμός μεταβλητών, τελεστών και τιμών ο οποίος έχει σαν αποτέλεσμα μία μοναδική τιμή.

αποτίμηση: Η απλοποίηση μίας έκφρασης εκτελώντας πράξεις προκειμένου να δώσει μία μόνο τιμή.

κανόνες προτεραιότητας: Το σύνολο των κανόνων βάσει των οποίων καθορίζεται η σειρά των πράξεων σε εκφράσεις που περιέχουν πολλούς τελεστές και τελεστέους.

συνένωση: Η σύνδεση δύο τελεστέων σε μία νέα ενιαία τιμή.

σχόλιο: Πληροφορίες μέσα σε ένα πρόγραμμα που προορίζονται για άλλους προγραμματιστές (ή οποιονδήποτε διαβάζει τον πηγαίο κώδικα) και δεν επηρεάζει την εκτέλεση του προγράμματος.

2.12 Ασκήσεις

Άσκηση 2.3. *Υποθέστε ότι εκτελούμε τις ακόλουθες δηλώσεις εκχώρησης:*

```
width = 17
height = 12.0
delimiter = '.'
```

Για καθεμία από τις ακόλουθες εκφράσεις, γράψτε την τιμή και τον τύπο της τιμής της έκφρασης.

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

Χρησιμοποιήστε τον διερμηνέα της Python για να ελέγξετε τις απαντήσεις σας.

Άσκηση 2.4. *Κάντε εξάσκηση χρησιμοποιώντας τον διερμηνέα της Python σαν αριθμομηχανή:*

1. *Ο όγκος μιας σφαίρας με ακτίνα r είναι $\frac{4}{3}\pi r^3$. Ποιος είναι ο όγκος μιας σφαίρας με ακτίνα 5 ; Σημείωση: το 392.7 είναι λάθος!*
2. *Υποθέστε ότι η αρχική τιμή ενός βιβλίου είναι $24.95, αλλά τα βιβλιοπωλεία έχουν 40% έκπτωση. Τα έξοδα μεταφοράς είναι $3 για το πρώτο αντίτυπο και 75 cents για κάθε επιπλέον αντίτυπο. Ποιο είναι το συνολικό κόστος για 60 αντίτυπα ;*
3. *Εάν φύγω από το σπίτι μου στις 6:52 π.μ. και τρέξω 1 μίλι σε αργό ρυθμό (8:15 ανά μίλι), μετά 3 μίλια με ρυθμό (7:12 ανά μίλι) και ένα μίλι πάλι σε αργό ρυθμό, τι ώρα θα έχω γυρίσει σπίτι για πρωινό;*

Κεφάλαιο 3

Συναρτήσεις

3.1 Κλήσεις συναρτήσεων

Στα πλαίσια του προγραμματισμού, μία **συνάρτηση** (function) είναι μία ακολουθία δηλώσεων με ένα συγκεκριμένο όνομα η οποία εκτελεί έναν υπολογισμό. Όταν ορίζουμε μια συνάρτηση δηλώνουμε το όνομα και την ακολουθία των δηλώσεων και στη συνέχεια μπορούμε να την καλέσουμε με το όνομά της. Έχουμε ήδη δει ένα παράδειγμα κλήσης συνάρτησης:

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

Το όνομα της συνάρτησης είναι type και η έκφραση μέσα στις παρενθέσεις ονομάζεται **όρισμα** (argument) της συνάρτησης. Το αποτέλεσμα γι αυτή την συνάρτηση είναι ο τύπος του ορίσματος.

Συνηθίζεται να λέμε πως η συνάρτηση ”παίρνει” ένα ή περισσότερα ορίσματα και ”επιστρέφει” κάποιο αποτέλεσμα. Το αποτέλεσμα αυτό ονομάζεται **επιστρεφόμενη τιμή** (return value).

3.2 Συναρτήσεις μετατροπής τύπων

Η Python παρέχει ενσωματωμένες συναρτήσεις οι οποίες μετατρέπουν τιμές από ένα τύπο σε έναν άλλο. Η συνάρτηση `int` δέχεται οποιαδήποτε τιμή και την μετατρέπει σε ακέραιο (αν μπορεί), διαφορετικά διαμαρτύρεται:

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

Η `int` μετατρέπει αριθμούς κινητής υποδιαστολής σε ακεραίους αλλά δεν κάνει στρογγυλοποίηση. Παραλείπει δηλαδή το δεκαδικό μέρος:

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

Η `float` μετατρέπει ακεραίους και συμβολοσειρές σε δεκαδικούς αριθμούς:

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

Και τέλος η `str` μετατρέπει το όρισμά της σε μία συμβολοσειρά:

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

3.3 Μαθηματικές Συναρτήσεις

Η Python έχει μια μαθηματική μονάδα λογισμικού (math module) η οποία περιέχει τις πιο γνωστές μαθηματικές συναρτήσεις. Μία **μονάδα λογισμικού** ή αλλιώς άρθρωμα **άρθρωμα** (module) είναι ένα αρχείο το οποίο περιέχει μια συλλογή από συσχετιζόμενες συναρτήσεις.

Προτού χρησιμοποιήσουμε μια μονάδα θα πρέπει να την εισάγουμε:

```
>>> import math
```

Αυτή η δήλωση δημιουργεί ένα **αντικείμενο της μονάδας λογισμικού** (module object) που ονομάζεται math. Εάν πληκτρολογήσετε ”print math” στο διερμηνέα θα πάρετε κάποιες πληροφορίες σχετικά με αυτή:

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

Το αντικείμενο της μονάδας περιέχει συναρτήσεις και μεταβλητές οι οποίες έχουν οριστεί στην μονάδα. Για να έχετε πρόσβαση σε μία από τις συναρτήσεις θα πρέπει να προσδιορίσετε το όνομα της μονάδας και το όνομα της συνάρτησης χωρισμένα με μία τελεία. Αυτή η μορφή ονομάζεται **συμβολισμός με τελεία** (dot notation).

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

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

Το πρώτο παράδειγμα χρησιμοποιεί την `log10` για να υπολογίσει το λόγο σήματος προς θόρυβο σε ντεσιμπέλ dB (θεωρούμε ότι οι μεταβλητές `signal_power` και `noise_power` έχουν οριστεί). Η μαθηματική μονάδα παρέχει επίσης τη συνάρτηση `log` η οποία υπολογίζει λογάριθμους βάσης `e`.

Το δεύτερο παράδειγμα βρίσκει το ημίτονο των ακτινίων (radians). Το όνομα της μεταβλητής υπαινίσσεται ότι η `sin` και οι άλλες τριγωνομετρικές συναρτήσεις (`cos`, `tan`, κλπ.) παίρνουν ακτίνια σαν ορίσματα. Για να μετατραπούν οι μοίρες σε ακτίνια, διαιρείτε με 360 και πολλαπλασιάζετε με 2π:

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

Η έκφραση `math.pi` χρησιμοποιεί την μεταβλητή `pi`, η τιμή της οποίας είναι μια προσέγγιση του π με ακρίβεια περίπου 15 ψηφίων, από τη μαθηματική μονάδα.

Μπορούμε να ελέγξετε το προηγούμενο αποτέλεσμα βάση τριγωνομετρίας συγκρίνοντάς το με την τετραγωνική ρίζα του δύο διαιρούμενη με το δύο:

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

3.4 Σύνθεση

Μέχρι στιγμής, έχουμε δει τα στοιχεία ενός προγράμματος (μεταβλητές, εκφράσεις και δηλώσεις) μεμονωμένα χωρίς όμως να έχουμε μιλήσει για το πως τα συνδυάσουμε.

Ένα από τα πιο χρήσιμα χαρακτηριστικά των γλωσσών προγραμματισμού είναι η ικανότητα τους να παίρνουν μικρά δομικά στοιχεία και να τα **συνθέτουν**. Για παράδειγμα, το όρισμα μιας συνάρτησης μπορεί να είναι οποιοδήποτε είδος έκφρασης συμπεριλαμβανομένου και αριθμητικών τελεστών:

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

Ακόμη και κλήσεις συναρτήσεων:

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

Μπορείτε να βάλετε και μια έκφραση σχεδόν οπουδήποτε όπου μπορείτε να βάλετε και μια τιμή με μία εξαίρεση: το αριστερό μέρος μιας δήλωσης εκχώρησης πρέπει να είναι ένα όνομα μεταβλητής. Οποιαδήποτε άλλη έκφραση στο αριστερό μέρος είναι ένα συντακτικό λάθος (θα δούμε τις εξαιρέσεις γι αυτόν τον κανόνα αργότερα).

```
>>> minutes = hours * 60                # right
>>> hours * 60 = minutes                # wrong!
SyntaxError: can't assign to operator
```

3.5 Προσθέτοντας νέες συναρτήσεις

Μέχρι στιγμής έχουμε χρησιμοποιήσει συναρτήσεις οι οποίες περιλαμβάνονται τη Python αλλά μπορούμε να προσθέσουμε και καινούργιες συναρτήσεις. Ο **ορισμός της συνάρτησης** (function definition) προσδιορίζει το όνομα μιας νέας συνάρτησης και τη σειρά των δηλώσεων οι οποίες εκτελούνται όταν καλείται η συνάρτηση.

Να ένα παράδειγμα:

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

Η `def` είναι μία λέξη κλειδί η οποία υποδεικνύει ότι πρόκειται για ορισμό συνάρτησης. Το όνομα της συνάρτησης είναι το `print_lyrics`. Οι κανόνες για τα ονόματα των συναρτήσεων είναι ίδιοι με αυτούς των ονομάτων μεταβλητών: γράμματα, αριθμοί και κάποια σημεία στίξης είναι επιτρεπτά αλλά ο πρώτος χαρακτήρας δεν μπορεί να είναι αριθμός. Δεν μπορείτε να χρησιμοποιήσετε μία λέξη κλειδί σαν όνομα συνάρτησης και θα πρέπει να αποφεύγετε να έχετε μια μεταβλητή και μια συνάρτηση με το ίδιο όνομα.

Οι κενές παρενθέσεις μετά το όνομα υποδεικνύουν ότι αυτή η συνάρτηση δεν παίρνει κανένα όρισμα.

Η πρώτη γραμμή του ορισμού της συνάρτησης ονομάζεται **επικεφαλίδα** (header) και το υπόλοιπο ονομάζεται **σώμα** (body) της συνάρτησης. Η επικεφαλίδα θα πρέπει να τελειώνει με μία άνω κάτω τελεία και το σώμα θα πρέπει να είναι ενδοπαραγραφοποιημένο. Κατά σύμβαση η ενδοπαραγραφοποίηση είναι σχεδόν πάντα τέσσερα κενά διαστήματα (βλ. παράγραφο 3.14) και το σώμα μπορεί να περιέχει οποιοδήποτε πλήθος δηλώσεων.

Οι συμβολοσειρές στις δηλώσεις εκτύπωσης (print) γράφονται μέσα σε διπλά εισαγωγικά παρόλο που τα μονά εισαγωγικά κάνουν το ίδιο πράγμα με τα διπλά. Οι περισσότεροι άνθρωποι χρησιμοποιούν τα μονά εισαγωγικά εκτός των περιπτώσεων όπως αυτή, όπου το μονό εισαγωγικό (το οποίο είναι επίσης και απόστροφος) εμφανίζεται μέσα στην συμβολοσειρά.

Εάν πληκτρολογήσετε τον ορισμό μιας συνάρτησης σε διαδραστική λειτουργία, ο διερμηνέας εμφανίζει στην οθόνη ... για να σας ενημερώσει ότι ο ορισμός δεν έχει ολοκληρωθεί:

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

Για να τερματίσετε τη συνάρτηση θα πρέπει να εισάγετε μία κενή γραμμή (αυτό δεν είναι απαραίτητο σε ένα σενάριο).

Ορίζοντας μία συνάρτηση δημιουργείται μία μεταβλητή με το ίδιο όνομα.

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

Η τιμή του `print_lyrics` είναι ένα **αντικείμενο συνάρτησης** (function object), το οποίο έχει τύπο `'function'`. Η σύνταξη για τη κλήση της νέας συνάρτησης είναι η ίδια με αυτή των ενσωματωμένων συναρτήσεων:

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

Από τη στιγμή που έχετε ορίσει μια συνάρτηση, μπορείτε να την χρησιμοποιήσετε μέσα σε μια άλλη συνάρτηση. Για παράδειγμα, για να επαναλάβουμε το προηγούμενο ρεφρέν, μπορούμε να γράψουμε μία συνάρτηση με όνομα `repeat_lyrics` :

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

Και μετά την καλούμε πληκτρολογώντας `repeat_lyrics` :

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

Αλλά το τραγούδι δεν πηγαίνει στην πραγματικότητα έτσι.

3.6 Ορισμοί και χρήσεις

Συγκεντρώνοντας τα κομμάτια κώδικα από την προηγούμενη ενότητα, το πρόγραμμα ολόκληρο φαίνεται κάπως έτσι:

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

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

Αυτό το πρόγραμμα περιέχει δύο ορισμούς συνάρτησης: τον `print_lyrics` και τον `repeat_lyrics`. Οι ορισμοί των συναρτήσεων εκτελούνται ακριβώς όπως οι άλλες δηλώσεις αλλά το αποτέλεσμα είναι η δημιουργία αντικειμένων συναρτήσεων. Οι δηλώσεις μέσα στη συνάρτηση δεν εκτελούνται μέχρις ότου να καλεστεί η συνάρτηση, και ο ορισμός της συνάρτησης δεν παράγει καμία έξοδο.

Όπως θα περιμένατε, πρέπει να δημιουργήσετε μία συνάρτηση για να μπορείτε να την εκτελέσετε. Με άλλα λόγια, ο ορισμός της συνάρτησης θα πρέπει να έχει ήδη εκτελεστεί προτού τη καλέσετε για πρώτη φορά.

Άσκηση 3.1. *Μετακινήστε την τελευταία γραμμή αυτού του προγράμματος στην κορυφή, έτσι ώστε η κλήση της συνάρτησης να εμφανίζεται πριν από τους ορισμούς. Τρέξτε το πρόγραμμα για να δείτε τι μήνυμα λάθους θα σας εμφανίσει.*
Άσκηση 3.2. *Μετακινήστε την κλήση της συνάρτησης ξανά στο τέλος του προγράμματος και μετακινήστε τον ορισμό της* `print_lyrics` *μετά τον ορισμό της* `repeat_lyrics`*. Τι συμβαίνει όταν τρέξετε αυτό το πρόγραμμα ;*

3.7 Ροή εκτέλεσης

Για να είστε σίγουροι ότι μία συνάρτηση έχει οριστεί πριν την πρώτη χρήση της, θα πρέπει να γνωρίζετε τη σειρά με την οποία εκτελούνται οι δηλώσεις και η οποία ονομάζεται **ροή εκτέλεσης** (flow of execution).

Η εκτέλεση ξεκινάει πάντα με την πρώτη δήλωση του προγράμματος και οι δηλώσεις εκτελούνται μία μία και με σειρά από πάνω προς τα κάτω.

Οι ορισμοί των συναρτήσεων δεν αλλάζουν την ροή εκτέλεσης του προγράμματος αλλά να θυμάστε ότι οι δηλώσεις μέσα στη συνάρτηση δεν εκτελούνται μέχρι να καλεστεί η συνάρτηση.

Μία κλήση συνάρτησης είναι σαν μία παράκαμψη στη ροή της εκτέλεσης. Αντί να πάει στην επόμενη δήλωση, η ροή πηδάει στο σώμα της συνάρτησης, εκτελεί όλες τις δηλώσεις εκεί και μετά επιστρέφει για να συνεχίσει από εκεί που σταμάτησε.

Αυτό μπορεί να φαίνεται αρκετά απλό αλλά θυμηθείτε ότι μία συνάρτηση μπορεί να καλέσει μία άλλη. Για παράδειγμα, στη μέση μιας συνάρτησης το πρόγραμμα μπορεί να χρειαστεί να εκτελέσει τις δηλώσεις μέσα σε μια άλλη συνάρτηση και καθώς εκτελεί αυτή τη νέα συνάρτηση ίσως εκτελέσει ακόμη μία συνάρτηση!

Ευτυχώς, η Python είναι καλή στο να ξέρει σε ποιο σημείο βρίσκεται. Έτσι, κάθε φορά που ολοκληρώνεται μία συνάρτηση, το πρόγραμμα επιστρέφει εκεί που σταμάτησε και όταν φτάσει στο τέλος του τερματίζει.

Ποιο είναι το ηθικό δίδαγμα αυτού του απεχθούς παραμυθιού; Όταν διαβάζετε ένα πρόγραμμα, δεν είναι πάντα βέλτιστο να το διαβάζετε από την κορυφή προς τα κάτω, ίσως είναι προτιμότερο να ακολουθείτε την ροή της εκτέλεσης μερικές φορές.

3.8 Παράμετροι και ορίσματα

Κάποιες από τις ενσωματωμένες συναρτήσεις που έχουμε δει απαιτούν ορίσματα. Για παράδειγμα, όταν καλείτε την `math.sin` περνάτε έναν αριθμό σαν όρισμα. Μερικές συναρτήσεις παίρνουν παραπάνω από ένα όρισμα: η `math.pow` παίρνει δύο, την βάση και τον εκθέτη.

Μέσα στην συνάρτηση τα ορίσματα εκχωρούνται σε μεταβλητές οι οποίες ονομάζονται **παράμετροι**. Αυτό είναι ένα παράδειγμα συνάρτησης ορισμένης από το χρήστη η οποία παίρνει ένα όρισμα:

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

Αυτή η συνάρτηση εκχωρεί το όρισμα σε μία παράμετρο με όνομα `bruce`. Όταν καλείται η συνάρτηση εμφανίζει την τιμή της παραμέτρου (όποια κι αν είναι αυτή) δύο φορές.

Αυτή η συνάρτηση δουλεύει με οποιαδήποτε τιμή μπορεί να εμφανιστεί.

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

Οι ίδιοι κανόνες σύνθεσης που εφαρμόζονται στις ενσωματωμένες συναρτήσεις, ισχύουν και στις οριζόμενες από το χρήστη συναρτήσεις. Έτσι, μπορούμε να χρησιμοποιήσουμε οποιοδήποτε είδος έκφρασης σαν όρισμα για την `print_twice`:

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

Τα ορίσματα αποτιμώνται πριν την κλήση της συνάρτησης. Άρα η έκφραση `'Spam '*4` και η έκφραση `math.cos(math.pi)` αποτιμώνται μόνο μία φορά.

Μπορείτε επίσης να χρησιμοποιήσετε και μία μεταβλητή σαν όρισμα:

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

```
Eric, the half a bee.
```

Το όνομα της μεταβλητής που περνάμε σαν όρισμα (michael) δεν έχει καμία σχέση με το όνομα της παραμέτρου (bruce). Δεν έχει σημασία τι τιμή θα επιστραφεί πίσω σε αυτόν που την κάλεσε, στην print_twice τους φωνάζουμε όλους bruce.

3.9 Οι μεταβλητές και οι παράμετροι είναι τοπικές

Όταν δημιουργείτε μια μεταβλητή μέσα σε μια συνάρτηση τότε αυτή είναι **τοπική** (local), το οποίο σημαίνει ότι υφίσταται μόνο μέσα στη συνάρτηση. Για παράδειγμα:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

Αυτή η συνάρτηση παίρνει δύο ορίσματα, τα συνενώνει και εμφανίζει το αποτέλεσμα δύο φορές. Αυτό είναι ένα παράδειγμα χρήσης της:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

Όταν η cat_twice τερματίσει η μεταβλητή cat καταστρέφεται και αν προσπαθσουμε να την εμφανίσουμε θα προκύψει μία εξαίρεση:

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

Τοπικές είναι επίσης και οι παράμετροι. Για παράδειγμα, έξω από την print_twice δεν υπάρχει κάτι αντίστοιχο της bruce.

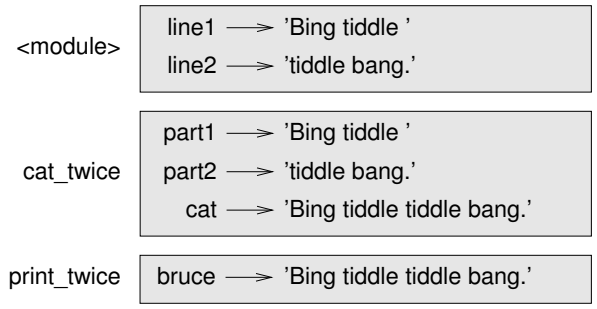
3.10 Διαγράμματα στοίβας

Κάποιες φορές, είναι χρήσιμο να σχεδιάζετε ένα **διάγραμμα στοίβας** (stack diagram) για να παρακολουθείτε που μπορεί να χρησιμοποιηθεί κάθε μεταβλητή. Όπως τα διαγράμματα κατάστασης, έτσι και τα διαγράμματα στοίβας δείχνουν την τιμή της κάθε μεταβλητής αλλά δείχνουν επίσης και τη συνάρτηση στην οποία ανήκει η κάθε μεταβλητή.

Κάθε συνάρτηση αναπαριστάται από ένα ***πλαίσιο*** (frame). Πλαίσιο είναι ένα κουτί δίπλα στο οποίο υπάρχει το όνομα μιας συνάρτησης και μέσα σε αυτό βρίσκονται οι παράμετροι και οι μεταβλητές. Το διάγραμμα στοίβας για το προηγούμενο παράδειγμα παρουσιάζεται στο Σχήμα 3.1.

Τα πλαίσια είναι διατεταγμένα σε μία στοίβα η οποία υποδεικνύει ποια συνάρτηση κάλεσε ποια και ούτω καθεξής. Σε αυτό το παράδειγμα, η print_twice καλέστηκε από την cat_twice και η cat_twice καλέστηκε από την __main__ η οποία είναι η συνάρτηση για το ανώτατο πλαίσιο. Όταν δημιουργείτε μία μεταβλητή έξω από οποιαδήποτε συνάρτηση τότε αυτή ανήκει στην __main__.

Κάθε παράμετρος αναφέρεται στην τιμή του αντίστοιχου ορίσματός της. Άρα η part1 έχει την ίδια τιμή με την line1, η part2 έχει την ίδια τιμή με την line2 και η bruce έχει την ίδια τιμή με την cat.



Σχήμα 3.1: Διάγραμμα στοίβας.

Αν προκύψει σφάλμα κατά την κλήση μιας συνάρτησης τότε η Python εμφανίζει το όνομα της συνάρτησης, το όνομα της συνάρτησης που την κάλεσε και το όνομα της συνάρτησης η οποία κάλεσε αυτήν μέχρις ότου φτάσει στην `__main__`.

Για παράδειγμα, αν προσπαθήσετε να χρησιμοποιήσετε την `cat` μέσα στην `print_twice` τότε θα πάρετε ένα μήνυμα λάθους `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print cat
NameError: name 'cat' is not defined
```

Αυτή η λίστα συναρτήσεων ονομάζεται **αναδρομική ανίχνευση** (traceback) και σας πληροφορεί σε ποιο φάκελο του προγράμματος παρουσιάστηκε το λάθος, σε ποια γραμμή και ποιες συναρτήσεις εκτελούνταν εκείνη τη στιγμή. Σας δείχνει επίσης τη γραμμή του κώδικα από την οποία προκλήθηκε το λάθος.

Η σειρά των συναρτήσεων στην αναδρομή είναι ίδια με την σειρά των πλαισίων στο διάγραμμα κατάστασης. Η συνάρτηση η οποία τρέχει αυτή τη στιγμή βρίσκεται στο κάτω μέρος.

3.11 Γόνιμες και κενές συναρτήσεις

Κάποιες από τις συναρτήσεις που χρησιμοποιούμε, όπως οι συναρτήσεις μαθηματικών, επιστρέφουν αποτελέσματα και λόγω έλλειψης κάποιου καλύτερου ονόματος τις αποκαλώ **γόνιμες συναρτήσεις** (fruitful functions). Άλλες συναρτήσεις, όπως η `print_twice`, εκτελούν μια ενέργεια αλλά δεν επιστρέφουν κάποια τιμή. Αυτές ονομάζονται **κενές συναρτήσεις** (void functions).

Σχεδόν κάθε φορά που καλείτε μία γόνιμη συνάρτηση θέλετε να χρησιμοποιήσετε και το αποτέλεσμα της. Για παράδειγμα, ίσως το εκχωρήσετε σε μία μεταβλητή ή το χρησιμοποιήσετε ως μέρος μιας έκφρασης:

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

Όταν καλείτε μια συνάρτηση σε διαδραστική λειτουργία τότε η Python εμφανίζει το αποτέλεσμα:

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

```
2.2360679774997898
```

Αλλά αν καλέσετε μια γόνιμη συνάρτηση σε ένα σενάριο από μόνη της, τότε η επιστρεφόμενη τιμή θα χαθεί για πάντα!

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

Αυτό το σενάριο υπολογίζει την τετραγωνική ρίζα του 5 αλλά από την στιγμή που δεν αποθηκεύει ή δεν εμφανίζει το αποτέλεσμα, δεν είναι και πολύ χρήσιμο.

Οι κενές (void) συναρτήσεις μπορεί να εμφανίζουν κάτι στην οθόνη ή να έχουν κάποιο αποτέλεσμα αλλά δεν έχουν επιστρεφόμενη τιμή. Αν προσπαθήσετε να εκχωρήσετε το αποτέλεσμα σε μία μεταβλητή τότε θα πάρετε μια ειδική τιμή με όνομα `None`.

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

Η τιμή `None` δεν είναι ίδια με την συμβολοσειρά `'None'`. Είναι μια ειδική τιμή η οποία έχει τον δικό της τύπο:

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

Όλες οι συναρτήσεις οι οποίες έχουμε γράψει μέχρι στιγμής είναι κενές (void). Θα ξεκινήσουμε να γράφουμε γόνιμες (fruitful) συναρτήσεις σε μερικά κεφάλαια.

3.12 Γιατί συναρτήσεις

Ίσως να μην είναι σαφής ο λόγος για τον οποίο αξίζει να χωρίζουμε ένα πρόγραμμα σε συναρτήσεις αλλά υπάρχουν αρκετοί λόγοι:

- Η δημιουργία μιας νέας συνάρτησης σας δίνει την δυνατότητα να ονοματίσετε ένα σύνολο δηλώσεων το οποίο κάνει το πρόγραμμα ευκολότερο στην ανάγνωση και στην αποσφαλμάτωση.
- Οι συναρτήσεις μπορούν να κάνουν ένα πρόγραμμα μικρότερο καταργώντας τον επαναλαμβανόμενο κώδικα. Αν κάνετε κάποια αλλαγή αργότερα τότε θα χρειαστεί να την κάνετε μόνο σε ένα σημείο.
- Η διαίρεση ενός μεγάλου προγράμματος σε συναρτήσεις σας επιτρέπει να αποσφαλματώνετε τα κομμάτια ένα ένα και μετά να τα ενώνετε όλα μαζί σε ένα ολόκληρο λειτουργικό πρόγραμμα.
- Οι καλοσχεδιασμένες συναρτήσεις είναι συχνά χρήσιμες για πολλά προγράμματα. Μόλις γράψετε μία και την αποσφαλματώσετε τότε μπορείτε να την χρησιμοποιήσετε πολλές φορές.

3.13 Εισαγωγή από μονάδα λογισμικού με from

Η Python παρέχει δύο τρόπους εισαγωγής μονάδων λογισμικού αλλά μέχρι στιγμής έχουμε ήδη δει μόνο αυτόν:

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

Αν εισάγετε την `math` τότε θα πάρετε ένα αντικείμενο μονάδας λογισμικού με όνομα `math` το οποίο περιέχει σταθερές όπως η `pi` και συναρτήσεις όπως η `sin` και η `exp`.

Δεν μπορείτε όμως να αποκτήσετε απευθείας πρόσβαση στην `pi`, θα πάρετε ένα μήνυμα λάθους:

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

Εναλλακτικά, μπορείτε να εισάγετε ένα αντικείμενο από μια μονάδα έτσι:

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

Τώρα μπορείτε να έχετε απευθείας πρόσβαση στην `pi` χωρίς τον συμβολισμό τελείας:

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

Ή μπορείτε να χρησιμοποιήσετε τον τελεστή αστεράκι για να τα εισάγετε όλα από την μονάδα:

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

Το πλεονέκτημα του να εισάγουμε τα πάντα από τη μαθηματική μονάδα λογισμικού είναι ότι μπορεί ο κώδικάς σας να είναι πιο συνοπτικός. Το μειονέκτημα είναι ότι μπορεί να προκύψουν συγκρούσεις στα ονόματα που έχουν οριστεί στις διάφορες μονάδες ή μεταξύ ενός ονόματος από μία μονάδα και μίας από τις μεταβλητές σας.

3.14 Αποσφαλμάτωση

Ίσως αντιμετωπίσετε προβλήματα με τα κενά διαστήματα και τους στηλοθέτες (tabs) αν χρησιμοποιείτε έναν επεξεργαστή κειμένου για να γράψετε τα σενάριά σας. Ο καλύτερος τρόπος για να αποφύγετε τέτοιου είδους προβλήματα είναι να χρησιμοποιείτε αποκλειστικά κενά διαστήματα (καθόλου στηλοθέτες). Οι περισσότεροι επεξεργαστές κειμένου που γνωρίζουν Python το κάνουν αυτό εξ ορισμού αλλά κάποιοι άλλοι δεν το κάνουν.

Συνήθως οι στηλοθέτες και τα κενά διαστήματα είναι αόρατα, γεγονός που δυσκολεύει την αποσφαλμάτωση και γι αυτό είναι προτιμότερο να βρείτε έναν επεξεργαστή που θα διαχειρίζεται αυτός την ενδοπαραγραφοποίηση για εσάς.

Επίσης, μην ξεχνάτε να αποθηκεύετε το πρόγραμμά σας προτού το τρέξετε. Κάποια περιβάλλοντα ανάπτυξης το κάνουν αυτόματα αλλά κάποια άλλα όχι. Σε αυτήν την περίπτωση, το πρόγραμμα που κοιτάτε στον επεξεργαστή κειμένου δεν είναι το ίδιο με το πρόγραμμα που τρέχετε.

Η αποσφαλμάτωση μπορεί να είναι χρονοβόρα αν τρέχετε συνεχώς το ίδιο εσφαλμένο πρόγραμμα ξανά και ξανά!

Σιγουρευτείτε πως ο κώδικας που κοιτάτε είναι ο κώδικας που τρέχετε. Αν δεν είστε σίγουροι, γράψτε μία `print 'hello'` στην αρχή του προγράμματός σας και τρέξτε το ξανά. Αν δεν δείτε `hello` τότε σημαίνει ότι δεν τρέχετε το σωστό πρόγραμμα!

3.15 Ορολογία

συνάρτηση: Μία ακολουθία δηλώσεων με συγκεκριμένη ονομασία η οποία εκτελεί κάποιες χρήσιμες λειτουργίες. Οι συναρτήσεις μπορεί να παίρνουν ορίσματα αλλά μπορεί και όχι. Επίσης μπορεί να παράγουν κάποιο αποτέλεσμα αλλά μπορεί και όχι.

ορισμός συνάρτησης: Είναι μία δήλωση η οποία δημιουργεί μια νέα συνάρτηση προσδιορίζοντας το όνομα της, τις παραμέτρους και τις δηλώσεις που εκτελεί.

αντικείμενο συνάρτησης: Μία τιμή η οποία δημιουργείται από έναν ορισμό συνάρτησης. Το όνομα της συνάρτησης είναι μία μεταβλητή η οποία αναφέρεται σε ένα αντικείμενο συνάρτησης.

επικεφαλίδα: Η πρώτη γραμμή του ορισμού μιας συνάρτησης.

σώμα: Η ακολουθία δηλώσεων μέσα σε έναν ορισμό συνάρτησης.

παράμετρος: Μία μεταβλητή που χρησιμοποιείται μέσα σε μία συνάρτηση και στην οποία αποδίδεται η τιμή που περνιέται σαν όρισμα.

κλήση συνάρτησης: Μία δήλωση η οποία εκτελεί μια συνάρτηση. Αποτελείται από το όνομα της συνάρτησης ακολουθούμενο από μία λίστα ορισμάτων.

όρισμα: Μία τιμή η οποία παρέχεται σε μια συνάρτηση όταν αυτή καλείται. Αυτή η τιμή εκχωρείται στην αντίστοιχη παράμετρο στη συνάρτηση.

τοπική μεταβλητή: Μία μεταβλητή ορισμένη μέσα σε μια συνάρτηση, η οποία μπορεί να χρησιμοποιηθεί (έχει εμβέλεια) μόνο μέσα στη συνάρτησή της.

επιστρεφόμενη τιμή: Το αποτέλεσμα μιας συνάρτησης. Αν μια κλήση συνάρτησης χρησιμοποιηθεί σαν έκφραση, η επιστρεφόμενη τιμή είναι η τιμή της έκφρασης.

γόνιμη συνάρτηση: Μία συνάρτηση η οποία επιστρέφει μια τιμή.

κενή (void) συνάρτηση: Μία συνάρτηση η οποία δεν επιστρέφει κάποια τιμή.

μονάδα λογισμικού ή άρθρωμα (module): Ένα αρχείο το οποίο περιέχει μία συλλογή συναφών συναρτήσεων και άλλους ορισμούς.

δήλωση εισαγωγής (import): Μία δήλωση η οποία διαβάζει μια μονάδα λογισμικού και δημιουργεί ένα αντικείμενο της μονάδας.

αντικείμενο μονάδας λογισμικού: Μία τιμή που δημιουργείται από την δήλωση `import` και παρέχει πρόσβαση στις τιμές που ορίζονται μέσα σε μία μονάδα λογισμικού.

συμβολισμός με τελεία: Η σύνταξη για την κλήση μιας συνάρτησης από μια μονάδα λογισμικού καθορίζοντας το όνομα της μονάδας ακολουθούμενο από μία τελεία και στη συνέχεια το όνομα της συνάρτησης.

σύνθεση: Η χρήση μιας έκφρασης ως μέρος μιας μεγαλύτερης έκφρασης ή μιας δήλωσης ως μέρος μιας μεγαλύτερης δήλωσης.

ροή εκτέλεσης: Η σειρά με την οποία εκτελούνται οι δηλώσεις κατά την εκτέλεση του προγράμματος.

διάγραμμα στοίβας: Μία γραφική αναπαράσταση μιας στοίβας από συναρτήσεις, των μεταβλητών τους και των τιμών στις οποίες αναφέρονται.

πλαίσιο: Ένα κουτί σε ένα διάγραμμα στοίβας το οποίο αναπαριστά μια κλήση συνάρτησης. Περιέχει τις τοπικές μεταβλητές και τις παραμέτρους της συνάρτησης.

αναδρομική ανίχνευση: Μία λίστα των συναρτήσεων που εκτελούνται και η οποία τυπώνεται όταν παρουσιαστεί μια εξαίρεση.

3.16 Ασκήσεις

Άσκηση 3.3. *Η Python έχει μία ενσωματωμένη συνάρτηση που ονομάζεται* `len` *και η οποία επιστρέφει το μήκος μιας συμβολοσειράς. Άρα η επιστρεφόμενη τιμή της* `len('allen')` *είναι 5.*

Γράψτε μία συνάρτηση με όνομα `right_justify` *η οποία θα παίρνει μία συμβολοσειρά με όνομα* `s` *σαν παράμετρο και θα εμφανίζει την συμβολοσειρά με αρκετά κενά διαστήματα έτσι ώστε το τελευταίο γράμμα της συμβολοσειράς να είναι στη στήλη 70 της οθόνης.*

```
>>> right_justify('allen')
                                                                allen
```

Άσκηση 3.4. *Ένα αντικείμενο συνάρτησης είναι μια τιμή την οποία μπορείτε να εκχωρήσετε σε μία μεταβλητή ή να την περάσετε σαν όρισμα. Για παράδειγμα, η* `do_twice` *είναι μία συνάρτηση η οποία παίρνει ένα αντικείμενο συνάρτησης σαν ένα όρισμα και το καλεί δύο φορές:*

```
def do_twice(f):
    f()
    f()
```

Αυτό είναι ένα παράδειγμα το οποίο χρησιμοποιεί την `do_twice` *για να καλέσει δύο φορές μία συνάρτηση που ονομάζεται* `print_spam`*:*

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


do_twice(print_spam)
```

1. *Πληκτρολογήστε αυτό το παράδειγμα σε ένα σενάριο και ελέγξτε το.*
2. *Τροποποιήστε την* `do_twice` *έτσι ώστε να δέχεται δύο ορίσματα, ένα αντικείμενο συνάρτησης και μία τιμή και να καλεί την συνάρτηση δύο φορές, περνώντας την τιμή σαν όρισμα.*
3. *Γράψτε μία γενικότερη εκδοχή της* `print_spam`*, με όνομα* `print_twice` *η οποία θα παίρνει μια συμβολοσειρά σαν παράμετρο και θα την τυπώνει δύο φορές.*
4. *Χρησιμοποιήστε την τροποποιημένη εκδοχή της* `do_twice` *για να καλέσετε την* `print_twice` *δύο φορές, περνώντας την* `'spam'` *σαν όρισμα.*
5. *Προσδιορίστε μία νέα συνάρτηση με όνομα* `do_four` *η οποία θα παίρνει ένα αντικείμενο συνάρτησης και μία τιμή και θα καλεί την συνάρτηση τέσσερις φορές, περνώντας την τιμή σαν παράμετρο. Πρέπει να υπάρχουν μόνο δύο δηλώσεις στο σώμα αυτής της συνάρτησης, όχι τέσσερις.*

Λύση: `http://thinkpython.com/code/do_four.py`*.*

Άσκηση 3.5. *Αυτή η άσκηση μπορεί να υλοποιηθεί με τη χρήση δηλώσεων και άλλων χαρακτηριστικών που έχουμε μάθει μέχρι στιγμής.*

1. *Γράψτε μία συνάρτηση η οποία σχεδιάζει ένα πλέγμα όπως το ακόλουθο :*

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

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

Σημείωση: Για να εμφανίσετε περισσότερες από μία τιμές σε μία γραμμή μπορείτε να τυπώσετε μία ακολουθία χωρισμένη με κόμματα:

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

Αν η ακολουθία τελειώνει με κόμμα, η Python αφήνει την γραμμή ημιτελή έτσι ώστε η επόμενη εμφανιζόμενη τιμή να εμφανιστεί στην ίδια γραμμή.

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

Η έξοδος αυτών των δηλώσεων είναι `'+ -'`.

Μία δήλωση `print` *από μόνη της ολοκληρώνει την τρέχουσα γραμμή και πηγαίνει στην επόμενη γραμμή.*

2. *Γράψτε μία συνάρτηση η οποία θα σχεδιάζει ένα παρόμοιο πλέγμα με τέσσερις σειρές και τέσσερις στήλες.*

Λύση: `http://thinkpython.com/code/grid.py`*. Αυτή η άσκηση είναι βασισμένη σε μία άσκηση του Oualline,* Practical C Programming, Third Edition*, O'Reilly Media, 1997.*

Κεφάλαιο 4

Μελέτη περίπτωσης: σχεδίαση διεπαφής

Τα παραδείγματα κώδικα αυτού του κεφαλαίου είναι διαθέσιμα στην διεύθυνση `http://thinkpython.com/code/polygon.py`.

4.1 TurtleWorld

Για να συνοδεύσει αυτό το βιβλίο, ο καθηγητής Allen έχει γράψει ένα πακέτο λογισμικού που ονομάζεται Swampy. Μπορείτε να κατεβάσετε το Swampy από τον σύνδεσμο `http://thinkpython.com/swampy` και να ακολουθήσετε τις οδηγίες εκεί για να το εγκαταστήσετε στον υπολογιστή σας.

Ένα **πακέτο λογισμικού** (package) είναι μια συλλογή από μονάδες λογισμικού. Μία από τις μονάδες του Swampy είναι η `TurtleWorld`, η οποία παρέχει ένα σύνολο συναρτήσεων για τον σχεδιασμό γραμμών κατευθύνοντας χελώνες στην οθόνη.

Αν το Swampy έχει εγκατασταθεί σαν ένα πακέτο λογισμικού στον υπολογιστή σας, τότε μπορείτε να εισάγετε την `TurtleWorld` με τον εξής τρόπο:

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

Αν κατεβάσατε τις μονάδες του Swampy αλλά δεν τις εγκαταστήσατε σαν πακέτο, τότε μπορείτε είτε να δουλέψετε στον κατάλογο ο οποίος περιέχει τα αρχεία του Swampy είτε να προσθέσετε αυτόν τον κατάλογο στην διαδρομή αναζήτησης της Python. Τώρα μπορείτε να εισάγετε την `TurtleWorld` έτσι:

```
from TurtleWorld import *
```

Οι λεπτομέρειες της διαδικασίας εγκατάστασης και της προσθήκης της διαδρομής αναζήτησης στην Python εξαρτώνται από το σύστημά σας. Γι' αυτό το λόγο, αντί να συμπεριλάβω αυτές τις λεπτομέρειες εδώ, θα προσπαθήσω να διατηρήσω τις τρέχουσες πληροφορίες για διάφορα συστήματα στην διεύθυνση: `http://thinkpython.com/swampy`.

Δημιουργήστε ένα αρχείο με όνομα `mypolygon.py` και γράψτε τον ακόλουθο κώδικα:

```
from swampy.TurtleWorld import *

world = TurtleWorld()
```

```
bob = Turtle()
print bob

wait_for_user()
```

Η πρώτη γραμμή εισάγει ό,τι υπάρχει στην μονάδα `TurtleWorld` του πακέτου `swampy`.

Οι επόμενες γραμμές δημιουργούν την TurtleWorld η οποία εκχωρείται στην `world` και μία Turtle η οποία εκχωρείται στην `bob`. Τυπώνοντας την `bob` παράγεται κάτι τέτοιο:

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

Αυτό σημαίνει ότι η `bob` αναφέρεται σε ένα **στιγμιότυπο** (instance) της Turtle όπως αυτή ορίζεται στην μονάδα λογισμικού `TurtleWorld`. Σε αυτό το πλαίσιο, το "στιγμιότυπο" είναι ένα τμήμα ενός συνόλου. Αυτή η Turtle είναι μία από το σύνολο των εφικτών Turtles.

Η `wait_for_user` λέει στην TurtleWorld να περιμένει από τον χρήστη να κάνει κάτι, παρόλο που σε αυτήν τη περίπτωση δεν υπάρχουν και πολλά που μπορεί να κάνει ο χρήστης εκτός από το να κλείσει το παράθυρο.

Η TurtleWorld παρέχει αρκετές συναρτήσεις "χελωνο-οδήγησης": την `fd` και την `bk` για μπροστά και πίσω, την `lt` και την `rt` για στροφή στα αριστερά και στα δεξιά. Επίσης, η κάθε χελώνα κρατάει ένα στυλό ο οποίος βρίσκεται είτε κάτω είτε πάνω. Αν το στυλό βρίσκεται κάτω τότε η χελώνα αφήνει μια γραμμή όταν κινείται. Οι συναρτήσεις `pu` και `pd` αντιπροσωπεύουν τα "pen up" και "pen down".

Προσθέστε αυτές τις γραμμές στο πρόγραμμα (αφού δημιουργήσετε την `bob` και προτού καλέσετε την `wait_for_user`) για να σχεδιάσετε μια ορθή γωνία:

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

Η πρώτη γραμμή λέει στην `bob` να κάνει 100 βήματα μπροστά. Η δεύτερη γραμμή της λέει να στρίψει αριστερά.

Όταν τρέξετε αυτό το πρόγραμμα, θα πρέπει να δείτε την `bob` να μετακινείται ανατολικά και μετά βόρεια, αφήνοντας δύο ευθύγραμμα τμήματα από πίσω.

Τροποποιήστε το πρόγραμμα για να σχεδιάσετε ένα τετράγωνο. Μη προχωρήσετε μέχρι να δείτε ότι δουλεύει.

4.2 Απλή επανάληψη

Κατά πάσα πιθανότητα γράψατε κάτι τέτοιο (χωρίς τον κώδικα ο οποίος δημιουργεί την TurtleWorld και περιμένει τον χρήστη):

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

fd(bob, 100)
lt(bob)

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

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

Μπορούμε να κάνουμε το ίδιο πράγμα πιο συνοπτικά με μία δήλωση `for`. Προσθέστε αυτό το παράδειγμα στην `mypolygon.py` και τρέξτε το ξανά:

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

Θα πρέπει να δείτε κάτι τέτοιο:

```
Hello!
Hello!
Hello!
Hello!
```

Αυτή είναι η πιο απλή χρήση της δήλωσης `for`. Αργότερα θα δούμε περισσότερες χρήσεις της αλλά αυτό είναι αρκετό ώστε να μπορείτε να ξαναγράψετε το πρόγραμμα σχεδιασμού τετραγώνων. Μην προχωρήσετε μέχρι να το κάνετε.

Αυτή είναι μια δήλωση `for` η οποία σχεδιάζει ένα τετράγωνο:

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

Η σύνταξη μίας δήλωσης `for` είναι παρόμοια με τον ορισμό μιας συνάρτησης. Έχει μία επικεφαλίδα που τελειώνει με μία άνω και κάτω τελεία και ένα σώμα εμφωλευμένο. Το σώμα μπορεί να έχει οποιοδήποτε πλήθος δηλώσεων.

Μία δήλωση `for` ονομάζεται και **βρόχος** (loop) επειδή η ροή εκτέλεσης περνά μέσα από το σώμα και στη συνέχεια επιστρέφει βροχοειδώς πίσω στην κορυφή. Στην συγκεκριμένη περίπτωση το σώμα εκτελείται τέσσερις φορές.

Στην πραγματικότητα, αυτή η έκδοση είναι λίγο διαφορετική από τον προηγούμενο κώδικα σχεδίασης τετραγώνων δεδομένου ότι κάνει μία ακόμη στροφή αφότου σχεδιάσει και την τελευταία πλευρά του τετραγώνου. Η επιπλέον στροφή παίρνει λίγο περισσότερο χρόνο αλλά απλοποιεί τον κώδικα αφού κάνουμε το ίδιο πράγμα κάθε φορά μέσα στο βρόχο. Αυτή η έκδοση έχει επίσης ως αποτέλεσμα να αφήνει τη χελώνα στην αρχική θέση, με φορά προς την αρχική κατεύθυνση.

4.3 Ασκήσεις

Ακολουθεί μία σειρά ασκήσεων οι οποίες χρησιμοποιούν τη TurtleWorld και στόχος τους είναι η διασκέδαση, αλλά παράλληλα έχουν κάποια σημασία. Προσπαθήστε να βρείτε τη σημασία τους καθώς δουλεύετε πάνω σε αυτές.

Οι παρακάτω ενότητες έχουν λύσεις για αυτές τις ασκήσεις, γι αυτό μην τις κοιτάξετε μέχρι να τις λύσετε (ή τουλάχιστον να προσπαθήστε).

1. Γράψτε μία συνάρτηση με όνομα `square` που θα παίρνει μια παράμετρο με όνομα `t`, η οποία θα είναι μία χελώνα (turtle). Θα πρέπει να χρησιμοποιεί τη χελώνα για να σχεδιάσει ένα τετράγωνο.

   Γράψτε μία κλήση συνάρτησης η οποία θα περνάει την `bob` σαν όρισμα στην `square` και μετά ξανατρέξτε το πρόγραμμα.

2. Προσθέστε μία ακόμα παράμετρο στη `square` με όνομα `length`. Στη συνέχεια τροποποιήστε το σώμα ώστε η `length` να είναι το μήκος των πλευρών και μετά τροποποιήστε την κλήση

της συνάρτησης δίνοντας ένα ακόμα όρισμα. Τρέξτε το πρόγραμμα ξανά και ελέγξτε το για διάφορες τιμές της `length`.

3. Οι συναρτήσεις `lt` και `rt` κάνουν στροφές 90 μοιρών από προεπιλογή, αλλά μπορείτε να δώσετε ένα δεύτερο όρισμα που θα καθορίζει τον αριθμό των μοιρών. Για παράδειγμα, η `lt(bob, 45)` περιστρέφει την bob 45 μοίρες στα αριστερά.

   Κάντε ένα αντίγραφο της `square` και αλλάξτε το όνομά της σε `polygon`. Προσθέστε μία παράμετρο με όνομα n και τροποποιήστε το σώμα ώστε να σχεδιάζει ένα κανονικό πολύγωνο n πλευρών. Σημείωση: Οι εξωτερικές γωνίες ενός κανονικού πολυγώνου n πλευρών είναι $360/n$ μοίρες.

4. Γράψτε μία συνάρτηση με όνομα `circle` η οποία θα παίρνει μία χελώνα `t` και μία ακτίνα `r` ως παραμέτρους και θα σχεδιάζει έναν κατά προσέγγιση κύκλο επικαλούμενη τη `polygon` με ένα κατάλληλο μήκος και αριθμό πλευρών. Ελέγξτε τη συνάρτησή σας με ένα εύρος τιμών της `r`.

   Σημείωση 1η: Υπολογίστε την περίμετρο του κύκλου και σιγουρευτείτε πως `length * n = circumference`.

   Σημείωση 2η: Αν η `bob` είναι πολύ αργή, μπορείτε να την επιταχύνετε αλλάζοντας την `bob.delay` η οποία είναι ο χρόνος μεταξύ των κινήσεων σε δευτερόλεπτα. Με `bob.delay = 0.01` θα πρέπει να κινείται γρηγορότερα.

5. Φτιάξτε μία πιο γενική έκδοση της `circle` με όνομα `arc` η οποία θα παίρνει μία επιπλέον παράμετρο με όνομα `angle` και η οποία θα καθορίζει ποιο τμήμα του κύκλου θα σχεδιαστεί. Η `angle` μετριέται σε μοίρες και έτσι όταν `angle=360` η `arc` θα πρέπει να σχεδιάσει ένα πλήρες κύκλο.

4.4 Ενθυλάκωση

Η πρώτη άσκηση σας ζητάει να βάλετε τον κώδικα σχεδίασης τετραγώνου σε έναν ορισμό συνάρτησης και μετά να καλέσετε τη συνάρτηση περνώντας τη χελώνα σαν παράμετρο. Αυτή είναι μία λύση:

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

square(bob)
```

Οι ενδότερες δηλώσεις `fd` και `lt` είναι διπλά εμφωλευμένες για να δείξουν ότι βρίσκονται εντός του βρόχου `for`, ο οποίος βρίσκεται μέσα στον ορισμό της συνάρτησης. Η επόμενη γραμμή, `square(bob)`, βρίσκεται στο ίδιο επίπεδο με το αριστερό όριο ώστε να είναι και στο τέλος του βρόχου `for` αλλά και του ορισμού της συνάρτησης.

Μέσα στη συνάρτηση, η `t` αναφέρεται στην ίδια χελώνα που αναφέρεται και η `bob`. Άρα η `lt(t)` έχει το ίδιο αποτέλεσμα με τη `lt(bob)`. Οπότε γιατί να μην καλέσουμε την παράμετρο `bob`; Η γενική ιδέα είναι ότι η `t` μπορεί να είναι οποιαδήποτε χελώνα (όχι μόνο η `bob`) και έτσι μπορείτε να δημιουργήσετε μία δεύτερη χελώνα και να την περάσετε ως όρισμα στη `square`:

```
ray = Turtle()
square(ray)
```

Το να συμπεριλάβετε ένα κομμάτι του κώδικα μέσα σε μία συνάρτηση ονομάζεται **ενθυλάκωση** (encapsulation). Ένα από τα πλεονεκτήματα της ενθυλάκωσης είναι ότι προσδίδει ένα όνομα στον κώδικα το οποίο χρησιμεύει ως είδος τεκμηρίωσης. Ένα άλλο πλεονέκτημα είναι πως αν θέλετε να χρησιμοποιήσετε ξανά τον κώδικα τότε είναι πιο λακωνικό να καλέσετε δύο φορές μία συνάρτηση παρά να αντιγράφετε ολόκληρο το σώμα της!

4.5 Γενίκευση

Το επόμενο βήμα είναι να προσθέσουμε μία παράμετρο `length` στην `square`. Λύση:

```
def square(t, length):
    for i in range(4):
        fd(t, length)
        lt(t)

square(bob, 100)
```

Η προσθήκη μίας παραμέτρου σε μία συνάρτηση ονομάζεται **γενίκευση** (generalization) διότι κάνει την συνάρτηση πιο γενική: στην προηγούμενη έκδοση το τετράγωνο είχε πάντα το ίδιο μέγεθος, ενώ σε αυτήν την έκδοση μπορεί να έχει οποιοδήποτε μέγεθος.

Το επόμενο βήμα είναι επίσης μία γενίκευση. Αντί η `polygon` να σχεδιάζει τετράγωνα, σχεδιάζει κανονικά πολύγωνα με οποιονδήποτε αριθμό πλευρών. Λύση:

```
def polygon(t, n, length):
    angle = 360.0 / n
    for i in range(n):
        fd(t, length)
        lt(t, angle)

polygon(bob, 7, 70)
```

Αυτός ο κώδικας σχεδιάζει ένα πολύγωνο 7 πλευρών με μήκος πλευράς 70. Αν έχετε περισσότερα από μερικά αριθμητικά ορίσματα τότε μπορεί να ξεχάσετε τι είναι τι ή σε ποια σειρά θα πρέπει να βρίσκονται. Είναι έγκυρο, και κάποιες φορές χρήσιμο να χρησιμοποιείτε τα ονόματα των παραμέτρων στη λίστα ορισμάτων:

```
polygon(bob, n=7, length=70)
```

Αυτά ονομάζονται **λέξεις-κλειδιά ορίσματα** (keyword arguments) επειδή περιέχουν τα ονόματα των παραμέτρων σαν "λέξεις-κλειδιά" (δεν θα πρέπει να συγχέονται με τις λέξεις κλειδιά της Python όπως η `while` και η `def`).

Αυτή η σύνταξη κάνει το πρόγραμμα πιο ευανάγνωστο. Είναι επίσης μία υπενθύμιση σχετικά με το πως λειτουργούν τα ορίσματα και οι παράμετροι: όταν καλείτε μία συνάρτηση τα ορίσματα εκχωρούνται στις παραμέτρους.

4.6 Σχεδίαση διεπαφής

Το επόμενο βήμα είναι να γράψετε τη `circle` η οποία παίρνει μία ακτίνα `r` ως παράμετρο. Μία απλή λύση η οποία χρησιμοποιεί την `polygon` για να σχεδιάσει ένα πολύγωνο 50 πλευρών είναι:

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

Η πρώτη γραμμή υπολογίζει την περίμετρο ενός κύκλου με ακτίνα r χρησιμοποιώντας τον τύπο $2\pi r$. Από τη στιγμή που χρησιμοποιούμε τη math.pi θα πρέπει να εισάγουμε τη math. Κατά συνθήκη, οι δηλώσεις import μπαίνουν συνήθως στην αρχή του σεναρίου.

Η n είναι το πλήθος των ευθύγραμμων τμημάτων στον κατά προσέγγιση κύκλο μας και η length είναι το μήκος κάθε τμήματος. Επομένως, η polygon σχεδιάζει ένα πολύγωνο 50 πλευρών το οποίο προσεγγίζει ένα κύκλο με ακτίνα r.

Ένας από τους περιορισμούς αυτής της λύσης είναι ότι η n είναι μία σταθερά, το οποίο σημαίνει ότι για πολύ μεγάλους κύκλους τα ευθύγραμμα τμήματα είναι πολύ μεγάλα και για μικρούς κύκλους σπαταλάμε χρόνο σχεδιάζοντας πολύ μικρά τμήματα. Μία λύση θα ήταν να γενικεύαμε τη συνάρτηση περνώντας την n σαν παράμετρο. Αυτό θα δώσει στο χρήστη περισσότερο έλεγχο (όταν καλεί την circle) αλλά η διεπαφή θα γίνει λίγο πιο πολύπλοκη.

Η **διεπαφή** (interface) μίας συνάρτησης είναι μία σύνοψη του πως αυτή χρησιμοποιείται: ποιες είναι οι παράμετροι; τι κάνει η συνάρτηση; ποια είναι η επιστρεφόμενη τιμή; Μία διεπαφή είναι "ξεκάθαρη" εάν είναι "όσο το δυνατόν πιο απλή αλλά όχι απλούστερη. (Αϊνστάιν)"

Σε αυτό το παράδειγμα, η r ανήκει στη διεπαφή επειδή προσδιορίζει τον κύκλο που θα σχεδιαστεί. Η n δεν είναι τόσο κατάλληλη επειδή αναφέρεται στις λεπτομέρειες του πως ο κύκλος θα πρέπει να υλοποιηθεί.

Αντί να υπερφορτώσουμε την διεπαφή, είναι καλύτερα να διαλέξουμε μία κατάλληλη τιμή για την n ανάλογα με την περιφέρεια :

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 1
    length = circumference / n
    polygon(t, n, length)
```

Τώρα, ο αριθμός των τμημάτων είναι (περίπου) circumference/3, έτσι το μήκος κάθε τμήματος είναι (περίπου) 3, το οποίο είναι αρκετά μικρό ώστε ο κύκλος να φαίνεται καλός αλλά και αρκετά μεγάλο για να είναι αποδοτικό και κατάλληλο για οποιοδήποτε μέγεθος κύκλου.

4.7 Ανακατασκευή κώδικα

Όταν έγραψα τη circle, είχα την δυνατότητα να χρησιμοποιήσω την polygon επειδή ένα πολύπλευρο πολύγωνο είναι μία καλή προσέγγιση ενός κύκλου. Αλλά η arc δεν είναι τόσο συνεργάσιμη. Δεν μπορούμε να χρησιμοποιήσουμε τη polygon ή τη circle για να σχεδιάσουμε ένα τόξο.

Μία εναλλακτική λύση είναι να ξεκινήσουμε με ένα αντίγραφο της polygon και να το μετατρέψουμε σε arc. Το αποτέλεσμα θα μπορούσε να είναι κάπως έτσι:

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
```

```
    for i in range(n):
        fd(t, step_length)
        lt(t, step_angle)
```

Το δεύτερο μισό αυτής της συνάρτησης μοιάζει με τη polygon αλλά δεν μπορούμε να επαναχρησιμοποιήσουμε τη polygon χωρίς να αλλάξουμε τη διεπαφή. Θα μπορούσαμε να γενικεύσουμε τη polygon έτσι ώστε να παίρνει μία γωνία σαν τρίτο όρισμα αλλά τότε το polygon δεν θα είναι πλέον ένα κατάλληλο όνομα! Αντ'αυτού, ας καλέσουμε την πιο γενική συνάρτηση polyline:

```
def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Τώρα μπορούμε να ξαναγράψουμε την polygon και την arc χρησιμοποιώντας την polyline:

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)

def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

Τέλος, μπορούμε να ξαναγράψουμε τη circle χρησιμοποιώντας και πάλι την arc:

```
def circle(t, r):
    arc(t, r, 360)
```

Αυτή η διαδικασία αναδιάταξης ενός προγράμματος για τη βελτίωση των διεπαφών των συναρτήσεων και τη διευκόλυνση της επαναχρησιμοποίησης του κώδικα ονομάζεται **ανακατασκευή κώδικα** (refactoring). Σε αυτήν την περίπτωση, παρατηρήσαμε ότι υπήρχε παρόμοιος κώδικας στη arc και στη polygon και έτσι τον "ανακατασκευάσαμε" στην polyline.

Εάν το είχαμε προβλέψει, ίσως είχαμε γράψει τη polyline εξαρχής και έτσι θα αποφεύγαμε την ανακατασκευή. Πολλές φορές όμως δεν γνωρίζετε αρκετά στην αρχή ενός έργου (project) για να σχεδιάσετε όλες τις διεπαφές. Μερικές φορές η ανακατασκευή είναι ένα σημάδι ότι έχετε μάθει κάτι.

4.8 Πλάνο ανάπτυξης

Ένα **πλάνο ανάπτυξης** (development plan) είναι μία διαδικασία για την συγγραφή προγραμμάτων. Η μέθοδος που χρησιμοποιήσαμε σε αυτήν την περίπτωση είναι η "ενθυλάκωση" και η "γενίκευση". Τα βήματα αυτής της μεθόδου είναι:

1. Ξεκινήστε γράφοντας ένα μικρό πρόγραμμα χωρίς ορισμούς συναρτήσεων.
2. Από τη στιγμή που το πρόγραμμά σας δουλεύει, ενθυλακώστε το σε μία συνάρτηση και δώστε του ένα όνομα.
3. Γενικεύστε τη συνάρτηση προσθέτοντας κατάλληλες παραμέτρους.

4. Επαναλάβετε τα βήματα 1-3 μέχρις ότου να έχετε ένα σύνολο από λειτουργικές συναρτήσεις. Αντιγράψτε τον λειτουργικό κώδικα για να αποφύγετε την επαναπληκτρολόγηση (και την επαναποσφαλμάτωση).

5. Ψάξτε για δυνατότητες βελτίωσης του προγράμματος μέσω της ανακατασκευής κώδικα. Για παράδειγμα, εάν έχετε παρόμοιο κώδικα σε μερικά σημεία, σκεφτείτε να τον ανακατασκευάσετε σε μία καταλληλότερη γενική συνάρτηση.

Αυτή η διαδικασία έχει κάποια μειονεκτήματα (θα δούμε εναλλακτικές λύσεις αργότερα) αλλά μπορεί να είναι χρήσιμη εάν δεν γνωρίζετε εκ των προτέρων πως να σπάσετε το πρόγραμμα σε συναρτήσεις. Αυτή η προσέγγιση σας επιτρέπει να σχεδιάζετε όσο προχωράτε.

4.9 Συμβολοσειρά τεκμηρίωσης

Μία **συμβολοσειρά τεκμηρίωσης** (docstring) είναι μία συμβολοσειρά στην αρχή μιας συνάρτησης η οποία εξηγεί τη διεπαφή (το "doc" είναι συντομογραφία του "documentation"). Για παράδειγμα:

```
def polyline(t, n, length, angle):
    """Draws n line segments with the given length and
    angle (in degrees) between them.  t is a turtle.
    """
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Αυτή η συμβολοσειρά τεκμηρίωσης είναι μια συμβολοσειρά με τριπλά εισαγωγικά, γνωστή και ως πολλυγραμμική (multiline) συμβολοσειρά επειδή τα τριπλά εισαγωγικά επιτρέπουν στη συμβολοσειρά να εκτείνεται σε περισσότερες από μία γραμμές.

Είναι λιτή αλλά περιέχει τις απαραίτητες πληροφορίες που χρειάζεται κάποιος για να χρησιμοποιήσει αυτή τη συνάρτηση. Εξηγεί συνοπτικά τι κάνει η συνάρτηση (χωρίς να υπεισέρχεται σε λεπτομέρειες για το πως το κάνει). Εξηγεί τι επίδραση έχει κάθε παράμετρος στη συμπεριφορά της συνάρτησης και τι τύπο θα πρέπει να έχει η κάθε παράμετρος (αν δεν είναι προφανές).

Η σύνταξη αυτού του είδους τεκμηρίωσης είναι ένα σημαντικό κομμάτι του σχεδιασμού διεπαφής. Μία καλοσχεδιασμένη διεπαφή θα πρέπει να είναι απλό να εξηγηθεί. Αν δυσκολεύεστε να εξηγήσετε κάποια από τις συναρτήσεις σας τότε αυτό είναι ένα σημάδι ότι η διεπαφή θα μπορούσε να βελτιωθεί.

4.10 Αποσφαλμάτωση

Μία διεπαφή είναι σαν μία σύμβαση μεταξύ μίας συνάρτησης και του καλούντος. Ο καλών συμφωνεί να παρέχει συγκεκριμένες παραμέτρους και η συνάρτηση δέχεται να κάνει μία συγκεκριμένη δουλειά.

Για παράδειγμα, η `polyline` απαιτεί τέσσερα ορίσματα: η `t` θα πρέπει να είναι μία χελώνα, η `n` είναι ο αριθμός των ευθύγραμμων τμημάτων (θα πρέπει να είναι ακέραιος), η `length` θα πρέπει να είναι ένας θετικός αριθμός και η `angle` θα πρέπει να είναι αριθμός (εννοείται σε μοίρες).

Οι απαιτήσεις αυτές ονομάζονται **προϋποθέσεις** (preconditions) επειδή πρέπει να είναι αληθείς προτού αρχίσει να εκτελείται η συνάρτηση. Αντιστρόφως, οι συνθήκες στο τέλος της συνάρτησης ονομάζονται **μετά-συνθήκες** (post-conditions). Οι μετά-συνθήκες περιλαμβάνουν το προβλεπόμενο

αποτέλεσμα της συνάρτησης (όπως τη σχεδίαση ευθύγραμμων τμημάτων) και οποιεσδήποτε άλλες παράπλευρες ενέργειες (όπως η κίνηση της χελώνας ή άλλες αλλαγές στον κόσμο).

Οι προϋποθέσεις είναι ευθύνη του καλούντος. Αν ο καλών παραβιάσει μία (σωστά τεκμηριωμένη) προϋπόθεση και η συνάρτηση δε λειτουργεί σωστά, τότε το λάθος είναι του καλούντος και όχι της συνάρτησης.

4.11 Ορολογία

στιγμιότυπο: Ένα μέρος ενός συνόλου. Η TurtleWorld σε αυτό το κεφάλαιο είναι ένα μέρος του συνόλου TurtleWorlds.

βρόχος: Ένα κομμάτι του προγράμματος που μπορεί να εκτελείται επανειλημμένα.

ενθυλάκωση: Η διαδικασία μετασχηματισμού μίας ακολουθίας δηλώσεων σε έναν ορισμό συνάρτησης.

γενίκευση: Η διαδικασία αντικατάστασης κάτι συγκεκριμένου (όπως ένας αριθμός) με κάτι πιο γενικευμένου (όπως μία μεταβλητή ή παράμετρος).

λέξη-κλειδί όρισμα: Ένα όρισμα το οποίο περιέχει το όνομα της παραμέτρου ως "λέξη-κλειδί".

διεπαφή: Μία περιγραφή του πώς χρησιμοποιείται μία συνάρτηση, συμπεριλαμβανομένου του ονόματος και της περιγραφής των ορισμάτων και της επιστρεφόμενης τιμής.

ανακατασκευή κώδικα: Η διαδικασία τροποποίησης ενός λειτουργικού προγράμματος για την βελτίωση των διεπαφών των συναρτήσεων και άλλων χαρακτηριστικών του κώδικα.

πλάνο ανάπτυξης: Μία μέθοδος γραφής προγραμμάτων.

συμβολοσειρά τεκμηρίωσης: Μία συμβολοσειρά η οποία εμφανίζεται σε ένα ορισμό συνάρτησης για να τεκμηριώσει τη διεπαφή της συνάρτησης.

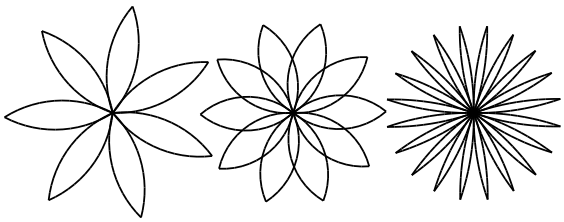
προϋπόθεση: Μία συνθήκη που πρέπει να ικανοποιείται από τον καλούντα προτού ξεκινήσει η συνάρτηση.

μετά-συνθήκη: Μία συνθήκη η οποία πρέπει να ικανοποιηθεί από τη συνάρτηση προτού τελειώσει.

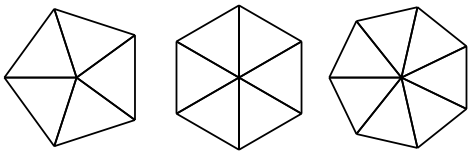
4.12 Ασκήσεις

Άσκηση 4.1. *Κατεβάστε τον κώδικα αυτού του κεφαλαίου από το σύνδεσμο* `http://thinkpython.com/code/polygon.py`.

1. *Γράψτε κατάλληλες συμβολοσειρές κειμένου για τις* `polygon`, `arc` *και* `circle`.
2. *Σχεδιάστε ένα διάγραμμα στοίβας το οποίο θα δείχνει την κατάσταση του προγράμματος ενώ εκτελείται η* `circle(bob, radius)`. *Μπορείτε να κάνετε τις πράξεις με το χέρι ή να προσθέσετε δηλώσεις* `print` *στον κώδικα.*
3. *Η έκδοση της* `arc` *στην ενότητα 4.7 δεν είναι πολύ ακριβής επειδή η γραμμική προσέγγιση του κύκλου δεν είναι ποτέ ένας αληθινός κύκλος. Σαν αποτέλεσμα, η χελώνα τελειώνει λίγο πιο μακριά από τον σωστό προορισμό. Η δική μου λύση δείχνει έναν τρόπο να περιορίσουμε το φαινόμενο αυτού του λάθους. Διαβάστε τον κώδικα και δείτε αν μπορείτε να βγάλετε κάποιο νόημα. Εάν σχεδιάσετε ένα διάγραμμα, ίσως καταλάβετε πως δουλεύει.*



Σχήμα 4.1: Turtle flowers.



Σχήμα 4.2: Turtle pies.

Άσκηση 4.2. *Γράψτε ένα σύνολο κατάλληλων γενικών συναρτήσεων οι οποίες θα μπορούν να σχεδιάσουν λουλούδια όπως αυτά στο Σχήμα 4.1.*

Λύση: `http://thinkpython.com/code/flower.py`. *Θα χρειαστείτε και την* `http://thinkpython.com/code/polygon.py`.

Άσκηση 4.3. *Γράψτε ένα σύνολο κατάλληλων γενικών συναρτήσεων οι οποίες θα μπορούν να σχεδιάσουν σχήματα όπως στην Εικόνα 4.2.*

Λύση : `http://thinkpython.com/code/pie.py`.

Άσκηση 4.4. *Τα γράμματα του αγγλικού αλφαβήτου μπορούν να σχεδιαστούν από ένα σύνολο βασικών στοιχείων, όπως κάθετες και οριζόντιες γραμμές και μερικές καμπύλες. Σχεδιάστε μία γραμματοσειρά η οποία θα μπορεί να υλοποιηθεί με ένα μικρό αριθμό βασικών στοιχείων και στη συνέχεια γράψτε συναρτήσεις η οποίες θα σχεδιάζουν γράμματα του αγγλικού αλφαβήτου.*

Πρέπει να γράψετε μία συνάρτηση για κάθε γράμμα, με ονόματα `draw_a`, `draw_b`, *κτλ., και να βάλτε της συναρτήσεις σε ένα αρχείο με όνομα* `letters.py`. *Μπορείτε να κατεβάσετε μία "χελώνα δακτυλογράφο" από εδώ :* `http://thinkpython.com/code/typewriter.py` *για να σας βοηθήσει να ελέγξετε τον κώδικά σας.*

Λύση : `http://thinkpython.com/code/letters.py`. *Θα χρειαστείτε και την :* `http://thinkpython.com/code/polygon.py`.

Άσκηση 4.5. *Διαβάστε σχετικά με τις σπείρες στο* `http://en.wikipedia.org/wiki/Spiral`, *και στη συνέχεια γράψτε ένα πρόγραμμα που θα σχεδιάζει μία Αρχιμήδεια σπείρα (ή κάποιο από τα υπόλοιπα είδη). Λύση:* `http://thinkpython.com/codespiral.py`.

Κεφάλαιο 5

Δηλώσεις υπό συνθήκη και αναδρομή

5.1 Τελεστής υπολογισμού υπολοίπου ακέραιας διαίρεσης

Ο **τελεστής υπόλοιπο** (modulus operator) εφαρμόζεται σε ακέραιους αριθμούς και επιστρέφει το υπόλοιπο όταν ο πρώτος τελεστέος διαιρείται από το δεύτερο. Στην Python, ο τελεστής "υπόλοιπο" είναι το σύμβολο επί τοις εκατό (%). Η σύνταξη είναι η ίδια όπως και με τους άλλους τελεστές:

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

Έτσι το 7 διαιρούμενο με το 3 μας δίνει υπόλοιπο 1.

Ο τελεστής "υπόλοιπο" μπορεί να αποδειχτεί ιδιαίτερα χρήσιμος. Για παράδειγμα, μπορείτε να ελέγξετε αν ένας αριθμός `x` διαιρείται από έναν άλλο αριθμό `y` αν η έκφραση `x % y` δίνει αποτέλεσμα μηδέν.

Επίσης, μπορείτε να εξάγετε το δεξιότερο ψηφίο ή ψηφία από έναν αριθμό. Για παράδειγμα, η έκφραση `x % 10` δίνει το τελευταίο ψηφίο του `x` (μονάδες στη βάση του 10) και η `x % 100` δίνει τα δύο τελευταία ψηφία.

5.2 Λογικές εκφράσεις

Μία **λογική έκφραση** (boolean expression) είναι μία έκφραση η οποία είναι είτε αληθής είτε ψευδής. Τα ακόλουθα παραδείγματα χρησιμοποιούν τον τελεστή `==`, ο οποίος συγκρίνει δύο τελεστέους και παράγει `True` αν είναι ίσοι ή `False` αλλιώς:

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

Η `True` και η `False` είναι ειδικές τιμές οι οποίες ανήκουν στον τύπο `bool`, δεν είναι συμβολοσειρές:

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

Ο τελεστής == είναι ένας από τους **σχεσιακούς τελεστές** (relational operators):

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
```

Πιθανώς να γνωρίζετε αυτές τις πράξεις αλλά τα σύμβολα της Python είναι διαφορετικά από τα σύμβολα των μαθηματικών. Ένα συνηθισμένο λάθος είναι η χρήση ενός μονό συμβόλου ισότητας (=) αντί για διπλό σύμβολο ισότητας (==). Να θυμάστε ότι το = είναι ένας τελεστής εκχώρησης και το == είναι ένας σχεσιακός τελεστής. Δεν υπάρχει κάτι αντίστοιχο των =< και =>.

5.3 Λογικοί τελεστές

Υπάρχουν τρεις **λογικοί τελεστές** (logical operators): ο `and`, ο `or` και ο `not`. Η σημασιολογική ερμηνεία αυτών των τελεστών είναι παρόμοια με το νόημά τους στα Αγγλικά. Για παράδειγμα, η `x > 0 and x < 10` είναι αληθής μόνο αν το `x` είναι μεγαλύτερο του μηδενός και μικρότερο του 10.

Η `n%2 == 0 or n%3 == 0` είναι αληθείς αν οποιαδήποτε από τις δύο συνθήκες είναι αληθής. Αν δηλαδή ο αριθμός διαιρείται ή με το 2 ή με το 3.

Τέλος ο τελεστής `not` ακυρώνει μια λογική έκφραση. Άρα η `not (x > y)` είναι αληθής αν η `x > y` είναι ψευδής και αυτό συμβαίνει αν το `x` είναι μικρότερο ή ίσο του `y`.

Αν θέλαμε να είμαστε αυστηροί, οι τελεστέοι των λογικών τελεστών θα έπρεπε να είναι λογικές εκφράσεις αλλά η Python δεν είναι τόσο αυστηρή. Κάθε μη μηδενικός αριθμός διερμηνεύεται ως ”αληθής” (”true”).

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

Αυτή η ευελιξία μπορεί να είναι χρήσιμη αλλά υπάρχουν κάποιες μικρές λεπτομέρειες οι οποίες μπορεί να προκαλέσουν σύγχυση και καλύτερα να το αποφύγετε (εκτός αν ξέρετε τι κάνετε).

5.4 Εκτέλεση υπό συνθήκη

Προκειμένου να γράψουμε χρήσιμα προγράμματα πρέπει να μπορούμε να ελέγχουμε συνθήκες και να αλλάζουμε αναλόγως τη συμπεριφορά του προγράμματος. Αυτή τη δυνατότητα μας την δίνουν οι **δηλώσεις υπό συνθήκη** (conditional statements) και η απλούστερη από αυτές είναι η δήλωση `if`:

```
if x > 0:
    print 'x is positive'
```

Η λογική έκφραση μετά την `if` ονομάζεται **συνθήκη** (condition). Αν είναι αληθής τότε η εμφωλευμένη δήλωση αρχίζει να εκτελείται ενώ αν είναι ψευδής δεν γίνεται τίποτα.

Οι δηλώσεις `if` έχουν την ίδια δομή με τους ορισμούς συνάρτησης: μία επικεφαλίδα ακολουθούμενη από ένα εμφωλευμένο σώμα. Δηλώσεις όπως αυτές ονομάζονται **σύνθετες δηλώσεις** (compound statements).

Δεν υπάρχει όριο στο πλήθος των δηλώσεων που μπορούμε να βάλουμε μέσα στο σώμα αλλά πρέπει να υπάρχει τουλάχιστον μία. Ενίοτε όμως είναι χρήσιμο να έχουμε ένα σώμα χωρίς δηλώσεις (σαν κρατημένο χώρο για τον κώδικα που δεν έχουμε γράψει ακόμη). Σε αυτή τη περίπτωση μπορούμε να χρησιμοποιήσουμε τη δήλωση `pass` η οποία δεν κάνει τίποτα.

```
if x < 0:
    pass          # need to handle negative values!
```

5.5 Εναλλακτική εκτέλεση

Μία δεύτερη μορφή της δήλωσης `if` είναι η **εναλλακτική εκτέλεση** (alternative execution), στην οποία υπάρχουν δύο ενδεχόμενα και το ποιο από τα δύο θα εκτελεστεί καθορίζεται από την συνθήκη. Η σύνταξη είναι κάπως έτσι:

```
if x%2 == 0:
    print 'x is even'
else:
    print 'x is odd'
```

Αν το υπόλοιπο της διαίρεσης του `x` με το 2 είναι 0 τότε αντιλαμβανόμαστε ότι το `x` είναι άρτιος και το πρόγραμμα εμφανίζει ένα μήνυμα γι αυτό το αποτέλεσμα. Αν η συνθήκη είναι ψευδής τότε εκτελείται το δεύτερο σύνολο δηλώσεων. Από τη στιγμή που η συνθήκη είναι είτε αληθής είτε ψευδής τότε μπορεί να εκτελεστεί μόνο μία από τις εναλλακτικές. Οι εναλλακτικές ονομάζονται **κλάδοι** (branches) επειδή στην ουσία ”διακλαδώνουν” τη ροή εκτέλεσης.

5.6 Αλυσιδωτές συνθήκες

Μερικές φορές χρειαζόμαστε περισσότερους από δύο κλάδους γιατί υπάρχουν περισσότερες από δύο πιθανότητες. Ένας τρόπος για να εκφράσουμε έναν τέτοιο υπολογισμό είναι μία **αλυσιδωτή συνθήκη** (chained conditional):

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

Η `elif` είναι συντομογραφία της “else if”. Δεν υπάρχει όριο στον αριθμό των δηλώσεων `elif` αλλά όπως και πριν θα εκτελεστεί μόνο ένας από τους κλάδους. Αν υπάρχει μία δήλωση `else` τότε θα πρέπει να βρίσκεται στο τέλος αλλά δεν είναι υποχρεωτικό να υπάρχει.

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
```

```
elif choice == 'c':
    draw_c()
```

Η κάθε συνθήκη ελέγχεται με τη σειρά. Αν η πρώτη είναι λάθος τότε ελέγχεται η δεύτερη και ούτω καθεξής. Αν μία από αυτές είναι αληθής τότε εκτελείται ο αντίστοιχος κλάδος και η δήλωση τερματίζει. Ακόμη κι αν περισσότερες από μία συνθήκες είναι αληθείς μόνο ο πρώτος αληθής κλάδος εκτελείται.

5.7 Εμφωλευμένες συνθήκες

Ένας όρος μπορεί να είναι εμφωλευμένος μέσα σε έναν άλλο. Επομένως, θα μπορούσαμε να είχαμε γράψει το παράδειγμα της τριχοτόμησης κάπως έτσι:

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

Η εξωτερική συνθήκη περιέχει δύο κλάδους. Ο πρώτος κλάδος περιέχει μία απλή δήλωση και ο δεύτερος περιέχει μία άλλη δήλωση `if` η οποία έχει δύο κλάδους από μόνη της. Αυτοί οι δύο κλάδοι είναι απλές δηλώσεις παρόλο που θα μπορούσαν επίσης να είναι δηλώσεις υπό συνθήκη.

Παρόλο που η ενδοπαραγραφοποίηση των δηλώσεων κάνει προφανή τη δομή, οι εμφωλευμένες συνθήκες δεν είναι εύκολο να διαβαστούν γρήγορα. Γενικά είναι προτιμότερο να τις αποφεύγετε αν μπορείτε.

Οι λογικοί τελεστές παρέχουν συχνά ένα τρόπο για την απλοποίηση των εμφωλευμένων δηλώσεων υπό συνθήκη. Για παράδειγμα, μπορούμε να γράψουμε ξανά τον ακόλουθο κώδικα χρησιμοποιώντας μία απλή υπόθεση:

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

Η δήλωση `print` εκτελείται μόνο αν είναι αληθείς και οι δυο συνθήκες, έτσι μπορούμε να έχουμε το ίδιο αποτέλεσμα με τον τελεστή `and`:

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

5.8 Αναδρομή

Μία συνάρτηση μπορεί να καλέσει μία άλλη συνάρτηση αλλά μπορεί επίσης να καλέσει και τον ίδιο της τον εαυτό. Η χρησιμότητα αυτού του χαρακτηριστικού μπορεί να μην είναι προφανής, αλλά αποδεικνύεται ένα από τα πιο μαγικά πράγματα που μπορεί να κάνει ένα πρόγραμμα. Για παράδειγμα δείτε την ακόλουθη συνάρτηση:

```
def countdown(n):
    if n <= 0:
        print 'Blastoff!'
```

```
    else:
        print n
        countdown(n-1)
```

Αν η παράμετρος `n` είναι 0 ή αρνητική τότε στην έξοδο θα έχουμε τη λέξη "Blastoff!". Αλλιώς θα τυπώσει την τιμή της `n` και στη συνέχεια θα καλέσει τον εαυτό της με όρισμα `n-1`.

Τι συμβαίνει αν καλέσουμε αυτή τη συνάρτηση έτσι:

```
>>> countdown(3)
```

Η εκτέλεση της `countdown` ξεκινάει με `n=3` και από τη στιγμή που η `n` είναι μεγαλύτερη του 0 τυπώνει την τιμή 3 και έπειτα καλεί τον εαυτό της...

> Η εκτέλεση της `countdown` ξεκινάει με `n=2` και από τη στιγμή που η `n` είναι μεγαλύτερη του 0 τυπώνει την τιμή 2 και έπειτα καλεί τον εαυτό της...
>
> > Η εκτέλεση της `countdown` ξεκινάει με `n=1` και από τη στιγμή που η `n` είναι μεγαλύτερη του 0 τυπώνει την τιμή 1 και έπειτα καλεί τον εαυτό της...
> >
> > > Η εκτέλεση της `countdown` ξεκινάει με `n=0` και από τη στιγμή που η `n` δεν είναι μεγαλύτερη του 0 τυπώνει τη λέξη "Blastoff!" και έπειτα επιστρέφει.
> >
> > Η `countdown` με `n=1` επιστρέφει.
>
> Η `countdown` με `n=2` επιστρέφει.

Η `countdown` με `n=3` επιστρέφει.

Και στη συνέχεια βρίσκεστε πίσω στη `__main__`. Έτσι η συνολική έξοδος είναι κάπως έτσι:

```
3
2
1
Blastoff!
```

Μία συνάρτηση η οποία καλεί τον εαυτό της ονομάζεται **αναδρομική** (recursive) και η διαδικασία ονομάζεται **αναδρομή** (recursion).

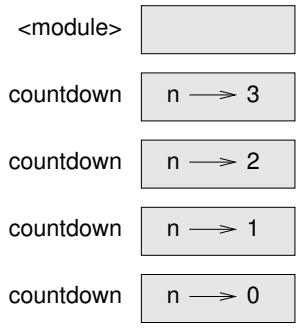
Ένα ακόμα παράδειγμα είναι μία συνάρτηση η οποία εμφανίζει μία συμβολοσειρά `n` φορές:

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

Αν `n <= 0` η δήλωση `return` τερματίζει τη συνάρτηση και η ροή εκτέλεσης επιστρέφει αμέσως στην συνάρτηση που την κάλεσε ενώ οι υπόλοιπες γραμμές της συνάρτησης δεν εκτελούνται.

Η υπόλοιπη συνάρτηση είναι παρόμοια με τη `countdown`: εάν η `n` είναι μεγαλύτερη του 0 τότε εμφανίζει την `s` και καλεί τον εαυτό της για να εμφανίσει την `s` $n - 1$ επιπλέον φορές. Έτσι ο αριθμός γραμμών της εξόδου είναι `1 + (n - 1)`, δηλαδή ίσο με `n`.

Για απλά παραδείγματα όπως αυτά, ίσως είναι ευκολότερο να χρησιμοποιήσετε ένα βρόχο `for`. Αλλά αργότερα θα δούμε παραδείγματα τα οποία είναι δύσκολο να γραφτούν με `for` και ευκολότερο να γραφτούν με αναδρομή. Γι αυτό το λόγο είναι καλό να ξεκινήσετε νωρίς.



Σχήμα 5.1: Διάγραμμα στοίβας.

5.9 Διαγράμματα στοίβας για αναδρομικές συναρτήσεις

Στην παράγραφο 3.10 χρησιμοποιήσαμε ένα διάγραμμα στοίβας για να αναπαραστήσουμε την κατάσταση του προγράμματος κατά τη διάρκεια μιας κλήσης συνάρτησης. Το ίδιο είδος διαγράμματος μπορεί να βοηθήσει για να ερμηνεύσουμε μια αναδρομική συνάρτηση.

Κάθε φορά που καλείται μία συνάρτηση, η Python δημιουργεί ένα καινούργιο πλαίσιο συνάρτησης το οποίο περιέχει τις τοπικές μεταβλητές και τις παραμέτρους της συνάρτησης. Για μία αναδρομική συνάρτηση μπορεί να υπάρχουν περισσότερα από ένα πλαίσια στη στοίβα ταυτόχρονα.

Το σχήμα 5.1 δείχνει ένα διάγραμμα στοίβας της `countdown` η οποία έχει καλεστεί με `n = 3`.

Ως συνήθως, στη κορυφή της στοίβας είναι το πλαίσιο για τη `__main__`. Είναι άδειο γιατί δε δημιουργήσαμε ούτε μεταβλητές στη `__main__`, ούτε της περάσαμε κάποιο όρισμα.

Τα τέσσερα πλαίσια `countdown` έχουν διαφορετικές τιμές για την παράμετρο `n`. Το κάτω μέρος της στοίβας όπου `n=0` ονομάζεται **περίπτωση βάσης** (base case). Δεν υπάρχουν άλλα πλαίσια γιατί δεν γίνεται άλλη αναδρομική κλήση.

Άσκηση 5.1. *Σχεδιάστε ένα διάγραμμα στοίβας για τη* `print_n` *η οποία θα καλείται με* `s = 'Hello'` *και* `n=2`.
Άσκηση 5.2. *Γράψτε μία συνάρτηση με όνομα* `do_n` *η οποία θα παίρνει ένα αντικείμενο συνάρτησης και έναν αριθμό* `n` *σαν ορίσματα και θα καλεί τη δοθείσα συνάρτηση* `n` *φορές.*

5.10 Άπειρη αναδρομή

Αν μία αναδρομή δε φτάνει ποτέ σε μία περίπτωση βάσης, τότε συνεχίζει κάνοντας αναδρομικές κλήσεις για πάντα και το πρόγραμμα δεν τερματίζει ποτέ. Αυτή είναι γνωστή ως **άπειρη αναδρομή** (infinite recursion) και γενικά δεν είναι καλή ιδέα. Αυτό είναι ένα πολύ μικρό πρόγραμμα με άπειρη αναδρομή:

```
def recurse():
    recurse()
```

Στα περισσότερα προγραμματιστικά περιβάλλοντα, ένα πρόγραμμα με άπειρη αναδρομή δεν τρέχει για πάντα. Η Python αναφέρει ένα μήνυμα λάθους όταν φτάσει στο μέγιστο βάθος αναδρομής:

```
  File "<stdin>", line 2, in recurse
```

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

Αυτή η αναδρομική ανίχνευση είναι λίγο μεγαλύτερη από αυτή που είδαμε στο προηγούμενο κεφάλαιο. Όταν συμβεί αυτό το λάθος θα υπάρχουν 1000 αναδρομικά πλαίσια στη στοίβα!

5.11 Είσοδος από το πληκτρολόγιο

Τα προγράμματα που έχουμε γράψει μέχρι στιγμής είναι λίγο αγενή με την έννοια ότι δε δέχονται καμία είσοδο από το χρήστη, κάνουν απλά το ίδιο πράγμα κάθε φορά.

Η Python 2 παρέχει μία ενσωματωμένη συνάρτηση με όνομα `raw_input` η οποία δέχεται είσοδο από το πληκτρολόγιο, ενώ στην Python 3 αυτή η συνάρτηση ονομάζεται `input`. Όταν καλείται αυτή η συνάρτηση, το πρόγραμμα σταματάει και περιμένει το χρήστη να πληκτρολογήσει κάτι. Όταν ο χρήστης πατήσει Return ή Enter, το πρόγραμμα συνεχίζει και η `raw_input` επιστρέφει ότι πληκτρολόγησε ο χρήστης σαν μία συμβολοσειρά.

```
>>> text = raw_input()
What are you waiting for?
>>> print text
What are you waiting for?
```

Πριν το πρόγραμμα πάρει την είσοδο από το χρήστη, καλό θα ήταν να εμφανίσει ένα μήνυμα προτροπής το οποίο θα λέει στο χρήστη τι να εισάγει. Η `raw_input` μπορεί να πάρει σαν όρισμα ένα τέτοιο μήνυμα:

```
>>> name = raw_input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> print name
Arthur, King of the Britons!
```

Το `\n` στο τέλος του μηνύματος αναπαριστά μία **newline**, η οποία είναι ένας ειδικός χαρακτήρας που προκαλεί αλλαγή γραμμής. Για αυτό η είσοδος του χρήστη εμφανίζεται κάτω από το μήνυμα προτροπής.

Εάν περιμένετε ο χρήστης να πληκτρολογήσει έναν ακέραιο, τότε μπορείτε να δοκιμάσετε να μετατρέψετε την επιστρεφόμενη τιμή σε `int`:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
```

Αλλά αν ο χρήστης πληκτρολογήσει κάτι διαφορετικό από μία σειρά ψηφίων τότε θα πάρετε ένα μήνυμα λάθους:

```
>>> speed = raw_input(prompt)
```

```
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int()
```

Θα δούμε πώς χειριζόμαστε τέτοιου είδους λάθη αργότερα.

5.12 Αποσφαλμάτωση

Η Python εμφανίζει αναδρομικά πολλές πληροφορίες όταν συμβεί κάποιο λάθος, κάτι το οποίο μπορεί να είναι υπερβολικό και ειδικά όταν υπάρχουν πολλά πλαίσια στη στοίβα. Συνήθως, τα πιο χρήσιμα σημεία είναι:

- Τι είδος λάθους ήταν, και
- Πότε συνέβη.

Τα συντακτικά λάθη είναι συνήθως εύκολο να βρεθούν αλλά υπάρχουν και μερικές παγίδες. Τα λάθη λευκού κενού διαστήματος (whitespace errors) μπορεί να είναι δυσεπίλυτα γιατί τα κενά διαστήματα (spaces) και οι στηλοθέτες (tabs) είναι αόρατα και συνηθίζουμε να τα αγνοούμε.

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

Σε αυτό το παράδειγμα το πρόβλημα είναι ότι η δεύτερη γραμμή έχει εσοχή ενός κενού διαστήματος αλλά το μήνυμα λάθους δείχνει το `y`, το οποίο είναι παραπλανητικό. Σε γενικές γραμμές τα μηνύματα λάθους υποδεικνύουν που ανέκυψε το πρόβλημα αλλά το πραγματικό λάθος μπορεί να είναι νωρίτερα στον κώδικα, ακόμη και σε προηγούμενη γραμμή.

Το ίδιο ισχύει και για τα λάθη χρόνου εκτέλεσης.

Υποθέστε ότι προσπαθείτε να υπολογίσετε ένα σηματοθορυβικό λόγο σε ντεσιμπέλ (dB). Ο τύπος είναι $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$ και ίσως το γράφατε κάπως έτσι:

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

Αλλά όταν το τρέχετε στην Python 2 παίρνετε ένα μήνυμα λάθους:

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

Το μήνυμα λάθους υποδεικνύει την γραμμή 5 αλλά δεν υπάρχει κάτι λάθος σε αυτήν τη γραμμή. Για να βρούμε το πραγματικό λάθος ίσως θα ήταν χρήσιμο να εμφανίζαμε την τιμή της `ratio`, η οποία αποδεικνύεται ότι είναι 0. Το πρόβλημα είναι στη γραμμή 4 επειδή όταν διαιρούμε δύο ακέραιους

αριθμούς έχουμε ακέραια διαίρεση. Η λύση είναι να αναπαραστήσουμε την ισχύ του σήματος και την ισχύ του θορύβου με αριθμούς κινητής υποδιαστολής.

Γενικά, τα μηνύματα λάθους σας λένε που ανέκυψε το πρόβλημα αλλά αυτό δεν σημαίνει ότι προκλήθηκε εκεί.

Στην Python 3 αυτό το παράδειγμα δεν προκαλεί σφάλμα γιατί ο τελεστής της διαίρεσης εκτελεί δεκαδική διαίρεση ακόμα και με ακέραιους τελεστέους.

5.13 Ορολογία

τελεστής υπολογισμού υπολοίπου ακέραιας διαίρεσης: Ένας τελεστής ο οποίος συμβολίζεται με το σύμβολο επί της εκατό (%). Δουλεύει με ακεραίους και επιστρέφει το υπόλοιπο όταν διαιρέσουμε δύο αριθμούς.

λογική έκφραση: Μία έκφραση της οποίας η τιμή είναι είτε αληθής (`True`) είτε ψευδής (`False`).

σχεσιακός τελεστής: Ένας τελεστής ο οποίος συγκρίνει δύο τελεστέους: `==`, `!=`, `>`, `<`, `>=` και `<=`.

λογικός τελεστής: Ένας τελεστής ο οποίος συνδιάζει δύο λογικές εκφράσεις: `and`, `or` και `not`.

δηλώσεις υπό συνθήκη: Μία δήλωση η οποία ρυθμίζει τη ροή εκτέλεσης με βάση κάποια συνθήκη.

συνθήκη: Η λογική έκφραση σε μία υπό συνθήκη δήλωση η οποία καθορίζει ποιος κλάδος θα εκτελεστεί.

σύνθετη δήλωση: Μία δήλωση η οποία αποτελείται από μία επικεφαλίδα και ένα σώμα. Η επικεφαλίδα τελειώνει με μία άνω και κάτω τελεία (:) και το σώμα είναι γραμμένο σε εσοχή σε σχέση με την επικεφαλίδα.

κλάδος: Μία από τις εναλλακτικές ακολουθίες εντολών σε μία υπό συνθήκη δήλωση.

αλυσιδωτές συνθήκες: Μία υπό συνθήκη δήλωση με μια σειρά εναλλακτικών κλάδων.

εμφωλευμένη συνθήκη: Μία υπό συνθήκη δήλωση η οποία εμφανίζεται σε έναν από τους κλάδους μίας άλλης υπό συνθήκης δήλωσης.

αναδρομή: Η διαδικασία κατά την οποία καλείται η συνάρτηση η οποία τρέχει αυτή τη στιγμή.

περίπτωση βάσης: Ένας υπό συνθήκη κλάδος σε μία αναδρομική συνάρτηση ο οποίος δεν κάνει καμία αναδρομική κλήση.

άπειρη αναδρομή: Μία αναδρομή η οποία δεν έχει περίπτωση βάσης ή δεν την φτάνει ποτέ (δηλαδή δεν την επαληθεύει). Και τελικά, μία άπειρη αναδρομή προκαλεί ένα λάθος χρόνου εκτέλεσης.

5.14 Ασκήσεις

Άσκηση 5.3. *Το τελευταίο θεώρημα του Fermat λέει ότι δεν υπάρχουν θετικοί ακέραιοι a, b, και c τέτοιοι ώστε*

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

για οποιαδήποτε τιμή του n μεγαλύτερη του 2.

1. *Γράψτε μία συνάρτηση με όνομα* `check_fermat` *η οποία θα παίρνει 4 παραμέτρους,* `a`*,* `b`*,* `c` *και* `n`*, και θα ελέγχει αν ισχύει το θεώρημα του Fermat. Αν το* n *είναι μεγαλύτερο του 2 και αποδεικνύεται ότι η σχέση*

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

είναι αληθής, το πρόγραμμα θα πρέπει να εμφανίζει το μήνυμα: "Holy smokes, Fermat was wrong!" Αλλιώς θα εμφανίζει: "No, that doesn't work."

2. *Γράψτε μία συνάρτηση η οποία ζητάει από το χρήστη να εισάγει τιμές για τις* `a`*,* `b`*,* `c` *και* `n`*, τις μετατρέπει σε ακέραιους και χρησιμοποιεί την* `check_fermat` *για να ελέγξει αν παραβιάζουν το θεώρημα του Fermat.*

Άσκηση 5.4. *Αν σας δοθούν τρία ξυλάκια τότε ίσως μπορέσετε να σχηματίσετε ένα τρίγωνο με αυτά, μπορεί όμως και όχι. Για παράδειγμα, εάν ένα από τα ξυλάκια είναι 12 εκατοστά και τα άλλα δύο είναι 1 εκατοστό το καθένα τότε είναι προφανές ότι δεν θα είστε σε θέση να ενώσετε τα κοντά ξυλάκια μεταξύ τους. Μπορούμε να κάνουμε έναν απλό έλεγχο για οποιοδήποτε συνδυασμό από τρία μήκη για να δούμε αν είναι εφικτό να σχεδιαστεί ένα τρίγωνο:*

> *Εάν ένα από τα τρία μήκη είναι μεγαλύτερο από το άθροισμα των άλλων δύο τότε δεν μπορεί να σχηματιστεί τρίγωνο, αλλιώς μπορεί. (Αν το άθροισμα δύο μηκών είναι ίσο με το τρίτο σχηματίζουν ένα "εκφυλισμένο" τρίγωνο.)*

1. *Γράψτε μία συνάρτηση με όνομα* `is_triangle` *η οποία θα παίρνει τρεις ακέραιους σαν ορίσματα και θα εμφανίζει στην έξοδο είτε "Yes" είτε "No" ανάλογα με το αν μπορείτε ή όχι να σχηματίσετε ένα τρίγωνο με τα ξυλάκια των δοθέντων μηκών.*

2. *Γράψτε μία συνάρτηση η οποία θα ζητάει από το χρήστη να εισάγει τρία μήκη για ξυλάκια, θα τα μετατρέπει σε ακέραιους και θα χρησιμοποιεί την* `is_triangle` *για να ελέγξει εάν τα ξυλάκια με τα δοθέντα μήκη μπορούν να σχεδιάσουν ένα τρίγωνο.*

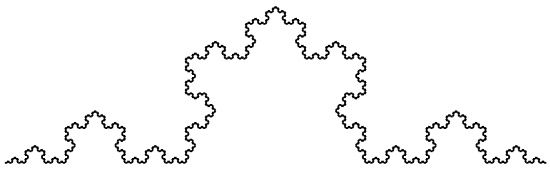
Οι ακόλουθες ασκήσεις χρησιμοποιούν την TurtleWorld από το Κεφάλαιο 4:

Άσκηση 5.5. *Διαβάστε την παρακάτω συνάρτηση και δείτε αν μπορείτε να καταλάβετε τι κάνει. Στη συνέχεια τρέξτε την και δείτε τα παραδείγματα στο Κεφάλαιο 4.*

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    fd(t, length*n)
    lt(t, angle)
    draw(t, length, n-1)
    rt(t, 2*angle)
    draw(t, length, n-1)
    lt(t, angle)
    bk(t, length*n)
```

Άσκηση 5.6. *Η καμπύλη του Koch είναι ένα φράκταλ σαν αυτό στο Σχήμα 5.2. Για να σχεδιάσετε μία καμπύλη του Koch με μήκος , το μόνο που έχετε να κάνετε είναι:*

1. *Σχεδιάστε μία καμπύλη του Koch με μήκος* $x/3$*.*
2. *Στρίψτε αριστερά 60 μοίρες.*
3. *Σχεδιάστε μία καμπύλη του Koch με μήκος* $x/3$*.*
4. *Στρίψτε δεξιά 120 μοίρες.*



Σχήμα 5.2: A Koch curve.

5. *Σχεδιάστε μία καμπύλη του Koch με μήκος* $x/3$.
6. *Στρίψτε αριστερά 60 μοίρες.*
7. *Σχεδιάστε μία καμπύλη του Koch με μήκος* $x/3$.

Η εξαίρεση είναι εάν το x *είναι μικρότερο του 3: σε αυτήν την περίπτωση μπορείτε απλά να σχεδιάσετε μία ευθεία γραμμή με μήκος* x.

1. *Γράψτε μία συνάρτηση με όνομα* `koch` *η οποία θα παίρνει μία χελώνα και ένα μήκος σαν παραμέτρους και χρησιμοποιεί τη χελώνα για να σχεδιάσει μία καμπύλη του Koch με το δοθέν μήκος.*
2. *Γράψτε μία συνάρτηση με όνομα* `snowflake` *η οποία θα σχεδιάζει τρεις καμπύλες του Koch για να φτιάξει το περίγραμμα μίας νιφάδας χιονιού.*

   Λύση : `http://thinkpython.com/code/koch.py`.
3. *Η καμπύλη του Koch μπορεί να γενικευθεί με διάφορους τρόπους. Δείτε εδώ:* `http://en.wikipedia.org/wiki/Koch_snowflake` *κάποια παραδείγματα και υλοποιήστε όποιο σας αρέσει.*

Κεφάλαιο 6

Γόνιμες Συναρτήσεις

6.1 Επιστρεφόμενες τιμές

Μερικές από τις ενσωματωμένες συναρτήσεις που έχουμε χρησιμοποιήσει, όπως οι μαθηματικές συναρτήσεις, παράγουν αποτελέσματα. Καλώντας τη συνάρτηση δημιουργείται μία τιμή, την οποία συνήθως την εκχωρούμε σε μία μεταβλητή ή τη χρησιμοποιούμε ως μέρος μίας έκφρασης.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

Όλες οι συναρτήσεις που έχουμε γράψει μέχρι τώρα είναι κενές (void). Εμφανίζουν κάτι στην έξοδο ή μετακινούν χελώνες αλλά η επιστρεφόμενη τιμή τους είναι `None`.

Σε αυτό το κεφάλαιο θα γράψουμε (επιτέλους) γόνιμες (fruitful) συναρτήσεις. Το πρώτο παράδειγμα είναι η `area`, η οποία επιστρέφει το εμβαδόν ενός κύκλου με τη δοθείσα ακτίνα:

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

Έχουμε ξαναδεί τη δήλωση `return` και νωρίτερα, αλλά σε μία γόνιμη συνάρτηση η δήλωση `return` περιλαμβάνει και μία έκφραση. Αυτή η δήλωση σημαίνει: "Επέστρεψε αμέσως από αυτή τη συνάρτηση και χρησιμοποίησε την ακόλουθη έκφραση σαν επιστρεφόμενη τιμή". Η έκφραση μπορεί να είναι αυθαίρετα πολύπλοκη, έτσι θα μπορούσαμε να είχαμε γράψει αυτή τη συνάρτηση πιο συνοπτικά:

```
def area(radius):
    return math.pi * radius**2
```

Αλλά από την άλλη πλευρά, οι **προσωρινές μεταβλητές** όπως η `temp` κάνουν συχνά την αποσφαλμάτωση ευκολότερη.

Μερικές φορές είναι χρήσιμο να έχουμε πολλαπλές δηλώσεις επιστροφής, μία για κάθε κλάδο της συνθήκης:

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

Από τη στιγμή που αυτές οι δηλώσεις επιστροφής είναι μέσα σε μία εναλλακτική συνθήκη τότε θα εκτελεστεί μόνο μία.

Η συνάρτηση τερματίζει αμέσως μόλις εκτελεστεί μία δήλωση επιστροφής, χωρίς να εκτελεσθεί καμία από τις επόμενες δηλώσεις. Ο κώδικας που βρίσκεται μετά από μία δήλωση επιστροφής `return` ή σε οποιοδήποτε άλλο μέρος που δεν φτάνει ποτέ η ροή εκτέλεσης, ονομάζεται **νεκρός κώδικας** (`dead code`).

Καλό θα ήταν, να υπάρχει μία δήλωση `return` για κάθε δυνατό μονοπάτι δια μέσου του κώδικα σε μία γόνιμη συνάρτηση. Για παράδειγμα:

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

Αυτή η συνάρτηση είναι λάθος γιατί εάν το `x` είναι 0, τότε καμία από τις συνθήκες δεν είναι επαληθεύεται και η συνάρτηση τερματίζει χωρίς να έχει "πετύχει" κάποια δήλωση `return`. Εάν η ροή εκτέλεσης φτάσει στο τέλος μίας συνάρτησης τότε η επιστρεφόμενη τιμή είναι `None`, η οποία δεν είναι η απόλυτη τιμή του 0.

```
>>> print absolute_value(0)
None
```

Παρεμπιπτόντως, η Python παρέχει μία ενσωματωμένη συνάρτηση με όνομα `abs` η οποία υπολογίζει απόλυτες τιμές.

Άσκηση 6.1. *Γράψτε μία συνάρτηση* `compare` *η οποία θα επιστρέφει* `1` *αν* `x > y`*,* `0` *αν* `x == y`*, και* `-1` *αν* `x < y`*.*

6.2 Σταδιακή ανάπτυξη

Όσο θα γράφετε όλο και μεγαλύτερες συναρτήσεις, ίσως διαπιστώσετε ότι αφιερώνετε περισσότερο χρόνο για αποσφαλμάτωση.

Για την αντιμετώπιση όλο και πιο πολύπλοκων προγραμμάτων, καλό θα ήταν να δοκιμάσετε μία διαδικασία που ονομάζεται **σταδιακή ανάπτυξη** (incremental development). Στόχος αυτής της διαδικασίας είναι η αποφυγή μεγάλων διαστημάτων αποσφαλμάτωσης, προσθέτοντας και δοκιμάζοντας μικρά κομμάτια κώδικα κάθε φορά.

Για παράδειγμα, υποθέστε ότι θέλετε να βρείτε την απόσταση μεταξύ δύο σημείων δοθέντων των συντεταγμένων (x_1, y_1) και (x_2, y_2). Βάση του Πυθαγόρειου θεωρήματος, η απόσταση δίνεται από τον τύπο:

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

Το πρώτο βήμα είναι να εξετάσετε πως θα πρέπει να είναι μία συνάρτηση απόστασης στην Python. Με άλλα λόγια, ποιες θα είναι οι είσοδοι (παράμετροι) και ποια θα είναι η έξοδος (επιστρεφόμενη τιμή).

Σε αυτήν τη περίπτωση, οι είσοδοι είναι δύο σημεία, τα οποία μπορείτε να αναπαραστήσετε χρησιμοποιώντας τέσσερις αριθμούς, και η επιστρεφόμενη τιμή είναι η απόσταση, η οποία είναι μία τιμή κινητής υποδιαστολής.

Είστε ήδη σε θέση να γράψετε ένα περίγραμμα της συνάρτησης:

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

Προφανώς, αυτή η έκδοση δεν υπολογίζει αποστάσεις και επιστρέφει πάντα μηδέν. Αλλά συντακτικά είναι σωστή και τρέχει, το οποίο σημαίνει ότι μπορείτε να την δοκιμάσετε πριν την κάνετε πιο πολύπλοκη.

Μπορείτε καλέστε την συνάρτηση με κάποια τυχαία ορίσματα για να την δοκιμάσετε:

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

Επέλεξα αυτές τις τιμές ούτως ώστε η οριζόντια απόσταση να είναι 3 και η κάθετη απόσταση να είναι 4. Με αυτόν τον τρόπο το αποτέλεσμα βγαίνει 5 (η υποτείνουσα ενός 3-4-5 τριγώνου). Όταν δοκιμάζουμε μία συνάρτηση, είναι χρήσιμο να γνωρίζουμε τη σωστή απάντηση εκ των προτέρων.

Σε αυτό το σημείο, έχουμε επιβεβαιώσει ότι η συνάρτηση είναι συντακτικά σωστή και μπορούμε να αρχίσουμε να προσθέτουμε κώδικα στο σώμα της. Το επόμενο λογικό βήμα είναι να βρούμε τις διαφορές $x_2 - x_1$ και $y_2 - y_1$. Η επόμενη έκδοση αποθηκεύει αυτές τις τιμές σε προσωρινές μεταβλητές και τις εμφανίζει στην έξοδο.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print 'dx is', dx
    print 'dy is', dy
    return 0.0
```

Εάν η συνάρτηση δουλεύει, θα πρέπει να εμφανίσει `'dx is 3'` και `'dy is 4'`. Τότε ξέρουμε ότι η συνάρτηση παίρνει τα σωστά ορίσματα και εκτελεί σωστά τον πρώτο υπολογισμό. Εάν όχι, πρέπει να ελέγξουμε μόνο μερικές γραμμές κώδικα.

Στη συνέχεια, υπολογίζουμε το άθροισμα των τετραγώνων των `dx` και `dy`:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print 'dsquared is: ', dsquared
    return 0.0
```

Σε αυτό το στάδιο θα πρέπει να εκτελέσετε πάλι το πρόγραμμα και να ελέγξετε την έξοδο (η οποία θα πρέπει να είναι 25). Και τέλος μπορείτε να χρησιμοποιήσετε την `math.sqrt` για να υπολογίσετε και να επιστρέψετε το αποτέλεσμα:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

Εάν αυτό δουλεύει σωστά τότε έχετε τελειώσει. Αλλιώς, καλό θα ήταν να εμφανίσετε την τιμή της `result` πριν τη δήλωση επιστροφής.

Η τελική έκδοση της συνάρτησης δεν εμφανίζει τίποτα όταν τρέχει, επιστρέφει μόνο μία τιμή. Οι δηλώσεις `print` που γράψαμε είναι χρήσιμες μόνο για αποσφαλμάτωση αλλά από τη στιγμή που

έχετε μία λειτουργική συνάρτηση, καλό θα ήταν να τις αφαιρέσετε. Ένας τέτοιος κώδικας ονομάζεται **σκαλωσιά** (scaffolding) επειδή είναι χρήσιμος για την χτίσιμο του προγράμματος αλλά δεν είναι μέρος του τελικού προϊόντος.

Στην αρχή, καλό θα ήταν να προσθέτετε μία ή δύο γραμμές κώδικα κάθε φορά. Όσο όμως αποκτάτε περισσότερη εμπειρία τότε θα διαπιστώσετε ότι θα γράφετε και θα αποσφαλματώνετε μεγαλύτερα κομμάτια κώδικα. Σε κάθε περίπτωση, η σταδιακή ανάπτυξη μπορεί να σας γλυτώσει από περιττό χρόνο αποσφαλμάτωσης.

Οι βασικές πτυχές της διαδικασίας είναι:

1. Ξεκινήστε με ένα λειτουργικό πρόγραμμα και κάντε μικρές και σταδιακές αλλαγές. Έτσι, αν υπάρχει κάποιο λάθος σε οποιοδήποτε σημείο τότε θα ξέρετε που περίπου είναι.
2. Χρησιμοποιείστε προσωρινές μεταβλητές για να κρατάτε τις ενδιάμεσες τιμές ούτως ώστε να είστε σε θέση να τις εμφανίσετε και να τις ελέγξετε.
3. Μόλις το πρόγραμμα γίνει λειτουργικό τότε μπορείτε να αφαιρέσετε τη σκαλωσιά ή ακόμα και να συνενώσετε πολλαπλές δηλώσεις σε σύνθετες εκφράσεις, αλλά μόνο εάν δεν επηρεάζουν αρνητικά την ανάγνωση του προγράμματος.

Άσκηση 6.2. *Γράψτε μία συνάρτηση με όνομα* `hypotenuse`*, χρησιμοποιώντας τη σταδιακή ανάπτυξη, η οποία θα επιστρέφει το μήκος της υποτείνουσας ενός ορθογωνίου τριγώνου περνώντας τα μήκη των δύο κάθετων πλευρών σαν ορίσματα. Καταγράψτε κάθε στάδιο της ανάπτυξης όσο προχωράτε.*

6.3 Σύνθεση

Όπως θα περιμένατε, μπορείτε να καλέσετε μία συνάρτηση μέσα από μία άλλη. Αυτή η δυνατότητα ονομάζεται **σύνθεση** (composition).

Σαν παράδειγμα, θα γράψουμε μία συνάρτηση η οποία θα παίρνει δύο σημεία, το κέντρο του κύκλου και ένα σημείο της περιφέρειας, και θα υπολογίζει το εμβαδόν του κύκλου.

Ας υποθέσουμε ότι το κεντρικό σημείο αποθηκεύεται στις μεταβλητές `xc` και `yc`, και το σημείο της περιφέρειας στις `xp` και `yp`. Το πρώτο βήμα είναι να βρούμε την ακτίνα του κύκλου, η οποία είναι ίση με την απόσταση των δύο σημείων. Αυτό το κάνει η προηγούμενη συνάρτηση (`distance`) που γράψαμε:

```
radius = distance(xc, yc, xp, yp)
```

Το επόμενο βήμα είναι να βρούμε το εμβαδόν του κύκλου με αυτήν την ακτίνα, το οποίο επίσης έχουμε γράψει νωρίτερα:

```
result = area(radius)
```

Εμπερικλείοντας αυτά τα βήματα σε μία συνάρτηση, παίρνουμε:

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

Οι προσωρινές μεταβλητές `radius` και `result` είναι χρήσιμες για την ανάπτυξη και την αποσφαλμάτωση, αλλά από τη στιγμή που το πρόγραμμα είναι λειτουργικό, μπορούμε να το κάνουμε πιο συνοπτικό συνθέτοντας τις κλήσεις συναρτήσεων:

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 Λογικές συναρτήσεις

Οι συναρτήσεις μπορούν να επιστρέφουν και λογικές τιμές, το οποίο είναι βολικό για να κρύβουμε πολύπλοκους ελέγχους μέσα σε συναρτήσεις. Για παράδειγμα :

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

Στις λογικές συναρτήσεις δίνουμε συνήθως ονόματα τα οποία μοιάζουν με ερωτήσεις μονολεκτικής απάντησης (ναι ή όχι). Η `is_divisible` επιστρέφει `True` ή `False` για να υποδείξει εάν το `x` διαιρείται από το `y` ή όχι.

Παράδειγμα:

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

Από τη στιγμή όμως που το αποτέλεσμα του τελεστή == είναι μία λογική τιμή, μπορούμε να γράψουμε την προηγούμενη συνάρτηση πιο συνοπτικά επιστρέφοντας κατευθείαν αυτήν την τιμή:

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

Οι λογικές συναρτήσεις χρησιμοποιούνται συχνά σε δηλώσεις υπό συνθήκη:

```
if is_divisible(x, y):
    print 'x is divisible by y'
```

Ίσως μπείτε στον πειρασμό να γράψετε κάτι τέτοιο:

```
if is_divisible(x, y) == True:
    print 'x is divisible by y'
```

Αλλά η επιπλέον σύγκριση είναι περιττή.

Άσκηση 6.3. *Γράψτε μία συνάρτηση με όνομα* `is_between(x, y, z)` *η οποία θα επιστρέφει* `True` *αν* $x \leq y \leq z$ *και* `False` *αλλιώς.*

6.5 Περισσότερη αναδρομή

Έχουμε καλύψει μόνο ένα μικρό υποσύνολο της Python, αλλά αυτό το υποσύνολο είναι μία πλήρης γλώσσα προγραμματισμού. Αυτό σημαίνει ότι οτιδήποτε μπορεί να υπολογιστεί μπορεί να εκφραστεί σε αυτή τη γλώσσα. Κάθε πρόγραμμα που έχει ποτέ γραφτεί, θα μπορούσε να ξαναγραφτεί χρησιμοποιώντας μόνο τα χαρακτηριστικά της γλώσσας που έχετε μάθει μέχρι τώρα (στην πραγματικότητα χρειάζεστε μερικές εντολές ακόμα για να ελέγχετε συσκευές όπως το πληκτρολόγιο, το ποντίκι, τους δίσκους, κλπ. αλλά τίποτα περισσότερο).

Η απόδειξη αυτού του ισχυρισμού είναι μία τετριμμένη άσκηση, η λύση της οποίας επιτεύχθηκε πρώτη φορά από τον Alan Turing, έναν από τους πρώτους επιστήμονες της πληροφορικής (κάποιοι ισχυρίζονται ότι ήταν μαθηματικός αλλά πολλοί από τους πρώτους επιστήμονες της πληροφορικής ξεκίνησαν σαν μαθηματικοί). Για αυτό τον λόγο είναι γνωστή και ως Διατριβή Turing. Για μία πιο ολοκληρωμένη (και ακριβής) ανάλυση της Διατριβής Turing προτείνω το βιβλίο του Michael Sipser *Introduction to the Theory of Computation*.

Για να σας δώσω μία ιδέα του τι μπορείτε να κάνετε με όσα μάθατε μέχρι τώρα, θα αποτιμήσουμε μερικές αναδρομικά οριζόμενες μαθηματικές συναρτήσεις. Ένας αναδρομικός ορισμός είναι παρόμοιος με κυκλικό ορισμό, με την έννοια ότι ο ορισμός περιέχει μία αναφορά σε κάτι το οποίο έχει οριστεί ήδη. Ένας αμιγώς κυκλικός ορισμός δεν είναι ιδιαίτερα χρήσιμος:

θανάσιμο: Ένας επιθετικός προσδιορισμός που χρησιμοποιείται για να περιγράψει ότι κάτι είναι θανάσιμο.

Κατά πάσα πιθανότητα θα σας ενοχλούσε αν βλέπατε αυτόν τον ορισμό σε ένα λεξικό. Αλλά από την άλλη μεριά, αν κοιτάξετε τον ορισμό της παραγοντικής συνάρτησης, η οποία συμβολίζεται με θαυμαστικό !, τότε θα δείτε το εξής:

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

Αυτός ο ορισμός λέει ότι το παραγοντικό του 0 είναι 1 και το παραγοντικό οποιουδήποτε άλλου αριθμού n είναι το n πολλαπλασιαζόμενο με το παραγοντικό του $n-1$.

Έτσι, το 3! είναι 3 φορές το 2! το οποίο είναι 2 φορές το 1! το οποίο είναι 1 φορά το 0!. Βάζοντάς τα όλα μαζί προκύπτει ότι το 3! είναι ίσο με 3 φορές το 2 φορές το 1 φορά το 1, το οποίο είναι 6.

Αν είστε σε θέση να γράψετε κάποιον αναδρομικό ορισμό, τότε συνήθως μπορείτε να γράψετε και ένα πρόγραμμα σε Python για να τον αποτιμήσει. Το πρώτο βήμα είναι να αποφασίσουμε ποιες πρέπει να είναι οι παράμετροι και σε αυτήν την περίπτωση είναι ξεκάθαρο ότι η συνάρτηση `factorial` θα παίρνει έναν ακέραιο:

```
def factorial(n):
```

Εάν το όρισμα είναι 0, τότε το μόνο που πρέπει να κάνουμε είναι να επιστρέψουμε 1:

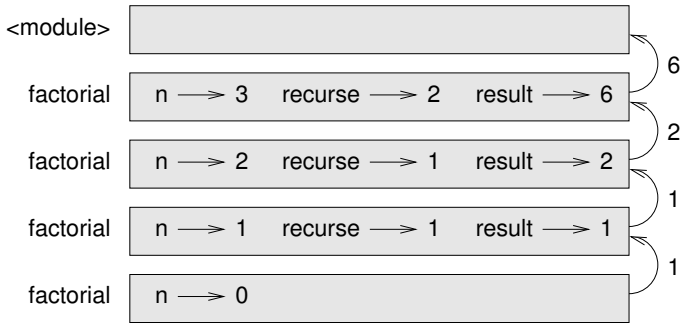
```
def factorial(n):
    if n == 0:
        return 1
```

Αλλιώς, και αυτό είναι το ενδιαφέρον κομμάτι, πρέπει να κάνουμε μία αναδρομική κλήση για να βρούμε το παραγοντικό του $n-1$ και μετά να το πολλαπλασιάσουμε με το n:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

Η ροή εκτέλεσης αυτού του προγράμματος είναι παρόμοια με τη ροή της `countdown` στην Ενότητα 5.8. Αν καλέσουμε την `factorial` με τιμή 3:

Αφού το 3 δεν είναι 0, παίρνουμε το δεύτερο κλάδο και υπολογίζουμε το παραγοντικό του `n-1`...



Σχήμα 6.1: Διάγραμμα Στοίβας.

Αφού το 2 δεν είναι 0, παίρνουμε το δεύτερο κλάδο και υπολογίζουμε το παραγοντικό του `n-1`...

Αφού το 1 δεν είναι 0, παίρνουμε το δεύτερο κλάδο και υπολογίζουμε το παραγοντικό του `n-1`...

Αφού το 0 είναι 0, παίρνουμε τον πρώτο κλάδο και επιστρέφουμε 1 χωρίς να κάνουμε άλλες αναδρομικές κλήσεις.

Η επιστρεφόμενη τιμή (1) πολλαπλασιάζεται με n, το οποίο είναι 1, και το αποτέλεσμα επιστρέφεται.

Η επιστρεφόμενη τιμή (1) πολλαπλασιάζεται με n, το οποίο είναι 2, και το αποτέλεσμα επιστρέφεται.

Η επιστρεφόμενη τιμή (2) πολλαπλασιάζεται με n, το οποίο είναι 3, και το αποτέλεσμα (6) γίνεται η επιστρεφόμενη τιμή της κλήσης συνάρτησης η οποία ξεκίνησε όλη τη διαδικασία.

Η εικόνα 6.1 δείχνει πως είναι το διάγραμμα στοίβας για αυτή τη σειρά των κλήσεων της συνάρτησης.

Οι επιστρεφόμενες τιμές εμφανίζονται να περνάνε προς τα πίσω στη στοίβα. Σε κάθε πλαίσιο, η επιστρεφόμενη τιμή είναι η τιμή της `result` η οποία είναι το γινόμενο της `n` με την `recurse`.

Στο τελευταίο πλαίσιο, οι τοπικές μεταβλητές `recurse` και `result` δεν υπάρχουν γιατί ο κλάδος που τις δημιουργεί δεν εκτελείται.

6.6 Άλμα πίστης

Ένας τρόπος να διαβάζουμε προγράμματα είναι να ακολουθούμε τη ροή εκτέλεσης, το οποίο όμως μπορεί να γίνει πολύ γρήγορα λαβύρινθος. Ένας εναλλακτικός τρόπος είναι αυτό που εγώ ονομάζω "άλμα πίστης" (leap of faith). Όταν βρεθείτε σε μία κλήση συνάρτησης, αντί να ακολουθήσετε τη ροή της εκτέλεσης, υποθέστε ότι η συνάρτηση δουλεύει σωστά και επιστρέφει το σωστό αποτέλεσμα.

Στην πραγματικότητα, εφαρμόζετε ήδη αυτό το άλμα πίστης όταν χρησιμοποιείτε τις ενσωματωμένες συναρτήσεις. Όταν καλείτε την `math.cos` ή `math.exp`, δεν εξετάζετε τα σώματα αυτών των συναρτήσεων. Απλώς θεωρείτε ότι δουλεύουν επειδή οι άνθρωποι που γράψανε τις ενσωματωμένες συναρτήσεις ήταν καλοί προγραμματιστές.

Το ίδιο ισχύει και όταν καλείτε μία από τις δικές σας συναρτήσεις. Για παράδειγμα, στην Ενότητα 6.4 γράψαμε μία συνάρτηση με όνομα `is_divisible` η οποία προσδιορίζει εάν ένας αριθμός διαιρείται από έναν άλλο. Από τη στιγμή που πειστήκαμε ότι αυτή η συνάρτηση είναι σωστή, εξετάζοντας και δοκιμάζοντας τον κώδικα, μπορούμε να χρησιμοποιήσουμε τη συνάρτηση χωρίς να ξανακοιτάξουμε το σώμα της.

Το ίδιο ισχύει και για τα αναδρομικά προγράμματα. Όταν βρεθείτε σε μία αναδρομική κλήση, αντί να ακολουθήσετε τη ροή της εκτέλεσης, θεωρήστε ότι η αναδρομική κλήση δουλεύει (παράγει το σωστό αποτέλεσμα) και στη συνέχεια αναρωτηθείτε: "Θεωρώντας ότι μπορώ να βρω το παραγοντικό του $n-1$, μπορώ να υπολογίσω το παραγοντικό του n;" Σε αυτή την περίπτωση είναι ξεκάθαρο ότι μπορείτε να το κάνετε πολλαπλασιάζοντας με n.

Φυσικά είναι λίγο περίεργο να θεωρούμε ότι η συνάρτηση δουλεύει σωστά από τη στιγμή που δεν έχουμε τελειώσει το γράψιμό της, αλλά για αυτό ονομάζεται και άλμα πίστης!

6.7 Ένα ακόμα παράδειγμα

Το επόμενο πιο συνηθισμένο παράδειγμα, μετά το παραγοντικό, αναδρομικά οριζόμενης μαθηματικής συνάρτησης είναι το `fibonacci`, του οποίου ο ορισμός είναι (βλ. `http://en.wikipedia.org/wiki/Fibonacci_number`):

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

Γραμμένο σε Python:

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

Αν προσπαθήσετε να ακολουθήσετε τη ροή της εκτέλεσης εδώ, ακόμα και για αρκετά μικρές τιμές του n, το κεφάλι σας θα εκραγεί. Αλλά σύμφωνα με το άλμα πίστης, εάν υποθέσετε ότι οι δύο αναδρομικές κλήσεις δουλεύουν σωστά τότε είναι σαφές ότι αν τις προσθέσετε θα πάρετε το σωστό αποτέλεσμα.

6.8 Έλεγχος τύπων

Τι θα συμβεί αν καλέσουμε την `factorial` και της δώσουμε σαν όρισμα 1.5;

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

Μοιάζει με άπειρη αναδρομή, αλλά πως γίνεται αυτό αφού υπάρχει μία περίπτωση βάσης όταν `n == 0`. Μπορεί να "χάσουμε" την περίπτωση βάσης αν το `n` δεν είναι ακέραιος αριθμός και έτσι να έχουμε αναδρομή επ' άπειρον.

Στην πρώτη αναδρομική κλήση η τιμή του n είναι 0.5 και στην επόμενη είναι -0.5. Από εκεί και πέρα γίνεται όλο και μικρότερη (πιο αρνητική) αλλά ποτέ δεν θα γίνει 0.

Έχουμε δύο επιλογές. Μπορούμε να γενικεύσουμε την συνάρτηση `factorial` για να δουλεύει και με αριθμούς κινητής υποδιαστολής ή να την μετατρέψουμε ούτως ώστε να ελέγχει τον τύπο του ορίσματός της. Η πρώτη επιλογή ονομάζεται συνάρτηση γάμμα και είναι εκτός του σκοπού αυτού του βιβλίου. Επομένως θα πάμε στη δεύτερη επιλογή.

Μπορούμε να χρησιμοποιήσουμε την ενσωματωμένη συνάρτηση `isinstance` για να επαληθεύσουμε τον τύπο του ορίσματος. Επίσης, αφού το κάνουμε αυτό, μπορούμε να σιγουρευτούμε ότι το όρισμα είναι θετικό:

```
def factorial (n):
    if not isinstance(n, int):
        print 'Factorial is only defined for integers.'
        return None
    elif n < 0:
        print 'Factorial is not defined for negative integers.'
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Η πρώτη περίπτωση βάσης χειρίζεται τους μη-ακέραιους και η δεύτερη "συλλαμβάνει" τους αρνητικούς ακέραιους. Και στις δύο περιπτώσεις, το πρόγραμμα εμφανίζει ένα μήνυμα λάθους και επιστρέφει `None` για να υποδηλώσει ότι κάτι πήγε στραβά:

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is not defined for negative integers.
None
```

Εάν περάσουμε και τους δύο ελέγχους, τότε ξέρουμε ότι το n είναι είτε θετικός είτε μηδέν και άρα μπορούμε να αποδείξουμε ότι η αναδρομή τερματίζει.

Αυτό το πρόγραμμα επιδεικνύει ένα πρότυπο το οποίο μερικές φορές ονομάζεται **φύλακας** (guardian). Οι πρώτες δύο συνθήκες λειτουργούν σαν φύλακες, προστατεύοντας τον κώδικα που ακολουθεί από τιμές οι οποίες μπορεί να προκαλέσουν σφάλμα. Οι φύλακες καθιστούν δυνατή την απόδειξη της ορθότητας του κώδικα.

Στην Ενότητα 11.3 θα δούμε μία πιο ευέλικτη εναλλακτική λύση για να εμφανίζουμε ένα μήνυμα λάθους εγείροντας μία εξαίρεση.

6.9 Αποσφαλμάτωση

Με το σπάσιμο ενός μεγάλου προγράμματος σε συναρτήσεις δημιουργούνται φυσικά σημεία ελέγχου για αποσφαλμάτωση. Αν μία συνάρτηση δε δουλεύει τότε πρέπει να εξετάσετε τρία ενδεχόμενα:

- Κάτι πήγε στραβά με τα ορίσματα που παίρνει η συνάρτηση. Δηλαδή παραβιάστηκε κάποια προϋπόθεση.

- Υπάρχει κάποιο λάθος στη συνάρτηση. Δηλαδή παραβιάστηκε κάποια μετασυνθήκη.

- Κάτι είναι λάθος με την επιστρεφόμενη τιμή ή με τον τρόπο που χρησιμοποιήθηκε.

Για να αποκλείσετε το πρώτο ενδεχόμενο, μπορείτε να προσθέσετε μία δήλωση `print` στην αρχή της συνάρτησης και να εμφανίσετε τις τιμές των παραμέτρων (και τους τύπους τους), ή μπορείτε να γράψετε κώδικα ο οποίος θα ελέγχει ρητά τις προϋποθέσεις.

Εάν οι παράμετροι φαίνονται εντάξει, προσθέστε μία δήλωση `print` πριν από κάθε δήλωση `return` η οποία θα εμφανίζει την επιστρεφόμενη τιμή. Εάν είναι εφικτό, ελέγξτε το αποτέλεσμα με το χέρι. Δοκιμάστε να καλέσετε τη συνάρτηση με τιμές που κάνουν εύκολο τον έλεγχο του αποτελέσματος (όπως στην Ενότητα 6.2).

Αν φαίνεται ότι η συνάρτηση δουλεύει, εξετάστε την κλήση συνάρτησης για να σιγουρευτείτε ότι η επιστρεφόμενη τιμή χρησιμοποιείται σωστά (ή αν χρησιμοποιείτε γενικά!)

Η προσθήκη δηλώσεων `print` στην αρχή και στο τέλος μίας συνάρτησης, μπορεί να κάνει πιο ορατή την ροή εκτέλεσης. Για παράδειγμα, αυτή είναι μία έκδοση της `factorial` με δηλώσεις `print`:

```
def factorial(n):
    space = ' ' * (4 * n)
    print space, 'factorial', n
    if n == 0:
        print space, 'returning 1'
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print space, 'returning', result
        return result
```

Το `space` είναι μία συμβολοσειρά από κενούς χαρακτήρες για τη ρύθμιση των εσοχών της εξόδου. Αυτό είναι το αποτέλεσμα της `factorial(5)`:

```
                    factorial 5
                factorial 4
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
                returning 24
                    returning 120
```

Αυτού του είδους η έξοδος μπορεί να σας φανεί πολύ χρήσιμη αν σας μπερδεύει η ροή εκτέλεσης. Χρειάζεται κάποιο χρόνο για να φτιαχτεί μία αποτελεσματική ”σκαλωσιά”, αλλά λίγη σκαλωσιά μπορεί να μας γλιτώσει από πολύ αποσφαλμάτωση.

6.10 Ορολογία

προσωρινή μεταβλητή: Μία μεταβλητή η οποία χρησιμοποιείτε για να αποθηκεύει μία ενδιάμεση τιμή σε έναν σύνθετο υπολογισμό.

νεκρός κώδικας: Μέρος του κώδικα το οποίο δεν μπορεί να εκτελεστεί ποτέ, επειδή εμφανίζεται συνήθως μετά από κάποια δήλωση `return`.

None: Μία ειδική τιμή η οποία επιστρέφεται από συναρτήσεις οι οποίες δεν έχουν καμία δήλωση επιστροφής ή έχουν δήλωση επιστροφής αλλά χωρίς όρισμα.

σταδιακή ανάπτυξη: Ένα πλάνο ανάπτυξης προγραμμάτων που στοχεύει στην αποφυγή της αποσφαλμάτωσης προσθέτοντας και δοκιμάζοντας μικρά κομμάτια κώδικα κάθε φορά.

σκαλωσιά: Κώδικας ο οποίος χρησιμοποιείται κατά την ανάπτυξη ενός προγράμματος αλλά δεν είναι μέρος της τελικής έκδοσης.

φύλακας: Ένα προγραμματιστικό πρότυπο το οποίο χρησιμοποιεί δηλώσεις υπό συνθήκη για να ελέγξει και να διαχειριστεί περιπτώσεις οι οποίες μπορεί να προκαλέσουν κάποιο σφάλμα.

6.11 Ασκήσεις

Άσκηση 6.4. *Σχεδιάστε ένα διάγραμμα στοίβας για το ακόλουθο πρόγραμμα. Τι εμφανίζει το πρόγραμμα;*

Λύση: `http://thinkpython.com/code/stack_diagram.py`.

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

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

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

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

Άσκηση 6.5. *Η συνάρτηση Άκερμαν, $A(m, n)$, ορίζεται:*

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

Βλ. `http://en.wikipedia.org/wiki/Ackermann_function`*. Γράψτε μία συνάρτηση με όνομα* `ack` *υπολογίζει τη συνάρτηση του Άκερμαν. Χρησιμοποιήστε τη συνάρτησή σας για να υπολογίσετε το* `ack(3, 4)`*, το οποίο θα πρέπει να βγει 125. Τι συμβαίνει για μεγαλύτερες τιμές του* `m` *και του* `n`*; Λύση:* `http://thinkpython.com/code/ackermann.py`*.*

Άσκηση 6.6. *Παλίνδρομο είναι μία λέξη η οποία συλλαβίζεται το ίδιο προς τα πίσω και προς τα εμπρός, όπως είναι η "noon" και η "redivider". Αναδρομικά, μία λέξη είναι παλίνδρομο αν το πρώτο και το τελευταίο γράμμα είναι ίδια και τα μεσαία είναι παλίνδρομο.*

Οι ακόλουθες, είναι συναρτήσεις οι οποίες παίρνουν μία συμβολοσειρά σαν όρισμα και επιστρέφουν το πρώτο, το τελευταίο και τα μεσαία γράμματα:

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

Θα δούμε πως δουλεύουν στο Κεφάλαιο 8.

1. *Πληκτρολογήστε αυτές τις συναρτήσεις μέσα σε ένα αρχείο με όνομα* `palindrome.py` *και δοκιμάστε τες. Τι συμβαίνει αν καλέσετε τη* `middle` *με μία συμβολοσειρά με δύο γράμματα; Με ένα γράμμα; Τι γίνεται με μία κενή συμβολοσειρά, η οποία γράφεται έτσι* `''` *και δεν περιέχει καθόλου γράμματα;*
2. *Γράψτε μία συνάρτηση με όνομα* `is_palindrome` *η οποία θα παίρνει σαν όρισμα μία συμβολοσειρά και θα επιστρέφει* `True` *εάν είναι παλίνδρομο ή* `False` *αλλιώς. Θυμηθείτε ότι μπορείτε να χρησιμοποιήσετε την ενσωματωμένη συνάρτηση* `len` *για να ελέγξετε το μήκος μίας συμβολοσειράς.*

Λύση: `http://thinkpython.com/code/palindrome_soln.py`*.*

Άσκηση 6.7. *Ένας αριθμός a είναι δύναμη του b εάν διαιρείται από τον b και το a/b είναι δύναμη του b. Γράψτε μία συνάρτηση με όνομα* `is_power` *η οποία θα παίρνει σαν παραμέτρους το* `a` *και το* `b` *και θα επιστρέφει* `True` *αν το* `a` *είναι δύναμη του* `b`*. Σημείωση: θα πρέπει να σκεφτείτε την περίπτωση βάσης.*

Άσκηση 6.8. *Ο μέγιστος κοινός διαιρέτης (GCD) του a και του b είναι ο μεγαλύτερος αριθμός που διαιρεί και τους δύο χωρίς να αφήνει υπόλοιπο.*

Ένας τρόπος για να βρούμε τον GCD δύο αριθμών είναι ο αλγόριθμος του Ευκλείδη. Αυτός ο αλγόριθμος βασίζεται παρατήρηση ότι αν το r είναι το υπόλοιπο όταν το a διαιρείται από το b, τότε $gcd(a, b) = gcd(b, r)$. Σαν περίπτωση βάσης μπορούμε να χρησιμοποιήσουμε την $gcd(a, 0) = a$.

Γράψτε μία συνάρτηση με όνομα `gcd` *η οποία παίρνει σαν παραμέτρους το* `a` *και το* `b` *και θα επιστρέφει τον μέγιστο κοινό διαιρέτη τους. Για περαιτέρω βοήθεια δείτε εδώ:* `http://en.wikipedia.org/wiki/Euclidean_algorithm`*.*

Αναφορά: Αυτή η άσκηση βασίζεται σε ένα παράδειγμα του βιβλίου Structure and Interpretation of Computer Programs *των Άμπελσον και Σούσμαν.*

Κεφάλαιο 7

Επανάληψη

7.1 Πολλαπλή εκχώρηση

Ενδεχομένως να έχετε διαπιστώσει ότι είναι έγκυρο να κάνουμε περισσότερες από μία εκχωρήσεις στην ίδια μεταβλητή. Μία νέα εκχώρηση κάνει μία μεταβλητή να αναφέρεται σε μία νέα τιμή (και παύει η αναφορά στην παλιά τιμή).

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

Η έξοδος αυτού του προγράμματος είναι `5 7`, επειδή την πρώτη φορά που τυπώνεται η `bruce` η τιμή της είναι 5, ενώ τη δεύτερη φορά η τιμή της είναι 7. Το κόμμα στο τέλος της πρώτης δήλωσης `print` καταστέλλει τη νέα γραμμή και γι' αυτόν το λόγο εμφανίζονται και οι δύο έξοδοι στην ίδια γραμμή.

Η εικόνα 7.1 απεικονίζει την **πολλαπλή εκχώρηση** (multiple assignmemt) σε ένα διάγραμμα κατάστασης.

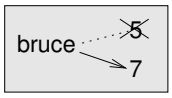
Στην πολλαπλή εκχώρηση, είναι ιδιαίτερα σημαντικό να γίνει διάκριση μεταξύ μίας εκχώρησης τιμής και μίας δήλωσης ισότητας. Επειδή η Python χρησιμοποιεί το σύμβολο της ισότητας (`=`) για εκχώρηση, τυχαίνει να ερμηνεύουμε συχνά μία τέτοια δήλωση `a = b` σαν μία δήλωση ισότητας. Δεν είναι!

Πρώτον, η ισότητα είναι μία συμμετρική σχέση ενώ η εκχώρηση όχι. Στα μαθηματικά για παράδειγμα, εάν $a = 7$ τότε και $7 = a$. Στην Python όμως, ενώ η δήλωση `a = 7` είναι έγκυρη, η `7 = a` δεν είναι.

Επί πλέον, στα μαθηματικά, μία δήλωση ισότητας είναι είτε αληθής είτε ψευδής για πάντα. Αν $a = b$ τώρα, τότε το a θα είναι πάντα ίσο με το b. Στην Python, μία δήλωση εκχώρησης μπορεί να κάνει δύο μεταβλητές ίσες αλλά δεν είναι απαραίτητο ότι θα μείνουν πάντα έτσι:

```
a = 5
b = a    # a and b are now equal
a = 3    # a and b are no longer equal
```

Η τρίτη γραμμή αλλάζει την τιμή της `a` αλλά δεν αλλάζει την τιμή της `b` και επομένως δεν είναι πλέον ίσες.



Σχήμα 7.1: Διάγραμμα κατάστασης.

Παρόλο που η πολλαπλή εκχώρηση είναι συχνά χρήσιμη, θα πρέπει να τη χρησιμοποιείτε με προσοχή. Μπορεί να δυσκολέψει η ανάγνωση και η αποσφαλμάτωση του κώδικα αν αλλάζουν συχνά οι τιμές των μεταβλητών.

7.2 Ενημέρωση μεταβλητών

Μία από τις συχνότερες μορφές της πολλαπλής εκχώρησης είναι η **ενημέρωση** (update), όπου η νέα τιμή της μεταβλητής εξαρτάται από την παλιά:

```
x = x+1
```

Αυτό σημαίνει: "πάρε την τιμή του x, πρόσθεσε ένα και μετά ενημέρωσε το x με τη νέα τιμή".

Αν προσπαθήσετε να ενημερώσετε μία μεταβλητή η οποία δεν υπάρχει, τότε θα πάρετε μήνυμα λάθους. Αυτό συμβαίνει επειδή η Python υπολογίζει πρώτα το δεξιό μέρος πριν εκχωρήσει την τιμή στο x:

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

Για να είστε σε θέση να ενημερώσετε μία μεταβλητή, θα πρέπει πρώτα να την **αρχικοποιήσετε** (initialize) και αυτό συνήθως γίνεται με μία απλή εκχώρηση:

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

Η ενημέρωση μίας μεταβλητής προσθέτοντας της 1 ονομάζεται **προσαύξηση**, ενώ αφαιρώντας 1 ονομάζεται **μείωση** (decrement) της μεταβλητής.

7.3 Η δήλωση `while`

Οι υπολογιστές χρησιμοποιούνται συχνά για την αυτοματοποίηση επαναλαμβανόμενων εργασιών. Η επανάληψη πανομοιότυπων ή παρόμοιων εργασιών χωρίς λάθη είναι εύκολη υπόθεση για τους υπολογιστές αλλά δεν ισχύει το ίδιο και για τους ανθρώπους.

Έχουμε δει δύο προγράμματα, το `countdown` και το `print_n`, τα οποία χρησιμοποιούν αναδρομή για να εκτελέσουν **επανάληψη** (iteration/repetition). Επειδή η επανάληψη χρησιμοποιείται συχνά, η Python παρέχει διάφορες λειτουργίες για να την κάνει ευκολότερη. Μία από αυτές είναι η δήλωση `for` που είδαμε στην Ενότητα 4.2 και θα επανέλθουμε σε αυτήν αργότερα.

Μία άλλη είναι η δήλωση `while`. Αυτή είναι μία έκδοση της `countdown` η οποία χρησιμοποιεί μία δήλωση `while`:

```
def countdown(n):
    while n > 0:
        print n
        n = n-1
```

```
    print 'Blastoff!'
```

Η δήλωση `while` διαβάζεται σχεδόν σαν να μία πρόταση στα Αγγλικά. Αυτό σημαίνει ότι : ” Όσο το `n` είναι μεγαλύτερο του 0, εμφάνισε την τιμή του `n` και στη συνέχεια μείωσε την τιμή του κατά 1. Όταν φτάσεις στο 0, εμφάνισε τη λέξη `Blastoff`!”

Πιο επίσημα, η ροή εκτέλεσης για μία δήλωση `while` είναι η εξής :

1. Αξιολόγησε τη συνθήκη επιστρέφοντας `True` ή `False`.
2. Αν η συνθήκη είναι ψευδής, βγες από τη δήλωση `while` και συνέχισε με την εκτέλεση της επόμενης δήλωσης.
3. Αν η συνθήκη είναι αληθής, εκτέλεσε το σώμα και μετά επέστρεψε στο πρώτο βήμα.

Αυτός ο τύπος ροής ονομάζεται **βρόχος** (loop) επειδή το τρίτο βήμα επιστρέφει τη ροή στην αρχή.

Το σώμα του βρόχου θα πρέπει να αλλάζει την τιμή ενός ή περισσοτέρων μεταβλητών έτσι ώστε τελικά η συνθήκη να γίνει ψευδής και να τερματίσει ο βρόχος. Σε οποιαδήποτε άλλη περίπτωση, ο βρόχος θα επαναλαμβάνεται επ’ άπειρον και ένας τέτοιος βρόχος ονομάζεται **ατέρμων βρόχος** (infinite loop). Μία ατέλειωτη πηγή διασκέδασης για τους επιστήμονες της πληροφορικής είναι η παρατήρηση ότι οι οδηγίες για ένα σαμπουάν, ”Σαπουνίστε, ξεπλύνετε και επαναλάβετε”, αποτελούν έναν ατέρμων βρόχο.

Στην περίπτωση της `countdown`, μπορούμε να αποδείξουμε ότι ο βρόχος τερματίζει επειδή ξέρουμε ότι η τιμή του `n` είναι πεπερασμένη, και μπορούμε να δούμε ότι η τιμή του μειώνεται σε κάθε επανάληψη του βρόχου, έτσι ώστε τελικά να πάρει την τιμή 0. Σε άλλες περιπτώσεις όμως δεν είναι τόσο εύκολο:

```
def sequence(n):
    while n != 1:
        print n,
        if n%2 == 0:        # n is even
            n = n/2
        else:               # n is odd
            n = n*3+1
```

Η συνθήκη για αυτό το βρόχο είναι η `n != 1`, άρα ο βρόχος συνεχίζει να εκτελείται μέχρι το `n` να γίνει `1`, το οποίο θα κάνει τη συνθήκη ψευδή.

Σε κάθε επανάληψη, το πρόγραμμα εμφανίζει την τιμή του `n` και ελέγχει εάν είναι άρτιος ή περιττός. Αν είναι άρτιος, το `n` διαιρείται με το 2. Αν είναι περιττός, η τιμή του `n` αντικαθίσταται με `n*3+1`. Για παράδειγμα, αν περάσουμε το 3 σαν όρισμα στην `sequence`, η σειρά των αποτελεσμάτων θα είναι: 3, 10, 5, 16, 8, 4, 2, 1.

Από τη στιγμή που το `n` άλλες φορές αυξάνεται και άλλες φορές μειώνεται, δεν υπάρχει κάποια προφανής απόδειξη ότι το `n` θα γίνει κάποια στιγμή ίσο με 1, ή ότι το πρόγραμμα τερματίζει. Για κάποιες συγκεκριμένες τιμές του `n`, μπορούμε να αποδείξουμε τον τερματισμό του προγράμματος. Για παράδειγμα, αν η αρχική τιμή του `n` είναι μία δύναμη του δύο, τότε η τιμή του `n` σε κάθε επανάληψη θα είναι άρτια μέχρι να γίνει ίση με 1. Το προηγούμενο παράδειγμα τελειώνει με μία τέτοια ακολουθία, ξεκινώντας με 16.

Το δύσκολο είναι να αποδείξουμε ότι το πρόγραμμα τερματίζει για οποιαδήποτε θετική τιμή του `n`. Μέχρι στιγμής, κανείς δεν μπόρεσε να το αποδείξει αλλά ούτε και να το διαψεύσει! (Βλ.`http://en.wikipedia.org/wiki/Collatz_conjecture`).

Άσκηση 7.1. *Ξαναγράψτε τη συνάρτηση* `print_n` *της Ενότητας 5.8 χρησιμοποιώντας επανάληψη αντί για αναδρομή.*

7.4 Η δήλωση `break`

Μερικές φορές δεν ξέρετε ότι ήρθε η ώρα για να τερματίσει ένας βρόχος μέχρι να φτάσετε στη μέση του σώματός του. Σε αυτήν την περίπτωση μπορείτε να χρησιμοποιήσετε τη δήλωση `break` για να βγείτε έξω από το βρόχο.

Για παράδειγμα, υποθέστε ότι θέλετε να δέχεστε είσοδο από το χρήστη μέχρις ότου πληκτρολογήσει `done`. Μπορείτε να γράψετε:

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

print 'Done!'
```

Η συνθήκη τερματισμού του βρόχου είναι η `True`, η οποία είναι πάντα αληθής, και άρα ο βρόχος εκτελείται μέχρι να φτάσει στην δήλωση break.

Σε κάθε επανάληψη, ζητάει είσοδο από το χρήστη με το σύμβολο '>'. Αν ο χρήστης πληκτρολογήσει `done`, τότε βγαίνει από το βρόχο με τη δήλωση `break`. Αλλιώς, το πρόγραμμα εμφανίζει ότι πληκτρολογεί ο χρήστης και επιστρέφει στην αρχή του βρόχου. Αυτό είναι ένα δείγμα εκτέλεσης:

```
> not done
not done
> done
Done!
```

Αυτός ο τρόπος γραφής βρόχων `while` είναι πολύ κοινός επειδή μπορείτε να ελέγξετε τη συνθήκη οπουδήποτε μέσα στο βρόχο (όχι μόνο στην αρχή) και επίσης μπορείτε να εκφράσετε τη συνθήκη τερματισμού καταφατικά (σταμάτα όταν συμβεί αυτό) αντί για αρνητικά (συνέχισε μέχρι να συμβεί αυτό).

7.5 Τετραγωνικές ρίζες

Οι βρόχοι χρησιμοποιούνται συχνά σε προγράμματα τα οποία υπολογίζουν αριθμητικά αποτελέσματα αρχίζοντας με μία κατά προσέγγιση απάντηση την οποία βελτιώνουν σε κάθε επανάληψη.

Για παράδειγμα, ένας τρόπος υπολογισμού της τετραγωνικής ρίζας ενός αριθμού είναι η μέθοδος του Νιούτον. Υποθέστε ότι θέλετε να υπολογίσετε την τετραγωνική ρίζα του a. Αν ξεκινήσετε με σχεδόν οποιαδήποτε εκτίμηση x, τότε μπορείτε να υπολογίσετε μία καλύτερη εκτίμηση με βάση τον ακόλουθο τύπο:

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

Για παράδειγμα, αν το a είναι 4 και το x είναι 3:

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

Το οποίο είναι πιο κοντά στη σωστή απάντηση ($\sqrt{4} = 2$). Αν επαναλάβουμε τη διαδικασία με τη νέα εκτίμηση, τότε πλησιάζουμε ακόμα περισσότερο:

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

Μετά από μερικές ακόμα ενημερώσεις, η εκτίμηση είναι σχεδόν ακριβής:

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

Σε γενικές γραμμές δεν γνωρίζουμε εκ των προτέρων πόσα βήματα θα χρειαστούμε για να φτάσουμε στη σωστή απάντηση, αλλά ξέρουμε ότι φτάσαμε όταν η εκτίμηση σταματήσει να αλλάζει:

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

Όταν `y == x`, τότε μπορούμε να σταματήσουμε. Ακολουθεί ένας βρόχος ο οποίος ξεκινάει με μία αρχική εκτίμηση `x`, και τη βελτιώνει μέχρι να σταματήσει να αλλάζει:

```
while True:
    print x
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

Για τις περισσότερες τιμές του `a` δουλεύει καλά, αλλά γενικά είναι επίφοβο να ελέγχουμε την ισότητα δεκαδικών αριθμών. Οι τιμές των αριθμών κινητής υποδιαστολής είναι σωστές μόνο κατά προσέγγιση. Οι περισσότεροι ρητοί όπως το $1/3$ και άρρητοι αριθμοί όπως η $\sqrt{2}$ δεν μπορούν να αναπαρασταθούν με ακρίβεια από μία `float` μεταβλητή.

Αντί να ελέγχουμε αν το `x` και το `y` είναι ακριβώς ίσα, είναι ασφαλέστερο να χρησιμοποιούμε την ενσωματωμένη συνάρτηση `abs` για υπολογίσουμε την απόλυτη τιμή ή το μέγεθος της διαφοράς τους:

```
    if abs(y-x) < epsilon:
```

```
        break
```

Όπου `epsilon`, είναι μία τιμή όπως η `0.0000001` η οποία καθορίζει πόσο κοντά είναι αρκετά κοντά.

Άσκηση 7.2. *Ενθυλακώστε αυτό το βρόχο σε μία συνάρτηση με όνομα* `square_root` *η οποία θα παίρνει το* `a` *σαν παράμετρο, θα επιλέγει μία λογική τιμή του* `x` *και θα επιστρέφει μία εκτίμηση της τετραγωνικής ρίζας του* `a`.

7.6 Αλγόριθμοι

Η μέθοδος του Νιούτον αποτελεί ένα παράδειγμα ενός **αλγορίθμου**: μία μηχανική διαδικασία για την επίλυση μίας κατηγορίας προβλημάτων (σε αυτήν την περίπτωση, ο υπολογισμός της τετραγωνικής ρίζας ενός αριθμού).

Δεν είναι εύκολο να οριστεί ένας αλγόριθμος. Θα σας βοηθούσε εάν ξεκινούσατε με κάτι το οποίο δεν είναι αλγόριθμος. Όταν μάθατε να πολλαπλασιάζετε μονοψήφιους αριθμούς, τότε μάλλον απομνημονεύσατε και την προπαίδεια. Και έτσι, απομνημονεύσατε 100 συγκεκριμένες λύσεις. Αυτού του τύπου η γνώση δεν είναι αλγόριθμος.

Αν ήσασταν όμως ”τεμπέληδες”, τότε μάλλον κάνατε ζαβολιές μαθαίνοντας μερικά κόλπα. Για παράδειγμα, για να βρείτε το γινόμενο του n με το 9, μπορείτε να γράψετε $n - 1$ για το πρώτο ψηφίο και $10 - n$ για το δεύτερο. Αυτό το κόλπο είναι μία γενική λύση για τον πολλαπλασιασμό ενός οποιουδήποτε μονοψήφιου αριθμού με το 9. Και αυτό είναι ένας αλγόριθμος!

Ομοίως, οι τεχνικές που μάθατε για την πρόσθεση με κρατούμενο, την αφαίρεση με δανεισμό και τη διαίρεση είναι όλες αλγόριθμοι. Ένα από τα χαρακτηριστικά των αλγορίθμων είναι ότι δεν απαιτούν ευφυΐα για να εκτελεστούν. Είναι μηχανικές διαδικασίες στις οποίες κάθε βήμα απορρέει από το προηγούμενο με βάση ένα απλό σύνολο κανόνων.

Κατά τη γνώμη μου, είναι πολύ δυσάρεστο το γεγονός ότι οι άνθρωποι ξοδεύουν τόσο πολύ χρόνο στα σχολεία για να μάθουν να εκτελούν αλγόριθμους. Είναι μία διαδικασία που στην κυριολεξία δεν χρειάζεται καθόλου ευφυΐα.

Από την άλλη πλευρά, η διαδικασία του σχεδιασμού αλγορίθμων είναι μία ενδιαφέρουσα πνευματική πρόκληση η οποία αποτελεί βασικό κομμάτι αυτού που ονομάζουμε προγραμματισμό.

Μερικά από τα πράγματα τα οποία οι άνθρωποι κάνουν φυσικά, χωρίς δυσκολία ή συνειδητή σκέψη, είναι πολύ δύσκολο να εκφραστούν αλγοριθμικά. Ένα καλό παράδειγμα είναι η κατανόηση μίας φυσικής γλώσσας. Αυτό το κάνουμε όλοι, αλλά μέχρι στιγμής κανένας δεν ήταν σε θέση να εξηγήσει ”πως” το κάνουμε ή τουλάχιστον όχι με τη μορφή αλγόριθμου.

7.7 Αποσφαλμάτωση

Όσο θα γράφετε όλο και μεγαλύτερα προγράμματα, τόσο θα διαπιστώνετε ότι αφιερώνετε όλο και περισσότερο χρόνο για αποσφαλμάτωση. Γράφοντας περισσότερο κώδικα αυξάνονται οι πιθανότητες να κάνετε κάποιο λάθος μιας και υπάρχει περισσότερος χώρος για να κρυφτούν τα σφάλματα.

Ένας τρόπος για να μειώσετε το χρόνο αποσφαλμάτωσης είναι η ”αποσφαλμάτωση με διχοτόμηση”. Για παράδειγμα, αν υπάρχουν 100 γραμμές στο πρόγραμμά σας και εσείς τις ελέγχατε όλες μία μία, τότε θα χρειαζόσασταν 100 βήματα.

Αντ' αυτού, δοκιμάστε να σπάσετε το πρόγραμμα στη μέση. Ψάξτε στη μέση του προγράμματος, ή κάπου κοντά σε αυτήν, για μία ενδιάμεση τιμή την οποία μπορείτε να ελέγξετε. Προσθέστε μία δήλωση `print` (ή κάτι άλλο το οποίο θα έχει επαληθεύσιμη επίδραση) και τρέξτε το πρόγραμμα.

Εάν αυτό το σημείο ελέγχου στη μέση είναι εσφαλμένο, τότε πρέπει να υπάρχει λάθος στο πρώτο μισό του προγράμματος. Αν είναι σωστό, τότε το πρόβλημα είναι στο δεύτερο μισό.

Κάθε φορά που εκτελείτε έναν τέτοιο έλεγχο, μειώνετε κατά το ήμισυ τις γραμμές που πρέπει να ψάξετε για να βρείτε το λάθος. Θεωρητικά, μετά από έξι βήματα (τα οποία είναι λιγότερα από 100) θα είστε στη μία ή δύο γραμμές του κώδικα που βρίσκετε το λάθος.

Στην πράξη, δεν είναι πάντα ξεκάθαρο ποια είναι η "μέση του προγράμματος" και ίσως να μην είναι και εφικτό να τη βρούμε. Δεν έχει νόημα να μετράμε γραμμές για να βρούμε ακριβώς το σημείο της μέσης. Αντ' αυτού, ψάξτε για περιοχές στο κώδικα που μπορεί να υπάρχουν λάθη και σημεία που είναι εύκολο να βάλετε έναν έλεγχο. Και στη συνέχεια διαλέξτε ένα σημείο όπου οι πιθανότητες, για το σφάλμα να είναι πριν ή μετά του σημείου ελέγχου, είναι περίπου ίδιες.

7.8 Ορολογία

πολλαπλή εκχώρηση: Το να γίνονται περισσότερες από μία εκχωρήσεις τιμής στην ίδια μεταβλητή κατά την εκτέλεση του προγράμματος.

ενημέρωση: Μία εκχώρηση όπου η νέα τιμή της μεταβλητής εξαρτάται από την παλιά.

αρχικοποίηση: Μία εκχώρηση η οποία δίνει αρχική τιμή σε μία μεταβλητή η οποία θα ενημερωθεί.

προσαύξηση: Μία ενημέρωση η οποία αυξάνει την τιμή μίας μεταβλητής (συνήθως κατά ένα).

μείωση: Μία ενημέρωση η οποία μειώνει την τιμή μίας μεταβλητής.

επανάληψη: Επαναλαμβανόμενη εκτέλεση ενός συνόλου δηλώσεων χρησιμοποιώντας είτε μία αναδρομική συνάρτηση είτε έναν βρόχο.

ατέρμων βρόχος: Ένας βρόχος στον οποίο η συνθήκη τέλους δεν ικανοποιείται ποτέ.

7.9 Ασκήσεις

Άσκηση 7.3. *Για να ελέγξετε τον αλγόριθμο υπολογισμού τετραγωνικής ρίζας αυτού του κεφαλαίου, μπορείτε να τον συγκρίνετε με τη* `math.sqrt`*. Γράψτε μία συνάρτηση με όνομα* `test_square_root` *η οποία θα εμφανίζει στην οθόνη έναν τέτοιο πίνακα :*

```
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
```

Η πρώτη στήλη είναι ένας αριθμός a, η δεύτερη είναι η τετραγωνική ρίζα του a υπολογισμένη με τη συνάρτηση της Ενότητα 7.5, η τρίτη στήλη είναι η τετραγωνική ρίζα υπολογισμένη από τη `math.sqrt`*, και η τέταρτη στήλη είναι η απόλυτη τιμή της διαφοράς μεταξύ των δύο υπολογισθέντων τιμών.*

Άσκηση 7.4. *Η ενσωματωμένη συνάρτηση* `eval` *παίρνει μία συμβολοσειρά σαν όρισμα και την υπολογίζει χρησιμοποιώντας τον διερμηνέα της Python. Για παράδειγμα:*

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<type 'float'>
```

Γράψτε μία συνάρτηση με όνομα `eval_loop` *η οποία προτρέπει επαναληπτικά τον χρήστη, παίρνει την είσοδο και την υπολογίζει χρησιμοποιώντας την* `eval` *και μετά εμφανίζει το αποτέλεσμα.*

Θα πρέπει να συνεχίζει μέχρι ο χρήστης να πληκτρολογήσει `'done'` *και στη συνέχεια να επιστρέφει την τελευταία έκφραση που υπολόγισε.*

Άσκηση 7.5. *Ο μαθηματικός Σρινιβάσα Ραμανούτζαν βρήκε μία άπειρη σειρά η οποία μπορεί να χρησιμοποιηθεί για να παράγει μία αριθμητική προσέγγιση του π:*

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Γράψτε μία συνάρτηση με όνομα `estimate_pi` *η οποία θα χρησιμοποιεί αυτόν τον τύπο για να υπολογίσει και να επιστρέψει μία εκτίμηση του π. Θα πρέπει να χρησιμοποιεί έναν βρόχο* `while` *για να υπολογίζει τους όρους της άθροισης μέχρι ο τελευταίος όρος να είναι μικρότερος του* `1e-15` *(ο οποίος είναι ο τρόπος συμβολισμού του 10^{-15} στην Python). Μπορείτε να ελέγξετε το αποτέλεσμα συγκρίνοντάς το με την* `math.pi`*.*

Λύση: `http://thinkpython.com/code/pi.py`*.*

Κεφάλαιο 8

Συμβολοσειρές

8.1 Μία συμβολοσειρά είναι μία ακολουθία

Μία συμβολοσειρά είναι μία ακολουθία χαρακτήρων. Μπορείτε να έχετε πρόσβαση στους χαρακτήρες (έναν τη φορά) με τους τελεστές αγκύλης:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

Η δεύτερη δήλωση επιλέγει το χαρακτήρα νούμερο 1 από τη μεταβλητή `fruit` και τον εκχωρεί στη μεταβλητή `letter`.

Η έκφραση μέσα στις αγκύλες ονομάζεται **δείκτης** (index). Ο δείκτης υποδεικνύει ποιον χαρακτήρα της ακολουθίας θέλετε (εξού και το όνομα).

Το αποτέλεσμα όμως μπορεί να μην είναι που θα περιμένατε:

```
>>> print letter
a
```

Για τους περισσότερους ανθρώπους, το πρώτο γράμμα της λέξης `'banana'` είναι το `b` και όχι το `a`. Αλλά για τους επιστήμονες των υπολογιστών, ο δείκτης είναι η απόκλιση από την αρχή της συμβολοσειράς και άρα η απόκλιση του πρώτου γράμματος είναι μηδέν.

```
>>> letter = fruit[0]
>>> print letter
b
```

Άρα το `b` είναι το 0ο γράμμα της λέξης `'banana'`, το `a` είναι το 1ο γράμμα και το `n` είναι το 2ο γράμμα.

Μπορείτε να χρησιμοποιήσετε οποιαδήποτε έκφραση, συμπεριλαμβανομένων μεταβλητών και τελεστών, σαν δείκτη, αλλά η τιμή του δείκτη πρέπει να είναι ακέραιος αριθμός. Αλλιώς θα πάρετε:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

8.2 Η δήλωση `len`

Η συνάρτηση `len` είναι μία ενσωματωμένη συνάρτηση η οποία επιστρέφει το πλήθος των χαρακτήρων μίας συμβολοσειράς:

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

Εάν θέλατε να πάρετε το τελευταίο γράμμα μίας συμβολοσειράς, ενδεχομένως να μπαίνατε στον πειρασμό να δοκιμάσετε κάτι τέτοιο:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

Το σφάλμα `IndexError` οφείλεται στο γεγονός ότι δεν υπάρχει κανένα γράμμα στη λέξη `'banana'` με δείκτη 6. Από τη στιγμή που αρχίζουμε να μετράμε από το μηδέν, τα έξι γράμματα αριθμούνται από το 0 ως το 5. Για να πάρετε τον τελευταίο χαρακτήρα θα πρέπει να αφαιρέσετε 1 από την `length`:

```
>>> last = fruit[length-1]
>>> print last
a
```

Εναλλακτικά, μπορείτε να χρησιμοποιήσετε αρνητικούς δείκτες, οι οποίοι μετρούν προς τα πίσω από το τέλος της συμβολοσειράς. Η έκφραση `fruit[-1]` επιστρέφει το τελευταίο γράμμα, η `fruit[-2]` το προτελευταίο και ούτω καθεξής.

8.3 Διάσχιση με `for`

Πολλοί υπολογισμοί περιλαμβάνουν επεξεργασία συμβολοσειράς χρησιμοποιώντας ένα χαρακτήρα τη φορά. Συνήθως ξεκινάνε από την αρχή επιλέγοντας και χρησιμοποιώντας κάθε χαρακτήρα με τη σειρά μέχρι το τέλος. Αυτό το πρότυπο επεξεργασίας ονομάζεται **διάσχιση** (traversal). Μπορούμε να γράψουμε μία διάσχιση χρησιμοποιώντας ένα βρόχο `while`:

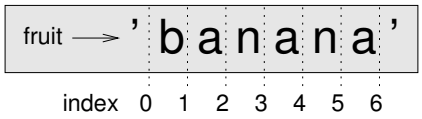
```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index = index + 1
```

Αυτός ο βρόχος διασχίζει τη συμβολοσειρά και εμφανίζει κάθε γράμμα της σε μία γραμμή. Η συνθήκη του βρόχου είναι η `index < len(fruit)`, έτσι ώστε όταν ο δείκτης `index` γίνει ίσος με το μήκος της συμβολοσειράς η συνθήκη γίνεται ψευδής και παύει να εκτελείται το σώμα του βρόχου. Ο τελευταίος χαρακτήρας που θα προσπελαστεί είναι αυτός με δείκτη `len(fruit)-1`, ο οποίος είναι και ο τελευταίος χαρακτήρας της συμβολοσειράς.

Άσκηση 8.1. *Γράψτε μία συνάρτηση η οποία θα παίρνει μία συμβολοσειρά σαν όρισμα και θα εμφανίζει όλα τα γράμματά της με τη σειρά, ένα σε κάθε γραμμή, ξεκινώντας από το τελευταίο.*

Ένας άλλος τρόπος για να γράψουμε μία διάσχιση είναι με έναν βρόχο `for`:

```
for char in fruit:
    print char
```



Σχήμα 8.1: Δείκτες τεμαχίων.

Σε κάθε επανάληψη του βρόχου, ο επόμενος χαρακτήρας της συμβολοσειράς εκχωρείται στη μεταβλητή `char`. Ο βρόχος εκτελείται μέχρις ότου να μην υπάρχουν άλλοι χαρακτήρες.

Το παρακάτω παράδειγμα δείχνει πως χρησιμοποιείται η συνένωση (πρόσθεση συμβολοσειρών) και ένας βρόχος `for` για να παραχθεί μία αλφαβητικά ταξινομημένη σειρά. Στο βιβλίο του Ρόμπερτ ΜακΚλάσκι *Make Way for Ducklings*, τα ονόματα από τα παπάκια είναι Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. Αυτός ο βρόχος έχει σαν έξοδο αυτά τα ονόματα με τη σειρά:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print letter + suffix
```

Η έξοδος είναι:

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

Αυτό φυσικά δεν είναι και πολύ σωστό αφού το “Ouack” και το “Quack” έχουν ορθογραφικό λάθος.

Άσκηση 8.2. *Τροποποιήστε το πρόγραμμα ούτως ώστε να διορθώσετε αυτό το λάθος.*

8.4 Τεμάχια συμβολοσειράς

Ένα τμήμα μίας συμβολοσειράς ονομάζεται **τεμάχιο** (slice). Η επιλογή ενός τεμαχίου είναι παρόμοια με την επιλογή ενός χαρακτήρα:

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:12]
Python
```

Ο τελεστής `[n:m]` επιστρέφει το τμήμα της συμβολοσειράς από το n οστό έως το m οστό χαρακτήρα, συμπεριλαμβανομένου του πρώτου αλλά όχι και του τελευταίου. Αυτή η συμπεριφορά είναι κάπως αντιφατική, αλλά ίσως σας βοηθήσει να φανταστείτε πως δείχνουν οι δείκτες ανάμεσα στους χαρακτήρες, όπως στην Εικόνα 8.1.

Εάν παραλείψετε τον πρώτο δείκτη (πριν την άνω κάτω τελεία), τότε το τεμάχιο θα ξεκινήσει από την αρχή της συμβολοσειράς. Αν παραλείψετε τον δεύτερο δείκτη, τότε το τεμάχιο θα φτάσει μέχρι

το τέλος της συμβολοσειράς:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

Αν ο πρώτος δείκτης είναι μεγαλύτερος ή ίσος με το δεύτερο το αποτέλεσμα θα είναι μία **κενή συμβολοσειρά** (empty string), η οποία αναπαριστάται από δύο μονά εισαγωγικά:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

Αν εξαιρέσουμε ότι μία κενή συμβολοσειρά δεν περιέχει κανένα χαρακτήρα και το μήκος της είναι 0, τότε δεν διαφέρει σε τίποτα άλλο από κάποια άλλη συμβολοσειρά.

Άσκηση 8.3. *Δοθέντος ότι η* `fruit` *είναι μία συμβολοσειρά, τι σημαίνει η έκφραση* `fruit[:]`*;*

8.5 Οι συμβολοσειρές είναι αμετάβλητες

Ίσως μπείτε στον πειρασμό να χρησιμοποιήσετε τον τελεστή `[]` στο αριστερό μέρος μίας εκχώρησης, με σκοπό να αλλάξετε κάποιο χαρακτήρα σε μία συμβολοσειρά. Για παράδειγμα:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

Το ”αντικείμενο” στην προκειμένη περίπτωση είναι η συμβολοσειρά και το ”στοιχείο” είναι ο χαρακτήρας που προσπαθήσαμε να εκχωρήσουμε. Για την ώρα, φανταστείτε ένα **αντικείμενο** (object) σαν μία τιμή, αλλά αργότερα θα βελτιώσουμε αυτόν τον ορισμό. Ένα **στοιχείο** (item) είναι μία από τις τιμές μέσα σε μία ακολουθία.

Το λάθος οφείλεται στο γεγονός ότι οι συμβολοσειρές είναι **αμετάβλητες** (immutable), το οποίο σημαίνει ότι δεν μπορείτε να αλλάξετε μία υπάρχουσα συμβολοσειρά. Το καλύτερο που μπορείτε να κάνετε είναι να δημιουργήσετε μία νέα συμβολοσειρά η οποία θα είναι μία παραλλαγή της αρχικής:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

Αυτό το παράδειγμα συνενώνει ένα νέο πρώτο γράμμα με ένα τεμάχιο της `greeting`, χωρίς να επηρεάζει καθόλου την αρχική συμβολοσειρά.

8.6 Αναζήτηση

Τι κάνει η ακόλουθη συνάρτηση;

```
def find(word, letter):
    index = 0
    while index < len(word):
```

```
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

Κατά μία έννοια, η `find` είναι το αντίθετο του τελεστή `[]`. Αντί να παίρνει ένα δείκτη και να εξάγει τον αντίστοιχο χαρακτήρα, παίρνει έναν χαρακτήρα και βρίσκει το δείκτη στον οποίο εμφανίζεται αυτός ο χαρακτήρας. Αν ο χαρακτήρας δεν βρεθεί, τότε η συνάρτηση επιστρέφει `-1`.

Αυτό είναι το πρώτο παράδειγμα που βλέπουμε με μία δήλωση `return` μέσα σε ένα βρόχο. Αν η συνθήκη `word[index] == letter` γίνει αληθής τότε η συνάρτηση "σπάει" (βγαίνει από το βρόχο) και επιστρέφει αμέσως.

Αν ο χαρακτήρας δε βρεθεί μέσα στη συμβολοσειρά, τότε το πρόγραμμα βγαίνει ομαλά από το βρόχο και επιστρέφει `-1`.

Αυτό το πρότυπο υπολογισμού, διάσχιση μιας συμβολοσειράς και επιστροφή όταν βρούμε αυτό που ψάχνουμε, ονομάζεται **αναζήτηση** (search).

Άσκηση 8.4. *Τροποποιήστε τη* `find` *έτσι ώστε να έχει σαν μία τρίτη παράμετρο το δείκτη από όπου πρέπει να ξεκινήσει να ψάχνει μέσα στη* `word`*.*

8.7 Επανάληψη και καταμέτρηση

Το ακόλουθο πρόγραμμα καταμετράει πόσες φορές εμφανίζεται το γράμμα `a` μέσα σε μία συμβολοσειρά:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print count
```

Αυτό το πρόγραμμα επιδεικνύει ένα άλλο πρότυπο υπολογισμού που ονομάζεται **μετρητής** (counter). Η μεταβλητή `count` αρχικοποιείται στο 0 και προσαυξάνεται κάθε φορά που βρίσκουμε ένα `a`. Όταν ο βρόχος τερματίσει, η `count` περιέχει το αποτέλεσμα, το πλήθος δηλαδή των `a`.

Άσκηση 8.5. *Ενθυλακώστε αυτόν τον κώδικα μέσα σε μία συνάρτηση με όνομα* `count` *και γενικεύστε την ούτως ώστε να δέχεται τη συμβολοσειρά και το γράμμα σαν ορίσματα.*

Άσκηση 8.6. *Ξαναγράψτε αυτή τη συνάρτηση έτσι ώστε αντί να διασχίζει τη συμβολοσειρά, να χρησιμοποιεί την τριών-παραμέτρων έκδοση της* `find` *από την προηγούμενη ενότητα.*

8.8 Μέθοδοι συμβολοσειρών

Μία **μέθοδος** (method) είναι παρόμοια με μία συνάρτηση, παίρνει ορίσματα και επιστρέφει μία τιμή, αλλά έχει διαφορετική σύνταξη. Για παράδειγμα, η μέθοδος `upper` παίρνει μία συμβολοσειρά και επιστρέφει μία νέα συμβολοσειρά με όλα τα γράμματα κεφαλαία:

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

```
>>> print new_word
BANANA
```

Αντί για τη σύνταξη συνάρτησης `upper(word)`, χρησιμοποιούμε τη σύνταξη μεθόδου `word.upper()`.

Αυτός ο τρόπος συμβολισμού με τελεία προσδιορίζει το όνομα της μεθόδου (`upper`) και το όνομα της συμβολοσειράς (`word`) στην οποία θα εφαρμοστεί η μέθοδος. Οι κενές παρενθέσεις υποδηλώνουν ότι η μέθοδος δεν παίρνει κανένα όρισμα.

Μία κλήση μεθόδου ονομάζεται **επίκληση** (invokation). Σε αυτήν την περίπτωση, θα λέγαμε ότι επικαλούμαστε την `upper` στη `word`.

Υπάρχει επίσης και μία μέθοδος συμβολοσειρών με όνομα `find` η οποία είναι εξαιρετικά όμοια με την συνάρτηση που γράψαμε:

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

Σε αυτό το παράδειγμα, επικαλούμαστε τη `find` στη `word` και περνάμε το γράμμα που ψάχνουμε σαν παράμετρο.

Στην πραγματικότητα, η μέθοδος `find` είναι πιο γενική από την δική μας συνάρτηση γιατί εκτός από χαρακτήρες μπορεί να βρει και "υποσυμβολοσειρές" (substrings):

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

Μπορεί να πάρει σαν δεύτερο όρισμα το δείκτη από όπου θα πρέπει να ξεκινήσει:

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

Και σαν τρίτο όρισμα το δείκτη όπου πρέπει να σταματήσει:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

Αυτή η αναζήτηση αποτυγχάνει επειδή το `b` δεν εμφανίζεται πουθενά μέσα στο εύρος των δεικτών από `1` έως `2` (μη συμπεριλαμβανομένου του `2`).

Άσκηση 8.7. *Υπάρχει μία μέθοδος συμβολοσειρών με όνομα* `count` *η οποία είναι παρόμοια με τη συνάρτηση της προηγούμενης άσκησης. Διαβάστε στην τεκμηρίωση αυτής της μεθόδου και γράψτε μία επίκληση η οποία θα μετράει το πλήθος των* `a` *στην λέξη* `'banana'`.

Άσκηση 8.8. *Διαβάστε την τεκμηρίωση των μεθόδων συμβολοσειρών στην διεύθυνση* `http://docs.python.org/2/library/stdtypes.html#string-methods`. *Ίσως θελήσετε να πειραματιστείτε με κάποιες από αυτές για να βεβαιωθείτε ότι καταλάβατε πως δουλεύουν. Η* `strip` *και η* `replace` *είναι ιδιαίτερα χρήσιμες.*

Η τεκμηρίωση χρησιμοποιεί μία σύνταξη η οποία να σας δυσκολεύει. Για παράδειγμα, στην `find(sub[, start[, end]])`*, οι αγκύλες υποδηλώνουν τα προαιρετικά ορίσματα. Άρα το* `sub` *είναι απαραίτητο αλλά το* `start` *είναι προαιρετικό, και αν συμπεριλάβετε το* `start` *τότε το* `end` *είναι προαιρετικό.*

8.9 Τελεστής in

Το in είναι ένας λογικός τελεστής (boolean) ο οποίος παίρνει δύο συμβολοσειρές και επιστρέφει True αν η πρώτη εμφανίζεται ως υποσυμβολοσειρά μέσα στη δεύτερη:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

Για παράδειγμα, η ακόλουθη συνάρτηση εμφανίζει όλα τα γράμματα της word1 τα οποία υπάρχουν και στη word2:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print letter
```

Πολλές φορές, ένα πρόγραμμα σε Python με σωστά επιλεγμένα ονόματα μεταβλητών διαβάζεται όπως ένα κείμενο στα αγγλικά. Αυτός ο βρόχος θα μπορούσε να διαβαστεί: “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

Στη συνέχεια βλέπετε τι θα παίρνατε αν συγκρίνατε μήλα (apples) με πορτοκάλια (oranges):

```
>>> in_both('apples', 'oranges')
a
e
s
```

8.10 Σύγκριση συμβολοσειρών

Οι σχεσιακοί τελεστές μπορούν να εφαρμοστούν και στις συμβολοσειρές. Για παράδειγμα, εάν θέλαμε να ελέγξουμε αν δύο συμβολοσειρές είναι ίδιες τότε θα γράφαμε:

```
if word == 'banana':
    print 'All right, bananas.'
```

Κάποιοι άλλοι σχεσιακοί τελεστές μας βοηθάνε να βάζουμε τις λέξεις σε αλφαβητική σειρά:

```
if word < 'banana':
    print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word,' + word + ', comes after banana.'
else:
    print 'All right, bananas.'
```

Η Python δεν μπορεί να χειριστεί τα κεφαλαία και τα πεζά με τον ίδιο τρόπο που το κάνουμε οι άνθρωποι. Όλα τα κεφαλαία γράμματα προηγούνται όλων των πεζών γραμμάτων, επομένως:

```
Your word, Pineapple, comes before banana.
```

Ο πιο συνηθισμένος τρόπος χειρισμού αυτού του προβλήματος είναι να μετατρέπουμε τις συμβολοσειρές σε μία προκαθορισμένη μορφή, όπως για παράδειγμα όλα κεφαλαία, πριν εκτελέσουμε τη σύγκριση. Κρατήστε το αυτό κατά νου σε περίπτωση που χρειαστεί να υπερασπιστείτε τον εαυτό σας ενάντια σε έναν άνδρα οπλισμένο με έναν Ανανά (Pineapple) (αμερικάνικη στρατιωτική αργκό όπου ο ανανάς σημαίνει χειροβομβίδα).

8.11 Αποσφαλμάτωση

Όταν χρησιμοποιείτε δείκτες για να διασχίσετε τις τιμές σε μία ακολουθία, είναι λίγο δύσκολο να πάρετε σωστά την αρχή και το τέλος της διάσχισης. Ακολουθεί μία συνάρτηση η οποία υποτίθεται ότι συγκρίνει δύο λέξεις και επιστρέφει `True` αν η μία είναι η αντίστροφη της άλλης, αλλά έχει δύο λάθη:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

Η πρώτη δήλωση `if` ελέγχει εάν οι λέξεις έχουν το ίδιο μήκος. Αν όχι, τότε μπορούμε να επιστρέψουμε `False` αμέσως ή αλλιώς να θεωρήσουμε ότι οι λέξεις έχουν το ίδιο μήκος για υπόλοιπο της συνάρτησης. Αυτό είναι ένα παράδειγμα πρότυπου φύλακα της Ενότητας 6.8.

Το `i` και το `j` είναι δείκτες. Ο πρώτος διασχίζει τη `word1` προς τα μπρος ενώ ο δεύτερος διασχίζει τη `word2` προς τα πίσω. Αν βρούμε δύο γράμματα τα οποία δεν ταιριάζουν τότε μπορούμε να επιστρέψουμε `False` αμέσως. Αν τελειώσει ολόκληρη η επανάληψη και όλα τα γράμματα ταιριάζουν μπορούμε να επιστρέψουμε `True`.

Αν δοκιμάσουμε αυτήν τη συνάρτηση με τις λέξεις ”pots” και ”stop”, τότε θα περιμέναμε η επιστρεφόμενη τιμή να είναι `True`, αλλά αντ’ αυτού παίρνουμε ένα σφάλμα (IndexError):

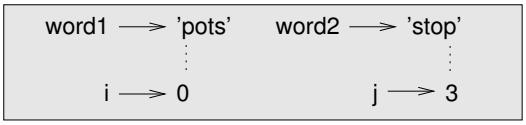
```
>>> is_reverse('pots', 'stop')
...
  File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

Η πρώτη μου κίνηση για αποσφαλμάτωση είναι να εμφανίσω τις τιμές των δεικτών ακριβώς πριν από τη γραμμή που εμφανίζεται το σφάλμα.

```
    while j > 0:
        print i, j        # print here

        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

Αν ξανατρέξω τώρα το πρόγραμμα θα πάρω περισσότερες πληροφορίες:

```
>>> is_reverse('pots', 'stop')
0 4
...
```



Σχήμα 8.2: Διάγραμμα κατάστασης.

```
IndexError: string index out of range
```

Στην πρώτη επανάληψη του βρόχου, η τιμή του `j` είναι 4, η οποία είναι εκτός εύρους για τη συμβολοσειρά `'pots'`. Ο δείκτης για τον τελευταίο χαρακτήρα είναι 3 και άρα η αρχική τιμή για το `j` θα πρέπει να είναι `len(word2)-1`.

Αν φτιάξω αυτό το σφάλμα και τρέξω ξανά το πρόγραμμα τότε παίρνω:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

Αυτή τη φορά παίρνουμε τη σωστή απάντηση, αλλά φαίνεται σαν να εκτελείτε μόνο τρεις φορές η επανάληψη, το οποίο είναι ύποπτο. Για να έχουμε μια καλύτερη ιδέα του τι συμβαίνει, είναι χρήσιμο να σχεδιάσουμε ένα διάγραμμα κατάστασης. Κατά τη διάρκεια της πρώτης επανάληψης, το πλαίσιο για την `is_reverse` εμφανίζεται στο Σχήμα 8.2.

Τοποθέτησα τις μεταβλητές μέσα στο πλαίσιο και πρόσθεσα διακεκομμένες λίγο αυθαίρετα για να δείξω ότι οι τιμές του `i` και του `j` δείχνουν χαρακτήρες στην `word1` και στην `word2`.

Άσκηση 8.9. *Ξεκινώντας με αυτό το διάγραμμα, εκτελέστε το πρόγραμμα στο χαρτί αλλάζοντας τις τιμές των* `i` *και* `j` *σε κάθε επανάληψη. Βρείτε και διορθώστε το δεύτερο λάθος σε αυτήν τη συνάρτηση.*

8.12 Ορολογία

αντικείμενο: Κάτι στο οποίο μπορεί να αναφέρεται μία μεταβλητή. Για την ώρα, μπορείτε να χρησιμοποιείτε τους όρους ”αντικείμενο” και ”τιμή” εναλλάξ.

ακολουθία: Ένα διατεταγμένο σύνολο τιμών στο οποίο κάθε τιμή προσδιορίζεται από έναν ακέραιο δείκτη.

στοιχείο: Μία από τις τιμές μίας ακολουθίας.

δείκτης: Μία ακέραια τιμή η οποία χρησιμοποιείται για να επιλέξουμε ένα στοιχείο μίας ακολουθίας, όπως είναι κάποιος χαρακτήρας μιας συμβολοσειράς.

τεμάχιο: Ένα τμήμα μίας συμβολοσειράς ορισμένο από ένα εύρος δεικτών.

κενή συμβολοσειρά: Μία συμβολοσειρά χωρίς χαρακτήρες και με μήκος 0, η οποία αναπαριστάται από δυο κενά εισαγωγικά.

αμετάβλητη: Η ιδιότητα μίας ακολουθίας της οποίας τα στοιχεία δεν μπορούν να αλλάξουν τιμές.

διάσχιση: Η διαδοχική προσπέλαση των στοιχείων μίας συμβολοσειράς με σκοπό την εκτέλεση κάποιας πράξης στο καθένα.

αναζήτηση: Ένα είδος διάσχισης η οποία σταματάει όταν βρει αυτό που ψάχνει.

μετρητής: Μία μεταβλητή η οποία χρησιμοποιείται για να μετρήσει κάτι, συνήθως αρχικοποιείται στο μηδέν και μετά προσαυξάνεται.

μέθοδος: Μία συνάρτηση η οποία συνδέεται με ένα αντικείμενο και καλείται χρησιμοποιώντας τον συμβολισμό με τελεία.

επίκληση: Μία δήλωση η οποία καλεί μία μέθοδο.

8.13 Ασκήσεις

Άσκηση 8.10. *Ένα τεμάχιο συμβολοσειράς μπορεί να πάρει ένα τρίτο όρισμα το οποίο καθορίζει το "μέγεθος βήματος", το οποίο είναι ο αριθμός των κενών μεταξύ των διαδοχικών χαρακτήρων. Ένα μέγεθος βήματος 2 σημαίνει κάθε δεύτερο χαρακτήρα, 3 σημαίνει κάθε τρίτο χαρακτήρα κτλ.*

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

Ένα μέγεθος βήματος ίσο με -1 διασχίζει τη λέξη προς τα πίσω, δηλαδή το τεμάχιο `[::-1]` *παράγει μία ανεστραμμένη συμβολοσειρά.*

Χρησιμοποιήστε αυτήν την ιδιότητα για να γράψετε μία έκδοση μίας γραμμής της `is_palindrome` *της Άσκησης 6.6.*

Άσκηση 8.11. *Οι ακόλουθες συναρτήσεις προορίζονται για να ελέγχουν εάν μία συμβολοσειρά περιέχει πεζά γράμματα αλλά κάποιες από αυτές είναι λάθος.*

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
```

```
    for c in s:
        if not c.islower():
            return False
    return True
```

Περιγράψτε τι κάνει πραγματικά η κάθε συνάρτηση, θεωρώντας ότι η παράμετρος είναι μία συμβολοσειρά.

Άσκηση 8.12. *Η ROT13 είναι μία αδύνατη μορφή κρυπτογράφησης η οποία βασίζεται στην "περιστροφή" (rotating) κάθε γράμματος μίας λέξης κατά 13 θέσεις. Περιστροφή ενός γράμματος σημαίνει μετατόπιση μέσα στο αλφάβητο, επιστρέφοντας μέχρι και στην αρχή εάν είναι απαραίτητο. Έτσι, το 'A' μετατοπισμένο κατά 3 γίνεται 'D' και το 'Z' μετατοπισμένο κατά 1 γίνεται 'A' στο αγγλικό αλφάβητο.*

Γράψτε μία συνάρτηση με όνομα `rotate_word` *η οποία θα παίρνει μία συμβολοσειρά και έναν ακέραιο σαν παραμέτρους και θα επιστρέφει μία νέα συμβολοσειρά η οποία θα περιέχει τα γράμματα της αρχικής "περιστραμμένα" με βάση το δοθέν ποσό.*

Για παράδειγμα, η λέξη "cheer" περιστραμμένη κατά 7 γίνεται "jolly" και η "melon" περιστραμμένη κατά -10 γίνεται "cubed".

Ενδεχομένως να σας φανούν χρήσιμες οι ενσωματωμένες συναρτήσεις `ord`*, η οποία μετατρέπει ένα χαρακτήρα σε έναν μαθηματικό κώδικα, και* `chr`*, η οποία μετατρέπει αριθμητικούς κώδικες σε χαρακτήρες.*

Δυνητικά προσβλητικά αστεία στο διαδίκτυο κρυπτογραφούνται μερικές φορές με ROT13. Αν δεν προσβάλλεστε εύκολα, βρείτε και αποκρυπτογραφήστε μερικά από αυτά. Λύση: `http://thinkpython.com/code/rotate.py`*.*

Κεφάλαιο 9

Μελέτη περίπτωσης: λογοπαίγνια

9.1 Διαβάζοντας λίστες λέξεων

Για τις ασκήσεις αυτού του κεφαλαίου θα χρειαστούμε μία λίστα αγγλικών λέξεων. Υπάρχουν πολλές τέτοιες λίστες διαθέσιμες στο διαδίκτυο αλλά η καταλληλότερη για το σκοπό μας είναι μία από τις λίστες που συλλέχθηκαν και προσφέρθηκαν ως "κοινό κτήμα" από τον Γκράντι Γουόρντ ως μέρος του έργου Μόντι (lexicon project Mody βλ. `http://wikipedia.org/wiki/Moby_Project`). Αυτή η λίστα αποτελείται από 113.809 "λέξεις σταυρόλεξων", λέξεις δηλαδή οι οποίες θεωρούνται έγκυρες στα σταυρόλεξα και σε άλλα παιχνίδια λέξεων. Στη συλλογή Μόντι, το όνομα του αρχείου είναι `113809of.fic`, αλλά εσείς μπορείτε να κατεβάσετε ένα αντίγραφο με το απλούστερο όνομα `words.txt` από το σύνδεσμο `http://thinkpython.com/code/words.txt`.

Αυτό το αρχείο είναι απλό κείμενο (plain text) και μπορείτε να το ανοίξετε με έναν κειμενογράφο, αλλά μπορείτε επίσης να το διαβάσετε και μέσω της Python. Η ενσωματωμένη συνάρτηση `open` παίρνει το όνομα ενός αρχείου σαν παράμετρο και επιστρέφει ένα αντικείμενο του αρχείου, το οποίο μπορείτε να χρησιμοποιήσετε για να διαβάσετε το αρχείο.

```
>>> fin = open('words.txt')
>>> print fin
<open file 'words.txt', mode 'r' at 0xb7f4b380>
```

Το `fin` είναι ένα από τα πιο συνηθισμένα ονόματα για αντικείμενα αρχείων τα οποία χρησιμοποιούνται ως είσοδος. Η λειτουργία `'r'` υποδεικνύει ότι αυτό το αρχείο είναι ανοιχτό για ανάγνωση (εν αντιθέσει με τη `'w'` η οποία είναι για γράψιμο).

Τα αντικείμενα αρχείων παρέχουν διάφορες μεθόδους για διάβασμα. Μεταξύ αυτών είναι και η `readline` η οποία διαβάζει χαρακτήρες από ένα αρχείο μέχρι να συναντήσει μία νέα γραμμή και επιστρέφει το αποτέλεσμα σαν συμβολοσειρά:

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

Η πρώτη λέξη σε αυτή τη συγκεκριμένη λίστα είναι η "aa", η οποία είναι ένα είδος λάβας. Η ακολουθία `\r\n` αναπαριστά δύο χαρακτήρες λευκού διαστήματος (μία επιστροφή φορέα και μία νέα γραμμή) οι οποίοι χωρίζουν αυτή τη λέξη από την επόμενη.

Το αντικείμενο του αρχείου καταγράφει τη θέση του μέσα στο αρχείο και έτσι αν ξανακαλέσετε την `readline` θα πάρετε την επόμενη λέξη:

```
>>> fin.readline()
```

```
'aah\r\n'
```

Η επόμενη λέξη είναι η "aah", η οποία θεωρείται απολύτως έγκυρη λέξη και για αυτό σταματήστε να με κοιτάτε έτσι. Αν σας ενοχλούν οι χαρακτήρες λευκού διαστήματος τότε μπορείτε να τους ξεφορτωθείτε με τη μέθοδο `strip`:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> print word
aahed
```

Επίσης, μπορείτε να χρησιμοποιήσετε ένα αντικείμενο αρχείου ως μέρος ενός βρόχου `for`. Αυτό το πρόγραμμα διαβάζει το `words.txt` και εμφανίζει όλες τις λέξεις μία μία:

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print word
```

Άσκηση 9.1. *Γράψτε ένα πρόγραμμα το οποίο θα διαβάζει το αρχείο* `words.txt` *και θα εμφανίζει μόνο τις λέξεις με περισσότερους από 20 χαρακτήρες (οι χαρακτήρες λευκού διαστήματος δεν μετράνε).*

9.2 Ασκήσεις

Οι λύσεις αυτών των ασκήσεων υπάρχουν στην επόμενη ενότητα. Θα πρέπει τουλάχιστον να προσπαθήσετε να τις λύσετε προτού συνεχίσετε στις λύσεις.

Άσκηση 9.2. *Το 1939 ο Ερνέστος Βίνσεντ Ράιτ δημοσίευσε μία νουβέλα 50.000 λέξεων με τίτλο* Gadsby *η οποία δεν περιέχει καθόλου το γράμμα "e". Αυτό δεν είναι και πολύ εύκολο αφού το γράμμα "e" είναι το πιο κοινό γράμμα στα Αγγλικά.*

Πράγματι, είναι δύσκολο να δομηθεί μία αυτεξούσια σκέψη χωρίς το πιο κοινό σύμβολο. Είναι πολύ αργό στην αρχή αλλά με σύνεση και πολλές ώρες εξάσκησης μπορείτε σταδιακά να αποκτήσετε ευχέρεια.

Εντάξει, σταματάω τώρα.

Γράψτε μία συνάρτηση με όνομα `has_no_e` *η οποία θα επιστρέφει* `True` *αν η δοθείσα λέξη δεν περιέχει το γράμμα "e".*

Τροποποιήστε το πρόγραμμα της προηγούμενης ενότητας για να εμφανίζει μόνο τις λέξεις που δεν έχουν καθόλου "e" και υπολογίστε το ποσοστό των λέξεων της λίστας οι οποίες δεν έχουν καθόλου "e".

Άσκηση 9.3. *Γράψτε μία συνάρτηση με όνομα* `avoids` *η οποία θα παίρνει ως πρώτο όρισμα μία λέξη και ως δεύτερο μία συμβολοσειρά από απαγορευμένα γράμματα και θα επιστρέφει* `True` *αν η λέξη δεν χρησιμοποιεί κανένα από τα αυτά τα γράμματα.*

Τροποποιήστε το πρόγραμμά σας ούτως ώστε να ζητάει από το χρήστη να εισάγει μία συμβολοσειρά απαγορευμένων γραμμάτων και μετά να εμφανίζει το πλήθος των λέξεων που δεν περιέχουν κανένα από αυτά. Μπορείτε να βρείτε έναν συνδυασμό από 5 απαγορευμένα γράμματα τα οποία αποκλείουν το μικρότερο πλήθος λέξεων;

Άσκηση 9.4. *Γράψτε μία συνάρτηση με όνομα* `uses_only` *η οποία θα παίρνει σαν πρώτο όρισμα μία λέξη και σαν δεύτερο όρισμα μία συμβολοσειρά γραμμάτων και θα επιστρέφει* `True` *αν η λέξη περιέχει μόνο τα γράμματα της ακολουθίας. Μπορείτε να κάνετε μία πρόταση χρησιμοποιώντας μόνο τα γράμματα* `acefhlo` *(εκτός της "Hoe alfalfa");*

Άσκηση 9.5. *Γράψτε μία συνάρτηση με όνομα* `uses_all` *η οποία θα παίρνει μία λέξη και μία συμβολοσειρά από απαιτούμενα γράμματα και θα επιστρέφει* `True` *αν η λέξη χρησιμοποιεί όλα τα απαιτούμενα γράμματα τουλάχιστον από μία φορά. Πόσες λέξεις υπάρχουν οι οποίες χρησιμοποιούν όλα τα φωνήεντα* `aeiou`*; Και πόσες οι οποίες χρησιμοποιούν όλα τα φωνήεντα* `aeiouy`*;*

Άσκηση 9.6. *Γράψτε μία συνάρτηση με όνομα* `is_abecedarian` *η οποία θα επιστρέφει* `True` *αν τα γράμματα σε μία λέξη εμφανίζονται σε αλφαβητική σειρά (τα διπλά γράμματα δεν πειράζουν). Πόσες "αλφαβητοποιημένες" λέξεις υπάρχουν;*

9.3 Αναζήτηση

Όλες οι ασκήσεις της προηγούμενης ενότητας έχουν κάτι κοινό μεταξύ τους. Μπορούν να λυθούν όλες με το πρότυπο αναζήτησης που είδαμε στην Ενότητα 8.6. Το πιο απλό παράδειγμα είναι:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

Ο βρόχος `for` διασχίζει τους χαρακτήρες της `word`. Αν βρούμε το γράμμα "e" τότε επιστρέφουμε αμέσως `False`, αλλιώς πρέπει να προχωρήσουμε στο επόμενο γράμμα. Εάν ο βρόχος τελειώσει κανονικά αυτό σημαίνει ότι δεν βρήκαμε κανένα "e" και άρα επιστρέφουμε `True`.

Η `avoids` είναι μία πιο γενικευμένη έκδοση της `has_no_e` αλλά έχει την ίδια δομή:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

Μπορούμε να επιστρέψουμε `False` αμέσως μόλις βρούμε ένα απαγορευμένο γράμμα. Αν φτάσουμε μέχρι το τέρμα του βρόχου τότε επιστρέφουμε `True`.

Η `uses_only` είναι σχεδόν ίδια, με τη μόνη διαφορά ότι η συνθήκη είναι ανεστραμμένη:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Αντί για μία λίστα απαγορευμένων γραμμάτων, έχουμε μία λίστα διαθέσιμων γραμμάτων. Αν βρούμε ένα γράμμα μέσα στη `word` το οποίο δεν υπάρχει στην `available` επιστρέφουμε `False`.

Η `uses_all` είναι παρόμοια αν εξαιρέσουμε ότι αντιστρέφουμε τον ρόλο της λέξης και της συμβολοσειράς των γραμμάτων:

```
def uses_all(word, required):
```

```
    for letter in required:
        if letter not in word:
            return False
    return True
```

Αντί ο βρόχος να διασχίζει τα γράμματα της `word`, διασχίζει τα γράμματα που απαιτούνται. Αν κάποιο από τα απαιτούμενα γράμματα δεν εμφανιστεί μέσα στη λέξη τότε επιστρέφουμε `False`.

Αν σκεφτόσασταν πραγματικά σαν ένας επιστήμονας της πληροφορικής, τότε θα είχατε αναγνωρίσει ότι η `uses_all` είναι ένα στιγμιότυπο ενός ήδη λυμένου προβλήματος και θα την γράφατε έτσι:

```
def uses_all(word, required):
    return uses_only(required, word)
```

Αυτό είναι ένα παράδειγμα μεθόδου ανάπτυξης προγράμματος η οποία ονομάζεται **αναγνώριση προβλήματος** (problem recognition), αυτό σημαίνει ότι αναγνωρίζετε το το πρόβλημα πάνω στο οποίο εργάζεστε ως ένα στιγμιότυπο ενός προβλήματος το οποίο έχει λυθεί ήδη και εφαρμόζετε μία ήδη ανεπτυγμένη λύση.

9.4 Βρόχοι επανάληψης με δείκτες

Στην προηγούμενη ενότητα, έγραψα τις συναρτήσεις με βρόχους `for` γιατί χρειαζόμουν μόνο τους χαρακτήρες μέσα στις συμβολοσειρές, δεν ήταν ανάγκη να χρησιμοποιήσω δείκτες.

Στην `is_abecedarian` πρέπει να συγκρίνουμε γειτονικά γράμματα, το οποίο είναι λίγο δύσκολο να υλοποιηθεί με έναν βρόχο `for`:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

Εναλλακτικά μπορούμε να χρησιμοποιήσουμε αναδρομή:

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

Μία άλλη επιλογή είναι να χρησιμοποιήσουμε ένα βρόχο `while`:

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

Ο βρόχος αρχίζει με `i=0` και τελειώνει όταν `i=len(word)-1`. Σε κάθε επανάληψη, συγκρίνει το iοστό χαρακτήρα (τον οποίο μπορείτε να σκέφτεστε σαν τον τρέχον χαρακτήρα) με τον $i + 1$οστό (σκεφτείτε τον σαν τον επόμενο).

Αν ο επόμενος χαρακτήρας είναι μικρότερος (αλφαβητικά προηγούμενος) από τον τρέχον, τότε έχουμε ανακαλύψει μία ρήξη στη αλφαβητική φορά και επιστρέφουμε `False`.

Αν ο βρόχος τερματίσει χωρίς να βρούμε κανένα σφάλμα τότε η λέξη έχει περάσει τον έλεγχο. Για να πείσετε τον εαυτό σας ότι ο βρόχος τερματίζει σωστά, σκεφτείτε ένα παράδειγμα όπως το `'flossy'`. Το μήκος της λέξης είναι 6, άρα την τελευταία φορά που τρέχει ο βρόχος το `i` είναι 4, το οποίο είναι ο δείκτης του προτελευταίου χαρακτήρα, δηλαδή αυτού που θέλουμε.

Ακολουθεί μία έκδοση της `is_palindrome` (βλ. Άσκηση 6.6) η οποία χρησιμοποιεί δύο δείκτες, ο ένας ξεκινάει από την αρχή και αυξάνεται και ο άλλος ξεκινάει από το τέλος και μειώνεται.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

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

    return True
```

Ή, αν είχατε παρατηρήσει ότι αυτό είναι ένα στιγμιότυπο ενός ήδη λυμένου προβλήματος, ίσως είχατε γράψει:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

Υποθέτοντας ότι έχετε κάνει την Άσκηση 8.9.

9.5 Αποσφαλμάτωση

Η δοκιμή και ο έλεγχος των προγραμμάτων είναι δύσκολες διαδικασίες γενικά. Οι συναρτήσεις σε αυτό το κεφάλαιο είναι σχετικά εύκολο να δοκιμαστούν γιατί μπορούμε να ελέγξουμε τα αποτελέσματα με το χέρι. Ακόμα κι έτσι όμως, η επιλογή ενός συνόλου λέξεων το οποίο θα καλύπτει όλα τα πιθανά σφάλματα είναι αν όχι αδύνατον, τότε πολύ δύσκολο.

Παίρνοντας σαν παράδειγμα την `has_no_e`, υπάρχουν δύο προφανείς περιπτώσεις για να ελέγξουμε: πρώτον οι λέξεις οι οποίες έχουν ένα "e" θα πρέπει να επιστρέφουν `False` και δεύτερον οι λέξεις που δεν έχουν κανένα "e" θα πρέπει να επιστρέφουν `True`. Λογικά, καμία από τις δύο δεν πρέπει να σας δυσκολεύει.

Σε κάθε περίπτωση υπάρχουν μερικές λιγότερο προφανείς υποπεριπτώσεις. Μεταξύ των λέξεων που έχουν ένα "e", θα πρέπει να ελέγξετε λέξεις με ένα "e" στην αρχή, στο τέλος και κάπου στη μέση. Θα πρέπει να ελέγξετε μεγάλες λέξεις, μικρές λέξεις, πολύ μικρές λέξεις ή ακόμα και κενές συμβολοσειρές. Η κενή συμβολοσειρά είναι ένα παράδειγμα **ειδικής περίπτωσης** (special case) η οποία ανήκει στις μη προφανείς περιπτώσεις που συχνά κρύβονται λάθη.

Εκτός από τις περιπτώσεις που δημιουργείτε εσείς, μπορείτε να δοκιμάσετε το πρόγραμμά σας με μία λίστα λέξεων όπως η `words.txt`. Ελέγχοντας προσεκτικά την έξοδο, ίσως μπορέσετε να βρείτε

σφάλματα αλλά να είστε προσεκτικοί γιατί μπορεί να βρείτε κάποιο σφάλμα (λέξεις οι οποίες δεν θα έπρεπε να συμπεριλαμβάνονται και υπάρχουν) αλλά όχι κάποιο άλλο (λέξεις οι οποίες θα έπρεπε να υπάρχουν και δεν συμπεριλαμβάνονται).

Γενικά, οι δοκιμές μπορούν να σας βοηθήσουν να βρείτε σφάλματα αλλά δεν είναι εύκολο να δημιουργήσετε ένα καλό σύνολο περιπτώσεων προς δοκιμή. Ακόμα και αν το καταφέρετε δεν μπορείτε να είστε σίγουροι ότι το πρόγραμμά σας είναι ολόσωστο.

Σύμφωνα με ένα θρυλικό επιστήμονα της πληροφορικής:

> Η δοκιμή των προγραμμάτων μπορεί να χρησιμοποιηθεί για να αναδείξει την παρουσία σφαλμάτων, αλλά ποτέ για να δείξει την απουσία τους!
>
> — Edsger W. Dijkstra

9.6 Ορολογία

αντικείμενο αρχείου: Μία τιμή η οποία αναπαριστά ένα ανοιχτό αρχείο.

αναγνώριση προβλήματος: Ένας τρόπος επίλυσης ενός προβλήματος εκφράζοντάς το ως ένα στιγμιότυπο ενός ήδη λυμένου προβλήματος.

ειδική περίπτωση: Μία άτυπη ή μη προφανής περίπτωση προς δοκιμή (και λιγότερο πιθανό να χειριστεί σωστά).

9.7 Ασκήσεις

Άσκηση 9.7. *Η ακόλουθη ερώτηση βασίζεται σε ένα γρίφο ο οποίος μεταδόθηκε στο ραδιοφωνικό σταθμό* Car Talk *(`http://www.cartalk.com/content/puzzlers`):*

> *Βρείτε μία αγγλική λέξη με τρεις συνεχόμενες δυάδες γραμμάτων. Για παράδειγμα, η λέξη committee: c-o-m-m-i-t-t-e-e θα ήταν σωστή αν εξαιρέσουμε το "i" στη μέση της και η λέξη Mississippi: M-i-s-s-i-s-s-i-p-p-i θα ήταν επίσης σωστή αν μπορούσαμε να βγάλουμε το δεύτερο και τρίτο "i". Ωστόσο, υπάρχει τουλάχιστον μία λέξη απ' όσο ξέρω η οποία έχει τρία συνεχόμενα ζευγάρια γραμμάτων. Ενδεχομένως να υπάρχουν ακόμα 500 λέξεις οι οποίες να ικανοποιούν αυτή τη συνθήκη αλλά εγώ μπορώ να σκεφτώ μόνο μία. Ποια είναι η λέξη;*

Γράψτε ένα πρόγραμμα για να βρείτε αυτή τη λέξη. Λύση : `http://thinkpython.com/code/cartalk1.py`.

Άσκηση 9.8. *Αυτός είναι ένας άλλος γρίφος από το* Car Talk *(`http://www.cartalk.com/content/puzzlers`):*

> *" Τις προάλλες, καθώς οδηγούσα στον αυτοκινητόδρομο, έτυχε να προσέξω το οδόμετρο μου, και όπως τα περισσότερα οδόμετρα έτσι και αυτό δείχνει με έξι ψηφία μόνο ολόκληρα μίλια. Έτσι, αν για παράδειγμα το αυτοκίνητό μου είχε 300.000 μίλια, θα έβλεπα 3-0-0-0-0-0.*
>
> *" Πρόσεξα λοιπόν κάτι πολύ ενδιαφέρον, τα τελευταία 4 ψηφία ήταν παλίνδρομα, διαβάζονται δηλαδή το ίδιο προς τα πίσω και προς τα μπρος. Για παράδειγμα, το 5-4-4-5 είναι ένα παλίνδρομο, έτσι το οδόμετρό μου μπορεί να έδειχνε 3-1-5-4-4-5.*

“Ένα μίλι αργότερα, τα τελευταία 5 νούμερα ήταν παλίνδρομα. Για παράδειγμα, μπορεί να ήταν 3-6-5-4-5-6. Ένα μίλι μετά από αυτό, οι τέσσερις μεσαίοι αριθμοί ήταν παλίνδρομοι και μαντέψτε τι έγινε μετά από ένα ακόμα μίλι. Και τα 6 ψηφία ήταν παλίνδρομα!

“Και η ερώτηση είναι η εξής, πόσα μίλια είχε το οδόμετρο όταν το κοίταξα πρώτη φορά;”

Γράψτε ένα πρόγραμμα σε Python το οποίο τεστάρει όλους τους εξαψήφιους αριθμούς και εμφανίζει όσους ικανοποιούν αυτές τις απαιτήσεις. Λύση: `http://thinkpython.com/code/cartalk2.py`.

Άσκηση 9.9. *Και αυτός είναι ένας ακόμα γρίφος του* Car Talk *τον οποίο μπορείτε να λύσετε με αναζήτηση (*`http://www.cartalk.com/content/puzzlers`*):*

“Πρόσφατα επισκέφτηκα τη μητέρα μου και καθώς συζητούσαμε, συνειδητοποιήσαμε ότι αν αντιστρέψουμε τα δύο ψηφία από τα οποία αποτελείται η ηλικία μου, τότε προκύπτει η δική της ηλικία. Για παράδειγμα, αν αυτή είναι 73 τότε εγώ είμαι 37. Αναρωτηθήκαμε πόσες φορές έχει συμβεί αυτό με την πάροδο των χρόνων αλλά αποπροσανατολιστήκαμε από άλλα θέματα και δεν καταλήξαμε σε κάποια απάντηση.

“Όταν γύρισα σπίτι, υπολόγισα ότι τα ψηφία των ηλικιών μας ήταν αντιστρέψιμα έξι φορές μέχρι τώρα. Υπολόγισα επίσης, ότι αν είμαστε τυχεροί, αυτό θα ξανασυμβεί μία ακόμα φορά μετά από αυτήν και αν είμαστε πολύ τυχεροί, θα μπορούσε να συμβεί 8 φορές συνολικά. Επομένως η ερώτηση είναι: πόσο χρονών είμαι τώρα;”

Γράψτε ένα πρόγραμμα σε Python το οποίο ψάχνει λύσεις για αυτόν το γρίφο. Υπόδειξη: ίσως σας φανεί χρήσιμη η μέθοδος `zfill`.

Λύση: `http://thinkpython.com/code/cartalk3.py`.

Κεφάλαιο 10

Λίστες

10.1 Η λίστα είναι μία ακολουθία

Όπως οι συμβολοσειρές, έτσι και οι **λίστες** (lists) είναι ακολουθίες τιμών, με τη διαφορά ότι σε μία συμβολοσειρά οι τιμές είναι χαρακτήρες ενώ σε μία λίστα μπορούν να είναι οποιουδήποτε τύπου. Οι τιμές μίας λίστας ονομάζονται **στοιχεία** (elements/items).

Υπάρχουν διάφοροι τρόποι για να δημιουργία μίας λίστας, αλλά ο πιο απλός είναι περικλείοντας τα στοιχεία μέσα σε αγκύλες (`[` και `]`):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

Το πρώτο παράδειγμα είναι μία λίστα η οποία αποτελείται από τέσσερις ακέραιους αριθμούς ενώ το δεύτερο είναι μία λίστα τριών συμβολοσειρών. Τα στοιχεία μίας λίστας μπορεί να είναι διαφορετικού τύπου. Η ακόλουθη λίστα περιέχει μία συμβολοσειρά, ένα δεκαδικό, έναν ακέραιο και μία άλλη λίστα:

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

Μία λίστα η οποία βρίσκεται μέσα σε μία άλλη λίστα ονομάζεται **εμφωλευμένη** (nested).

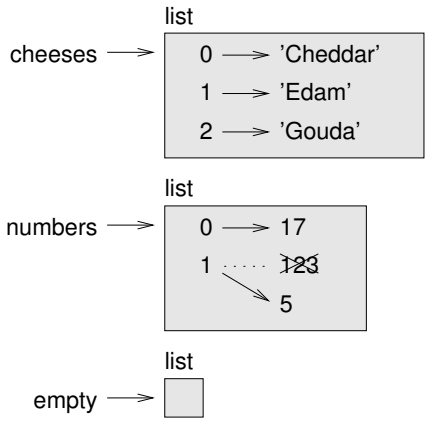
Μία λίστα η οποία δεν περιέχει κανένα στοιχείο ονομάζεται κενή λίστα. Η κενή δημιουργείται με κενές αγκύλες `[]`.

Όπως θα περιμένατε, μπορείτε να εκχωρήσετε τις τιμές μίας λίστας σε μεταβλητές:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

10.2 Οι λίστες είναι μεταβλητές

Η σύνταξη για την πρόσβαση στα στοιχεία μίας λίστας είναι η ίδια με τη σύνταξη για την πρόσβαση στους χαρακτήρες μίας συμβολοσειράς, χρησιμοποιώντας δηλαδή τον τελεστή αγκύλης. Θυμηθείτε ότι οι δείκτες ξεκινάνε από το 0:



Σχήμα 10.1: Διάγραμμα κατάστασης.

```
>>> print cheeses[0]
Cheddar
```

Σε αντίθεση με τις συμβολοσειρές, οι λίστες είναι μεταβλητές. Όταν ο τελεστής αγκύλης εμφανίζεται στο αριστερό μέλος μίας εκχώρησης τότε προσδιορίζει το στοιχείο της λίστας στο οποίο θα εκχωρηθεί το δεξιό μέλος.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

Το στοιχείο της `numbers` με δείκτη 1 το οποίο ήταν 123, τώρα είναι 5.

Μπορείτε να φανταστείτε μία λίστα σαν μία σχέση μεταξύ δεικτών και στοιχείων. Αυτή η σχέση ονομάζεται **αντιστοίχιση** (mapping). Κάθε δείκτης δηλαδή, αντιστοιχεί σε ένα από τα στοιχεία. Η Εικόνα 10.1 δείχνει το διάγραμμα κατάστασης για τις `cheeses`, `numbers` και `empty`:

Οι λίστες αναπαριστώνται από κουτιά με τη λέξη "list" εξωτερικά και τα στοιχεία της λίστας εντός. Η `cheeses` αναφέρεται σε μία λίστα τριών στοιχείων με δείκτες 0, 1 και 2. Η `numbers` περιέχει δύο στοιχεία εκ των οποίων το ένα άλλαξε τιμή (επανεκχωρήθηκε) από 123 σε 5. Η `empty` αναφέρεται σε μία λίστα χωρίς κανένα στοιχείο.

Οι δείκτες λιστών δουλεύουν με τον ίδιο τρόπο όπως και οι δείκτες συμβολοσειρών:

- Κάθε ακέραια έκφραση μπορεί να χρησιμοποιηθεί σαν δείκτης.
- Αν προσπαθήσετε να διαβάσετε ή να γράψετε ένα στοιχείο το οποίο δεν υπάρχει τότε θα πάρετε ένα `IndexError`.
- Αν ένας δείκτης έχει αρνητική τιμή τότε μετράει προς τα πίσω από το τέλος της λίστας.

Στις λίστες δουλεύει επίσης και ο τελεστής `in`:

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

10.3 Διασχίζοντας μία λίστα

Ο πιο συνηθισμένος τρόπος διάσχισης των στοιχείων μίας λίστας είναι με ένα βρόχο `for`. Η σύνταξη είναι ίδια με τη διάσχιση συμβολοσειρών:

```
for cheese in cheeses:
    print cheese
```

Αυτό δουλεύει μια χαρά αν θέλετε μόνο να διαβάσετε τα στοιχεία μίας λίστας. Αν όμως θέλετε να γράψετε ή να ενημερώσετε στοιχεία τότε χρειάζεστε δείκτες. Αυτό γίνεται, συνήθως, συνδυάζοντας τις συναρτήσεις `range` και `len`:

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

Αυτός ο βρόχος διασχίζει τη λίστα και ενημερώνει κάθε στοιχείο. Η `len` επιστρέφει το πλήθος των στοιχείων της λίστας και η `range` επιστρέφει μία λίστα δεικτών από το 0 έως το $n - 1$, όπου n το μήκος της λίστας. Σε κάθε επανάληψη η `i` παίρνει το δείκτη του επόμενου στοιχείου και χρησιμοποιείται από τη δήλωση εκχώρησης μέσα στο σώμα για να διαβάσει τη παλιά τιμή του στοιχείου και να εκχωρήσει την νέα.

Ένας βρόχος `for` επί μίας κενής λίστας δεν εκτελεί ποτέ το σώμα του:

```
for x in []:
    print 'This never happens.'
```

Παρόλο που μία λίστα μπορεί να περιέχει μία άλλη λίστα, η εμφωλευένη λίστα δεν παύει να είναι ένα στοιχείο από μόνη της. Άρα, το μήκος της ακόλουθης λίστας είναι τέσσερα:

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

10.4 Πράξεις με λίστες

Ο τελεστής + συνενώνει λίστες:

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

Παρομοίως, ο τελεστής * επαναλαμβάνει μία λίστα δοθέντος ενός αριθμού επανάληψης:

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

Το πρώτο παράδειγμα επαναλαμβάνει την `[0]` τέσσερις φορές και το δεύτερο επαναλαμβάνει τη λίστα `[1, 2, 3]` τρεις φορές.

10.5 Λίστες και τεμάχια

Ο τελεστής τεμαχισμού δουλεύει επίσης με τις λίστες:

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

Αν παραλείψετε τον πρώτο δείκτη τότε το τεμάχιο θα ξεκινήσει από την αρχή. Αν παραλείψετε το δεύτερο τότε το τεμάχιο θα φτάσει μέχρι το τέλος. Επομένως, αν παραλείψετε και τους δύο το τεμάχιο θα είναι ένα αντίγραφο ολόκληρης της λίστας.

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

Από τη στιγμή που οι λίστες είναι μεταβλητές, είναι συχνά χρήσιμο να κρατάτε ένα αντίγραφο πριν περιστρέψετε, ακρωτηριάσετε ή εφαρμόσετε κάποια συνάρτηση σε μία λίστα.

Ένας τελεστής τεμάχιο στο αριστερό μέλος μίας εκχώρησης μπορεί να ενημερώσει πολλαπλά στοιχεία:

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

10.6 Μέθοδοι λιστών

Η Python παρέχει διάφορες μεθόδους οι οποίες εφαρμόζονται πάνω σε λίστες. Για παράδειγμα, η `append` προσθέτει ένα νέο στοιχείο στο τέλος μίας λίστας:

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

Η `extend` παίρνει μία λίστα σαν όρισμα και προσαρτά όλα τα στοιχεία της:

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

Αυτό το παράδειγμα αφήνει ανεπηρέαστη την `t2`.

Η `sort` ταξινομεί τα στοιχεία της λίστας από το χαμηλότερο προς το υψηλότερο (αύξουσα σειρά):

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

Όλες οι μέθοδοι των λιστών είναι κενές (void), τροποποιούν δηλαδή τη λίστα στην οποία εφαρμόζονται και επιστρέφουν `None`. Αν γράψετε, κατά λάθος, `t = t.sort()` τότε θα απογοητευτείτε από το αποτέλεσμα.

10.7 Map, filter και reduce

Για να προσθέσετε τους αριθμούς μίας λίστας μπορείτε να χρησιμοποιήσετε ένα βρόχο σαν αυτόν:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

Η `total` αρχικοποιείται στο 0 και σε κάθε επανάληψη το `x` παίρνει ένα στοιχείο της λίστας. Ο τελεστής `+=` είναι ένας εύκολος τρόπος για να ενημερώσουμε μία μεταβλητή. Αυτή η **δήλωση εκχώρησης προσαύξησης** (augmented assignment statement):

```
    total += x
```

είναι ισοδύναμη με την:

```
    total = total + x
```

Όσο εκτελείται ο βρόχος, η `total` συσσωρεύει το άθροισμα των στοιχείων. Μία μεταβλητή που χρησιμοποιείται κατ' αυτόν τον τρόπο ονομάζεται **συσσωρευτής** (accumulator) ή **αθροιστής**.

Η άθροιση των στοιχείων μίας λίστας αποτελεί μία τόσο κοινή διαδικασία που η Python την παρέχει σαν ενσωματωμένη συνάρτηση, την `sum`:

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

Μία τέτοια διαδικασία, η οποία συνενώνει μία ακολουθία στοιχείων σε μία ενιαία τιμή, ονομάζεται πολλές φορές **μείωση** (reduce).

Άσκηση 10.1. *Γράψτε μία συνάρτηση με όνομα* `nested_sum` *η οποία θα παίρνει μία λίστα από άλλες εμφωλευμένες λίστες ακεραίων, θα αθροίζει τα στοιχεία από όλες τις εμφωλευμένες λίστες και θα επιστρέφει το άθροισμα.*

Μερικές φορές, θα θέλετε να διασχίσετε μία λίστα καθώς οικοδομείτε μία άλλη. Για παράδειγμα, η ακόλουθη συνάρτηση παίρνει μία λίστα συμβολοσειρών και επιστρέφει μία νέα λίστα η οποία περιέχει τις συμβολοσειρές κεφαλαιοποιημένες (με κεφαλαίο δηλαδή το πρώτο γράμμα):

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

Η `res` αρχικοποιείται με μία κενή λίστα και σε κάθε επανάληψη προσαρτούμε το επόμενο στοιχείο. Επομένως, η `res` είναι ένα ακόμα είδος συσσωρευτή.

Μερικές φορές, μία διαδικασία όπως η `capitalize_all` ονομάζεται **map** επειδή αντιστοιχίζει μία συνάρτηση (σε αυτήν την περίπτωση την μέθοδο `capitalize`) σε κάθε ένα από τα στοιχεία μίας

ακολουθίας.

Άσκηση 10.2. *Χρησιμοποιήστε την* `capitalize_all` *για να γράψετε μία συνάρτηση με όνομα* `capitalize_nested` *η οποία θα παίρνει μία λίστα από εμφωλευμένες λίστες συμβολοσειρών και θα επιστρέφει μία νέα λίστα με όλες τις συμβολοσειρές κεφαλαιοποιημένες.*

Μία ακόμη συνηθισμένη διαδικασία είναι η επιλογή κάποιων στοιχείων μίας λίστας και η επιστροφή μίας υπολίστας (sublist). Για παράδειγμα, η παρακάτω συνάρτηση παίρνει μία λίστα συμβολοσειρών και επιστρέφει μία λίστα η οποία περιέχει μόνο τις κεφαλαίες συμβολοσειρές (αυτές δηλαδή που έχουν όλα τα γράμματα κεφαλαία):

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

Η `isupper` είναι μία μέθοδος συμβολοσειρών η οποία επιστρέφει `True` αν η συμβολοσειρά περιέχει μόνο κεφαλαία γράμματα.

Μία διαδικασία όπως η `only_upper` ονομάζεται **φίλτρο** (filter) γιατί επιλέγει κάποια από τα στοιχεία και αγνοεί τα υπόλοιπα.

Οι πιο συνηθισμένες πράξεις στις λίστες μπορούν να εκφραστούν σαν ένας συνδυασμός από map, filter και reduce. Επειδή αυτές η διαδικασίες είναι τόσο κοινές, η Python παρέχει κάποια χαρακτηριστικά για να τις υποστηρίξει, συμπεριλαμβανομένων της ενσωματωμένης συνάρτησης `map` και ενός τελεστή που ονομάζεται "κατανόηση λίστας".

Άσκηση 10.3. *Γράψτε μία συνάρτηση η οποία θα παίρνει μία λίστα από αριθμούς και θα επιστρέφει το συσσωρευτικό άθροισμά τους, το οποίο θα είναι μία νέα λίστα όπου το* i*οστό στοιχείο θα είναι το άθροισμα των πρώτων* $i+1$ *στοιχείων της αρχικής λίστας. Για παράδειγμα, το άθροισμα των στοιχείων της* `[1, 2, 3]` *θα είναι* `[1, 3, 6]`.

10.8 Διαγραφή στοιχείων

Υπάρχουν διάφοροι τρόποι για να διαγράψετε στοιχεία από μία λίστα. Αν γνωρίζετε το δείκτη του στοιχείου που θέλετε τότε μπορείτε να χρησιμοποιήσετε την `pop`:

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

Η `pop` τροποποιεί τη λίστα και επιστρέφει το στοιχείο που αφαιρέθηκε. Αν δεν της δώσετε κάποιο δείκτη, διαγράφει και επιστρέφει το τελευταίο στοιχείο της λίστας.

Αν δεν χρειάζεστε τη διαγραμμένη τιμή, μπορείτε να χρησιμοποιήσετε τον τελεστή `del`:

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

Αν γνωρίζετε το στοιχείο που θέλετε να αφαιρέσετε (αλλά όχι το δείκτη), μπορείτε να χρησιμοποιήσετε την `remove`:

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

Η επιστρεφόμενη τιμή από τη `remove` είναι `None`.

Για να αφαιρέσετε περισσότερα από ένα στοιχεία, μπορείτε να χρησιμοποιήσετε την `del` με δείκτη ένα τεμάχιο:

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

Όπως συνήθως, το τεμάχιο επιλέγει όλα τα στοιχεία μέχρι, αλλά μη συμπεριλαμβανομένου, τον δεύτερο δείκτη.

Άσκηση 10.4. *Γράψτε μία συνάρτηση με όνομα* `middle` *η οποία θα παίρνει μία λίστα και θα επιστρέφει μία νέα λίστα η οποία θα περιέχει όλα τα στοιχεία της πρώτης, εκτός από το πρώτο και το τελευταίο. Έτσι, δοθείσας της* `middle([1,2,3,4])` *θα πρέπει να επιστρέψει* `[2,3]`.

Άσκηση 10.5. *Γράψτε μία συνάρτηση με όνομα* `chop` *η οποία θα παίρνει μία λίστα, θα την τροποποιεί αφαιρώντας το πρώτο και το τελευταίο στοιχείο και θα επιστρέφει* `None`.

10.9 Λίστες και συμβολοσειρές

Μία συμβολοσειρά είναι μία ακολουθία χαρακτήρων και μία λίστα είναι μία ακολουθία τιμών, αλλά μία λίστα χαρακτήρων δεν είναι το ίδιο με μία συμβολοσειρά. Για να μετατρέψετε μία συμβολοσειρά σε μία λίστα χαρακτήρων μπορείτε να χρησιμοποιήσετε την `list`:

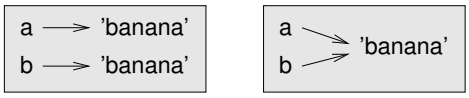
```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

Επειδή το `list` είναι το όνομα μίας ενσωματωμένης συνάρτησης, θα πρέπει να αποφεύγετε να το χρησιμοποιείτε σαν όνομα μεταβλητής. Εγώ αποφεύγω επίσης την χρήση του `l` επειδή μοιάζει πάρα πολύ με το `1`. Γι' αυτό χρησιμοποιώ το `t`.

Η συνάρτηση `list` σπάει μία συμβολοσειρά σε μεμονωμένα γράμματα. Αν θέλετε να σπάσετε μία συμβολοσειρά σε λέξεις, μπορείτε να χρησιμοποιήσετε τη μέθοδο `split`:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
```

Ένα προαιρετικό όρισμα με όνομα **διαχωριστής** (delimiter) καθορίζει ποιοι χαρακτήρες θα χρησιμοποιηθούν σαν όρια των λέξεων. Το ακόλουθο παράδειγμα χρησιμοποιεί για διαχωριστή μία παύλα:



Σχήμα 10.2: Διάγραμμα κατάστασης.

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

Η `join` είναι το αντίστροφο της `split`. Παίρνει μία λίστα συμβολοσειρών και συνενώνει τα στοιχεία. Αφού η `join` είναι μία μέθοδος συμβολοσειρών, θα πρέπει να την επικαλεστείτε στο διαχωριστή και να περάσετε τη λίστα σαν παράμετρο:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

Σε αυτήν την περίπτωση ο διαχωριστής είναι ο χαρακτήρας κενού διαστήματος και άρα η `join` βάζει ένα κενό ανάμεσα στις λέξεις. Για συνενώσετε συμβολοσειρές χωρίς κενά μπορείτε να χρησιμοποιήσετε σαν διαχωριστή την κενή συμβολοσειρά (`''`)

10.10 Αντικείμενα και τιμές

Αν εκτελέσουμε αυτές τις δηλώσεις εκχώρησης:

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

Ξέρουμε ότι τόσο η `a` όσο και η `b` αναφέρονται σε μία συμβολοσειρά αλλά δεν εάν αναφέρονται στην ίδια συμβολοσειρά. Υπάρχουν δύο πιθανές καταστάσεις οι οποίες φαίνονται στην Εικόνα 10.2.

Στη μία περίπτωση η `a` και `b` αναφέρονται σε δύο διαφορετικά αντικείμενα τα οποία έχουν την ίδια τιμή, ενώ στη δεύτερη περίπτωση αναφέρονται στο ίδιο αντικείμενο.

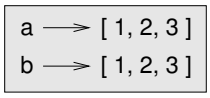
Για να ελέγξετε εάν δύο μεταβλητές αναφέρονται στο ίδιο αντικείμενο, μπορείτε να χρησιμοποιήσετε τον τελεστή `is`.

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

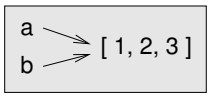
Σε αυτό το παράδειγμα η Python δημιούργησε μόνο ένα αντικείμενο συμβολοσειράς στο οποίο αναφέρεται και η `a` και η `b`.

Αν δημιουργείτε όμως δύο λίστες τότε θα έχετε δύο ξεχωριστά αντικείμενα:

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



Σχήμα 10.3: Διάγραμμα κατάστασης.



Σχήμα 10.4: Διάγραμμα κατάστασης.

Έτσι το διάγραμμα κατάστασης είναι όπως αυτό στην Εικόνα 10.3.

Σε αυτήν την περίπτωση, θα λέγαμε ότι οι δύο λίστες είναι **ισότιμες** επειδή έχουν τα ίδια στοιχεία, αλλά όχι **ταυτόσημες** αφού είναι δύο διαφορετικά αντικείμενα. Αν δύο αντικείμενα είναι ταυτόσημα τότε είναι και ισότιμα, αλλά αν είναι ισότιμα τότε δεν είναι κατ' ανάγκη και ταυτόσημα.

Μέχρι τώρα χρησιμοποιούσαμε τους όρους "αντικείμενο" και "τιμή" εναλλάξ, αλλά για να ήμαστε ακριβείς είναι πιο σωστό να πούμε ότι ένα αντικείμενο έχει μία τιμή. Αν εκτελέσετε `[1,2,3]` θα πάρετε ένα αντικείμενο λίστας, η τιμή του οποίου είναι μία ακολουθία ακεραίων. Αν μία λίστα έχει τα ίδια στοιχεία με αυτήν τότε λέμε ότι έχει την ίδια τιμή αλλά δεν είναι το ίδιο αντικείμενο.

10.11 Ψευδώνυμα

Αν μία μεταβλητή a αναφέρεται σε ένα αντικείμενο και εσείς την εκχωρήσετε σε μία άλλη μεταβλητή b (`b = a`), τότε και οι δύο μεταβλητές θα αναφέρονται στο ίδιο αντικείμενο:

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

Το διάγραμμα κατάστασης φαίνεται στην Εικόνα 10.4.

Η σύνδεση μίας μεταβλητής με ένα αντικείμενο ονομάζεται **αναφορά** (reference). Σε αυτό το παράδειγμα, υπάρχουν δύο αναφορές στο ίδιο αντικείμενο.

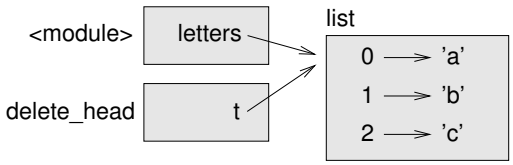
Ένα αντικείμενο με περισσότερες από μία αναφορές έχει και περισσότερα από ένα ονόματα, έτσι λέμε ότι το αντικείμενο έχει **ψευδώνυμα** (aliases).

Αν ένα αντικείμενο με ψευδώνυμο είναι μεταβλητό, οι αλλαγές που γίνονται με το ένα ψευδώνυμο επηρεάζουν και το άλλο:

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

Παρόλο που αυτή η συμπεριφορά μπορεί να φανεί χρήσιμη, είναι επιρρεπής σε λάθη. Γενικά, είναι ασφαλέστερο να αποφεύγετε τα ψευδώνυμα όταν εργάζεστε με αντικείμενα που μπορεί να αλλάξει η τιμή τους (mutable).

Για αμετάβλητα αντικείμενα όπως οι συμβολοσειρές, η εκχώρηση ψευδώνυμων δεν αποτελεί τόσο μεγάλο πρόβλημα. Σε αυτό το παράδειγμα:



Σχήμα 10.5: Διάγραμμα κατάστασης.

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

Δεν έχει σχεδόν καμία διαφορά είτε η a και b αναφέρονται στην ίδια είτε διαφορετική συμβολοσειρά.

10.12 Ορίσματα λίστας

Όταν περνάτε μία λίστα σαν όρισμα σε μία συνάρτηση, η συνάρτηση αποκτά μία αναφορά σε αυτή τη λίστα. Αυτό σημαίνει ότι αν η συνάρτηση τροποποιήσει μία παράμετρο της λίστας τότε ο καλών βλέπει αυτήν την αλλαγή. Για παράδειγμα, η delete_head αφαιρεί το πρώτο στοιχείο από μία λίστα:

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

Εδώ φαίνεται πώς χρησιμοποιείται:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

Η παράμετρος t και η μεταβλητή letters είναι ψευδώνυμα για το ίδιο αντικείμενο. Το διάγραμμα κατάστασης φαίνεται στην Εικόνα 10.5.

Σχεδίασα τη λίστα ανάμεσα στα δύο πλαίσια γιατί είναι κοινόχρηστη μεταξύ τους.

Είναι σημαντικό να γίνει διάκριση μεταξύ των πράξεων οι οποίες τροποποιούν λίστες και των πράξεων οι οποίες δημιουργούν νέες λίστες. Για παράδειγμα, η μέθοδος append τροποποιεί μία λίστα, αλλά ο τελεστής + δημιουργεί μία νέα λίστα:

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

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

Αυτή η διαφορά είναι πολύ σημαντική όταν γράφετε συναρτήσεις οι οποίες υποτίθεται ότι τροποποιούν λίστες. Για παράδειγμα, αυτή η συνάρτηση δεν διαγράφει την κεφαλή μίας λίστας:

```
def bad_delete_head(t):
    t = t[1:]               # WRONG!
```

Ο τελεστής τεμάχιο δημιουργεί μία νέα λίστα και η εκχώρηση κάνει την t να αναφέρεται σε αυτήν την λίστα, αλλά τίποτα από αυτά δεν επηρεάζει τη λίστα που περάστηκε σαν όρισμα.

Μία εναλλακτική λύση είναι να γράψετε μία συνάρτηση η οποία θα δημιουργεί και θα επιστρέφει μία νέα λίστα. Για παράδειγμα, η tail επιστρέφει όλα τα στοιχεία μίας λίστας εκτός από το πρώτο:

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

Αυτή η συνάρτηση αφήνει την αρχική λίστα αμετάβλητη. Αυτό είναι ένα παράδειγμα χρήσης της:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

10.13 Αποσφαλμάτωση

Η απρόσεκτη χρήση λιστών (και άλλων μεταβλητών αντικειμένων) μπορεί να οδηγήσει σε πολύωρη αποσφαλμάτωση. Αυτές είναι μερικές από τις πιο συνηθισμένες παγίδες και μερικοί τρόποι για να τις αποφεύγετε:

1. Να θυμάστε ότι οι περισσότερες μέθοδοι λιστών τροποποιούν το όρισμα και επιστρέφουν None, εν αντιθέσει με τις μεθόδους συμβολοσειρών οι οποίες επιστρέφουν μία νέα συμβολοσειρά και αφήνουν την αρχική απείραχτη.

   Αν έχετε συνηθίσει να γράφετε τέτοιο κώδικα για συμβολοσειρές:

   ```
   word = word.strip()
   ```

   Τότε ίσως μπείτε στον πειρασμό να γράψετε και τέτοιο κώδικα για λίστες:

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

   Η επόμενη πράξη που θα εκτελέσετε με την t πιθανόν να αποτύχει, αφού η sort επιστρέφει None.

   Καλό θα ήταν, προτού χρησιμοποιήσετε μεθόδους και τελεστές στις λίστες, να διαβάσετε την τεκμηρίωση προσεκτικά και να τις δοκιμάσετε σε διαδραστική λειτουργία. Οι μέθοδοι και οι τελεστές που μοιράζονται οι λίστες με άλλες ακολουθίες (όπως οι συμβολοσειρές) τεκμηριώνονται στην διεύθυνση http://docs.python.org/2/library/stdtypes.html#typesseq. Οι μέθοδοι και οι τελεστές που εφαρμόζονται μόνο σε μεταβλητές ακολουθίες τεκμηριώνονται στη διεύθυνση http://docs.python.org/2/library/stdtypes.html#typesseq-mutable.

2. Επιλέξτε ένα τρόπο γραφής και εμμείνετε σε αυτόν.

   Ένα μέρος του προβλήματος με τις λίστες είναι ότι υπάρχουν πολλοί τρόποι για να κάνετε ένα συγκεκριμένο πράγμα. Για παράδειγμα, για να αφαιρέσετε ένα στοιχείο από μία λίστα μπορείτε να χρησιμοποιήσετε την pop, την remove, την del ή ακόμα και μία εκχώρηση με τεμάχιο.

   Για να προσθέσετε ένα στοιχείο, μπορείτε να χρησιμοποιήσετε είτε την μέθοδο append είτε τον τελεστή +. Υποθέτοντας ότι η t είναι μία λίστα και το x είναι ένα στοιχείο της λίστας τότε αυτά είναι σωστά:

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

Και αυτά είναι λάθος:

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

Δοκιμάστε κάθε ένα από αυτά τα παραδείγματα στη διαδραστική λειτουργία για να σιγουρευτείτε ότι καταλαβαίνετε τι κάνουν. Παρατηρήστε ότι μόνο το τελευταίο προκαλεί σφάλμα χρόνου εκτέλεσης, ενώ τα άλλα τρία είναι έγκυρα αλλά δεν κάνουν αυτό που θα έπρεπε.

3. Φτιάξτε αντίγραφα για να αποφεύγετε τα ψευδώνυμα.

   Εάν θέλετε να χρησιμοποιήσετε μία μέθοδο όπως η `sort`, η οποία τροποποιεί το όρισμα, αλλά ταυτόχρονα θέλετε να κρατήσετε και την αρχική λίστα απείραχτη τότε μπορείτε να κάνετε ένα αντίγραφο.

   ```
   orig = t[:]
   t.sort()
   ```

   Σε αυτό το παράδειγμα θα μπορούσατε επίσης να χρησιμοποιήσετε την ενσωματωμένη συνάρτηση `sorted`, η οποία επιστρέφει μία νέα ταξινομημένη λίστα χωρίς να πειράζει την αρχική. Αλλά σε αυτήν την περίπτωση θα πρέπει να αποφύγετε τη χρήση της λέξης `sorted` σαν όνομα μεταβλητής!

10.14 Ορολογία

λίστα: Μία ακολουθία τιμών.

στοιχείο: Μία από τις τιμές μίας λίστας (ή κάποιας άλλης ακολουθίας).

δείκτης: Μία ακέραια τιμή η οποία δείχνει κάποιο στοιχείο μέσα σε μία λίστα.

εμφωλευμένη λίστα: Μία λίστα η οποία είναι στοιχείο μίας άλλης λίστας.

διάσχιση λίστας: Η διαδοχική προσπέλαση κάθε στοιχείου μίας λίστας.

αντιστοίχιση: Μία σχέση στην οποία κάθε στοιχείο ενός συνόλου αντιστοιχεί σε ένα στοιχείο κάποιου άλλου συνόλου. Για παράδειγμα, μία λίστα είναι μία αντιστοίχιση από δείκτες σε στοιχεία.

συσσωρευτής: Μία μεταβλητή η οποία χρησιμοποιείται σε ένα βρόχο για να αθροίζει ή να συσσωρεύει ένα αποτέλεσμα. Είναι γνωστή και ως αθροιστής.

εκχώρηση προσαύξησης: Μία δήλωση η οποία ενημερώνει την τιμή μίας μεταβλητής χρησιμοποιώντας ένα τελεστή όπως ο `+=`.

reduce: Ένα πρότυπο επεξεργασίας το οποίο διασχίζει μία ακολουθία και συσσωρεύει τα στοιχεία της σε ένα μεμονωμένο αποτέλεσμα.

map: Ένα πρότυπο επεξεργασίας το οποίο διασχίζει μία ακολουθία και εκτελεί μία συγκεκριμένη πράξη σε κάθε ένα στοιχείο.

filter: Ένα πρότυπο επεξεργασίας το οποίο διασχίζει μία λίστα και επιλέγει τα στοιχεία τα οποία ικανοποιούν ένα συγκεκριμένο κριτήριο.

αντικείμενο: Κάτι στο οποίο μπορεί να αναφερθεί μία μεταβλητή. Ένα αντικείμενο έχει τύπο και τιμή.

ισότιμα: Έχουν την ίδια τιμή.

ταυτόσημα: Να είναι το ίδιο αντικείμενο (συνεπάγεται και ισοτιμία).

αναφορά: Η σχέση μεταξύ μίας μεταβλητής και της τιμής της.

ψευδώνυμα: Όταν δύο ή περισσότερες μεταβλητές αναφέρονται στο ίδιο αντικείμενο.

διαχωριστής: Ένας χαρακτήρας ή μία συμβολοσειρά που χρησιμοποιείται για να υποδείξει που πρέπει να χωριστεί μία συμβολοσειρά.

10.15 Ασκήσεις

Άσκηση 10.6. *Γράψτε μία συνάρτηση με όνομα* `is_sorted` *η οποία θα παίρνει μία λίστα σαν παράμετρο και θα επιστρέφει* `True` *αν η λίστα είναι ταξινομημένη σε αύξουσα σειρά και* `False` *αλλιώς. Μπορείτε να θεωρήσετε (ως προϋπόθεση) ότι τα στοιχεία της λίστας μπορούν να συγκριθούν μεταξύ τους με τη χρήση των σχεσιακών τελεστών* `<`, `>`, *κτλ.*

Για παράδειγμα, η `is_sorted([1,2,2])` *θα πρέπει να επιστρέψει* `True` *και η* `is_sorted(['b','a'])` *θα πρέπει να επιστρέψει* `False`.

Άσκηση 10.7. *Δύο λέξεις είναι αναγραμματισμοί αν με αναδιάταξη των γραμμάτων της μίας μπορούμε να συλλαβίσουμε την άλλη. Γράψτε μία συνάρτηση με όνομα* `is_anagram` *η οποία θα παίρνει δύο λέξεις και θα επιστρέφει* `True` *αν είναι αναγραμματισμοί.*

Άσκηση 10.8. *Το (λεγόμενο) Παράδοξο των Γενεθλίων:*

1. *Γράψτε μία συνάρτηση με όνομα* `has_duplicates` *η οποία θα παίρνει μία λίστα και θα επιστρέφει* `True` *αν υπάρχει έστω και ένα στοιχείο το οποίο εμφανίζεται περισσότερες από μία φορές. Δεν πρέπει να τροποποιεί την αρχική αρχική λίστα.*
2. *Έστω ότι υπάρχουν 23 φοιτητές στην τάξη σας, ποιες είναι οι πιθανότητες δύο από εσάς να έχουν την ίδια ηλικία; Μπορείτε να υπολογίσετε αυτήν την πιθανότητα μέσω της παραγωγής τυχαίων δειγμάτων για 23 ημέρες γενεθλίων και ελέγχοντας ποιες από αυτές ταιριάζουν. Σημείωση: μπορείτε να παράγετε τυχαίες ημέρες γενεθλίων με τη συνάρτηση* `randint` *του αρθρώματος* `random`.

Μπορείτε να διαβάσετε για αυτό το πρόβλημα στη διεύθυνση `http://en.wikipedia.org/wiki/Birthday_paradox`, *και μπορείτε να κατεβάσετε τη δική μου λύση από εδώ:* `http://thinkpython.com/code/birthday.py`.

Άσκηση 10.9. *Γράψτε μία συνάρτηση με όνομα* `remove_duplicates` *η οποία θα παίρνει μία λίστα και θα επιστρέφει μία νέα λίστα η οποία θα περιέχει μόνο τα μοναδικά στοιχεία από την αρχική. Σημείωση: Δεν είναι απαραίτητο να είναι με την ίδια σειρά.*

Άσκηση 10.10. *Γράψτε μία συνάρτηση η οποία θα διαβάζει το αρχείο* `words.txt` *και θα κατασκευάζει μία λίστα με ένα στοιχείο ανά λέξη. Γράψτε δύο εκδόσεις αυτής της συνάρτησης, μία χρησιμοποιώντας τη μέθοδο* `append` *και μία άλλη χρησιμοποιώντας την ιδιότητα* `t = t + [x]`. *Ποια χρειάζεται περισσότερο χρόνο για να τρέξει και γιατί;*

Σημείωση: Χρησιμοποιήστε το άρθρωμα `time` *για να μετρήσετε το χρόνο που χρειάζεται η κάθε μία. Λύση:* `http://thinkpython.com/code/wordlist.py`.

Άσκηση 10.11. *Για να ελέγξετε αν μία λέξη υπάρχει μέσα σε μία λίστα λέξεων, μπορείτε να χρησιμοποιήσετε τον τελεστή* `in`*, αλλά αυτό θα είναι αργό γιατί ψάχνει όλες τις λέξεις με τη σειρά.*

Αφού οι λέξεις είναι σε αλφαβητική σειρά, μπορούμε να επιταχύνουμε λίγο τα πράγματα χρησιμοποιώντας την αναζήτηση με διχοτόμηση (γνωστή και ως δυαδική αναζήτηση), η οποία δουλεύει παρόμοια με τον τρόπο που θα ψάχνατε μία λέξη σε ένα λεξικό. Ξεκινάτε από τη μέση και ελέγχετε για να δείτε εάν η λέξη που ψάχνετε βρίσκεται πριν τη λέξη στη μέση της λίστας. Αν ναι, τότε ψάχνετε το πρώτο μισό της λίστας με τον ίδιο τρόπο. Αλλιώς ψάχνετε το δεύτερο μισό.

Σε κάθε περίπτωση, κόβετε το υπόλοιπο διάστημα αναζήτησης στο μισό. Αν λίστα λέξεων έχει 113.809 λέξεις θα χρειαστεί περίπου 17 βήματα για να βρεθεί η λέξη ή να καταλήξετε ότι δεν υπάρχει.

Γράψτε μία συνάρτηση με όνομα `bisect` *η οποία θα παίρνει μία ταξινομημένη λίστα και μία τιμή στόχο και θα επιστρέφει τον δείκτη της τιμής στη λίστα εάν υπάρχει ή None εάν δεν υπάρχει.*

Ή μπορείτε να διαβάσετε την τεκμηρίωση του αρθρώματος `bisect` *και να χρησιμοποιήσετε αυτό! Λύση:* `http://thinkpython.com/code/inlist.py`*.*

Άσκηση 10.12. *Δύο λέξεις ονομάζονται "αντίστροφο ζεύγος" αν η μία είναι αντίστροφη της άλλης. Γράψτε ένα πρόγραμμα το οποίο θα βρίσκει όλα τα αντίστροφα ζευγάρια στη λίστα λέξεων. Λύση:* `http://thinkpython.com/code/reverse_pair.py`*.*

Άσκηση 10.13. *Δύο λέξεις "αλληλοσυνδέονται" αν σχηματίζεται μία νέα λέξη παίρνοντας εναλλάξ γράμματα από την κάθε μία. Για παράδειγμα, οι λέξεις "shoe" και "cold" αλληλοσυνδέονται για να σχηματίσουν τη λέξη "schooled". Λύση:* `http://thinkpython.com/code/interlock.py`*. Αναφορά: Αυτή η άσκηση είναι εμπνευσμένη από ένα παράδειγμα στη σελίδα* `http://puzzlers.org`*.*

1. *Γράψτε ένα πρόγραμμα το οποίο θα βρίσκει όλα τα ζευγάρια των λέξεων τα οποία αλληλοσυνδέονται. Σημείωση: Μην απαριθμήσετε όλα τα ζευγάρια!*
2. *Μπορείτε να βρείτε λέξεις οι οποίες είναι τριοδικά "αλληλοσυνδεόμενες"; Αυτό σημαίνει ότι κάθε τρίτο γράμμα σχηματίζει μία λέξη, ξεκινώντας από το πρώτο, το δεύτερο ή το τρίτο.*

Κεφάλαιο 11

Λεξικά

Ένα **λεξικό** μοιάζει με μία λίστα, αλλά είναι κάπως πιο γενικό. Σε μία λίστα, οι δείκτες πρέπει να είναι ακέραιοι αριθμοί αλλά σε ένα λεξικό μπορούν να είναι σχεδόν οποιουδήποτε τύπου.

Μπορείτε να φανταστείτε ένα λεξικό σαν μία αντιστοίχιση μεταξύ ενός συνόλου δεικτών (οι οποίοι ονομάζονται **κλειδιά**) και ενός συνόλου τιμών. Κάθε κλειδί αντιστοιχεί σε μία τιμή. Αυτή η σχέση ονομάζεται **ζευγάρι κλειδιού-τιμής** ή μερικές φορές **στοιχείο**.

Σαν παράδειγμα, θα φτιάξουμε ένα λεξικό το οποίο θα αντιστοιχίζει Αγγλικές με Ισπανικές λέξεις. Επομένως, τόσο τα κλειδιά όσο και οι τιμές θα είναι συμβολοσειρές.

Η συνάρτηση `dict` δημιουργεί ένα νέο λεξικό χωρίς καθόλου στοιχεία. Επειδή το `dict` είναι το όνομα μίας ενσωματωμένης συνάρτησης, θα πρέπει να αποφεύγετε να το χρησιμοποιείτε σαν όνομα μεταβλητής.

```
>>> eng2sp = dict()
>>> print eng2sp
{}
```

Τα άγκιστρα `{}` αναπαριστούν ένα κενό λεξικό. Για να προσθέσετε στοιχεία στο λεξικό μπορείτε να χρησιμοποιήσετε αγκύλες:

```
>>> eng2sp['one'] = 'uno'
```

Αυτή η γραμμή δημιουργεί ένα στοιχείο αντιστοιχίζοντας το κλειδί `'one'` με την τιμή `'uno'`. Αν ξανατυπώσουμε το λεξικό τότε θα δούμε ένα ζευγάρι κλειδιού-τιμής με μία άνω κάτω τελεία μεταξύ του κλειδιού και της τιμής:

```
>>> print eng2sp
{'one': 'uno'}
```

Αυτή η μορφή εξόδου είναι επίσης και μία μορφή εισόδου. Για παράδειγμα, μπορείτε να δημιουργήσετε ένα νέο λεξικό με τρία αντικείμενα κατ' αυτόν τον τρόπο:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Αλλά μάλλον θα εκπλαγείτε αν εμφανίσετε το `eng2sp`:

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

Η σειρά των ζευγαριών κλειδιού-τιμής είναι διαφορετική. Στην πραγματικότητα, αν πληκτρολογήσετε το ίδιο παράδειγμα στον υπολογιστή σας τότε μάλλον θα έχετε πάλι διαφορετικό αποτέλεσμα.

Σε γενικές γραμμές, η σειρά των στοιχείων σε ένα λεξικό απρόβλεπτη.

Αλλά αυτό δεν αποτελεί πρόβλημα επειδή τα αντικείμενα ενός λεξικού δεν ευρετηριάζονται με ακέραιους δείκτες. Αντ' αυτού, μπορείτε να χρησιμοποιήσετε τα κλειδιά για να αναζητήσετε τις αντίστοιχες τιμές:

```
>>> print eng2sp['two']
'dos'
```

Το κλειδί `'two'` αντιστοιχεί πάντα στην τιμή `'dos'`, άρα η σειρά των αντικειμένων δεν έχει σημασία.

Αν το κλειδί δεν υπάρχει στο λεξικό τότε θα πάρετε μία εξαίρεση:

```
>>> print eng2sp['four']
KeyError: 'four'
```

Όταν η συνάρτηση `len` εφαρμόζεται σε ένα λεξικό επιστρέφει το πλήθος ζευγαριών κλειδιού-τιμής:

```
>>> len(eng2sp)
3
```

Ο τελεστής `in` δουλεύει επίσης στα λεξικά και μας ενημερώνει εάν κάτι εμφανίζεται σαν κλειδί μέσα σε ένα λεξικό (δεν είναι τόσο καλός με τις τιμές ωστόσο).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

Για να δείτε αν κάτι εμφανίζεται σαν τιμή σε ένα λεξικό μπορείτε αρχικά να εφαρμόσετε τη μέθοδο `values`, η οποία επιστρέφει τις τιμές σαν μία λίστα, και μετά χρησιμοποιώντας τον τελεστή `in`:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

Ο τελεστής `in` χρησιμοποιεί διαφορετικούς αλγόριθμους για τις λίστες και διαφορετικούς για τα λεξικά. Για τις λίστες χρησιμοποιεί έναν αλγόριθμο αναζήτησης όπως αυτόν στην Ενότητα 8.6. Όσο μεγαλώνει η λίστα, μεγαλώνει ανάλογα και ο χρόνος αναζήτησης. Για τα λεξικά, η Python χρησιμοποιεί έναν αλγόριθμο ο οποίος ονομάζεται **hashtable** και ο οποίος έχει μία αξιοσημείωτη ιδιότητα: ο τελεστής `in` χρειάζεται περίπου τον ίδιο χρόνο ανεξάρτητα από το πόσα στοιχεία υπάρχουν μέσα σε ένα λεξικό. Δεν θα εξηγήσω πως είναι δυνατόν αυτό αλλά μπορείτε να διαβάσετε σχετικά με αυτόν στο σύνδεσμο `http://en.wikipedia.org/wiki/Hash_table`.

Άσκηση 11.1. *Γράψτε μία συνάρτηση η οποία θα διαβάζει τις λέξεις από το αρχείο* `words.txt` *και θα τις αποθηκεύει σαν κλειδιά σε ένα λεξικό. Δεν έχει σημασία ποιες θα είναι οι τιμές. Στη συνέχεια μπορείτε να χρησιμοποιήσετε τον τελεστή* `in` *ως ένα γρήγορο τρόπο για να ελέγξετε εάν μία συμβολοσειρά υπάρχει μέσα στο λεξικό.*

Αν κάνατε την Άσκηση 10.11, τότε μπορείτε να συγκρίνετε την ταχύτητα αυτής της υλοποίησης με τον τελεστή `in`*, όταν εφαρμόζεται σε λίστα, και με την δυαδική αναζήτηση.*

11.1 Το λεξικό ως ένα σύνολο από μετρητές

Υποθέστε ότι σας δίνεται μία συμβολοσειρά και θέλετε να μετρήσετε πόσες φορές εμφανίζεται κάθε γράμμα. Υπάρχουν διάφοροι τρόποι που μπορείτε να το κάνετε αυτό :

1. Μπορείτε να δημιουργήσετε 26 μεταβλητές, μία για κάθε γράμμα του αγγλικού αλφαβήτου. Έπειτα μπορείτε να διασχίσετε τη συμβολοσειρά και, για κάθε χαρακτήρα, να προσαυξάνετε τον αντίστοιχο μετρητή, χρησιμοποιώντας πιθανώς μία αλυσιδωτή συνθήκη.
2. Μπορείτε να δημιουργήσετε μία λίστα με 26 στοιχεία και στη συνέχεια να μετατρέψετε κάθε χαρακτήρα σε έναν αριθμό (χρησιμοποιώντας την ενσωματωμένη συνάρτηση `ord`), να χρησιμοποιήσετε κάθε αριθμό σαν δείκτη μέσα σε μία λίστα και να προσαυξάνετε τον κατάλληλο μετρητή.
3. Μπορείτε να δημιουργήσετε ένα λεξικό με τους χαρακτήρες σαν κλειδιά και τις αντίστοιχες τιμές σαν μετρητές. Την πρώτη φορά που θα συναντήσετε ένα χαρακτήρα θα προσθέσετε ένα αντικείμενο στο λεξικό και έκτοτε θα αυξάνετε την τιμή ενός υπάρχοντος αντικειμένου.

Όλες οι παραπάνω επιλογές εκτελούν τον ίδιο υπολογισμό αλλά κάθε μία από αυτές τον υλοποιεί με διαφορετικό τρόπο.

Υλοποίηση είναι ο τρόπος με τον οποίο εκτελείται ένας υπολογισμός. Μερικές υλοποιήσεις είναι καλύτερες από κάποιες άλλες. Για παράδειγμα, ένα πλεονέκτημα της υλοποίησης με λεξικό είναι ότι δεν χρειάζεται να γνωρίζουμε εκ των προτέρων ποια γράμματα εμφανίζονται σε μία συμβολοσειρά και δεν έχουμε παρά να κάνουμε χώρο για τα γράμματα που εμφανίζονται.

Ο κώδικας θα μπορούσε να είναι κάπως έτσι :

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

Το όνομα της συνάρτησης είναι **histogram**, το οποίο είναι ένας στατιστικός όρος για ένα σύνολο μετρητών (ή συχνοτήτων).

Η πρώτη γραμμή της συνάρτησης δημιουργεί ένα κενό λεξικό και ο βρόγχος `for` διασχίζει τη συμβολοσειρά. Σε κάθε επανάληψη, αν ο χαρακτήρας `c` δεν είναι μέσα στο λεξικό, δημιουργούμε ένα νέο αντικείμενο με κλειδί τον `c` και αρχική τιμή 1 (δεδομένου ότι έχουμε δει αυτό το γράμμα μια φορά). Αν ο `c` υπάρχει ήδη μέσα στο λεξικό τότε αυξάνουμε την `d[c]`.

Εδώ φαίνεται πως δουλεύει :

```
>>> h = histogram('brontosaurus')
>>> print h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

Το ιστόγραμμα υποδεικνύει ότι τα γράμματα `'a'` και `'b'` εμφανίζονται μία φορά, το `'o'` εμφανίζεται δύο φορές και ούτω καθεξής.

Άσκηση 11.2. *Τα λεξικά έχουν μία μέθοδο με όνομα* `get` *η οποία παίρνει σαν ορίσματα ένα κλειδί και μία αρχική τιμή. Εάν το κλειδί εμφανίζεται μέσα στο λεξικό τότε η* `get` *εμφανίζει την αντίστοιχη τιμή, αλλιώς επιστρέφει την αρχική τιμή. Για παράδειγμα :*

```
>>> h = histogram('a')
```

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

Χρησιμοποιήστε την `get` *για να γράψετε την* `histogram` *πιο συνοπτικά. Θα πρέπει να είστε σε θέση να αφαιρέσετε τη δήλωση* `if`.

11.2 Λεξικά και βρόχοι

Αν χρησιμοποιήσετε ένα λεξικό μέσα σε μία δήλωση `for` τότε ο βρόχος θα διασχίσει τα κλειδιά του λεξικού. Για παράδειγμα, η `print_hist` εμφανίζει κάθε κλειδί και την αντίστοιχη τιμή :

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

Η έξοδος θα είναι κάπως έτσι :

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Και πάλι τα κλειδιά δεν είναι με κάποια συγκεκριμένη σειρά.

Άσκηση 11.3. *Τα λεξικά έχουν μία μέθοδο με όνομα* `keys` *η οποία επιστρέφει τα κλειδιά ενός λεξικού, χωρίς συγκεκριμένη σειρά, με τη μορφή λίστας.*

Τροποποιήστε την `print_hist` *ούτως ώστε να εμφανίζει τα κλειδιά και τις τιμές τους σε αλφαβητική σειρά.*

11.3 Αντίστροφη αναζήτηση

Δοθέντος ενός λεξικού `d` και ενός κλειδιού `k` , είναι εύκολο να βρούμε την αντίστοιχη τιμή `v = d[k]`. Αυτή η διαδικασία ονομάζεται αναζήτηση.

Τι θα κάνατε όμως αν είχατε την `v` και θέλατε να βρείτε το `k`; Έχετε δύο προβλήματα : πρώτον, μπορεί να υπάρχουν περισσότερα από ένα κλειδιά τα οποία αντιστοιχούν στην τιμή `v`. Ανάλογα με την εφαρμογή, θα μπορούσατε να διαλέξετε ένα ή θα πρέπει να φτιάξετε μία λίστα η οποία να τα περιέχει όλα. Δεύτερον, δεν υπάρχει απλός τρόπος για να συνάξετε μία αντίστροφη αναζήτηση, θα πρέπει να ψάξετε.

Αυτή είναι μία συνάρτηση η οποία παίρνει μία τιμή και επιστρέφει το πρώτο κλειδί το οποίο αντιστοιχεί σε αυτήν την τιμή :

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
```

```
            return k
    raise ValueError
```

Αυτή η συνάρτηση είναι ένα ακόμα παράδειγμα πρότυπου αναζήτησης, αλλά χρησιμοποιεί μία λειτουργία την οποία δεν έχουμε ξαναδεί, τη `raise`. Η δήλωση `raise` προκαλεί μία εξαίρεση, σε αυτήν την περίπτωση προκαλεί ένα λάθος τιμής `ValueError`, το οποίο υποδεικνύει γενικά ότι υπάρχει κάποιο λάθος με την τιμή μίας παραμέτρου.

Αν φτάσουμε στο τέλος του βρόχου, τότε σημαίνει ότι η v δεν εμφανίζεται στο λεξικό σαν τιμή και επομένως πρέπει να εγείρουμε μία εξαίρεση.

Αυτό είναι ένα παράδειγμα από μία επιτυχή αντίστροφη αναζήτηση :

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> print k
r
```

Και μία ανεπιτυχή :

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in reverse_lookup
ValueError
```

Το αποτέλεσμα όταν εγείρετε εσείς μία εξαίρεση είναι το ίδιο όπως όταν εγείρει η Python: εμφανίζει μία αναδρομή και ένα μήνυμα λάθους.

Η δήλωση `raise` μπορεί να πάρει ένα λεπτομερές μήνυμα σαν όρισμα. Για παράδειγμα :

```
>>> raise ValueError, 'value does not appear in the dictionary'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: value does not appear in the dictionary
```

Μία αντίστροφη αναζήτηση είναι πολύ πιο αργή από μία προς τα εμπρός αναζήτηση. Αν πρέπει να την κάνετε συχνά ή αν το λεξικό γίνει μεγάλο, η απόδοση του προγράμματός σας θα υποφέρει.

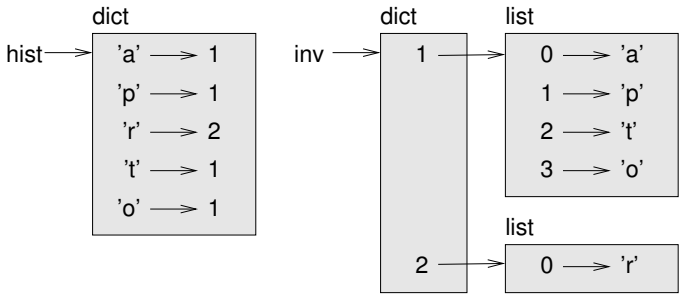
Άσκηση 11.4. *Τροποποιήστε την* `reverse_lookup` *έτσι ώστε να φτιάχνει και να επιστρέφει μία λίστα με όλα τα κλειδιά τα οποία αντιστοιχούν στην* v*, ή μία κενή λίστα εάν δεν υπάρχει κανένα.*

11.4 Λεξικά και λίστες

Οι λίστες μπορούν να εμφανιστούν σαν τιμές σε ένα λεξικό. Για παράδειγμα, αν σας δινόταν ένα λεξικό το οποίο αντιστοιχούσε γράμματα σε συχνότητες ίσως να θέλατε να το αντιστρέψετε, να αντιστοιχεί δηλαδή τις συχνότητες στα γράμματα. Από τη στιγμή που μπορεί να υπάρχουν αρκετά γράμματα με την ίδια συχνότητα, κάθε τιμή στο αντεστραμμένο λεξικό θα πρέπει να είναι μία λίστα γραμμάτων.

Αυτή είναι μία συνάρτηση η οποία αντιστρέφει ένα λεξικό :

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
```



Σχήμα 11.1: Διάγραμμα κατάστασης.

```
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

Σε κάθε επανάληψη, η `key` παίρνει ένα κλειδί από το `d` και η `val` παίρνει την αντίστοιχη τιμή. Αν η `val` δεν υπάρχει στο `inverse` σημαίνει ότι δεν την έχουμε ξαναδεί πιο πριν, επομένως δημιουργούμε ένα νέο αντικείμενο και το αρχικοποιούμε με μία μονήρη λίστα (singleton μία λίστα η οποία περιέχει μόνο ένα στοιχείο). Διαφορετικά, έχουμε ξαναδεί αυτήν την τιμή οπότε επισυνάπτουμε το αντίστοιχο κλειδί στη λίστα.

Για παράδειγμα :

```
>>> hist = histogram('parrot')
>>> print hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> print inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

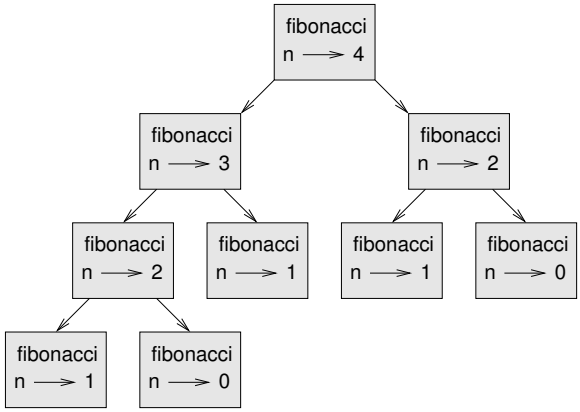
Η εικόνα 11.1 είναι ένα διάγραμμα κατάστασης το οποίο δείχνει το `hist` και το `inverse`. Ένα λεξικό αναπαριστάται από ένα κουτί με τον τύπο `dict` από πάνω του και τα ζευγάρια κλειδιού-τιμής μέσα του. Αν οι τιμές είναι ακέραιοι, δεκαδικοί ή συμβολοσειρές, τις σχεδιάζω συνήθως μέσα στο κουτί, αλλά αν είναι λίστες τότε απέξω, για να κρατάω το διάγραμμα απλό.

Οι λίστες μπορεί να είναι τιμές σε ένα λεξικό, όπως είδαμε σε αυτό το παράδειγμα, αλλά δεν μπορεί να είναι κλειδιά. Να τι θα συμβεί αν δοκιμάσετε κάτι τέτοιο :

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

Όπως ανέφερα προηγουμένως, ένα λεξικό υλοποιείται χρησιμοποιώντας έναν πίνακα κατακερματισμού και αυτό σημαίνει ότι τα κλειδιά πρέπει να είναι ” κατακερματίσημα ”.

Συνάρτηση κατακερματισμού είναι μία συνάρτηση η οποία παίρνει μία τιμή (οποιουδήποτε τύπου) και επιστρέφει έναν ακέραιο αριθμό. Τα λεξικά χρησιμοποιούν αυτούς τους ακέραιους, οι οποίοι ονομάζονται τιμές κατακερματισμού ή αλλιώς τιμές κατατεμαχισμού, για να αποθηκεύουν και να αναζητούν ζευγάρια κλειδιού-τιμής.



Σχήμα 11.2: Διάγραμμα κλήσεων.

Αυτό το σύστημα δουλεύει μια χαρά αν τα κλειδιά είναι αμετάβλητα. Αλλά αν τα κλειδιά είναι ευμετάβλητα, όπως είναι λίστες, τότε συμβαίνουν " κακά πράγματα ". Για παράδειγμα, όταν δημιουργείτε ένα ζευγάρι κλειδιού-τιμής, η Python κατακερματίζει το κλειδί και το αποθηκεύει στην αντίστοιχη θέση. Αν τροποποιήσετε το κλειδί και το επανακατακερματίσετε τότε θα πάει σε διαφορετική θέση. Σε αυτήν την περίπτωση ή θα έχετε δύο εγγραφές για το ίδιο κλειδί ή δεν θα μπορείτε να βρείτε κανένα κλειδί. Σε κάθε περίπτωση, το λεξικό δεν θα δουλεύει σωστά.

Γι' αυτόν το λόγο τα κλειδιά πρέπει να είναι κατακερματίσημα, και για αυτό οι ευμετάβλητοι τύποι όπως οι λίστες είναι ακατάλληλοι. Ο απλούστερος τρόπος για να ξεφύγουμε από αυτόν τον περιορισμό είναι να χρησιμοποιήσουμε πλειάδες, τις οποίες θα τις δούμε στο επόμενο κεφάλαιο.

Από τη στιγμή που τα λεξικά είναι ευμετάβλητα δεν μπορούν να χρησιμοποιηθούν σαν κλειδιά, αλλά μπορούν να χρησιμοποιηθούν σαν τιμές.

Άσκηση 11.5. *Διαβάστε την τεκμηρίωση της μεθόδου των λεξικών* `setdefault` *και χρησιμοποιήστε την για να γράψετε μία πιο λακωνική έκδοση της* `invert_dict`. *Λύση :* `http://thinkpython.com/code/invert_dict.py`.

11.5 Σημείωμα

Αν παίξατε με τη συνάρτηση `fibonacci` της Ενότητας 6.7, θα παρατηρήσατε ίσως ότι όσο μεγαλύτερο όρισμα παρέχετε, τόσο περισσότερο χρόνο χρειάζεται η συνάρτηση για να τρέξει. Επιπροσθέτως, ο χρόνος εκτέλεσης αυξάνεται πολύ γρήγορα.

Για να καταλάβετε το γιατί, μελετήστε την Εικόνα 11.2, η οποία δείχνει το διάγραμμα κλήσεων για την `fibonacci` με `n=4`:

Ένα διάγραμμα κλήσεων δείχνει ένα σύνολο από πλαίσια συναρτήσεων με γραμμές οι οποίες συνδέουν κάθε πλαίσιο με τα πλαίσια των συναρτήσεων που αυτό καλεί. Στην κορυφή του διαγράμματος, η `fibonacci` με `n=4` καλεί την `fibonacci` με `n=3` και `n=2`. Με τη σειρά της, η `fibonacci` με `n=3` καλεί τη `fibonacci` με `n=2` και `n=1` και ούτω καθεξής.

Μετρήστε πόσες φορές καλέστηκαν οι `fibonacci(0)` και `fibonacci(1)`. Αυτή η λύση είναι αναποδοτική και χειροτερεύει όσο μεγαλώνει το όρισμα.

Μία άλλη λύση είναι να παρακολουθούμε τις τιμές οι οποίες έχουν ήδη υπολογιστεί αποθηκεύοντάς τες σε ένα λεξικό. Μία ήδη υπολογισμένη τιμή η οποία αποθηκεύτηκε για μεταγενέστερη χρήση

ονομάζεται σημείωμα. Αυτή είναι μία υλοποίηση της fibonacci χρησιμοποιώντας σημειώματα :

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

def fibonacci(n):
    if n in known:
        return known[n]

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

Το known είναι ένα λεξικό το οποίο κρατάει όλους τους αριθμούς Fibonacci που ήδη ξέρουμε. Ξεκινάει με δύο αντικείμενα : το 0 αντιστοιχεί στο 0 και το 1 αντιστοιχεί στο 1.

Κάθε φορά που καλείται η fibonacci ελέγχει το known. Αν το αποτέλεσμα υπάρχει ήδη εκεί επιστρέφει αμέσως. Αλλιώς, πρέπει να υπολογίσει τη νέα τιμή, να την προσθέσει στο λεξικό και να την επιστρέψει.

Άσκηση 11.6. *Τρέξτε αυτήν την έκδοση της* fibonacci *και την αρχική με ένα εύρος παραμέτρων και συγκρίνετε τους χρόνους εκτέλεσης.*

Άσκηση 11.7. *Τροποποιήστε τη συνάρτηση του Άκερμαν της άσκησης 6.5 χρησιμοποιώντας σημειώματα και δείτε αν είναι δυνατόν να υπολογίσετε τη συνάρτηση με μεγαλύτερα ορίσματα. Σημείωση : όχι. Λύση :* `http://thinkpython.com/code/ackermann_memo.py`.

11.6 Καθολικές μεταβλητές

Στο προηγούμενο παράδειγμα, το known δημιουργείται εκτός της συνάρτησης, άρα υπάγεται στο ειδικό πλαίσιο που ονομάζεται __main__. Συνήθως, οι μεταβλητές στη __main__ ονομάζονται καθολικές επειδή μπορούν να προσπελαστούν από οποιαδήποτε συνάρτηση. Αντιθέτως με τις τοπικές μεταβλητές, οι οποίες εξαφανίζονται όταν τελειώσει η συνάρτηση η οποία τις δημιούργησε, οι καθολικές μεταβλητές εξακολουθούν να υπάρχουν από τη μία κλήση συνάρτησης στην επόμενη.

Οι καθολικές μεταβλητές χρησιμοποιούνται συχνά για **flags**, οι οποίες είναι μεταβλητές αληθείας που δείχνουν τη " σημαία " όταν μία συνθήκη είναι αληθής. Για παράδειγμα, μερικά προγράμματα χρησιμοποιούν μία σημαία με όνομα verbose για να ελέγχουν με ακρίβεια την έξοδο :

```
verbose = True

def example1():
    if verbose:
        print 'Running example1'
```

Αν προσπαθήσετε να αλλάξετε την τιμή μίας καθολικής μεταβλητής, μάλλον θα εκπλαγείτε. Το ακόλουθο παράδειγμα υποτίθεται ότι ελέγχει εάν έχει κληθεί η συνάρτηση :

```
been_called = False

def example2():
    been_called = True          # WRONG
```

Αλλά αν το τρέξετε θα δείτε ότι η τιμή της been_called δεν έχει αλλάξει. Το πρόβλημα είναι ότι η example2 δημιουργεί μία νέα τοπική μεταβλητή με όνομα been_called η οποία χάνεται όταν τελειώσει η συνάρτηση χωρίς να επηρεάζει την καθολική μεταβλητή.

Για να τροποποιήσετε μία καθολική μεταβλητή μέσα σε μία συνάρτηση πρέπει να δηλώσετε τη καθολική μεταβλητή πριν τη χρησιμοποιήσετε :

```
been_called = False

def example2():
    global been_called
    been_called = True
```

Η διασαφήνιση `global` λέει στο διερμηνέα κάτι τέτοιο : " Σε αυτήν την συνάρτηση, όταν λέω `been_called`, εννοώ την καθολική μεταβλητή, μην δημιουργείς νέα τοπική ".

Αυτό είναι ένα παράδειγμα το οποίο προσπαθεί να ενημερώσει μία καθολική μεταβλητή :

```
count = 0

def example3():
    count = count + 1          # WRONG
```

Αν το τρέξετε θα πάρετε :

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

Η Python υποθέτει ότι η `count` είναι τοπική, το οποίο σημαίνει ότι προσπαθείτε να την διαβάσετε προτού της εκχωρήσετε τιμή. Η λύση είναι πάλι να δηλώσετε την `count` σαν καθολική :

```
def example3():
    global count
    count += 1
```

Αν η καθολική μεταβλητή είναι ευμετάβλητη, μπορείτε να την τροποποιήσετε χωρίς να τη δηλώσετε :

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

def example4():
    known[2] = 1
```

Επομένως, μπορείτε να προσθέσετε, να αφαιρέσετε και να αντικαταστήσετε τα στοιχεία μίας καθολικής λίστας ή ενός λεξικού, αλλά αν θέλετε να αλλάξετε την τιμή της μεταβλητής πρέπει να τη δηλώσετε :

```
def example5():
    global known
    known = dict()
```

11.7 Ακέραιοι μεγάλου μήκους

Αν υπολογίσετε την `fibonacci(50)` θα πάρετε :

```
>>> fibonacci(50)
12586269025L
```

Το `L` στο τέλος υποδεικνύει ότι το αποτέλεσμα είναι ένας ακέραιος αριθμός μεγάλου μήκους `long`. Αυτός ο τύπος μεταβλητής δεν υπάρχει στην Python 3, όλοι οι ακέραιοι, ακόμα και οι πολύ μεγάλοι είναι τύπου `int`.

Οι τιμές με τύπο `int` έχουν περιορισμένο εύρος. Οι ακέραιοι μεγάλου μήκους μπορεί ναι είναι αυθαίρετα μεγάλοι, αλλά όσο μεγαλώνουν καταναλώνουν περισσότερο χώρο και χρόνο.

Τόσο οι μαθηματικοί τελεστές όσο και οι συναρτήσεις της μονάδας λογισμικού `math` δουλεύουν και στους ακέραιους μεγάλου μήκους. Επομένως, σε γενικές γραμμές, οποιοσδήποτε κώδικας δουλεύει με `int` θα δουλεύει και με `long`.

Κάθε φορά που το αποτέλεσμα ενός υπολογισμού είναι πολύ μεγάλο για να αναπαρασταθεί από έναν ακέραιο, η Python μετατρέπει το αποτέλεσμα σε ένα ακέραιο μεγάλου μήκους :

```
>>> 1000 * 1000
1000000
>>> 100000 * 100000
10000000000L
```

Στην πρώτη περίπτωση, το αποτέλεσμα έχει τύπο `int`, ενώ στη δεύτερη είναι `long`.

Άσκηση 11.8. *Η ύψωση σε δύναμη μεγάλων ακεραίων είναι η βάση των πιο συνηθισμένων αλγορίθμων κρυπτογράφησης δημόσιου κλειδιού. Διαβάστε τη σελίδα της Wikipedia για τον αλγόριθμο RSA (`http://en.wikipedia.org/wiki/RSA`) και γράψτε συναρτήσεις οι οποίες θα κωδικοποιούν και θα αποκωδικοποιούν μηνύματα.*

11.8 Αποσφαλμάτωση

Καθώς θα εργάζεστε με όλο και μεγαλύτερα σύνολα δεδομένων, η χειροκίνητη αποσφαλμάτωση με εμφάνιση και έλεγχο δεδομένων μπορεί να γίνει ιδιαίτερα δύσχρηστη. Αυτές είναι μερικές υποδείξεις για την αποσφαλμάτωση μεγάλων όγκων δεδομένων :

Περιορίστε την είσοδο : Μειώστε το μέγεθος του συνόλου δεδομένων αν είναι δυνατό. Για παράδειγμα, αν το πρόγραμμα διαβάζει ένα αρχείο κειμένου, ξεκινήστε μόνο με τις πρώτες 10 γραμμές ή με το μικρότερο παράδειγμα που μπορείτε να βρείτε. Μπορείτε είτε να επεξεργαστείτε τα ίδια τα αρχεία, είτε (καλύτερα) να τροποποιήσετε το πρόγραμμα ούτως ώστε να διαβάζει μόνο τις πρώτες `n` γραμμές.

Αν προκύψει κάποιο σφάλμα, μπορείτε να μειώσετε το `n` στη μικρότερη τιμή που εμφανίζεται το σφάλμα και στη συνέχεια να την αυξάνετε σταδιακά όσο εντοπίζετε και διορθώνετε λάθη.

Ελέγξτε συνόψεις και τύπους: Αντί να εμφανίζετε και να ελέγχετε ολόκληρο το σύνολο δεδομένων, δοκιμάστε να εμφανίσετε συνόψεις των δεδομένων. Για παράδειγμα, μια σύνοψη μπορεί να είναι ο πλήθος των αντικειμένων ενός λεξικού ή το σύνολο των αριθμών μίας λίστας.

Μία συχνή αιτία για τα σφάλματα χρόνου εκτέλεσης είναι μία τιμή με λάθος τύπο δεδομένων. Για την αποσφαλμάτωση αυτού του τύπου σφάλματος, τις περισσότερες φορές αρκεί να εμφανίσουμε τον τύπο μίας μεταβλητής.

Γράψτε αυτοελέγχους: Μερικές φορές μπορείτε να γράψετε κώδικα ο οποίος θα ελέγχει για σφάλματα αυτόματα. Για παράδειγμα, αν υπολογίζατε το μέσο όρο μίας λίστας αριθμών, θα μπορούσατε να ελέγξετε ότι το αποτέλεσμα δεν είναι μεγαλύτερο από το μεγαλύτερο στοιχείο της λίστας ή μικρότερο από το μικρότερο στοιχείο. Αυτός ονομάζεται ” λογικός έλεγχος ” γιατί ανιχνεύει αποτελέσματα τα οποία είναι ” παράλογα ”.

Ένα άλλο είδος ελέγχου συγκρίνει τα αποτελέσματα από δύο διαφορετικούς υπολογισμούς για να δει αν συνάδουν. Αυτός ονομάζεται ” έλεγχος συνέπειας ”.

Ομορφύνετε την έξοδο : Η μορφοποίηση της εξόδου αποσφαλμάτωσης μπορεί να κάνει ευκολότερο τον εντοπισμό ενός λάθους. Είδαμε ένα τέτοιο παράδειγμα στην Ενότητα 6.9. Η μονάδα

λογισμικού `pprint` παρέχει μία συνάρτηση `pprint` η οποία εμφανίζει τους ενσωματωμένους τύπους σε μία πιο ευανάγνωστη μορφή.

Και πάλι, ο χρόνος που ξοδεύετε για να φτιάξετε μία σκαλωσιά μπορεί να μειώσει το χρόνο που θα ξοδεύατε για αποσφαλμάτωση.

‘

11.9 Ορολογία

λεξικό : Μία αντιστοίχιση ενός συνόλου κλειδιών με τις αντίστοιχες τιμές τους.

ζευγάρι κλειδιού-τιμής : Η αναπαράσταση μίας αντιστοίχισης ενός κλειδιού με μία τιμή.

αντικείμενο: Μία άλλη ονομασία για ένα ζευγάρι κλειδιού-τιμής.

κλειδί : Ένα αντικείμενο το οποίο εμφανίζεται μέσα σε ένα λεξικό σαν το πρώτο μέρος ενός ζευγαριού κλειδιού-τιμής.

τιμή : Ένα αντικείμενο το οποίο εμφανίζεται μέσα σε ένα λεξικό σαν το δεύτερο μέρος ενός ζευγαριού κλειδιού-τιμής. Αυτή είναι πιο συγκεκριμένη χρήση της λέξης ” τιμή ” σε σχέση με την προηγούμενη.

υλοποίηση : Ένας τρόπος εκτέλεσης ενός υπολογισμού.

πίνακας κατακερματισμού : Ο αλγόριθμος που χρησιμοποιείται για να υλοποιήσει λεξικά στην Python.

συνάρτηση κατακερματισμού : Μία συνάρτηση που χρησιμοποιείται από ένα πίνακα κατακερματισμού για να υπολογίσει την θέση ενός κλειδιού.

κατακερματίσημος : Ένας τύπος ο οποίος έχει μία συνάρτηση κατακερματισμού. Οι αμετάβλητοι τύποι όπως οι ακέραιοι, οι δεκαδικοί και οι συμβολοσειρές είναι κατακερματίσημοι, ενώ οι ευμετάβλητοι τύποι όπως είναι οι λίστες και τα λεξικά δεν είναι.

αναζήτηση : Μία διαδικασία η οποία παίρνει ένα κλειδί και βρίσκει την αντίστοιχη τιμή.

αντίστροφη αναζήτηση : Μία διαδικασία η οποία παίρνει μία τιμή και βρίσκει ένα ή περισσότερα κλειδιά τα οποία της αντιστοιχούν.

μονήρης : Μία λίστα (ή κάποια άλλη ακολουθία) με μόνο ένα στοιχείο.

διάγραμμα κλήσεων : Ένα διάγραμμα το οποίο δείχνει κάθε πλαίσιο το οποίο δημιουργείται κατά την εκτέλεση ενός προγράμματος, με ένα βέλος από τον κάθε καλούντα στον κάθε καλούμενο.

ιστόγραμμα : Ένα σύνολο από μετρητές.

σημείωμα : Μία ήδη υπολογισμένη τιμή η οποία αποθηκεύεται για την αποφυγή περιττού υπολογισμού στο μέλλον.

καθολική μεταβλητή : Μία μεταβλητή η οποία ορίζεται έξω από μία συνάρτηση. Οι καθολικές μεταβλητές μπορούν να προσπελαστούν από οποιαδήποτε συνάρτηση.

σημαία : Μία μεταβλητή αληθείας η οποία χρησιμοποιείται για να υποδείξει αν μία συνθήκη είναι αληθής ή όχι.

διασαφήνιση : Μία δήλωση όπως η `global` η οποία λέει στο διερμηνέα κάτι σχετικά με μία μεταβλητή.

11.10 Ασκήσεις

Άσκηση 11.9. *Αν κάνατε την Άσκηση 10.8, έχετε ήδη μία συνάρτηση με όνομα* `has_duplicates` *η οποία παίρνει μία λίστα σαν παράμετρο και επιστρέφει* `True` *αν υπάρχει τουλάχιστον ένα αντικείμενο το οποίο εμφανίζεται περισσότερες από μία φορά μέσα στη λίστα.*

Χρησιμοποιήστε ένα λεξικό για να γράψετε μία πιο γρήγορη και απλούστερη έκδοση της `has_duplicates`*.*
Λύση : `http://thinkpython.com/code/has_duplicates.py`*.*

Άσκηση 11.10. *Δύο λέξεις είναι " αντεστραμμένο ζεύγος " αν περιστρέφοντας τη μία από αυτές παίρνετε την άλλη (δείτε την* `rotate_word` *στην Άσκηση 8.12).*

Γράψτε ένα πρόγραμμα το οποίο θα διαβάζει μία λίστα λέξεων και θα βρίσκει όλα τα αντεστραμμένα ζεύγη. Λύση : `http://thinkpython.com/code/rotate_pairs.py`*.*

Άσκηση 11.11. *Αυτός είναι ένας ακόμα γρίφος από την* Car Talk *(*`http://www.cartalk.com/content/puzzlers`*):*

> *Αυτό το έστειλε ένας συνάδελφος, ο Dan O'Leary, ο οποίος πρόσφατα, έπεσε τυχαία πάνω σε μία συνηθισμένη μονοσύλλαβη λέξη πέντε γραμμάτων η οποία έχει την εξής μοναδική ιδιότητα. Όταν αφαιρέσετε το πρώτο γράμμα, τα υπόλοιπα σχηματίζουν ένα ομόφωνο της αρχικής λέξης, το οποίο είναι μία λέξη η οποία ακούγεται ακριβώς το ίδιο. Αντικαταστήστε το πρώτο γράμμα, ήτοι, βάλτε το πάλι πίσω και αφαιρέστε το δεύτερο γράμμα και το αποτέλεσμα είναι πάλι ένα ομόφωνο της αρχικής λέξης. Και η ερώτηση είναι, ποια είναι η λέξη ;*
>
> *Τώρα θα σας δώσω ένα παράδειγμα το οποίο όμως δεν δουλεύει. Ας δούμε την πέντε γραμμάτων λέξη "wrack". W-R-A-C-K, όπως λέμε "wrack with pain" (ταλαιπωρήστε από τον πόνο). Αν αφαιρέσω το πρώτο γράμμα, μου μένει μία λέξη με τέσσερα γράμματα, η "R-A-C-K". Όπως λέμε : 'Holy cow, did you see the rack on that buck! It must have been a nine-pointer!' Είναι ένα τέλειο ομόφωνο. Αντ' αυτού, αν ξαναβάλετε πίσω το γράμμα 'w' και αφαιρέσετε το 'r' θα προκύψει η λέξη "wack", η οποία είναι μία πραγματική λέξη, δεν είναι ομόφωνο από τις άλλες δύο λέξεις.*
>
> *Αλλά υπάρχει, ωστόσο, τουλάχιστον μία λέξη την οποία γνωρίζουμε και η οποία αποδίδει δύο ομόφωνα αν αφαιρέσετε ένα από τα δύο πρώτα γράμματα φτιάχνοντας δύο νέες λέξεις τεσσάρων γραμμάτων. Η ερώτηση είναι , ποια είναι η λέξη ;*

Μπορείτε να χρησιμοποιήσετε το λεξικό της Άσκησης 11.1 για να ελέγξετε αν μία συμβολοσειρά υπάρχει μέσα στη λίστα λέξεων.

Για να ελέγξετε αν δύο λέξεις είναι ομόφωνα, μπορείτε να χρησιμοποιήσετε το λεξικό προφοράς CMU. Μπορείτε να το κατεβάσετε από το σύνδεσμο `http://www.speech.cs.cmu.edu/cgi-bin/cmudict` *ή από τον σύνδεσμο* `http://thinkpython.com/code/c06d` *και επίσης μπορείτε να κατεβάσετε και το* `http://thinkpython.com/code/pronounce.py`*, το οποίο παρέχει μία συνάρτηση με όνομα* `read_dictionary` *η οποία διαβάζει το λεξικό προφοράς και επιστρέφει ένα λεξικό σε Python το οποίο αντιστοιχεί κάθε λέξη σε μία συμβολοσειρά η οποία περιγράφει την βασική προφορά της.*

Γράψτε ένα πρόγραμμα το οποίο θα εμφανίζει όλες τις λέξεις η οποίες επιλύουν το γρίφο. Λύση : `http://thinkpython.com/code/homophone.py`*.*

Κεφάλαιο 12

Πλειάδες

12.1 Οι πλειάδες είναι αμετάβλητες

Μία πλειάδα είναι μία ακολουθία τιμών οποιουδήποτε τύπου οι οποίες ευρετηριάζονται με ακεραίους αριθμούς. Από αυτήν την άποψη οι πλειάδες μοιάζουν πολύ με τις λίστες. Η σημαντική διαφορά είναι ότι οι λίστες είναι αμετάβλητες.

Συντακτικά, μία πλειάδα είναι μία λίστα τιμών χωρισμένες με κόμματα :

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

Παρόλο που δεν είναι απαραίτητο, συνηθίζεται να εμπερικλείουμε τις πλειάδες μέσα σε παρενθέσεις :

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

Για να δημιουργήσετε μία πλειάδα με μόνο ένα στοιχείο, θα πρέπει να συμπεριλάβετε ένα κόμμα στο τέλος :

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

Μία τιμή μέσα σε παρενθέσεις δεν είναι πλειάδα :

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

Ένας άλλος τρόπος για να δημιουργήσετε μία πλειάδα είναι ενσωματωμένη συνάρτηση `tuple`. Χωρίς ορίσματα δημιουργεί μία κενή πλειάδα :

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

Αν το όρισμα είναι μία ακολουθία (συμβολοσειρά, λίστα ή πλειάδα), το αποτέλεσμα είναι μία πλειάδα με τα στοιχεία της ακολουθίας :

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

Επειδή το tuple είναι το όνομα μίας ενσωματωμένης συνάρτησης θα πρέπει να αποφεύγετε να το χρησιμοποιείτε σαν όνομα μεταβλητής.

Οι περισσότεροι τελεστές που χρησιμοποιούνται στις λίστες δουλεύουν και με τις πλειάδες. Για παράδειγμα, ο τελεστής αγκύλης δείχνει ένα στοιχείο :

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

Και ο τελεστής τεμαχίου επιλέγει ένα εύρος στοιχείων :

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

Αλλά αν προσπαθήσετε να τροποποιήσετε ένα από τα στοιχεία της πλειάδας, θα πάρετε ένα λάθος :

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

Δεν μπορείτε να τροποποιήσετε τα στοιχεία μίας πλειάδας, αλλά μπορείτε να αντικαταστήσετε μία πλειάδα με μία άλλη :

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

12.2 Εκχώρηση πλειάδων

Είναι συχνά χρήσιμο να αντιμεταθέτουμε τις τιμές δύο μεταβλητών. Με τις συμβατικές εκχωρήσεις είναι απαραίτητο να χρησιμοποιήσετε μία προσωρινή μεταβλητή. Για παράδειγμα, για να αντιμεταθέσετε την a και την b:

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

Αυτή η λύση είναι κάπως επαχθής σε σχέση με την εκχώρηση πλειάδων, η οποία είναι πιο κομψή :

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

Στην αριστερή πλευρά είναι μία πλειάδα μεταβλητών και στη δεξιά πλευρά είναι μία πλειάδα εκφράσεων. Κάθε τιμή εκχωρείται στην αντίστοιχη μεταβλητή. Όλες οι εκφράσεις στην δεξιά πλευρά υπολογίζονται πριν από οποιαδήποτε εκχώρηση.

Το πλήθος των μεταβλητών στα αριστερά και το πλήθος των τιμών στα δεξιά πρέπει να είναι τα ίδια :

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

Γενικότερα, η δεξιά πλευρά μπορεί να είναι οποιοδήποτε είδος ακολουθίας (συμβολοσειρά, λίστα ή πλειάδα). Για παράδειγμα, για να χωρίσετε μία διεύθυνση ηλεκτρονικού ταχυδρομείου στο όνομα χρήστη και στον τομέα, θα μπορούσατε να γράψετε :

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

Η επιστρεφόμενη τιμή από την `split` είναι μία λίστα με δύο στοιχεία. Το πρώτο στοιχείο εκχωρείται στην `uname` και το δεύτερο στην `domain`.

```
>>> print uname
monty
>>> print domain
python.org
```

12.3 Οι πλειάδες σαν επιστρεφόμενες τιμές

Στην πραγματικότητα, μία συνάρτηση μπορεί να επιστρέψει μόνο μία τιμή, αλλά αν η τιμή είναι μία πλειάδα τότε είναι σαν να επιστρέφει πολλές τιμές. Για παράδειγμα, αν θέλατε να διαιρέσετε δύο ακέραιους και να υπολογίσετε το πηλίκο και το υπόλοιπο είναι προτιμότερο να τα υπολογίσετε ταυτόχρονα παρά να υπολογίσετε πρώτα το `x/y` και μετά το `x%y`.

Η ενσωματωμένη συνάρτηση `divmod` παίρνει δύο ορίσματα και επιστρέφει μία πλειάδα δύο τιμών, το πηλίκο και το υπόλοιπο. Μπορείτε να αποθηκεύσετε το αποτέλεσμα σαν μία πλειάδα :

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

Ή να αποθηκεύσετε τα στοιχεία χωριστά χρησιμοποιώντας εκχώρηση πλειάδας :

```
>>> quot, rem = divmod(7, 3)
>>> print quot
2
>>> print rem
1
```

Αυτό είναι ένα παράδειγμα μιας συνάρτησης η οποία επιστρέφει μία πλειάδα :

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

Η `max` και η `min` είναι ενσωματωμένες συναρτήσεις η οποίες βρίσκουν το μεγαλύτερο και το μικρότερο στοιχείο μίας ακολουθίας. Η `min_max` υπολογίζει και τα δύο και επιστρέφει μία πλειάδα δύο τιμών.

12.4 Οι πλειάδες σαν ορίσματα μεταβλητού μήκους

Οι συναρτήσεις μπορούν να πάρουν ένα μεταβλητό πλήθος από ορίσματα. Μία παράμετρος το όνομα της οποίας ξεκινάει με `*` συσσωρεύει ορίσματα σε μία πλειάδα. Για παράδειγμα, η `printall` παίρνει οποιοδήποτε πλήθος ορισμάτων και τα εμφανίζει :

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

Η παράμετρος συσσώρευσης μπορεί να έχει οποιοδήποτε όνομα θέλετε, αλλά το πιο συνηθισμένο είναι το `args`. Εδώ φαίνεται πως δουλεύει η συνάρτηση :

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

Το σύνολο της συσσώρευσης είναι σκεδάσιμο, μπορεί δηλαδή να σπάσει στα επιμέρους στοιχεία. Αν έχετε μία ακολουθία από τιμές και θέλετε να την περάσετε σε μία συνάρτηση σαν πολλαπλά ορίσματα, μπορείτε να χρησιμοποιήσετε τον τελεστή `*`. Για παράδειγμα, η `divmod` παίρνει ακριβώς δύο ορίσματα, δεν δουλεύει με μία πλειάδα :

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

Αλλά αν διαμερίσετε την πλειάδα, δουλεύει :

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

Άσκηση 12.1. *Πολλές από τις ενσωματωμένες συναρτήσεις χρησιμοποιούν μεταβλητού μήκους όρισμα πλειάδας. Για παράδειγμα, η* `max` *και η* `min` *μπορούν να πάρουν οποιοδήποτε αριθμό ορισμάτων :*

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

Η `sum` *όμως όχι :*

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

Γράψτε μία συνάρτηση με όνομα `sumall` *η οποία θα παίρνει οποιοδήποτε πλήθος ορισμάτων και θα επιστρέφει το άθροισμά τους.*

12.5 Λίστες και πλειάδες

Η `zip` είναι μία ενσωματωμένη συνάρτηση η οποία παίρνει δύο ή περισσότερες ακολουθίες και τις ” ζιπάρει ” σε μία λίστα πλειάδων όπου η κάθε μία πλειάδα περιέχει ένα στοιχείο από κάθε ακολουθία. Στην Python 3, η `zip` επιστρέφει ένα επαναλήπτη πλειάδων, αλλά τις περισσότερες φορές συμπεριφέρεται σαν μία λίστα.

Αυτό το παράδειγμα ζιπάρει μία συμβολοσειρά και μία λίστα :

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

Το αποτέλεσμα είναι μία λίστα από πλειάδες όπου κάθε πλειάδα περιέχει ένα χαρακτήρα από τη συμβολοσειρά και το αντίστοιχο στοιχείο από τη λίστα.

Αν οι ακολουθίες δεν έχουν το ίδιο μήκος, το αποτέλεσμα έχει το μήκος της μικρότερης.

```
>>> zip('Anne', 'Elk')
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

Μπορείτε να χρησιμοποιήσετε εκχώρηση πλειάδας σε ένα βρόγχο `for` για να διασχίσετε μία λίστα πλειάδων :

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

Σε κάθε επανάληψη, η Python επιλέγει την επόμενη πλειάδα μέσα στη λίστα και εκχωρεί τα στοιχεία στη `letter` και στη `number`. Η έξοδος αυτού του βρόγχου είναι :

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

Αν συνδυάσετε την `zip`, την `for` και την εκχώρηση πλειάδας, τότε προκύπτει ένα χρήσιμο ιδίωμα για διάσχιση δύο (ή περισσότερων) ακολουθιών ταυτόχρονα. Για παράδειγμα, η `has_match` παίρνει δύο ακολουθίες, την `t1` και την `t2`, και επιστρέφει `True` αν υπάρχει δείκτης τέτοιος ώστε `t1[i] == t2[i]`:

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

Αν θέλετε να διασχίσετε τα στοιχεία μίας ακολουθίας μαζί με τους δείκτες τους, μπορείτε να χρησιμοποιήσετε την ενσωματωμένη συνάρτηση `enumerate`:

```
for index, element in enumerate('abc'):
    print index, element
```

Η έξοδος αυτού του βρόγχου είναι :

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

Ξανά.

12.6 Λεξικά και πλειάδες

Τα λεξικά έχουν μία μέθοδο με όνομα `items` η οποία επιστρέφει μία λίστα από πλειάδες, όπου η κάθε πλειάδα είναι ένα ζευγάρι κλειδιού-τιμής.

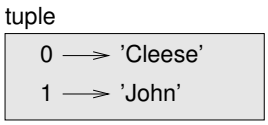
```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> print t
[('a', 0), ('c', 2), ('b', 1)]
```

Όπως θα περιμένατε από ένα λεξικό, τα αντικείμενα δεν είναι ταξινομημένα. Στην Python 3 η `items` επιστρέφει έναν επαναλήπτη, αλλά συνήθως, οι επαναλήπτες συμπεριφέρονται όπως οι λίστες.

Πηγαίνοντας αντίστροφα, μπορείτε να χρησιμοποιήσετε μία λίστα πλειάδων για να αρχικοποιήσετε ένα νέο λεξικό :

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

Συνδυάζοντας την `dict` με την `zip` προκύπτει ένας συνοπτικός τρόπος για τη δημιουργία ενός λεξικού :



Σχήμα 12.1: Διάγραμμα κατάστασης.

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

Η μέθοδος update των λεξικών μπορεί επίσης να πάρει μία λίστα από πλειάδες και να τις προσθέσει, σαν ζευγάρια κλειδιού-τιμής, σε ένα υπάρχον λεξικό.

Συνδυάζοντας την items, την εκχώρηση πλειάδων την for, προκύπτει το ιδίωμα για διάσχιση τόσο των κλειδιών όσο και των τιμών ενός λεξικού :

```
for key, val in d.items():
    print val, key
```

Η έξοδος αυτού του βρόγχου είναι :

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

Πάλι.

Είναι συνηθισμένο να χρησιμοποιούνται πλειάδες σαν κλειδιά μέσα σε ένα λεξικό, πρωτίστως γιατί δεν μπορούμε να χρησιμοποιήσουμε λίστες. Για παράδειγμα, ένας τηλεφωνικός κατάλογος μπορεί να αντιστοιχεί ζευγάρια επωνύμου, ονόματος σε τηλεφωνικούς αριθμούς. Θεωρώντας ότι έχουμε ορίσει την last, την first και την number, θα μπορούσαμε να γράψουμε :

```
directory[last,first] = number
```

Η έκφραση μέσα στις αγκύλες είναι μία πλειάδα. Θα μπορούσαμε να χρησιμοποιήσουμε την εκχώρηση πλειάδων για να διασχίσουμε αυτό το λεξικό.

```
for last, first in directory:
    print first, last, directory[last,first]
```

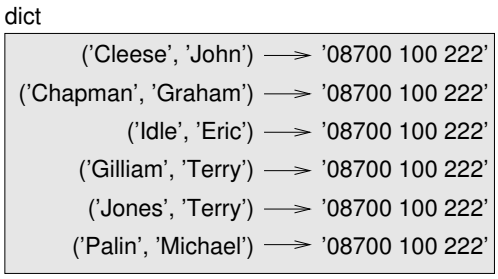
Αυτός ο βρόχος διασχίζει τα κλειδιά μέσα στο directory, τα οποία είναι πλειάδες. Εκχωρεί τα στοιχεία της κάθε πλειάδας στην last και στην first, και μετά εμφανίζει το ονοματεπώνυμο με τον αντίστοιχο τηλεφωνικό αριθμό.

Υπάρχουν δύο τρόποι να αναπαραστήσουμε πλειάδες σε ένα διάγραμμα κατάστασης. Η πιο λεπτομερής εκδοχή δείχνει τους δείκτες και τα στοιχεία ακριβώς όπως εμφανίζονται μέσα σε μία λίστα. Για παράδειγμα, η πλειάδα ('Cleese', 'John') θα φαίνεται όπως στην Εικόνα 12.1.

Αλλά σε ένα μεγαλύτερο διάγραμμα μπορεί να θέλετε να παραλήψετε τις λεπτομέρειες. Για παράδειγμα, ένα διάγραμμα του τηλεφωνικού καταλόγου θα μπορούσε να είναι όπως αυτό στην Εικόνα 12.2.

Εδώ οι πλειάδες εμφανίζονται χρησιμοποιώντας το συντακτικό της Python σαν μία γραφική στενογραφία.

Ο τηλεφωνικός αριθμός μέσα στο διάγραμμα είναι η γραμμή παραπόνων για το BBC, οπότε παρακαλώ μην τον καλέσετε.



Σχήμα 12.2: Διάγραμμα κατάστασης

12.7 Συγκρίνοντας πλειάδες

Οι σχεσιακοί τελεστές δουλεύουν με τις πλειάδες και άλλες ακολουθίες. Η Python ξεκινάει συγκρίνοντας τα πρώτα στοιχεία κάθε ακολουθίας. Αν είναι ίσα, πηγαίνει στα επόμενα, και ούτω καθεξής, μέχρι να βρει στοιχεία τα οποία είναι διαφορετικά. Τα ψηφία που ακολουθούν δεν λαμβάνονται υπόψιν (ακόμα και αν είναι πραγματικά μεγάλα).

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

Η συνάρτηση `sort` δουλεύει με το ίδιο τρόπο. Ταξινομεί αρχικά με βάση το πρώτο ψηφίο, αλλά σε περίπτωση ισότητας ταξινομεί με το δεύτερο και ούτω καθεξής.

Αυτό το χαρακτηριστικό δανείζει τον εαυτό του σε ένα πρότυπο με όνομα **DSU**:

Decorate Διακοσμούμε μία ακολουθία δημιουργώντας μία λίστα από πλειάδες με ένα η περισσότερα κλειδιά ταξινόμησης να προηγούνται των στοιχείων της ακολουθίας,

Sort ταξινομούμε τη λίστα πλειάδων, και

Undecorate ” αποδιακοσμούμε ” εξάγοντας τα ταξινομημένα στοιχεία από την ακολουθία.

Για παράδειγμα, υποθέστε ότι έχετε μία λίστα από λέξεις και θέλετε να τις ταξινομήσετε από την μεγαλύτερη προς τη μικρότερη :

```
def sort_by_length(words):
    t = []
    for word in words:
       t.append((len(word), word))

    t.sort(reverse=True)

    res = []
    for length, word in t:
        res.append(word)
    return res
```

Ο πρώτος βρόχος δημιουργεί μία λίστα από πλειάδες, όπου η κάθε πλειάδα είναι μία λέξη και το αντίστοιχο μήκος να προηγείται αυτής.

Η `sort` συγκρίνει το αρχικά το πρώτο ψηφίο, δηλαδή το μήκος, και εξετάζει το δεύτερο μόνο σε περιπτώσεις ισοψηφίας. Το όρισμα `reverse=True` λέει στη `sort` να ακολουθήσει φθίνουσα σειρά.

Ο δεύτερος βρόχος διασχίζει τη λίστα πλειάδων και δημιουργεί μία λίστα από λέξεις σε φθίνουσα σειρά με βάση το μήκος.

Άσκηση 12.2. *Σε αυτό το παράδειγμα, σε περίπτωση ισοψηφίας γίνεται σύγκριση των λέξεων, άρα οι λέξεις με ίδιο μήκος εμφανίζονται σε αντίστροφη αλφαβητική σειρά. Σε άλλες εφαρμογές όμως, ίσως προτιμούσατε να σπάσετε την ισοψηφία τυχαία. Τροποποιήστε αυτό το παράδειγμα έτσι ώστε οι λέξεις με το ίδιο μήκος να ταξινομούνται σε τυχαία σειρά. Υπόδειξη : δείτε την συνάρτηση* `random` *στη μονάδα λογισμικού* `random`*. Λύση :* `http://thinkpython.com/code/unstable_sort.py`*.*

12.8 Ακολουθίες ακολουθιών

Έχω επικεντρωθεί σε λίστες πλειάδων, αλλά σχεδόν όλα τα παραδείγματα αυτού του κεφαλαίου δουλεύουν επίσης με λίστες λιστών, πλειάδες πλειάδων και πλειάδες λιστών. Για να αποφεύγουμε την αρίθμηση των δυνατών συνδυασμών, είναι ευκολότερο να μιλάμε για ακολουθίες ακολουθιών.

Σε πολλές περιπτώσεις, τα διαφορετικά είδη ακολουθιών (συμβολοσειρές, λίστες και πλειάδες) μπορούν να χρησιμοποιηθούν εναλλάξ. Επομένως πως και γιατί να επιλέξετε κάποια έναντι των άλλων ;

Αρχίζοντας με το προφανές, οι συμβολοσειρές είναι πιο περιορισμένες σε σχέση με τις υπόλοιπες ακολουθίες επειδή τα στοιχεία πρέπει να είναι χαρακτήρες. Και επίσης είναι αμετάβλητες. Αν χρειαζόσασταν την δυνατότητα να αλλάξετε τους χαρακτήρες σε μία συμβολοσειρά (σε αντίθεση με την δημιουργία μίας νέας συμβολοσειράς), θα ήταν προτιμότερο να χρησιμοποιήσετε μία λίστα από χαρακτήρες αντ'αυτού.

Οι λίστες είναι πιο συνηθισμένες από τις πλειάδες, κυρίως γιατί είναι έχουν τη δυνατότητα να μεταβληθούν. Αλλά υπάρχουν μερικές περιπτώσεις που μάλλον θα προτιμούσατε τις πλειάδες :

1. Σε μερικές περιπτώσεις, όπως σε μία δήλωση `return`, είναι συντακτικά απλούστερο να δημιουργήσετε μία λίστα αντί μία πλειάδα. Σε άλλες περιπτώσεις ίσως προτιμήσετε μία λίστα.
2. Αν θέλατε να χρησιμοποιήσετε μία ακολουθία σαν κλειδί ενός λεξικού, πρέπει να χρησιμοποιήσετε έναν αμετάβλητο τύπο όπως μία πλειάδα ή μία συμβολοσειρά.
3. Η χρήση πλειάδων, όταν θέλετε να περάσετε μία ακολουθία σαν όρισμα σε μία συνάρτηση, μειώνει τις πιθανότητες απροσδόκητης συμπεριφοράς λόγω ψευδωνύμων.

Επειδή οι πλειάδες είναι αμετάβλητες, δεν παρέχουν μεθόδους όπως η `sort` και `reverse`, οι οποίες τροποποιούν ήδη υπάρχουσες λίστες. Αλλά η Python παρέχει ενσωματωμένες τις συναρτήσεις `sorted` και `reversed`, οι οποίες παίρνουν σαν παράμετρο οποιαδήποτε ακολουθία και επιστρέφουν μία νέα λίστα με τα ίδια στοιχεία σε διαφορετική σειρά.

12.9 Αποσφαλμάτωση

Οι λίστες, τα λεξικά και οι πλειάδες είναι γνωστές γενικά σαν " δομές δεδομένων ". Σε αυτό το κεφάλαιο αρχίσαμε να βλέπουμε σύνθετες δομές δεδομένων, όπως λίστες πλειάδων και λεξικά τα οποία περιέχουν πλειάδες σαν κλειδιά και λίστες σαν τιμές. Οι σύνθετες δομές δεδομένων είναι πολύ χρήσιμες αλλά είναι επιρρεπείς σε αυτό που εγώ ονομάζω " σφάλματα μορφής ", λάθη δηλαδή που προκαλούνται όταν μία δομή δεδομένων έχει λάθος τύπο, μέγεθος ή σύνθεση. Για παράδειγμα, αν περιμένετε μία λίστα με έναν ακέραιο και εγώ σας δώσω έναν απλό ακέραιο αριθμό (όχι μέσα σε λίστα), δεν θα δουλέψει.

Για να σας βοηθήσω να αποσφαλματώσετε αυτού του είδους τα σφάλματα, έχω γράψει μία μονάδα λογισμικού με όνομα `structshape`, η οποία παρέχει μία συνάρτηση με το ίδιο όνομα, η οποία παίρνει οποιοδήποτε είδος δομής δεδομένων και σαν όρισμα και επιστρέφει μία συμβολοσειρά η οποία συνοψίζει τη μορφή της. Μπορείτε να την κατεβάσετε από εδώ : `http://thinkpython.com/code/structshape.py`.

Αυτό είναι το αποτέλεσμα για μία απλή λίστα :

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> print structshape(t)
list of 3 int
```

Ένα κομψότερο πρόγραμμα ίσως έγραφε "list of 3 ints," αλλά είναι ευκολότερο να μην ασχοληθούμε πληθυντικούς. Αυτή είναι μία λίστα από λίστες :

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

Αν τα στοιχεία της λίστας δεν είναι του ίδιου τύπου, η `structshape` τα ομαδοποιεί σε σειρά με βάση τον τύπο :

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

Αυτή είναι μία λίστα πλειάδων :

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

Και αυτό είναι ένα λεξικό με 3 στοιχεία τα οποία αντιστοιχούν ακεραίους σε συμβολοσειρές :

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

Αν αντιμετωπίζετε προβλήματα στην παρακολούθηση των δομών δεδομένων σας, τότε η `structshape` μπορεί να σας βοηθήσει.

12.10 Ορολογία

πλειάδα: Μία αμετάβλητη ακολουθία στοιχείων.

εκχώρηση πλειάδας: Μία εκχώρηση με μία ακολουθία στο δεξιό μέρος και μία πλειάδα μεταβλητών στα αριστερά. Αποτιμάται το δεξιό μέρος και στη συνέχεια εκχωρούνται τα στοιχεία του στις μεταβλητές του αριστερού μέρους.

συσσώρευση: Η διαδικασία σύνθεσης ενός μεταβλητού μήκους ορίσματος πλειάδας.

σκέδαση: Η διαδικασία κατά την οποία μεταχειριζόμαστε μία ακολουθία σαν μία λίστα ορισμάτων.

DSU: Ακρωνύμιο των λέξεων "decorate-sort-undecorate," ένα πρότυπο το οποίο αφορά τη δημιουργία μίας λίστας πλειάδων, την ταξινόμηση και την εξαγωγή μέρους του αποτελέσματος.

δομή δεδομένων: Μία συλλογή συναφών μεταβλητών, οργανωμένες συνήθως σε λίστες, λεξικά, πλειάδες κτλ.

μορφή (μίας δομής δεδομένων): Μία σύνοψη του τύπου, του μεγέθους και της σύνθεσης μίας δομής δεδομένων.

12.11 Ασκήσεις

Άσκηση 12.3. *Γράψτε μία συνάρτηση με όνομα* `most_frequent` *η οποία θα παίρνει μία συμβολοσειρά και θα εμφανίζει τα γράμματα σε φθίνουσα σειρά με βάση τη συχνότητα. Βρείτε δείγματα κειμένου από διάφορες γλώσσες και δείτε πως η συχνότητα των γραμμάτων ποικίλλει μεταξύ των λέξεων. Συγκρίνετε τα αποτελέσματά σας με τους πίνακες στον σύνδεσμο* `http://en.wikipedia.org/wiki/Letter_frequencies`*. Λύση :* `http://thinkpython.com/code/most_frequent.py`*.*

Άσκηση 12.4. *Περισσότεροι αναγραμματισμοί!*

1. *Γράψτε ένα πρόγραμμα το οποίο θα διαβάζει μία λίστα λέξεων από ένα αρχείο (δείτε την Ενότητα 9.1) και θα εμφανίζει όλα τα σύνολα των λέξεων τα οποία είναι αναγραμματισμοί.*

   Αυτό είναι ένα παράδειγμα που δείχνει πως περίπου θα πρέπει να είναι η έξοδος :

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

   Σημείωση : Ίσως θέλατε να φτιάξετε ένα λεξικό το οποίο να αντιστοιχεί ένα σύνολο γραμμάτων σε μία λίστα λέξεων η οποία θα μπορεί να γραφτεί με αυτά τα γράμματα. Το ερώτημα είναι, πως μπορείτε να αναπαραστήσετε το σύνολο των γραμμάτων με τέτοιο τρόπο ώστε να μπορεί να χρησιμοποιηθεί σαν κλειδί.

2. *Τροποποιήστε το προηγούμενο πρόγραμμα έτσι ώστε να εμφανίζει το μεγαλύτερο σύνολο αναγραμματισμών πρώτο, ακολουθούμενο από το δεύτερο μεγαλύτερο και ούτω καθεξής.*

3. *Στο Σκραμπλ, έχετε πετύχει "bingo" όταν έχετε παίξει και τα εφτά γράμματα που έχετε σε συνδυασμό με ένα γράμμα στο ταμπλό σχηματίζοντας μία λέξη οκτώ γραμμάτων(στο αμερικάνικο σκραμπλ). Ποιο σύνολο 8 γραμμάτων σχηματίζει τα πιο πιθανά "bingos"; Σημείωση : Υπάρχουν επτά.*

   Λύση : `http://thinkpython.com/code/anagram_sets.py`*.*

Άσκηση 12.5. *Δύο λέξεις αποτελούν ένα " ζεύγος μετάθεσης " αν μπορείτε να σχηματίσετε τη μία από την άλλη μεταθέτοντας δύο γράμματα. Για παράδειγμα, η "converse", και η "conserve" είναι ένα τέτοιο ζεύγος. Γράψτε ένα πρόγραμμα το οποίο θα βρίσκει όλα τα ζεύγη μετάθεσης σε ένα λεξικό. Σημείωση : μην ελέγξετε όλα τα ζευγάρια των λέξεων και μην δοκιμάσετε όλες τις πιθανές μεταθέσεις. Λύση :* `http://thinkpython.com/code/metathesis.py`*. Αναφορά : Αυτή η άσκηση είναι εμπνευσμένη από ένα παράδειγμα στο σύνδεσμο* `http://puzzlers.org`*.*

Άσκηση 12.6. *Αυτός είναι ένας ακόμα γρίφος από την εκπομπή Car Talk (*`http://www.cartalk.com/content/puzzlers`*):*

> *Ποια είναι η μεγαλύτερη αγγλική λέξη η οποία παραμένει έγκυρη αγγλική λέξη όσο αφαιρείτε ένα ένα τα γράμματά της;*

Μπορείτε να αφαιρείτε γράμματα είτε από το τέλος είτε από τη μέση αλλά δεν μπορείτε να αναδιατάξετε κανένα από τα γράμματα. Κάθε φορά που βγάζετε ένα γράμμα, προκύπτει μία άλλη αγγλική λέξη. Αν το κάνετε αυτό, θα καταλήξετε με ένα γράμμα το οποίο θα είναι επίσης μία αγγλική λέξη, από τη στιγμή που υπάρχει μέσα στο λεξικό. Θέλω να μάθω ποια είναι είναι η μεγαλύτερη λέξη και πόσα γράμματα έχει.

Θα σας δώσω ένα μικρό παράδειγμα, τη λέξη Sprite. Ξεκινάτε με τη sprite, αφαιρείτε ένα γράμμα, ένα από το εσωτερικό της λέξης, βγάλτε το r, και προκύπτει η λέξη spite, στη συνέχεια βγάζουμε το e από το τέλος και προκύπτει η λέξη spit, βγάζουμε το s, προκύπτει η pit, it και I.

Γράψτε ένα πρόγραμμα για να βρείτε όλες τις λέξεις οι οποίες μπορούν να ελαττωθούν κατ' αυτόν τον τρόπο και μετά βρείτε τη μεγαλύτερη.

Αυτή η άσκηση είναι λίγο δυσκολότερη από τις υπόλοιπες, για αυτό θα σας δώσω μερικές υποδείξεις :

1. *Καλό θα ήταν να γράφατε μία συνάρτηση η οποία θα παίρνει μία λέξη και θα υπολογίζει μία λίστα με όλες τις λέξεις η οποίες σχηματίζονται αφαιρώντας ένα γράμμα. Αυτές είναι τα " παιδιά " της λέξης.*
2. *Αναδρομικά, μία λέξη είναι αναγώγιμη αν κάποιο από τα παιδιά της είναι αναγώγιμο. Σαν περίπτωση βάσης, μπορείτε να θεωρήσετε την κενή συμβολοσειρά αναγώγιμη.*
3. *Η λίστα λέξεων που παρέχω,* `words.txt`*, δεν περιέχει λέξεις μεμονωμένων γραμμάτων. Επομένως καλό θα ήταν να προσθέτατε το "I", "a" και την κενή συμβολοσειρά.*
4. *Για να βελτιώσετε την απόδοση του προγράμματός σας χρησιμοποιήστε σημειώματα για τις λέξεις οι οποίες είναι γνωστό ότι είναι αναγώγιμες.*

Λύση : `http://thinkpython.com/code/reducible.py`.

Κεφάλαιο 13

Μελέτη περίπτωσης: επιλογή δομής δεδομένων

13.1 Ανάλυση συχνότητας λέξεων

Ως συνήθως, Θα πρέπει τουλάχιστον να προσπαθήσετε τις ακόλουθες ασκήσεις προτού διαβάσετε τις λύσεις μου.

Άσκηση 13.1. *Γράψτε ένα πρόγραμμα το οποίο θα διαβάζει ένα αρχείο, θα σπάει κάθε γραμμή σε λέξεις, θα αφαιρεί τους χαρακτήρες λευκού διαστήματος και τη στίξη από τις λέξεις και θα μετατρέπει όλους τους χαρακτήρες σε πεζούς.*

Σημείωση : Η μονάδα λογισμικού `string` *παρέχει τις συμβολοσειρές* `whitespace` *η οποία περιέχει το κενό, την εσοχή κειμένου, τη νέα γραμμή, κτλ., και την* `punctuation` *η οποία περιέχει τα σημεία στίξης. Ας δούμε αν μπορούμε να κάνουμε την Python να βρίσει :*

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

Επίσης, μπορείτε να χρησιμοποιήσετε της μεθόδους συμβολοσειρών `strip`, `replace` *και* `translate`.

Άσκηση 13.2. *Επισκεφθείτε το Project Gutenberg στο σύνδεσμο (`gutenberg.org`) και κατεβάστε το αγαπημένο σας βιβλίο δημόσιας κυριότητας σε μορφή απλού κειμένου.*

Τροποποιήστε το πρόγραμμά σας από την προηγούμενη άσκηση ούτως ώστε να διαβάζει το βιβλίο που κατεβάσατε, πηδήξτε τις πληροφορίες της επικεφαλίδας στην αρχή του βιβλίου, και επεξεργαστείτε τις υπόλοιπες λέξεις όπως και πριν.

Στη συνέχεια, τροποποιήστε το πρόγραμμα έτσι ώστε να μετράει το συνολικό αριθμό των λέξεων στο βιβλίο, και το πλήθος των φορών που χρησιμοποιείται κάθε λέξη.

Εμφανίστε τον αριθμό των διαφορετικών λέξεων που χρησιμοποιούνται μέσα στο βιβλίο. Συγκρίνετε διαφορετικά βιβλία από διαφορετικούς συγγραφείς, γραμμένα σε διαφορετικές εποχές. Ποιος συγγραφέας χρησιμοποιεί το πιο εκτενές λεξιλόγιο ;

Άσκηση 13.3. *Τροποποιήστε το πρόγραμμα της προηγούμενης άσκησης ώστε να εμφανίζει τις 20 λέξεις που χρησιμοποιούνται πιο συχνά μέσα στο βιβλίο.*

Άσκηση 13.4. *Τροποποιήστε το προηγούμενο πρόγραμμα ώστε να διαβάζει μία λίστα λέξεων (βλ. Ενότητα 9.1) και στη συνέχεια εμφανίστε όλες τις λέξεις του βιβλίου οι οποίες δεν βρίσκονται μέσα*

στη λίστα. Πόσες από αυτές έχουν ορθογραφικά λάθη ; Πόσες από αυτές είναι συνηθισμένες λέξεις και θα έπρεπε να βρίσκονται μέσα στη λίστα και πόσες από αυτές είναι πραγματικά δυσνόητες ;

13.2 Τυχαίοι αριθμοί

Δοθέντων των ίδιων εισόδων, τα περισσότερα προγράμματα υπολογιστών παράγουν τις ίδιες εξόδους κάθε φορά, επομένως λέμε ότι είναι ντετερμινιστικά. Ο ντετερμινισμός είναι συνήθως κάτι καλό, αφού αναμένουμε ότι ο ίδιος υπολογισμός θα αποδώσει το ίδιο αποτέλεσμα. Για μερικές εφαρμογές εν τούτοις, θέλουμε ο υπολογιστής να είναι απρόβλεπτος. Τα παιχνίδια είναι ένα πασιφανές παράδειγμα αλλά υπάρχουν και άλλα.

Δεν είναι εύκολο να κάνουμε ένα πρόγραμμα πραγματικά μη ντετερμινιστικό, αλλά υπάρχουν τρόποι να το κάνουμε τουλάχιστον να μοιάζει μη ντετερμινιστικό. Ένας από αυτούς είναι να χρησιμοποιήσουμε αλγόριθμους οι οποίοι παράγουν ψευδοτυχαίους αριθμούς. Οι ψευδοτυχαίοι αριθμοί δεν είναι πραγματικά τυχαίοι γιατί παράγονται από έναν ντετερμινιστικό υπολογισμό, αλλά απλά κοιτώντας τους είναι σχεδόν αδύνατον να τους ξεχωρίσουμε από κάποιους πραγματικά τυχαίους.

Η μονάδα λογισμικού `random` παρέχει συναρτήσεις η οποίες παράγουν ψευδοτυχαίους αριθμούς (τους οποίους από εδώ και πέρα θα τους αποκαλώ τυχαίους).

Η συνάρτηση `random` επιστρέφει έναν τυχαίο δεκαδικό ανάμεσα στο 0.0 και στο 1.0 (συμπεριλαμβανομένου του 0.0 αλλά όχι του 1.0). Κάθε φορά που καλείτε τη `random`, σε μία σειρά. Για να δείτε ένα δείγμα, τρέξτε αυτόν το βρόχο :

```
import random

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

Η συνάρτηση `randint` παίρνει παίρνει σαν παραμέτρους την `low` και την `high` και επιστρέφει έναν ακέραιο μεταξύ της `low` και της `high` (συμπεριλαμβανομένων και των δύο).

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

Για να διαλέξετε ένα στοιχείο από μία ακολουθία τυχαία, μπορείτε να χρησιμοποιήσετε την `choice`:

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

Η μονάδα λογισμικού `random` παρέχει επίσης συναρτήσεις για να παράγει τυχαίες τιμές από συνεχείς κατανομές συμπεριλαμβανομένων της Γκαουσιανής, της εκθετικής, της γάμμα και μερικών ακόμα.

Άσκηση 13.5. .

Γράψτε μία συνάρτηση με όνομα `choose_from_hist` *η οποία παίρνει ένα ιστόγραμμα όπως αυτό ορίζεται στην Ενότητα 11.1 και επιστρέφει μία τυχαία τιμή από το ιστόγραμμα, επιλεγμένη με πιθανότητα σε αναλογία με τη συχνότητα. Για παράδειγμα, για αυτό το ιστόγραμμα :*

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> print hist
{'a': 2, 'b': 1}
```

η συνάρτησή σας θα πρέπει να επιστρέψει `'a'` *με πιθανότητα* $2/3$ *και* `'b'` *με πιθανότητα* $1/3$.

13.3 Ιστόγραμμα λέξεων

Θα πρέπει να προσπαθήσετε τις προηγούμενες ασκήσεις προτού προχωρήσετε. Μπορείτε να κατεβάσετε τη λύση μου από το σύνδεσμο : `http://thinkpython.com/code/analyze_book.py`. Θα χρειαστείτε επίσης και το `http://thinkpython.com/code/emma.txt`.

Αυτό είναι ένα πρόγραμμα το οποίο διαβάζει ένα αρχείο και φτιάχνει ένα ιστόγραμμα των λέξεων που περιέχει :

```
import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist

def process_line(line, hist):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()

        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

Αυτό το πρόγραμμα διαβάζει το `emma.txt`, το οποίο περιέχει το κείμενο από το μυθιστόρημα ” Έμμα ” της Τζέιν Ώστιν.

Η `process_file` διασχίζει τις γραμμές του αρχείου περνώντας τες μία μία στην `process_line`. Το ιστόγραμμα `hist` χρησιμοποιείται σαν ένας συσσωρευτής.

Η `process_line` χρησιμοποιεί τη μέθοδο συμβολοσειρών `replace` για να αντικαταστήσει τις παύλες με κενά πριν χρησιμοποιήσει την την `split` για να σπάσει τη γραμμή σε μία λίστα συμβολοσειρών. Διασχίζει τη λίστα των λέξεων και χρησιμοποιεί τη `strip` και τη `lower` για να αφαιρέσει τη στίξη και να μετατρέψει τα γράμματα σε πεζά. (Χάρη συντομίας λέμε ότι οι συμβολοσειρές μετατρέπονται αλλά θυμηθείτε ότι οι συμβολοσειρές είναι αμετάβλητες, επομένως οι μέθοδοι όπως η `strip` και η `lower` επιστρέφουν νέες συμβολοσειρές).

Και τέλος, η `process_line` ενημερώνει το ιστόγραμμα δημιουργώντας ένα νέο στοιχείο ή προσαυξάνοντας ένα υπάρχον.

Για να υπολογίσουμε το συνολικό αριθμό των λέξεων σε ένα αρχείο, μπορούμε να προσθέσουμε τις συχνότητες του ιστογράμματος :

```
def total_words(hist):
    return sum(hist.values())
```

Το πλήθος των διαφορετικών λέξεων είναι ίσο με το πλήθος των στοιχείων μέσα στο λεξικό :

```
def different_words(hist):
    return len(hist)
```

Με αυτόν τον κώδικα εμφανίζουμε τα αποτελέσματα :

```
print 'Total number of words:', total_words(hist)
print 'Number of different words:', different_words(hist)
```

Και τα αποτελέσματα είναι :

```
Total number of words: 161080
Number of different words: 7214
```

13.4 Οι πιο συνηθέστερες λέξεις

Για να βρούμε τις συνηθέστερες λέξεις, μπορούμε να εφαρμόσουμε το πρότυπο DSU. Η `most_common` παίρνει ένα ιστόγραμμα και επιστρέφει μία λίστα από πλειάδες συχνοτήτων λέξεων, ταξινομημένες σε αντίστροφη σειρά με βάση τη συχνότητα :

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

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

Αυτός ο βρόχος εμφανίζει τις 10 συνηθέστερες λέξεις :

```
t = most_common(hist)
print 'The most common words are:'
for freq, word in t[0:10]:
    print word, '\t', freq
```

Και αυτά είναι τα αποτελέσματα από την *Emma*:

```
The most common words are:
to  5242
the  5205
and  4897
of  4295
i  3191
a  3130
it  2529
her  2483
was  2400
she  2364
```

13.5 Προαιρετικές παράμετροι

Έχουμε δει ενσωματωμένες συναρτήσεις και μεθόδους οι οποίες παίρνουν ένα μεταβλητό αριθμό παραμέτρων. Μπορούμε να γράψουμε τέτοιες συναρτήσεις οριζόμενες από το χρήστη με προαιρετικά ορίσματα. Για παράδειγμα, αυτή είναι μία συνάρτηση η οποία εμφανίζει τις συνηθέστερες λέξεις σε ένα ιστόγραμμα :

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print 'The most common words are:'
    for freq, word in t[:num]:
        print word, '\t', freq
```

Η πρώτη παράμετρος είναι απαραίτητη, η δεύτερη είναι προαιρετική. Η προεπιλεγμένη τιμή της num είναι 10.

Αν παρέχετε μόνο ένα όρισμα :

```
print_most_common(hist)
```

η num παίρνει την προεπιλεγμένη τιμή. Αν παρέχετε δύο ορίσματα :

```
print_most_common(hist, 20)
```

η num παίρνει την τιμή του ορίσματος αντ' αυτού. Με άλλα λόγια, το προαιρετικό όρισμα παρακάμπτει την προεπιλεγμένη τιμή.

Αν η συνάρτηση έχει και απαιτούμενες και προαιρετικές παραμέτρους, όλες οι απαιτούμενες πρέπει να προηγούνται των προαιρετικών.

13.6 Αφαίρεση λεξικών

Η εύρεση των λέξεων του βιβλίου οι οποίες δεν υπάρχουν στη λίστα των λέξεων από το `word.txt` είναι ένα πρόβλημα το οποίο θα αναγνωρίζατε ίσως σαν αφαίρεση συνόλων. Ήτοι, θέλουμε να βρούμε όλες τις λέξεις από το ένα σύνολο (τις λέξεις του βιβλίου) οι οποίες δεν υπάρχουν σε ένα άλλο σύνολο (τις λέξεις μέσα στη λίστα).

Η `subtract` παίρνει δύο λεξικά, το `d1` και το `d2` και επιστρέφει ένα νέο λεξικό το οποίο περιέχει όλα τα κλειδιά από το `d1` τα οποία δεν υπάρχουν στο `d2`. Από τη στιγμή που δεν ενδιαφερόμαστε πραγματικά για τις τιμές, τις θέτουμε όλες ίσες με None.

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

Για να βρούμε τις λέξεις του βιβλίου οι οποίες δεν υπάρχουν μέσα στο `words.txt`, μπορούμε να χρησιμοποιήσουμε την `process_file` για να φτιάξουμε ένα ιστόγραμμα για το `words.txt`, και μετά να αφαιρέσουμε :

```
words = process_file('words.txt')
diff = subtract(hist, words)

print "The words in the book that aren't in the word list are:"
```

```
for word in diff.keys():
    print word,
```

Αυτά είναι κάποια από τα αποτελέσματα για την *Emma*:

```
The words in the book that aren't in the word list are:
 rencontre jane's woodhouses disingenuousness
friend's venice apartment ...
```

Κάποιες από αυτές τις λέξεις είναι ονόματα και και κτητικές αντωνυμίες. Άλλες, όπως η "rencontre", δεν χρησιμοποιούνται ευρέως πλέον. Αλλά μερικές είναι λέξεις οι οποίες χρησιμοποιούνται συχνά και θα έπρεπε πραγματικά να υπάρχουν μέσα στη λίστα.

Άσκηση 13.6. *Η Python παρέχει μία δομή δεδομένων με όνομα* `set` *η οποία με τη σειρά της παρέχει πολλές από τις συνηθέστερες πράξεις συνόλων. Διαβάστε την τεκμηρίωση στο σύνδεσμο* `http://docs.python.org/2/library/stdtypes.html#types-set` *και γράψτε ένα πρόγραμμα το οποίο θα χρησιμοποιεί την αφαίρεση συνόλων για να βρει τις λέξεις του βιβλίου οι οποίες δεν υπάρχουν στη λίστα λέξεων. Λύση :* `http://thinkpython.com/code/analyze_book2.py`.

13.7 Τυχαίες λέξεις

Για να διαλέξετε μία τυχαία λέξη από το ιστόγραμμα, ο απλούστερος αλγόριθμος είναι να φτιάξουμε μία λίστα με πολλαπλά αντίγραφα της κάθε λέξης, σύμφωνα με τη παρατηρηθείσα συχνότητα, και στη συνέχεια να διαλέξετε από τη λίστα :

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

    return random.choice(t)
```

Η έκφραση `[word] * freq` δημιουργεί μία λίστα με `freq` αντίγραφα της συμβολοσειράς `word`. Η μέθοδος `extend` είναι παρόμοια με την `append` με τη διαφορά ότι το όρισμα είναι μία ακολουθία.

Άσκηση 13.7. *Αυτός ο αλγόριθμος δουλεύει, αλλά δεν είναι πολύ αποδοτικός. Κάθε φορά που επιλέγετε μία τυχαία λέξη, ο αλγόριθμος ξαναφτιάχνει τη λίστα, η οποία είναι τόσο μεγάλη όσο και το πρωτότυπο βιβλίο. Μία προφανής βελτίωση είναι η δημιουργία της λίστας μία φορά και μετά να κάνετε πολλαπλές επιλογές, αλλά η λίστα είναι ακόμη μεγάλη.*

Μία εναλλακτική λύση είναι :

1. *Χρησιμοποιήστε* `keys` *για να πάρετε μία λίστα των λέξεων του βιβλίου.*
2. *Φτιάξτε μία λίστα η οποία θα περιέχει το συγκεντρωτικό άθροισμα των συχνοτήτων των λέξεων (βλ. ΑΣΚΗΣΗ 10.3). Το τελευταίο στοιχείο αυτής της λίστας θα είναι ο συνολικός αριθμός των λέξεων του βιβλίου, n.*
3. *Διαλέξτε έναν τυχαίο αριθμό από το 1 έως το n. Χρησιμοποιήστε μία δυαδική αναζήτηση (βλ. ΑΣΚΗΣΗ 10.11) για να βρείτε το δείκτη όπου θα εισαχθεί ο τυχαίο αριθμός στο συγκεντρωτικό άθροισμα.*
4. *Χρησιμοποιήστε το δείκτη για να βρείτε την αντίστοιχη λέξη μέσα στη λίστα των λέξεων.*

Γράψτε ένα πρόγραμμα το οποίο θα χρησιμοποιεί αυτόν τον αλγόριθμο για να επιλέξει μία τυχαία λέξη από το βιβλίο. Λύση : `http://thinkpython.com/code/analyze_book3.py`.

13.8 Ανάλυση Μαρκόφ

Αν επιλέξετε τυχαία λέξεις από το βιβλίο, μπορείτε να πιάσετε το νόημα του λεξιλογίου αλλά πιθανότατα δεν θα πάρετε μία πρόταση :

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

Σπάνια μία σειρά από λέξεις έχει νόημα γιατί δεν υπάρχει σχέση μεταξύ διαδοχικών λέξεων. Για παράδειγμα, σε μία πραγματική πρόταση θα περιμένατε ένα άρθρο όπως το "the" να ακολουθείται από έναν επίθετο ή ένα ουσιαστικό και πιθανώς όχι από κάποιο ρήμα ή επίρρημα.

Ένας τρόπος για τη μέτρηση αυτού του είδους των σχέσεων είναι η ανάλυση Μαρκόφ, η οποία χαρακτηρίζει, για μία δοθείσα ακολουθία λέξεων, την πιθανότητα της λέξης που ακολουθεί. Για παράδειγμα, το τραγούδι *Eric, the Half a Bee* αρχίζει :

> 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?

Σε αυτό το κείμενο, η φράση "half the" ακολουθείται πάντα από τη λέξη "bee", αλλά η φράση "the bee" μπορεί να ακολουθείται είτε από "has" είτε από "is".

Το αποτέλεσμα της ανάλυσης Μαρκόφ είναι μία αντιστοίχιση του κάθε προθέματος (όπως το "half the" και το "the bee") με όλα τα πιθανά επιθήματα (όπως είναι το "has" και το "is").

Δοσμένης αυτής της αντιστοίχισης, μπορείτε να δημιουργήσετε ένα τυχαίο κείμενο ξεκινώντας με ένα οποιοδήποτε πρόθεμα και διαλέγοντας στην τύχη κάποιο από τα πιθανά επιθήματα. Στη συνέχεια, μπορείτε να συνδυάσετε το τέλος του προθέματος και του νέου επιθήματος για να σχηματίσετε το νέο πρόθεμα και να επαναλάβετε.

Για παράδειγμα, αν ξεκινήσετε με το πρόθεμα "Half a" τότε η επόμενη λέξη πρέπει να είναι η "bee," επειδή το πρόθεμα εμφανίζεται μόνο μία φορά στο κείμενο. Το επόμενο πρόθεμα είναι το "a bee," άρα το επόμενο επίθημα θα μπορούσε να είναι το "philosophically," το "be" ή το "due".

Σε αυτό το παράδειγμα το μήκος του προθέματος είναι πάντα δύο, αλλά μπορείτε να κάνετε ανάλυση Μαρκόφ με οποιοδήποτε μήκος προθέματος. Το μήκος του επιθήματος ονομάζεται " τάξη " της ανάλυσης.

Άσκηση 13.8. *Ανάλυση Μαρκόφ :*

1. *Γράψτε ένα πρόγραμμα το οποίο θα διαβάζει ένα κείμενο από ένα αρχείο και θα εκτελεί ανάλυση Μαρκόφ. Το αποτέλεσμα θα πρέπει να είναι ένα λεξικό το οποίο θα αντιστοιχεί προθέματα σε μία συλλογή από πιθανά επιθήματα. Η συλλογή θα μπορούσε να είναι μία λίστα, μία πλειάδα ή ένα λεξικό. Είναι στο χέρι σας να κάνετε την κατάλληλη επιλογή. Μπορείτε να ελέγξετε το πρόγραμμα σας με μήκος προθέματος δύο αλλά θα πρέπει να γράψετε το πρόγραμμα με τέτοιο τρόπο ούτως ώστε να είναι εύκολο να δοκιμάσετε και άλλα μήκη.*

2. *Προσθέστε μία συνάρτηση στο προηγούμενο πρόγραμμα η οποία να παράγει τυχαίο κείμενο με βάση την ανάλυση Μαρκόφ. Αυτό είναι ένα παράδειγμα από την* Emma *με μήκος προθέματος 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.

Για αυτό το παράδειγμα, άφησα τα σημεία της στίξης τα οποία είναι επισυναπτόμενα στις λέξεις. Το αποτέλεσμα, συντακτικά είναι σχεδόν σωστό αλλά όχι τελείως και σημασιολογικά βγάζει νόημα αλλά όχι αρκετά.

Τι συμβαίνει αν αυξήσετε το μήκος προθέματος ; Έχει περισσότερο νόημα το τυχαίο κείμενο ;

3. *Άπαξ και γίνει λειτουργικό το πρόγραμμα σας και δουλεύει, ίσως θα θέλατε να δοκιμάσετε ένα mash-up: Αν αναλύσετε κείμενο από δύο ή περισσότερα βιβλία, το τυχαίο κείμενο που θα δημιουργήσετε θα συνδυάζει το λεξιλόγιο και τις φράσεις από τις πηγές με ενδιαφέροντες τρόπους.*

Αναφορά : Αυτή η μελέτη περίπτωσης είναι βασισμένη σε ένα παράδειγμα από το βιβλίο The Practice of Programming *Addison-Wesley, 1999 των Κέρνιγκαν και Πάικ.*

Θα πρέπει να προσπαθήσετε αυτή την άσκηση προτού προχωρήσετε. Μπορείτε να κατεβάσετε τη λύση μου από το σύνδεσμο : `http://thinkpython.com/code/markov.py`*. Θα χρειαστείτε επίσης και το* `http://thinkpython.com/code/emma.txt`*.*

13.9 Δομές δεδομένων

Η χρήση της ανάλυσης Μαρκόφ για την παραγωγή τυχαίου κειμένου είναι διασκεδαστική, αλλά σε αυτή την άσκηση υπάρχει και κάτι πολύ σημαντικό και αυτό είναι η επιλογή της δομής δεδομένων. Για να λύσετε τις προηγούμενες ασκήσεις έπρεπε να επιλέξετε :

- Πώς θα αναπαραστήσετε τα προθέματα.
- Πως θα αναπαραστήσετε τη συλλογή των πιθανών επιθημάτων.
- Πως θα αναπαραστήσετε την αντιστοίχιση του κάθε προθέματος με τη συλλογή των πιθανών επιθημάτων.

Εντάξει, το τελευταίο είναι το πιο εύκολο, αφού ο μοναδικός τύπος αντιστοίχισης που έχουμε δει είναι ένα λεξικό, άρα είναι η φυσική επιλογή.

Για τα προθέματα, οι πιο προφανείς επιλογές είναι μία συμβολοσειρά, μία λίστα συμβολοσειρών ή μία πλειάδα συμβολοσειρών. Για τα επιθήματα, μία επιλογή είναι μία λίστα και μία άλλη είναι ένα ιστόγραμμα (λεξικό).

Πως πρέπει να επιλέξετε ; Το πρώτο βήμα είναι να εξετάσετε τις πράξεις που θα χρειαστεί να υλοποιήσετε για την κάθε δομή δεδομένων. Για τα προθέματα, πρέπει να είμαστε σε θέση να αφαιρέσουμε λέξεις από την αρχή και να προσθέσουμε στο τέλος. Για παράδειγμα, αν το τρέχον πρόθεμα είναι το "Half a" και η επόμενη λέξη είναι λέξη είναι η "bee," πρέπει να είστε σε θέση να σχηματίσετε το επόμενο πρόθεμα, το "a bee".

Η πρώτη σας επιλογή θα μπορούσε να είναι μία λίστα, αφού είναι εύκολο να προσθέσουμε και να αφαιρέσουμε στοιχεία, αλλά πρέπει επίσης να έχουμε τη δυνατότητα να χρησιμοποιήσουμε τα προθέματα σαν κλειδιά σε ένα λεξικό, επομένως αποκλείουμε τις λίστες. Με τις πλειάδες, δεν μπορείτε να προσθέσετε ή να αφαιρέσετε, αλλά μπορείτε να χρησιμοποιήσετε τον τελεστή της πρόσθεσης για να σχηματίσετε μία νέα πλειάδα :

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

Η `shift` παίρνει μία πλειάδα λέξεων, την `prefix`, και μία συμβολοσειρά, την `word`, και σχηματίζει μία νέα πλειάδα η οποία έχει όλες τις λέξεις στην `prefix` εκτός της πρώτης και την `word` προστιθεμένη στο τέλος.

Για τη συλλογή των προθεμάτων, οι πράξεις που θα χρειαστεί να εκτελέσουμε περιλαμβάνουν την προσθήκη ενός νέου προθέματος (ή την προσαύξηση ενός υπάρχοντος) και την επιλογή ενός τυχαίου επιθήματος.

Η προσθήκη ενός νέου προθέματος είναι το ίδιο εύκολη τόσο για την υλοποίηση με λίστα όσο και με ιστόγραμμα. Η επιλογή ενός τυχαίου στοιχείου από μία λίστα είναι εύκολη, ενώ η επιλογή από ένα ιστόγραμμα είναι δύσκολο να γίνει αποδοτικά (βλ. ΑΣΚΗΣΗ 13.7).

Μέχρι στιγμής έχουμε μιλήσει κυρίως για την ευκολία της υλοποίησης, αλλά υπάρχουν και άλλοι παράγοντες που πρέπει να εξετάσουμε κατά την επιλογή δομών δεδομένων. Ένας είναι ο χρόνος εκτέλεσης. Μερικές φορές υπάρχει ένας θεωρητικός λόγος για τον οποίο θα περιμέναμε μία δομή δεδομένων να είναι γρηγορότερη από μία άλλη. Για παράδειγμα, ανέφερα ότι ο τελεστής `in` είναι γρηγορότερος με τα λεξικά σε σχέση με τις λίστες, όταν τουλάχιστον το πλήθος των στοιχείων είναι μεγάλο.

Αλλά συνήθως δεν μπορείτε να ξέρετε εκ των προτέρων ποια υλοποίηση θα είναι γρηγορότερη. Μία επιλογή είναι να υλοποιήσετε και τις δύο και να δείτε ποια είναι γρηγορότερη. Αυτή η προσέγγιση ονομάζεται **benchmarking** (συγκριτική αξιολόγηση). Μία εναλλακτική πρακτική είναι να διαλέξετε τη δομή που υλοποιείται ευκολότερα και στη συνέχεια να δείτε αν είναι αρκετά γρήγορη για την εφαρμογή που προορίζεται. Αν ναι, δεν υπάρχει λόγος να συνεχίσετε. Αν όχι, υπάρχουν εργαλεία, όπως η μονάδα λογισμικού `profile`, τα οποία μπορούν να εντοπίσουν τα μέρη ενός προγράμματος που χρειάζονται τον περισσότερο χρόνο.

Ο άλλος παράγοντας που πρέπει να λάβουμε υπόψη μας είναι ο χώρος αποθήκευσης. Για παράδειγμα, αν χρησιμοποιήσετε ένα ιστόγραμμα για τη συλλογή των επιθημάτων ίσως καταλάμβανε λιγότερο χώρο επειδή αρκεί να αποθηκεύσετε κάθε λέξη μία μόνο φορά, ανεξαρτήτως του πόσες φορές εμφανίζεται μέσα στο κείμενο. Σε μερικές περιπτώσεις, εξοικονομώντας χώρο μπορεί να κάνετε το πρόγραμμα να τρέχει γρηγορότερα, και στην ακραία περίπτωση, το πρόγραμμά σας μπορεί να μην τρέχει καθόλου αν ξεμείνετε από μνήμη. Αλλά για τις περισσότερες εφαρμογές, ο χώρος είναι δευτερεύουσας σημασίας μετά το χρόνο εκτέλεσης.

Μια τελευταία σκέψη : με βάση αυτά που έχω πει μέχρι στιγμής, έχω υπονοήσει ότι θα πρέπει να χρησιμοποιούμε μία δομή δεδομένων τόσο για την ανάλυση όσο και για τη δημιουργία. Αλλά από τη στιγμή που αυτές είναι δύο ξεχωριστές φάσεις, θα μπορούσαμε να χρησιμοποιήσουμε μία δομή για την ανάλυση και στη συνέχεια να την τροποποιήσουμε για την δημιουργία. Αν ο χρόνος που εξοικονομήθηκε κατά τη διάρκεια της δημιουργίας υπερβαίνει του χρόνου που δαπανήθηκε στη μετατροπή τότε θα είχαμε καθαρό κέρδος.

13.10 Αποσφαλμάτωση

Όταν αποσφαλματώνετε ένα πρόγραμμα, και ειδικά όταν δουλεύετε με ένα δύσκολο σφάλμα, υπάρχουν τέσσερα πράγματα που μπορείτε να δοκιμάσετε :

ανάγνωση : Εξετάστε τον κώδικά σας, διαβάστε τον στο εαυτό σας και ελέγξτε ότι λέει αυτό που θέλατε να λέει.

τρέξιμο : Πειραματιστείτε κάνοντας αλλαγές και τρέχοντας διαφορετικές εκδόσεις. Συχνά, αν εμφανίσετε το σωστό πράγμα στο σωστό μέρος του προγράμματος, το πρόβλημα γίνεται προφανές, αλλά μερικές φορές πρέπει να αφιερώσετε λίγο χρόνο για να φτιάξετε σκαλωσιά.

συλλογισμό : Πάρτε λίγο χρόνο για να σκεφτείτε! Τι είδους σφάλμα είναι : συντακτικό, χρόνου εκτέλεσης ή σημασιολογικό ; Τι πληροφορία μπορείτε να πάρετε από τα μηνύματα λάθους ή από την έξοδο του προγράμματος ; Τι είδους σφάλμα θα μπορούσε να προκαλέσει το πρόβλημα που βλέπετε ; Ποια ήταν η τελευταία αλλαγή που κάνατε πριν εμφανιστεί το πρόβλημα ;

υποχώρηση : Σε κάποιο σημείο, το καλύτερο είναι να κάνετε πίσω αναιρώντας πρόσφατες αλλαγές, μέχρι να πάρετε πίσω ένα πρόγραμμα που δουλεύει και που μπορείτε να κατανοήσετε. Τότε μπορείτε να ξεκινήσετε την ανακατασκευή.

Μερικές φορές, οι νέοι προγραμματιστές κολλάνε σε μία από αυτές τις διαδικασίες και ξεχνάνε τις υπόλοιπες. Κάθε δραστηριότητα συνοδεύεται από τη δική της κατάσταση αποτυχίας.

Για παράδειγμα, η ανάγνωση του κώδικα μπορεί να βοηθήσει αν το πρόβλημα είναι ένα τυπογραφικό λάθος, αλλά όχι αν το πρόβλημα είναι μία εννοιολογική παρεξήγηση. Αν δεν κατανοείτε τι κάνει το πρόγραμμά σας, μπορείτε να το διαβάσετε 100 φορές και να μην καταλάβετε ποτέ το λάθος, επειδή το λάθος είναι μέσα στο κεφάλι σας.

Οι πειραματισμοί εκτέλεσης μπορούν να βοηθήσουν, ειδικά αν τρέχετε μικρές και απλές δοκιμές. Αλλά αν τρέχετε πειράματα χωρίς να σκέφτεστε ή να διαβάζετε τον κώδικά σας τότε μάλλον ακολουθείτε ένα μοτίβο το οποίο αποκαλώ ” προγραμματισμό στα τυφλά ”, μία μέθοδος κατά την οποία κάνουμε τυχαίες αλλαγές μέχρι το πρόγραμμά μας να κάνει το σωστό πράγμα. Είναι περιττό να αναφέρω ότι αυτός ο προγραμματισμός μπορεί να πάρει πάρα πολύ χρόνο.

Πρέπει να αφιερώσετε χρόνο για να σκεφτείτε. Η αποσφαλμάτωση είναι σαν την πειραματική επιστήμη. Πρέπει να έχετε μία τουλάχιστον υπόθεση σχετικά με το ποιο είναι το πρόβλημα. Αν υπάρχουν δύο οι περισσότερες πιθανότητες, προσπαθήστε να σκεφτείτε έναν τεστ το οποίο θα εξαλείψει ένα από αυτά.

Ένα διάλειμμα σας βοηθάει να σκεφτείτε. Το ίδιο κάνει και η ομιλία. Αν εξηγήσετε το πρόβλημα σε κάποιον άλλο (ή ακόμα και στον ίδιο σας τον εαυτό), μερικές φορές θα βρίσκετε την απάντηση πριν ολοκληρώσετε την ερώτησή σας.

Αλλά ακόμα και οι καλύτερες τεχνικές αποσφαλμάτωσης θα αποτύχουν αν υπάρχουν πάρα πολλά λάθη ή αν ο κώδικας που προσπαθείτε να διορθώσετε είναι πολύ μεγάλος και πολύπλοκος. Μερικές φορές, η καλύτερη επιλογή είναι να υποχωρήσετε και να απλοποιήσετε το πρόγραμμα μέχρι να πάρετε κάτι το οποίο δουλεύει και το καταλαβαίνετε.

Οι νέοι προγραμματιστές είναι συχνά απρόθυμοι να υποχωρήσουν επειδή δεν αντέχουν να διαγράψουν ούτε μία γραμμή κώδικα (ακόμα και αν είναι λάθος). Αν αυτό σας κάνει να αισθάνεστε καλύτερα, αντιγράψτε το πρόγραμμα σας σε ένα άλλο αρχείο προτού ξεκινήσετε να το απογυμνώνετε. Στη συνέχεια μπορείτε να επικολλήσετε τα κομμάτια πίσω στο αρχικό αρχείο, λίγο λίγο τη φορά.

Η ανεύρεση ενός δύσκολου σφάλματος απαιτεί ανάγνωση, τρέξιμο, συλλογισμό και μερικές φορές υποχώρηση. Αν κολλήσετε σε μία από αυτές τις διαδικασίες δοκιμάστε τις άλλες.

13.11 Ορολογία

ντετερμινιστικό: Αναφέρεται σε ένα πρόγραμμα το οποίο κάνει το ίδιο πράγμα κάθε φορά που τρέχει, δοθέντων των ίδιων εισόδων.

ψευδοτυχαία: Αναφέρεται σε μία ακολουθία από αριθμούς οι οποίοι φαίνεται να είναι τυχαίοι, αλλά παράγονται από ένα ντετερμινιστικό πρόγραμμα.

προεπιλεγμένη τιμή: Η τιμή που δίνεται σε μία προαιρετική παράμετρο αν δεν παρασχεθεί κανένα όρισμα.

παράκαμψη: Η αντικατάσταση μίας προεπιλεγμένης τιμής με ένα όρισμα.

συγκριτική αξιολόγηση: Η διαδικασία της επιλογής μεταξύ δομών δεδομένων εφαρμόζοντας εναλλακτικές λύσεις και δοκιμάζοντάς τες με ένα δείγμα πιθανών εισόδων.

13.12 Ασκήσεις

Άσκηση 13.9. *Ο " βαθμός " μιας λέξης είναι η θέση της σε μία λίστα από λέξεις ταξινομημένη με βάση τη συχνότητα : η πιο συνηθισμένη λέξη είναι βαθμού 1, η δεύτερη πιο συνηθισμένη έχει βαθμό 2 και ούτω καθεξής.*

Ο νόμος του Ζιπφ περιγράφει μία σχέση μεταξύ των βαθμών και των συχνοτήτων των λέξεων στις φυσικές γλώσσες (`http://en.wikipedia.org/wiki/Zipf's_law`*). Ειδικότερα, προβλέπει ότι η συχνότητα, f, της λέξης με βαθμό r είναι :*

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

όπου s και c είναι οι παράμετροι οι οποίες εξαρτώνται από τη γλώσσα και το κείμενο. Αν πάρετε το λογάριθμο και των δύο πλευρών αυτής της εξίσωσης, τότε παίρνετε :

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

Επομένως αν σχεδιάσετε το $\log f$ σε σχέση με το r, θα πρέπει να πάρετε μία ευθεία με κλίση $-s$ και σημείο τομής $\log c$.

Γράψτε ένα πρόγραμμα το οποίο θα διαβάζει ένα κείμενο από ένα αρχείο, θα μετράει τις συχνότητες των λέξεων και θα εμφανίζει μία γραμμή για κάθε λέξη, σε φθίνουσα σειρά με βάση τη συχνότητα, με το $\log f$ και το $\log r$. Χρησιμοποιήστε το πρόγραμμα γραφημάτων της επιλογή σας για να σχεδιάσετε τα αποτελέσματα και να ελέγξετε αν σχηματίζουν ευθεία γραμμή. Μπορείτε να εκτιμήσετε την τιμή του s;

Λύση : `http://thinkpython.com/code/zipf.py`*. Για να φτιάξετε ένα γράφημα ίσως πρέπει να εγκαταστήσετε την matplotlib (see* `http://matplotlib.sourceforge.net/`*).*

Κεφάλαιο 14

Αρχεία

14.1 Διάρκεια

Τα περισσότερα από τα προγράμματα που έχουμε δει μέχρι τώρα είναι παροδικά με την έννοια ότι τρέχουν για ένα μικρό χρονικό διάστημα και παράγουν κάποια έξοδο, αλλά όταν τελειώσουν, τα δεδομένα τους εξαφανίζονται. Αν εκτελέσετε ξανά το πρόγραμμα θα ξεκινήσει από μηδενική βάση.

Άλλα προγράμματα είναι διαρκή, τρέχουν δηλαδή για ένα μεγάλο χρονικό διάστημα (ή συνεχώς). Αυτά τα προγράμματα κρατάνε τουλάχιστον κάποια από δεδομένα τους σε κάποια μόνιμη αποθήκευση (ένα σκληρό δίσκο για παράδειγμα) και αν κλείσουν ή επανεκκινηθούν τότε ξεκινάνε από εκεί που σταμάτησαν.

Παραδείγματα διαρκών προγραμμάτων είναι τα λειτουργικά συστήματα, τα οποία τρέχουν λίγο πολύ κάθε φορά που ένας υπολογιστής είναι ανοιχτός, και οι εξυπηρετητές ιστού, οι οποίοι τρέχουν όλη την ώρα περιμένοντας αιτήσεις για το δίκτυο.

Ένας από τους απλούστερους τρόπους για να διατηρούν τα προγράμματα τα δεδομένα τους είναι μέσω της ανάγνωσης και εγγραφής αρχείων κειμένου. Έχουμε ήδη δει προγράμματα τα οποία διαβάζουν αρχεία κειμένου και σε αυτό το κεφάλαιο θα δούμε προγράμματα τα οποία τα γράφουν.

Μία εναλλακτική λύση είναι να αποθηκεύουμε την κατάστασή του προγράμματος σε μία βάση δεδομένων. Σε αυτό το κεφάλαιο θα σας παρουσιάσω μία απλή βάση και μία μονάδα λογισμικού, την `pickle`, η οποία κάνει εύκολη την αποθήκευση των δεδομένων ενός προγράμματος.

14.2 Διάβασμα και γράψιμο

Ένα αρχείο κειμένου είναι μία ακολουθία χαρακτήρων αποθηκευμένη σε ένα μέσο μόνιμης αποθήκευσης όπως ο σκληρός δίσκος, η μνήμη φλας ή το CD-ROM. Είδαμε πως να ανοίγουμε και να διαβάζουμε ένα αρχείο στην Ενότητα 9.1.

Για να γράψετε ένα αρχείο, πρέπει να το ανοίξετε σε κατάσταση `'w'` ορίζοντάς την σαν δεύτερη παράμετρο :

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

Αν το αρχείο υπάρχει ήδη και το ανοίξετε σε κατάσταση εγγραφής τότε σβήνονται τα παλιά δεδομένα και αρχίζει καθαρό, οπότε να είστε προσεκτικοί ! Αν το αρχείο δεν υπάρχει τότε δημιουργείται ένα νέο.

Η μέθοδος `write` βάζει δεδομένα στο αρχείο.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

Και πάλι, το αντικείμενο αρχείου παρακολουθεί το που βρίσκεται και άρα αν ξανακαλέσετε τη `write` θα προσθέσει τα νέα δεδομένα στο τέλος.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
```

Όταν τελειώσετε με το γράψιμο θα πρέπει να κλείσετε το αρχείο.

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

14.3 Τελεστής διαμόρφωσης

Το όρισμα της `write` πρέπει να είναι μία συμβολοσειρά. Επομένως, αν θέλουμε να βάλουμε άλλες τιμές σε ένα αρχείο, πρέπει να τις μετατρέψουμε πρώτα σε συμβολοσειρές. Ο ευκολότερος τρόπος για να το κάνουμε αυτό είναι με την `str`:

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

Μία εναλλακτική λύση είναι να χρησιμοποιήσουμε τον τελεστή διαμόρφωσης, %. Όταν εφαρμόζεται σε ακέραιους αριθμούς λειτουργεί σαν τελεστής υπολογισμού υπολοίπου αλλά όταν ο πρώτος τελεστέος είναι μία συμβολοσειρά τότε ο % λειτουργεί σαν τελεστής διαμόρφωσης.

Ο πρώτος τελεστέος είναι η συμβολοσειρά διαμόρφωσης, η οποία περιέχει μία ή περισσότερες ακολουθίες διαμόρφωσης, η οποία καθορίζει πως θα διαμορφωθεί ο δεύτερος τελεστέος. Το αποτέλεσμα είναι μία συμβολοσειρά.

Για παράδειγμα, η ακολουθία διαμόρφωσης `'%d'` σημαίνει ότι ο δεύτερος τελεστέος θα πρέπει να διαμορφωθεί σαν ένας ακέραιος (το `d` σημαίνει “decimal”):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

Το αποτέλεσμα είναι η συμβολοσειρά `'42'`, η οποία δεν πρέπει να συγχέεται με την ακέραια τιμή 42.

Μία ακολουθία διαμόρφωσης μπορεί να εμφανίζεται οπουδήποτε μέσα στη συμβολοσειρά και έτσι μπορείτε να ενσωματώσετε μία τιμή μέσα σε μία πρόταση :

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

Αν υπάρχουν περισσότερες από μία ακολουθίες διαμόρφωσης μέσα στη συμβολοσειρά, το δεύτερο όρισμα πρέπει να είναι μία πλειάδα. Κάθε ακολουθία διαμόρφωσης αντιστοιχίζεται με ένα στοιχείο της πλειάδας με τη σειρά.

Το ακόλουθο παράδειγμα χρησιμοποιεί την '%d' για να διαμορφώσει έναν ακέραιο, την '% g' για να διαμορφώσει έναν αριθμό κινητής υποδιαστολής (μην ρωτήσετε γιατί), και την '%s' για να διαμορφώσει μία συμβολοσειρά :

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

Ο αριθμός των στοιχείων στην πλειάδα πρέπει να ταιριάζει με τον αριθμό των ακολουθιών διαμόρφωσης μέσα στη συμβολοσειρά. Επίσης, οι τύποι των στοιχείων πρέπει να ταιριάζουν με τις ακολουθίες διαμόρφωσης :

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

Στο πρώτο παράδειγμα, δεν υπάρχουν αρκετά στοιχεία και στο δεύτερο το στοιχείο έχει λάθος τύπο.

Ο τελεστής διαμόρφωσης είναι πολύ ισχυρός, αλλά μπορεί να είναι δύσκολο να χρησιμοποιηθεί. Μπορείτε να διαβάσετε περισσότερα σχετικά με αυτόν στο σύνδεσμο `http://docs.python.org/2/library/stdtypes.html#string-formatting`.

14.4 Ονόματα αρχείων και διαδρομές

Τα αρχεία είναι οργανωμένα σε καταλόγους (γνωστοί και ως φάκελοι). Κάθε πρόγραμμα που τρέχει έχει έναν " τρέχων κατάλογο ", ο οποίος είναι ο προεπιλεγμένος κατάλογος για τις περισσότερες λειτουργίες. Για παράδειγμα, όταν ανοίγετε ένα αρχείο για διάβασμα, η Python το ψάχνει στον τρέχων κατάλογο.

Η μονάδα λογισμικού `os` παρέχει συναρτήσεις για να μπορούμε να δουλέψουμε με αρχεία και καταλόγους (το "os" σημαίνει "operating system"). Η `os.getcwd` επιστρέφει το όνομα του τρέχοντος καταλόγου :

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

Η `cwd` σημαίνει "current working directory". Το αποτέλεσμα σε αυτό το παράδειγμα είναι `/home/dinsdale`, το οποίο είναι ο κατάλογος home του χρήστη με όνομα `dinsdale`.

Μία συμβολοσειρά όπως η `cwd` η οποία προσδιορίζει ένα αρχείο ονομάζεται διαδρομή (**path**). Μία σχετική διαδρομή ξεκινάει από τον τρέχων κατάλογο ενώ μία απόλυτη διαδρομή ξεκινάει από τον ανώτατο κατάλογο του συστήματος αρχείων.

Οι διαδρομές που έχουμε δει μέχρι τώρα είναι απλά ονόματα αρχείων, άρα είναι σε σχέση με τον τρέχων κατάλογο. Για να βρείτε την απόλυτη διαδρομή ενός αρχείου μπορείτε να χρησιμοποιήσετε την `os.path.abspath`:

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

Η `os.path.exists` ελέγχει αν υπάρχει το αρχείο ή ο κατάλογος :

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

Αν υπάρχει, η os.path.isdir ελέγχει αν είναι κατάλογος :

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

Ομοίως, η os.path.isfile ελέγχει αν είναι αρχείο.

Η os.listdir επιστρέφει μία λίστα από τα αρχεία (και τους άλλους καταλόγους) στο δοθέν κατάλογο : Similarly, os.path.isfile checks whether it's a file.

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

Για να κάνουμε μία επίδειξη αυτών των συναρτήσεων, το ακόλουθο παράδειγμα περπατάει μέσα σε έναν κατάλογο, εμφανίζει τα ονόματα όλων των αρχείων και καλεί αναδρομικά τον εαυτό του σε όλους τους καταλόγους.

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print path
        else:
            walk(path)
```

Η os.path.join παίρνει έναν κατάλογο και ένα όνομα αρχείου και τα εντάσσει σε μία πλήρη διαδρομή.

Άσκηση 14.1. *Η μονάδα λογισμικού* os *παρέχει μία συνάρτηση με όνομα* walk *η οποία είναι παρόμοια με αυτήν αλλά περισσότερο ευέλικτη. Διαβάστε την τεκμηρίωση και χρησιμοποιήστε την για να εμφανίσετε τα ονόματα των αρχείων σε ένα δοθέν κατάλογο και τους υποκαταλόγους του.*

Λύση : http://thinkpython.com/code/walk.py.

14.5 Πιάσιμο εξαιρέσεων

Πολλά πράγματα μπορούν να πάνε στραβά όταν προσπαθείτε να διαβάσετε και να γράψετε αρχεία. Αν προσπαθήσετε να ανοίξετε ένα αρχείο το οποίο δεν υπάρχει τότε θα πάρετε ένα IOError:

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

Αν δεν έχετε δικαίωμα πρόσβασης για ένα αρχείο :

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

Και αν προσπαθήσετε να ανοίξετε έναν κατάλογο για ανάγνωση θα πάρετε :

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

Για να αποφύγετε αυτά τα σφάλματα, μπορείτε να χρησιμοποιείτε συναρτήσεις όπως η os.path.exists και η os.path.isfile, αλλά θα χρειαστεί πολύ χρόνο και κώδικα για να ελέγξετε όλες τις πιθανότητες (εάν το "Errno 21" είναι κάποια ένδειξη, υπάρχουν τουλάχιστον 21 πράγματα που μπορεί να πάνε στραβά).

Είναι καλύτερο να προχωρήσετε και δοκιμάσετε να ασχοληθείτε με προβλήματα μόνο αν υπάρχουν. Αυτό ακριβώς κάνει η δήλωση `try`, της οποίας η σύνταξη είναι παρόμοια με μία δήλωση `if`:

```
try:
    fin = open('bad_file')
    for line in fin:
        print line
    fin.close()
except:
    print 'Something went wrong.'
```

Η Python ξεκινάει εκτελώντας την παράγραφο `try`. Αν όλα πάνε καλά, παραλείπει την παράγραφο `except` και προχωράει. Αν προκύψει μία εξαίρεση, πηδά έξω από την `try` και εκτελεί την `except`.

Ο χειρισμός μιας εξαίρεσης με μία δήλωση `try` ονομάζεται ” πιάσιμο εξαίρεσης ”. Σε αυτό το παράδειγμα, η παράγραφος `except` εμφανίζει ένα μήνυμα λάθους το οποίο δεν είναι πολύ χρήσιμο. Σε γενικές γραμμές, το πιάσιμο μίας εξαίρεσης σας δίνει την δυνατότητα να διορθώσετε το πρόβλημα, ή να ξαναδοκιμάσετε, ή τουλάχιστον να τερματίσετε το πρόγραμμα ομαλά.

Άσκηση 14.2. *Γράψτε μία συνάρτηση με όνομα* `sed` *η οποία θα παίρνει σαν ορίσματα μία πρότυπη συμβολοσειρά, μία συμβολοσειρά αντικατάστασης και δύο ονόματα αρχείων. Θα πρέπει να διαβάζει το πρώτο αρχείο και να γράφει τα περιεχόμενά του μέσα στο δεύτερο (δημιουργώντας το αν χρειάζεται). Αν η πρότυπη συμβολοσειρά εμφανίζεται οπουδήποτε μέσα στο αρχείο θα πρέπει να αντικατασταθεί με τη συμβολοσειρά αντικατάστασης.*

Αν συμβεί κάποιο λάθος κατά το άνοιγμα, το διάβασμα, το γράψιμο ή το κλείσιμο των αρχείων το πρόγραμμα θα πρέπει να πιάσει την εξαίρεση, να εμφανίσει ένα μήνυμα λάθους και τερματίσει. Λύση : `http://thinkpython.com/code/sed.py`.

14.6 Βάσεις δεδομένων

Μία βάση δεδομένων είναι ένα αρχείο το οποίο είναι οργανωμένο για την αποθήκευση δεδομένων. Οι περισσότερες βάσεις δεδομένων είναι οργανωμένες όπως τα λεξικά με την έννοια ότι αντιστοιχίζουν κλειδιά σε τιμές. Η μεγαλύτερη διαφορά είναι ότι η βάση δεδομένων είναι στον δίσκο (ή σε κάποιο άλλο μέσο μόνιμης αποθήκευσης) και άρα εξακολουθεί να υπάρχει και μετά το τέλος του προγράμματος.

Η μονάδα λογισμικού `anydbm` παρέχει μία διεπαφή για τη δημιουργία και ενημέρωση αρχείων βάσεων δεδομένων. Σαν παράδειγμα, θα δημιουργήσω μία βάση η οποία θα περιέχει λεζάντες για αρχεία εικόνων.

Το άνοιγμα μίας βάση δεδομένων είναι παρόμοιο με το άνοιγμα άλλων αρχείων :

```
>>> import anydbm
>>> db = anydbm.open('captions.db', 'c')
```

Η κατάσταση `'c'` σημαίνει ότι η βάση δεδομένων θα πρέπει να δημιουργηθεί αν δεν υπάρχει ήδη. Το αποτέλεσμα είναι ένα αντικείμενο βάσης δεδομένων το οποίο μπορεί να χρησιμοποιηθεί (για τις περισσότερες πράξεις) όπως ένα λεξικό. Αν δημιουργήσετε ένα νέο στοιχείο, η `anydbm` ενημερώνει το αρχείο της βάσης.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

Όταν αποκτάτε πρόσβαση σε ένα από τα στοιχεία, η `anydbm` διαβάζει το αρχείο :

```
>>> print db['cleese.png']
Photo of John Cleese.
```

Αν κάνετε μία επιπλέον εκχώρηση σε ένα υπάρχον κλειδί, η anydbm αντικαθιστά την παλιά τιμή :

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> print db['cleese.png']
Photo of John Cleese doing a silly walk.
```

Πολλές μέθοδοι των λιστών, όπως η keys και η items, δουλεύουν και με αντικείμενα των βάσεων δεδομένων. Το ίδιο ισχύει και με μία δήλωση for:

```
for key in db:
    print key
```

Όπως και με τα άλλα αρχεία, θα πρέπει να κλείνετε τη βάση δεδομένων όταν τελειώσετε :

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

14.7 Σειριοποίηση

Ένας περιορισμός του anydbm είναι ότι τα κλειδιά και οι τιμές πρέπει να είναι συμβολοσειρές. Αν προσπαθήσετε να χρησιμοποιήσετε οποιονδήποτε άλλο τύπο θα πάρετε ένα σφάλμα.

Σε αυτήν την περίπτωση μπορεί να σας βοηθήσει το άρθρωμα pickle. Μεταφράζει σχεδόν οποιοδήποτε τύπο αντικειμένου σε μία συμβολοσειρά κατάλληλη για αποθήκευση σε μία βάση δεδομένων και στη συνέχεια τα ξαναμετατρέπει σε αντικείμενα.

Η pickle.dumps παίρνει ένα αντικείμενο σαν παράμετρο και επιστρέφει μία αναπαράσταση συμβολοσειράς (το dumps είναι συντομογραφία του “dump string”):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
'(lp0\nI1\naI2\naI3\na.'
```

Η διαμόρφωση δεν είναι εμφανής στους αναγνώστες αλλά εννοείται ότι είναι εύκολο για την pickle να διερμηνεύει. Η pickle.loads ανασυνθέτει το αντικείμενο :

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

Παρόλο που το νέο αντικείμενο έχει την ίδια τιμή με το παλιό, δεν είναι το ίδιο αντικείμενο γενικά :

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

Με άλλα λόγια, η σειριοποίηση και η αποσειριοποίηση έχουν το ίδιο αποτέλεσμα με την αντιγραφή του αντικειμένου.

Μπορείτε να χρησιμοποιήσετε την pickle για να αποθηκεύσετε μη-συμβολοσειρές σε μία βάση δεδομένων. Στην πραγματικότητα, αυτός ο συνδυασμός είναι τόσο κοινός που έχει ενθυλακωθεί σε ένα άρθρωμα με όνομα shelve.

Άσκηση 14.3. *Αν κατεβάσετε τη λύση μου για την Άσκηση 12.4 από το σύνδεσμο* `http://thinkpython.com/code/anagram_sets.py`*, θα δείτε ότι δημιουργεί ένα λεξικό το οποίο αντιστοιχεί ταξινομημένες συμβολοσειρές ή γράμματα στη λίστα των λέξεων οι οποίες μπορούν να γραφτούν με αυτά τα γράμματα. Για παράδειγμα, η* `'opst'` *αντιστοιχίζεται στη λίστα :* `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Γράψτε ένα άρθρωμα το οποίο θα εισάγει το `anagram_sets` *και θα παρέχει δύο νέες συναρτήσεις : την* `store_anagrams` *η οποία θα πρέπει να αποθηκεύει το λεξικό των αναγραμματισμών σε ένα " ράφι " και την* `read_anagrams` *η οποία θα πρέπει να αναζητάει μία λέξη και να επιστρέφει μία λίστα με τους αναγραμματισμούς της. Λύση :* `http://thinkpython.com/code/anagram_db.py`.

14.8 Σωληνώσεις

Τα περισσότερα λειτουργικά συστήματα παρέχουν μία διασύνδεση γραμμής εντολών, γνωστή και ως κέλυφος. Τα κελύφη παρέχουν συνήθως εντολές για την περιήγηση στο σύστημα αρχείων και την εκκίνηση εφαρμογών. Για παράδειγμα, στο Unix μπορείτε να αλλάξετε κατάλογο με την `cd`, να εμφανίσετε τα περιεχόμενα ενός καταλόγου με την `ls` και να εκκινήσετε έναν περιηγητή ιστού πληκτρολογώντας για παράδειγμα `firefox`.

Οποιοδήποτε πρόγραμμα το οποίο μπορείτε να εκκινήσετε από το κέλυφος μπορεί επίσης να εκκινηθεί και από την Python χρησιμοποιώντας μία σωλήνωση (pipe). Μία σωλήνωση είναι ένα αντικείμενο το οποίο αντιπροσωπεύει ένα πρόγραμμα το οποίο τρέχει.

Για παράδειγμα, η εντολή `ls -l` του Unix εμφανίζει κανονικά τα περιεχόμενα του τρέχοντος καταλόγου (σε πλήρη μορφή). Μπορείτε να εκκινήσετε την `ls` με την `os.popen`[1]:

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

Το όρισμα είναι μία συμβολοσειρά η οποία περιέχει μία εντολή κελύφους. Η επιστρεφόμενη τιμή είναι ένα αντικείμενο το οποίο συμπεριφέρεται όπως ένα ανοιχτό αρχείο. Μπορείτε να διαβάσετε την έξοδο της διεργασίας `ls` γραμμή γραμμή με την `readline` ή να πάρετε ολόκληρη την έξοδο με τη μία χρησιμοποιώντας την `read`:

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

Όταν τελειώσετε κλείνετε τη σωλήνωση όπως ένα αρχείο :

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

Η επιστρεφόμενη τιμή είναι η τελική κατάσταση της διεργασίας `ls`. Η `None` σημαίνει ότι τελείωσε κανονικά (χωρίς σφάλματα).

Για παράδειγμα, τα περισσότερα συστήματα Unix παρέχουν μία εντολή με όνομα `md5sum` η οποία διαβάζει τα περιεχόμενα ενός αρχείου και υπολογίζει ένα " άθροισμα ελέγχου ". Μπορείτε να διαβάσετε σχετικά τον MD5 στο σύνδεσμο `http://en.wikipedia.org/wiki/Md5`. Αυτή η εντολή παρέχει έναν αποδοτικό τρόπο για να ελέγχουμε αν δύο αρχεία έχουν τα ίδια περιεχόμενα. Η πιθανότητα, δύο διαφορετικά περιεχόμενα να αποδώσουν το ίδιο άθροισμα ελέγχου είναι πολύ μικρή (τόσο μικρή που είναι απίθανο να συμβεί πριν τη συντέλεια του κόσμου).

[1] `popen` έχει καταργηθεί τώρα, το οποίο σημαίνει ότι θα πρέπει να σταματήσουμε να τη χρησιμοποιούμε και να αρχίσουμε να χρησιμοποιούμε το άρθρωμα `subprocess`. Αλλά σε απλές περιπτώσεις, θεωρώ ότι το `subprocess` είναι περισσότερο περίπλοκο παρά απαραίτητο. Επομένως θα συνεχίσω να χρησιμοποιώ την `popen` μέχρι να την αφαιρέσουν τελείως.

Μπορείτε να χρησιμοποιήσετε μία σωλήνωση για να τρέξετε την `md5sum` από την Python και να πάρετε το αποτέλεσμα :

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

Άσκηση 14.4. *Σε μια μεγάλη συλλογή από αρχεία MP3, μπορεί να υπάρχουν περισσότερα από ένα αντίγραφα του ίδιου τραγουδιού, αποθηκευμένα σε διαφορετικούς καταλόγους ή με διαφορετικά ονόματα αρχείων. Ο στόχος αυτής της άσκησης είναι να αναζητήσουμε διπλότυπα.*

1. *Γράψτε ένα πρόγραμμα το οποίο θα ψάχνει έναν κατάλογο και όλους τους υποκαταλόγους του, αναδρομικά, και θα επιστρέφει μία λίστα με τις πλήρεις διαδρομές όλων των αρχείων δεδομένου ενός επιθήματος (όπως το* `.mp3` *). Σημείωση : η* `os.path` *παρέχει διάφορες χρήσιμες συναρτήσεις για το χειρισμό ονομάτων αρχείων και διαδρομών.*
2. *Για να αναγνωρίσετε τα διπλότυπα, μπορείτε να χρησιμοποιήσετε την* `md5sum` *για υπολογίσει ένα άθροισμα ελέγχου για κάθε αρχείο. Αν δύο αρχεία έχουν το ίδιο άθροισμα ελέγχου, τότε πιθανότατα έχουν τα ίδια περιεχόμενα.*
3. *Για να διπλοελέγξετε, μπορείτε να χρησιμοποιήσετε την εντολή του Unix* `diff`.

Λύση : `http://thinkpython.com/code/find_duplicates.py`.

14.9 Γράψιμο αρθρωμάτων

Κάθε αρχείο το οποίο περιέχει κώδικα Pyhton μπορεί να εισαχθεί σαν ένα άρθρωμα. Για παράδειγμα, υποθέστε ότι έχετε ένα αρχείο με όνομα `wc.py` με τον ακόλουθο κώδικα :

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

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

Αν τρέξετε αυτό το πρόγραμμα, θα διαβάσει τον εαυτό του και θα εμφανίσει τον αριθμό των γραμμών του αρχείου, οι οποίες είναι 7. Επίσης, μπορείτε να το εισάγετε κάπως έτσι :

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

Τώρα έχετε ένα αντικείμενο αρθρώματος `wc`:

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

Το οποίο παρέχει μία συνάρτηση με όνομα `linecount`:

```
>>> wc.linecount('wc.py')
```

```
7
```

Αρά κάπως έτσι γράφετε αρθρώματα στην Python.

Το μοναδικό πρόβλημα με αυτό το παράδειγμα είναι ότι όταν εισάγετε το άρθρωμα εκτελείτε και ο κώδικας ελέγχου στο τέλος. Κανονικά, όταν εισάγετε ένα άρθρωμα ορίζει νέες συναρτήσεις αλλά δεν τις εκτελεί.

Συχνά, τα προγράμματα τα οποία εισάγονται σαν αρθρώματα χρησιμοποιούν τον ακόλουθο ιδιωματισμό :

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

Η `__name__` είναι μία ενσωματωμένη μεταβλητή η οποία ορίζεται όταν ξεκινάει το πρόγραμμα. Στην περίπτωση που το πρόγραμμα τρέχει σαν σενάριο, η `__name__` έχει την τιμή `__main__` και εκτελείτε ο κώδικας ελέγχου. Διαφορετικά, αν το άρθρωμα έχει εισαχθεί, ο κώδικας ελέγχου παραλείπεται.

Άσκηση 14.5. *Γράψτε τον κώδικα αυτού του παραδείγματος σε ένα αρχείο με όνομα* `wc.py` *και τρέξτε το σαν σενάριο. Στη συνέχεια τρέξτε τον διερμηνέα της Python και γράψτε* `import wc` *για να εισάγετε το άρθρωμα. Ποια είναι η τιμή της* `__name__` *όταν εισάγεται το άρθρωμα ;*

Προσοχή : Αν εισάγετε ένα άρθρωμα το οποίο έχει ήδη εισαχθεί, η Python δεν θα κάνει τίποτα. Δεν ξαναδιαβάζει το αρχείο, ακόμα και αν έχει αλλάξει.

Αν θέλετε να ξαναφορτώσετε ένα άρθρωμα, μπορείτε να χρησιμοποιήσετε την ενσωματωμένη συνάρτηση `reload` *αλλά μπορεί να σας δυσκολέψει. Ο ασφαλέστερος τρόπος να το κάνετε είναι να επανεκκινήσετε τον διερμηνέα και στη συνέχεια να εισάγετε ξανά το άρθρωμα.*

14.10 Αποσφαλμάτωση

Όταν διαβάζετε και γράφετε αρχεία μπορεί να συναντήσετε προβλήματα με τους χαρακτήρες λευκού διαστήματος. Αυτά τα λάθη είναι δύσκολο να εντοπιστούν και να διορθωθούν επειδή τα κενά, οι στηλοθέτες και οι νέες γραμμές είναι συνήθως αόρατα :

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

Η ενσωματωμένη συνάρτηση `repr` μπορεί να σας βοηθήσει. Παίρνει σαν όρισμα ένα οποιοδήποτε αντικείμενο και επιστρέφει μία αναπαράσταση συμβολοσειράς του αντικειμένου. Για τις συμβολοσειρές, αναπαριστά τους χαρακτήρες λευκού διαστήματος με ακολουθίες ανάστροφων καθέτων :

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

Αυτό είναι πολύ χρήσιμο για την αποσφαλμάτωση.

Ένα άλλο πρόβλημα που μπορεί να συναντήσετε είναι ότι διαφορετικά συστήματα χρησιμοποιούν διαφορετικούς χαρακτήρες για δηλώσουν το τέλος μίας γραμμής. Κάποια συστήματα χρησιμοποιούν μία νέα γραμμή η οποία αναπαριστάται με `\n`. Κάποια άλλα χρησιμοποιούν έναν χαρακτήρα επιστροφής ο οποίος αναπαριστάται με `\r`. Και μερικά χρησιμοποιούν και τους δύο συμβολισμούς. Αν μετακινείτε αρχεία μεταξύ διαφορετικών συστημάτων τότε αυτές οι αντιφάσεις μπορεί να προκαλέσουν προβλήματα.

Για τα περισσότερα συστήματα, υπάρχουν εφαρμογές με τις οποίες μπορείτε να μετατρέψετε την μία μορφή στην άλλη. Μπορείτε να τις βρείτε (και να διαβάσετε περισσότερα σχετικά με αυτές) στο σύνδεσμο `http://en.wikipedia.org/wiki/Newline`. Ή, φυσικά, μπορείτε να γράψετε μία δική σας.

14.11 Ορολογία

διαρκή: Αφορά ένα πρόγραμμα το οποίο τρέχει επ' αόριστον και διατηρεί τουλάχιστον κάποια από τα δεδομένα του σε ένα μέσο μόνιμης αποθήκευσης.

τελεστής διαμόρφωσης: Ένας τελεστής, %, ο οποίος παίρνει μία συμβολοσειρά διαμόρφωσης και μία πλειάδα και παράγει μία συμβολοσειρά η οποία περιέχει τα στοιχεία της πλειάδας διαμορφωμένα όπως ορίζεται από τη συμβολοσειρά διαμόρφωσης.

συμβολοσειρά διαμόρφωσης: Μία συμβολοσειρά η οποία περιέχει ακολουθίες διαμόρφωσης

ακολουθία διαμόρφωσης: Μία ακολουθία χαρακτήρων μέσα σε μία συμβολοσειρά διαμόρφωσης, όπως η `%d`, η οποία καθορίζει πως θα διαμορφωθεί μία τιμή.

αρχείο κειμένου: Μία ακολουθία χαρακτήρων η οποία είναι αποθηκευμένη σε ένα μέσο μόνιμης αποθήκευσης όπως ο σκληρός δίσκος.

κατάλογος: Μία συλλογή αρχείων που ονομάζεται επίσης και φάκελος.

διαδρομή: Μία συμβολοσειρά η οποία προσδιορίζει ένα αρχείο.

σχετική διαδρομή: Μία διαδρομή η οποία ξεκινάει από τον τρέχον κατάλογο.

απόλυτη διαδρομή: Μία διαδρομή η οποία ξεκινάει από τον ανώτατο κατάλογο στο σύστημα αρχείων.

πιάσιμο: Η χρήση των δηλώσεων `try` και `except` για την αποφυγή του τερματισμού ενός προγράμματος λόγω κάποιας εξαίρεσης.

βάση δεδομένων: Ένα αρχείο του οποίου τα περιεχόμενα είναι οργανωμένα όπως ένα λεξικό με κλειδιά τα οποία αντιστοιχούν σε τιμές.

14.12 Ασκήσεις

Άσκηση 14.6. *Το άρθρωμα* `urllib` *παρέχει μεθόδους για την διαχείριση URLs και το κατέβασμα πληροφοριών από τον ιστό. Το ακόλουθο παράδειγμα κατεβάζει και εμφανίζει ένα κρυφό μήνυμα από το* `thinkpython.com`*:*

```
import urllib

conn = urllib.urlopen('http://thinkpython.com/secret.html')
for line in conn:
    print line.strip()
```

Τρέξτε αυτόν τον κώδικα και ακολουθήστε τις οδηγίες που θα δείτε εκεί. Λύση : `http://thinkpython.com/code/zip_code.py`.

Κεφάλαιο 15

Κλάσεις και αντικείμενα

Τα παραδείγματα κώδικα αυτού του κεφαλαίου είναι διαθέσιμα στην διεύθυνση `http://thinkpython.com/code/Point1.py` και οι λύσεις των ασκήσεων στην διεύθυνση `http://thinkpython.com/code/Point1_soln.py`.

15.1 Τύποι ορισμένοι από το χρήστη

Έχουμε χρησιμοποιήσει πολλούς από τους ενσωματωμένους τύπους της Python αλλά τώρα θα δούμε πως ορίζουμε ένα νέο τύπο. Σαν παράδειγμα, θα δημιουργήσουμε ένα τύπο με όνομα `Point` ο οποίος θα αναπαριστά ένα σημείο στο δισδιάστατο χώρο.

Στη μαθηματική σημειολογία, τα σημεία γράφονται συνήθως μέσα σε παρενθέσεις με ένα κόμμα που χωρίζει τις συντεταγμένες. Για παράδειγμα, το $(0, 0)$ αναπαριστά την αρχή και το (x, y) αναπαριστά το σημείο x μονάδων δεξιά και y μονάδων προς τα πάνω από την αρχή.

Υπάρχουν διάφοροι τρόποι που θα μπορούσαμε να αναπαραστήσουμε σημεία στην Python:

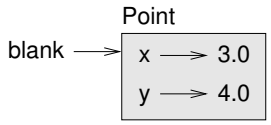
- Θα μπορούσαμε να αποθηκεύσουμε τις συντεταγμένες σε δύο μεταβλητές ξεχωριστά, την `x` και την `y`.
- Θα μπορούσαμε να αποθηκεύσουμε τις συντεταγμένες ως στοιχεία μέσα σε μία λίστα ή μία πλειάδα.
- Θα μπορούσαμε να δημιουργήσουμε ένα νέο τύπο για να αναπαραστήσουμε τα σημεία σαν αντικείμενα.

Η δημιουργία ενός νέου τύπου είναι λίγο πολυπλοκότερη σε σχέση με τις άλλες επιλογές, αλλά έχει πλεονεκτήματα τα οποία θα γίνουν εμφανή σύντομα.

Ένας τύπος ορισμένος από το χρήστη ονομάζεται επίσης και κλάση. Ένας ορισμός κλάσης μοιάζει κάπως έτσι :

```
class Point(object):
    """Represents a point in 2-D space."""
```

Αυτή η κεφαλίδα υποδεικνύει ότι η νέα κλάση είναι ένα σημείο, το οποίο είναι ένα είδος αντικειμένου, το οποίο είναι ένας ενσωματωμένος τύπος.



Σχήμα 15.1: Διάγραμμα αντικειμένου

Το σώμα είναι μία συμβολοσειρά τεκμηρίωσης η οποία εξηγεί τι κάνει η κλάση. Μπορείτε να ορίσετε μεταβλητές και συναρτήσεις μέσα σε έναν ορισμό κλάσης, αλλά θα επιστρέψουμε σε αυτό αργότερα.

Ο ορισμός μία κλάσης με όνομα `Point` δημιουργεί ένα αντικείμενο κλάσης.

```
>>> print Point
<class '__main__.Point'>
```

Επειδή η `Point` ορίζεται στην αρχή, ” το πλήρες όνομά ” της είναι `__main__.Point`.

Το αντικείμενο της κλάσης είναι ένα εργοστάσιο παραγωγής αντικειμένων. Για να δημιουργήσετε ένα σημείο θα καλέσετε την `Point` σαν να ήταν μία συνάρτηση.

```
>>> blank = Point()
>>> print blank
<__main__.Point instance at 0xb7e9d3ac>
```

Η επιστρεφόμενη τιμή είναι μία αναφορά σε ένα αντικείμενο Point το οποίο εκχωρούμε στην `blank`. Η δημιουργία ενός νέου αντικειμένου ονομάζεται ” δημιουργία στιγμιότυπου ” και το αντικείμενο ονομάζεται στιγμιότυπο της κλάσης.

Όταν εμφανίζετε ένα στιγμιότυπο, η Python σας λέει σε ποια κλάση ανήκει και που είναι αποθηκευμένο στη μνήμη (το πρόθεμα `0x` σημαίνει ότι ο αριθμός που ακολουθεί είναι δεκαεξαδικός).

15.2 Ιδιότητες

Μπορείτε να εκχωρήσετε τιμές σε ένα στιγμιότυπο χρησιμοποιώντας τον συμβολισμό με τελεία :

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

Αυτή η σύνταξη είναι παρόμοια με τη σύνταξη για την επιλογή μίας μεταβλητής από ένα άρθρωμα, όπως η `math.pi` και η `string.whitespace`. Σε αυτήν την περίπτωση όμως, εκχωρούμε τιμές στα στοιχεία ενός αντικειμένου. Αυτά τα στοιχεία ονομάζονται ιδιότητες.

Το ακόλουθο διάγραμμα δείχνει το αποτέλεσμα των εκχωρήσεων. Ένα διάγραμμα κατάστασης το οποίο δείχνει ένα αντικείμενο και και τις ιδιότητές του ονομάζεται διάγραμμα αντικειμένου (βλ. Εικόνα 15.1).

Η μεταβλητή `blank` αναφέρεται σε ένα αντικείμενο Point, το οποίο περιέχει δύο ιδιότητες. Κάθε ιδιότητα αναφέρεται σε ένα αριθμό κινητής υποδιαστολής.

Μπορείτε να διαβάσετε την τιμή μίας ιδιότητας χρησιμοποιώντας την ίδια σύνταξη :

```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
```

```
3.0
```

Η έκφραση `blank.x` σημαίνει : ” Πήγαινε στο αντικείμενο στο οποίο αναφέρεται η `blank` και πάρε την τιμή του `x`”. Σε αυτήν την περίπτωση, εκχωρούμε αυτήν την τιμή σε μία μεταβλητή με όνομα `x`. Δεν υπάρχει καμία σύγκρουση μεταξύ της μεταβλητής `x` και της ιδιότητας `x`.

Μπορείτε να χρησιμοποιήσετε το συμβολισμό με τελεία σαν μέρος οποιασδήποτε έκφρασης. Για παράδειγμα :

```
>>> print '(%g, %g)' % (blank.x, blank.y)
(3.0, 4.0)
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> print distance
5.0
```

Μπορείτε να περάσετε ένα στιγμιότυπο σαν όρισμα με το γνωστό τρόπο. Για παράδειγμα :

```
def print_point(p):
    print '(%g, %g)' % (p.x, p.y)
```

Η `print_point` παίρνει ένα Point σαν όρισμα και το εμφανίζει με μαθηματικό συμβολισμό. Για να την δοκιμάσετε, μπορείτε να περάσετε τη `blank` σαν όρισμα :

```
>>> print_point(blank)
(3.0, 4.0)
```

Μέσα στη συνάρτηση, η `p` είναι ένα ψευδώνυμο για την `blank` , άρα αν η συνάρτηση τροποποιήσει την `p` αλλάζει και η `blank`.

Άσκηση 15.1. *Γράψτε μία συνάρτηση με όνομα* `distance_between_points` *η οποία θα παίρνει δύο σημεία σαν ορίσματα και θα επιστρέφει την απόσταση μεταξύ τους.*

15.3 Ορθογώνια παραλληλόγραμμα

Μερικές φορές είναι προφανές ποιες θα πρέπει να είναι οι ιδιότητες ενός αντικειμένου, αλλά κάποιες άλλες όχι. Για παράδειγμα, φανταστείτε ότι σχεδιάζετε μία κλάση για να αναπαραστήσετε ορθογώνια παραλληλόγραμμα. Ποιες ιδιότητες θα χρησιμοποιούσατε για να καθορίσετε τη θέση και το μέγεθος ενός ορθογωνίου παραλληλογράμμου ; Για να απλοποιήσουμε τα πράγματα, υποθέστε ότι το παραλληλόγραμμο είναι είτε κάθετο είτε οριζόντιο.

Υπάρχουν τουλάχιστον δύο ενδεχόμενα :

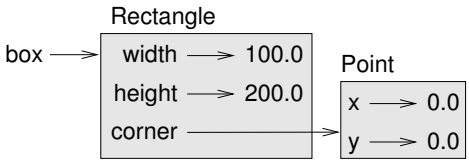
- Μπορείτε να καθορίσετε μία γωνία ή το κέντρο, το πλάτος και το ύψος.
- Μπορείτε να καθορίσετε δύο απέναντι γωνίες.

Σε αυτό το σημείο είναι δύσκολο να πούμε ποια είναι η καλύτερη επιλογή και έτσι, σαν παράδειγμα, θα υλοποιήσουμε την πρώτη.

Αυτός είναι ο ορισμός της κλάσης :

```
class Rectangle(object):
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

Η συμβολοσειρά τεκμηρίωσης απαριθμεί τις ιδιότητες : η `width` και η `height` είναι αριθμοί και η `corner` είναι ένα αντικείμενο Point η οποία ορίζει την κάτω-αριστερή γωνία.



Σχήμα 15.2: Διάγραμμα αντικειμένου.

Για να αναπαραστήσετε ένα ορθογώνιο παραλληλόγραμμο, πρέπει να δημιουργήσουμε ένα αντικείμενο Renctangle και να εκχωρήσουμε τιμές στις ιδιότητες :

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

Η έκφραση `box.corner.x` σημαίνει : ” Πήγαινε στο αντικείμενο το οποίο αναφέρεται η `box` και επέλεξε την ιδιότητα με όνομα `corner` και στη συνέχεια πήγαινε σε αυτό το αντικείμενο και επέλεξε την `x`”.

Η Εικόνα 15.2 δείχνει την κατάσταση αυτού του αντικειμένου. Ένα αντικείμενο το οποίο είναι ιδιότητα κάποιου άλλου αντικειμένου είναι ενσωματωμένο.

15.4 Τα στιγμιότυπα σαν επιστρεφόμενες τιμές

Οι συναρτήσεις μπορούν να επιστρέψουν στιγμιότυπα. Για παράδειγμα, η `find_center` παίρνει ένα `Rectangle` σαν όρισμα και επιστρέφει ένα `Point` το οποίο περιέχει τις συντεταγμένες του κέντρου του ορθογωνίου παραλληλογράμμου :

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2.0
    p.y = rect.corner.y + rect.height/2.0
    return p
```

Αυτό είναι ένα παράδειγμα το οποίο περνάει το `box` σαν όρισμα και εκχωρεί το σημείο που προκύπτει στην `center`:

```
>>> center = find_center(box)
>>> print_point(center)
(50.0, 100.0)
```

15.5 Τα αντικείμενα είναι μεταβλητά

Μπορείτε να αλλάξετε την κατάσταση ενός αντικειμένου κάνοντας μία εκχώρηση σε μία από τις ιδιότητες του. Για παράδειγμα, για να αλλάξετε το μέγεθος ενός ορθογωνίου παραλληλογράμμου χωρίς να αλλάξει η θέση του, μπορείτε να τροποποιήσετε την τιμή της `width` και της `height`:

```
box.width = box.width + 50
```

```
box.height = box.width + 100
```

Μπορείτε επίσης να γράψετε συναρτήσεις οι οποίες θα τροποποιούν αντικείμενα. Για παράδειγμα, η grow_rectangle παίρνει ένα Rectangle και δύο αριθμούς, το dwidth και dheight, και προσθέτει τους αριθμούς στο πλάτος και το ύψος του ορθογωνίου :

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Αυτό το παράδειγμα επιδεικνύει το αποτέλεσμα :

```
>>> print box.width
100.0
>>> print box.height
200.0
>>> grow_rectangle(box, 50, 100)
>>> print box.width
150.0
>>> print box.height
300.0
```

Μέσα στη συνάρτηση, το rect είναι ένα ψευδώνυμο για το box, οπότε αν η συνάρτηση τροποποιήσει το rect, αλλάζει και το box.

Άσκηση 15.2. *Γράψτε μία συνάρτηση με όνομα* move_rectangle *η οποία θα παίρνει ένα Rectangle και δύο αριθμούς με ονόματα* dx *και* dy*. Θα πρέπει να αλλάζει την θέση του ορθογωνίου παραλληλογράμμου προσθέτοντας το* dx *στη συντεταγμένη* x *του* corner *και το* dy *στη συντεταγμένη* y *του* corner*.*

15.6 Αντιγραφή

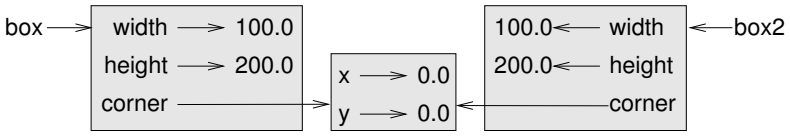
Τα ψευδώνυμα μπορεί να κάνουν ένα πρόγραμμα δύσκολο στην ανάγνωση επειδή οι αλλαγές σε ένα μέρος θα μπορούσαν να έχουν απροσδόκητες συνέπειες σε ένα άλλο μέρος. Είναι δύσκολο να παρακολουθήσουμε όλες τις μεταβλητές που θα μπορούσαν να αναφέρονται σε ένα συγκεκριμένο αντικείμενο.

Συχνά, η αντιγραφή ενός αντικειμένου είναι μία εναλλακτική λύση στην εκχώρηση ψευδωνύμων. Το άρθρωμα copy περιέχει μία συνάρτηση με όνομα copy η οποία μπορεί να αντιγράψει οποιοδήποτε αντικείμενο :

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

Το p1 και το p2 περιέχουν τα ίδια δεδομένα, αλλά είναι δύο διαφορετικά σημεία.

```
>>> print_point(p1)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)
>>> p1 is p2
False
```



Σχήμα 15.3: Διάγραμμα αντικειμένων.

```
>>> p1 == p2
False
```

Ο τελεστής `is` υποδεικνύει αυτό που περιμέναμε, ότι δηλαδή το `p1` και το `p2` δεν είναι το ίδιο αντικείμενο. Αλλά ίσως περιμένατε ότι ο τελεστής == θα απέδιδε `True` αφού αυτά τα σημεία περιέχουν τα ίδια δεδομένα. Σε αυτήν την περίπτωση, θα απογοητευθείτε, αφού για τα στιγμιότυπα η προεπιλεγμένη συμπεριφορά του τελεστή == είναι η ίδια με αυτήν του τελεστή `is`. Και οι δύο, ελέγχουν την ταυτότητα των αντικειμένων και όχι την ισοδυναμία τους. Αυτή η συμπεριφορά μπορεί να αλλάξει, θα δούμε πως αργότερα.

Αν χρησιμοποιήσετε την `copy.copy` για να αντιγράψετε ένα ορθογώνιο παραλληλόγραμμο, θα δείτε ότι αντιγράφει το αντικείμενο Rectangle αλλά όχι το ενσωματωμένο σημείο.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Η Εικόνα 15.3 δείχνει πως είναι το διάγραμμα αντικειμένων.

Αυτή η διαδικασία ονομάζεται **ρηχή αντιγραφή** επειδή αντιγράφει το αντικείμενο και όλες τις αναφορές που περιέχει, αλλά όχι τα ενσωματωμένα αντικείμενα.

Στις περισσότερες εφαρμογές όμως, αυτό δεν το θέλετε. Σε αυτό το παράδειγμα, η επίκληση της `grow_rectangle` σε ένα από τα ορθογώνια παραλληλόγραμμα δεν θα επηρέαζε το άλλο, αλλά η επίκληση της `move_rectangle` σε ένα οποιοδήποτε θα επηρέαζε και τα δύο. Αυτή η συμπεριφορά προκαλεί σύγχυση και είναι επιρρεπή σε λάθη.

Ευτυχώς, το άρθρωμα `copy` περιέχει μία μέθοδο με όνομα `deepcopy` η οποία δεν αντιγράφει μόνο το αντικείμενο αλλά και τα αντικείμενα στα οποία αναφέρεται, τα αντικείμενα στα οποία αναφέρονται αυτά και ούτω καθεξής. Όπως θα περιμένατε, αυτή η διαδικασία ονομάζεται βαθιά αντιγραφή.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

Το `box3` και το `box` είναι δύο τελείως διαφορετικά αντικείμενα.

Άσκηση 15.3. *Γράψτε μία έκδοση της* `move_rectangle` *η οποία θα δημιουργεί και θα επιστρέφει ένα νέο Rectangle αντί να τροποποιεί το παλιό.*

15.7 Αποσφαλμάτωση

Όταν αρχίσετε να δουλεύετε με αντικείμενα, είναι πιθανό να αντιμετωπίσετε κάποιες νέες εξαιρέσεις. Αν δοκιμάσετε να αποκτήσετε πρόσβαση σε μία ιδιότητα η οποία δεν υπάρχει θα πάρετε ένα `AttributeError`:

```
>>> p = Point()
>>> print p.z
AttributeError: Point instance has no attribute 'z'
```

Αν δεν είστε σίγουροι για τον τύπο του αντικειμένου μπορείτε να ρωτήσετε :

```
>>> type(p)
<type '__main__.Point'>
```

Αν δεν είστε σίγουροι για το αν ένα αντικείμενο έχει μία συγκεκριμένη ιδιότητα, μπορείτε να χρησιμοποιήσετε την ενσωματωμένη συνάρτηση `hasattr`:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

Το πρώτο όρισμα μπορεί να είναι ένα οποιοδήποτε αντικείμενο και το δεύτερο είναι μία συμβολοσειρά η οποία περιέχει το όνομα της ιδιότητας.

15.8 Ορολογία

κλάση: Ένας τύπος ορισμένος από το χρήστη. Ο ορισμός μίας κλάσης δημιουργεί ένα νέο αντικείμενο κλάσης.

αντικείμενο κλάσης: Ένα αντικείμενο το οποίο περιέχει πληροφορίες σχετικά με έναν τύπο ορισμένο από το χρήστη. Το αντικείμενο της κλάσης μπορεί να χρησιμοποιηθεί για να δημιουργηθούν στιγμιότυπα αυτού του τύπου.

στιγμιότυπο: Ένα αντικείμενο το οποίο ανήκει σε μία κλάση.

ιδιότητα: Μία από τις μεταβλητές που σχετίζονται με ένα αντικείμενο.

ενσωματωμένο αντικείμενο: Ένα αντικείμενο το οποίο αποθηκεύεται σαν ιδιότητα ενός άλλου αντικειμένου.

ρηχή αντιγραφή: Η αντιγραφή των περιεχομένων ενός αντικειμένου, συμπεριλαμβανομένων και των αναφορών στα ενσωματωμένα αντικείμενα. Υλοποιείται με τη συνάρτηση `copy` του αρθρώματος `copy`.

βαθιά αντιγραφή: Η αντιγραφή των περιεχομένων ενός αντικειμένου καθώς επίσης και των ενσωματωμένων αντικειμένων, και των αντικειμένων που είναι ενσωματωμένα σε αυτά και ούτω καθεξής. Υλοποιείται από τη συνάρτηση `deepcopy` του αρθρώματος `copy`.

διάγραμμα αντικειμένων: Ένα διάγραμμα το οποίο δείχνει τα αντικείμενα, τις ιδιότητές τους και τις τιμές των ιδιοτήτων.

15.9 Ασκήσεις

Άσκηση 15.4. *Το πακέτο Swampy (βλ. Κεφάλαιο 4) παρέχει ένα άρθρωμα με όνομα* `World`*, το οποίο ορίζει έναν τύπο ορισμένο από το χρήστη, το όνομα του οποίου είναι επίσης* `World`*. Μπορείτε να τον εισάγετε έτσι :*

```
from swampy.World import World
```

Ή, ανάλογα με το πως εγκαταστήσατε το Swampy, έτσι :

```
from World import World
```

Ο ακόλουθος κώδικας δημιουργεί ένα αντικείμενο World και καλεί τη μέθοδο `mainloop` *, η οποία περιμένει το χρήστη.*

```
world = World()
world.mainloop()
```

Θα πρέπει να εμφανιστεί ένα παράθυρο με μία γραμμή τίτλου και ένα άδειο τετράγωνο. Θα χρησιμοποιήσουμε αυτό το παράθυρο για να σχεδιάσουμε σημεία (Points), ορθογώνια παραλληλόγραμμα (Rectanlges) και άλλα σχήματα. Προσθέστε τις παρακάτω γραμμές προτού καλέσετε την `mainloop` *και ξανατρέξτε το πρόγραμμα.*

```
canvas = world.ca(width=500, height=500, background='white')
bbox = [[-150,-100], [150, 100]]
canvas.rectangle(bbox, outline='black', width=2, fill='green4')
```

Θα πρέπει να δείτε ένα πράσινο ορθογώνιο παραλληλόγραμμο με μαύρο περίγραμμα. Η πρώτη γραμμή δημιουργεί έναν καμβά, ο οποίος εμφανίζεται στο παράθυρο σαν ένα λευκό τετράγωνο. Το αντικείμενο Canvas παρέχει μεθόδους όπως η `rectangle` *για το σχεδιασμό διαφόρων σχημάτων.*

Η `bbox` *είναι μία λίστα από λίστες η οποία αναπαριστά το " πλαίσιο οριοθέτησης " του ορθογωνίου παραλληλογράμμου. Το πρώτο ζευγάρι συντεταγμένων είναι η κάτω-αριστερή γωνία του ορθογωνίου και το δεύτερο ζευγάρι είναι η πάνω-δεξιά γωνία.*

Μπορείτε να σχεδιάσετε ένα κύκλο κάπως έτσι :

```
canvas.circle([-25,0], 70, outline=None, fill='red')
```

Η πρώτη παράμετρος είναι το ζευγάρι συντεταγμένων για το κέντρο του κύκλου και η δεύτερη είναι η ακτίνα.

Αν προσθέσετε αυτή τη γραμμή στο πρόγραμμα, το αποτέλεσμα θα πρέπει να μοιάζει με την σημαία του Μπανγκλαντές (βλ. `http://en.wikipedia.org/wiki/Gallery_of_sovereign-state_flags` *).*

1. *Γράψτε μία συνάρτηση με όνομα* `draw_rectangle` *η οποία θα παίρνει ένα αντικείμενο Canvas και και ένα Rectangle σαν ορίσματα και θα σχεδιάζει μία αναπαράσταση του Rectangle στο Canvas.*

2. *Προσθέστε μία ιδιότητα με όνομα* `color` *στα αντικείμενα Rectangle και τροποποιήστε την* `draw_rectangle` *έτσι ώστε να χρησιμοποιεί αυτήν την ιδιότητα σαν χρώμα γεμίσματος.*

3. *Γράψτε μία συνάρτηση με όνομα* `draw_point` *η οποία θα παίρνει ένα αντικείμενο Canvas και ένα Point σαν ορίσματα και θα σχεδιάζει μία αναπαράσταση του Point στο Canvas.*

4. *Ορίστε μία νέα κλάση με όνομα Circle με τις κατάλληλες ιδιότητες και δημιουργήστε μερικά αντικείμενα Circle. Γράψτε μία συνάρτηση με όνομα* `draw_circle` *η οποία θα σχεδιάζει κύκλους στον καμβά.*

5. *Γράψτε ένα πρόγραμμα το οποίο θα σχεδιάζει τη σημαία της Τσεχίας. Σημείωση : μπορείτε να σχεδιάσετε ένα πολύγωνο κάπως έτσι :*

```
points = [[-150,-100], [150, 100], [150, -100]]
canvas.polygon(points, fill='blue')
```

Έχω γράψει ένα μικρό πρόγραμμα το οποίο απαριθμεί τα διαθέσιμα χρώματα. Μπορείτε να το κατεβάσετε από εδώ : `http://thinkpython.com/code/color_list.py`.

Κεφάλαιο 16

Κλάσεις και συναρτήσεις

Τα παραδείγματα αυτού του κεφαλαίου είναι διαθέσιμα στο σύνδεσμο : `http://thinkpython.com/code/Time1.py`.

16.1 Ώρα

Σαν ένα ακόμα παράδειγμα τύπου οριζόμενο από το χρήστη, θα ορίσουμε μία κλάση με όνομα `Time` η οποία θα καταγράφει την ώρα της ημέρας. Ο ορισμός της κλάσης είναι ο εξής :

```
class Time(object):
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

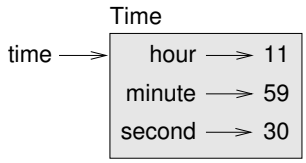
Μπορούμε να δημιουργήσουμε ένα νέο αντικείμενο `Time` και να εκχωρήσουμε ιδιότητες για τις ώρες, τα λεπτά και τα δευτερόλεπτα :

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

Το διάγραμμα κατάστασης για το αντικείμενο `Time` είναι όπως αυτό στην Εικόνα 16.1.

Άσκηση 16.1. *Γράψτε μία συνάρτηση με όνομα* `print_time` *η οποία θα παίρνει ένα αντικείμενο Time και θα το εμφανίζει στη μορφή* `hour:minute:second`*. Σημείωση : η ακολουθία διαμόρφωσης* `'%.2d'` *εμφανίζει έναν ακέραιο χρησιμοποιώντας τουλάχιστον δύο ψηφία, συμπεριλαμβανομένου και ενός μηδενικού στην αρχή αν είναι απαραίτητο.*

Άσκηση 16.2. *Γράψτε μία λογική συνάρτηση με όνομα* `is_after` *η οποία θα παίρνει δύο αντικείμενα Time, το* `t1` *και το* `t2`*, και θα επιστρέφει* `True` *αν το* `t1` *ακολουθεί το* `t2` *χρονολογικά και* `False` *αλλιώς. Πρόκληση : μην χρησιμοποιήσετε τη δήλωση* `if`*.*



Σχήμα 16.1: Διάγραμμα αντικειμένου.

16.2 Αγνές συναρτήσεις

Στις επόμενες ενότητες, θα γράψουμε δύο συναρτήσεις οι οποίες θα προσθέτουν τιμές ώρας. Αυτές επιδεικνύουν δύο είδη συναρτήσεων : τις αγνές συναρτήσεις και τις συναρτήσεις τροποποίησης. Επίσης, επιδεικνύουν και ένα πλάνο ανάπτυξης, το οποίο αποκαλώ ” πρωτότυπο και επιδιόρθωση ”, το οποίο αποτελεί ένα τρόπο αντιμετώπισης ενός σύνθετου προβλήματος ξεκινώντας με ένα απλό πρωτότυπο και αντιμετωπίζοντας σταδιακά την πολυπλοκότητα.

Αυτό είναι ένα απλό πρωτότυπο της `add_time`:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

Η συνάρτηση δημιουργεί ένα νέο αντικείμενο `Time`, αρχικοποιεί τις ιδιότητές του και επιστρέφει μία αναφορά σε ένα νέο αντικείμενο. Αυτή ονομάζεται αγνή συνάρτηση επειδή δεν τροποποιεί κανένα από τα αντικείμενα που της περάστηκαν σαν ορίσματα και δεν έχει και καμία επίδραση, όπως να εμφανίζει μία τιμή ή να δέχεται είσοδο από το χρήστη, εκτός από την επιστρεφόμενη τιμή.

Για να δοκιμάσουμε αυτήν τη συνάρτηση, θα δημιουργήσω δύο αντικείμενα Time: το `start` περιέχει την ώρα έναρξης μιας ταινίας, όπως η *Monty Python and the Holy Grail*, και το `duration` περιέχει την διάρκεια της ταινίας, η οποία είναι μία ώρα και 35 λεπτά.

Η `add_time` υπολογίζει πότε θα τελειώσει η ταινία :

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

Το αποτέλεσμα, `10:80:00`, ίσως να μην είναι αυτό που θα περιμένατε. Το πρόβλημα έγκειται στο ότι αυτή η συνάρτηση δεν χειρίζεται τις περιπτώσεις όπου ο αριθμός των δευτερολέπτων ή των λεπτών γίνεται μεγαλύτερος από εξήντα. Όταν συμβαίνει αυτό, θα πρέπει να ” μεταφέρουμε ” τα έξτρα δευτερόλεπτα στη στήλη των λεπτών ή τα έξτρα λεπτά στη στήλη της ωρών.

Αυτή είναι μία βελτιωμένη έκδοση :

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

Παρόλο που αυτή η συνάρτηση είναι σωστή, αρχίζει να γίνεται μεγάλη. Θα δούμε μία άλλη εναλλακτική λύση αργότερα.

16.3 Συναρτήσεις τροποποίησης

Μερικές φορές είναι χρήσιμο μια συνάρτηση να τροποποιεί τα αντικείμενα που παίρνει σαν παραμέτρους. Σε αυτήν την περίπτωση, οι αλλαγές είναι ορατές στον καλούντα. Οι συναρτήσεις που δουλεύουν κατ' αυτόν τον τρόπο ονομάζονται " συναρτήσεις τροποποίησης ".

Η `increment`, η οποία προσθέτει ένα δεδομένο αριθμό δευτερολέπτων σε ένα αντικείμενο `Time`, μπορεί να γραφτεί φυσικά σαν μία συνάρτηση τροποποίησης. Αυτό είναι ένα προσχέδιο :

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

Η πρώτη γραμμή εκτελεί τη βασική λειτουργία και το υπόλοιπο διαχειρίζεται τις ειδικές περιπτώσεις που είδαμε νωρίτερα.

Είναι σωστή αυτή η συνάρτηση ; Τι συμβαίνει αν η παράμετρος `seconds` είναι πολύ μεγαλύτερη από εξήντα ;

Σε αυτήν την περίπτωση δεν είναι αρκετό να μεταφέρουμε μία φορά, πρέπει να συνεχίσουμε μέχρι η `time.second` να γίνει μικρότερη από εξήντα. Μία λύση είναι να αντικαταστήσουμε τις δηλώσεις `if` με δηλώσεις `while`. Αυτό θα κάνει την συνάρτηση σωστή αλλά όχι και πολύ αποδοτική.
Άσκηση 16.3. *Γράψτε μία σωστή έκδοση της* `increment` *η οποία δεν θα περιέχει κανένα βρόχο.*

Οτιδήποτε μπορεί να υλοποιηθεί με συναρτήσεις τροποποίησης μπορεί επίσης να υλοποιηθεί και με αγνές συναρτήσεις. Στην πραγματικότητα, μερικές γλώσσες προγραμματισμού επιτρέπουν μόνο αγνές

συναρτήσεις. Υπάρχουν κάποιες ενδείξεις ότι τα προγράμματα που χρησιμοποιούν αγνές συναρτήσεις είναι πιο γρήγορα στην ανάπτυξη και λιγότερο επιρρεπή στα λάθη σε σχέση με τα προγράμματα που χρησιμοποιούν συναρτήσεις τροποποίησης. Αλλά, μερικές φορές, οι συναρτήσεις τροποποίησης είναι πολύ βολικές και λειτουργικά προγράμματα έχουν την τάση να είναι λιγότερο αποδοτικά.

Σε γενικές γραμμές, συνιστώ να γράφετε αγνές συναρτήσεις όπου είναι λογικό και να καταφεύγετε στις συναρτήσεις τροποποίησης μόνο όταν υπάρχει σημαντικό πλεονέκτημα. Αυτή η προσέγγιση μπορεί να χαρακτηριστεί ως " συναρτησιακό ύφος προγραμματισμού ".
Άσκηση 16.4. *Γράψτε μία "αγνή" έκδοση της* `increment` *η οποία θα δημιουργεί και θα επιστρέφει ένα νέο αντικείμενο Time αντί να τροποποιεί την παράμετρο.*

16.4 Πρωτοτυποποίηση εναντίον σχεδιασμού

Το πλάνο ανάπτυξης το οποίο επιδεικνύω ονομάζεται " πρωτότυπο και ενημέρωση ". Για κάθε συνάρτηση, έγραφα ένα πρωτότυπο το οποίο εκτελούσε το βασικό υπολογισμό και στη συνέχεια το δοκίμαζα, διορθώνοντας λάθη στην πορεία.

Αυτή η προσέγγιση μπορεί να είναι αποτελεσματική, ειδικά αν δεν έχετε κατανοήσει καλά το πρόβλημα εξαρχής. Αλλά η σταδιακή ενημέρωση μπορεί να δημιουργήσει κώδικα ο οποίος θα είναι ασκόπως περίπλοκος (αφού θα εξετάζει πολλές ειδικές περιπτώσεις) και αναξιόπιστος (από τη στιγμή που είναι δύσκολο να γνωρίζετε αν πρέπει να βρείτε όλα τα σφάλματα).

Μία εναλλακτική λύση είναι η ανάπτυξη βάση σχεδίου, κατά την οποία η υψηλού επιπέδου επίγνωση του προβλήματος μπορεί να κάνει τον προγραμματισμό ευκολότερο. Σε αυτήν την περίπτωση, η επίγνωση έγκειται στο ότι το αντικείμενο Time είναι ένας τριψήφιος αριθμός στη βάση του 60 (βλ. `http://en.wikipedia.org/wiki/Sexagesimal`)! Η ιδιότητα `second` είναι η " στήλη των μονάδων ", η ιδιότητα `minute` είναι η " στήλη των εξήντα " και η ιδιότητα `hour` είναι η " η στήλη των τριάντα έξι εκατοντάδων ".

Όταν γράψαμε την `add_time` και την `increment`, ουσιαστικά κάναμε πρόσθεση στην βάση του 60, αυτός είναι και ο λόγος που έπρεπε να κάνουμε μεταφορά από τη μία στήλη στην επόμενη.

Αυτή η διαπίστωση μας υποδεικνύει μία άλλη προσέγγιση του όλου προβλήματος. Μπορούμε να μετατρέψουμε τα αντικείμενα Time σε ακέραιους αριθμούς και να επωφεληθούμε από το γεγονός ότι ο υπολογιστής ξέρει να κάνει αριθμητική ακεραίων.

Αυτή είναι μία συνάρτηση η οποία μετατρέπει αντικείμενα Time σε ακέραιους :

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

Και αυτή είναι η συνάρτηση που μετατρέπει τους ακέραιους σε αντικείμενα Time (θυμηθείτε ότι η `divmod` διαιρεί το πρώτο όρισμα με το δεύτερο και επιστρέφει το πηλίκο και το υπόλοιπο σαν μία πλειάδα).

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

Ίσως πρέπει να σκεφτείτε λίγο και να τρέξετε μερικά παραδείγματα, ούτως ώστε να πειστείτε ότι αυτές οι συναρτήσεις είναι σωστές. Ένας τρόπος για να τις δοκιμάσετε είναι να τσεκάρετε ότι

`time_to_int(int_to_time(x)) == x` για διάφορες τιμές της `x`. Αυτό είναι ένα παράδειγμα ενός ελέγχου συνέπειας.

Αφού πειστείτε ότι είναι σωστές, μπορείτε να τις χρησιμοποιήσετε για να ξαναγράψετε την `add_time`:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Αυτή η έκδοση είναι μικρότερη από την αρχική και ευκολότερη στην επαλήθευση
Άσκηση 16.5. *Ξαναγράψτε την* `increment` *χρησιμοποιώντας τις* `time_to_int` *και* `int_to_time`.

Σε μερικές περιπτώσεις, η μετατροπή από τη βάση του 60 στη βάση του 10 και πίσω είναι δυσκολότερη σε σχέση με τον χειρισμό ωρών. Η μετατροπή βάσης είναι κάπως αφηρημένη διαδικασία σε σχέση με τον απλό χειρισμό των τιμών ώρας.

Αλλά αν έχουμε την διορατικότητα να μεταχειριστούμε τις ώρες στη βάση του 60 και επενδύσουμε στο γράψιμο των συναρτήσεων μετατροπής (`time_to_int` και `int_to_time`), τότε θα πάρουμε ένα πρόγραμμα το οποίο θα είναι μικρότερο, ευκολότερο στο διάβασμα και την αποσφαλμάτωση και πιο αξιόπιστο.

Επίσης είναι ευκολότερο να προσθέτουμε λειτουργίες εκ των υστέρων. Για παράδειγμα, φανταστείτε να αφαιρούσατε δύο αντικείμενα Time για να βρείτε τη διάρκεια μεταξύ τους. Η αφελής προσέγγιση θα ήταν να υλοποιούσαμε αφαίρεση με δανεισμό. Χρησιμοποιώντας τις συναρτήσεις μετατροπής θα ήταν ευκολότερο και ορθότερο.

Μερικές φορές, κατά ειρωνικό τρόπο, το να κάνουμε ένα πρόβλημα δυσκολότερο (ή πιο γενικό) το κάνει ευκολότερο (επειδή υπάρχουν λιγότερες ειδικές περιπτώσεις και λιγότερες πιθανότητες για σφάλματα).

16.5 Αποσφαλμάτωση

Ένα αντικείμενο Time είναι καλά διαμορφωμένο αν οι τιμές της `minute` και της `second` είναι μεταξύ του 0 και του 60 (συμπεριλαμβανομένου του 0 αλλά όχι του 60) και αν η `hour` είναι θετική. Η `hour` και η `minute` θα πρέπει να είναι ακέραιες τιμές, αλλά η θα μπορούσε να έχει ένα κλασματικό μέρος.

Αυτού του είδους οι απαιτήσεις ονομάζονται ” αμετάβλητες ” επειδή πρέπει να είναι πάντα αληθείς. Για να το θέσουμε διαφορετικά, αν δεν είναι αληθείς, τότε κάτι πήγε στραβά.

Το γράψιμο κώδικα για τον έλεγχο των αμετάβλητων συνθηκών μπορεί να σας βοηθήσει να εντοπίσετε λάθη και να βρείτε τις αιτίες τους. Για παράδειγμα, μπορεί να έχετε μία συνάρτηση όπως η `valid_time` η οποία παίρνει ένα αντικείμενο Time και επιστρέφει `False` αν παραβιάζεται μία αμετάβλητη συνθήκη :

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

Στην συνέχεια, στην αρχή κάθε συνάρτησης, μπορείτε να ελέγξετε τα ορίσματα για να σιγουρευτείτε ότι είναι έγκυρα :

```
def add_time(t1, t2):
```

```
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError, 'invalid Time object in add_time'
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Ή μπορείτε να χρησιμοποιήσετε μία δήλωση `assert`, η οποία ελέγχει μία δεδομένη σταθερά και εγείρει μία εξαίρεση αν αποτύχει :

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Οι δηλώσεις `assert` είναι χρήσιμες επειδή ξεχωρίζουν τον κώδικα που ασχολείται με τις κανονικές συνθήκες από τον κώδικα που ελέγχει για σφάλματα.

16.6 Ορολογία

πρωτότυπο και ενημέρωση: Ένα πλάνο ανάπτυξης το οποίο περιλαμβάνει το γράψιμο ενός προσχεδίου του προγράμματος, τις δοκιμές και τη διόρθωση σφαλμάτων όταν και αν εντοπιστούν.

ανάπτυξη βάση σχεδίου: Ένα πλάνο ανάπτυξης το οποίο περιλαμβάνει υψηλού επιπέδου επίγνωση του προβλήματος και περισσότερο σχεδιασμό σε σχέση με τη σταδιακή ανάπτυξη ή ανάπτυξη βάση πρωτότυπου.

αγνή συνάρτηση: Μία συνάρτηση η οποία δεν τροποποιεί κανένα από τα αντικείμενα που δέχεται σαν ορίσματα. Οι περισσότερες αγνές συναρτήσεις είναι γόνιμες.

συνάρτηση τροποποίησης: Μία συνάρτηση η οποία μεταβάλλει ένα ή περισσότερα από τα αντικείμενα που δέχεται σαν ορίσματα. Οι περισσότερες συναρτήσεις τροποποίησης είναι άκαρπες.

συναρτησιακό ύφος προγραμματισμού: Ένα ύφος σχεδίασης προγράμματος στο οποίο η πλειονότητα των συναρτήσεων είναι αγνές.

σταθερή: Μία συνθήκη η οποία θα πρέπει να είναι συνεχώς αληθής κατά την εκτέλεση ενός προγράμματος.

16.7 Ασκήσεις

Τα παραδείγματα κώδικα αυτού του κεφαλαίου είναι διαθέσιμα στο σύνδεσμο `http://thinkpython.com/code/Time1.py` και οι λύσεις αυτών των ασκήσεων στο σύνδεσμο `http://thinkpython.com/code/Time1_soln.py`.

Άσκηση 16.6. *Γράψτε μία συνάρτηση με όνομα* `mul_time` *η οποία θα παίρνει ένα αντικείμενο Time και έναν αριθμό και θα επιστρέφει ένα νέο αντικείμενο Time το οποίο θα περιέχει το γινόμενο του αρχικού αντικειμένου και του αριθμού.*

Στη συνέχεια χρησιμοποιήστε την `mul_time` *για να γράψετε μία συνάρτηση η οποία θα παίρνει ένα αντικείμενο Time το οποίο θα αντιπροσωπεύει το χρόνο τερματισμού ενός αγώνα και έναν αριθμό ο οποίος θα αντιπροσωπεύει την απόσταση, και θα επιστρέφει ένα αντικείμενο Time το οποίο θα αντιπροσωπεύει το μέσο ρυθμό (χρόνος ανά μίλι).*

Άσκηση 16.7. *Το άρθρωμα* `datetime` *παρέχει τα αντικείμενα* `date` *και* `time` *τα οποία είναι παρόμοια με τα Date και Time αυτού του κεφαλαίου με τη διαφορά ότι παρέχουν ένα ευρύ σύνολο μεθόδων και τελεστών. Διαβάστε την τεκμηρίωση στο σύνδεσμο* `http://docs.python.org/2/library/datetime.html`.

1. *Χρησιμοποιήστε το άρθρωμα* `datetime` *για να γράψετε ένα πρόγραμμα το οποίο θα παίρνει την τρέχουσα ημερομηνία και θα εμφανίζει τη μέρα της εβδομάδας*
2. *Γράψτε ένα πρόγραμμα το οποίο θα παίρνει σαν είσοδο μία ημερομηνία γέννησης και θα εμφανίζει την ηλικία του χρήστη και τον αριθμό των ημερών, ωρών, λεπτών και δευτερολέπτων μέχρι την επόμενη ημέρα γενεθλίων.*
3. *Για δύο άτομα που έχουν γεννηθεί σε διαφορετικές ημερομηνίες, υπάρχει μία ημέρα στην οποία ο ένας είναι δύο φορές μεγαλύτερος από τον άλλο. Αυτή είναι η " Διπλή τους Μέρα ". Γράψτε ένα πρόγραμμα το οποίο θα παίρνει δύο ημερομηνίες γέννησης και θα υπολογίζει την Διπλή τους Μέρα.*
4. *Για μεγαλύτερη πρόκληση, γράψτε μία πιο γενική έκδοση η οποία θα υπολογίζει την ημέρα όπου το ένα άτομο είναι n φορές μεγαλύτερο από το άλλο*

Κεφάλαιο 17

Κλάσεις και μέθοδοι

Τα παραδείγματα αυτού του κεφαλαίου είναι διαθέσιμα στο σύνδεσμο : `http://thinkpython.com/code/Time2.py`.

17.1 Αντικειμενοστραφή χαρακτηριστικά

Η Python είναι μία αντικειμενοστραφής γλώσσα προγραμματισμού, το οποίο σημαίνει ότι παρέχει λειτουργίες οι οποίες υποστηρίζουν αντικειμενοστραφή προγραμματισμό.

Δεν είναι εύκολο να προσδιορίσουμε τον αντικειμενοστραφή προγραμματισμό, αλλά έχουμε ήδη δει κάποια από τα χαρακτηριστικά του :

- Τα προγράμματα αποτελούνται από ορισμούς αντικειμένων και συναρτήσεων, και οι περισσότεροι υπολογισμοί εκφράζονται με πράξεις επί των αντικειμένων.
- Κάθε ορισμός αντικειμένου αντιστοιχεί σε κάποιο αντικείμενο ή έννοια του πραγματικού κόσμου και οι συναρτήσεις που λειτουργούν πάνω σε αυτά τα αντικείμενα αντιστοιχούν στους τρόπους που αυτά τα αντικείμενα αλληλεπιδρούν μεταξύ τους στον πραγματικό κόσμο.

Για παράδειγμα, η κλάση `Time` η οποία ορίστηκε στο Κεφάλαιο 16 αντιστοιχεί στον τρόπο με τον οποίο οι άνθρωποι καταγράφουν την ώρα της ημέρας και οι συναρτήσεις που ορίσαμε αντιστοιχούν στα είδη των πραγμάτων που κάνουν οι άνθρωποι με τις ώρες. Παρομοίως, οι κλάσεις `Point` και `Rectangle` αντιστοιχούν στις μαθηματικές έννοιες ενός σημείου και ενός ορθογωνίου παραλληλογράμμου.

Μέχρι στιγμής, δεν έχουμε επωφεληθεί από τις λειτουργίες που παρέχει Python για να υποστηρίξει αντικειμενοστραφή προγραμματισμό. Αυτές οι λειτουργίες δεν είναι απολύτως αναγκαίες, αφού οι περισσότερες από αυτές παρέχουν μία εναλλακτική σύνταξη για πράγματα τα οποία έχουμε ήδη κάνει. Αλλά σε πολλές περιπτώσεις, η εναλλακτική λύση είναι πιο λακωνική και εκφράζει με μεγαλύτερη ακρίβεια την δομή του προγράμματος.

Για παράδειγμα, στο πρόγραμμα `Time`, δεν υπάρχει κάποια φανερή σύνδεση μεταξύ του ορισμού της κλάσης και των ορισμών των συναρτήσεων που ακολουθούν. Εξετάζοντας όμως καλύτερα τον κώδικα, προκύπτει ότι κάθε συνάρτηση παίρνει τουλάχιστον ένα αντικείμενο `Time` σαν όρισμα.

Αυτή η παρατήρηση αποτελεί το κίνητρο για τις μεθόδους, οι οποίες είναι συναρτήσεις οι οποίες σχετίζονται με μία συγκεκριμένη κλάση. Έχουμε δει μεθόδους για τις συμβολοσειρές, τις λίστες, τα

λεξικά και τις πλειάδες. Σε αυτό το κεφάλαιο, θα ορίσουμε μεθόδους για τύπους ορισμένους από τον χρήστη.

Οι μέθοδοι είναι σημασιολογικά ίδιες με τις συναρτήσεις, αλλά υπάρχουν δύο συντακτικές διαφορές :

- Οι μέθοδοι ορίζονται μέσα στον ορισμό μιας κλάσης προκειμένου να γίνει σαφής η σχέση μεταξύ της κλάσης και της μεθόδου.
- Η σύνταξη για την επίκληση μίας μεθόδου είναι διαφορετική από τη σύνταξη για την κλήση μίας συνάρτησης.

Στις επόμενες ενότητες, θα πάρουμε τις συναρτήσεις των δύο προηγουμένων κεφαλαίων και θα τις μετατρέψουμε σε μεθόδους. Αυτή η μετατροπή είναι καθαρά μηχανική και μπορείτε να την κάνετε απλά ακολουθώντας μία σειρά βημάτων. Αν είστε εξοικειωμένοι με την μετατροπή από μία μορφή σε άλλη, τότε είστε σε θέση να διαλέξετε την καλύτερη μορφή για οτιδήποτε κάνετε.

17.2 Εκτύπωση αντικειμένων

Στο Κεφάλαιο 16, ορίσαμε μία κλάση με όνομα `Time` και στην Άσκηση 16.1, γράψατε μία συνάρτηση με όνομα `print_time`:

```
class Time(object):
    """Represents the time of day."""

def print_time(time):
    print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

Για να καλέσετε αυτήν την συνάρτηση, πρέπει να τις περάσετε ένα αντικείμενο `Time` σαν όρισμα :

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

Για να κάνουμε την `print_time` μία μέθοδο, το μόνο που πρέπει να κάνουμε είναι να μεταφέρουμε τον ορισμό της συνάρτησης μέσα στον ορισμό της κλάσης. Παρατηρήστε την αλλαγή στην εσοχή.

```
class Time(object):
    def print_time(time):
        print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

Τώρα υπάρχουν δύο τρόποι να καλέσουμε την `print_time`. Ο πρώτος (και πιο σπάνιος) είναι να χρησιμοποιήσουμε τη σύνταξη συνάρτησης :

```
>>> Time.print_time(start)
09:45:00
```

Σε αυτήν την χρήση του συμβολισμού με τελεία, το `Time` είναι το όνομα της κλάσης και το `print_time` είναι το όνομα της μεθόδου. Η `start` περνιέται σαν παράμετρος.

Ο δεύτερος (και πιο συνοπτικός) τρόπος είναι να χρησιμοποιήσουμε τη σύνταξη μεθόδου :

```
>>> start.print_time()
09:45:00
```

Σε αυτή τη χρήση του συμβολισμού με τελεία, το `print_time` είναι το όνομα της μεθόδου (πάλι) και το `start` είναι το αντικείμενο πάνω στο οποίο επικαλείται η μέθοδος, και ονομάζεται υποκείμενο. Όπως το υποκείμενο μίας πρότασης είναι αυτό για το οποίο γίνεται λόγος στην πρόταση, έτσι και το υποκείμενο μίας επίκλησης μεθόδου είναι αυτό υποδηλώνει για τι περίπου πρόκειται η μέθοδος.

Μέσα στη μέθοδο, το υποκείμενο εκχωρείται στην πρώτη παράμετρο, άρα σε αυτήν την περίπτωση το `start` εκχωρείται στην `time`.

Κατά συνθήκη, η πρώτη παράμετρος μίας μεθόδου ονομάζεται `self`, άρα θα ήταν πιο σωστό να γράφαμε την `print_time` με αυτόν τον τρόπο :

```
class Time(object):
    def print_time(self):
        print '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

Ο λόγος για αυτήν την συνθήκη είναι μία έμμεση μεταφορά :

- Η σύνταξη για μία κλήση συνάρτησης, `print_time(start)`, υποδηλώνει ότι η συνάρτηση είναι ο παράγοντας που που ενεργεί. Λέει δηλαδή κάτι τέτοιο : ”Έι `print_time`! αυτό είναι ένα αντικείμενο για να εκτυπώσεις ”.
- Στον αντικειμενοστραφή προγραμματισμό, τα αντικείμενα είναι οι ενεργοί παράγοντες. Μία επίκληση μεθόδου όπως η `start.print_time()` λέει : ”Έι `start`! εκτύπωσε τον εαυτό σου παρακαλώ ”.

Αυτή η αλλαγή μελλοντικά μπορεί να σας φανεί πιο βολική, αλλά ακόμα δεν είναι προφανές ότι είναι χρήσιμη. Στα παραδείγματα που είδαμε μέχρι τώρα μπορεί να μην είναι, αλλά μερικές φορές η μετάθεση ευθύνης από τις συναρτήσεις στα αντικείμενα μας δίνει την δυνατότητα να γράψουμε πιο ευέλικτες συναρτήσεις και κάνει ευκολότερη την συντήρηση και την επαναχρησιμοποίηση του κώδικα.

Άσκηση 17.1. *Ξαναγράψτε την* `time_to_int` *(από την Ενότητα 16.4) σαν μέθοδο. Πιθανόν η* `int_to_time` *να μην είναι κατάλληλη για να γραφτεί σαν μέθοδος. Σε τι αντικείμενο θα επικαλούνταν ;*

17.3 Ένα ακόμη παράδειγμα

Αυτή είναι μία έκδοση της `increment` (από την Ενότητα 16.3) γραμμένη σαν μέθοδος :

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

Αυτή η έκδοση υποθέτει ότι η `time_to_int` έχει γραφτεί σαν μέθοδος, όπως στην Άσκηση 17.1. Επίσης, προσέξτε ότι είναι μία αγνή συνάρτηση, όχι μία συνάρτηση τροποποίησης.

Εδώ φαίνεται πως θα επικαλεστείτε την `increment`:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
```

```
>>> end.print_time()
10:07:17
```

Το υποκείμενο `start` εκχωρείται στην πρώτη παράμετρο, τη `self`. Το όρισμα `1337` εκχωρείται στην δεύτερη παράμετρο, την `seconds`.

Αυτός ο μηχανισμός μπορεί να προκαλέσει σύγχυση, ειδικά αν κάνετε ένα λάθος. Για παράδειγμα, αν επικαλεστείτε την `increment` με δύο ορίσματα, θα πάρετε :

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes exactly 2 arguments (3 given)
```

Το μήνυμα λάθους προκαλεί σύγχυση αρχικά, επειδή υπάρχουν μόνο δύο ορίσματα στις παρενθέσεις. Αλλά το υποκείμενο θεωρείται επίσης ένα όρισμα, άρα όλα μαζί είναι τρία.

17.4 Ένα πιο σύνθετο παράδειγμα

Η `is_after` (από την Άσκηση 16.2) είναι ελαφρώς πιο σύνθετη επειδή παίρνει δύο αντικείμενα Time σαν παραμέτρους. Σε αυτήν την περίπτωση συνηθίζεται να ονομάζουμε την πρώτη παράμετρο `self` και την δεύτερη `other`:

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Για να χρησιμοποιήσετε αυτήν τη μέθοδο, πρέπει να την επικαλεστείτε σε ένα αντικείμενο και να περάσετε το άλλο σαν όρισμα :

```
>>> end.is_after(start)
True
```

Το καλό με αυτήν τη σύνταξη είναι ότι διαβάζεται σχεδόν όπως στα Αγγλικά : “end is after start?”

17.5 Η μέθοδος init

Η μέθοδος init (συντομογραφία του “initialization”) είναι μία ειδική μέθοδος η οποία επικαλείται όταν ένα αντικείμενο αρχικοποιείται. Το πλήρες όνομά της είναι `__init__` (δύο κάτω παύλες πριν και μετά της λέξης `init`). Μία μέθοδος init για την κλάση `Time` θα μπορούσε να είναι κάπως έτσι :

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

Οι παράμετροι της `__init__` συνηθίζεται να έχουν τα ίδια ονόματα με τις ιδιότητες. Η δήλωση :

```
        self.hour = hour
```

αποθηκεύει την τιμή της παραμέτρου hour σαν μία ιδιότητα της self.

Οι παράμετροι είναι προαιρετικές, άρα αν καλέσετε την Time χωρίς ορίσματα τότε θα πάρετε τις προεπιλεγμένες τιμές :

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

Αν δώσετε ένα όρισμα, τότε αντικαθιστά την hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

Αν δώσετε δύο ορίσματα, τότε αυτά αντικαθιστούν την hour και την minute:

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

Και αν δώσετε τρία ορίσματα, τότε αυτά αντικαθιστώνται και τις τρεις προεπιλεγμένες τιμές.
Άσκηση 17.2. *Γράψτε μία μέθοδο init για την κλάση* Point *η οποία θα παίρνει την* x *και την* y *σαν προαιρετικές παραμέτρους και θα τις εκχωρεί στις αντίστοιχες ιδιότητες.*

17.6 Η μέθοδος __str__

Η __str__ είναι μία ειδική μέθοδος, όπως η __init__, η οποία υποτίθεται ότι επιστρέφει μία συμβολοσειρά αναπαράστασης ενός αντικειμένου.

Για παράδειγμα, αυτή είναι μία μέθοδος str για αντικείμενα Time:

```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

Όταν τυπώνετε ένα αντικείμενο, η Python επικαλείται την μέθοδο str:

```
>>> time = Time(9, 45)
>>> print time
09:45:00
```

Όταν γράφω μία νέα κλάση, σχεδόν πάντα ξεκινάω γράφοντας την __init__, η οποία κάνει ευκολότερη την αρχικοποίηση των αντικειμένων, και την __str__, η οποία είναι χρήσιμη στην αποσφαλμάτωση.
Άσκηση 17.3. *Γράψτε μία μέθοδο* str *για την κλάση Point. Δημιουργήστε ένα αντικείμενο Point και τυπώστε το.*

17.7 Υπερφόρτωση τελεστών

Με τον ορισμό άλλων ειδικών συναρτήσεων, μπορείτε να καθορίσετε τη συμπεριφορά των τελεστών στους τύπους που ορίζονται από το χρήστη. Για παράδειγμα, αν ορίσετε μία μέθοδο με όνομα __add__ για την κλάση Time, μπορείτε να χρησιμοποιήσετε τον τελεστή + σε αντικείμενα Time.

Ο ορισμός μπορεί να είναι κάπως έτσι :

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

Και μπορείτε να την χρησιμοποιήσετε κάπως έτσι :

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
```

Όταν εφαρμόζετε τον τελεστή + σε αντικείμενα Time, η Python επικαλείται την `__add__`. Όταν τυπώνετε το αποτέλεσμα, η Python επικαλείται την `__str__`. Πολλά πράγματα επομένως συμβαίνουν στο παρασκήνιο !

Η αλλαγή της συμπεριφοράς ενός τελεστή ούτως ώστε να δουλεύει με τύπους ορισμένους από το χρήστη ονομάζεται **υπερφόρτωση τελεστών**. Για κάθε τελεστή στην Python υπάρχει μία αντίστοιχη ειδική μέθοδος, όπως η `__add__`. Για περισσότερες λεπτομέρειες δείτε εδώ : `http://docs.python.org/2/reference/datamodel.html#specialnames`.

Άσκηση 17.4. *Γράψτε μία μέθοδο* `add` *για την κλάση Point.*

17.8 Αποστολή βάση τύπου

Στην προηγούμενη ενότητα προσθέσαμε δύο αντικείμενα Time, αλλά μπορεί επίσης να θέλατε να προσθέσετε έναν ακέραιο σε ένα αντικείμενο ώρας. Ακολουθεί μία έκδοση της `__add__` η οποία ελέγχει τον τύπο της `other` και επικαλείται είτε την `add_time` είτε την `increment`:

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

Η ενσωματωμένη συνάρτηση `isinstance` παίρνει μία τιμή και ένα αντικείμενο κλάσης και επιστρέφει `True` αν η τιμή είναι ένα στιγμιότυπο της κλάσης.

Αν η `other` είναι ένα αντικείμενο Time, η `__add__` επικαλείται την `add_time`. Αλλιώς υποθέτει ότι η παράμετρος είναι ένας αριθμός και επικαλείται την `increment`. Αυτή η λειτουργία ονομάζεται ” αποστολή βάση τύπου ” επειδή αποστέλλει τον υπολογισμό σε διαφορετικές μεθόδους ανάλογα με τον τύπο των ορισμάτων.

Αυτά είναι κάποια παραδείγματα που χρησιμοποιούν τον τελεστή + με διαφορετικούς τύπους :

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
>>> print start + 1337
10:07:17
```

Δυστυχώς, αυτή η υλοποίηση της πρόσθεσης δεν είναι αντιμεταθετική. Αν ο ακέραιος είναι ο πρώτος τελεστέος τότε θα πάρετε :

```
>>> print 1337 + start
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

Το πρόβλημα είναι ότι αντί να ζητάμε από το αντικείμενο Time να προσθέσει έναν ακέραιο, η Python ζητάει από έναν ακέραιο να προσθέσει ένα αντικείμενο Time αλλά δεν ξέρει πως να το κάνει αυτό. Υπάρχει όμως μία έξυπνη λύση για αυτό το πρόβλημα : η ειδική μέθοδος `__radd__`, το όνομα της οποίας αντιπροσωπεύει την " πρόσθεση δεξιού μέρους " (right-side add). Αυτή η μέθοδος επικαλείται όταν ένα αντικείμενο Time εμφανίζεται στο δεξιό μέρος του τελεστή +. Αυτός είναι ο ορισμός :

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

Και εδώ φαίνεται πως χρησιμοποιείται :

```
>>> print 1337 + start
10:07:17
```

Άσκηση 17.5. *Γράψτε μία μέθοδο* `add` *για σημεία η οποία θα δουλεύει είτε με ένα αντικείμενο Point είτε με μία πλειάδα :*

- *Αν ο δεύτερος τελεστέος είναι ένα αντικείμενο Point, τότε η μέθοδος θα πρέπει να επιστρέφει ένα νέο Point του οποίου η συντεταγμένη x θα είναι το άθροισμα των συντεταγμένων x των τελεστέων και το ίδιο θα γίνεται και με τις συντεταγμένες y.*
- *Αν ο δεύτερος τελεστέος είναι μία πλειάδα, η μέθοδος θα πρέπει να προσθέτει το πρώτο στοιχείο της πλειάδας στη συντεταγμένη x και το δεύτερο στοιχείο στη συντεταγμένη y, και θα επιστρέφει ένα νέο αντικείμενο Point με το αποτέλεσμα.*

17.9 Πολυμορφισμός

Η αποστολή βάση τύπου είναι χρήσιμη όταν είναι απαραίτητη, αλλά (ευτυχώς) δεν είναι πάντα αναγκαία. Συχνά μπορείτε να την αποφύγετε γράφοντας συναρτήσεις οι οποίες δουλεύουν σωστά με ορίσματα διαφορετικού τύπου.

Στην πραγματικότητα, πολλές από τις συναρτήσεις που γράψαμε για τις συμβολοσειρές δουλεύουν με οποιοδήποτε είδος ακολουθίας. Για παράδειγμα, στην Ενότητα 11.1 χρησιμοποιήσαμε την `histogram` για να μετρήσουμε πόσες φορές εμφανίζεται κάθε γράμμα μέσα σε μία λέξη.

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
```

```
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

Αυτή η συνάρτηση δουλεύει επίσης και με λίστες, πλειάδες, ακόμα και με λεξικά, από τη στιγμή που τα στοιχεία της s είναι κατακερματίσιμα, έτσι ώστε να μπορούν να χρησιμοποιηθούν σαν κλειδιά στο d.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Οι συναρτήσεις που μπορούν να δουλέψουν με διάφορους τύπους ονομάζονται ” πολυμορφικές ”. Για παράδειγμα, η ενσωματωμένη συνάρτηση sum, η οποία προσθέτει τα στοιχεία μίας ακολουθίας, δουλεύει όταν τα στοιχεία της ακολουθίας υποστηρίζουν πρόσθεση.

Από τη στιγμή που τα αντικείμενα Time παρέχουν μία μέθοδο add, δουλεύουν με τη sum:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print total
23:01:00
```

Σε γενικές γραμμές, αν όλες οι λειτουργίες μέσα σε μία συνάρτηση δουλεύουν με ένα δεδομένο τύπο, τότε η συνάρτηση δουλεύει με αυτόν τον τύπο.

Το καλύτερο είδος πολυμορφισμού είναι το ακούσιο είδος, όπου ανακαλύπτετε ότι μία συνάρτηση που έχετε ήδη γράψει μπορεί να εφαρμοστεί σε έναν τύπο για τον οποίο δεν έχετε προβλέψει.

17.10 Αποσφαλμάτωση

Μπορείτε να προσθέσετε ιδιότητες στα αντικείμενα σε οποιοδήποτε σημείο εκτέλεσης ενός προγράμματος, αλλά αν είστε σχολαστικοί όσον αναφορά τη θεωρία των τύπων, τότε είναι αμφιλεγόμενη πρακτική να έχετε αντικείμενα του ίδιου τύπου με διαφορετικά σύνολα ιδιοτήτων. Είναι προτιμότερο να αρχικοποιείτε όλες τις ιδιότητες των αντικειμένων στην μέθοδο init.

Αν δεν είστε σίγουροι για το αν ένα αντικείμενο έχει μία συγκεκριμένη ιδιότητα, μπορείτε να χρησιμοποιήσετε την ενσωματωμένη συνάρτηση hasattr (δείτε την Ενότητα 15.7).

Ένας άλλος τρόπος για να αποκτήσετε πρόσβαση στις ιδιότητες ενός αντικειμένου είναι μέσω τις ειδικής ιδιότητας __dict__, η οποία είναι ένα λεξικό που αντιστοιχεί ονόματα ιδιοτήτων (σαν συμβολοσειρές) σε τιμές :

```
>>> p = Point(3, 4)
>>> print p.__dict__
{'y': 4, 'x': 3}
```

Ίσως σας φανεί χρήσιμο, για λόγους αποσφαλμάτωσης, να έχετε αυτήν τη συνάρτηση εύχερη :

```
def print_attributes(obj):
    for attr in obj.__dict__:
        print attr, getattr(obj, attr)
```

Η print_attributes διασχίζει τα ζευγάρια κλειδιού-τιμής στο λεξικό του αντικειμένου και τυπώνει όλα τα ονόματα των ιδιοτήτων με τις αντίστοιχες τιμές.

Η ενσωματωμένη συνάρτηση getattr παίρνει ένα αντικείμενο και ένα όνομα ιδιότητας (ως συμβολοσειρά) και επιστρέφει την τιμή της ιδιότητας.

17.11 Διεπαφή και υλοποίηση

Ένας από τους στόχους του αντικειμενοστραφή σχεδιασμού είναι να βοηθήσει στην συντήρηση του λογισμικού. Αυτό σημαίνει ότι μπορείτε να τροποποιήσετε το πρόγραμμα ενώ τρέχει, ούτως ώστε να μπορεί να ανταποκριθεί σε νέες απαιτήσεις.

Μία αρχή σχεδιασμού η οποία βοηθάει στην επίτευξη αυτού του στόχου είναι να κρατάτε τις ξεχωριστά τις διεπαφές από τις υλοποιήσεις. Για τα αντικείμενα, αυτό σημαίνει ότι οι μέθοδοι που παρέχει μία κλάση δεν θα πρέπει να εξαρτώνται από το πως αναπαριστώνται οι ιδιότητες.

Για παράδειγμα, σε αυτό το κεφάλαιο αναπτύξαμε μία κλάση η οποία αναπαριστά την ώρα της ημέρας. Οι μέθοδοι που παρέχει αυτή η κλάση περιλαμβάνουν την time_to_int, την is_after, και την add_time.

Μπορούμε να υλοποιήσουμε αυτές τις μεθόδους με διάφορους τρόπους. Οι λεπτομέρειες τις υλοποίησης εξαρτώνται από το πως αναπαριστούμε την ώρα. Σε αυτό το κεφάλαιο, οι ιδιότητες ενός αντικειμένου Time είναι η hour, η minute και η second.

Σαν μία εναλλακτική λύση, μπορούμε να αντικαταστήσουμε αυτές τις ιδιότητες με έναν μοναδικό ακέραιο ο οποίος θα αναπαριστά τον αριθμό των δευτερολέπτων από τα μεσάνυχτα. Αυτή η υλοποίηση κάνει μερικές μεθόδους, όπως η is_after, ευκολότερο να γραφτούν αλλά κάποιες άλλες δυσκολότερο.

Αφού αναπτύξετε μία νέα κλάση, μπορεί να ανακαλύψετε μία καλύτερη υλοποίηση. Αν τα άλλα μέρη του προγράμματος χρησιμοποιούν την κλάση σας, μπορεί να είναι χρονοβόρο και επιρρεπές σε λάθη να αλλάξετε τη διεπαφή.

Αλλά αν έχετε σχεδιάσει τη διεπαφή προσεκτικά τότε μπορείτε να αλλάξετε την υλοποίηση χωρίς να αλλάξετε τη διεπαφή, το οποίο σημαίνει ότι τα άλλα μέρη του προγράμματος δεν χρειάζεται να αλλάξουν.

Η διατήρηση των διεπαφών χωριστά από την υλοποίηση σημαίνει ότι πρέπει να αποκρύψετε τις ιδιότητες. Ο κώδικας σε άλλα μέρη του προγράμματος (εκτός του ορισμού της κλάσης) θα πρέπει να χρησιμοποιεί τις μεθόδους για να διαβάζει και να τροποποιεί την κατάσταση ενός αντικειμένου. Δεν πρέπει να έχουν απευθείας πρόσβαση στις ιδιότητες. Αυτή η αρχή ονομάζεται ” απόκρυψη πληροφοριών ” (βλ. http://en.wikipedia.org/wiki/Information_hiding).

Άσκηση 17.6. *Κατεβάστε τον κώδικα αυτού του κεφαλαίου από εδώ (http://thinkpython.com/code/Time2.py). Αλλάξτε τις ιδιότητες της* Time *ώστε να είναι μόνο ένας ακέραιος ο οποίος θα αναπαριστά τα δευτερόλεπτα από τα μεσάνυχτα. Στη συνέχεια τροποποιήστε τις μεθόδους (και τη συνάρτηση* int_to_time *) για να δουλεύουν με την νέα υλοποίηση. Δεν θα πρέπει να χρειαστεί να τροποποιήσετε τον υπόλοιπο κώδικα στην* main*. Όταν τελειώσετε, η έξοδος θα πρέπει να είναι ίδια με πριν. Λύση :* http://thinkpython.com/code/Time2_soln.py

17.12 Ορολογία

αντικειμενοστραφής γλώσσα: Μία γλώσσα η οποία παρέχει χαρακτηριστικά, όπως κλάσεις οριζόμενες από το χρήστη και σύνταξη μεθόδων, που διευκολύνουν τον αντικειμενοστραφή προ-

γραμματισμό.

αντικειμενοστραφής προγραμματισμός: Ένας τρόπος προγραμματισμού στον οποίο τα δεδομένα και οι λειτουργίες είναι οργανωμένα σε κλάσεις και μεθόδους.

μέθοδος: Μία συνάρτηση η οποία ορίζεται μέσα σε έναν ορισμό κλάσης και επικαλείται σε στιγμιότυπα αυτής της κλάσης.

υποκείμενο: Το αντικείμενο στο οποίο επικαλείται μία μέθοδος.

υπερφόρτωση τελεστή: Η αλλαγή της συμπεριφοράς ενός τελεστή όπως ο + ούτως ώστε να δουλεύει με έναν τύπο ορισμένο από το χρήστη.

αποστολή βάση τύπου: Ένα πρότυπο προγραμματισμού το οποίο ελέγχει τον τύπο ενός τελεστέου και επικαλείται διαφορετικές συναρτήσεις για διαφορετικούς τύπους.

πολυμορφική: Μία συνάρτηση η οποία μπορεί να δουλέψει με περισσότερους του ενός τύπου.

απόκρυψη πληροφοριών: Η αρχή ότι η διεπαφή που παρέχει ένα αντικείμενο δεν θα πρέπει να εξαρτάται από την υλοποίηση, ειδικότερα από την αναπαράσταση των ιδιοτήτων.

17.13 Ασκήσεις

Άσκηση 17.7. *Αυτή η άσκηση αποτελεί ένα διδακτικό παραμύθι σχετικά με ένα από τα συνηθέστερα, και δύσκολο να βρεθούν, λάθη στην Python. Γράψτε έναν ορισμό για μια κλάση με όνομα* `Kangaroo` *με τις εξής μεθόδους :*

1. *Μία μέθοδο* `__init__` *η οποία θα αρχικοποιεί μία ιδιότητα με όνομα* `pouch_contents` *σε μία κενή λίστα.*
2. *Μία μέθοδο με όνομα* `put_in_pouch` *η οποία θα παίρνει ένα οποιουδήποτε τύπου αντικείμενο και θα το προσθέτει στην* `pouch_contents`.
3. *Μία* `__str__` *η οποία θα επιστρέφει μία συμβολοσειρά αναπαράστασης του αντικειμένου Kangaroo και τα περιεχόμενα της σακούλας.*

Δοκιμάστε τον κώδικά σας δημιουργώντας δύο αντικείμενα `Kangaroo`*, εκχωρώντας τα στις μεταβλητές* `kanga` *και* `roo`*, και προσθέτοντας το* `roo` *στα περιεχόμενα της σακούλας του* `kanga`.

Κατεβάστε το `http://thinkpython.com/code/BadKangaroo.py`*. Περιέχει μία λύση στο προηγούμενο πρόβλημα με ένα μεγάλο και βρωμερό σφάλμα. Βρείτε και διορθώστε το σφάλμα.*

Αν κολλήσετε, μπορείτε να κατεβάσετε το `http://thinkpython.com/code/GoodKangaroo.py`*, το οποίο εξηγεί το πρόβλημα και επιδεικνύει μία λύση.*

Άσκηση 17.8. *Το Visual είναι ένα άρθρωμα της Python το οποίο παρέχει τρισδιάστατα γραφικά. Δεν συμπεριλαμβάνεται πάντα στην εγκατάσταση της Python και έτσι μπορεί να χρειαστεί να την εγκαταστήσετε από την αποθήκη λογισμικού, ή αν δεν είναι εκεί από τον σύνδεσμο* `vpython.org`.

Το ακόλουθο παράδειγμα φτιάχνει έναν τρισδιάστατο χώρο ο οποίος έχει 256 μονάδες πλάτος, μήκος και ύψος, θέτει το κέντρο στο σημείο $(128, 128, 128)$ *και σχεδιάζει μία μπλε σφαίρα.*

```
from visual import *

scene.range = (256, 256, 256)
scene.center = (128, 128, 128)

color = (0.1, 0.1, 0.9)          # mostly blue
sphere(pos=scene.center, radius=128, color=color)
```

Η `color` *είναι μία RGB πλειάδα, δηλαδή τα στοιχεία της είναι επίπεδα του κόκκινου (Red), του πράσινου (Green) και του μπλε (Blue) μεταξύ του 0.0 και του 1.0 (βλ.* `http://en.wikipedia.org/wiki/RGB_color_model` *).*

Αν τρέξετε αυτόν τον κώδικα θα πρέπει να δείτε ένα παράθυρο με μαύρο φόντο και μία μπλε σφαίρα. Αν σύρετε το μεσαίο κουμπί πάνω κάτω μπορείτε να το μεγεθύνετε ή να το σμικρύνετε. Μπορείτε επίσης να περιστρέψετε την σκηνή σύροντας το δεξί κουμπί, αλλά με μόνο μία σφαίρα στον κόσμο, είναι δύσκολο να δούμε τη διαφορά.

Ο ακόλουθος βρόγχος δημιουργεί ένα κύβο από σφαίρες :

```
t = range(0, 256, 51)
for x in t:
    for y in t:
        for z in t:
            pos = x, y, z
            sphere(pos=pos, radius=10, color=color)
```

1. *Βάλτε αυτόν τον κώδικα σε ένα σενάριο και σιγουρευτείτε ότι δουλεύει.*
2. *Τροποποιήστε το πρόγραμμα έτσι ώστε κάθε σφαίρα στον κύβο να έχει το χρώμα που αντιστοιχεί στην θέση της στον RGB χώρο. Προσέξτε ότι οι συντεταγμένες είναι στο εύρος 0–255 αλλά οι RGB πλειάδες είναι στο εύρος 0.0–1.0.*
3. *Κατεβάστε το* `http://thinkpython.com/code/color_list.py` *και χρησιμοποιήστε την συνάρτηση* `read_colors` *για να δημιουργήσετε μία λίστα από τα διαθέσιμα χρώματα στο σύστημά σας, τα ονόματά τους και τις τιμές RGB. Σχεδιάστε, για κάθε χρώμα, μία σφαίρα στη θέση που αντιστοιχεί στις RGB τιμές της.*

Μπορείτε να δείτε τη λύση μου στον σύνδεσμο : `http://thinkpython.com/code/color_space.py`.

Κεφάλαιο 18

Κληρονομικότητα

Σε αυτό το κεφάλαιο θα σας παρουσιάσω κλάσεις οι οποίες αναπαριστούν τραπουλόχαρτα, τράπουλες και χέρια στο πόκερ. Αν δεν ξέρετε να παίζετε πόκερ, μπορείτε να διαβάσετε σχετικά με αυτό στο σύνδεσμο `http://en.wikipedia.org/wiki/Poker`, αλλά δεν είναι απαραίτητο. Θα σας πω ότι χρειάζεστε να ξέρετε για τις ασκήσεις. Τα παραδείγματα κώδικα αυτού του κεφαλαίου μπορείτε να τα βρείτε στο σύνδεσμο : `http://thinkpython.com/code/Card.py`.

Αν δεν είστε εξοικειωμένοι με αγγλοαμερικανικά χαρτιά, μπορείτε να διαβάσετε σχετικά εδώ : `http://en.wikipedia.org/wiki/Playing_cards`.

18.1 Αντικείμενα τραπουλόχαρτων

Υπάρχουν πενήντα δύο χαρτιά σε μία τράπουλα, το καθένα από τα οποία ανήκει σε ένα από τα τέσσερα χρώματα και σε μία από τις δεκατρείς τάξεις. Τα χρώματα είναι τα Μπαστούνια, οι Καρδιές, τα Καρό και τα Σπαθιά. Οι τάξεις είναι ο Άσος, το 2, 3, 4, 5, 6, 7, 8, 9, 10, ο Βαλές, η Ντάμα και ο Ρήγας. Ανάλογα με το παιχνίδι που παίζετε, ο Άσος μπορεί να είναι μεγαλύτερος του Ρήγα και μικρότερος του 2.

Αν θέλετε να ορίσετε ένα νέο αντικείμενο για να αναπαραστήσετε ένα τραπουλόχαρτο, είναι προφανές ότι οι ιδιότητες θα είναι η τάξη (`rank`) και το χρώμα (`suit`). Δεν είναι όμως το ίδιο προφανές τι τύπου θα πρέπει να είναι οι ιδιότητες. Ένα ενδεχόμενο είναι να χρησιμοποιήσουμε συμβολοσειρές που θα περιέχουν λέξεις όπως η `'Spade'` για τα χρώματα και η `'Queen'` για τις τάξεις. Έ να πρόβλημα με αυτήν την υλοποίηση είναι ότι δεν θα είναι εύκολο να συγκρίνουμε χαρτιά για να δούμε πιο έχει υψηλότερη τάξη ή χρώμα.

Μία εναλλακτική λύση είναι να χρησιμοποιήσουμε ακέραιους για να κωδικοποιήσουμε τις τάξεις και τα χρώματα. Σε αυτό το πλαίσιο, ” κωδικοποίηση ” σημαίνει ότι θα ορίσουμε μία αντιστοίχιση μεταξύ αριθμών και χρωμάτων ή μεταξύ αριθμών και τάξεων. Αυτού του είδους η κωδικοποίηση δεν πρόκειται να είναι κρυφή (αλλιώς θα ήταν κρυπτογράφηση).

Για παράδειγμα, αυτός ο πίνακας δείχνει τα χρώματα και τους αντίστοιχους κώδικες ακεραίων :

Spades	$\mapsto$	3
Hearts	$\mapsto$	2
Diamonds	$\mapsto$	1
Clubs	$\mapsto$	0

Αυτός ο κώδικας κάνει ευκολότερη την σύγκριση τραπουλόχαρτων, επειδή τα μεγαλύτερα χρώματα αντιστοιχούν σε μεγαλύτερους αριθμούς. Μπορούμε να συγκρίνουμε χρώματα συγκρίνοντας τους κωδικούς τους.

Η αντιστοίχιση για τις τάξεις είναι αρκετά προφανής. Κάθε μία από τις αριθμητικές τάξεις αντιστοιχεί στον αντίστοιχο ακέραιο και για τις φιγούρες :

Jack $\mapsto$ 11
Queen $\mapsto$ 12
King $\mapsto$ 13

Χρησιμοποιώ το σύμβολο $\mapsto$για να ξεκαθαρίσω ότι αυτές οι αντιστοιχίσεις δεν είναι μέρος του προγράμματος Python. Είναι μέρος του σχεδιασμού του προγράμματος, αλλά δεν εμφανίζονται ρητά μέσα στον κώδικα.

Ο ορισμός της κλάσης για την `Card` έχει την εξής μορφή :

```
class Card(object):
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

Ως συνήθως, η μέθοδος init παίρνει μία προαιρετική παράμετρο για κάθε ιδιότητα. Το προεπιλεγμένο τραπουλόχαρτο είναι το 2 Σπαθί.

Για δημιουργήσετε ένα χαρτί, καλείτε την `Card` με το χρώμα και την τάξη που θέλετε :

```
queen_of_diamonds = Card(1, 12)
```

18.2 Ιδιότητες κλάσεων

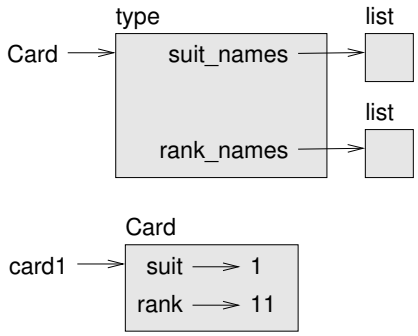
Προκειμένου να τυπώσουμε αντικείμενα Card με τέτοιο τρόπο ώστε οι άνθρωποι να μπορούν εύκολα να τα διαβάσουν εύκολα, χρειαζόμαστε μία αντιστοίχιση από τους ακέραιους κωδικούς στις αντίστοιχες τάξεις και χρώματα. Ένας τρόπος για να το κάνουμε αυτό είναι λίστες συμβολοσειρών. Εκχωρούμε αυτές τις λίστες στις ιδιότητες της κλάσης :

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

Μεταβλητές όπως η `suit_names` και η `rank_names`, οι οποίες ορίζονται μέσα σε μία κλάση αλλά εκτός οποιασδήποτε μεθόδου, ονομάζονται ιδιότητες κλάσης επειδή συνδέονται με το αντικείμενο της κλάσης `Card`.

Αυτός ο όρος τις διακρίνει από μεταβλητές όπως η `suit` και η `rank`, οι οποίες ονομάζονται ιδιότητες στιγμιότυπου επειδή σχετίζονται με ένα συγκεκριμένο στιγμιότυπο.

Και τα δύο είδη των ιδιοτήτων είναι προσβάσιμα μέσω του συμβολισμού με τελεία. Για παράδειγμα, στην `__str__`, η `self` είναι ένα αντικείμενο Card, και η `self.rank` είναι η τάξη του. Παρομοίως,



Σχήμα 18.1: Διάγραμμα αντικειμένων.

το Card είναι ένα αντικείμενο κλάσης και η Card.rank_names είναι μία λίστα από συμβολοσειρές που σχετίζονται με την κλάση.

Κάθε τραπουλόχαρτο έχει την δική του suit και rank, αλλά υπάρχει μόνο ένα αντίγραφο της suit_names και της rank_names.

Βάζοντάς τα όλα μαζί, η έκφραση Card.rank_names[self.rank] σημαίνει " χρησιμοποίησε την ιδιότητα rank του αντικειμένου self σαν δείκτη στη λίστα rank_names της κλάσης Card, και επέλεξε την κατάλληλη συμβολοσειρά ".

Το πρώτο στοιχείο της rank_names είναι None επειδή δεν υπάρχει τραπουλόχαρτο με τάξη μηδέν. Συμπεριλαμβάνοντας την None έχουμε μία αντιστοίχιση με το εξής ωραίο χαρακτηριστικό ότι ο δείκτης 2 αντιστοιχεί στη συμβολοσειρά '2', και ούτω καθεξής. Για να αποφύγουμε αυτήν την τροποποίηση θα μπορούσαμε να είχαμε χρησιμοποιήσει ένα λεξικό αντί για λίστα.

Με τις μεθόδους που έχουμε μέχρι τώρα, μπορούμε να δημιουργήσουμε και να τυπώσουμε τραπουλόχαρτα :

```
>>> card1 = Card(2, 11)
>>> print card1
Jack of Hearts
```

Η Εικόνα 18.1 είναι ένα διάγραμμα του αντικειμένου της κλάσης Card και ένα του στιγμιότυπου Card. Η Card είναι μία κλάση αντικειμένων, άρα έχει τύπο type. Το card1 έχει τύπο Card. (Για εξοικονόμηση χώρου, δεν σχεδίασα τα περιεχόμενα των suit_names και rank_names).

18.3 Συγκρίνοντας τραπουλόχαρτα

Για τους ενσωματωμένους τύπους υπάρχουν σχεσιακοί τελεστές (<, >, ==, κτλ.) οι συγκρίνουν τιμές και προσδιορίζουν ποια είναι μεγαλύτερη, ποια είναι μικρότερη ή αν είναι ίσες. Για τους τύπους που είναι ορισμένοι από το χρήστη, μπορούμε να παρακάμψουμε τη συμπεριφορά των ενσωματωμένων τελεστών παρέχοντας μία μέθοδο με όνομα __cmp__.

Η __cmp__ παίρνει δύο παραμέτρους, την self και την other, και επιστρέφει ένα θετικό αριθμό αν το πρώτο αντικείμενο είναι μεγαλύτερο, ένα αρνητικό αριθμό αν το δεύτερο αντικείμενο είναι μεγαλύτερο και 0 αν είναι ίσα το ένα με το άλλο.

Η σωστή σειρά για τα τραπουλόχαρτα δεν είναι προφανής. Ποιο είναι καλύτερο για παράδειγμα, το 3 σπαθί ή το 2 καρό ; Το ένα έχει υψηλότερη τάξη αλλά το άλλο έχει μεγαλύτερο χρώμα. Προκειμένου

να μπορέσετε να συγκρίνετε τραπουλόχαρτα, πρέπει να αποφασίσετε ποιο από τα δύο, η τάξη ή το χρώμα, είναι σημαντικότερο.

Η απάντηση μπορεί να εξαρτάται από το παιχνίδι που παίζετε, αλλά για να απλοποιήσουμε τα πράγματα, θα κάνουμε την αυθαίρετη επιλογή ότι το χρώμα είναι σημαντικότερο, άρα όλα τα μπαστούνια είναι είναι ανώτερα από όλα τα καρό και ούτω καθεξής.

Με βάση αυτό μπορούμε να γράψουμε την `__cmp__`:

```
# inside class Card:

    def __cmp__(self, other):
        # check the suits
        if self.suit > other.suit: return 1
        if self.suit < other.suit: return -1

        # suits are the same... check ranks
        if self.rank > other.rank: return 1
        if self.rank < other.rank: return -1

        # ranks are the same... it's a tie
        return 0
```

Μπορείτε να την γράψετε πιο συνοπτικά χρησιμοποιώντας σύγκριση πλειάδων :

```
# inside class Card:

    def __cmp__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return cmp(t1, t2)
```

η ενσωματωμένη συνάρτηση `cmp` έχει την ίδια διεπαφή με τη μέθοδο `__cmp__`: παίρνει δύο τιμές και επιστρέφει ένα θετικό αριθμό αν η πρώτη είναι μεγαλύτερη, έναν αρνητικό αν η δεύτερη είναι μεγαλύτερη και 0 αν είναι ίσες.

Στην Python 3 δεν υπάρχει πλέον η `cmp` και η μέθοδος `__cmp__` δεν υποστηρίζεται. Αντί αυτών θα πρέπει να παρέχετε την `__lt__`, η οποία επιστρέφει `True` αν η `self` είναι μικρότερη της `other`. Μπορείτε να υλοποιήσετε την `__lt__` χρησιμοποιώντας πλειάδες και τον τελεστή `<`.

Άσκηση 18.1. *Γράψτε μία μέθοδο `__cmp__` για αντικείμενα Time. Σημείωση : μπορείτε να χρησιμοποιήσετε σύγκριση πλειάδων, αλλά λάβετε υπόψιν σας ότι μπορείτε να χρησιμοποιήσετε και αφαίρεση ακεραίων.*

18.4 Τράπουλες

Τώρα που έχουμε τραπουλόχαρτα, το επόμενο βήμα είναι να ορίσουμε τράπουλες. Αφού μία τράπουλα αποτελείται από τραπουλόχαρτα, είναι φυσικό κάθε τράπουλα να περιέχει μία λίστα από τραπουλόχαρτα σαν μία ιδιότητα.

Το παρακάτω είναι ένας ορισμός κλάσης για τράπουλες. Η μέθοδος init δημιουργεί την ιδιότητα `cards` και παράγει ένα τυπικό σύνολο από πενήντα δύο φίλα :

```
class Deck(object):

    def __init__(self):
```

```
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

Ο ευκολότερος τρόπος για να συμπληρώσουμε μία τράπουλα είναι με έναν εμφωλευμένο βρόχο. Ο εξωτερικός βρόχος απαριθμεί τα χρώματα από το 0 έως το 3. Ο εσωτερικός βρόχος απαριθμεί τις τάξεις από το 1 έως το 13. Κάθε επανάληψη δημιουργεί ένα νέο τραπουλόχαρτο με το τρέχον χρώμα και τάξη, και το προσαρτά στην `self.cards`.

18.5 Τύπωση τράπουλας

Αυτή είναι μία μέθοδος `__str__` για την `Deck`:

```
#inside class Deck:

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

Αυτή η μέθοδος επιδεικνύει έναν αποδοτικό τρόπο για τη συσσώρευση μίας μεγάλης συμβολοσειράς : φτιάχνοντας μία λίστα συμβολοσειρών και στη συνέχεια χρησιμοποιώντας την `join`. Η ενσωματωμένη συνάρτηση `str` επικαλείται τη μέθοδο `__str__` σε κάθε φίλο και επιστρέφει μία αναπαράσταση συμβολοσειράς.

Από τη στιγμή που επικαλεστήκαμε την `join` σε έναν χαρακτήρα νέας γραμμής, τα φύλλα θα είναι χωρισμένα με νέες γραμμές. Αυτό είναι το αποτέλεσμα :

```
>>> deck = Deck()
>>> print deck
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Παρόλο που το αποτέλεσμα εμφανίζεται σε 52 γραμμές, είναι μία μεγάλη συμβολοσειρά η οποία περιέχει νέες γραμμές.

18.6 Προσθήκη, αφαίρεση, ανακάτεμα και ταξινόμηση

Για να μοιράσουμε φύλλα, χρειαζόμαστε μία μέθοδο η οποία θα αφαιρεί ένα φύλλο από την τράπουλα και θα το επιστρέφει. Η μέθοδος λιστών pop παρέχει έναν εύκολο τρόπο για να γίνει αυτό :

```
#inside class Deck:

    def pop_card(self):
        return self.cards.pop()
```

Αφού η pop αφαιρεί το τελευταίο φύλλο της λίστας, σημαίνει ότι μοιράζουμε από το τέλος της τράπουλας. Στην πραγματικότητα το μοίρασμα από το τέλος δεν είναι σωστό, αλλά στην προκειμένη περίπτωση δεν υπάρχει πρόβλημα.

Για να προσθέσουμε ένα φύλλο, μπορούμε να χρησιμοποιήσουμε την μέθοδο λιστών append:

```
#inside class Deck:

    def add_card(self, card):
        self.cards.append(card)
```

Μία τέτοια μέθοδος η οποία χρησιμοποιεί μία άλλη συνάρτηση χωρίς να κάνει πολύ δουλειά από μόνη της ονομάζεται καπλαμάς (**veneer**). Η έκφραση προέρχεται από την επεξεργασία του ξύλου, όπου συνηθίζεται να κολλάμε ένα λεπτό στρώμα καλής ποιότητας ξύλου στην επιφάνεια ενός φθηνότερου κομματιού ξύλου.

Στην συγκεκριμένη περίπτωση ορίζουμε μία " λεπτή " μέθοδο η οποία εκφράζει μία λειτουργία της λίστας με όρους που είναι κατάλληλοι για τράπουλες.

Σαν ένα ακόμα παράδειγμα, μπορούμε να γράψουμε μία μέθοδο με όνομα shuffle χρησιμοποιώντας τη συνάρτηση shuffle του αρθρώματος random:

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

Μην ξεχάσετε να εισάγετε το άρθρωμα random.

Άσκηση 18.2. *Γράψτε μία μέθοδο για την Deck με όνομα* sort *η οποία θα χρησιμοποιεί τη μέθοδο λιστών* sort *για να ταξινομήσει τα φύλλα. Η* sort *θα χρησιμοποιεί τη μέθοδο* __cmp__ *που ορίσαμε για να καθορίσουμε τη σειρά ταξινόμησης.*

18.7 Κληρονομικότητα

Το χαρακτηριστικό της γλώσσας που την συνδέει περισσότερο με τον αντικειμενοστραφή προγραμματισμό η κληρονομικότητα. Κληρονομικότητα είναι η δυνατότητα να ορίσουμε μία νέα κλάση η οποία θα είναι μία τροποποιημένη έκδοση μίας υπάρχουσας κλάσης.

Ονομάζεται " κληρονομικότητα " επειδή η νέα κλάση κληρονομεί τις μεθόδους της υπάρχουσας κλάσης. Επεκτείνοντας αυτήν τη μεταφορά, η υπάρχουσα κλάση ονομάζεται γονέας και η νέα κλάση ονομάζεται παιδί.

Σαν παράδειγμα, ας πούμε ότι θέλουμε μία κλάση για να αναπαραστήσουμε ένα " χέρι ", δηλαδή το σύνολο των καρτών που έχει στην κατοχή του ένας παίκτης. Ένα χέρι είναι παρόμοιο με μία τράπουλα : και τα δύο είναι φτιαγμένα από σύνολα φύλλων, και τα δύο χρειάζονται λειτουργίες όπως προσθήκης και αφαίρεσης φύλλων.

Ένα χέρι όμως είναι και διαφορετικό από μία τράπουλα : υπάρχουν λειτουργίες που θέλουμε να έχουν τα χέρια αλλά δεν έχουν νόημα για μια τράπουλα. Για παράδειγμα, στο πόκερ μπορεί να συγκρίνουμε δύο χέρια για να δούμε ποιο κερδίζει. Στη γέφυρα, μπορεί να υπολογίσουμε ένα σκορ για ένα χέρι προκειμένου να κάνουμε ένα ποντάρισμα.

Αυτή η σχέση μεταξύ κλάσεων (όμοιες αλλά διαφορετικές) προσφέρεται για κληρονομικότητα.

Ο ορισμός μίας κλάσης παιδί είναι όπως οι ορισμοί των άλλων κλάσεων, με τη διαφορά ότι το όνομα της κλάσης γονέας εμφανίζεται μέσα σε παρενθέσεις :

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

Αυτός ο ορισμός υποδεικνύει ότι η Hand κληρονομεί την Deck. Αυτό σημαίνει ότι μπορούμε να χρησιμοποιήσουμε μεθόδους όπως η pop_card και η add_card τόσο για χέρια όσο και για τράπουλες.

Η Hand κληρονομεί επίσης και την __init__ από την Deck, αλλά δεν κάνει αυτό που πραγματικά θα θέλαμε. Αντί να γεμίζει το χέρι με 52 νέα φύλλα, θα έπρεπε να αρχικοποιεί την cards με μία κενή λίστα.

Αν παράσχουμε μία μέθοδο init στην κλάση Hand, τότε θα υπερισχύσει αυτής στην κλάση Deck:

```
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

Άρα όταν δημιουργείτε ένα χέρι, η Python επικαλείται αυτήν την μέθοδο init:

```
>>> hand = Hand('new hand')
>>> print hand.cards
[]
>>> print hand.label
new hand
```

Αλλά οι άλλες μέθοδοι κληρονομούνται από την Deck, και έτσι μπορούμε να χρησιμοποιήσουμε την pop_card και την add_card για να μοιράσουμε ένα φύλλο :

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print hand
King of Spades
```

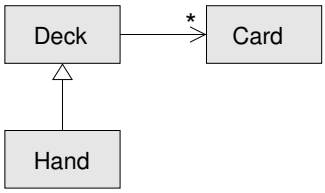
Το επόμενο λογικό βήμα είναι να ενθυλακώσουμε αυτόν τον κώδικα μέσα σε μία μέθοδο με όνομα move_cards:

```
#inside class Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

Η move_cards παίρνει δύο ορίσματα, ένα αντικείμενο Hand και τον αριθμό φύλλων που θα μοιραστούν. Τροποποιεί την self και την hand, και επιστρέφει None.

Σε μερικά παιχνίδια, τα φύλλα μεταφέρονται από ένα χέρι πίσω στην τράπουλα. Μπορείτε να χρησιμοποιήσετε την move_cards για οποιαδήποτε από αυτές τις λειτουργίες : η self μπορεί να είναι είτε μία τράπουλα είτε ένα χέρι και η hand, ανεξαρτήτως ονόματος, μπορεί να είναι ακόμα και μια τράπουλα.

Άσκηση 18.3. *Γράψτε μία μέθοδο για την Deck με όνομα* deal_hands *η οποία θα παίρνει δύο παραμέτρους, τον αριθμό των χεριών και τον αριθμό των φύλλων ανά χέρι. Θα δημιουργεί αντικείμενα*



Σχήμα 18.2: Διάγραμμα κλάσεων.

Hand, θα μοιράζει τον κατάλληλο αριθμό φύλλων σε κάθε χέρι και θα επιστρέφει μία λίστα από αντικείμενα Hand.

Η κληρονομικότητα είναι ένα χρήσιμο χαρακτηριστικό. Μερικά προγράμματα τα οποία θα ήταν επαναλαμβανόμενα χωρίς κληρονομικότητα μπορούν να γραφτούν πιο κομψά με αυτή. Η κληρονομικότητα μπορεί να διευκολύνει την επαναχρησιμοποίηση κώδικα, αφού μπορείτε να προσαρμόσετε την συμπεριφορά των γονικών κλάσεων χωρίς να χρειαστεί να τις τροποποιήσετε. Σε μερικές περιπτώσεις, η δομή της κληρονομικότητας αντανακλά τη φυσική δομή του προβλήματος, το οποίο κάνει ευκολότερη την κατανόηση του προβλήματος.

Από την άλλη μεριά, η κληρονομικότητα μπορεί να κάνει δυσκολότερη την ανάγνωση των προγραμμάτων. Μερικές φορές, όταν επικαλείται μία μέθοδος δεν είναι ξεκάθαρο που θα βρούμε τον ορισμό της. Ο σχετικός κώδικας μπορεί να είναι διασκορπισμένος σε διάφορα αρθρώματα. Επίσης, πολλά πράγματα που μπορούν να γίνουν χρησιμοποιώντας κληρονομικότητα μπορούν να γίνουν επίσης ή και καλύτερα χωρίς αυτήν.

18.8 Διαγράμματα κλάσεων

Μέχρι στιγμής έχουμε δει διαγράμματα στοίβας, τα οποία δείχνουν την κατάσταση ενός προγράμματος, και διαγράμματα αντικειμένων, τα οποία δείχνουν τις ιδιότητες ενός αντικειμένου και τις τιμές τους. Αυτά τα διαγράμματα αναπαριστούν ένα στιγμιότυπο της εκτέλεσης ενός προγράμματος, επομένως αλλάζουν όσο τρέχει το πρόγραμμα.

Είναι επίσης πολύ λεπτομερή και σε ορισμένες περιπτώσεις πάρα πολύ λεπτομερή. Ένα διάγραμμα κλάσης είναι μία πιο αφηρημένη αναπαράσταση της δομής ενός προγράμματος. Αντί να δείχνει τα επιμέρους αντικείμενα, δείχνει τις κλάσεις και τις σχέσεις μεταξύ τους.

Υπάρχουν διάφορα είδη σχέσεων μεταξύ κλάσεων :

- Τα αντικείμενα μίας κλάσης μπορεί να περιέχουν αναφορές σε αντικείμενα άλλων κλάσεων. Για παράδειγμα, κάθε Rectangle περιέχει μία αναφορά σε ένα Point, και κάθε Deck περιέχει αναφορές σε πολλά Cards. Αυτού του είδους η σχέση ονομάζεται ” ΕΧΕΙ-ΕΝΑ ”, όπως στη φράση ” ένα Rectangle έχει ένα Point”.
- Μία κλάση μπορεί να κληρονομεί από μία άλλη. Αυτή η σχέση ονομάζεται ” ΕΙΝΑΙ-ΕΝΑ ”, όπως στη φράση ” ένα Hand είναι ένα είδος μίας Deck”.
- Μία κλάση μπορεί να εξαρτάται από μία άλλη υπό την έννοια ότι οι αλλαγές σε μία κλάση θα απαιτούσαν αλλαγές και στην άλλη.

Ένα διάγραμμα κλάσεων είναι μία γραφική αναπαράσταση αυτών των σχέσεων. Για παράδειγμα, η Εικόνα 18.2 δείχνει τις σχέσεις μεταξύ των `Card`, `Deck` και `Hand`.

Το βέλος με το κοίλο τριγωνάκι αναπαριστά μία σχέση ΕΙΝΑΙ-ΕΝΑ. Σε αυτήν την περίπτωση υποδεικνύει ότι ένα Hand κληρονομεί από μία Deck.

Το κλασικό βέλος αναπαριστά μία σχέση EXEI-ENA. Σε αυτήν την περίπτωση μία Deck έχει αναφορές σε αντικείμενα Card.

Το αστεράκι (*) δίπλα στο βέλος είναι μία πολλαπλότητα, η οποία δείχνει πόσα φύλλα έχει μία τράπουλα. Μία πολλαπλότητα μπορεί να είναι ένας απλός αριθμός, όπως το 52, ένα εύρος, όπως το 5..7, ή ένα αστεράκι το οποίο υποδεικνύει ότι μία τράπουλα μπορεί να έχει έναν οποιοδήποτε αριθμό φύλλων.

Ένα πιο λεπτομερές διάγραμμα μπορεί να δείχνει ότι μία τράπουλα περιέχει μία λίστα από φύλλα, αλλά οι ενσωματωμένοι τύποι όπως οι λίστες και τα λεξικά δεν περιλαμβάνονται σε διαγράμματα κλάσεων.

Άσκηση 18.4. *Διαβάστε τα* `TurtleWorld.py`, `World.py` *και* `Gui.py` *και σχεδιάστε ένα διάγραμμα κλάσεων το οποίο θα δείχνει τις σχέσεις μεταξύ των κλάσεων που ορίζονται εκεί.*

18.9 Αποσφαλμάτωση

Η κληρονομικότητα μπορεί να κάνει την αποσφαλμάτωση μία πρόκληση επειδή όταν επικαλείστε μία μέθοδο σε ένα αντικείμενο, μπορεί να μην ξέρετε ποια μέθοδος επικαλείται.

Υποθέστε ότι γράφετε μία συνάρτηση η οποία θα δουλεύει με αντικείμενα Hand. Τότε, σίγουρα, Θα θέλατε να δουλεύει με όλα τα είδη χεριών, όπως τα PokerHands, BridgeHands και ούτω καθεξής. Αν επικαλεστείτε μία μέθοδο όπως τη `shuffle`, μπορεί να πάρετε αυτήν που ορίστηκε στην `Deck`, αλλά αν κάποια από τις υποκλάσεις παρακάμψει αυτήν τη μέθοδο τότε θα πάρετε εκείνη την έκδοση.

Κάθε φορά που δεν είστε σίγουροι για την ροή εκτέλεσης του προγράμματός σας, η απλούστερη λύση είναι να προσθέσετε δηλώσεις print στην αρχή των σχετικών μεθόδων. Αν η `Deck.shuffle` εμφανίσει ένα μήνυμα το οποίο λέει κάτι τέτοιο `Running Deck.shuffle`, τότε ιχνηλατεί τη ροή εκτέλεσης καθώς τρέχει το πρόγραμμα.

Μία εναλλακτική λύση είναι να χρησιμοποιήσετε αυτήν την συνάρτηση, η οποία παίρνει ένα αντικείμενο και ένα όνομα μεθόδου (σαν συμβολοσειρά) και επιστρέφει την κλάση η οποία παρέχει τον ορισμό της μεθόδου :

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

Αυτό είναι ένα παράδειγμα :

```
>>> hand = Hand()
>>> print find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

Άρα η μέθοδος `shuffle` για αυτό το χέρι είναι αυτή στην `Deck`.

Η `find_defining_class` χρησιμοποιεί τη μέθοδο `mro` για να για να πάρει τη λίστα των αντικειμένων των κλάσεων που θα αναζητηθούν για για μεθόδους. “MRO” είναι τα αρχικά από “method resolution order.”

Αυτή είναι μία πρόταση σχεδιασμού ενός προγράμματος : όποτε παρακάμπτετε μία μέθοδο, η διεπαφή την νέας μεθόδου θα πρέπει να είναι ίδια με της παλιάς. Θα πρέπει να παίρνει τις ίδιες παραμέτρους, να επιστρέφει τον ίδιο τύπο, και να υπακούει στις ίδιες προϋποθέσεις και μετασυνθήκες. Αν ακολουθήσετε αυτόν τον κανόνα, θα ανακαλύψετε ότι κάθε συνάρτηση η οποία σχεδιάστηκε για

να δουλεύει με ένα στιγμιότυπο μία υπερκλάσης, όπως η Deck, δουλεύει επίσης και με στιγμιότυπα των υποκλάσεων όπως της Hand ή της PokerHand.

Αν παραβιάσετε αυτόν τον κανόνα, ο κώδικάς σας θα καταρρεύσει σαν ένα σπίτι από τραπουλόχαρτα.

18.10 Ενθυλάκωση δεδομένων

Το Κεφάλαιο 16 επιδεικνύει ένα πλάνο ανάπτυξης που θα μπορούσαμε να αποκαλέσουμε " αντικειμενοστραφή σχεδίαση ". Προσδιορίσαμε τα αντικείμενα που χρειαζόμασταν (`Time`, `Point` και `Rectangle`) και ορίσαμε κλάσεις για να τα αναπαραστήσουμε. Για κάθε περίπτωση υπάρχει μία προφανής αντιστοίχιση μεταξύ ενός αντικειμένου με κάποια οντότητα του πραγματικού κόσμου (ή τουλάχιστον ενός μαθηματικού κόσμου).

Αλλά μερικές φορές δεν είναι προφανές τι αντικείμενα χρειάζεστε και πως θα πρέπει να αλληλεπιδρούν. Σε αυτήν την περίπτωση χρειάζεστε ένα διαφορετικό πλάνο ανάπτυξης. Με τον ίδιο τρόπο που ανακαλύψαμε τις διεπαφές των συναρτήσεων ενθυλακώνοντας και γενικεύοντας, μπορούμε να ανακαλύψουμε και τις διεπαφές των κλάσεων ενθυλακώνοντας δεδομένα.

Η ανάλυση Μαρκόφ, από την Ενότητα 13.8, παρέχει ένα καλό παράδειγμα. Αν κατεβάσετε τον κώδικά μου από τον σύνδεσμο `http://thinkpython.com/code/markov.py`, θα δείτε ότι χρησιμοποιεί δύο καθολικές μεταβλητές, την `suffix_map` και την `prefix`, οι οποίες διαβάζονται και γράφονται από αρκετές συναρτήσεις.

```
suffix_map = {}
prefix = ()
```

Επειδή αυτές οι μεταβλητές είναι καθολικές μπορούμε να τρέξουμε μία ανάλυση τη φορά. Αν διαβάσουμε δύο κείμενα, τα προθέματα και τα επιθέματα τους θα προστεθούν στις ίδιες δομές δεδομένων (το οποίο δημιουργεί κάποιο ενδιαφέρον κείμενο).

Για να τρέξουμε πολλαπλές αναλύσεις, και να τις κρατήσουμε χωριστά, μπορούμε να ενθυλακώσουμε την κατάσταση κάθε ανάλυσης σε ένα αντικείμενο. Κάπως έτσι δηλαδή :

```
class Markov(object):

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

Στη συνέχεια, μετατρέπουμε τις συναρτήσεις σε μεθόδους. Για παράδειγμα, αυτή είναι η `process_word`:

```
    def process_word(self, word, order=2):
        if len(self.prefix) < order:
            self.prefix += (word,)
            return

        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # if there is no entry for this prefix, make one
            self.suffix_map[self.prefix] = [word]

        self.prefix = shift(self.prefix, word)
```

Η μετατροπή ενός προγράμματος κατά αυτό τον τρόπο, αλλάζοντας τον σχεδιασμό χωρίς να αλλάξουμε τη συνάρτηση, είναι ένα ακόμα παράδειγμα ανακατασκευής κώδικα (βλ. Ενότητα 4.7).

Αυτό το παράδειγμα προτείνει ένα πλάνο ανάπτυξης για το σχεδιασμό αντικειμένων και μεθόδων :

1. Ξεκινήστε γράφοντας συναρτήσεις οι οποίες θα διαβάζουν και θα γράφουν καθολικές μεταβλητές (όταν είναι απαραίτητο).
2. Αφού φτιάξετε ένα λειτουργικό πρόγραμμα, ψάξτε για συσχετίσεις μεταξύ των καθολικών μεταβλητών και των συναρτήσεων που τις χρησιμοποιούν.
3. Ενθυλακώστε σχετικές μεταβλητές ως ιδιότητες ενός αντικειμένου.
4. Μετατρέψτε τις αντίστοιχες συναρτήσεις σε μεθόδους μίας νέας κλάσης.

Άσκηση 18.5. *Κατεβάστε τον κώδικά μου από την Ενότητα 13.8 (*`http://thinkpython.com/code/markov.py`*), και ακολουθήστε τα βήματα που περιγράψαμε παραπάνω για να ενθυλακώσετε τις καθολικές μεταβλητές σαν ιδιότητες μίας νέας κλάσης με όνομα* `Markov`*. Λύση :* `http://thinkpython.com/code/Markov.py` *(προσοχή στο κεφαλαίο Μ).*

18.11 Ορολογία

κωδικοποίηση: Η αναπαράσταση ενός συνόλου τιμών χρησιμοποιώντας ένα άλλο σύνολο τιμών και υλοποιώντας μία αντιστοίχιση μεταξύ τους.

ιδιότητα κλάσης: Μία ιδιότητα που σχετίζεται με ένα αντικείμενο κλάσης. Οι ιδιότητες των κλάσεων ορίζονται μέσα στον ορισμό μιας κλάσης αλλά έξω από οποιαδήποτε μέθοδο.

ιδιότητα στιγμιότυπου: Μία ιδιότητα που σχετίζεται με ένα στιγμιότυπο μιας κλάσης.

καπλαμάς: Μία μέθοδος ή μία συνάρτηση η οποία παρέχει μία διαφορετική διεπαφή σε μία άλλη συνάρτηση χωρίς να κάνει κάποιον ιδιαίτερο υπολογισμό.

κληρονομικότητα: Η ικανότητα να ορίσουμε μία νέα κλάση η οποία είναι μία τροποποιημένη έκδοση μίας ήδη ορισμένης κλάσης.

κλάση γονέας: Η κλάση από όπου μία κλάση παιδί κληρονομεί.

κλάση παιδί: Μία νέα κλάση που δημιουργήθηκε κληρονομώντας μία υπάρχουσα κλάση. Ονομάζεται επίσης και ” υποκλάση ”.

σχέση ΕΙΝΑΙ-ΕΝΑ: Η σχέση μεταξύ μίας κλάσης παιδί και της κλάσης γονέα.

σχέση ΕΧΕΙ-ΕΝΑ: Η σχέση μεταξύ δύο κλάσεων όπου τα στιγμιότυπα της μίας κλάσης περιέχουν αναφορές στα στιγμιότυπα της άλλης.

διάγραμμα κλάσεων: Ένα διάγραμμα το οποίο δείχνει τις κλάσεις ενός προγράμματος και τις σχέσεις μεταξύ τους.

πολλαπλότητα: Μία σημειογραφία σε ένα διάγραμμα κλάσεων η οποία δείχνει, για μία σχέση ΕΧΕΙ-ΕΝΑ, πόσες αναφορές υπάρχουν σε στιγμιότυπα μίας άλλης κλάσης.

18.12 Ασκήσεις

Άσκηση 18.6. *Τα ακόλουθα είναι τα πιθανά χέρια στο πόκερ, σε αύξουσα σειρά αξίας (και φθίνουσα σειρά πιθανότητας) :*

pair: *δύο φύλλα της ίδιας τάξης*

two pair: *δύο ζεύγη φύλλων*

three of a kind: *τρία φύλλα της ίδιας τάξης*

straight: *πέντε φύλλα στη σειρά (οι άσοι μπορούν να είναι είτε υψηλής είτε χαμηλής τάξης, έτσι το Άσος-2-3-4-5 και το 10-Βαλές-Ντάμα-Παπάς-Άσος είναι κέντα αλλά το Ντάμα-Παπάς-Άσος-2-3 δεν είναι).*

flush: *πέντε φύλλα με το ίδιο χρώμα*

full house: *τρία φύλλα ίδιας τάξης και ένα ζεύγος*

four of a kind: *τέσσερα φύλλα της ίδιας τάξης*

straight flush: *πέντε φύλλα στη σειρά με το ίδιο χρώμα*

Στόχος αυτών των ασκήσεων είναι ο υπολογισμός της πιθανότητας να σχεδιαστούν αυτά τα χέρια.

1. *Κατεβάστε αυτά τα αρχεία από το σύνδεσμο* `http://thinkpython.com/code`*:*

   `Card.py` *: Μία πλήρης έκδοση των κλάσεων* `Card`, `Deck` *και* `Hand` *αυτού του κεφαλαίου.*

   `PokerHand.py` *: Μία ελλιπή υλοποίηση μίας κλάσης που αναπαριστά ένα χέρι στο πόκερ και κάποιος κώδικας που την ελέγχει.*

2. *Αν τρέξετε το* `PokerHand.py`*, θα δείτε ότι μοιράζει εφτά χέρια πόκερ των εφτά φύλλων και ελέγχει αν κάποιο από αυτά περιέχει flush. Διαβάστε προσεκτικά αυτόν τον κώδικα προτού συνεχίσετε.*

3. *Προσθέστε μεθόδους στο* `PokerHand.py` *με ονόματα* `has_pair`, `has_twopair` *κτλ. οι οποίες θα επιστρέφουν True ή False ανάλογα με το αν το χέρι ικανοποιεί ή όχι τα κατάλληλα κριτήρια. Ο κώδικάς σας θα πρέπει να δουλεύει σωστά για όλα τα χέρια, ανεξαρτήτως του αριθμού των φύλλων που περιέχουν (παρόλο που τα συνηθέστερα μεγέθη είναι 5 και 7).*

4. *Γράψτε μία μέθοδο με όνομα* `classify` *η οποία θα υπολογίζει την υψηλότερη αξία βάση του συνδυασμού των φύλλων για ένα χέρι και θα θέτει την ιδιότητα* `label` *αναλόγως. Για παράδειγμα, αν ένα χέρι 7 φύλλων περιέχει και ένα φλος και ένα ζεύγος, θα πρέπει να χαρακτηριστεί σαν φλος.*

5. *Όταν σιγουρευτείτε ότι οι μέθοδοι κατηγοριοποίησης δουλεύουν, τότε το επόμενο βήμα είναι να υπολογίσετε τις πιθανότητες των διαφόρων χεριών. Γράψτε μία συνάρτηση μέσα στο* `PokerHand.py` *η οποία θα ανακατεύει μία τράπουλα, θα την χωρίζει σε χέρια, θα κατηγοριοποιεί τα χέρια και θα μετράει πόσες φορές εμφανίζεται κάθε κατηγοριοποίηση.*

6. *Τυπώστε ένα πίνακα με τις κατηγοριοποιήσεις και τις πιθανότητές τους. Τρέξτε το πρόγραμμα με όλο και μεγαλύτερο αριθμό χεριών μέχρι οι τιμές εξόδου να συγκλίνουν σε έναν ικανοποιητικό βαθμό ακρίβειας. Συγκρίνετε τα αποτελέσματά σας με τις τιμές στην διεύθυνση* `http://en.wikipedia.org/wiki/Hand_rankings`*.*

Λύση : `http://thinkpython.com/code/PokerHandSoln.py`*.*

Άσκηση 18.7. *Αυτή η άσκηση χρησιμοποιεί το TurtleWorld του Κεφαλαίου 4. Θα γράψετε κώδικα ο οποίος θα κάνει τις χελώνες να παίξουν tag. Αν δεν γνωρίζετε το παιχνίδι διαβάστε εδώ τους κανόνες :* `http://en.wikipedia.org/wiki/Tag_(game)`*.*

1. *Κατεβάστε το* `http://thinkpython.com/code/Wobbler.py` *και τρέξτε το. Θα πρέπει να δείτε έναν TurtleWorld με τρεις χελώνες. Αν πιέσετε το κουμπί* Run *οι χελώνες περιπλανιούνται τυχαία.*

2. *Διαβάστε τον κώδικα και σιγουρευτείτε ότι καταλαβαίνετε πως δουλεύει. Η κλάση* `Wobbler` *κληρονομεί την* `Turtle`*, που σημαίνει ότι οι μέθοδοι της* `Turtle` `lt`, `rt`, `fd` *και* `bk` *δουλεύουν και σε στιγμιότυπα* `Wobbler`*.*

Η μέθοδος `step` *επικαλείται από την TurtleWorld. Και αυτή με τη σειρά της επικαλείται την* `steer`*, η οποία γυρίζει τη χελώνα στην επιθυμητή κατεύθυνση, την* `wobble`*, η οποία κάνει μία τυχαία στροφή ανάλογα με την απροσεξία της χελώνας, και την* `move`*, η οποία τη μετακινεί μπροστά μερικά πίξελς ανάλογα με την ταχύτητα της.*

3. *Δημιουργήστε ένα αρχείο με όνομα* `Tagger.py`*. Εισάγετε τα πάντα από την* `Wobbler` *και στη συνέχεια ορίστε μία κλάση με όνομα* `Tagger` *η οποία θα κληρονομεί την* `Wobbler`*. Καλέστε την* `make_world` *περνώντας την* `Tagger` *σαν όρισμα.*

4. *Προσθέστε μία μέθοδο* `steer` *στην* `Tagger` *για να παρακάμψει αυτή της* `Wobbler`*. Αρχικά, γράψτε μία έκδοση η οποία στρέφει τη χελώνα στην αρχική της κατεύθυνση. Σημείωση : χρησιμοποιήστε την μαθηματική συνάρτηση* `atan2` *και τις ιδιότητες της χελώνας* `x`*,* `y` *και* `heading`*.*

5. *Τροποποιήστε την* `steer` *έτσι ώστε οι χελώνες να μένουν στα όρια. Μπορείτε να χρησιμοποιήσετε το κουμπί* Step *για την αποσφαλμάτωση, το οποίο επικαλείται την* `step` *μία φορά σε κάθε χελώνα.*

6. *Τροποποιήστε την* `steer` *έτσι ώστε κάθε χελώνα να στρέφεται προς την κοντινότερη γειτονική. Σημείωση : οι χελώνες έχουν μία ιδιότητα, την* `world`*, η οποία είναι μία αναφορά στον TurtleWorld που ζουν, και ο TurtleWorld έχει μία ιδιότητα, την* `animals`*, η οποία είναι μία λίστα με όλες τις χελώνες του κόσμου.*

7. *Τροποποιήστε την* `steer` *έτσι ώστε οι χελώνες να παίζουν tag. Μπορείτε να προσθέσετε μεθόδους στην* `Tagger` *και να παρακάμψετε την* `steer` *και την* `__init__`*, αλλά δεν μπορείτε να τροποποιήσετε ή να παρακάμψετε την* `step`*,* `wobble` *ή* `move`*. Επίσης, η* `steer` *επιτρέπεται να αλλάζει την κατεύθυνση της χελώνας αλλά όχι την θέση της.*

   Προσαρμόστε τους κανόνες και την μέθοδο `steer` *για ένα παιχνίδι καλής ποιότητας. Για παράδειγμα, θα πρέπει να είναι εφικτό για μία αργή χελώνα να κάνει tag τις γρηγορότερες.*

Λύση : `http://thinkpython.com/code/Tagger.py`*.*

Κεφάλαιο 19

Tkinter

19.1 Γραφική διασύνδεση χρήστη

Τα περισσότερα από τα προγράμματα που έχουμε δει μέχρι στιγμής είναι βασισμένα σε κείμενο, αλλά πολλά προγράμματα χρησιμοποιούν γραφική διασύνδεση χρήστη, γνωστή και ως **GUI**.

Η Python παρέχει αρκετές επιλογές για την συγγραφή προγραμμάτων βασισμένα σε GUI, συμπεριλαμβανομένων των wxPython, Tkinter, και Qt. Καθένα έχει τα θετικά του και τα αρνητικά του, για αυτόν το λόγο η Python δεν συγκλίνει προς ένα πρότυπο.

Αυτό που θα σας παρουσιάσω σε αυτό το κεφάλαιο είναι το Tkinter επειδή πιστεύω ότι είναι το ευκολότερο με το οποίο μπορεί να αρχίσει κανείς. Τα περισσότερα τα παραδείγματα αυτού του κεφαλαίου μπορούν να υλοποιηθούν και με τα άλλα αρθώματα.

Υπάρχουν αρκετά βιβλία και ιστοσελίδες σχετικά με το Tkinter. Μία από τις καλύτερες διαδικτυακές πηγές είναι το *An Introduction to Tkinter* του Fredrik Lundh.

Έχω γράψει ένα άρθρωμα με όνομα `Gui.py` το οποίο συνοδεύει το Swampy. Παρέχει μία απλοποιημένη διασύνδεση με τις συναρτήσεις και τις κλάσεις του Tkinter. Τα παραδείγματα αυτού του κεφαλαίου είναι βασισμένα σε αυτό το άρθρωμα.

Αυτό είναι ένα απλό παράδειγμα το οποίο δημιουργεί και εμφανίζει μία γραφική διασύνδεση :

Για να δημιουργήσετε μία τέτοια διασύνδεση, πρέπει να εισάγετε το `Gui` από το Swampy:

```
from swampy.Gui import *
```

Ή έτσι, ανάλογα με το πως εγκαταστήσατε το Swampy:

```
from Gui import *
```

Στη συνέχεια δημιουργήστε ένα αντικείμενο Gui:

```
g = Gui()
g.title('Gui')
g.mainloop()
```

Όταν τρέξετε αυτόν τον κώδικα, θα πρέπει να εμφανιστεί ένα παράθυρο με ένα κενό γκρι τετράγωνο και τον τίτλο Gui. Η `mainloop` τρέχει την **event loop**, η οποία περιμένει το χρήστη να κάνει κάτι και αποκρίνεται ανάλογα. Είναι ένας ατέρμων βρόχος, ο οποίος τρέχει μέχρι ο χρήστης να κλείσει το παράθυρο ή να πατήσει Control-C ή να κάνει κάτι που θα προκαλέσει το κλείσιμο του προγράμματος.

Αυτό το Gui δεν κάνει πολλά επειδή δεν έχει καθόλου **widgets**. Τα widgets είναι στοιχεία τα οποία συνθέτουν ένα GUI. Περιλαμβάνουν :

Κουμπί: Ένα γραφικό στοιχείο, που περιέχει κείμενο ή εικόνα, το οποίο εκτελεί μία ενέργεια όταν πατηθεί.

Καμβάς: Μία περιοχή η οποία μπορεί να εμφανίσει γραμμές, ορθογώνια παραλληλόγραμμα, κύκλους και άλλα σχήματα.

Είσοδος: Μία περιοχή όπου οι χρήστες μπορούν να πληκτρολογήσουν κείμενο.

Μπάρα κύλισης: Ένα γραφικό στοιχείο το οποίο ρυθμίζει το ορατό μέρος ενός άλλου γραφικού στοιχείου.

Πλαίσιο: Ένας περιέκτης, συχνά αόρατος, ο οποίος περιέχει άλλα γραφικά στοιχεία.

Το κενό γκρι τετράγωνο που βλέπετε όταν δημιουργείτε ένα Gui είναι ένα πλαίσιο. Όταν δημιουργείτε ένα νέο γραφικό στοιχείο, τότε προστίθεται σε αυτό το πλαίσιο.

19.2 Κουμπιά και επιστροφές κλήσεων

Η μέθοδος `bu` δημιουργεί ένα κουμπί :

```
button = g.bu(text='Press me.')
```

Οι επιστρεφόμενη τιμή της `bu` είναι ένα αντικείμενο Button. Το κουμπί που εμφανίζεται στο πλαίσιο (Frame) είναι μία γραφική αναπαράσταση αυτού του αντικειμένου. Μπορείτε να ρυθμίσετε το κουμπί επικαλλώντας μεθόδους πάνω του.

Η `bu` παίρνει μέχρι 32 παραμέτρους οι οποίες την εμφάνιση και τη λειτουργία του κουμπιού. Αυτές οι παράμετροι ονομάζονται **επιλογές**. Αντί να παρέχετε τιμές τιμές και για τις 32 επιλογές, μπορείτε να χρησιμοποιήσετε ορίσματα με λέξεις κλειδιά, όπως το `text='Press me.'`, για να ορίσετε μόνο τις επιλογές που χρειάζεστε και να χρησιμοποιήσετε τις προκαθορισμένες τιμές για τις υπόλοιπες.

Όταν προσθέτετε ένα γραφικό στοιχείο στο πλαίσιο, τότε αυτό συρρικνώνεται γύρω από αυτό. Δηλαδή, το πλαίσιο συρρικνώνεται στο μέγεθος του κουμπιού. Αν προσθέσετε περισσότερα γραφικά στοιχεία, το πλαίσιο μεγαλώνει για να τα φιλοξενήσει όλα.

Η μέθοδος `la` δημιουργεί μία επιγραφή (Label):

```
label = g.la(text='Press the button.')
```

Από προεπιλογή, το Tkinter στοιβάζει τα στοιχεία από πάνω προς τα κάτω και τα κεντράρει. Θα δούμε πως παρακάμπτουμε αυτή τη συμπεριφορά σύντομα.

Αν πατήσετε το κουμπί, θα δείτε ότι δεν κάνει πολλά πράγματα. Αυτό συμβαίνει επειδή δεν το έχετε ” συνδέσει ”, δηλαδή δεν του έχετε πει τι να κάνει !

Η επιλογή που ρυθμίζει τη συμπεριφορά ενός κουμπιού είναι η `command`. Η τιμή της `command` είναι μία συνάρτηση η οποία εκτελείται όταν πατιέται το κουμπί. Για παράδειγμα, αυτή η συνάρτηση δημιουργεί ένα νέο Label:

```
def make_label():
    g.la(text='Thank you.')
```

Τώρα μπορούμε να δημιουργήσουμε ένα κουμπί με αυτήν την συνάρτηση ως εντολή (command):

```
button2 = g.bu(text='No, press me!', command=make_label)
```

Όταν πιέσετε αυτό το κουμπί, θα πρέπει να εκτελεστεί η `make_label` και να εμφανιστεί μία νέα επιγραφή.

Η τιμή της επιλογής `command` είναι ένα αντικείμενο συνάρτησης, το οποίο είναι γνωστό ως επιστροφή κλήσης (callback), επειδή αφού καλέσετε την `bu` για να δημιουργήσετε ένα κουμπί, η ροή εκτέλεσης επιστρέφει την κλήση όταν ο χρήστης πατάει το κουμπί.

Αυτού του είδους η ροή είναι χαρακτηριστικό του προγραμματισμού **χειρισμού συμβάντων**. Οι ενέργειες του χρήστη, όπως το πάτημα ενός κουμπιού και οι πληκτρολογήσεις, ονομάζονται **συμβάντα**. Στον προγραμματισμό χειρισμού συμβάντων, η ροή εκτέλεσης καθορίζεται από τις ενέργειες του χρήστη αντί από τον προγραμματιστή.

Η πρόκληση με αυτόν τον προγραμματισμό είναι να κατασκευάσετε ένα σύνολο από γραφικά στοιχεία και επιστροφές κλήσεων τα οποία θα δουλεύουν σωστά (ή τουλάχιστον θα παράγουν τα κατάλληλα μηνύματα λάθους) για οποιαδήποτε σειρά ενεργειών του χρήστη.

Άσκηση 19.1. *Γράψτε ένα πρόγραμμα το οποίο θα δημιουργεί μία διασύνδεση χρήστη με ένα κουμπί. Όταν πατηθεί αυτό το κουμπί θα πρέπει να δημιουργεί ένα δεύτερο κουμπί. Και όταν πατηθεί το δεύτερο, θα πρέπει να δημιουργεί μία επιγραφή η οποία θα λέει : "Nice job!".*

Τι συμβαίνει αν πατήσετε τα κουμπιά παραπάνω από μία φορές ; Λύση : `http://thinkpython.com/code/button_demo.py`

19.3 Γραφικά στοιχεία Καμβά

Ένα από τα πιο πολύπλευρα γραφικά στοιχεία είναι ο καμβάς, ο οποίος δημιουργεί μία περιοχή για τον σχεδιασμό γραμμών, κύκλων και άλλων σχημάτων. Αν έχετε κάνει την Άσκηση 15.4 θα είστε ήδη εξοικειωμένοι με τους καμβάδες.

Η μέθοδος `ca` δημιουργεί ένα νέο καμβά :

```
canvas = g.ca(width=500, height=500)
```

Η `width` και η `height` είναι οι διαστάσεις του καμβά σε εικονοστοιχεία.

Μπορείτε να αλλάξετε τις τιμές των επιλογών, ακόμα και μετά την δημιουργία ενός γραφικού στοιχείου, με τη μέθοδο `config`. Για παράδειγμα, η επιλογή `bg` αλλάζει το χρώμα του φόντου :

```
canvas.config(bg='white')
```

Η τιμή της `bg` είναι μία συμβολοσειρά με το όνομα ενός χρώματος. Το σύνολο των έγκυρων ονομάτων για τα χρώματα είναι διαφέρει ανάλογα με την υλοποίηση της Python, αλλά όλες οι υλοποιήσεις παρέχουν τουλάχιστον τα :

```
white   black
red     green    blue
cyan    yellow   magenta
```

Τα σχήματα σε έναν καμβά ονομάζονται στοιχεία. Για παράδειγμα, η μέθοδος `circle` σχεδιάζει (το μαντέψατε) έναν κύκλο :

```
item = canvas.circle([0,0], 100, fill='red')
```

Το πρώτο όρισμα είναι ένα ζεύγος συντεταγμένων το οποίο ορίζει το κέντρο του κύκλου και το δεύτερο την ακτίνα.

Το `Gui.py` παρέχει ένα κλασικό καρτεσιανό σύστημα συντεταγμένων με αρχή το κέντρο του καμβά και τον θετικό άξονα y να δείχνει προς τα πάνω. Αυτό είναι διαφορετικό από άλλα γραφικά συστήματα όπου η αρχή είναι πάνω αριστερή γωνία και ο άξονας y δείχνει προς τα κάτω.

Η επιλογή `fill` διευκρινίζει ότι ο κύκλος θα πρέπει να γεμίσει με το κόκκινο χρώμα.

Η επιστρεφόμενη τιμή της `circle` είναι ένα αντικείμενο στοιχείου το οποίο παρέχει μεθόδους για την τροποποίηση του στοιχείου στον καμβά. Για παράδειγμα, μπορείτε να χρησιμοποιήσετε την `config` για να αλλάξετε οποιαδήποτε από τις επιλογές του κύκλου :

```
item.config(fill='yellow', outline='orange', width=10)
```

Η `width` είναι το πάχος του περιγράμματος σε εικονοστοιχεία και η `outline` το χρώμα.
Άσκηση 19.2. *Γράψτε ένα πρόγραμμα το οποίο θα δημιουργεί ένα καμβά και ένα κουμπί. Όταν ο χρήστης πατάει το κουμπί, θα πρέπει να σχεδιάζεται ένας κύκλος στον καμβά.*

19.4 Ακολουθίες συντεταγμένων

Η μέθοδος `rectangle` παίρνει μία ακολουθία συντεταγμένων η οποία προσδιορίζει τις απέναντι γωνίες του ορθογωνίου. Αυτό το παράδειγμα, σχεδιάζει ένα πράσινο ορθογώνιο παραλληλόγραμμο με την κάτω αριστερή γωνία στην αρχή και την πάνω δεξιά γωνία στο σημείο $(200, 100)$ του καμβά :

```
canvas.rectangle([[0, 0], [200, 100]],
                 fill='blue', outline='orange', width=10)
```

Αυτός ο τρόπος προσδιορισμού των γωνιών ονομάζεται **πλαίσιο οριοθέτησης** επειδή τα δύο σημεία οριοθετούν το ορθογώνιο.

Η `oval` παίρνει ένα πλαίσιο οριοθέτησης και σχεδιάζει ένα ωοειδές εντός του καθορισμένου ορθογωνίου :

```
canvas.oval([[0, 0], [200, 100]], outline='orange', width=10)
```

Η `line` παίρνει μία ακολουθία συντεταγμένων και σχεδιάζει μία γραμμή η οποία συνδέει τα σημεία. Αυτό το παράδειγμα σχεδιάζει τα δύο σκέλη ενός τριγώνου :

```
canvas.line([[0, 100], [100, 200], [200, 100]], width=10)
```

Η `polygon` παίρνει τα ίδια ορίσματα, αλλά σχεδιάζει το τελευταίο σκέλος ενός πολυγώνου (αν είναι απαραίτητο) και το γεμίζει :

```
canvas.polygon([[0, 100], [100, 200], [200, 100]],
               fill='red', outline='orange', width=10)
```

19.5 Περισσότερα γραφικά στοιχεία

Το Tkinter παρέχει δύο γραφικά στοιχεία τα οποία επιτρέπουν στους χρήστες να πληκτρολογήσουν κείμενο : την είσοδο (Entry), η οποία είναι μία μονή γραμμή, και το Text, το οποίο έχει πολλές γραμμές.

Η `en` δημιουργεί ένα νέο Entry:

```
entry = g.en(text='Default text.')
```

Η επιλογή `text` σας επιτρέπει να βάλετε κείμενο στην είσοδο όταν δημιουργηθεί. Η μέθοδος `get` επιστρέφει τα περιεχόμενα της εισόδου (τα οποία μπορεί να έχει αλλάξει ο χρήστης) :

```
>>> entry.get()
'Default text.'
```

Η `te` δημιουργεί ένα γραφικό στοιχείο Text:

```
text = g.te(width=100, height=5)
```

Η `width` και η `height` είναι οι διαστάσεις του στοιχείου σε χαρακτήρες και γραμμές.

Η `insert` βάζει κείμενο μέσα στο Text:

```
text.insert(END, 'A line of text.')
```

Η `END` είναι ένας ειδικός δείκτης ο οποίος υποδεικνύει τον τελευταίο χαρακτήρα μέσα στο Text.

Μπορείτε επίσης να προσδιορίσετε έναν χαρακτήρα χρησιμοποιώντας έναν διάστικτο δείκτη, όπως ο `1.1`, ο οποίος έχει τον αριθμό των γραμμών πριν την τελεία και τον αριθμό των στηλών μετά. Το παρακάτω παράδειγμα προσθέτει τα γράμματα `'nother'` μετά τον πρώτο χαρακτήρα της πρώτης γραμμής.

```
>>> text.insert(1.1, 'nother')
```

Η μέθοδος `get` διαβάζει το κείμενο μέσα από το γραφικό στοιχείο παίρνοντας τους δείκτες αρχής και τέλους σαν ορίσματα. Το παράδειγμα που ακολουθεί επιστρέφει όλο το κείμενο που βρίσκεται μέσα στο στοιχείο, συμπεριλαμβανομένου του χαρακτήρα αλλαγής γραμμής :

```
>>> text.get(0.0, END)
'Another line of text.\n'
```

Η μέθοδος `delete` διαγράφει το κείμενο από το στοιχείο. Αυτό το παράδειγμα τα σβήνει όλα εκτός από τους δύο πρώτους χαρακτήρες :

```
>>> text.delete(1.2, END)
>>> text.get(0.0, END)
'An\n'
```

Άσκηση 19.3. *Τροποποιήστε την λύση της άσκησης 19.2 προσθέτοντας ένα Entry και ένα δεύτερο κουμπί. Όταν ο χρήστης πατάει το δεύτερο κουμπί, θα πρέπει να διαβάζει ένα χρώμα από το Entry και να το χρησιμοποιεί για να αλλάξει το χρώμα γεμίσματος του κύκλου. Χρησιμοποιήστε την* `config` *για να τροποποιήσετε τον ήδη υπάρχον κύκλο, μην δημιουργήσετε καινούριο.*

Το πρόγραμμά σας θα πρέπει να μπορεί να χειριστεί την περίπτωση όπου ο χρήστης θα προσπαθήσει να αλλάξει το χρώμα ενός κύκλου ο οποίος δεν έχει δημιουργηθεί, και την περίπτωση όπου το όνομα του χρώματος δεν είναι έγκυρο.

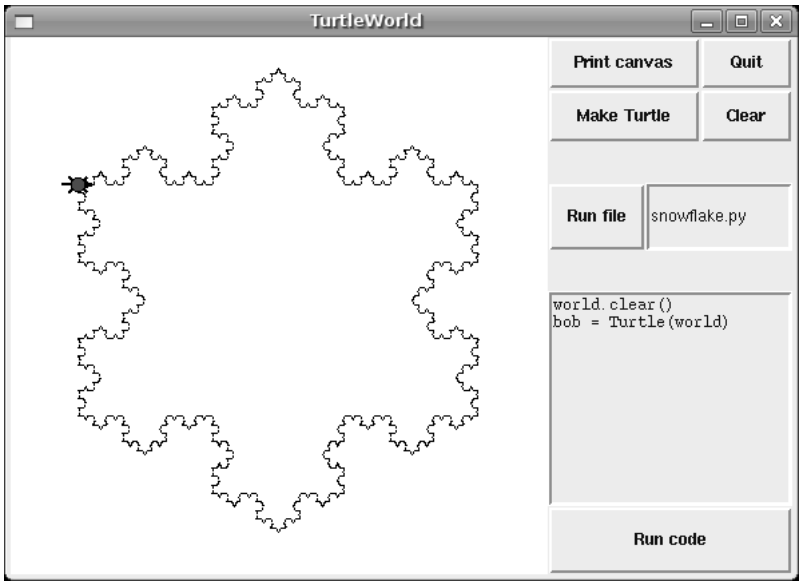
Μπορείτε να δείτε την λύση μου στον σύνδεσμο : `http://thinkpython.com/code/circle_demo.py`.

19.6 Πακετάρισμα γραφικών στοιχείων

Μέχρι στιγμής έχουμε στοιβάξει τα γραφικά στοιχεία σε μία ενιαία στήλη, αλλά στα περισσότερα GUIs η διάταξη είναι πιο πολύπλοκη. Για παράδειγμα, η Εικόνα 19.1 δείχνει μία απλοποιημένη έκδοση του TurtleWorld (βλ. Κεφάλαιο 4).

Σε αυτήν την ενότητα παρουσιάζεται ο κώδικας που δημιουργεί αυτό το GUI, σπασμένος σε μια σειρά βημάτων. Μπορείτε να κατεβάσετε ολόκληρο το παράδειγμα από εδώ : `http://thinkpython.com/code/SimpleTurtleWorld.py`.

Στο ανώτατο επίπεδο, αυτό το GUI περιέχει δύο γραφικά στοιχεία, έναν Καμβά και ένα Πλαίσιο, διατεταγμένα σε μια σειρά. Άρα το πρώτο βήμα είναι να δημιουργήσουμε αυτήν τη σειρά.



Σχήμα 19.1: Διάγραμμα κλάσεων.

```
class SimpleTurtleWorld(TurtleWorld):
    """This class is identical to TurtleWorld, but the code that
    lays out the GUI is simplified for explanatory purposes."""

    def setup(self):
        self.row()
        ...
```

Η setup είναι η συνάρτηση που δημιουργεί και οργανώνει τα γραφικά στοιχεία. Η οργάνωση των στοιχείων σε ένα GUI ονομάζεται πακετάρισμα.

Η row δημιουργεί ένα πλαίσιο σειράς και το κάνει το ” τρέχον πλαίσιο ”. Όλα τα επόμενα γραφικά στοιχεία πακετάρονται σε μία σειρά, μέχρι να κλείσει αυτό το πλαίσιο ή να δημιουργηθεί ένα άλλο.

Αυτός είναι ο κώδικας που δημιουργεί τον καμβά και το πλαίσιο στήλης όπου κρατάει τα άλλα στοιχεία :

```
        self.canvas = self.ca(width=400, height=400, bg='white')
        self.col()
```

Το πρώτο στοιχείο στην στήλη είναι ένα πλαίσιο πλέγματος, το οποίο περιέχει τέσσερα κουμπιά διατεταγμένα ανά δύο :

```
        self.gr(cols=2)
        self.bu(text='Print canvas', command=self.canvas.dump)
        self.bu(text='Quit', command=self.quit)
        self.bu(text='Make Turtle', command=self.make_turtle)
        self.bu(text='Clear', command=self.clear)
        self.endgr()
```

Η gr δημιουργεί ένα πλέγμα, με όρισμα τον αριθμό των στηλών. Τα στοιχεία διατάσσονται στο πλέγμα από τα αριστερά προς τα δεξιά και από πάνω προς τα κάτω.

Το πρώτο κουμπί χρησιμοποιεί την `self.canvas.dump` σαν επιστροφή κλήσης και το δεύτερο χρησιμοποιεί την `self.quit`. Αυτές είναι **δεσμευμένες μέθοδοι**, το οποίο σημαίνει ότι σχετίζονται με ένα συγκεκριμένο αντικείμενο. Όταν γίνεται επίκλησή τους, επικαλούνται στο αντικείμενο.

Το επόμενο γραφικό στοιχείο μέσα στη στήλη είναι ένα πλαίσιο γραμμής το οποίο περιέχει ένα κουμπί και μία είσοδο :

```
self.row([0,1], pady=30)
self.bu(text='Run file', command=self.run_file)
self.en_file = self.en(text='snowflake.py', width=5)
self.endrow()
```

Το πρώτο όρισμα στην `row` είναι μία λίστα από βάρη η οποία καθορίζει πόσος επιπλέον χώρος διατίθεται μεταξύ των γραφικών στοιχείων. Η λίστα `[0,1]` σημαίνει ότι όλος ο επιπλέον χώρος επιμερίζεται στο δεύτερο στοιχείο, το οποίο είναι η είσοδος (Entry). Αν τρέξετε αυτόν τον κώδικα και αυξήσετε το μέγεθος του παραθύρου, θα δείτε ότι η είσοδος μεγαλώνει αλλά το κουμπί όχι.

Η επιλογή `pady` " γεμίζει " αυτήν τη γραμμή προς την κατεύθυνση y, προσθέτοντας 30 εικονοστοιχεία πάνω και κάτω.

Η `endrow` περατώνει αυτήν τη γραμμή στοιχείων και άρα τα επόμενα στοιχεία πακετάρονται στο πλαίσιο στήλης. Το `Gui.py` κρατάει μία στοίβα από πλαίσια :

- Όταν χρησιμοποιείτε τις `row`, `col` ή `gr` για να δημιουργήσετε ένα πλαίσιο (Frame), πηγαίνει στην αρχή της στοίβας και γίνεται το τρέχον πλαίσιο.
- Όταν χρησιμοποιείτε τις `endrow`, `endcol` ή `endgr` για να κλείσετε ένα πλαίσιο, πηδάει έξω από τη στοίβα και το προηγούμενο πλαίσιο γίνεται τρέχον.

Η μέθοδος `run_file` διαβάζει τα περιεχόμενα της εισόδου, τα χρησιμοποιεί σαν όνομα αρχείου, διαβάζει τα περιεχόμενα και τα περνάει στην `run_code`. Η `self.inter` είναι ένα αντικείμενο διερμηνέα (Interpreter) το οποίο ξέρει πως να πάρει μία συμβολοσειρά και να την εκτελέσει σαν κώδικα Python.

```
def run_file(self):
    filename = self.en_file.get()
    fp = open(filename)
    source = fp.read()
    self.inter.run_code(source, filename)
```

Τα τελευταία δύο γραφικά στοιχεία είναι ένα Text και ένα κουμπί (Button):

```
self.te_code = self.te(width=25, height=10)
self.te_code.insert(END, 'world.clear()\n')
self.te_code.insert(END, 'bob = Turtle(world)\n')

self.bu(text='Run code', command=self.run_text)
```

Η `run_text` παρόμοια με την `run_file` με τη διαφορά ότι παίρνει κώδικα από το στοιχείο Text αντί από ένα αρχείο :

```
def run_text(self):
    source = self.te_code.get(1.0, END)
    self.inter.run_code(source, '<user-provided code>')
```

Δυστυχώς, οι λεπτομέρειες της διάταξης ενός γραφικού στοιχείου είναι διαφορετικές σε άλλες γλώσσες και στα άλλα αρθρώματα της Python. Μόνο το Tkinter παρέχει τρεις διαφορετικούς μηχανισμούς

για την οργάνωση των στοιχείων. Αυτοί οι μηχανισμοί ονομάζονται **διαχειριστές γεωμετρίας**. Αυτός που έδειξα σε αυτήν την ενότητα είναι ο "grid" και οι άλλοι δύο ονομάζονται "pack" και "place".

Ευτυχώς, οι περισσότερες έννοιες αυτής της ενότητας εφαρμόζονται και σε άλλα αρθρώματα γραφικής διασύνδεσης και άλλες γλώσσες.

19.7 Μενού και αντικείμενα με δυνατότητα κλήσης

Ένα κουμπί μενού (Menubutton) είναι ένα γραφικό στοιχείο που μοιάζει με ένα απλό κουμπί, αλλά όταν πατηθεί αναδύεται ένα μενού. Το μενού εξαφανίζεται όταν ο χρήστης επιλέξει από τα στοιχεία του.

Αυτός ο κώδικας δημιουργεί ένα έγχρωμο μενού επιλογών (μπορείτε να το κατεβάσετε από `http://thinkpython.com/code/menubutton_demo.py`):

```
g = Gui()
g.la('Select a color:')
colors = ['red', 'green', 'blue']
mb = g.mb(text=colors[0])
```

Η `mb` δημιουργεί ένα κουμπί μενού. Αρχικά, το κείμενο στο κουμπί είναι το όνομα του προεπιλεγμένου χρώματος. Ο ακόλουθος βρόχος δημιουργεί τα στοιχεία του μενού, ένα για κάθε χρώμα :

```
for color in colors:
    g.mi(mb, text=color, command=Callable(set_color, color))
```

Το πρώτο όρισμα της `mi` είναι το κουμπί μενού με το οποίο σχετίζονται αυτά τα στοιχεία.

Η επιλογή `command` είναι ένα αντικείμενο με δυνατότητα κλήσης, το οποίο είναι κάτι νέο. Μέχρι στιγμής έχουμε δει συναρτήσεις και μεθόδους οριοθέτησης να χρησιμοποιούνται σαν επιστροφές κλήσεων, το οποίο δουλεύει μια χαρά αν δεν χρειάζεται να περάσετε κάποιο όρισμα στη συνάρτηση. Σε αντίθετη περίπτωση, πρέπει να κατασκευάσετε ένα αντικείμενο με δυνατότητα επιστροφής κλήσης (Callable object) το οποίο θα περιέχει μία συνάρτηση, όπως η `set_color`, και τα ορίσματά της, όπως το `color`.

Αυτό το αντικείμενο αποθηκεύει μία αναφορά στη συνάρτηση και τα ορίσματα σαν ιδιότητες. Αργότερα, όταν ο χρήστης κλικάρει σε ένα στοιχείο του μενού, η επιστρεφόμενη κλήση καλεί τη συνάρτηση και περνάει τα αποθηκευμένα ορίσματα.

Η `set_color` θα μπορούσε να είναι αυτή :

```
def set_color(color):
    mb.config(text=color)
    print color
```

Όταν ο χρήστης επιλέξει ένα στοιχείο από το μενού, καλείται η `set_color`, διαμορφώνει το κουμπί του μενού ούτως ώστε να δείχνει το νέο-επιλεγμένο χρώμα και στη συνέχεια το εμφανίζει. Αν δοκιμάσετε να τρέξετε αυτό το παράδειγμα, μπορείτε να επιβεβαιώσετε ότι η `set_color` καλείται όταν επιλέγετε ένα στοιχείο (και όχι όταν δημιουργείται το αντικείμενο Callable).

19.8 Δεσμοί

Ένας **δεσμός** (binding) είναι μία σχέση μεταξύ ενός γραφικού στοιχείου, ενός συμβάντος και μιας επιστροφής κλήσης : όταν προκληθεί ένα συμβάν (όπως το πάτημα ενός κουμπιού), επικαλείται η επιστροφή κλήσης.

Πολλά γραφικά στοιχεία έχουν προεπιλεγμένους δεσμούς. Για παράδειγμα, όταν πατάτε ένα κουμπί, ο προεπιλεγμένος δεσμός αλλάζει το ανάγλυφο του κουμπιού και το κάνει να φαίνεται κάπως μελαγχολικό. Όταν αφήνετε το κουμπί, ο δεσμός επαναφέρει την εμφάνιση του και επικαλείται την επιστροφή κλήσης που καθορίζεται από την επιλογή `command`.

Μπορείτε να χρησιμοποιήσετε την μέθοδο `bind` για να παρακάμψετε τους προεπιλεγμένους δεσμούς ή για να προσθέσετε νέους. Για παράδειγμα, αυτός ο κώδικας δημιουργεί ένα νέο δεσμό για έναν καμβά (μπορείτε να κατεβάσετε τον κώδικα αυτής της ενότητας από εδώ `http://thinkpython.com/code/draggable_demo.py`):

```
ca.bind('<ButtonPress-1>', make_circle)
```

Το πρώτο όρισμα είναι μία συμβολοσειρά συμβάντος, το οποίο ενεργοποιείται όταν ο χρήστης πατήσει το αριστερό κουμπί του ποντικιού. Άλλα συμβάντα του ποντικιού είναι το `ButtonMotion`, το `ButtonRelease` και το `Double-Button`.

Το δεύτερο όρισμα είναι ένας χειριστής συμβάντων. Ένας χειριστής συμβάντων είναι μία συνάρτηση ή μία μέθοδος οριοθέτησης, όπως μία επιστροφή κλήσης, αλλά με τη διαφορά ότι ένας χειριστής παίρνει ένα αντικείμενο συμβάντος σαν παράμετρο. Αυτό είναι ένα παράδειγμα :

```
def make_circle(event):
    pos = ca.canvas_coords([event.x, event.y])
    item = ca.circle(pos, 5, fill='red')
```

Το αντικείμενο συμβάντος περιέχει πληροφορίες σχετικά με τον τύπο του συμβάντος και λεπτομέρειες όπως οι συντεταγμένες του δείκτη του ποντικιού. Σε αυτό το παράδειγμα η πληροφορία που χρειαζόμαστε είναι η θέση του κλικ του ποντικιού. Αυτές οι τιμές είναι συντεταγμένες σε εικονοστοιχεία, οι οποίες καθορίζονται από το γραφικό υποσύστημα. Η μέθοδος `canvas_coords` τις μετατρέπει σε συντεταγμένες καμβά, οι οποίες είναι συμβατές με τις μεθόδους του καμβά όπως η `circle`.

Για τα γραφικά στοιχεία εισόδου, συνηθίζεται να δεσμεύουμε το συμβάν `<Return>`, το οποίο ενεργοποιείται όταν ο χρήστης πατάει το πλήκτρο Return ή το Enter. Για παράδειγμα, ο ακόλουθος κώδικας δημιουργεί ένα κουμπί (Button) και μία είσοδο (Entry):

```
bu = g.bu('Make text item:', make_text)
en = g.en()
en.bind('<Return>', make_text)
```

Η `make_text` καλείται όταν πατιέται το κουμπί ή όταν ο χρήστης πατήσει το Return ενώ πληκτρολογεί στην είσοδο. Για να το κάνουμε να δουλέψει, χρειαζόμαστε μία συνάρτηση η οποία θα μπορεί να καλεστεί σαν επιλογή (χωρίς ορίσματα) ή σαν ένας χειριστής συμβάντος (με ένα συμβάν σαν όρισμα) :

```
def make_text(event=None):
    text = en.get()
    item = ca.text([0,0], text)
```

Η `make_text` παίρνει τα περιεχόμενα της εισόδου και το κείμενό της σε ένα στοιχείο Text στον καμβά.

Επίσης, μπορούμε να δημιουργήσουμε δεσμούς για τα στοιχεία του καμβά. Ακολουθεί ένας ορισμός για την κλάση `Draggable`, η οποία είναι παιδί της κλάσης `Item` η οποία παρέχει δεσμούς οι οποίοι υλοποιούν τη δυνατότητα να σύρουμε και να αφήσουμε (drag-and-drop).

```
class Draggable(Item):

    def __init__(self, item):
        self.canvas = item.canvas
        self.tag = item.tag
        self.bind('<Button-3>', self.select)
        self.bind('<B3-Motion>', self.drag)
        self.bind('<Release-3>', self.drop)
```

Η μέθοδος init παίρνει ένα στοιχείο σαν παράμετρο, αντιγράφει τις ιδιότητές του και στη συνέχεια δημιουργεί δεσμούς για τρία συμβάντα : το πάτημα του κουμπιού, την κίνηση του κουμπιού και την ελευθέρωσή του.

Ο χειριστής συμβάντων `select` αποθηκεύει τις συντεταγμένες του τρέχοντος συμβάντος και το αρχικό χρώμα του στοιχείου και στη συνέχεια αλλάζει το χρώμα σε κίτρινο :

```
    def select(self, event):
        self.dragx = event.x
        self.dragy = event.y

        self.fill = self.cget('fill')
        self.config(fill='yellow')
```

Το `cget` σημαίνει “get configuration”. Παίρνει αυτό το όνομα από μία επιλογή σαν συμβολοσειρά και επιστρέφει την τρέχουσα τιμή αυτής της επιλογής.

Η `drag` υπολογίζει πόσο μακριά μετακινήθηκε το αντικείμενο σε σχέση με την αρχική του θέση, ενημερώνει τις αποθηκευμένες συντεταγμένες και στη συνέχεια μετακινεί το στοιχείο.

```
    def drag(self, event):
        dx = event.x - self.dragx
        dy = event.y - self.dragy

        self.dragx = event.x
        self.dragy = event.y

        self.move(dx, dy)
```

Αυτός ο υπολογισμός γίνεται σε συντεταγμένες εικονοστοιχείων (δεν υπάρχει λόγος να μετατραπούν σε συντεταγμένες καμβά).

Τέλος, η `drop` επαναφέρει το αρχικό χρώμα του στοιχείου :

```
    def drop(self, event):
        self.config(fill=self.fill)
```

Μπορείτε να χρησιμοποιήσετε την κλάση `Draggable` για προσθέσετε δυνατότητα drag-and-drop σε ένα υπάρχον στοιχείο. Για παράδειγμα, αυτή είναι μία τροποποιημένη έκδοση της `make_circle` η οποία χρησιμοποιεί την `circle` για να δημιουργήσει ένα στοιχείο και την `Draggable` για να το κάνει ” συρόμενο ”:

```
def make_circle(event):
    pos = ca.canvas_coords([event.x, event.y])
    item = ca.circle(pos, 5, fill='red')
```

```
    item = Draggable(item)
```

Αυτό το παράδειγμα επιδεικνύει ένα από τα πλεονεκτήματα της κληρονομικότητας : μπορείτε να τροποποιήσετε της δυνατότητες μίας γονικής κλάσης χωρίς να τροποποιήσετε τον ορισμό της. Αυτό είναι ιδιαίτερα χρήσιμο αν θέλετε να αλλάξετε τη συμπεριφορά που ορίζεται σε ένα άρθρωμα που δεν γράψατε εσείς.

19.9 Αποσφαλμάτωση

Μία από τις προκλήσεις του προγραμματισμού με γραφική διασύνδεση είναι η παρακολούθηση του τι συμβαίνει όταν χτίζεται το GUI και του τι γίνεται αργότερα ως απόκριση στις ενέργειες του χρήστη.

Για παράδειγμα, είναι σύνηθες λάθος όταν φτιάχνετε μία επιστροφή κλήσης να καλείτε την συνάρτηση αντί να περνάτε μία αναφορά σε αυτήν :

```
def the_callback():
    print 'Called.'

g.bu(text='This is wrong!', command=the_callback())
```

Αν τρέξετε αυτόν τον κώδικα, θα δείτε ότι καλεί την `the_callback` αμέσως, και στη συνέχεια δημιουργεί ένα κουμπί. Όταν πατάτε το κουμπί, δεν κάνει τίποτα επειδή η επιστρεφόμενη τιμή της `the_callback` είναι `None`. Δεν θα θέλατε να επικαλεστείτε μία επιστροφή κλήσης κατά την ανάπτυξη ενός GUI αλλά μόνο μετά, σαν απόκριση σε μία ενέργεια του χρήστη.

Μία άλλη πρόκληση του προγραμματισμού με γραφική διασύνδεση είναι ότι δεν έχετε τον έλεγχο της ροής εκτέλεσης. Οι ενέργειες του χρήστη καθορίζουν ποια τμήματα του κώδικα θα εκτελεστούν και με ποια σειρά. Αυτό σημαίνει ότι πρέπει να σχεδιάσετε το πρόγραμμά σας με τέτοιο τρόπο ούτως ώστε να δουλεύει σωστά με οποιαδήποτε πιθανή σειρά συμβάντων.

Για παράδειγμα, το GUI της άσκησης 19.3 έχει δύο γραφικά στοιχεία : το ένα δημιουργεί ένα στοιχείο Circle και το άλλο αλλάζει το χρώμα του κύκλου. Αν ο χρήστης δημιουργήσει τον κύκλο και στη συνέχεια αλλάξει το χρώμα του δεν υπάρχει κανένα πρόβλημα. Αλλά αν ο χρήστης αλλάξει το χρώμα ενός κύκλου που δεν υπάρχει ακόμα ; Ή δημιουργήσει περισσότερους του ενός κύκλου ;

Όσο αυξάνεται ο αριθμός των γραφικών στοιχείων, γίνεται όλο και πιο δύσκολο να φανταστούμε όλες τις πιθανές σειρές συμβάντων. Ένας τρόπος για να χειριστούμε αυτήν την πολυπλοκότητα είναι να ενθυλακώσουμε την κατάσταση του συστήματος σε ένα αντικείμενο και στη συνέχεια να εξετάσουμε :

- Ποιες είναι οι πιθανές καταστάσεις ; Στο παράδειγμα με τον κύκλο, θα πρέπει να εξετάσουμε δύο καταστάσεις : πριν και μετά τη δημιουργία του πρώτου κύκλου από το χρήστη.
- Σε κάθε κατάσταση, ποια συμβάντα μπορεί να προκύψουν; Στο παράδειγμα, ο χρήστης μπορεί να πατήσει οποιοδήποτε από τα κουμπιά ή να το κλείσει.
- Για κάθε ζευγάρι κατάστασης-συμβάντος, ποιο είναι το επιθυμητό αποτέλεσμα ; Από τη στιγμή που υπάρχουν δύο καταστάσεις και δύο κουμπιά, υπάρχουν τέσσερα ζευγάρια κατάστασης-συμβάντος που πρέπει να εξεταστούν.
- Τι μπορεί να προκαλέσει μία μετάβαση από μία κατάσταση σε μία άλλη ; Σε αυτήν την περίπτωση, υπάρχει μία μετάβαση όταν ο χρήστης δημιουργεί τον πρώτο κύκλο.

Επίσης, μπορεί να σας φανεί χρήσιμο να ορίσετε και να ελέγξετε σταθερές οι οποίες θα κρατάνε ανεξάρτητα τη σειρά των συμβάντων.

Αυτή η προσέγγιση στον προγραμματισμό με διασύνδεση χρήστη μπορεί να σας βοηθήσει να γράψετε σωστό κώδικα χωρίς να αφιερώσετε χρόνο στον έλεγχο κάθε πιθανής ακολουθίας συμβάντων !

19.10 Ορολογία

GUI: Μία γραφική διασύνδεση χρήστη.

widget: Ένα από τα γραφικά στοιχεία που συνθέτουν μία γραφική διασύνδεση χρήστη, συμπεριλαμβανομένων κουμπιών, μενού, πεδία εισαγωγής κειμένου κτλ.

επιλογή: Μία τιμή η οποία ρυθμίζει την εμφάνιση ή τη συνάρτηση ενός γραφικού στοιχείου.

όρισμα λέξη κλειδί: Ένα όρισμα το οποίο υποδεικνύει το όνομα της παραμέτρου σαν μέρος της κλήσης συνάρτησης.

επιστροφή κλήσης: Μία συνάρτηση συσχετισμένη με ένα widget η οποία καλείται όταν ο χρήστης εκτελεί μία ενάργεια.

μέθοδος οριοθέτησης: Μία μέθοδος η οποία σχετίζεται με ένα συγκεκριμένο στιγμιότυπο.

προγραμματισμός βάση συμβάντων: Ένα είδος προγραμματισμού στο οποίο η ροή της εκτέλεσης καθορίζεται από τις ενέργειες του χρήστη.

συμβάν: Μία ενέργεια του χρήστη, όπως το κλικ του ποντικιού ή το πάτημα ενός κουμπιού, η οποία προκαλεί την απόκριση του GUI.

βρόχος συμβάντων: Ένας ατέρμον βρόχος ο οποίος περιμένει τις ενέργειες του χρήστη και αποκρίνεται.

στοιχείο: Ένα γραφικό στοιχείο σε ένα widget καμβά.

πλαίσιο οριοθέτησης: Ένα ορθογώνιο παραλληλόγραμμο το οποίο περικλείει ένα σύνολο στοιχείων, συνήθως καθορισμένα από δύο απέναντι γωνίες.

πακέτο: Για την οργάνωση και εμφάνιση των στοιχείων ενός GUI.

διαχειριστής γεωμετρίας: Ένα σύστημα για το πακετάρισμα γραφικών στοιχείων.

δεσμός: Μία συσχέτιση μεταξύ ενός γραφικού στοιχείου, ενός συμβάντος και ενός χειριστή συμβάντων. Ο χειριστής συμβάντων καλείται όταν προκύψει ένα συμβάν σε ένα γραφικό στοιχείο.

19.11 Ασκήσεις

Άσκηση 19.4. *Για αυτήν την άσκηση θα γράψετε ένα πρόγραμμα προβολής εικόνων. Αυτό είναι ένα απλό παράδειγμα :*

```
g = Gui()
canvas = g.ca(width=300)
photo = PhotoImage(file='danger.gif')
canvas.image([0,0], image=photo)
g.mainloop()
```

Η PhotoImage διαβάζει ένα αρχείο και επιστρέφει ένα αντικείμενο PhotoImage το οποίο μπορεί να εμφανίσει το Tkinter. Η Canvas.image τοποθετεί την εικόνα στον καμβά, κεντραρισμένη στις δοθείσες συντεταγμένες. Μπορείτε επίσης να βάλετε εικόνες και πάνω σε ετικέτες, κουμπιά και κάποια άλλα γραφικά στοιχεία :

```
g.la(image=photo)
g.bu(image=photo)
```

Η PhotoImage μπορεί να χειριστεί μόνο μερικούς τύπους εικόνας, όπως τον GIF και τον PPM, αλλά μπορούμε να χρησιμοποιήσουμε την βιβλιοθήκη εικόνων της Python (PIL) για να διαβάσουμε και άλλα αρχεία.

Το όνομα του αρθρώματος PIL είναι Image, αλλά το Tkinter ορίζει ένα αντικείμενο με το ίδιο όνομα. Για να αποφευχθεί η σύγκρουση, μπορείτε να χρησιμοποιήσετε την import...as με αυτόν τον τρόπο :

```
import Image as PIL
import ImageTk
```

Η πρώτη γραμμή εισάγει το Image και του δίνει το τοπικό όνομα PIL. Η δεύτερη γραμμή εισάγει το ImageTk, το οποίο μπορεί να μεταφράσει μία εικόνα PIL σε ένα Tkinter PhotoImage. Για παράδειγμα :

```
image = PIL.open('allen.png')
photo2 = ImageTk.PhotoImage(image)
g.la(image=photo2)
```

1. Κατεβάστε τα image_demo.py, danger.gif και allen.png από εδώ `http://thinkpython.com/code`. Τρέξτε το image_demo.py. Ίσως χρειαστεί να εγκαταστήσετε τα PIL και ImageTk. Θα τα βρείτε πιθανώς στις αποθήκες λογισμικού, αλλά αν όχι τότε μπορείτε να τα κατεβάσετε από το σύνδεσμο `pythonware.com/products/pil/`.

2. Στο image_demo.py αλλάξτε το όνομα του δεύτερου PhotoImage από photo2 σε photo και ξανατρέξτε το πρόγραμμα. Θα πρέπει να δείτε το δεύτερο PhotoImage αλλά όχι το πρώτο.

   Το πρόβλημα είναι ότι όταν επανεκχωρείτε την photo παρακάμπτεται η αναφορά στο πρώτο PhotoImage, το οποίο στη συνέχεια εξαφανίζεται. Το ίδιο συμβαίνει και αν εκχωρήσετε ένα PhotoImage σε μια τοπική μεταβλητή, εξαφανίζεται με το τέλος της συνάρτησης.

   Για να αποφύγετε αυτό το πρόβλημα, πρέπει να αποθηκεύσετε μία αναφορά σε κάθε PhotoImage που θέλετε να κρατήσετε. Μπορείτε να χρησιμοποιήσετε μία καθολική μεταβλητή ή να αποθηκεύσετε τα PhotoImages σε δομές δεδομένων ή σαν μία ιδιότητα ενός αντικειμένου.

   Αυτή η συμπεριφορά μπορεί είναι απογοητευτική, για αυτό σας προειδοποιώ (και για αυτό η εικόνα του παραδείγματος λέει "Danger!").

3. Γράψτε ένα πρόγραμμα βασισμένοι σε αυτό το παράδειγμα, το οποίο θα παίρνει το όνομα ενός καταλόγου και θα διέρχεται μέσω όλων των αρχείων, εμφανίζοντας τα αρχεία που η το PIL αναγνωρίζει σαν εικόνες. Μπορείτε να χρησιμοποιήσετε μία δήλωση try για να πιάσετε τα αρχεία που δεν αναγνωρίζει το PIL.

   Όταν ο χρήστης κάνει κλικ πάνω στην εικόνα, το πρόγραμμα θα πρέπει να εμφανίζει την επόμενη.

4. Το PIL παρέχει διάφορες μεθόδους για τον χειρισμό εικόνων. Μπορείτε να διαβάσετε σχετικά με αυτό στο σύνδεσμο `http://pythonware.com/library/pil/handbook`. Σαν πρόκληση, διαλέξτε μερικές από αυτές τις μεθόδους και δημιουργήστε ένα GUI για να τις εφαρμόσετε σε εικόνες.

Λύση : `http://thinkpython.com/code/ImageBrowser.py`.

Άσκηση 19.5. *Ένας επεξεργαστής διανυσματικών γραφικών είναι ένα πρόγραμμα το οποίο δίνει στους χρήστες τη δυνατότητα να σχεδιάσουν και να επεξεργαστούν σχήματα στην οθόνη και να δημιουργήσουν αρχεία εξόδου σε τύπους διανυσματικών γραφικών όπως το Postscript και το SVG.*

Γράψτε έναν απλό επεξεργαστή διανυσματικών γραφικών χρησιμοποιώντας το Tkinter. Θα πρέπει, τουλάχιστον, να επιτρέπει στους χρήστες να σχεδιάσουν γραμμές, κύκλους και ορθογώνια. Χρησιμοποιήστε την `Canvas.dump` *για παράγετε το σχετικό Postscript των περιεχομένων του καμβά.*

Σαν πρόκληση, μπορείτε να επιτρέψετε τους χρήστες να επιλέγουν και να αλλάζουν το μέγεθος των στοιχείων του καμβά.

Άσκηση 19.6. *Χρησιμοποιήστε το Tkinter για να γράψετε ένα απλό περιηγητή ιστού. Θα πρέπει να έχει ένα γραφικό στοιχείο Text όπου ο χρήστης θα μπορεί να εισάγει ένα URL και ένα Canvas για να δείχνει τα περιεχόμενα της σελίδας.*

Μπορείτε να χρησιμοποιήσετε το άρθρωμα `urllib` *για να κατεβάσετε αρχεία (βλ. άσκηση 14.6) και το άρθρωμα* `HTMLParser` *για την ανάλυση των HTML tags (δείτε εδώ* `http://docs.python.org/2/library/htmlparser.html`*).*

Κατ'ελάχιστο, ο περιηγητής σας θα πρέπει να χειρίζεται απλό κείμενο και υπερσυνδέσμους. Σαν πρόκληση μπορείτε να χειριστείτε χρώματα φόντου, διαμόρφωση ετικετών και εικόνων.

Παράρτημα Α΄

Αποσφαλμάτωση

Σε ένα πρόγραμμα μπορούν να συμβούν διαφόρων ειδών σφάλματα και είναι χρήσιμο να γίνει διάκριση μεταξύ τους προκειμένου να μπορούμε να τα εντοπίσουμε γρηγορότερα :

- Τα συντακτικά λάθη παράγονται από την Python όταν μεταφράζει τον πηγαίο κώδικα σε byte code (όχι γλώσσα μηχανής). Συνήθως, υποδεικνύουν ότι υπάρχει κάποιο λάθος στη σύνταξη του προγράμματος. Παράδειγμα : Η παράλειψη της άνω κάτω τελείας στο τέλος μίας δήλωσης `def` αποδίδει το κάπως περιττό μήνυμα `SyntaxError: invalid syntax`.
- Τα λάθη χρόνου εκτέλεσης παράγονται από τον διερμηνέα αν πάει κάτι στραβά κατά την εκτέλεση του προγράμματος. Τα περισσότερα μηνύματα αυτών των σφαλμάτων περιέχουν πληροφορίες σχετικά με το που συνέβη το σφάλμα και τι συναρτήσεις εκτελούνταν. Παράδειγμα : Μία άπειρη αναδρομή, τελικώς, θα προκαλέσει το σφάλμα χρόνου εκτέλεσης "maximum recursion depth exceeded."
- Τα σημασιολογικά λάθη είναι αποτελούν προβλήματα σε ένα πρόγραμμα το οποίο τρέχει χωρίς να παράγει κάποιο μήνυμα λάθους αλλά δεν κάνει αυτό που θα έπρεπε. Παράδειγμα : Μία έκφραση μπορεί να μην υπολογίζεται με τη σειρά που θα περιμένατε, αποδίδοντας ένα εσφαλμένο αποτέλεσμα.

Το πρώτο βήμα της αποσφαλμάτωσης είναι να καταλάβετε με τι είδους λάθος έχετε να κάνετε. Παρόλο που οι ακόλουθες ενότητες είναι οργανωμένες με βάση τον τύπο του σφάλματος, μερικές τεχνικές μπορούν να εφαρμοστούν σε περισσότερες από μία περιπτώσεις.

Α΄.1 Συντακτικά λάθη

Συνήθως, τα συντακτικά λάθη είναι εύκολο να διορθωθούν από τη στιγμή που καταλάβετε ποια είναι. Δυστυχώς, τις περισσότερες φορές τα μηνύματα λάθους δεν είναι και πολύ χρήσιμα. Τα συνηθέστερα μηνύματα είναι το `SyntaxError: invalid syntax` και το `SyntaxError: invalid token`, εκ των οποίων κανένα δεν είναι ιδιαίτερα κατατοπιστικό.

Αφετέρου, το μήνυμα σας λέει που συνέβη το σφάλμα μέσα στο πρόγραμμα. Για την ακρίβεια, σας λέει που παρατήρησε το πρόβλημα η Python, το οποίο δεν είναι απαραίτητα και το που ακριβώς είναι το λάθος. Μερικές φορές το λάθος είναι πριν τη θέση του μηνύματος λάθους. Συνήθως είναι στην αμέσως προηγούμενη γραμμή.

Εάν χτίζετε σταδιακά το πρόγραμμα, τότε μάλλον γνωρίζεται που είναι το λάθος. Θα είναι στην τελευταία γραμμή που προσθέσατε.

Αν αντιγράφετε κώδικα από ένα βιβλίο, ξεκινήστε συγκρίνοντας πολύ προσεκτικά τον κώδικά σας με αυτόν του βιβλίου. Ελέγξτε κάθε χαρακτήρα. Συγχρόνως, θυμηθείτε ότι μπορεί να είναι λάθος το βιβλίο, επομένως αν δείτε κάτι που μοιάζει με συντακτικό λάθος τότε μπορεί και να είναι.

Αυτοί είναι μερικοί τρόποι για να αποφύγετε τα πιο συνηθισμένα συντακτικά λάθη :

1. Σιγουρευτείτε ότι δεν χρησιμοποιείτε κάποια λέξη κλειδί σαν όνομα μεταβλητής.
2. Ελέγξτε ότι έχετε άνω και κάτω τελεία στο τέλος της επικεφαλίδας κάθε σύνθετης δήλωσης, συμπεριλαμβανομένων των `for`, `while`, `if`, και `def`.
3. Σιγουρευτείτε ότι όλες οι συμβολοσειρές μέσα στον κώδικα έχουν σωστές αντιστοιχίες εισαγωγικών.
4. Αν έχετε συμβολοσειρές πολλαπλών γραμμών με τριπλά εισαγωγικά (μονά ή διπλά), σιγουρευτείτε ότι τερματίσατε κατάλληλα τη συμβολοσειρά. Μία μη τερματισμένη συμβολοσειρά μπορεί να προκαλέσει ένα λάθος `invalid token` στο τέλος του προγράμματος, ή μπορεί να χειριστεί το υπόλοιπο μέρος του προγράμματος σαν μία συμβολοσειρά μέχρι να φτάσει στην επόμενη συμβολοσειρά. Στην δεύτερη περίπτωση, μπορεί να μην παραχθεί κανένα μήνυμα λάθους !
5. Ένας τελεστής ανοίγματος που δεν έχει κλείσει, όπως οι `(`, `{`, και `[`, κάνει την Python να συνεχίσει στην επόμενη γραμμή σαν να ήταν μέρος της τρέχουσας δήλωσης. Σε γενικές γραμμές, θα προκύψει λάθος σχεδόν αμέσως στην επόμενη γραμμή.
6. Ελέγξτε για το κλασικό = αντί του == μέσα σε μία συνθήκη.
7. Ελέγξτε την ενδοπαραγραφοποίηση για να σιγουρευτείτε ότι ευθυγραμμίζονται με τον τρόπο που θα έπρεπε. Η Python μπορεί να χειριστεί κενά διαστήματα και στηλοθέτες, αλλά αν τα ανακατέψετε μπορεί να προκαλέσετε προβλήματα. Ο καλύτερος τρόπος για να αποφύγετε τέτοια προβλήματα είναι να χρησιμοποιείτε έναν κειμενογράφο που γνωρίζει την Python και παράγει κατάλληλη ενδοπαραγραφοποίηση.

Αν δεν δουλέψει τίποτα από αυτά, προχωρήστε στην επόμενη ενότητα...

Α΄.1.1 Συνεχίζω να κάνω αλλαγές αλλά δεν υπάρχει διαφορά.

Αν ο διερμηνέας λέει ότι υπάρχει ένα λάθος και εσείς δεν το βλέπετε, τότε μπορεί εσείς και ο διερμηνέας να κοιτάτε διαφορετικό κώδικα. Ελέγξτε το προγραμματιστικό σας περιβάλλον για σιγουρευτείτε ότι το πρόγραμμα που επεξεργάζεστε είναι το ίδιο με αυτό που προσπαθεί να τρέξει η Python.

Αν δεν είστε σίγουροι, δοκιμάστε να βάλετε ένα προφανές συντακτικό λάθος στην αρχή του κώδικα σκόπιμα. Τώρα ξανατρέξτε το. Αν ο διερμηνέας δεν βρει το νέο λάθος, τότε δεν τρέχετε το νέο κώδικα.

Υπάρχουν κάποιοι πιθανοί ένοχοι :

- Επεξεργαστήκατε το αρχείο και ξεχάσατε να αποθηκεύσετε τις αλλαγές πριν το ξανατρέξετε. Μερικά προγραμματιστικά περιβάλλοντα το κάνουν αυτό από μόνα τους, αλλά κάποια άλλα όχι.
- Αλλάξατε το όνομα του αρχείου αλλά προσπαθείτε να το τρέξετε με το παλιό όνομα.
- Κάτι στο προγραμματιστικό σας περιβάλλον δεν έχει ρυθμιστεί σωστά.
- Αν γράφετε ένα άρθρωμα και χρησιμοποιείτε την `import`, σιγουρευτείτε ότι δεν δώσατε στο άρθρωμά σας το ίδιο όνομα με ένα από τα πρότυπα αρθρώματα της Python.

- Αν χρησιμοποιείτε την `import` για να διαβάσετε ένα άρθρωμα, θυμηθείτε ότι πρέπει να επανεκκινήστε τον διερμηνέα ή να χρησιμοποιήσετε την `reload` για να διαβάσετε ένα τροποποιημένο αρχείο. Αν επανεισάγετε απλά το άρθρωμα, δεν θα κάνει κάτι.

Αν κολλήσετε και δεν μπορείτε να καταλάβετε τι συμβαίνει, μία προσέγγιση είναι να ξεκινήσετε από την αρχή με ένα νέο πρόγραμμα όπως το "Hello, World!" και να σιγουρευτείτε ότι μπορείτε να τρέξετε ένα τέτοιο γνωστό πρόγραμμα. Στη συνέχεια προσθέστε σταδιακά τα κομμάτια του αρχικού προγράμματος στο νέο.

Α΄.2 Λάθη χρόνου εκτέλεσης

Από τη στιγμή που το πρόγραμμά σας είναι συντακτικά σωστό, η Python μπορεί να το μεταγλωττίσει και να αρχίσει τουλάχιστον να το τρέχει. Τι θα μπορούσε ενδεχομένως να πάει στραβά ;

Α΄.2.1 Το πρόγραμμά μου δεν κάνει απολύτως τίποτα.

Συνήθως, αυτό το πρόβλημα προκύπτει όταν το αρχείο σας αποτελείται από συναρτήσεις και κλάσεις αλλά στην πραγματικότητα δεν επικαλείται τίποτα για να αρχίσει η εκτέλεση. Αυτό μπορεί να είναι σκόπιμο αν σχεδιάζετε να εισάγεται αυτό το άρθρωμα για να παρέχετε κλάσεις και συναρτήσεις.

Αν δεν είναι σκόπιμο, σιγουρευτείτε ότι επικαλείστε μία συνάρτηση για να αρχίσει η εκτέλεση, ή εκτελέστε μία στην διαδραστική λειτουργία. Δείτε επίσης και την ενότητα " Ροή Εκτέλεσης " παρακάτω.

Α΄.2.2 Το πρόγραμμά μου κρεμάει.

Αν ένα πρόγραμμα σταματήσει και φαίνεται να μην κάνει τίποτα, τότε λέμε ότι " κρέμασε ". Συχνά, αυτό σημαίνει ότι το έχει κολλήσει μέσα σε έναν ατέρμων βρόχο ή μία ατέρμονη αναδρομή.

- Αν υποψιάζεστε ότι το πρόβλημα είναι ένας συγκεκριμένος βρόχος, προσθέστε μία δήλωση `print` αμέσως πριν το βρόχο η οποία θα λέει "entering the loop" και μία άλλη αμέσως μετά η οποία θα λέει "exiting the loop."

  Δοκιμάστε να ξανατρέξετε το πρόγραμμα. Αν πάρετε το πρώτο μήνυμα αλλά όχι το δεύτερο τότε έχετε έναν ατέρμων βρόχο. Πηγαίνετε στην ενότητα " Ατέρμων Βρόχος " παρακάτω.

- Τις περισσότερες φορές, μία ατέρμονη αναδρομή θα αναγκάσει το πρόγραμμα να τρέξει για λίγο και στη συνέχεια θα παραχθεί ένα σφάλμα "RuntimeError: Maximum recursion depth exceeded". Αν συμβεί αυτό πηγαίνετε στην ενότητα "Ατέρμονη Αναδρομή" παρακάτω.

  Ακόμα και αν δεν πάρετε αυτό το λάθος αλλά υποπτεύεστε ότι υπάρχει πρόβλημα με μία αναδρομική μέθοδο ή συνάρτηση, μπορείτε να χρησιμοποιήσετε τις ίδιες τεχνικές τις ενότητας " Ατέρμονη Αναδρομή ".

- Αν δεν δουλέψει κανένα από τα παραπάνω βήματα, αρχίστε να ελέγχετε άλλους βρόχους και άλλες αναδρομικές συναρτήσεις και μεθόδους.

- Αν δεν δουλέψει ούτε αυτό, τότε είναι πιθανό να μην έχετε καταλάβει τη ροή εκτέλεσης του προγράμματος. Πηγαίνετε στην ενότητα " Ροή Εκτέλεσης " παρακάτω.

Ατέρμων Βρόχος

Αν πιστεύετε ότι έχετε έναν ατέρμων βρόχο και νομίζετε ότι ξέρετε ποιος βρόχος προκαλεί το πρόβλημα, προσθέστε μία δήλωση `print` στο τέλος του βρόχου, η οποία θα εμφανίζει τις τιμές των μεταβλητών μέσα στην συνθήκη και την τιμή της συνθήκης.

Για παράδειγμα :

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y

    print "x: ", x
    print "y: ", y
    print "condition: ", (x > 0 and y < 0)
```

Τώρα, όταν τρέξετε το πρόγραμμα, θα δείτε τρεις γραμμές εξόδου για κάθε επανάληψη του βρόχου. Στην τελευταία επανάληψη, η συνθήκη θα πρέπει να είναι ψευδής. Αν ο βρόχος συνεχίσει να εκτελείται, θα είστε σε θέση να δείτε τις τιμές των `x` και `y`, και μπορεί να καταλάβετε για ποιο λόγο δεν ενημερώνονται σωστά.

Ατέρμονη Αναδρομή

Τις περισσότερες φορές, μία ατέρμονη αναδρομή προκαλεί την εκτέλεση του προγράμματος για λίγο και στη συνέχεια παράγεται ένα σφάλμα `Maximum recursion depth exceeded`.

Αν υποψιάζεστε ότι μία συνάρτηση ή μία μέθοδος προκαλεί ατέρμονη αναδρομή, ξεκινήστε ελέγχοντας αν υπάρχει περίπτωση βάσης. Με άλλα λόγια, θα πρέπει να υπάρχει μία συνθήκη η οποία θα αναγκάσει τη συνάρτηση ή τη μέθοδο να επιστρέψει χωρίς να κάνει αναδρομική επίκληση. Αν όχι, τότε θα πρέπει να επανεξετάσετε τον αλγόριθμο και να προσδιορίσετε μία περίπτωση βάσης.

Αν υπάρχει περίπτωση βάσης αλλά το πρόγραμμα δεν φαίνεται να την φτάνει, προσθέστε μία δήλωση `print` στην αρχή της συνάρτησης ή της μεθόδου η οποία θα τυπώνει τις παραμέτρους. Τώρα, όταν τρέξετε το πρόγραμμα, θα δείτε μερικές γραμμές εξόδου για κάθε επίκληση της συνάρτησης ή της μεθόδου με τις παραμέτρους. Αν οι παράμετροι δεν κινούνται προς την περίπτωση βάσης, θα πάρετε μερικές ιδέες σχετικά με το πρόβλημα.

Ροή Εκτέλεσης

Αν δεν έχετε καταλάβει ακριβώς τη ροή εκτέλεσης του προγράμματός σας, προσθέστε δηλώσεις `print` στην αρχή κάθε συνάρτησης με ένα μήνυμα όπως το "entering function `foo`", όπου `foo` το όνομα της συνάρτησης.

Τώρα, όταν τρέξετε το πρόγραμμά, θα τυπώσει ένα ίχνος για κάθε συνάρτηση καθώς θα γίνεται επίκληση σε αυτή.

Α΄.2.3 Παίρνω μία εξαίρεση όταν τρέχω το πρόγραμμα.

Αν κάτι πάει στραβά κατά την εκτέλεση, η Python τυπώνει ένα μήνυμα το οποίο περιέχει το όνομα της εξαίρεσης, τη γραμμή του προγράμματος που συνέβη το πρόβλημα και ένα traceback.

Το traceback προσδιορίζει τη συνάρτηση που εκτελείται αυτή τη στιγμή, τη συνάρτηση που την επικαλέστηκε, τη συνάρτηση που επικαλέστηκε αυτή και ούτω καθεξής. Με άλλα λόγια, ιχνηλατεί

τη σειρά με την οποία έγιναν οι επικλήσεις μέχρι εκεί που είστε. Επίσης, περιέχει τους αριθμούς των γραμμών του αρχείου που συνέβησαν αυτές οι κλήσεις.

Το πρώτο βήμα είναι να εξετάσετε το σημείο του προγράμματος που συνέβη το λάθος και να δείτε να μπορείτε να καταλάβετε τι έγινε. Αυτά είναι μερικά από τα πιο συνηθισμένα λάθη χρόνου εκτέλεσης :

NameError: Προσπαθείτε να χρησιμοποιήσετε μία μεταβλητή η οποία δεν υπάρχει στο συγκεκριμένο περιβάλλον. Θυμηθείτε ότι οι τοπικές μεταβλητές είναι τοπικές. Δεν μπορείτε να αναφέρεστε σε αυτές εκτός της συνάρτησης που ορίστηκαν.

TypeError: Υπάρχουν πολλές πιθανές αιτίες :

- Προσπαθείτε να χρησιμοποιήσετε μία τιμή εσφαλμένα. Παράδειγμα : ευρετηρίαση συμβολοσειράς, λίστας ή πλειάδας με κάτι που δεν είναι ακέραιος.
- Υπάρχει μία αναντιστοιχία μεταξύ των στοιχείων μίας συμβολοσειρά διαμόρφωσης και των στοιχείων που περνιούνται για μετατροπή. Αυτό μπορεί να συμβεί είτε αν δεν ταιριάζει ο αριθμός των στοιχείων είτε αν καλείται μία ακατάλληλη αναδρομή.
- Περνάτε λάθος αριθμό ορισμάτων σε μία συνάρτηση ή μέθοδο. Για τις μεθόδους, κοιτάξτε τον ορισμό και ελέγξτε αν η πρώτη παράμετρος είναι η `self`. Στη συνέχεια κοιτάξτε την επίκληση της μεθόδου για σιγουρευτείτε ότι επικαλείστε τη μέθοδο σε ένα αντικείμενο σωστού τύπου και ότι παρέχετε σωστά τα υπόλοιπα ορίσματα.

KeyError: Προσπαθείτε να αποκτήσετε πρόσβαση σε κάποιο από τα στοιχεία ενός λεξικού χρησιμοποιώντας ένα κλειδί που δεν περιέχει το λεξικό.

AttributeError: Προσπαθείτε να αποκτήσετε πρόσβαση σε μία ιδιότητα ή μία μέθοδο που δεν υπάρχει. Ελέγξτε την ορθογραφία ! Μπορεί να χρησιμοποιήσετε την `dir` για να εμφανίσετε τις ιδιότητες που υπάρχουν.

Αν ένα AttributeError υποδεικνύει ότι ένα αντικείμενο έχει `NoneType`, αυτό σημαίνει ότι είναι `None`. Μία κοινή αιτία είναι ότι μπορεί να ξεχάσατε να επιστρέψετε μία τιμή. Αν φτάσετε στο τέλος μίας συνάρτησης χωρίς να πετύχετε μία δήλωση `return`, τότε η συνάρτηση επιστρέφει `None`. Μία άλλη κοινή αιτία είναι η χρήση του αποτελέσματος μίας μεθόδου λιστών, όπως η `sort`, η οποία επιστρέφει `None`.

IndexError: Ο δείκτης που χρησιμοποιείτε για να αποκτήσετε πρόσβαση σε μία λίστα, συμβολοσειρά ή πλειάδα είναι μεγαλύτερος από το μήκος της μείον ένα. Προσθέστε μία δήλωση `print` αμέσως πριν την περιοχή του σφάλματος, για να εμφανίσετε την τιμή του δείκτη και το μήκος της διάταξης. Έχει ο πίνακας το σωστό μέγεθος ; Έχει ο δείκτης τη σωστή τιμή ;

Ο αποσφαλματωτής της Python (`pdb`) είναι χρήσιμος για τον εντοπισμό εξαιρέσεων επειδή σας επιτρέπει να εξετάσετε την κατάσταση του προγράμματος ακριβώς πριν το σφάλμα. Μπορείτε να διαβάσετε σχετικά με τον `pdb` στην διεύθυνση : `http://docs.python.org/2/library/pdb.html`.

Α΄.2.4 Πρόσθεσα πολλές δηλώσεις `print` και πελάγωσα με την έξοδο.

Ένα από τα προβλήματα με τη χρήση δηλώσεων `print` για την αποσφαλμάτωση είναι ότι μπορεί να καταλήξετε " θαμμένοι μέσα στην έξοδο ". Υπάρχουν δύο τρόποι για να αντιμετωπίσετε μία τέτοια κατάσταση : να απλοποιήσετε την έξοδο ή να απλοποιήσετε το πρόγραμμα.

Για να απλοποιήσετε την έξοδο, μπορείτε να αφαιρέσετε ή να μετατρέψετε σε σχόλια τις δηλώσεις `print` που δεν σας βοηθάνε, ή να τις συνενώσετε ή ακόμα και να διαμορφώσετε την έξοδο έτσι ώστε να είναι ευκολότερο να την καταλάβετε.

Για να απλοποιήσετε το πρόγραμμα, μπορείτε να ακολουθήσετε διάφορους τρόπους. Αρχικά, περιορίστε το πρόβλημα του προγράμματος. Για παράδειγμα, αν ψάχνετε μία λίστα, ψάξτε για μία μικρή λίστα. Αν το πρόγραμμα παίρνει δέχεται είσοδο από το χρήστη, δώστε την μικρότερη είσοδο που προκαλεί το πρόβλημα.

Δεύτερον, καθαρίστε το πρόγραμμα. Αφαιρέστε νεκρό κώδικα και αναδιοργανώστε το πρόγραμμα για να κάνετε όσο το δυνατόν ευκολότερη την ανάγνωσή του. Για παράδειγμα, αν υποψιάζεστε ότι το πρόβλημα βρίσκεται σε ιδιαίτερα εμφωλευμένο σημείο του προγράμματος, δοκιμάστε να ξαναγράψετε αυτό το κομμάτι κώδικα με μία απλούστερη δομή. Αν υποψιάζεστε μία μεγάλη συνάρτηση, δοκιμάστε να τη χωρίσετε σε μικρότερες συναρτήσεις και να τις δοκιμάσετε χωριστά.

Συχνά, η διαδικασία ανεύρεσης της απλούστερης περίπτωσης προς δοκιμή σας οδηγεί στο σφάλμα. Αν διαπιστώσετε ότι ένα πρόγραμμα δουλεύει σε μία περίπτωση αλλά όχι σε μία άλλη, τότε αυτό σας δίνει μια ιδέα για το τι μπορεί να συμβαίνει.

Παρομοίως, η επανεγγραφή ενός μέρος του κώδικα μπορεί να σας βοηθήσει να βρείτε ανεπαίσθητα σφάλματα. Αν κάνετε μία αλλαγή η οποία θεωρείτε ότι δεν επηρεάζει το πρόγραμμα, και το επηρεάσει, τότε αυτό μπορεί να σας φανεί χρήσιμο.

Α΄.3 Σημασιολογικά λάθη

Κατά μία έννοια, τα σημασιολογικά λάθη είναι τα δυσκολότερα στην αποσφαλμάτωση, επειδή ο διερμηνέας δεν παρέχει καμία πληροφορία σχετικά με το τι πήγε στραβά. Μόνο εσείς γνωρίζετε τι υποτίθεται ότι πρέπει να κάνει το πρόγραμμα.

Το πρώτο βήμα είναι να κάνετε μία σύνδεση μεταξύ του κειμένου του προγράμματος και της συμπεριφοράς που βλέπετε. Χρειάζεστε μία υπόθεση σχετικά με το τι κάνει το πρόγραμμα στην πραγματικότητα. Ένα από τα πράγματα που κάνουν δυσκολότερη την αποσφαλμάτωση τέτοιων λαθών είναι ότι οι υπολογιστές τρέχουν πολύ γρήγορα.

Συχνά, θα θέλατε να μπορούσατε να επιβραδύνετε την εκτέλεση του προγράμματος σε μία ανθρώπινη ταχύτητα. Αυτό μπορείτε να το κάνετε με κάποιους αποσφαλματωτές, αλλά ο χρόνος που χρειάζεται για να εισάγετε μερικές ” καλά τοποθετημένες ” δηλώσεις `print` είναι συνήθως μικρότερος σε σχέση με το χρόνο που χρειάζεται για να στήσετε έναν αποσφαλματωτή, να εισάγετε και να αφαιρείτε σημεία διακοπής και να καθυστερήσετε το πρόγραμμα εκεί που συμβαίνει το λάθος.

Α΄.3.1 Το πρόγραμμά μου δεν δουλεύει.

Θα πρέπει να κάνετε στον εαυτό σας τις ακόλουθες ερωτήσεις :

- Υπάρχει κάτι που υποτίθεται ότι έπρεπε να κάνει το πρόγραμμα αλλά δεν φαίνεται να συμβαίνει ; Βρείτε το τμήμα του κώδικα που βρίσκεται η αντίστοιχη συνάρτηση και σιγουρευτείτε ότι εκτελείται όταν πρέπει.
- Υπάρχει κάτι που συμβαίνει αλλά δεν θα έπρεπε συμβαίνει ; Βρείτε τον κώδικα της αντίστοιχης συνάρτησης μέσα στο πρόγραμμά σας και ελέγξτε αν εκτελείται όταν δεν θα έπρεπε.
- Υπάρχει τμήμα του κώδικα που παράγει μη αναμενόμενο αποτέλεσμα ; Σιγουρευτείτε ότι καταλαβαίνετε τον εν λόγω κώδικα, ειδικά αν περιλαμβάνει επικλήσεις σε συναρτήσεις ή μεθόδους σε άλλα αρθρώματα της Python. Διαβάστε την τεκμηρίωση για τις συναρτήσεις που επικαλείστε.

Για να προγραμματίσετε, πρέπει να έχετε ένα νοητικό μοντέλο σχετικά με το πως δουλεύουν τα προγράμματα. Πολύ συχνά, αν γράψετε ένα πρόγραμμα το οποίο δεν κάνει αυτό που θα περιμένατε, το πρόβλημα δεν βρίσκεται μέσα στο πρόγραμμα αλλά μέσα στο νοητικό σας μοντέλο.

Ο καλύτερος τρόπος για να διορθώσετε το νοητικό σας μοντέλο είναι να σπάσετε το πρόγραμμα στις επιμέρους συνιστώσες (συνήθως τις συναρτήσεις και τις μεθόδους) και να ελέγξετε κάθε συνιστώσα ανεξάρτητα. Από τη στιγμή που θα βρείτε τη διαφορά μεταξύ του μοντέλου σας και της πραγματικότητας, μπορείτε να λύσετε το πρόβλημα.

Φυσικά, θα πρέπει να κατασκευάζετε και να ελέγχετε τις συνιστώσες όσο αναπτύσσετε το πρόγραμμα. Έτσι, αν αντιμετωπίσετε ένα πρόβλημα θα υπάρχει μόνο ένα νέο μικρό κομμάτι κώδικα το οποίο δεν θα γνωρίζετε αν είναι σωστό.

Α΄.3.2 Έχω μία μεγάλη έκφραση η οποία δεν κάνει αυτό που θα περίμενα.

Το γράψιμο σύνθετων εκφράσεων είναι μια χαρά όσο είναι αναγνώσιμες, αλλά μπορεί να μπορεί να αποδειχθεί σπαζοκεφαλιά στην αποσφαλμάτωση. Συχνά, είναι καλή ιδέα να σπάτε μία σύνθετη έκφραση σε μία σειρά από εκχωρήσεις σε προσωρινές μεταβλητές.

Για παράδειγμα η :

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

Μπορεί να γραφτεί και ως :

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

Αυτή η ρητή έκδοση είναι ευκολότερο να διαβαστεί επειδή τα ονόματα των μεταβλητών παρέχουν επιπλέον τεκμηρίωση και είναι ευκολότερη στην αποσφαλμάτωση επειδή μπορείτε να ελέγξετε τους τύπους των ενδιάμεσων μεταβλητών και να εμφανίσετε τις τιμές τους.

Ένα ακόμα πρόβλημα που μπορεί να προκύψει από τις μεγάλες εκφράσεις είναι ότι η σειρά εκτίμησης των πράξεων μπορεί να είναι διαφορετική από αυτήν που θα περιμένατε. Για παράδειγμα, αν μεταφράζατε την έκφραση $\frac{x}{2\pi}$ στην Python, ίσως γράφατε :

```
y = x / 2 * math.pi
```

Το οποίο όμως δεν είναι σωστό επειδή ο πολλαπλασιασμός και η διαίρεση έχουν την ίδια προτεραιότητα και υπολογίζονται από τα αριστερά προς τα δεξιά. Άρα αυτή η έκφραση υπολογίζεται έτσι $x\pi/2$.

Ένας καλός τρόπος για την αποσφαλμάτωση εκφράσεων είναι η προσθήκη παρενθέσεων για να γίνει σαφή η σειρά των πράξεων :

```
y = x / (2 * math.pi)
```

Κάθε φορά που αμφιβάλλετε για τη σειρά των πράξεων, χρησιμοποιήστε παρενθέσεις. Όχι μόνο για να είναι σωστό το πρόγραμμα (με την έννοια ότι θα κάνει αυτό που θα θέλατε) αλλά και για να είναι πιο ευανάγνωστο για όσους ανθρώπους δεν έχουν απομνημονεύσει τους κανόνες προτεραιότητας.

Α΄.3.3 Έχω μία μέθοδο ή μία συνάρτηση η οποία δεν επιστρέφει αυτό που θα περίμενα.

Αν έχετε μία δήλωση `return` με μία σύνθετη έκφραση δεν έχετε καμία ελπίδα να τυπώσετε την επιστρεφόμενη τιμή προτού επιστρέψει η συνάρτηση ή η μέθοδος. Και πάλι, μπορείτε να χρησιμοποιήσετε μία προσωρινή μεταβλητή. Για παράδειγμα, αντί για αυτό :

```
return self.hands[i].removeMatches()
```

μπορείτε να γράψετε : you could write:

```
count = self.hands[i].removeMatches()
return count
```

Τώρα έχετε τη δυνατότητα να εμφανίσετε την τιμή της `count` και πριν την επιστροφή.

Α΄.3.4 Έχω κολλήσει πραγματικά και χρειάζομαι βοήθεια.

Πρώτα απ' όλα, δοκιμάστε να απομακρυνθείτε από τον υπολογιστή για μερικά λεπτά. Οι υπολογιστές εκπέμπουν κύματα τα οποία επηρεάζουν τον εγκέφαλο και προκαλούν αυτά τα συμπτώματα :

- Frustration and rage.
- Δεισιδαίμονες πεποιθήσεις (" ο υπολογιστής με μισεί ") και μαγικές προκαταλήψεις (" το πρόγραμμα δουλεύει μόνο όταν φοράω ανάποδα το καπέλο μου ").
- Προγραμματισμό στα τυφλά (η προσπάθεια να προγραμματίσετε γράφοντας κάθε πιθανό πρόγραμμα και διαλέγοντας αυτό που δίνει το σωστό αποτέλεσμα).

Αν διαπιστώσετε ότι πάσχετε από κάποιο από αυτά τα συμπτώματα, σηκωθείτε και πάτε για μια βόλτα. Όταν ηρεμήσετε, σκεφτείτε ξανά το πρόγραμμα. Τι ακριβώς κάνει ; Ποιες είναι οι πιθανές αιτίες για αυτήν τη συμπεριφορά ; Πότε ήταν η τελευταία φορά που δούλευε το πρόγραμμα και τι κάνατε μετά ;

Μερικές φορές χρειάζεται λίγο χρόνο για να βρείτε το σφάλμα. Συχνά, διαπιστώνω ότι βρίσκω σφάλματα όταν είμαι μακριά από τον υπολογιστή και αφήνω το μυαλό μου ελεύθερο. Μερικά από τα καλύτερα μέρη για να βρείτε σφάλματα είναι τα τραίνα, το ντους και το κρεβάτι ακριβώς πριν πέσετε για ύπνο.

Α΄.3.5 Όχι, χρειάζομαι πραγματικά βοήθεια.

Συμβαίνει. Ακόμα και οι καλύτεροι προγραμματιστές κολλάνε ενίοτε. Μερικές φορές, δουλεύετε πάνω σε ένα πρόγραμμα για πολύ καιρό και για αυτό δεν μπορείτε να δείτε το λάθος. Αυτό που χρειάζεστε είναι ένα φρέσκο ζευγάρι μάτια.

Πριν φωνάξετε κάποιον άλλο για βοήθεια, σιγουρευτείτε ότι είστε προετοιμασμένοι. Το πρόγραμμά σας πρέπει να είναι όσο το δυνατόν απλούστερο και θα πρέπει να δουλεύετε με τη μικρότερη είσοδο που προκαλεί το πρόβλημα. Θα πρέπει να έχετε δηλώσεις `print` στα κατάλληλα σημεία (και η έξοδός τους θα πρέπει να είναι κατανοητή). Θα πρέπει να έχετε καταλάβει αρκετά καλά το πρόβλημα για μπορέσετε να το περιγράψετε συνοπτικά.

Όταν φέρετε κάποιον για βοήθεια, φροντίστε να του δώσετε τις πληροφορίες που χρειάζεται :

- Αν υπάρχει κάποιο μήνυμα λάθους, ποιό είναι αυτό και ποιο μέρος του προγράμματος υποδεικνύει ;

- Ποιο ήταν το τελευταίο πράγμα που κάνατε προτού προκληθεί αυτό το λάθος ; Ποιες ήταν οι τελευταίες γραμμές κώδικα που γράψατε ή ποια είναι η τελευταία περίπτωση που ελέγξατε και απέτυχε ;
- Τι έχετε δοκιμάσει μέχρι στιγμής και τι έχετε αντιληφθεί ;

Όταν βρείτε το σφάλμα, αφιερώστε λίγο χρόνο στο να σκεφτείτε τι θα μπορούσατε να είχατε κάνει για να το βρείτε γρηγορότερα. Την επόμενη φορά που θα δείτε κάτι παρόμοιο, θα είστε σε θέση να βρείτε το λάθος πολύ πιο γρήγορα.

Θυμηθείτε, ο στόχος δεν είναι μόνο να κάνετε το πρόγραμμα να δουλέψει, Ο στόχος είναι να μάθετε πως θα κάνετε το πρόγραμμα να δουλέψει.

Παράρτημα Β΄

Ανάλυση Αλγορίθμων

> Αυτό το παράρτημα αποτελεί ένα τροποποιημένο απόσπασμα από το βιβλίο *Think Complexity*, του Allen B. Downey, το οποίο έχει επίσης δημοσιευτεί από την O'Reilly Media (2011). Όταν τελειώσετε με αυτό το βιβλίο, ίσως θα θέλατε να συνεχίσετε με εκείνο.

Η ανάλυση αλγορίθμων είναι ένας κλάδος της επιστήμης των υπολογιστών ο οποίος μελετάει την απόδοση των αλγορίθμων και ειδικότερα το χρόνο εκτέλεσης και τις απαιτήσεις σε χώρο. (βλ. `http://en.wikipedia.org/wiki/Analysis_of_algorithms`).

Ο στόχος της ανάλυσης αλγορίθμων είναι να προβλέψει τις επιδόσεις των διαφόρων αλγορίθμων προκειμένου να κατευθύνει τις αποφάσεις σχεδιασμού.

Κατά την προεκλογική περίοδο των Ηνωμένων Πολιτειών του 2008, ζητήθηκε από τον υποψήφιο πρόεδρο Μπαράκ Ομπάμα να κάνει μία αυτοσχέδια ανάλυση όταν επισκέφτηκε την Google. Ο διευθύνων σύμβουλος Έρικ Σμιτ αστειευόμενος τον ρώτησε για " τον αποτελεσματικότερο τρόπο να ταξινομηθούν ένα εκατομμύριο ακέραιοι των 32-bit". Ο Ομπάμα είχε πληροφορηθεί προφανώς, αφού απάντησε αμέσως : " Νομίζω ότι η ταξινόμηση φυσαλίδας θα ήταν λανθασμένη επιλογή ". Δείτε το βίντεο στο youtube: `http://www.youtube.com/watch?v=k4RRi_ntQc8`.

Αυτό είναι αλήθεια : η ταξινόμηση φυσαλίδας είναι εννοιολογικά απλή αλλά αργή για μεγάλα σύνολα δεδομένων. Η απάντηση που αναζητούσε ο Σμιτ ήταν μάλλον η ταξινόμηση βάσης ("radix sort" `http://en.wikipedia.org/wiki/Radix_sort`) [1].

Στόχος της ανάλυσης αλγορίθμων είναι να κάνει έγκυρες συγκρίσεις μεταξύ αλγορίθμων, αλλά υπάρχουν μερικά θέματα :

- Η σχετική απόδοση των αλγορίθμων μπορεί να εξαρτάται από τα χαρακτηριστικά του υλικού του υπολογιστή, έτσι ένας αλγόριθμος μπορεί να είναι γρηγορότερος σε μία μηχανή Α και ένας άλλος μπορεί να είναι γρηγορότερος σε μια μηχανή Β. Η γενική λύση σε αυτό το πρόβλημα είναι να ορίσετε ένα μοντέλο μηχανής και να αναλύσετε τον αριθμό των βημάτων ή των ενεργειών που απαιτεί ένας αλγόριθμος σύμφωνα με ένα δοσμένο μοντέλο.
- Η σχετική απόδοση μπορεί να εξαρτάται από τις λεπτομέρειες του συνόλου δεδομένων. Για παράδειγμα, μερικοί αλγόριθμοι τρέχουν γρηγορότερα αν τα δεδομένα είναι μερικώς ταξινο-

[1] Αλλά αν δεχτείτε ερώτηση σαν αυτή σε μία συνέντευξη, θεωρώ ότι αυτή θα ήταν καλύτερη απάντηση : " Ο γρηγορότερος τρόπος για να ταξινομήσω ένα εκατομμύριο ακεραίους είναι να χρησιμοποιήσω οποιαδήποτε συνάρτηση παρέχεται από τη γλώσσα που χρησιμοποιώ. Η απόδοσή της είναι αρκετά καλή για τη συντριπτική πλειοψηφία των εφαρμογών, αλλά αν αποδειχθεί ότι η εφαρμογή μου είναι πολύ αργή, θα χρησιμοποιούσα ένα profiler για να δω που δαπανάται ο χρόνος. Αν δω ότι ένας γρηγορότερος αλγόριθμος ταξινόμησης έχει σημαντικό αντίκτυπο στην απόδοση, τότε θα ψάξω για μία καλή υλοποίηση της ταξινόμησης βάσης. "

μημένα αλλά μερικοί άλλοι τρέχουν πιο αργά σε αυτήν την περίπτωση. Ένας συνηθισμένος τρόπος αποφυγής αυτού του προβλήματος είναι η ανάλυση της χειρότερης περίπτωσης. Μερικές φορές είναι χρήσιμο να αναλύσετε την απόδοση της μέσης περίπτωσης, αλλά αυτό είναι συνήθως δυσκολότερο και μπορεί να μην είναι προφανές ποια σύνολα περιπτώσεων πρέπει να υπολογίσετε.

- Η σχετική απόδοση εξαρτάται επίσης από το μέγεθος του προβλήματος. Ένας αλγόριθμος ταξινόμησης που είναι γρήγορος για μικρές λίστες μπορεί να είναι αργός για μεγάλες λίστες. Η πιο συνηθισμένη λύση σε αυτό το πρόβλημα είναι να εκφράσετε το χρόνο εκτέλεσης (ή τον αριθμό των πράξεων) σαν συνάρτηση του μεγέθους του προβλήματος και να συγκρίνετε (ασυμπτωτικά) τις συναρτήσεις όσο αυξάνεται το μέγεθος του προβλήματος.

Το καλό με αυτού του είδους την σύγκριση είναι ότι προσφέρεται για μία απλή ταξινόμηση των αλγορίθμων. Για παράδειγμα, αν ξέρω ότι ο χρόνος εκτέλεσης του αλγόριθμου Α τείνει να είναι ανάλογος του μεγέθους της εισόδου n, και ο αλγόριθμος Β τείνει να είναι να είναι ανάλογος του n^2, τότε περιμένω ότι ο Α θα είναι γρηγορότερος από τον Β για μεγάλες τιμές του n.

Αυτού του είδους η ανάλυση έχει κάποιους περιορισμούς, αλλά θα φτάσουμε σε αυτό αργότερα.

Β΄.1 Τάξη αύξησης

Υποθέστε ότι έχετε αναλύσει δύο αλγόριθμους και ότι έχετε εκφράσει τους χρόνους εκτέλεσής σε σχέση με το μέγεθος της εισόδου : Ο αλγόριθμος Α χρειάζεται $100n + 1$ βήματα για να λύσει ένα πρόβλημα μεγέθους n, ενώ ο αλγόριθμος Β χρειάζεται $n^2 + n + 1$ βήματα.

ο παρακάτω πίνακας δείχνει τους χρόνους εκτέλεσης αυτών των αλγορίθμων για διαφορετικά μεγέθη προβλήματος :

Input size	Run time of Algorithm A	Run time of Algorithm B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

Για $n = 10$, ο αλγόριθμος Α δεν είναι και πολύ αποδοτικός αφού χρειάζεται σχεδόν 10 φορές τον χρόνο του Β. Αλλά για $n = 100$ είναι περίπου το ίδιο γρήγοροι και για μεγαλύτερες τιμές ο Α είναι πολύ καλύτερος.

Αυτό οφείλεται κυρίως στο ότι για μεγάλες τιμές του n, κάθε συνάρτηση που περιέχει έναν όρο n^2 θα αυξηθεί γρηγορότερα από μία συνάρτηση που ο κύριος όρος είναι n. Ο κύριος όρος είναι αυτός με τον μεγαλύτερο εκθέτη.

Για τον αλγόριθμο Α, ο κύριος όρος έχει μεγάλο συντελεστή (100) και για αυτόν το λόγο ο Β είναι καλύτερος από τον Α για μικρό n. Αλλά ανεξαρτήτως από τους συντελεστές, υπάρχει πάντα μία τιμή του n όπου $an^2 > bn$.

Η ίδια λογική ισχύει και για τους μη-κύριους όρους. Ακόμα και αν ο χρόνος εκτέλεσης του αλγόριθμου Α ήταν $n + 1000000$, θα ήταν και πάλι καλύτερος από τον αλγόριθμο Β για αρκετά μεγάλο n.

Σε γενικές γραμμές, περιμένουμε ότι ένας αλγόριθμος με μικρό κύριο όρο θα είναι καλύτερος για μεγάλα προβλήματα, αλλά για μικρότερα προβλήματα μπορεί να υπάρχει κάποιο σημείο διασταύρωσης όπου ένας άλλος αλγόριθμος είναι καλύτερος. Το σημείο διασταύρωσης εξαρτάται από τις

λεπτομέρειες των αλγορίθμων, τις εισόδους και το υλικό. Για αυτόν το λόγο συνήθως δεν λαμβάνεται υπόψιν στην ανάλυση αλγορίθμων αλλά αυτό δεν σημαίνει ότι πρέπει να το ξεχάσετε.

Αν δύο αλγόριθμοι έχουν τον ίδιο κύριο όρο τότε είναι δύσκολο να αποφανθούμε για το ποιος είναι ο καλύτερος. Και πάλι η απάντηση βρίσκεται στις λεπτομέρειες. Επομένως, οι συναρτήσεις με τον ίδιο κύριο όρο θεωρούνται ισοδύναμες, ακόμα και αν έχουν διαφορετικούς συντελεστές.

Μία τάξη αύξησης είναι ένα σύνολο από συναρτήσεις των οποίων η ασυμπτωτική συμπεριφορά αύξησης θεωρείται ισοδύναμη. Για παράδειγμα, οι $2n$, $100n$ και $n + 1$ ανήκουν στην ίδια τάξη αύξησης, η οποία γράφεται $O(n)$ στον συμβολισμό του μεγάλου Ο και συχνά αποκαλείται γραμμικό επειδή κάθε συνάρτηση του συνόλου αυξάνεται γραμμικά με το n.

Όλες οι συναρτήσεις με κύριο όρο n^2 ανήκουν στο $O(n^2)$ και άρα είναι τετραγωνικές.

Ο ακόλουθος πίνακας δείχνει κάποιες από τις τάξεις αύξησης οι οποίες εμφανίζονται συνήθως στην ανάλυση αλγορίθμων, σε αύξουσα σειρά προς την χειρότερη.

Order of growth	Name
$O(1)$	constant
$O(\log_b n)$	logarithmic (for any b)
$O(n)$	linear
$O(n \log_b n)$	“en log en”
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(c^n)$	exponential (for any c)

Για τους λογαριθμικούς όρους, η βάση του λογαρίθμου δεν έχει σημασία αφού η αλλαγή βάσεων είναι το αντίστοιχο με τον πολλαπλασιασμό με μια σταθερά, η οποία δεν αλλάζει την τάξη αύξησης. Παρομοίως, όλες οι εκθετικές συναρτήσεις ανήκουν στην ίδια τάξη ανάπτυξης ανεξαρτήτως από τη βάση του εκθέτη. Οι εκθετικές συναρτήσεις αυξάνονται πολύ γρήγορα και έτσι οι εκθετικοί αλγόριθμοι χρησιμοποιούνται μόνο για μικρά προβλήματα.

Άσκηση Β΄.1. *Διαβάστε στην Wikipedia σχετικά με το συμβολισμό του μεγάλου Ο (`http://en.wikipedia.org/wiki/Big_O_notation`) και απαντήστε τις ακόλουθες ερωτήσεις :*

1. *Ποια είναι η τάξη αύξησης των συναρτήσεων $n^3 + n^2$, $1000000n^3 + n^2$ και $n^3 + 1000000n^2$;*
2. *Ποια είναι η τάξη αύξησης της $(n^2+n)\cdot(n+1)$; Πριν αρχίσετε να πολλαπλασιάζετε, θυμηθείτε ότι χρειάζεστε μόνο τον κύριο όρο.*
3. *Αν η f ανήκει στην $O(g)$ για μία απροσδιόριστη συνάρτηση g, τι μπορούμε να πούμε για την $af + b$;*
4. *Αν οι f_1 και f_2 ανήκουν στην $O(g)$, τι μπορούμε να πούμε για την $f_1 + f_2$;*
5. *Αν η f_1 ανήκει στην $O(g)$ και η f_2 ανήκει στην $O(h)$, τι μπορούμε να πούμε για την $f_1 + f_2$;*
6. *Αν η f_1 ανήκει στην $O(g)$ και η f_2 ανήκει στην $O(h)$, τι μπορούμε να πούμε για την $f_1 \cdot f_2$;*

Οι προγραμματιστές που νοιάζονται για την απόδοση συχνά δεν αποδέχονται αυτό το είδος ανάλυσης. Και έχουν κάποιο δίκιο : μερικές φορές οι συντελεστές και οι μη-κύριοι όροι μπορεί να κάνουν τη διαφορά. Κάποιες λεπτομέρειες του υλικού, της γλώσσας προγραμματισμού και τα χαρακτηριστικά της εισόδου μπορεί να έχουν σημαντική επίδραση στην απόδοση. Και για μικρά προβλήματα η ασυμπτωτική συμπεριφορά είναι άνευ σημασίας.

Αλλά αν έχετε αυτούς τους περιορισμούς κατά νου τότε η αλγοριθμική ανάλυση είναι ένα χρήσιμο εργαλείο. Τουλάχιστον για μεγάλα προβλήματα, οι ” καλύτεροι ” αλγόριθμοι είναι συνήθως καλύτεροι και μερικές φορές είναι πολύ καλύτεροι. Η διαφορά μεταξύ δύο αλγορίθμων της ίδιας τάξης

αύξησης είναι συνήθως ένας σταθερός όρος, αλλά η διαφορά μεταξύ ενός καλού αλγόριθμου και ενός κακού είναι τεράστια !

Β΄.2 Ανάλυση των βασικών πράξεων της γλώσσας

Οι περισσότερες μαθηματικές πράξεις είναι σταθερού χρόνου. Ο πολλαπλασιασμός συνήθως χρειάζεται περισσότερο χρόνο από την πρόσθεση και την αφαίρεση και η διαίρεση χρειάζεται ακόμα περισσότερο, αλλά αυτοί οι χρόνοι εκτέλεσης δεν εξαρτώνται από τα μεγέθη των τελεστέων. Εξαίρεση αποτελούν οι πολύ μεγάλοι ακέραιοι γιατί ο χρόνος εκτέλεσης αυξάνεται με τον αριθμό των ψηφίων.

Οι πράξεις ευρετηρίασης (το γράψιμο και το διάβασμα στοιχείων των στοιχείων μίας ακολουθίας ή ενός λεξικού) είναι επίσης σταθερού χρόνου, ανεξαρτήτως του μεγέθους της δομής δεδομένων.

Ένας βρόχος `for` που διασχίζει μία ακολουθία ή ένα λεξικό είναι συνήθως γραμμικός, όσο όλες οι πράξεις στο σώμα του βρόχου είναι σταθερού χρόνου. Για παράδειγμα, η προσθήκη στοιχείων σε μια λίστα είναι γραμμική :

```
total = 0
for x in t:
    total += x
```

Η ενσωματωμένη συνάρτηση `sum` είναι επίσης γραμμική επειδή κάνει ακριβώς το ίδιο πράγμα, αλλά τείνει να είναι γρηγορότερη αφού έχει πιο αποτελεσματική υλοποίηση. Στην γλώσσα της ανάλυσης αλγορίθμων θα λέγαμε ότι έχει έναν μικρότερο κύριο συντελεστή.

Αν χρησιμοποιήσετε τον ίδιο βρόχο για να προσθέσετε λίστες συμβολοσειρών, ο χρόνος εκτέλεσης γίνεται τετραγωνικός αφού και η συνένωση συμβολοσειρών είναι γραμμική.

Η μέθοδος συμβολοσειρών `join` είναι συνήθως γρηγορότερη επειδή είναι γραμμική σε όλο το μήκος των συμβολοσειρών.

Κατά κανόνα, αν το σώμα του βρόχου ανήκει στην $O(n^a)$ τότε ολόκληρος ο βρόχος ανήκει στην $O(n^{a+1})$. Η μοναδική εξαίρεση είναι αν μπορείτε να δείξετε ότι ο βρόχος τερματίζει μετά από έναν σταθερό αριθμό επαναλήψεων. Αν ο βρόχος τρέχει k φορές ανεξαρτήτως του n, τότε ο βρόχος ανήκει στην $O(n^a)$, ακόμα και για μεγάλα k.

Ο πολλαπλασιασμός με k δεν αλλάζει την τάξη αύξησης, το ίδιο και η διαίρεση. Επομένως, αν το σώμα ενός βρόχου ανήκει στην $O(n^a)$ και τρέχει n/k φορές, ο βρόχος ανήκει στην $O(n^{a+1})$, ακόμα και μεγάλα k.

Οι περισσότερες πράξεις στις πλειάδες και στις συμβολοσειρές είναι γραμμικές, εκτός από την ευρετηρίαση και την `len`, οι οποίες είναι σταθερού χρόνου. Οι ενσωματωμένες συναρτήσεις `min` και `max` είναι γραμμικές. Ο χρόνος εκτέλεσης μίας πράξης τεμαχισμού είναι ανάλογος του μήκους της εισόδου, αλλά ανεξάρτητος από το μέγεθος της εισόδου.

Όλες οι μέθοδοι συμβολοσειρών είναι γραμμικές, αλλά αν τα μήκη των συμβολοσειρών οριοθετούνται από μία σταθερά (για παράδειγμα πράξεις σε μονούς χαρακτήρες) θεωρούνται σταθερού χρόνου.

Οι περισσότερες μέθοδοι των λιστών είναι γραμμικές, αλλά υπάρχουν μερικές εξαιρέσεις :

- Η πρόσθεση ενός στοιχείου στο τέλος μίας λίστας είναι σταθερού χρόνου κατά μέσο όρο. Όταν μένει από χώρο γράφετε περιστασιακά σε μία μεγαλύτερη θέση, αλλά ο συνολικός χρόνος για n πράξεις είναι $O(n)$, άρα λέμε ότι χρόνος " απόσβεσης " για μία πράξη είναι $O(1)$.

- Η αφαίρεση ενός στοιχείου από το τέλος μίας λίστας είναι σταθερού χρόνου.
- Η ταξινόμηση είναι $O(n \log n)$.

Οι περισσότερες πράξεις και μέθοδοι στα λεξικά είναι σταθερού χρόνου, αλλά υπάρχουν μερικές εξαιρέσεις :

- Ο χρόνος εκτέλεσης της `copy` είναι ανάλογος του πλήθους των στοιχείων, αλλά όχι του μεγέθους των στοιχείων (αντιγράφει μόνο αναφορές, όχι τα στοιχεία αυτά καθαυτά).
- Ο χρόνος εκτέλεσης της `update` είναι ανάλογος του μεγέθους του λεξικού που περνιέται σαν παράμετρος, όχι του λεξικού που ενημερώνεται.
- Οι `keys`, `values` και `items` είναι γραμμικές επειδή επιστρέφουν νέες λίστες. Οι `iterkeys`, `itervalues` και `iteritems` είναι σταθερού χρόνου επειδή επιστρέφουν επαναλήπτες. Αλλά αν χρησιμοποιήσετε βρόχο για τους επαναλήπτες τότε αυτός θα είναι γραμμικός. Η χρήση συναρτήσεων επανάληψης μπορεί να σας εξοικονομήσει λίγο χρόνο, αλλά δεν αλλάζει την τάξη αύξησης εκτός και αν ο αριθμός των στοιχείων που προσπελάζετε είναι οριοθετημένος.

Η απόδοση των λεξικών είναι ένα από τα ελάχιστα θαύματα της επιστήμης των υπολογιστών. Θα δούμε πως δουλεύουν στην Ενότητα Β΄.4.

Άσκηση Β΄.2. *Διαβάστε στην Wikipedia για τους αλγόριθμους ταξινόμησης (`http://en.wikipedia.org/wiki/Sorting_algorithm`) και απαντήστε τις ακόλουθες ερωτήσεις :*

1. *Τι είναι μία ταξινόμηση σύγκρισης ; Ποια είναι η καλύτερη τάξη αύξησης στη χειρότερη περίπτωση για μία comparison sort; Ποια είναι η καλύτερη τάξη αύξησης στην χειρότερη περίπτωση για οποιονδήποτε αλγόριθμο ταξινόμησης ;*
2. *Ποια είναι η τάξη αύξησης της ταξινόμησης φυσαλίδας και γιατί ο Μπαράκ Ομπάμα θεωρεί ότι " δεν είναι σωστός τρόπος ";*
3. *Ποια είναι η τάξη αύξησης για την ταξινόμηση βάσης ; Ποιες προϋποθέσεις πρέπει να πληρούνται για να μπορέσουμε να την χρησιμοποιήσουμε ;*
4. *Τί είναι μία σταθερή ταξινόμηση και γιατί θα μπορούσε να έχει σημασία στην πράξη ;*
5. *Ποιος είναι ο χειρότερος αλγόριθμος ταξινόμησης (ο οποίος έχει όνομα) ;*
6. *Ποιόν αλγόριθμο ταξινόμησης χρησιμοποιεί η βιβλιοθήκη της C; Ποιόν η Python; Είναι αυτοί οι αλγόριθμοι σταθεροί ; Μπορεί να χρειαστεί να χρησιμοποιήσετε το Google για να βρείτε αυτές τις απαντήσεις.*
7. *Πολλές από τις ταξινομήσεις χωρίς σύγκριση είναι γραμμικές. Γιατί η Python χρησιμοποιεί μία $O(n \log n)$ ταξινόμηση σύγκρισης ;*

Β΄.3 Ανάλυση των αλγόριθμων αναζήτησης

Μία αναζήτηση είναι ένας αλγόριθμος ο οποίος παίρνει μία συλλογή και ένα στοιχείο στόχος και προσδιορίζει αν αυτό το στοιχείο υπάρχει μέσα στη συλλογή, επιστρέφοντας συχνά και τον δείκτη του.

Ο απλούστερος αλγόριθμος αναζήτησης είναι η " γραμμική αναζήτηση ", η οποία διασχίζει τα στοιχεία της συλλογής με τη σειρά και σταματάει αν βρει τον στόχο. Στην χειρότερη περίπτωση θα πρέπει να διασχίσει ολόκληρη τη συλλογή και άρα ο χρόνος εκτέλεσης είναι γραμμικός.

Ο τελεστής `in` για τις ακολουθίες χρησιμοποιεί μία γραμμική αναζήτηση. Το ίδιο και οι μέθοδοι `find` και `count`.

Αν τα στοιχεία της ακολουθίας είναι ταξινομημένα, μπορείτε να χρησιμοποιήσετε μία δυαδική αναζήτηση, η οποία είναι της τάξης $O(\log n)$. Η δυαδική αναζήτηση είναι παρόμοια με τον αλγόριθμο που χρησιμοποιείτε πιθανώς για να βρείτε μία λέξη σε ένα λεξικό (σε ένα πραγματικό λεξικό, όχι μία δομή δεδομένων). Αντί να αρχίσετε από την αρχή και να ελέγχετε ένα ένα τα στοιχεία με τη σειρά, ξεκινάτε από τη μέση και ελέγχετε αν η λέξη που ψάχνετε βρίσκεται πριν η μετά. Αν βρίσκεται πριν, τότε την αναζητάτε στο πρώτο μισό της ακολουθίας. Αλλιώς την ψάχνετε στο δεύτερο μισό. Είτε έτσι είτε αλλιώς, σπάτε τον αριθμό των εναπομεινάντων στοιχείων προς αναζήτηση στη μέση.

Αν η ακολουθία έχει 1.000.000 στοιχεία, θα χρειαστούν περίπου 20 βήματα για να βρείτε τη λέξη ή να συμπεράνετε ότι δεν υπάρχει. Επομένως είναι περίπου 50.000 φορές γρηγορότερη από μία γραμμική αναζήτηση.

Άσκηση Β΄.3. *Γράψτε μία συνάρτηση με όνομα* `bisection` *η οποία θα παίρνει μία ταξινομημένη λίστα και μία τιμή στόχο και θα επιστρέφει τον δείκτη της τιμής μέσα στη λίστα, αν υπάρχει, ή* `None` *αν δεν υπάρχει.*

Ή μπορείτε να διαβάσετε την τεκμηρίωση του αρθρώματος `bisect` *και να χρησιμοποιήσετε αυτό!*

Η δυαδική αναζήτηση μπορεί να είναι γρηγορότερη από την γραμμική αλλά απαιτεί να είναι ταξινομημένη η ακολουθία, το οποίο μπορεί να χρειάζεται επιπλέον δουλειά.

Υπάρχει ακόμα μία δομή δεδομένων με όνομα ” πίνακας κατακερματισμού ” η οποία είναι ακόμη γρηγορότερη. Μπορεί να κάνει αναζήτηση σε σταθερό χρόνο και δεν απαιτεί να είναι ταξινομημένα τα στοιχεία. Τα λεξικά της Python είναι υλοποιημένα χρησιμοποιώντας πίνακες κατακερματισμού, γι αυτόν το λόγο οι περισσότερες πράξεις των λεξικών, συμπεριλαμβανομένου του τελεστή `in`, είναι σταθερού χρόνου.

Β΄.4 Πίνακες κατακερματισμού

Για να εξηγήσω πως δουλεύουν οι πίνακες κατακερματισμού και γιατί είναι τόσο καλοί, θα ξεκινήσω με μία απλή υλοποίηση μίας αντιστοίχισης και σταδιακά θα την βελτιώσω μέχρι να γίνει πίνακας κατακερματισμού.

Χρησιμοποιώ την Python για να επιδείξω αυτές τις υλοποιήσεις, αλλά στην πραγματικότητα δεν θα γράφατε ποτέ κώδικα όπως αυτός στην Python, θα χρησιμοποιούσατε απλώς ένα λεξικό ! Άρα για το υπόλοιπο αυτού του κεφαλαίου θα πρέπει να υποθέσετε ότι τα λεξικά δεν υπάρχουν και ότι θέλετε να υλοποιήσετε μία δομή δεδομένων που αντιστοιχεί κλειδιά σε τιμές. Οι πράξεις που πρέπει να υλοποιήσετε είναι :

`add(k, v)`**:** Προσθήκη ενός νέου στοιχείου που αντιστοιχεί ένα κλειδί `k` σε μία τιμή `v`. Με ένα λεξικό `d` της Python, αυτή η πράξη θα γραφόταν έτσι `d[k] = v`.

`get(target)`**:** Αναζήτηση και επιστροφή της τιμής που αντιστοιχεί στο κλειδί `target`. Με ένα λεξικό `d` της Python, αυτή η πράξη θα γραφόταν ή έτσι `d[target]` ή έτσι `d.get(target)`.

Για την ώρα, υποθέτω ότι κάθε κλειδί εμφανίζεται μόνο μία φορά. Η απλούστερη υλοποίηση αυτής της διεπαφής χρησιμοποιεί μία λίστα πλειάδων, όπου κάθε πλειάδα αποτελεί ένα ζευγάρι κλειδιού-τιμής.

```
class LinearMap(object):

    def __init__(self):
        self.items = []

    def add(self, k, v):
```

```
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

Η add επισυνάπτει μία πλειάδα κλειδιού-τιμής στην λίστα στοιχείων, το οποίο χρειάζεται σταθερό χρόνο.

Η get χρησιμοποιεί ένα βρόχο for για να ψάξει τη λίστα : αν βρει το κλειδί στόχο επιστρέφει την αντίστοιχη τιμή, αλλιώς εγείρει μία KeyError. Άρα η get είναι γραμμική.

Μία εναλλακτική λύση είναι να κρατήσουμε τη λίστα ταξινομημένη με βάση τα κλειδιά. Τότε η get θα μπορούσε να χρησιμοποιήσει μία δυαδική αναζήτηση, η οποία ανήκει στην $O(\log n)$. Αλλά η εισαγωγή ενός νέου στοιχείου στη μέση μίας λίστας είναι γραμμική, άρα αυτή ίσως να μην είναι και η καλύτερη επιλογή. Υπάρχουν και άλλες δομές δεδομένων (βλ. http://en.wikipedia.org/wiki/Red-black_tree) οι οποίες μπορούν να υλοποιήσουν την add και την get σε λογαριθμικό χρόνο, αλλά και πάλι δεν είναι το ίδιο με τον σταθερό χρόνο, οπότε ας προχωρήσουμε.

Ένας τρόπος για να βελτιώσουμε την LinearMap είναι να σπάσουμε τη λίστα των ζευγαριών κλειδιού-τιμής σε μικρότερες λίστες. Αυτή είναι μία υλοποίηση με όνομα BetterMap, η οποία αποτελεί μία λίστα από 100 LinearMaps. Όπως θα δούμε αμέσως μετά, η τάξη αύξησης για την get εξακολουθεί να είναι γραμμική, αλλά η BetterMap είναι ένα βήμα προς τους πίνακες κατακερματισμού :

```
class BetterMap(object):

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

Η __init__ φτιάχνει μία λίστα από n LinearMaps.

Η find_map χρησιμοποιείται από την add και την get για να υπολογιστεί ποια αντιστοίχιση θα εισάγει το νέο στοιχείο ή ποια αντιστοίχιση θα αναζητηθεί.

Η find_map χρησιμοποιεί την ενσωματωμένη συνάρτηση hash, η οποία παίρνει σχεδόν ένα οποιοδήποτε αντικείμενο της Python και επιστρέφει έναν ακέραιο. Ο περιορισμός αυτής της υλοποίησης είναι ότι δουλεύει μόνο με κατακερματίσιμα κλειδιά. Οι μεταβλητοί τύποι όπως οι λίστες και τα λεξικά είναι μη-κατακρματίσιμοι.

Τα κατακερματίσιμα αντικείμενα που θεωρούνται ίσα επιστρέφουν την ίδια τιμή κατακερματισμού,

αλλά το αντίστροφο δεν είναι κατ'ανάγκη αληθές : δύο διαφορετικά αντικείμενα μπορούν να επιστρέψουν την ίδια τιμή κατακερματισμού.

Η `find_map` χρησιμοποιεί τον τελεστή ακέραιας διαίρεσης για να ορίσει τις τιμές κατακερματισμού στο εύρος από 0 μέχρι `len(self.maps)`, έτσι ώστε το αποτέλεσμα να είναι ένας έγκυρος δείκτης της λίστας. Φυσικά, αυτό σημαίνει ότι πολλές τιμές κατακερματισμού θα οριστούν στον ίδιο δείκτη. Αλλά αν η συνάρτηση κατακερματισμού εξαπλωθεί ομοιόμορφα (το οποίο είναι αυτό που είναι σχεδιασμένες να κάνουν οι συναρτήσεις κατακερματισμού), τότε υπολογίζουμε $n/100$ στοιχεία ανά LinearMap.

Από τη στιγμή που ο χρόνος εκτέλεσης της `LinearMap.get` είναι ανάλογος του πλήθους των στοιχείων, περιμένουμε ότι η BetterMap θα είναι 100 φορές γρηγορότερη από την LinearMap. Η τάξη αύξησης είναι ακόμη γραμμική αλλά ο κύριος συντελεστής είναι μικρότερος. Αυτό είναι ωραίο αλλά όχι τόσο καλό όσο ένας πίνακας κατακερματισμού.

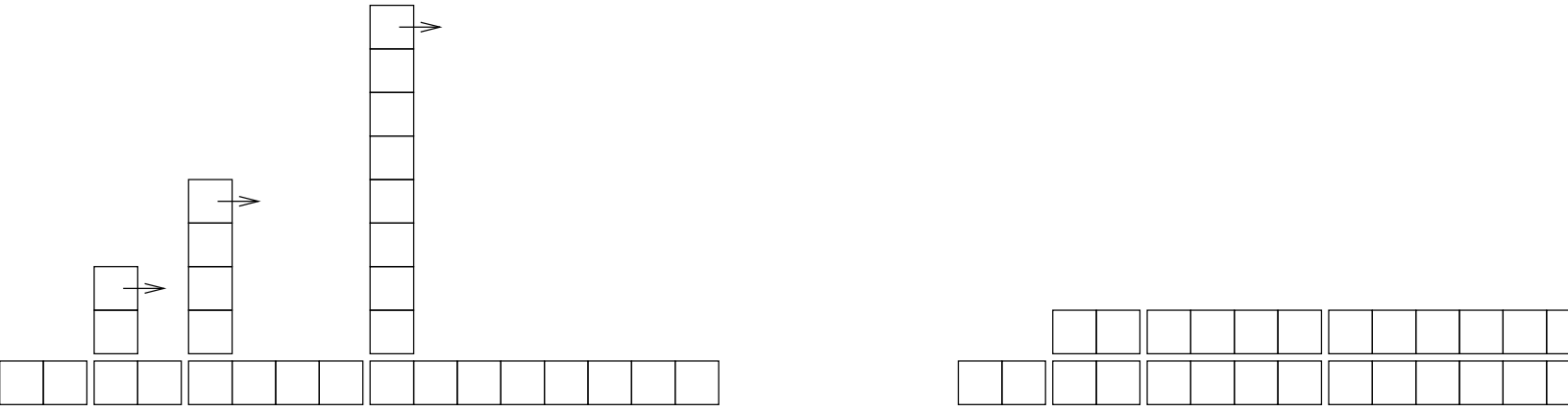
Και εδώ έρχεται (επιτέλους) η βασική ιδέα που κάνει του πίνακες κατακερματισμού γρήγορους : αν μπορείτε να κρατήσετε εντός ορίων το μήκος της LinearMaps, τότε η `LinearMap.get` γίνεται σταθερού χρόνου. Το μόνο που έχετε να κάνετε είναι να παρακολουθείτε το πλήθος των στοιχείων και όταν το πλήθος αν LinearMap υπερβεί ένα κατώτατο όριο τότε αλλάζετε το μέγεθος του πίνακα κατακερματισμού προσθέτοντας περισσότερες LinearMaps.

Αυτή είναι μία υλοποίηση ενός πίνακα κατακερματισμού :

```
class HashMap(object):

    def __init__(self):
        self.maps = BetterMap(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 resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

Κάθε `HashMap` περιέχει μία `BetterMap`. Η `__init__` αρχίζει με μόλις 2 LinearMaps και αρχικοποιεί την `num`, η οποία θα κρατάει τον αριθμό των στοιχείων.

Η `get` απλά στέλνει στην `BetterMap`. Όλη η δουλειά γίνεται στην `add`, η οποία ελέγχει τον αριθμό των στοιχείων και το μέγεθος της `BetterMap`: αν είναι ίσα τότε ο μέσος όρος των στοιχείων για κάθε LinearMap είναι 1, και έτσι καλεί την `resize`.



Σχήμα Β΄.1: Το κόστος μίας προσθήκης πίνακα κατακερματισμού.

Η `resize` φτιάχνει μία νέα `BetterMap`, δύο φορές μεγαλύτερη από την προηγούμενη, και στη συνέχεια " επανακατακερματίζει " τα στοιχεία από την παλιά στη καινούρια.

Ο επανακατακερματισμός είναι απαραίτητος επειδή αλλάζοντας τον αριθμό των LinearMaps αλλάζει και ο παρονομαστής του τελεστή ακέραιας διαίρεσης στην `find_map`. Αυτό σημαίνει ότι κάποια αντικείμενα τα οποία οριζόταν μέσα στην ίδια LinearMap θα χωριστούν (αυτό που θέλαμε, σωστά ;).

Ο επανακατακερματισμός είναι γραμμικός, επομένως και η `resize` είναι γραμμική, το οποίο μπορεί να φαίνεται κακό αφού υποσχέθηκα ότι η `add` θα είναι σταθερού χρόνου. Αλλά θυμηθείτε ότι δεν χρειάζεται να αλλάζουμε το μέγεθος κάθε φορά, άρα η `add` θα είναι συνήθως σταθερού χρόνου και μόνο περιστασιακά γραμμική. Το σύνολο του έργου που χρειάζεται για να τρέξουμε την `add` n είναι ανάλογο του n, άρα ο μέσος χρόνος για κάθε `add` είναι σταθερός !

Για να καταλάβετε πως δουλεύει, σκεφτείτε ότι αρχίζουμε με ένα κενό HashTable και προσθέτουμε μία ακολουθία στοιχείων. Ξεκινάμε με 2 LinearMaps, επομένως οι δύο πρώτες προσθήκες είναι γρήγορες (δεν απαιτείται αλλαγή μεγέθους). Ας πούμε ότι χρειάζονται μία μονάδα έργου η κάθε μία. Η επόμενη προσθήκη χρειάζεται μία αλλαγή μεγέθους και άρα πρέπει να επανακατακερματίσουμε τα δύο πρώτα στοιχεία (ας πούμε ότι αυτό είναι 2 επιπλέον μονάδες έργου) και στη συνέχεια προσθέτουμε το τρίτο στοιχείο (μία ακόμα μονάδα). Η προσθήκη του επόμενου στοιχείου μας στοιχίζει 1 μονάδα και έτσι το σύνολο μέχρι στιγμής είναι 6 μονάδες έργου για 4 στοιχεία.

Η επόμενη `add` κοστίζει 5 μονάδες, αλλά οι επόμενες τρεις είναι μόνο μία μονάδα η κάθε μία, άρα το σύνολο γίνεται 14 μονάδες για τις πρώτες 8 προσθήκες.

Η επόμενη `add` κοστίζει 9 μονάδες, αλλά τότε μπορούμε να προσθέσουμε ακόμα 7 πριν την επόμενη αλλαγή μεγέθους, έτσι το σύνολο είναι 30 μονάδες για τις πρώτες 16 προσθήκες.

Μετά από 32 προσθήκες, το συνολικό κόστος θα είναι 62 μονάδες έργου και ελπίζω ότι αρχίσατε να βλέπετε το μοτίβο. Μετά από n προσθήκες, όπου n είναι μία δύναμη του δύο, το συνολικό κόστος θα είναι $2n - 2$ μονάδες. Άρα το μέσο έργο για κάθε προσθήκη είναι λίγο λιγότερο από 2 μονάδες. Αυτή είναι η βέλτιστη περίπτωση, όταν το n είναι μία δύναμη του δύο. Για άλλες τιμές του n το μέσο έργο είναι λίγο περισσότερο αλλά αυτό είναι μικρής σημασίας. Το σημαντικό είναι ότι είναι $O(1)$.

Η Εικόνα Β΄.1 δείχνει πως δουλεύει όλο αυτό γραφικά. Κάθε τετράγωνο αναπαριστά μία μονάδα έργου. Οι στήλες δείχνουν το συνολικό έργο για κάθε προσθήκη αρχίζοντας από τα αριστερά προς τα δεξιά : οι πρώτες δύο `adds` κοστίζουν 1 μονάδα, η τρίτη 3 μονάδες, κτλ.

Το επιπλέον έργο για τον επανακατακερματισμό εμφανίζεται σαν μία ακολουθία από όλο και υψηλότερους πύργους με όλο και μεγαλύτερα διαστήματα μεταξύ τους. Τώρα, αν γκρεμίσετε τους πύργους, μοιράζοντας το κόστος σε όλες τις προσθήκες, θα δείτε γραφικά ότι το συνολικό κόστος μετά από n προσθήκες είναι $2n - 2$.

Ένα σημαντικό χαρακτηριστικό αυτού του αλγορίθμου είναι ότι όταν αλλάζουμε το μέγεθος του πίνακα κατακερματισμού τότε αυξάνεται γεωμετρικά. Ήτοι πολλαπλασιάζουμε το μέγεθος με μια σταθερά. Αν αυξήσετε το μέγεθος αριθμητικώς (προσθέτοντας ένα συγκεκριμένο αριθμό κάθε φορά), ο μέσος χρόνος για κάθε `add` είναι γραμμικός.

Μπορείτε να κατεβάσετε την υλοποίηση μου για την HashMap από εδώ `http://thinkpython/code/Map.py`, αλλά θυμηθείτε ότι δεν υπάρχει λόγος να την χρησιμοποιήσετε. Αν θέλετε να αντιστοιχίσετε τότε μπορείτε απλά να χρησιμοποιήσετε ένα λεξικό της Python.

Παράρτημα Γ΄

Lumpy

Καθ'όλη τη διάρκεια του βιβλίου, έχω χρησιμοποιήσει διαγράμματα για να αναπαραστήσω την κατάσταση των προγραμμάτων που εκτελούνται.

Στην Ενότητα 2.2, χρησιμοποιήσαμε ένα διάγραμμα κατάστασης για να καταδείξουμε τα ονόματα και τις τιμές των μεταβλητών. Στην Ενότητα 3.10 παρουσίασα ένα διάγραμμα κατάστασης στο οποίο κάθε κλήση συνάρτησης αναπαριστάται από ένα πλαίσιο. Κάθε πλαίσιο δείχνει τις παραμέτρους και τις τοπικές μεταβλητές για κάθε συνάρτηση ή μέθοδο. Τα διαγράμματα κατάστασης για τις αναδρομικές συναρτήσεις φαίνονται στην Ενότητα 5.9 και στην Ενότητα 6.5.

Η Ενότητα 10.2 δείχνει πως φαίνεται μία λίστα σε ένα διάγραμμα κατάστασης, η Ενότητα 11.4 δείχνει πως φαίνεται ένα λεξικό και η Ενότητα 12.6 δείχνει δύο τρόπους για την αναπαράσταση των πλειάδων.

Στην Ενότητα 15.2 παρουσιάζονται τα διαγράμματα αντικειμένων, τα οποία δείχνουν την κατάσταση των ιδιοτήτων ενός αντικειμένου, των ιδιοτήτων αυτών των ιδιοτήτων και ούτω καθεξής. Η Ενότητα 15.3 έχει διαγράμματα αντικειμένων για τα ορθογώνια (Rectangles) και τα ενσωματωμένα σημεία τους Points. Η Ενότητα 16.1 δείχνει την κατάσταση ενός αντικειμένου ώρας (Time). Η Ενότητα 18.2 έχει ένα διάγραμμα το οποίο περιέχει ένα αντικείμενο κλάσης και ένα στιγμιότυπο, τα καθένα με τις ιδιότητές του.

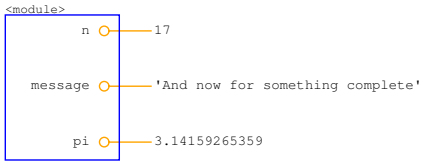
Τέλος, στην Ενότητα 18.8 παρουσιάζονται τα διαγράμματα κλάσεων, τα οποία δείχνουν τις κλάσεις που συνθέτουν ένα πρόγραμμα και τις σχέσεις μεταξύ τους.

Αυτά τα διαγράμματα βασίζονται στην UML (Unified Modeling Language), η οποία αποτελεί μία πρότυπη γλώσσα γραφικής απεικόνισης. Η UML χρησιμοποιείται από τους μηχανικούς λογισμικού για να συνεννοούνται σχετικά με το σχεδιασμό του προγράμματος, ειδικά για τα αντικειμενοστραφή προγράμματα.

Είναι μία πλούσια γλώσσα με πολλά είδη διαγραμμάτων τα οποία αναπαριστούν σχέσεις μεταξύ αντικειμένων και κλάσεων. Σε αυτό το βιβλίο παρουσίασα μόνο ένα μικρό υποσύνολο της γλώσσας, αλλά αυτό είναι και το υποσύνολο που χρησιμοποιείται συνήθως στην πράξη.

Σκοπός αυτού του παραρτήματος είναι η ανασκόπηση των διαγραμμάτων που παρουσιάστηκαν στα προηγούμενα κεφάλαια και η εισαγωγή στη Lumpy. Η Lumpy, η οποία σημαίνει "UML in Python" με λίγο αναγραμματισμό, είναι μέρος του Swampy, το οποίο έχετε ήδη εγκαταστήσει αν δουλέψατε στην μελέτη περίπτωσης του Κεφαλαίου 4 ή του Κεφαλαίου 19, ή αν κάνατε την Άσκηση 15.4.

Η Lumpy χρησιμοποιεί το άρθρωμα `inspect` της Python για να εξετάσει την κατάσταση ενός προ-



Σχήμα Γ΄.1: Διάγραμμα κατάστασης που δημιουργείται από την Lumpy.

γράμματος που εκτελείται και να παράξει διαγράμματα αντικειμένων (συμπεριλαμβανομένων των διαγραμμάτων κατάστασης) και διαγράμματα κλάσεων.

Γ΄.1 Διάγραμμα κατάστασης

Αυτό είναι ένα παράδειγμα το οποίο χρησιμοποιεί την Lumpy για να δημιουργήσει ένα διάγραμμα κατάστασης.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

message = 'And now for something completely different'
n = 17
pi = 3.1415926535897932

lumpy.object_diagram()
```

Η πρώτη γραμμή εισάγει την κλάση Lumpy από το `swampy.Lumpy`. Αν δεν έχετε εγκαταστήσει το Swampy σαν πακέτο, σιγουρευτείτε ότι τα αρχεία του Swampy βρίσκονται στην διαδρομή αναζήτησης της Python και χρησιμοποιήστε αυτή τη δήλωση `import`:

```
from Lumpy import Lumpy
```

Οι επόμενες γραμμές δημιουργούν ένα αντικείμενο `Lumpy` και φτιάχνουν ένα σημείο αναφοράς, το οποίο σημαίνει ότι η Lumpy καταγράφει τα αντικείμενα που έχουν οριστεί μέχρι στιγμής.

Στη συνέχεια ορίζουμε νέες μεταβλητές και επικαλούμαστε την `object_diagram`, η οποία σχεδιάζει τα αντικείμενα που έχουν οριστεί από το σημείο αναφοράς, σε αυτήν την περίπτωση τα `message`, `n` και `pi`.

Η Εικόνα Γ΄.1 δείχνει το αποτέλεσμα. Τα γραφικά είναι διαφορετικά από αυτά που έδειξα νωρίτερα. Για παράδειγμα, κάθε αναφορά αναπαριστάται από έναν κύκλο δίπλα στο όνομα της μεταβλητής και μία γραμμή προς στην τιμή. Και οι μεγάλες συμβολοσειρές είναι περικομμένες. Αλλά η πληροφορία που παρέχει το διάγραμμα είναι η ίδια.

Τα ονόματα των μεταβλητών είναι σε ένα πλαίσιο με ετικέτα `<module>`, το οποίο υποδεικνύει ότι αυτές είναι μεταβλητές του αρθρώματος, γνωστές και ως καθολικές.

Μπορείτε να κατεβάσετε αυτό το παράδειγμα από εδώ : `http://thinkpython.com/code/lumpy_demo1.py`. Δοκιμάστε να προσθέσετε μερικές επιπλέον εκχωρήσεις και δείτε πως θα φαίνεται το διάγραμμα.

Σχήμα Γ΄.2: Διάγραμμα στοίβας.

Γ΄.2 Διάγραμμα στοίβας

Αυτό είναι ένα παράδειγμα που χρησιμοποιεί την Lumpy για να δημιουργήσει ένα διάγραμμα στοίβας. Μπορείτε να το κατεβάσετε από εδώ : `http://thinkpython.com/code/lumpy_demo2.py`.

```
from swampy.Lumpy import Lumpy

def countdown(n):
    if n <= 0:
        print 'Blastoff!'
        lumpy.object_diagram()
    else:
        print n
        countdown(n-1)

lumpy = Lumpy()
lumpy.make_reference()
countdown(3)
```

Η Εικόνα Γ΄.2 δείχνει το αποτέλεσμα. Κάθε πλαίσιο αναπαριστάται από ένα κουτί το οποίο έχει το όνομα της συνάρτησης απέξω και τις μεταβλητές στο εσωτερικό του. Αφού αυτή η συνάρτηση είναι αναδρομική, υπάρχει ένα πλαίσιο για κάθε επίπεδο της αναδρομής.

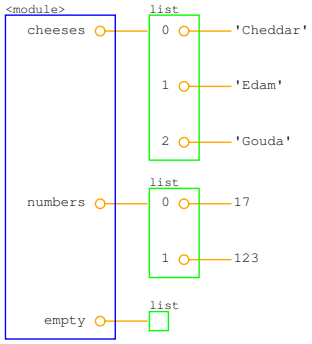
Θυμηθείτε ότι ένα διάγραμμα στοίβας δείχνει την κατάσταση του προγράμματος σε ένα συγκεκριμένο σημείο της εκτέλεσής του. Για να πάρετε το διάγραμμα που θέλετε, μερικές φορές πρέπει να σκεφτείτε που πρέπει να επικαλεστείτε την `object_diagram`.

Σε αυτήν την περίπτωση την επικαλέστηκα μετά την εκτέλεση της περίπτωσης βάσης της αναδρομής. Με αυτόν τον τρόπο το διάγραμμα στοίβας δείχνει όλα τα επίπεδα της αναδρομής. Μπορείτε να καλέσετε την `object_diagram` περισσότερες από μία φορές ούτως ώστε να πάρετε μία σειρά από στιγμιότυπα της εκτέλεσης του προγράμματος.

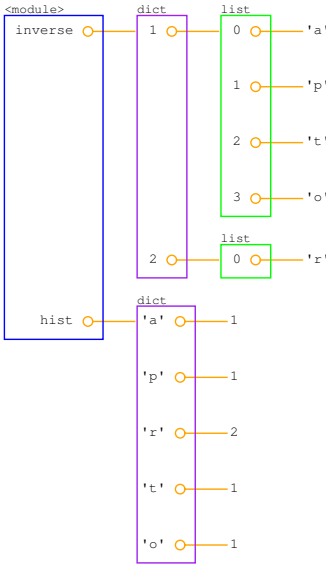
Γ΄.3 Διαγράμματα αντικειμένων

Αυτό το παράδειγμα παράγει ένα διάγραμμα αντικειμένου το οποίο δείχνει τις λίστες της Ενότητας 10.1. Μπορείτε να το κατεβάσετε από εδώ : `http://thinkpython.com/code/lumpy_demo3.py`.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

cheeses = ['Cheddar', 'Edam', 'Gouda']
numbers = [17, 123]
empty = []
```



Σχήμα Γ΄.3: Διάγραμμα αντικειμένων.



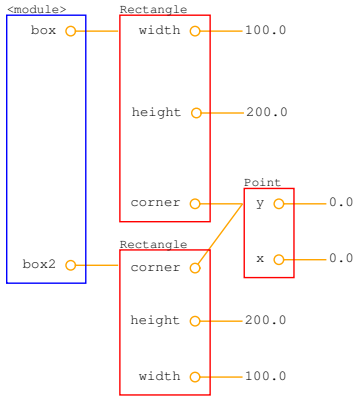
Σχήμα Γ΄.4: Διάγραμμα αντικειμένων.

```
lumpy.object_diagram()
```

Η Εικόνα Γ΄.3 δείχνει το αποτέλεσμα. Οι λίστες αναπαριστώνται από ένα κουτί το οποίο δείχνει τους δείκτες να αντιστοιχίζονται στα στοιχεία. Αυτή η αναπαράσταση είναι ελαφρώς παραπλανητική, αφού οι δείκτες δεν είναι μέρος της λίστας στην πραγματικότητα, αλλά θεωρώ ότι κάνουν ευκολότερη την ανάγνωση του διαγράμματος. Η κενή λίστα αναπαριστάται από ένα άδειο κουτί.

Και αυτό είναι ένα παράδειγμα το οποίο δείχνει τα λεξικά από την Ενότητα 11.4. Μπορείτε να το κατεβάσετε από εδώ : `http://thinkpython.com/code/lumpy_demo4.py`.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()
```



Σχήμα Γ΄.5: Διάγραμμα αντικειμένων.

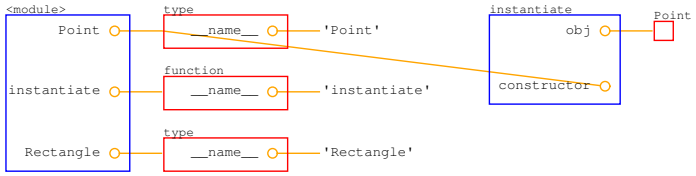
```
hist = histogram('parrot')
inverse = invert_dict(hist)

lumpy.object_diagram()
```

Η Εικόνα Γ΄.4 δείχνει το αποτέλεσμα. Το `hist` είναι ένα λεξικό το οποίο αντιστοιχεί χαρακτήρες (συμβολοσειρές με ένα γράμμα) σε ακέραιους αριθμού. Το `inverse` αντιστοιχεί ακέραιους σε λίστες συμβολοσειρών.

Αυτό το παράδειγμα παράγει ένα διάγραμμα αντικειμένων για αντικείμενα Point και Rectangle, όπως στην Ενότητα 15.6. Μπορείτε να το κατεβάσετε από εδώ : `http://thinkpython.com/code/lumpy_demo5.py`.

```
import copy
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

box2 = copy.copy(box)

lumpy.object_diagram()
```

Η Εικόνα Γ΄.5 δείχνει το αποτέλεσμα. Η `copy.copy` κάνει μία ρηχή αντιγραφή, έτσι τα `box` και `box2` έχουν τα δικά τους `width` και `height`, αλλά μοιράζονται το ίδιο ενσωματωμένο αντικείμενο Point. Αυτού του είδους η κοινή χρήση είναι συνήθως εντάξει με αμετάβλητα αντικείμενα, αλλά είναι πολύ επιρρεπή σε λάθη με μεταβλητούς τύπους.



Σχήμα Γ΄.6: Διάγραμμα αντικειμένων.

Γ΄.4 Αντικείμενα συναρτήσεων και κλάσεων

Συνήθως, όταν χρησιμοποιώ την Lumpy για να φτιάξω διαγράμματα αντικειμένων, ορίζω τις συναρτήσεις και τις κλάσεις προτού φτιάξω το σημείο αναφοράς. Με αυτόν τον τρόπο, τα αντικείμενα των συναρτήσεων και των κλάσεων δεν εμφανίζονται μέσα στο διάγραμμα.

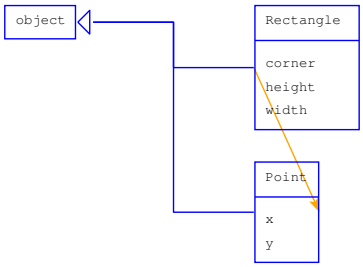
Αλλά αν περάσετε τις συναρτήσεις και τις κλάσεις σαν παραμέτρους, ίσως θέλατε και να εμφανίζονται. Αυτό είναι ένα παράδειγμα το οποίο μπορείτε να κατεβάσετε από εδώ : `http://thinkpython.com/code/lumpy_demo6.py`.

```
import copy
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

class Point(object):
    """Represents a point in 2-D space."""

class Rectangle(object):
    """Represents a rectangle."""

def instantiate(constructor):
    """Instantiates a new object."""
    obj = constructor()
    lumpy.object_diagram()
    return obj

point = instantiate(Point)
```

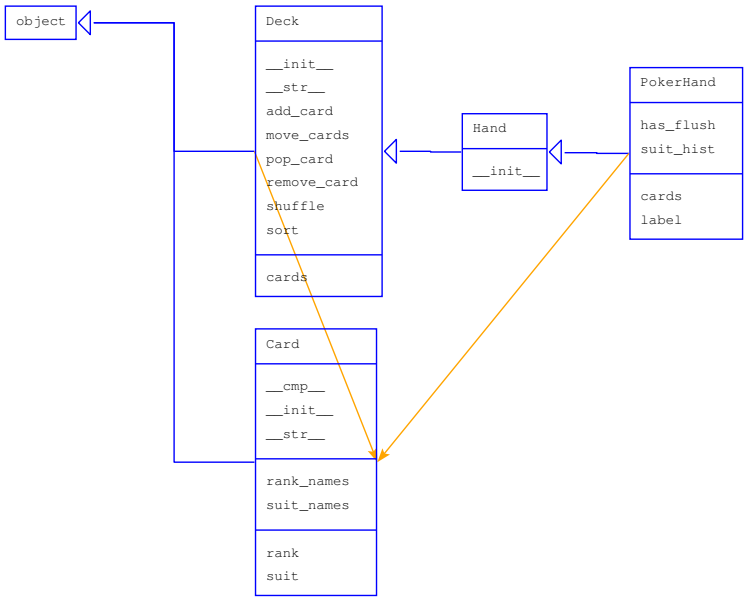
Η Εικόνα Γ΄.6 δείχνει το αποτέλεσμα. Αφού επικαλεστήκαμε την `object_diagram` μέσα σε μία συνάρτηση, θα πάρουμε ένα διάγραμμα στοίβας με ένα πλαίσιο για τις καθολικές μεταβλητές και για τις επικλήσεις της `instantiate`.

Στο επίπεδο αρθρώματος, τα `Point` και `Rectangle` αναφέρονται σε αντικείμενα κλάσεων (τα οποία έχουν τύπο `type`) και το `instantiate` αναφέρεται σε ένα αντικείμενο συνάρτησης.

Αυτό το διάγραμμα μπορεί να αποσαφηνίσει δύο από συνηθέστερα σημεία σύγχυσης : (1) τη διαφορά μεταξύ του αντικειμένου κλάσης, `Point`, και του στιγμιότυπου ενός Point, `obj`, και (2) τη διαφορά μεταξύ του αντικειμένου συνάρτησης που δημιουργήθηκε όταν ορίστηκε η `instantiate` και του πλαισίου που δημιουργήθηκε όταν καλέστηκε αυτή.



Σχήμα Γ΄.7: Διάγραμμα κλάσεων.



Σχήμα Γ΄.8: Διάγραμμα κλάσεων.

Γ΄.5 Διαγράμματα Κλάσεων

Παρά το γεγονός ότι έκανα διάκριση μεταξύ των διαγραμμάτων κατάστασης, στοίβας και αντικειμένων, είναι επί το πλείστον το ίδιο πράγμα : δείχνουν την κατάσταση ενός προγράμματος που εκτελείται σε μία δεδομένη χρονική στιγμή.

Τα διαγράμματα κλάσεων είναι διαφορετικά. Δείχνουν τις κλάσεις που συνθέτουν ένα πρόγραμμα και τις σχέσεις μεταξύ τους. Είναι διαχρονικές υπό την έννοια ότι περιγράφουν το πρόγραμμα στο σύνολο του, όχι σε μία συγκεκριμένη χρονική στιγμή. Για παράδειγμα, αν ένα στιγμιότυπο της κλάσης A γενικά μία αναφορά σε ένα στιγμιότυπο της κλάσης B, τότε λέμε ότι υπάρχει μία σχέση ” EXEI-ENA ” μεταξύ αυτών των κλάσεων.

Αυτό είναι ένα παράδειγμα το οποίο δείχνει μία σχέση EXEI-ENA. Μπορείτε να το κατεβάσετε από εδώ : `http://thinkpython.com/code/lumpy_demo7.py`.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()
```

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

lumpy.class_diagram()
```

Η Εικόνα Γ΄.7 δείχνει το αποτέλεσμα. Κάθε κλάση αναπαριστάται με ένα κουτί το οποίο περιέχει το όνομα της συνάρτησης, όλες μεθόδους που παρέχει η κλάση, όλες τις μεταβλητές της κλάσης και όλα τα μεταβλητές των στιγμιοτύπων, αλλά όχι μεθόδους και μεταβλητές κλάσεων.

Το βέλος από η `Rectangle` στην `Point` δείχνει ότι τα ορθογώνια περιέχουν ένα ενσωματωμένο σημείο. Επιπλέον, τόσο η `Rectangle` όσο και η `Point` κληρονομούν την `object`, η οποία αναπαριστάται στο διάγραμμα από ένα βέλος με τριγωνάκι στην κεφαλή του.

Αυτό είναι ένα πιο σύνθετο παράδειγμα στο οποίο χρησιμοποιώ την λύση μου για την Άσκηση 18.6. Μπορείτε να κατεβάσετε τον κώδικα από εδώ : `http://thinkpython.com/code/lumpy_demo8.py`. Θα χρειαστείτε επίσης και το : `http://thinkpython.com/code/PokerHand.py`.

```
from swampy.Lumpy import Lumpy

from PokerHand import *

lumpy = Lumpy()
lumpy.make_reference()

deck = Deck()
hand = PokerHand()
deck.move_cards(hand, 7)

lumpy.class_diagram()
```

Η Εικόνα Γ΄.8 δείχνει το αποτέλεσμα. Η `PokerHand` κληρονομεί την `Deck`. Και η `Deck` και η `PokerHand` έχουν τραπουλόχαρτα.

Αυτό το διάγραμμα δεν δείχνει ότι και η `Hand` έχει τραπουλόχαρτα, επειδή δεν υπάρχουν στιγμιότυπα της Hand στο πρόγραμμα. Αυτό το παράδειγμα καταδεικνύει έναν περιορισμό της Lumpy. Η Lumpy γνωρίζει μόνο τις ιδιότητες και τις σχέσεις ΕΧΕΙ-ΕΝΑ των αντικειμένων τα οποία έχουν αρχικοποιηθεί.