ΑΝΩΤΑΤΟ ΤΕΧΝΟΛΟΓΙΚΟ
ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΚΡΗΤΗΣ
ΠΑΡΑΡΤΗΜΑ ΧΑΝΙΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ Τ.Ε.

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάπτυξη δομημένων λύσεων λογισμικού με χρήση Python

ΑΝΤΩΝΙΟΣ Β. ΡΟΥΣΣΟΣ
Α.Μ. 3271

Επιβλέπων καθηγητής:
Δρ.(Ph.D) ΚΩΝΣΤΑΝΤΑΡΑΣ ΑΝΤΩΝΙΟΣ

ΧΑΝΙΑ 2014

"The Slithery Caress of Python"

Copyright © 2014, Αντώνιος Ρούσσος
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

ΠΕΡΙΛΗΨΗ

Η πτυχιακή αυτή χωρίζεται σε δυο μέρη. Στο πρώτο μέρος η έμφαση δίνεται στο να γίνει μια παρουσίαση της βασικής δομής της γλώσσας προγραμματισμού Python. Ακόμη, παρουσιάζονται αρκετά από τα εργαλεία και μεθόδους που ενσωματώνει η Python για την ανάπτυξη αποτελεσματικού κώδικα. Έχοντας πλέον την στοιχειώδη γνωστική υποδομή από το μέρος ένα, ο αναγνώστης αναμένεται να έχει προφανώς αποκτήσει το κατάλληλο υπόβαθρο για να προχωρήσει στην μελέτη τριών πιο δύσκολων θεμάτων κλιμακωτής δυσκολίας στο δεύτερο μέρος.

Στο πρώτο μέρος, εκθέτονται οι λόγοι που καθιστούν την γλώσσα Python ελκυστική και όλο και πιο δημοφιλή σε ένα ευρύτερο κοινό τα τελευταία χρόνια. Κατόπιν, περιγράφονται τέσσερα μοντέλα προγραμματισμού. Μετά, στρέφεται η προσοχή στην παρουσίαση αυτής καθαυτής της γλώσσας Python από το μηδέν. Περιγράφονται οι βασικοί τύποι μεταβλητών και οι αριθμοί, οι συναρτήσεις σε ξεχωριστά αφιερωμένο κεφάλαιο, οι επαναληπτικές δομές και γενικότερα εντολές ελέγχου ροής προγράμματος. Επιπλέον, παρατίθενται οι βασικοί τύποι δεδομένων που βρίσκουν εφαρμογή στην γλώσσα προγραμματισμού Python όπως είναι η λίστα, το λεξικό και οι πλειάδες. Επιπρόσθετα, αφιερώνεται ένα μεγάλο κεφάλαιο παρουσιάζοντας σε ικανό βάθος και έκταση έξι σημαντικά modules,δηλαδή αυτόνομα αρχεία που περιέχουν συναρτήσεις που ονομάζονται μέθοδοι.

Στο δεύτερο μέρος , το βάρος της προσοχής δίνεται σε τρία ιδιαίτερα θέματα. Αυτά είναι ο εντοπισμός και η αντιμετώπιση σφαλμάτων, ένας αλγόριθμος για μια μηχανή πεπερασμένων καταστάσεων και έναν αλγόριθμο εύρεσης μονοπατιού. Για το πρώτο θέμα, παρουσιάζεται το πρόβλημα και προτείνονται τρόποι προσέγγισης και αντιμετώπισης τέτοιων δυσάρεστων καταστάσεων. Προηγείται των υπόλοιπων δυο θεμάτων σκοπίμως γιατί στοχεύει να προετοιμάσει τον αναγνώστη στο πώς να αντιμετωπίζει τα σφάλματα. Εν συνέχεια, τα άλλα δυο θέματα είναι δυο αλγόριθμοι που σχετίζονται με την τεχνητή νοημοσύνη AI, η έμφαση δεν δίνεται στην ανακάλυψη των αλγορίθμων αλλά στην ανάλυση τους. Έτσι, παρουσιάζονται δυο εκτενείς αναλύσεις που είναι και προτεινόμενοι τρόποι αντιμετώπισης όταν συναντάμε άγνωστο κώδικα.

SYNOPSIS

This thesis is divided to two parts. In part one the emphasis is being given to present the basic structure of Python programming language. Moreover, there are presented many of the tools and methods which are incorporated into Python so as to develop a more effective code. Having already the fundamental knowledge infrastructure from part one, the reader is obviously being expected to have the suitable substrate so as to proceed, in part two, in the study of three more challenging topics of escalating difficulty.

In part one, there are demonstrated some of the reasons which render Python language so attractive, making it more and more popular for a wider audience in the recent years. Moreover, there are described four programming models. Thereafter, the attention is turned into the presentation of the Python language itself from scratch. There are described the basic variable types and numbers, the functions in Python in a separately devoted chapter, the iterative structures and more generally all the flow control techniques of a program. In addition, there are illustrated the basic data types which are applied in Python programming language such as a list, a dictionary and tuples. Even more, there is a dedicated, relatively big chapter on six important modules, which are presented up to a decent depth and extent. That is, modules are independent files which contain functions which are called methods.

In part two, the attention is turning into three special cases. These are the tracing and battling of errors, a Finite State Machine and an algorithm for tracking a path. In the first case, it is stated the problem and there are demonstrated some recommended ways of approaching and solving of such unpleasant situations. Case one is intentionally preceding the rest, because it is aiming to prepare the reader on how to combat errors. Next, the rest two cases are two algorithms which are dealing with artificial intelligence AI, the emphasis is not being laid on the discovery of these algorithms but in their analysis. Thus, there are depicted two extensive analyses which are the recommended ways of dealing with unknown to us code.

ΠΕΡΙΕΧΟΜΕΝΑ

ΜΕΡΟΣ Ι Η γλώσσα Python 1

Κεφάλαιο 1 Εισαγωγή..2
1.1 Γιατί Python 2
1.1.1 Η ελκυστικότητα της 2
1.2 Πως τρέχει ένα πρόγραμμα Python..3
1.3 Το περιβάλλον της Python συνοπτικά 3
1.4 Δομή προγράμματος..4
1.4.1 Μη δομημένος προγραμματισμός..4
1.4.2 Δομημένος προγραμματισμός με υποπρογράμματα..4
1.4.3 Τμηματικός προγραμματισμός..4
1.4.4 Αντικειμενοστραφής προγραμματισμός..5

Κεφάλαιο 2 Μεταβλητές, σταθερές, τελεστές και εκφράσεις 6
2.1 Ακέραιοι και κινητής υποδιαστολής αριθμοί..6
2.2 Δηλώσεις versus εκφράσεις..7
2.3 Γραμμές, εσοχές και δηλώσεις πολλαπλών γραμμών..7
2.4 Το σύμβολο ‘ ; ’ 7
2.5 Κατηγορίες αριθμών..8
2.6 Περισσότερα για τελεστές..8

Κεφάλαιο 3 Συναρτήσεις..11
3.1 Τι είναι η συνάρτηση..11
3.2 Ορισμός και σύνταξη συνάρτησης..12
3.3 Μετατροπές μεταξύ τύπων 13
3.4 Το module math 14
3.5 Συνάρτηση μέσα σε συνάρτηση..14
3.5.1 Ενθυλάκωση και τοπικότητα μεταβλητής 15
3.6 Μέθοδος ανάπτυξης μιας συνάρτησης..16
3.6.1 Προκαθορισμένες τιμές στο interface συνάρτησης..17
3.7 Η import και from..17
3.8 Προσπέλαση docstrings..19

Κεφάλαιο 4 Εντολές ελέγχου ροής προγράμματος και επαναληπτικές δομές 20
4.1 Η εντολή if, elif και η else..20
4.2 Οι επαναληπτικές δομές for και while..21
4.2.1 Οι break, continue και pass..22
4.3 Τι είναι το lambda..22

Κεφάλαιο 5 Τύποι δεδομένων..23
5.1 Strings..23
5.1.1 Δημιουργία string..23
5.1.2 Επεξεργασία και διαμόρφωση string..24
5.1.3 Περισσότερα για τροποποίηση string και σύγκριση..27
5.2 Lists..29
5.2.1 Επεξεργασία λίστας και string..30

5.2.2 Τυπική προσπέλαση βρόχου με λίστα. 32
5.2.3 Στοίβα και ουρά 33
5.2.4 Πίνακες με φωλιασμένες λίστες 34
5.3 Tuples. 35
5.3.1 Σύνολα. 36
5.3.2 Ο τελεστής * στην πλειάδα. 37
5.3.3 Βρόχος με πλειάδες μέσα σε λίστα. 37
5.4 Dictionary. 38
5.4.1 Συνάρτηση και λεξικό 38
5.4.2 Εφαρμογή: μέτρημα στοιχείων. 39
5.4.3 Αναζήτηση κλειδιών σε λεξικό. 40
5.4.4 Ορισμός λίστας ως τιμή σε λεξικό. 40
5.4.5 Hash table στην Python. 41

Κεφάλαιο 6 Θέματα από Python STL και methods. 43
6.1 Η λεπτή διαφορά μεταξύ τυπικών συναρτήσεων και methods.. 43
6.2 Θέματα από Python STL 44
6.2.1 Ενσωματωμένες συναρτήσεις, θεμελιώδεις και μη 44
6.2.2 Exceptions 45
6.2.3 Exceptions παραγόμενα από τον χρήστη και στάνταρ exceptions. 45
6.2.4 File objects και είσοδος από πληκτρολόγιο. 49
6.3 Πρόσβαση αρχείων και καταλόγων με modules. 52
6.3.1 Το os module ως διεπαφή του συστήματος 52
6.3.2 Το os.path module 55
6.3.3 Το sys module 60
6.3.4 Το shutil module. 63
6.3.5 Το glob module. 66
6.4 Μετρώντας το χρόνο. 68

ΜΕΡΟΣ Ι Ι Case study 71

Κεφάλαιο 7.1 Εντοπισμός και διαχείριση σφαλμάτων. 72
7.1.1 Συντακτικά λάθη 73
7.1.2 Σφάλματα runtime 74
7.1.3 Semantic σφάλματα 77
7.2 Μελέτη FSM 77
7.2.1 Ένας αλγόριθμος FSM και μέθοδος προσέγγισης του. 79
7.2.2 Γραμμή προς γραμμή ανάλυση του κώδικα 85
7.3 Εύρεση συντομότερης τυφλής διαδρομής. 87
7.3.1 Ο αλγόριθμος και η ανάλυση του. 88

Συμπεράσματα μέρους Ι και μέρους ΙΙ. 92
Παραπομπές. 93

Ευχαριστίες

Όταν ξεκίνησα την σχολή Ηλεκτρονικών Μηχανικών Τ.Ε. δεν ήμουν σίγουρος αν θα καταφέρω να φτάσω μέχρι το τέλος για διάφορους λόγους... Κόντρα και πείσμα στις αντιξοότητες και τους κακούς δαίμονες το πάλεψα και το ήθελα πολύ από ένα σημείο και μετά γιατί η ηλεκτρονική και γενικότερα η Φυσική ως επιστήμη με προσελκύει. Αποφοιτώ με πολύ μεγάλο βαθμό πτυχίου για να κλείσουν και μερικά στόματα… Ο προπτυχιακός κύκλος σπουδών τελειώνει κι αρχίζει ένα άλλο μεγάλο κεφάλαιο, αυτό της ανακάλυψης χωρίς καθοδήγηση. Θα ήθελα να ευχαριστήσω πρωτίστως και τους δυο γονείς μου και έναν λόγο παραπάνω για την μητέρα μου, που μου συμπαραστάθηκε σα βράχος σε εκείνες τις δύσκολες στιγμές που ήμουν σε τέλμα, που κάθισε και άκουγε τα προβλήματα μου και μου έδινε κουράγιο να συνεχίσω.

Επίσης θα ήθελα να ευχαριστήσω τον καθηγητή Δρ. Κωνσταντάρα Αντώνιο που δέχτηκε να μου κάνει την τιμή να είναι ο επιβλέπων καθηγητής της παρούσας πτυχιακής εργασίας. Τέλος, θα ήθελα να ευχαριστήσω τους φίλους μου που με την παρέα τους απέκτησα κοινή λογική, ανταλλαγή απόψεων και πάρα πολλές χαρούμενες στιγμές ζωής. Πιο ειδικές ευχαριστίες θα ήθελα να δώσω στους Γρηγόρη, Νίκο και Γιάννη.

Αφιερωμένο στους γονείς μου Βασίλειο και Γεωργία

Αφιερωμένο στον Γρηγόρη, Νίκο και Γιάννη

«Αφιερωμένο» σε αυτούς που μου έβαλαν τρικλοποδιές… για να αποτύχω αλλά απέτυχαν!

I remembered black skies, the lightning all around me
I remembered each flash as time began to blur
Like a startling sign that fate had finally found me
And your voice was all I heard that I get what I deserve

So give me reason to prove me wrong, to wash this memory clean
Let the floods cross the distance in your eyes
Give me reason to fill this hole, connect the space between
Let it be enough to reach the truth that lies across this new divide

There was nothing in sight but memories left abandoned
There was nowhere to hide, the ashes fell like snow
And the ground caved in between where we were standing
And your voice was all I heard that I get what I deserve

So give me reason to prove me wrong, to wash this memory clean
Let the floods cross the distance in your eyes across this new divide

In every loss, in every lie, in every truth that you'd deny
And each regret and each goodbye was a mistake too great to hide
And your voice was all I heard that I get what I deserve

So give me reason to prove me wrong, to wash this memory clean
Let the floods cross the distance in your eyes
Give me reason to fill this hole, connect the space between
Let it be enough to reach the truth that lies across this new divide

-New divide , Linking Park-

XI

ΜΕΡΟΣ I

Η γλώσσα Python

Εισαγωγή

Η Python είναι μια γλώσσα προγραμματισμού γενικού σκοπού, υψηλού επιπέδου, διαδραστική, που χρησιμοποιεί διερμηνευτή(interpreter) με δυνατότητα συγγραφής αποδοτικού αντικειμενοστραφή κώδικα, είναι γρήγορη και εύκολη η εκμάθηση της. Ο πατέρας της Python είναι ο Guido van Rossum και δημιουργήθηκε στα τέλη της δεκαετίας '80 με αρχές της δεκαετίας '90 στην Ολλανδία. Η έκδοση που θα χρησιμοποιηθεί είναι η Python v2.7.6 και μπορούμε να την αναζητήσουμε στον ιστότοπο http://www.python.org .Η συγκεκριμένη έκδοση προτιμάται επειδή μέχρι τη στιγμή που συγγράφεται η παρούσα πτυχιακή είναι πολύ δημοφιλής, χαίρει μεγάλης υποστήριξης από τους χρήστες και στο διαδίκτυο υπάρχει μια τεράστια γκάμα σε δωρεάν διαθέσιμα προγράμματα για κάθε εφαρμογή. Οι διαφορές με τον διάδοχο της δεν είναι χαοτικές και με μερικές τροποποιήσεις στον πηγαίο κώδικα μπορούμε να πάρουμε εκτελέσιμο κώδικα συμβατό για την έκδοση 3.x.x δηλαδή τον διάδοχο της 2.x.x . Στόχος αυτού του μέρους 1 είναι να γίνει μια θεμελίωση κάποιων βασικών εννοιών που θα χρησιμοποιούνται μετέπειτα σιωπηρά χωρίς να επεξηγούνται. Το περιβάλλον είναι τα windows.

1.1 Γιατί Python

Σίγουρα υπάρχουν πολλές γλώσσες προγραμματισμού όπως λόγου χάρη η C++, Java, Perl, Lisp και άλλες για να διαλέξει ο χρήστης να γράψει κώδικα, η κάθε μία είναι καλύτερη από άλλες σε κάποιες εφαρμογές, ενώ σε άλλες περιπτώσεις υστερεί. Δεν υπάρχει μια μονάχα γλώσσα προγραμματισμού που να είναι η ιδανική και βέλτιστη επιλογή για κάθε εφαρμογή. Συνεπώς, ο χρήστης θα προτιμήσει την Python για τους παρακάτω λόγους:

- Είναι scripting γλώσσα. Ο κώδικας διερμηνεύεται γραμμή προς γραμμή καθώς εκτελείται, αυτό επιτρέπει τη γρηγορότερη ανάπτυξη κώδικα, το επιθυμητό προϊόν.
- Εύκολη διαχείριση και διόρθωση των λαθών
- Μπορούμε να δοκιμάζουμε τον κώδικα καθώς τον γράφουμε χάρη στον interpreter.
- Αν και μερικές διανομές κοστίζουν, η Python είναι πρακτικά δωρεάν και ότι γράψουμε μας ανήκει.
- Ταχύτατους επιστημονικούς και matrix υπολογισμούς με τη βοήθεια των πακέτων NymPy και SciPy.
- Πολύ ισχυρά GUI εργαλεία όπως τα πακέτα wxPython και pyQt.
- Πανίσχυρα πακέτα σχεδιασμού γραφημάτων(plotting packages) π.χ. MatPlotLib, Chaco
- 3D απεικόνιση με χρήση του πακέτου mayavi
- Πρόσβαση σε beamline controls με χρήση των πακέτων ca_util ή του PyEPICS

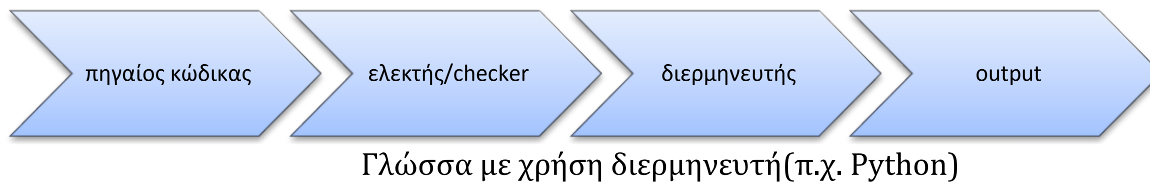
1.1.1 Η ελκυστικότητα της

Η Python αυξάνει ραγδαία τα τελευταία χρόνια σε δημοτικότητα κι όχι τυχαία γιατί διαθέτει μια σειρά από χαρακτηριστικά που την καθιστούν ελκυστική. Έτσι, το ποιο αξιοσημείωτο χαρακτηριστικό εμπίπτει στους όρους της άδειας χρήσης (licence) και ειδικότερα στον όρο νούμερο δύο που αναφέρει τα πνευματικά δικαιώματα και δυνατότητα δημιουργίας, τροποποίησης και αναπαραγωγής κώδικα

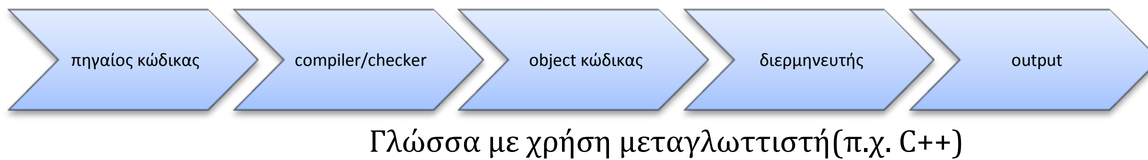
γραμμένο σε Python χωρίς να πρέπει να καταβάλουμε αντίτιμο στον δημιουργό της γλώσσας, δηλαδή την Python Software Foundation ("PSF").Όλα αυτά υπό την προϋπόθεση ότι τηρούμε τους όρους όπως αναφέρονται στην ιστοσελίδα http://docs.python.org/2/license.html. Άλλα δευτερεύοντα χαρακτηριστικά είναι η ευκολία ανάγνωσης του κώδικα και συνεπώς η συντήρηση/αναβάθμιση, η ευκολία εκμάθησης της, η μεγάλη κοινότητα ανθρώπων που χρησιμοποιούν την Python, η δωρεάν υποστήριξη που μπορούμε να λάβουμε και το interactive testing(διαδραστική ανάπτυξη κώδικά λόγω διερμηνευτή).

1.2 Πως τρέχει ένα πρόγραμμα Python

Η γλώσσα προγραμματισμού Python είναι μια interpreted scripting language(γλώσσα σεναρίων με διερμηνευτή) και ως εκ τούτου διαφοροποιείται από τις γλώσσες που μεταγλωττίζουν τον κώδικα (compiler). Παρακάτω δίνεται σχηματικά η διαφορά τους:



Γλώσσα με χρήση διερμηνευτή(π.χ. Python)

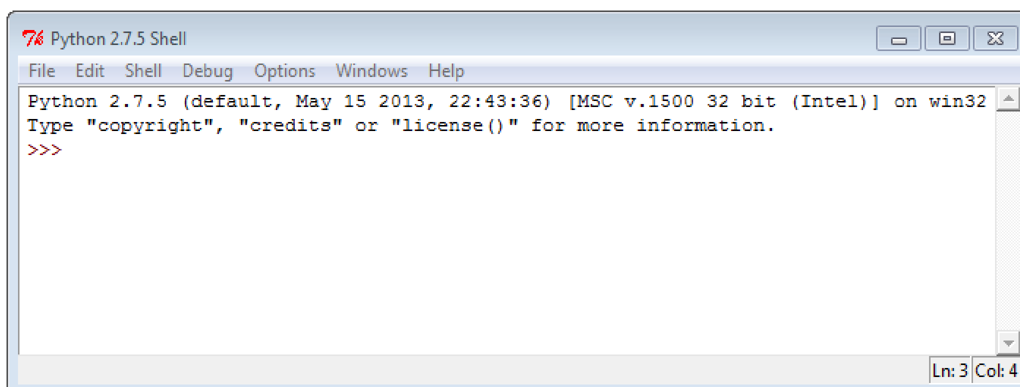


Γλώσσα με χρήση μεταγλωττιστή(π.χ. C++)

Όπως διαπιστώνουμε οι γλώσσες με χρήση μεταγλωττιστή χρειάζονται ένα επιπλέον στάδιο μέχρι να εξάγουν τον κώδικα σε γλώσσα μηχανής που είναι κατανοητός στους υπολογιστές.

1.3 Το περιβάλλον της Python συνοπτικά

Αφού εγκαταστήσουμε την αγαπημένη μας έκδοση και τρέξουμε το IDLE(**I**nteractive **D**eve**L**opment **E**nvironment) που δεν είναι τίποτα άλλο παρά ένα απλό Python GUI θα εμφανιστεί στην οθόνη μας ένα παράθυρο όπως το εικονιζόμενο παρακάτω με τίτλο Python Shell.



Σχήμα 1 - τυπικό Python shell

Δίπλα από το σύμβολο ' >>> ' μπορούμε να γράφουμε εντολές και να πατάμε enter για να εκτελεστούν άμεσα. Εάν επιθυμούμε να γράφουμε script(σενάριο) πρέπει να επιλέξουμε file-> New Window και δε πρέπει να ξεχάσουμε να το σώσουμε με ένα όνομα με κατάληξη .py π.χ. file->save as-> mypython.py .Για

 Copyright © 2014

να τρέξουμε τον κώδικα μας επιλέγουμε από το μενού Run-> Run Module ή πιο απλά πατάμε F5. Με το κουμπί F1 αποκτάμε πρόσβαση σε πολύτιμα κείμενα που μας επεξηγούν την λειτουργία του όλου συστήματος. Το περιβάλλον μπορεί να λειτουργεί και ως ένας υπολογιστής τσέπης (Calculator).Τέλος, να επισημανθεί ότι η Python είναι *case sensitive* που σημαίνει ότι γίνεται διάκριση μεταξύ κεφαλαίων και πεζών, συνεπώς το Start με το start είναι δυο διαφορετικές λέξεις. Τα σχόλια μιας γραμμής τα γράφουμε με το σύμβολο της δίεσης #, ειδάλλως για σχόλια πολλών γραμμών ξεκινάμε με ''' και τελειώνουμε πάλι με ''' .

1.4 Δομή προγράμματος

Ένα πρόγραμμα έχει μια δομή, ένα σκελετό με το οποίο χτίζεται. Υπάρχουν διάφοροι τρόποι ανάπτυξης κώδικα σε μια γλώσσα υψηλού επιπέδου, δηλαδή μια γλώσσα που πλησιάζει την ανθρώπινη γλώσσα, εντούτοις θα περιγραφούν σύντομα και περιεκτικά τέσσερις κλασικές τεχνικές που είναι:

- Μη δομημένος προγραμματισμός
- Δομημένος προγραμματισμός με υποπρογράμματα
- Τμηματικός προγραμματισμός
- Αντικειμενοστραφής προγραμματισμός

1.4.1 Μη δομημένος προγραμματισμός

Πρόκειται για την πιο απλή περίπτωση που τείνουν να προτιμούν οι αρχάριοι χρήστες. Το πρόγραμμα απαρτίζεται από ένα ενιαίο σύνολο εντολών, το κύριο πρόγραμμα. Όλα τα δεδομένα είναι διαθέσιμα στο πρόγραμμα με συνέπεια να έχει πρόσβαση σε αυτά από κάθε σημείο είτε είναι επιθυμητό είτε όχι. Απαιτείται στοιχειώδης προσοχή γιατί μπορεί να γραφεί κώδικας που σύντομα γίνεται δυσανάγνωστος καθώς μεγαλώνει σε έκταση. Αυτό αμέσως συνεπάγεται στη δημιουργία spaghetti προγραμμάτων που δύσκολα συντηρούνται ή αναβαθμίζονται.

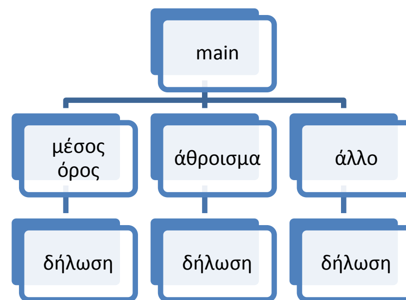
1.4.2 Δομημένος προγραμματισμός με υποπρογράμματα

Σε αυτή την περίπτωση η εφαρμογή σπάει σε υποπρογράμματα διαδικασίες, κάθε μία είναι στοχευόμενη στο να επιτελεί ένα συγκεκριμένο έργο. Το καλό εδώ με αυτή τη δομή προγραμματισμού είναι ότι έχουμε 1)καθολικά δεδομένα που φαίνονται σε όλο το πρόγραμμα από άκρη σε άκρη,2)τοπικά δεδομένα σε κάθε υποπρόγραμμα που φαίνονται, έχει πρόσβαση μόνο το συγκεκριμένο υποπρόγραμμα διαδικασία και πουθενά αλλού.

Μια διαδικασία έχει τη δυνατότητα να επεξεργάζεται δεδομένα και να επιστρέφει αποτελέσματα στο πρόγραμμα που την κάλεσε. Φυσικά υπάρχει το κυρίως πρόγραμμα που τίθεται ως ο επικεφαλής όλων των υποπρογραμμάτων. Αυτό σημαίνει ότι έχουμε ένα κύριο πρόγραμμα που καλεί αυτόνομα αυτοδύναμα υποπρογράμματα ανάλογα με τις απαιτήσεις τις εφαρμογής μας.

1.4.3 Τμηματικός προγραμματισμός

Διαίρει και βασίλευε. Οι πολλές συναρτήσεις στις οποίες βασίζεται σπάνε ένα μεγάλο περίπλοκο πρόγραμμα σε μικρότερα κομμάτια ευκολότερα διαχειρίσημα, επιλύνονται ξεχωριστά, επιστρέφουν το κάθε ένα το δικό του αποτέλεσμα και τέλος συνδυάζονται για να δώσουν την τελική λύση. Στο σχήμα 2 παρατίθεται μια απλοποιημένη περίπτωση με γραφική απεικόνιση για αυτό το στυλ προγραμματισμού.



Σχήμα 2 - ένα απλό παράδειγμα τμηματικού προγραμματισμού

1.4.4 Αντικειμενοστραφής προγραμματισμός

Τελευταίο αφήσαμε τον *αντικειμενοστραφή προγραμματισμό*(OOP). Από μόνος του αποτελεί μια αρκετά διαφορετική φιλοσοφία προγραμματισμού διότι χρησιμοποιεί κλάσεις και αντικείμενα και φυσικά στην Python τα πάντα μπορούν να θεωρηθούν αντικείμενα(objects).Εδώ πλέον γεφυρώνεται το χάσμα μεταξύ διαδικασιών και δεδομένων, θυμίζει αρκετά σε τμηματικό και δομημένο προγραμματισμό γιατί έχει δανειστεί στοιχεία από αυτά. Τρεις έννοιες πολύ σημαντικές και χαρακτηριστικές στον OOP είναι αυτές της κληρονομικότητας, ενθυλάκωσης και του πολυμορφισμού.

Η κληρονομικότητα με απλά λόγια είναι η ικανότητα να κληρονομεί μια νέα κλάση χαρακτηριστικά γνωρίσματα από μια ήδη υπάρχουσα. Δεν είναι τίποτα περισσότερο από μια απλή αντιγραφή – επικόλληση(copy-paste) και προσθήκη επιπλέον γνωρισμάτων κατά το δοκούν.

Η ενθυλάκωση από την άλλη είναι η χρήση συναρτήσεων προκειμένου να μπορούν οι OOP γλώσσες να απομονώνουν και να ομαδοποιούν διαδικασίες και δεδομένα των αντικειμένων.

Τέλος, ο πολυμορφισμός ανάλογα με τι δεδομένα εισέρχονται ,μπορεί να ακολουθηθεί μια διαφορετική λειτουργία, άγνωστη στο χρήστη, προκειμένου να παράγει το κατάλληλο αποτέλεσμα. Με άλλα λόγια, τα αντικείμενα ανάλογα με τον τρόπο που χρησιμοποιούνται είναι ικανά να συμπεριφέρονται διαφορετικά.
Ένα ενδιαφέρον γνώρισμα των OOP είναι ότι μπορούμε να αλλάξουμε τον συνηθισμένο τρόπο λειτουργίας των τελεστών ώστε να υπολογίζουν και ασυνήθιστα πράγματα. Για παράδειγμα μπορούμε να χρησιμοποιήσουμε τον τελεστή + όχι μόνο για αριθμητικές πράξεις αλλά και για να ενώσουμε 2 αλφαριθμητικά σε ένα στην γλώσσα Python. Αυτή η δυνατότητα αλλαγής συμπεριφοράς των τελεστών καλείται *υπερφόρτωση τελεστών*.

Copyright © 2014

Κεφάλαιο
2

Μεταβλητές, σταθερές, τελεστές και εκφράσεις

Όπως γίνεται πάντα οι τέσσερις θεμελιώδεις αριθμητικοί τελεστές βρίσκουν εφαρμογή σε όλες τις γλώσσες προγραμματισμού. Παρακάτω δίδονται στον πίνακα 1.

Πίνακας 1.Τελεστές και λειτουργία τους

τελεστής	λειτουργία
+	Πρόσθεση
-	Αφαίρεση
*	Πολλαπλασιασμός
/	Διαίρεση

Αξίζει να επισημανθεί η διαφορετική λειτουργία που επιτελούν κάποιοι συνηθισμένοι τελεστές. Έτσι ο τελεστής ύψωσης σε δύναμη είναι τα δυο αστεράκια π.χ. 2**3=8,μπορούμε να χρησιμοποιήσουμε την συνάρτηση pow(base,exponent) αν θέλουμε αντί για τον τελεστή ** . Ο τελεστής ^ δεν κάνει ύψωση σε δύναμη στην Python αλλά είναι ο δυαδικός τελεστής για την XOR gate.Επίσης, με τον τελεστή + εκτός από αριθμητική πρόσθεση μπορούμε να ενώσουμε αλφαριθμητικά σε ένα. Για παράδειγμα ‘stop ‘ + ‘stop’ δίνει ένα νέο αλφαριθμητικό το ‘stop stop’.τα κενά έξω από τα αυτάκια ‘ ’ δεν επηρεάζουν, δεν διαβάζονται, μπορούμε να βάλουμε όσα θέλουμε. Αν γράψουμε “stop ” + “stop” θα πάρουμε έξοδο “stop stop”. Για την περίπτωση της διαίρεσης ακέραιος δια δεκαδικός ή το αντίστροφο δίνει αποτέλεσμα πάντα ακέραιο. Για να πάρουμε δεκαδικό πρέπει να κάνουμε *casting.*

Η προτεραιότητα εκτέλεσης των πράξεων είναι με αυτή την σειρά:

- Παρενθέσεις. Το περιεχόμενο μέσα στις παρενθέσεις εκτελείται πάντα πρώτο.
- Εκθέτες. Έπονται των παρενθέσεων σε προτεραιότητα.
- Πολλαπλασιασμός. Είναι το τρίτο στην ιεραρχία προτεραιότητας
- Διαίρεση. Μετά τον πολλαπλασιασμό ακολουθεί η διαίρεση σε βαθμό σημαντικότητας.
- Πρόσθεση και αφαίρεση. Τυπικά είναι ισοδύναμα γι αυτό βάζουμε παρενθέσεις αν συνυπάρχουν.

Παράδειγμα: (2+3)*(2-1)-1+2+2**2*3=5*1-1+2+4*3=5+1+12=18

Μια ακόμα σημαντική παράμετρος είναι οι *δεσμευμένες λέξεις* της Python 2.x.x οι οποίες είναι:

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

Πίνακας 2.Οι 31 δεσμευμένες λέξεις της Python

Αυτός ο πίνακας είναι χρήσιμος για την περίπτωση που ο interpreter «παραπονιέται» για ονόματα μεταβλητών που έχουμε εισάγει και δεν ξέρουμε γιατί παραπονιέται. Ίσως χρησιμοποιούμε δεσμευμένη λέξη γι' αυτό παίρνουμε μήνυμα λάθους.

2.1 Ακέραιοι και κινητής υποδιαστολής αριθμοί

Όπως συμβαίνει στα μαθηματικά έτσι και στον προγραμματισμό οι αριθμοί όπως για παράδειγμα 4, 66, -1, +3 είναι ακέραιοι. Οι αριθμοί 24.1, π, ½,-6.4 είναι κινητής υποδιαστολής. Χρειάζεται προσοχή γιατί λόγου χάριν το 9 είναι ακέραιος στην Python αλλά το 9.0 δεν είναι ακέραιος. Εξαιτίας της τελείας θεωρείται δεκαδικός. Αυτή η απεικόνιση με μια τελεία λέγεται *casting*(ρητή δήλωση τιμής) και είναι χρήσιμη για μορφοποίηση τοπικά ενός αποτελέσματος και για να κάνουμε διαίρεση ακέραιο με δεκαδικό και το αποτέλεσμα να προκύψει ως δεκαδικός ή ακέραιος ανάλογα αν κάνουμε casting ή όχι.

2.2 Δηλώσεις versus εκφράσεις

Οι δηλώσεις και οι εκφράσεις διαφέρουν κατά κάποιο τρόπο και γι αυτό γίνεται εδώ ειδική μνεία. Μια *έκφραση* μπορεί να απαρτίζεται από τιμές, μεταβλητές και τελεστές. Με αυστηρούς όρους κάθε έκφραση είναι δήλωση. Η διαφορά πηγάζει στο γεγονός, ότι μια *έκφραση* φέρει οπωσδήποτε τιμή ενώ μια δήλωση όχι απαραίτητα, μπορεί να έχει τιμή μπορεί και όχι. Ένα παράδειγμα μπορεί να ξεκαθαρίσει το τοπίο καλύτερα. Συνεπώς,

>> 17 είναι μια έκφραση, κάθε τιμή από μονή της αποτελεί μια έκφραση .
>> x μια μεταβλητή αποτελεί μια έκφραση.
>> y=2 ομοίως.
>> y+15 αυτή η δήλωση είναι και έκφραση.

>>print x είναι δήλωση αλλά **όχι** έκφραση.
>> if z==0: ομοίως.

2.3 Γραμμές, εσοχές και δηλώσεις πολλαπλών γραμμών

Μια διαφοροποίηση της γλώσσας Python που γίνεται γρήγορα αντιληπτή είναι ο τρόπος σύνταξης των δηλώσεων. Στην python οι εσοχές παίζουν ρόλο. Μια ομάδα από γραμμές κώδικα, που βρίσκονται για παράδειγμα εντός μιας εντολής επανάληψης(π.χ. η for) ή μιας εντολής ελέγχου όπως η if, πρέπει να βρίσκονται στοιχισμένες η μια εντολή κάτω από την άλλη με ίσης απόστασης εσοχή.

```
>> x=3
>> for item in range(0,3):
>>      print 'current is=%d', %item
>>      x-=3
>>      if x==0:
>>              print '\n the end'
>>              exit()
```
Σχήμα 3 - Σωστές εσοχές.

```
>>      x=3
>> for item in range(0,3):
>>      print 'current is=%d', %item
>>      x-=3
>>      if x==0:
>>      print '\n the end'
>>      exit()
```
Σχήμα 4 - παράδειγμα με **λάθος** σύνταξη

Τυπικά οι δηλώσεις τελειώνουν με κάθε νέα γραμμή. Παρόλα αυτά, μπορούμε να χρησιμοποιήσουμε τον χαρακτήρα \ για να δείξουμε ότι συνεχίζεται η ίδια γραμμή κώδικα στην από κάτω γραμμή, άρα έχουμε δήλωση πολλών γραμμών. Ότι βρίσκεται εντός των { }, [ ] και ( ) δεν χρειάζεται να χρησιμοποιήσει το *χαρακτήρα συνέχισης γραμμής,* δηλαδή το σύμβολο: \ .

```
>>word=item_1 + \
       item_2 + \
       item_3
```
Σχήμα 5 - Παράδειγμα με χρήση του συμβόλου \

2.4 Το σύμβολο ' ; '

Όποιος έχει γράψει κώδικα στην C και την C++ έχει διαπιστώσει ότι η κάθε εντολή πρέπει να τερματίζει με το ερωτηματικό. Αυτό δεν ισχύει στην Python. Στην Python αντί για ερωτηματικό χρησιμοποιούνται εσοχές. Μπορούμε να χρησιμοποιήσουμε το ερωτηματικό για να διαχωρίσουμε δυο απλές εντολές που βρίσκονται στην ίδια γραμμή. Παράδειγμα ορθής χρήσης:
>> x=1; **print** x
>> 1

2.5 Κατηγορίες αριθμών

Η Python υποστηρίζει τέσσερις διαφορετικούς *αριθμητικούς τύπους*:

- float (πραγματικοί αριθμοί κινητής υποδιαστολής)
- int (προσημασμένοι ακέραιοι)
- long (ακέραιοι τύπου long,ισχύει και για οκταδικό και δεκαεξαδικό σύστημα αρίθμησης)
- complex (μιγαδικούς αριθμούς)

int	**float**	**long**	**complex**
-500	160.4	0XEDAFBACDEFBABEL	3+4j
-3	-28.43e9	012345678902L	23.j
1	0.0	214L	-0.2+0.45j
10	-6.	-1251231279358L	6+0j
100	18.2+E5	10101L	0e+9j
43	4-E12	-0X210L	-8-0.666j

Πίνακας 3. Μερικά παραδείγματα αριθμητικών τύπων

Μιας και αναφερόμαστε σε κατηγορίες αριθμών να επισημανθεί ότι στην Python τα πάντα μπορούν να θεωρηθούν αντικείμενα. Έτσι *αντικείμενα αριθμοί*(number object) δημιουργούνται όταν στο αντικείμενο ανατεθεί αριθμητική τιμή, δηλαδή... >> variable=1

2.6 Περισσότερα για τελεστές

Στην αρχή αυτού του κεφαλαίου παρατέθηκαν οι τέσσερις βασικοί τελεστές. Στην παράγραφο αυτή σε μορφή πίνακα θα παρουσιαστεί μια πιο εκτενής λίστα με περισσότερους τελεστές που είναι αξιοποιήσιμοι, καθώς και μια περιεκτική λεκτική περιγραφή της λειτουργία που επιτελεί έκαστος.

Πίνακας 4. Τελεστές

τελεστής	λειτουργία	παράδειγμα
+	Πρόσθεση όρων	x + y= z , με z το όποιο αποτέλεσμα
-	Αφαίρεση όρων	a - b= c
*	Πολλαπλασιασμός όρων	A*B=Ψ

τελεστής	λειτουργία	παράδειγμα
/	Διαίρεση όρων	a/b=πηλίκο
%	Μοντούλο(Modulus). Παρέχει το υπόλοιπο μιας διαίρεσης	A%B=υπόλοιπο
**	Το εκθετικό, ύψωση σε δύναμη.	2**3=8
>	Σύμβολο ανίσωσης, μεγαλύτερο από	3>12, δεν ισχύει η ανίσωση
<	Σύμβολο ανίσωσης, μικρότερο από	3<12, ισχύει
=	Τελεστής ανάθεσης τιμής	1+1=2
<=	Μικρότερο ή ίσον από	4<=5 , 6<=6 , ισχύουν και τα δυο
>=	Μεγαλύτερο ή ίσον από	8>=2
==	Λογικός τελεστής σύγκρισης	5==5 αληθές, 1==2 ψευδές
!=	Διάφορο από	1!=2 αληθές, 3!=3 ψευδές
<>	Ελέγχει αν οι δυο ποσότητες είναι ίδιες. Όμοιος με τον !=	1<>2 αληθές, 3<>3 ψευδές
//	Floor division.Διαίρεση με στρογγυλοποίηση προς τα κάτω	9.0//2.0=4.0 και 9//2=4
-=	Συντόμευση πρόσθεσης και καταχώρησης	x-=2 ισοδύναμο με το x=x-2
+=	Συντόμευση αφαίρεσης και καταχώρησης	x+=2 ισοδύναμο με το x=x+2
*=	Συντόμευση πολλαπλασιασμού και καταχώρησης	x*=2 ισοδύναμο με το x=x*2
%=	Συντόμευση υπόλοιπου διαίρεσης και καταχώρησης	x%=2 ισοδύναμο με το x=x%2
/=	Συντόμευση διαίρεσης και καταχώρησης	x/=2 ισοδύναμο με το x=x/2

<table>
<tr><th>τελεστής</th><th>λειτουργία</th><th>παράδειγμα</th></tr>
<tr><td>//=</td><td>Συντόμευση του floor division</td><td>x//=y ισάξιο με x=x//y</td></tr>
<tr><td>**=</td><td>Συντόμευση του εκθετικού</td><td>x**=y ισάξιο με x=x**y</td></tr>
<tr><td>~</td><td>Συμπλήρωμα ως προς ένα</td><td>~7= -8</td></tr>
<tr><td>|</td><td>Δυαδικό OR.Ένας όρος αρκεί να ισχύει</td><td>if x=0 | y=0: z=1</td></tr>
<tr><td>^</td><td>XOR</td><td>11 xor 8=3</td></tr>
<tr><td>&</td><td>Δυαδικό και. Πρέπει να ισχύουν και τα δυο</td><td>if x=0 & y=0: z=1</td></tr>
<tr><td>>></td><td>Ολίσθηση προς τα δεξιά, όσες φορές ορίζει ο δεξιός όρος</td><td>8>>1=4 γιατί ο δεξιός όρος είναι 1</td></tr>
<tr><td><<</td><td>Ολίσθηση προς τα αριστερά, όσες φορές ορίζει ο δεξιός όρος</td><td>7<<2=1 γιατί ο αριστερός όρος=2</td></tr>
<tr><td>or</td><td>Το λογικό ή. Αρκεί ένας όρος να είναι αληθές.</td><td>if x=0 or x=1: print ‘hello’</td></tr>
<tr><td>and</td><td>Το λογικό και. Πρέπει και οι δυο όροι να είναι αληθείς</td><td>If x=1 and y=1:print ‘bye’</td></tr>
<tr><td>not</td><td>Το λογικό NOT, ο τελεστής άρνησης</td><td>a not b</td></tr>
<tr><td>is</td><td>Αληθές αν σε μια πρόταση και οι δυο τιμές ταυτίζονται</td><td>1 is 1 αληθές</td></tr>
<tr><td>in</td><td>Αληθές αν το ένα στοιχείο είναι μέλος του άλλου</td><td>a in b αληθές αν το a ανήκει στο b</td></tr>
<tr><td>is not</td><td>Αληθές αν σε μια πρόταση και οι δυο τιμές δεν ταυτίζονται</td><td>2 is not 2 αληθές</td></tr>
<tr><td>not in</td><td>Αληθές αν σε μια πρόταση δεν βρεθεί η ζητούμενη τιμή</td><td>x=’s’ ; if ‘b’ not in x: print ‘true’</td></tr>
</table>

Κεφάλαιο 3

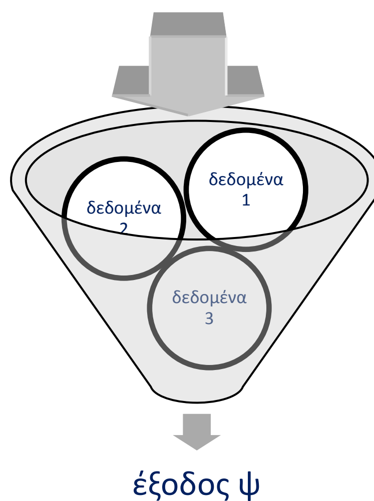
Συναρτήσεις

Αυτό το κεφάλαιο θα αφιερωθεί σε μια σπουδαία λειτουργία που είναι οι συναρτήσεις. Θα οριστεί μια συνάρτηση, θα περιγραφεί ο τρόπος για να τις χρησιμοποιούμε, θα δοθούν ενδεικτικά παραδείγματα χρήσης και θα παρουσιαστούν βοηθητικά στοιχεία που συνοδεύουν μια συνάρτηση. Μια πολύ απλή και χρήσιμη ενσωματωμένη συνάρτηση είναι η **type(όρισμα)** π.χ.

```
>>> type(12)
<type 'int'>
```

3.1 Τι είναι η συνάρτηση

Στα μαθηματικά λέμε ότι είναι η σχέση που συνδέει το x με το F(x) δηλαδή το ψ. Έχουμε την ανεξάρτητη μεταβλητή x που μπορεί να παίρνει διάφορες τιμές και κοιτάζουμε τι παθαίνει, δηλαδή τι τιμή θα πάρει, η εξαρτημένη μεταβλητή ψ για τις διάφορες τιμές του x(x1,x2,x3,...). Στο προγραμματισμό δε διαφέρει, η λογική αυτή υπάρχει και ακολουθείται επίσης. Είναι δυνατόν να έχουμε πολλές εισόδους-πολλά δεδομένα x και ένα ψ ως έξοδο αποτέλεσμα, την *τιμή επιστροφής*.



Σχήμα 6 – ένα απλό μοντέλο με γραφική απεικόνιση συνάρτησης

3.2 Ορισμός και σύνταξη συνάρτησης

Έχοντας στα μυαλό μας την σχηματική αναπαράσταση της συνάρτησης από την προηγούμενη παράγραφο θα επεκταθούμε σε ένα πιο συστηματικό ορισμό. Μια συνάρτηση ορίζεται με τον ακόλουθο τρόπο:

def όνομα_συνάρτησης (παράμετρος1,παράμετρος2,…):
    δήλωση 1
    δήλωση 2
        …
    δήλωση κ

Οπότε με λόγια μπορούμε να πούμε ότι στον προγραμματισμό η *συνάρτηση* είναι μια ακολουθία δηλώσεων που πραγματοποιούν έναν υπολογισμό. Μπορεί να παίρνει ορίσματα μπορεί και όχι.

Επεξηγήσεις:

- Μια συνάρτηση στην Python ορίζεται ξεκινώντας με το συνθηματικό def.
- Για όνομα συνάρτησης διαλέγουμε ότι θέλουμε αρκεί να μην είναι δεσμευμένη λέξη και να μην ξεκινά ο πρώτος χαραχτήρας με αριθμό. Επίσης δεν επιτρέπονται τα κενά, για κενό χρησιμοποιούμε ένα υποκατάστατο του κενού χαραχτήρα, την κάτω παύλα _ (underscore).
- Προσέχουμε να μην ξεχάσουμε να γράψουμε την άνω και κάτω τελεία **:** ,είναι υποχρεωτικό.
- Οτιδήποτε βρίσκεται εντός των παρενθέσεων καλείται *interface ή περιβάλλον δηλώσεων.*
- Αν δεν θέλουμε ορίσματα στο interface το αφήνουμε κενό το περιεχόμενο των παρενθέσεων.
- Δουλεύουμε με εσοχές, είναι υποχρεωτικό. Όπως βλέπουμε και στην σύνταξη η ‘δήλωση 1’ ξεκινά με εσοχή κατά σύμβαση τεσσάρων κενών, το ίδιο ακολουθούν και οι υπόλοιπες. Δεν παίζει ρόλο ο αριθμός των κενών αρκεί να είμαστε συνεπείς και να χρησιμοποιούμε τον ίδιο αριθμό κενών.
- Μπορούμε να χρησιμοποιήσουμε φωλιασμένες δηλώσεις, δηλαδή φωλιασμένες δηλώσεις μέσα σε δηλώσεις, αρκεί να είμαστε συνεπείς με τις εσοχές όπως έγινε στο Σχήμα 3, θα δοθεί παράδειγμα κώδικα παρακάτω για να γίνει πιο κατανοητό.
- Η συνάρτηση επιστρέφει μια τιμή ανάλογα τι δουλειά της έχουμε βάλει να επιτελέσει, μπορεί να μην επιστρέφει και καμία τιμή αν το θέλουμε.

Για την καλύτερη κατανόηση των παραπάνω θα δοθεί τώρα ένα καλό παράδειγμα και κατόπιν η ανάλυση του.

```
def findsqrt(myvalue,aproximation):
'''this function might fail if myvalue is floating point
or irrational number. myvalue means find sqrt(myvalue)
aproximation means any number lower than myvalue,if
possible use numbers greater than 1.
'''
    accuracy=0.0001
    while True:
        #print aproximation
        resul=(aproximation+myvalue/aproximation)/2.0
        if abs(resul-aproximation)<accuracy:
            break
        aproximation=resul
    print resul
findsqrt(4,1)
```

Σχήμα 7 – εύρεση τετραγωνικής ρίζας με τη μέθοδο του Νεύτωνα

Στόχος δεν είναι να κατανοήσουμε τον κώδικα τι λέει, πως και τι κάνει αλλά αποτελεί ένα υπόδειγμα άριστης χρήσης δομημένου προγραμματισμού ,των σωστών εσοχών και σχολίων που είναι και ο στόχος αυτής της παραγράφου. Βλέπουμε ότι για να ορίσουμε μια συνάρτηση ξεκινάμε με το συνθηματικό def, κατόπιν το όνομα που θέλουμε και μέσα στην παρένθεση τις παραμέτρους. Άρα το interface αυτής της συνάρτησης έχει δυο ορίσματα. Ακριβώς από κάτω φτιάχνουμε σχόλια(με το πράσινο χρώμα),αυτό το πεδίο είναι σημαντικό αλλά και προαιρετικό, αν θέλουμε το κάνουμε. Καλό είναι να το συμπεριλαμβάνουμε στο κώδικα μας και να μην γράφουμε ‘πως’ το κάνει ο κώδικας αλλά ‘τι’ θέλει να κάνει. Το ‘πως’ μπορεί να απαντηθεί διαβάζοντας τον κώδικα αλλά το ‘τι’ χρειάζεται να το επεξηγούμε γιατί μετά από κάποιες εβδομάδες, μήνες ή χρόνια αν θέλουμε να ξαναχρησιμοποιήσουμε τον κώδικα δε

θα θυμόμαστε τι λειτουργία επιτελεί. Τα σχόλια τα τοποθετούμε όπου θέλουμε κατά βούληση, δηλαδή δεν χρειάζεται να υπακούουν σε κανόνες με εσοχές. Ο interpreter όταν διαβάζει τον κώδικα τα προσπερνά, δε τα λαμβάνει υπόψη του.

Το σώμα της συνάρτησης με τον κώδικα που μας ενδιαφέρει αρχίζει ακριβώς από την λέξη accuracy(τα σχόλια που προηγούνται τα προσπερνά ο interpreter) και τελειώνει στην προτελευταία γραμμή στην εντολή :print resul,που όντως είναι στην ίδια κατακόρυφο νοητή γραμμή με την εντολή: accuracy=0.0001. Παρατηρούμε ότι στο σώμα της συνάρτησης η πρώτη εντολή ξεκινά με εσοχή απαραιτήτως. Ακριβώς από κάτω έχουμε μια άλλη εντολή, την while. Η while απαιτεί όλες οι εντολές στις οποίες θα δράσει να έχουν μια εσοχή πιο μέσα από την ίδια την while. Η *εμβέλεια*-δηλαδή το πεδίο δράσης- της while τελειώνει στην εντολή: approximation=resul. Η εντολή if έχει επίσης το δικό της συντακτικό και πάλι έχουμε μια επιπλέον εσοχή. Βλέπουμε επίσης ότι με την δίεση # εισάγαμε και ένα βοηθητικό σχόλιο μιας γραμμής. Με την δίεση εισάγουμε σχόλια μονάχα μιας γραμμής, μετά ξανά πάλι δίεση και συνεχίζουμε. Τέλος, επειδή η findsqrt(4,1) βρίσκεται στην ίδια νοητή κάθετο με την def findsqrt(4,1) συμπεραίνουμε ότι <u>δεν ανήκει</u> στο σώμα της συνάρτησης η findsqrt(4,1),αλλά αποτελεί μια άλλη εντολή που κάτι κάνει. Στην προκείμενη περίπτωση απλά καλούμε την συνάρτηση να μας υπολογίσει την τετραγωνική ρίζα του τέσσερα. Άρα, μόλις είδαμε πως καλούμε μια συνάρτηση, απλώς γράφουμε το όνομα της και τις παραμέτρους <u>χωρίς</u> το συνθηματικό def, είδαμε επίσης την ροή εκτέλεσης των εντολών από πάνω προς τα κάτω.

3.3 Μετατροπές μεταξύ τύπων

Είναι σημαντικό να δείξουμε πως μπορούμε να μετατρέπουμε έναν αριθμητικό τύπο σε κάποιο άλλο με χρήση ενσωματωμένων συναρτήσεων της Python.Αυτό γιατί δεν γίνεται να κάνουμε υπολογισμούς όπως αλφαριθμητικό με ακέραιο, πρέπει όλα τα δεδομένα να είναι ίδιο τύπου. Οπότε το κόλπο είναι να μετατρέπουμε όλα τα δεδομένα σε ένα τύπο και να κάνουμε τους λογισμούς. Θα παρουσιαστούν διάφοροι συνδυασμοί υπολογισμών με τους τύπους int, float και str.

Τύπος int. Ξεκινάμε με την μετατροπή ενός float αριθμού σε int, μετά ενός σκέτου αριθμού string(str) σε int και τέλος ενός str με γράμματα σε int που θα δώσει μήνυμα λάθους όπως θα φανεί γιατί το 'bye' δεν είναι αριθμός και άρα δεν γίνεται να μετατραπεί σε int.

```
>>> int(15.5)
>>> 15
>>> int ('10')
>>>10
>>>int('bye')
ValueError: invalid literal for int() with base 10: 'bye'
```

Τύπος string.Το 100 είναι τύπου int και μετατρέπεται σε '100' άρα string. Παρομοίως για τα υπόλοιπα.

```
>>> str(100)
>>>'100'
>>> str(1.1)
>>> '1.1'
>>> str(x)
NameError: name 'x' is not defined
```

Προσοχή το x είναι μη ορισμένη μεταβλητή ,δεν είναι αριθμός.

Τύπος float. Ίδια λογική με παραπάνω.

```
>>> float(10)
>>> 10.0
>>> float('25.5')
>>> 25.5
>>>float (16.66)
>>>16.66
```

3.4 Το module math

Για να χρησιμοποιήσουμε τις συναρτήσεις που βρίσκονται μέσα στην βιβλιοθήκη/module math θα πρέπει να χρησιμοποιήσουμε την εντολή *import* για να την εισάγουμε και να γίνει διαθέσιμη. Σε επόμενο κεφάλαιο θα δούμε ότι η βιβλιοθήκη math είναι ένα *module object*(αυτοτελές αντικείμενο).
>>> **import** math
Αυτή η δήλωση δημιουργεί ένα module object με όνομα math.Ας δούμε τι θα γίνει αν τυπώσουμε:
>>> **print** math
<module 'math' (built-in)>

Αυτό το math module object εμπεριέχει μεταβλητές και συναρτήσεις για μαθηματικά. Για να χρησιμοποιήσουμε ας πούμε το sin(x) θα πρέπει να εφαρμόσουμε το συμβολισμό με την τελεία: math.sin(x) .Πρώτα γράφουμε το module object από όπου θα αντλήσουμε την συνάρτηση και κατόπιν την επιθυμητή συνάρτηση. Παράδειγμα σωστής εφαρμογής όπως το ακόλουθο σχήμα.

```
>>> import math
>>> A=3
>>> R=A/(2*math.sin(A))
>>> R
10.62925109360578
>>> print R
10.6292510936
```

Σχήμα 8 – χρήση της συνάρτησης sin(x).Προσέξτε την διαφορά στην ακρίβεια του αποτελέσματος της print R από το σκέτο R

Πρέπει να θυμόμαστε ότι όλα τα τριγωνομετρικά ορίσματα πρέπει υποχρεωτικά να δίνονται σε rad, είναι η προεπιλογή της γλώσσας. Αν θέλουμε φτιάχνουμε μια φόρμουλα μετατροπής από μόνοι μας γνωρίζοντας ότι για να μετατρέψουμε μοίρες σε ακτίνια διαιρούμε με το 360 και πολλαπλασιάζουμε με 2π. Παράδειγμα:

```
>>> import math
>>> deg =45
>>> rad = deg / 360.0 * 2 * math.pi
>>> math.sin(rad)
0.7071067811865475
```

Το π και το log καθώς και πολλές άλλες βρίσκονται μέσα στην math.Μπορούμε να γράψουμε help('math') για περισσότερες λεπτομέρειες. Δε πρέπει να ξεχνάμε την εντολή import math αλλιώς δεν θα δουλεύει.

3.5 Συνάρτηση μέσα σε συνάρτηση

Από την στιγμή που έχουμε κατασκευάσει μια συνάρτηση μπορούμε να την χρησιμοποιήσουμε έτοιμη σε μελλοντικά προγράμματα. Μπορεί να είναι όρισμα σε μια άλλη νέα συνάρτηση ή να βρίσκεται μέσα στο σώμα της νέας συνάρτησης. Πρέπει να τονιστεί ότι μια συνάρτηση πρέπει να πρώτα εκτελεστεί μια φορά πριν χρησιμοποιηθεί ως όρισμα ή ως δήλωση κάπου αλλού.
Σε μορφή script(ctrl+N για να ανοίξει νέο παράθυρο) γράφουμε τον παρακάτω κώδικα.

```
def copyme():
    dsp()
    dsp()
def dsp():
    print "I'm new to python language."
    print "I want to learn how to code in Python."
copyme()
```

Το αποτέλεσμα είναι να τυπωθούν οι δυο γραμμές κείμενο δυο φορές. Αυτό που συμβαίνει είναι ότι ορίσαμε μια συνάρτηση copyme() χωρίς ορίσματα την οποία την καλούμε στην τελευταία γραμμή για να δούμε την δουλειά μας. Μέσα στο σώμα της copyme() έχουμε την συνάρτηση dsp(display) και στο σώμα

της τελευταίας γράψαμε απλώς ένα κείμενο χάριν απλότητας. Αφού έχουμε την dsp(display) έτοιμη πλέον ,δε μας εμποδίζει τίποτα να την ξαναχρησιμοποιήσουμε όσες φορές θέλουμε κι αυτό κάναμε, την εντάξαμε στο σώμα της copyme() εις διπλούν. Έχοντας παρουσιάσει αυτό το απλό παράδειγμα επαναχρησιμοποίησης της dsp() μέσα στην συνάρτηση copyme() θα αποκαλύψουμε ένα πιο ολοκληρωμένο κώδικα που έχει τη δυνατότητα πολλαπλών εκτυπώσεων.

```
def show1x(k):
   print k

def show2x(kk):
   show1x(kk)
   show1x(kk)

def show3x(kkk):
   show1x(kkk)
   show2x(kkk)

def show4x(kkkk):
   show2x(kkkk)
   show2x(kkkk)

word="I'm sexy and I know it. "*2
show3x(word)
```

Το αποτέλεσμα είναι:

```
I'm sexy and I know it. I'm sexy and I know it.
I'm sexy and I know it. I'm sexy and I know it.
I'm sexy and I know it. I'm sexy and I know it.
```

Αν γράψουμε show1x(word) αντί για show3x(word) τότε θα εμφανιστεί μια γραμμή μόνο. Αν γράψουμε show2x(word) θα πάρουμε δυο γραμμές. Αν γράψουμε show4x(word) θα εμφανιστούν 4 γραμμές. Ο κώδικας μπορεί να τροποποιηθεί να τυπώνει όσες φορές θέλουμε αν αυτό επιθυμούμε.

3.5.1 Ενθυλάκωση και τοπικότητα μεταβλητής

Δυο ακόμα πολύ χρήσιμα χαρακτηριστικά γνωρίσματα που μπορούμε να διακρίνουμε στις συναρτήσεις είναι η *τοπικότητα* μιας μεταβλητής και η *ενθυλάκωση*. Θα παρατεθεί ένα νέο παράδειγμα για να τονιστούν αυτές οι δυο έννοιες. Σε ένα αρχείο script γράφουμε τον παρακάτω κώδικα, το σώζουμε και το εκτελούμε.

```
def mydiv(a,b):
   result=a/b
   print result
```

Κατόπιν πάμε στον interpreter και πληκτρολογούμε το εξής απλό:

```
>>> mydiv(6,3)
2
```

Προφανώς η διαίρεση του 6 με το 3 δίνει 2.Αν γράψουμε στον interpreter σκέτο mydiv παίρνουμε την εξής περίεργη απάντηση:

<function mymul at 0x02153AF0>

Αυτό απλώς μας βεβαιώνει ότι η mydiv είναι ένα αντικείμενο που δημιουργήθηκε και βρίσκεται αποθηκευμένο στην θέση μνήμης 0x02153AF0 η οποία θα διαφέρει σίγουρα από υπολογιστή σε υπολογιστή. Οποιαδήποτε μεταβλητή ορίζεται εντός μιας συνάρτησης όπως η result έχουν *τοπική εμβέλεια*, συνεπώς παύει να υφίσταται η τιμή της μεταβλητής εκτός της συνάρτησης. Αυτό σημαίνει ότι

αν έχω μια νέα συνάρτηση την mymul που χρησιμοποιεί επίσης την μεταβλητή result τότε η result της συνάρτησης mydiv θα έχει διαφορετική τιμή από την result της mymul.Αυτό που συμβαίνει είναι ότι θα τρέξει ο κώδικας για την mydiv ,η result θα πάρει μια τιμή ,μόλις βγούμε από αυτή τη συνάρτηση θα εξαφανιστεί η τιμή που έφερε η result και μετά όταν θα εκτελεστεί η mymul η result θα πάρει μια νέα τιμή και μόλις βγούμε από την mymul θα χαθεί πάλι η τιμή της result. Για όση ώρα βρισκόμαστε εντός μιας συνάρτησης η *τοπική μεταβλητή* result διατηρεί την κατάσταση της σύμφωνα με το κώδικα που έχουμε γράψει.

Ας εξετάσουμε τον εξής κώδικα:

```
>>> a=2
>>> b=2
>>>result=a*b
>>> print result
4
>>> result=3+3
>>> print result
6
```

Το κακό με τον παραπάνω κώδικα είναι ότι ορίσαμε τον πολλαπλασιασμό δυο αριθμών και αποθηκεύσαμε αυτή την πράξη σε μια μεταβλητή result αν θελήσουμε να ξανακάνουμε ένα πολλαπλασιασμό θα πρέπει να ξαναγράψουμε ολόκληρο τον κώδικα από την αρχή γιατί βλέπουμε παρακάτω ότι η result πλέον κάνει πρόσθεση, μετά την πρόσθεση η result έχει την τιμή 6 και δεν ξανακάνει πρόσθεση δεύτερη φορά. Αυτό το μοντέλο προγραμματισμού δεν εξυπηρετεί γιατί κάθε φορά πρέπει να γράφουμε ολόκληρο τον κώδικα από την αρχή. Η λύση είναι η *ενθυλάκωση*, δηλαδή να εσωκλείσουμε την επιθυμητή πράξη μέσα σε μια συνάρτηση όπως παρακάτω.

```
def mymul(a,b):
    result=a*b
    print result
```

Η result είναι τοπική μεταβλητή, κάθε φορά που θέλουμε να κάνουμε ένα πολλαπλασιασμό δυο αριθμών απλώς καλούμε την συνάρτηση με το όνομα της δηλαδή mymul, εισάγουμε τους αριθμούς a,b που θέλουμε και παίρνουμε το αποτέλεσμα με την print.

3.6 Μέθοδος ανάπτυξης μιας συνάρτησης

Αν σκοπεύουμε να χρησιμοποιήσουμε ένα κομμάτι κώδικα(code block) πολλές φορές καλό είναι να το εσωκλείσουμε μέσα σε μια συνάρτηση και κατόπιν απλώς να το καλούμε με το όνομα της συνάρτησης. Έτσι γλιτώνουμε από το να γράφουμε ξανά και ξανά τα ίδια. Ένας κώδικας πρέπει να είναι αρκετά απλός, κατανοητός και να επιτελεί μια διεργασία μόνο όχι πολλές. Ο κώδικας πρέπει να είναι απλός αλλά όχι υπερβολικά απλός. Μπορούμε να ξεκινούμε την ανάπτυξη του αλγορίθμου από το ειδικό και μετά να πηγαίνουμε στο γενικό. Οι προαναφερθέντες κώδικες τηρούν αυτές τις αιτιάσεις.

Έστω ότι θέλουμε να βρούμε το άθροισμα των αριθμών 3 και 5. Σε ένα script αρχείο γράφουμε:

```
def myadd():
    result=3+5
    print result
```

Όσες φορές κι αν γράψουμε myadd() στο python shell θα παίρνουμε πάντα το ίδιο αποτέλεσμα, 8. Δεν χρειάζονται ορίσματα για ένα τόσο συγκεκριμένο παράδειγμα γι αυτό το interface της συνάρτησης είναι κενό. Αν θέλουμε η συνάρτηση να βρίσκει το άθροισμα δυο οποιοδήποτε αριθμών θα πρέπει να γενικεύσουμε αυτό το ειδικό παράδειγμα σε ένα ποιο γενικό.

```
def myadd(a,b):
    result=a+b
    print result
```

Σ αυτόν τον τελευταίο κώδικα γενικεύσαμε τη συνάρτηση myadd() και τις δώσαμε επίσης δυο ορίσματα, τα a,b.Αυτή η γενίκευση μας επιτρέπει να εισάγουμε και να υπολογίζουμε το άθροισμα των δυο εκάστοτε αριθμών. Το πλεονέκτημα αυτής της μεθόδου, δηλαδή της *προοδευτικής γενίκευσης* μιας συνάρτησης, είναι προφανές γιατί ξέρουμε πώς να επιτελέσουμε μια πολύ εξειδικευμένη διεργασία, αποτυπώνουμε τον αλγόριθμο και μετά τον γενικεύουμε ώστε να καλύπτει ένα ευρύτερο φάσμα περιπτώσεων. Το κέρδος είναι ότι μπορούμε να επαναχρησιμοποιήσουμε τον κώδικα, να τον βελτιστοποιήσουμε και να τον αναβαθμίσουμε κατά βούληση. Τέλος, δε ξεχνάμε να εισάγουμε και μερικά βοηθητικά σχόλια ώστε όταν περάσει λίγος καιρός να μπορέσουμε να θυμηθούμε τι ακριβώς κάνει απλώς διαβάζοντας τα σχόλια.

3.6.1 Προκαθορισμένες τιμές στο interface συνάρτησης

Αξίζει να γίνει ξεχωριστή μνεία για τις *default values*(προκαθορισμένες τιμές) συνάρτησης. Είδαμε την έννοια της προοδευτικής γενίκευσης και πως υλοποιείται στην πράξη, οπότε λείπει να συμπεριλάβουμε ένα ακόμα στοιχείο για να έχουμε μια ολοκληρωμένη μέθοδο ανάπτυξης συνάρτησης.

```
def mymul(a=1,b=1):
   result=a*b
   print result
```

Στο interface συνάρτησης στα a,b δώσαμε χειροκίνητα default values.Αυτό το κόλπο διαδραματίζει τεράστιο ρόλο σε μεγάλης έκτασης προγράμματα. Το γιατί είναι απλό. Για default values επιλέγουμε, ανάλογα με το κώδικα τι κάνει, τις κατάλληλες τιμές ώστε σε περίπτωση που πάει κάτι στραβά να μην επηρεαστεί η ομαλή λειτουργία του συνολικού προγράμματος. Στο προκείμενο απλουστευμένο παράδειγμα επιλέξαμε το ουδέτερο στοιχείο του πολλαπλασιασμού. Αν για παράδειγμα ξεχάσουμε στη διάρκεια του debugging να βάλουμε μια τιμή, θα χρησιμοποιηθεί η προκαθορισμένη. Επίσης, με αυτή την τεχνική προστατευόμαστε από την ανεπιθύμητη κατάσταση να παγώσει το πρόγραμμα επειδή δεν ορίσαμε τιμή ή ακόμα χειρότερα να μπει μια τιμή άκρως ανεπιθύμητη εν αγνοία μας με απρόβλεπτα αποτελέσματα και δύσκολο να εντοπιστεί η αιτία του κακού. Ενώ με τα default values γνωρίζουμε ότι βάζουμε εσκεμμένα ουδέτερες τιμές που δεν θα επηρεάζουν την ομαλή εκτέλεση του προγράμματος, αναλόγως την περίπτωση. Το όφελος είναι ότι κερδίζουμε σε χρόνο(λιγότερο debugging) και κόπο.

3.7 Η import και from

Έγινε μια σύντομη αναφορά νωρίτερα στην import, μιλήσαμε για το module math, εδώ θα δούμε ποιο διεξοδικά τρόπους για να εισάγουμε modules και objects που εμπεριέχονται σε modules.Ο σκοπός είναι να εισάγουμε έτοιμες συναρτήσεις που είναι αποθηκευμένες στο σκληρό δίσκο, που έχουμε φτιάξει εμείς ή κάποιος άλλος ή ακόμα τις ενσωματωμένες της Python.

Ο πρώτος τρόπος για να εισάγουμε modules είναι με την import και το όνομα του module να ακολουθεί:

```
>>> import math
>>> print math.pi
3.14159265359
```

Στο κώδικα βλέπουμε ότι αφού εισάγουμε τη βιβλιοθήκη/module math μπορούμε πλέον να αποκτάμε πρόσβαση στο περιεχόμενο του module που στη προκείμενη περίπτωση είναι το pi(δηλαδή το π=3.1415…). Αν γράφαμε σκέτο το pi και πατάγαμε enter θα παίρναμε μήνυμα λάθους. Εάν κάνουμε import ένα module αυτό παραμένει στη μνήμη ,ο μόνος σίγουρος τρόπος να το ξεφορτωθούμε είναι να πατήσουμε επανεκκίνηση shell restart(ctrl+F6).Προφανώς μία φορά import αρκεί δε χρειάζεται να γράφουμε πολλές φορές την εντολή. Αν κάνω αλλαγές σε κάποιο module πρέπει να τερματίσω το session και να αρχίσω από το μηδέν και ξανά νέο import.Άλλος τρόπος είναι με την εντολή reload() να κάνω ενημέρωση του module δίχως επανεκκίνηση του προγράμματος όπως περιγράφηκε παραπάνω. Συμπέρασμα, με την import ‘φορτώνουμε’ συγκεκριμένα περιεχόμενα-στοιχεία από αυτή την βιβλιοθήκη.

Ένα πιο αναλυτικό παράδειγμα σωστής αξιοποίησης αυτής τις εντολής με ένα δικό μας αυτοσχέδιο αρχείο:

Υποθέτουμε ότι γράφουμε πολύ συχνά και πολλές φορές τις ίδιες συναρτήσεις κατ' επανάληψη, οπότε είναι άκρως επιθυμητό να συγκεντρώσω όλες τις πολύ χρησιμοποιούμενες συναρτήσεις μέσα σε ένα αρχείο, δηλαδή ένα module, που θα έχει οπωσδήποτε κατάληξη .py ,βαφτίζουμε αυτό το αρχείο ας πούμε apothiki.py . Ας φανταστούμε ότι μια συνάρτηση που χρησιμοποιούμε πάρα πολύ είναι η ακόλουθη:

```
def myfan( ):
        print ' =menu start is initialized= '
```

Εφόσον την γράψουμε σε μορφή script την σώζουμε μέσα στο module apothiki ,δηλαδή σώζουμε ως (save as...) apothiki.py . Πλέον αυτό το module περιέχει μια συνάρτηση και μπορούμε να τη καλέσουμε με τον εξής τρόπο στο Python Shell:

```
>>> import apo8iki
>>> apo8iki.myfan( )
=menu start is initialized=
```

Το συγκεκριμένο αποτελεί ένα απλό παράδειγμα γι αυτό δεν χρειάζεται ορίσματα η myfan(). Ας θεωρήσουμε ότι επιθυμούμε να αναβαθμίσουμε την myfan() να κάνει κάτι διαφορετικό:

```
def myfan( ):
        print ' #main menu has been commenced# '
```

εάν πάμε στη γραμμή εντολών και γράψουμε:

```
>>> import apo8iki
>>> apo8iki.myfan( )
=menu start is initialized=
```

Δυστυχώς παρατηρούμε ότι δεν πέρασε η αλλαγή που κάναμε άλλο περιμέναμε να δούμε, σίγουρα όχι: =menu start is initialized= . Για να δουλέψει σωστά έπρεπε να είχαμε κλείσει τελείως το τωρινό μας session, δηλαδή να κάνουμε επανεκκίνηση. Από το μενού: Shell → Restart Shell (Ctrl+F6) και τώρα όντως παίρνουμε σωστά αποτελέσματα:

```
>>> import apo8iki
>>> apo8iki.myfan( )
#main menu has been commenced#
```

Αν θέλουμε μπορούμε να πάμε ένα βήμα παρακάτω τον παραπάνω κώδικα για απλοποίηση και περιορισμό ανθρώπινου λάθους από απροσεξία:

```
>>> import apo8iki
>>> nea=apo8iki.myfan()
>>> nea()
#main menu has been commenced#
```

Αυτό που κάναμε είναι ουσιαστικά αλλαγή μεταβλητής. Πήραμε την μακροσκελή πρόταση apo8iki.myfan() που τυχαίνει να μην έχει ορίσματα χάριν απλότητας και την αντιστοιχίσαμε στην μεταβλητή nea. Κατόπιν καλούμε την nea ως nea() συνάρτηση! Άλλος τρόπος για πιο γρήγορα αποτελέσματα χωρίς να κλείσουμε το τρέχον session-δηλαδή να μην κάνουμε επανεκκίνηση- είναι με την εντολή reload() :

```
>>> reload(apothiki)
```

Βέβαια υπάρχει και άλλος τρόπος με την βοήθεια της εντολής from αλλά και του αστερίσκου.

```
>>> from math import pi
>>> print pi
3.14159265359
```

Βλέπουμε πόσο κοντά στην ανθρώπινη γλώσσα τείνει να είναι ο κώδικας, από το module math εισάγουμε το pi και κατόπιν εμφανίζεται η τιμή του.

Υπάρχει η δυνατότητα με τη βοήθεια του αστερίσκου να αποκτήσουμε απευθείας πρόσβαση στο pi ή όποιο άλλο στοιχείο θέλουμε με τον εξής τρόπο:

```
>>> from math import *
>>> pi
3.141592653589793
>>> cos(-pi)
-1.0
```

Το μειονέκτημα με αυτό το τελευταίο παράδειγμα είναι ότι μπορεί να έχουμε ίδια ονόματα που προέρχονται από διαφορετικά modules ή από δικά μας ονόματα που έχουμε ορίσει. Το πλεονέκτημα είναι ότι χρησιμοποιήσαμε κατευθείαν ένα συγκεκριμένο τριγωνομετρικό στοιχείο(επειδή στο συγκεκριμένο παράδειγμα έτυχε να είναι από το math module).

3.8 Προσπέλαση docstrings

Όταν διαθέτουμε μια μεγάλη βιβλιοθήκη από συναρτήσεις και δεν θυμόμαστε ποια συνάρτηση κάνει την δουλειά που θέλουμε, ένας τρόπος είναι να διαβάσουμε το κώδικα και ένας άλλος τρόπος είναι να χρησιμοποιήσουμε κατάλληλες εντολές και να διαβάσουμε τα σχόλια όπως αυτά στο σχήμα 7 που έχουμε φροντίσει να συμπεριλαμβάνουμε στους κώδικες μας. Οι εντολές που μπορούν να μας κάνουν τη ζωή ευκολότερη είναι η help, η doc και η dir. Θα εφαρμόσουμε αυτές τις τρεις χρησιμοποιώντας την math ως βοήθημα για να γίνει καλύτερη αντιληπτή η σωστή χρήση τους.

```
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos',
sh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
il', 'copysign', 'cos', 'cosh', 'degrees', 'e',
f', 'erfc', 'exp', 'expm1', 'fabs', 'factorial'
>>> math.__doc__
'This module is always available.  It provides access to
the\nmathematical functions defined by the C standard.'
>>>
>>> help(math)
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
```

Σχήμα 9

Θυμίζεται ότι χρειάζεται μόνο μία φορά να γίνει import για οποιοδήποτε module.Βλέπουμε ότι η dir(math) μας δείχνει τι συναρτήσεις περιέχει το module μέσα του. Έτσι, αν έχουμε φτιάξει ένα δικό μας module με αυτό τον τρόπο μπορούμε γρήγορα να δούμε μέσα από το περιβάλλον της Python τι συναρτήσεις περιέχει. Η math.__doc__ (με δυο κάτω παύλες πριν και μετά τη λέξη doc) μας δίνει μια γενική περιγραφή που αν την είχαμε εφαρμόσει στο σχήμα 7 θα εμφάνιζε τα πράσινα γράμματα. Τέλος, η help εμφανίζει μια πιο λεπτομερή λίστα με το τι περιέχει το module.Στο κομμάτι 'FUNCTIONS' μας δίνεται αναλυτική περιγραφή της κάθε διαθέσιμης συνάρτησης που φέρει το module μέσα του, κάτι που έχει παραλειφθεί στο σχήμα. Αν θέλαμε μια μικρή βοήθεια μόνο για το συνημίτονο γράφουμε απλά help("math.cos").

Κεφάλαιο

4

Εντολές ελέγχου ροής προγράμματος και επαναληπτικές δομές

Σε αυτό το κεφάλαιο θα παρουσιαστούν μερικά βασικά και συνάμα χρήσιμα εργαλεία για την ανάπτυξη αλγορίθμων στην γλώσσα Python όπως λόγου χάρη For, if, while, break, continue και is instance.Σκοπός του κεφαλαίου είναι να αποτυπωθεί ο τρόπος σύνταξης τους και ορθής χρήσης αυτών με χρήση απλών παραδειγμάτων.

4.1 Η εντολή if, elif και η else

Η πιο απλή περίπτωση που θα μπορούσε να περιγραφεί είναι η εντολή ελέγχου ροής if. Το συντακτικό είναι πολύ απλό:

```
if συνθήκη ελέγχου:
   δήλωση 1
   δήλωση 2
      ...
   δήλωση κ
```

Παράδειγμα:

```
if 5>3 :
   print 'true'
```

Παρατηρούμε ότι ξεκινάμε την σύνταξη με την δεσμευμένη λέξη κλειδί if με μικρά γράμματα μετά γράφουμε την συνθήκη ελέγχου που θέλουμε να επαληθεύσουμε και δεν ξεχνάμε την άνω και κάτω τελεία. Στις επόμενες γραμμές που ακολουθούν γράφουμε το σώμα του κώδικα μας. Αν θέλουμε μπορούμε να έχουμε περισσότερες if η μια μέσα στην άλλη, δηλαδή *φωλιασμένη if*:

```
if x>z:
   if z>y:
      print 'z is between x and y'
```

ή πιο απλά:

```
if x>z>y
   print 'z is between x and y'
```

Βέβαια το x>z>y είναι ισοδύναμο του x>z & z>y.Φυσικά μπορούμε να αξιοποιήσουμε την δυνατότητα της if…else και να ξαναγράψουμε τον παραπάνω κώδικα με αυτήν. Το συντακτικό είναι:

Συντακτικό:

```
if συνθήκη ελέγχου:
   δηλώση 1
elif συνθήκη ελέγχου:
   δήλωση i
elif συνθήκη ελέγχου:
   δήλωση ii
     ....
else:
   δήλωση 2
```

παράδειγμα:

```
if x>z:
   print x
elif  z>y:
   print z
else:
   print y
```

Στο παράδειγμα γράψαμε μόνο μία elif αν χρειαζόταν θα γράφαμε περισσότερες. Τελευταία πάντα η else.

Καλό είναι να αποφεύγουμε να χρησιμοποιούμε πολλές if και πολλές φωλιασμένες if γιατί γρήγορα μπορεί να οδηγηθούμε σε αυτό που λέμε spaghetti προγράμματα που είναι δυσανάγνωστα.

4.2 Οι επαναληπτικές δομές for και while

Το να βασιστούμε μόνο στην εντολή if για να φτιάξουμε μεγάλης έκτασης προγράμματα δεν είναι αρκετό, αλλά με την βοήθεια των επαναληπτικών δομών μπορούμε να αναπτύξουμε πολύ προχωρημένα προγράμματα. Θα παρουσιαστεί πρώτα η while, η σύνταξη της είναι εύκολη.

```
while συνθήκη :
    δήλωση 1
    δήλωση 2
    ...
    δήλωση κ
```

Μέσα στο σώμα της while μπορούμε να ενσωματώσουμε όπως εντολές if ,for και ότι άλλο χρειάζεται για να αναπτύξουμε το κώδικα μας. Ένα απλό παράδειγμα ακολουθεί με χρήση function για να αξιοποιήσουμε ότι έχει παρουσιαστεί μέχρι στιγμής:

```
def antistrofi_metrisi(x):
    while x>0:
        print x
        x-=1
    if x==0:
        print 'the end'
```

Ξεκινάμε σε ένα αρχείο script. Ονοματίσαμε τη συνάρτηση antistrofi_metrisi με παράμετρο x, θα βάλουμε μια τιμή όταν θα καλέσουμε την συνάρτηση στο Python shell(IDLE). Στην επόμενη γραμμή ξεκινάμε, με κατάλληλη εσοχή, την while η οποία θα είναι αληθής μέχρι να μηδενιστεί το x.Όσο το x είναι διάφορο του μηδέν τυπώνει αριθμούς με φθίνουσα σειρά. Μόλις μηδενιστεί το x τερματίζεται η while και εκτελείται η if x==0 και η δήλωση που βρίσκεται μέσα στο σώμα αυτής. Αυτό που πρέπει να προσέξουμε είναι ο μετρητής x-=1. Όταν χρησιμοποιούμε την εντολή επανάληψης while πρέπει να χρησιμοποιούμε ένα μετρητή αλλιώς θα παγιδευτεί ο κώδικας σε *ατέρμονα βρόχο*(infinite loop). Τέλος, δε πρέπει να ξεχνάμε να αρχικοποιούμε τον μετρητή σε μια ορισμένη τιμή.

Μια ακόμη πολύ σημαντική εντολή ροής προγράμματος είναι η for. Με αυτή την εντολή διατρέχουμε, προσπερνάμε, στοιχεία ενός συνόλου(δηλ. λίστα ή αλφαριθμητικό) το ένα μετά το άλλο με βάση την σειρά εμφάνισης τους στην ακολουθία γεγονότων. Ο προεπιλεγμένος τρόπος ανάγνωσης γίνεται πάντα από αριστερά προς τα δεξιά, εκτός κι αν ορίσουμε κάτι διαφορετικό. Ο τρόπος σύνταξης της for με μορφή παραδείγματος είναι:

```
>>> Mylist=1,2,3,4,5,6,7,8,9
>>> for item in Mylist:
 >>>   print item
```

Αυτό που θα γίνει είναι να τυπώσει τους αριθμούς από το 1 μέχρι το 9 κατακόρυφα, δηλαδή έναν αριθμό σε κάθε γραμμή. Αν θέλουμε όλους τους αριθμούς σε μια μόνο γραμμή βάζουμε ένα κόμμα στο τέλος της λέξης item, γράφουμε δηλαδή: >>> print item, . Το **in** είναι δεσμευμένη λέξη τελεστής κι όπως θα φανεί σε κώδικες του παρόντος συγκράματος σε επόμενα κεφάλαια η for δρα πάντα πάνω σε λίστες και αλφαριθμητικά. Φυσικά υπάρχει και καλύτερος τρόπος να διατρέχουμε στοιχεία σειριακά με την βοήθεια της εντολής range(start,stop,step).Παρατηρούμε ότι δέχεται τρία ορίσματα το interface της συνάρτησης αυτής και μπορεί αν θέλουμε να διατρέχει στοιχεία από το τέλος προς την αρχή με το μείον, δηλαδή range(-9,-1,-1).Παρακάτω δίνεται εφαρμογή με τη κανονική φορά.

```
>>> for item in range(1,9,1):
>>>     print item
```

4.2.1 Οι break, continue και pass

Αν επιθυμούμε ευελιξία στους αλγορίθμους που γράφουμε υπάρχουν τρεις εντολές με τις οποίες μπορούμε να ελιχθούμε μέσα στις επαναληπτικές δομές όπως η for και η while. Η εντολή break αυτό που κάνει είναι να διακόπτει για παράδειγμα την λειτουργία της for,πηγαίνει οριστικά έξω από το πεδίο δράσης της for και εκτελεί το υπόλοιπο κομμάτι του κώδικα που βρίσκεται από κάτω.

```
def spotBot(x,y,word):
  for item in word:
    if item==x or item==y:
      print item
      break
  else:
    print 'no "a" either "e" was found'
```

Εφόσον το έχουμε αποθηκεύσει ως αρχείο script, ο κώδικας διαβάζει μια λέξη κι αν εντοπίσει ένα από τα δυο φωνήεντα (a,e) το τυπώνει και τερματίζεται η for νωρίτερα πριν διαβαστεί ολόκληρη η μεταβλητή word.Παρατηρούμε τη θέση του else,δεν πρόκειται για λάθος είναι όντως η εναλλακτική επιλογή αν αποτύχει η for. Μπορούμε να δοκιμάσουμε να γράψουμε στον interpreter για παράδειγμα:
spotBot('a', 'b', 'window') και θα προκύψει η περίπτωση με την else.

Στη περίπτωση που χρησιμοποιήσουμε την εντολή continue αυτό που συμβαίνει είναι ότι συνεχίζει με την επόμενη επανάληψη του βρόχου.

```
for item in range(2, 10,1):
   if item % 2 == 0:
     print "even,", item
     continue
   print "figure= ", item
```

Υπάρχει περίπτωση να φτιάχνουμε κάποιον αλγόριθμο και σε ένα κομμάτι του κώδικα δεν έχουμε αποφασίζει τι ακριβώς θα κάνει όποτε μπορούμε να γράψουμε απλώς pass και αργότερα όταν αποφασίζουμε να ολοκληρώσουμε το ελλιπή κώδικα επιστρέφουμε και να τον συμπληρώνουμε.

```
for item in range(2, 10,1):    #για βήμα αύξησης ανά ένα σωστό επίσης και το  range(2,10)
   pass
for item in word:
   print item
```

Όπως βλέπουμε για τη πρώτη for είμαστε αναποφάσιστοι ,αλλά για να μην ξεχάσουμε να συμπληρώσουμε αυτό το χώρο στο μέλλον γράφουμε pass κάτι που λέει στον interpreter μην κάνει σε αυτό το σημείο τίποτα και να προχωρήσει παρακάτω. Η pass είναι εξαιρετικά χρήσιμη γιατί μπορούμε να αναβάλουμε για αργότερα τη συγγραφή κάποιων περιοχών κώδικα ,να την χρησιμοποιούμε σαν παράμετρο σε διάφορα σημεία που απλώς θέλουμε προσωρινά να τα προσπεράσουμε και να τα ελέγξουμε μετέπειτα και ασφαλώς με τη χρήση αυτής της εντολής δεν θα πάρουμε μήνυμα λάθους πράγμα που βοηθά στην εκσφαλμάτωση.

4.3 Τι είναι το lambda

Η γλώσσα Python υποστηρίζει ένα ιδιότυπο συντακτικό. Μας επιτρέπει να ορίσουμε μίνι συναρτήσεις μιας γραμμής αλλά και για γενίκευση ενός αλγόριθμου(π.χ. στην κρυπτογράφηση). Παράδειγμα:

```
def squares(x):
        oneline = lambda x: x**2 #εύρεση τετράγωνου ενός αριθμού με όρισμα x και φόρμουλα:x**2
        return oneline(x)
```

Οι τύποι δεδομένων

Οι τυποποιημένοι *τύποι δεδομένων* που υποστηρίζει η γλώσσα είναι:

- String(αλφαριθμητικό)
- Numbers(αριθμούς)
- List(λίστα)
- Tuple(πλειάδα)
- Dictionary(λεξικό)

Οι αριθμοί έχουν καλυφθεί επαρκώς σε προηγούμενο κεφάλαιο και δε θα ξαναπαρουσιαστούν.

5.1 Strings

Ένα string(αλφαριθμητικό) είναι μια ακολουθία από χαραχτήρες σταθερού μήκους. Αυτό πρακτικά σημαίνει ότι μόλις οριστεί ένα string δε μπορεί να αλλάξει. Φυσικά, αν θέλουμε να τροποποιήσουμε ένα string θα πρέπει να δημιουργήσουμε ένα κλώνο που θα περιέχει την τροποποίηση ,το αρχικό δεν αλλάζει. Η τροποποίηση θα μπορούσε να γίνει με την γνωστή από άλλο κεφάλαιο συνάρτηση replace(). Τυπικά, τα string στην Python αποθηκεύονται εσωτερικά σε μορφή 8-bit ASCII.

Μερικές χαρακτηριστικές δυνατότητες με τα strings είναι ότι μπορούμε να τα επεξεργαστούμε και να εφαρμόσουμε πολλές συναρτήσεις με αυτά. Έτσι, μπορούμε να επιλέξουμε να εμφανίσουμε ένα μόνο κομμάτι του string,μπορούμε να δημιουργήσουμε Unicode string, να δοθούν ως ορίσματα σε συναρτήσεις, σε tuples και dictionaries.Αυτές είναι κάποιες από τις χαρακτηριστικές δυνατότητες τους.

5.1.1 Δημιουργία string

Στην Python ένα string μπορεί να δημιουργηθεί με μονούς *μνημονικούς αποστρόφους*(quotes) , διπλούς ή τριπλούς(όπως έχει ήδη δειχθεί). Η ιδιαιτερότητα ειδικά με τα τριπλά quotes είναι ότι επιτρέπεται να καταλαμβάνουν πολλαπλές γραμμές.

```
>>> variable1= 'welcome'
>>> variable2= "in Python."
>>> variable3= ''' this is a
multiline quotation text
for demostration'''
```

Εάν το επιθυμούμε μπορούμε να χρησιμοποιήσουμε την ανάποδη κάθετο \ (backslash), όπως για παράδειγμα:

```
>>> show_text="welcome to Python, \"they said\" "
```

 Copyright © 2014

Πως θα εμφανιστεί:
>>>show_text
' welcome to Python, "they said" '
Είναι ένα string όπως δηλώνουν και τα quotes.Εντούτοις με την print επιστρέφει κείμενο *raw string*:
>>> **print** show_text
welcome to Python, "they said"
Εάν γράψουμε: r'[εισαγωγή κειμένου]' χωρίς κενά:
>>> **print** r'my doc'
Θα πάρουμε το εξής:
my doc

Αυτό ήταν μια χρήση ενός raw string.Κάθε χαραχτήρας που εισάγουμε σε μορφή raw string παραμένει απαράλλακτος και σε αυτή την περίπτωση ο χαραχτήρας backslash δεν αντιμετωπίζεται ως ειδικός χαρακτήρας. Στην περίπτωση που θέλουμε Unicode string αντί για r γράφουμε u ή U:
>>> print u'Hello my python world!'
Hello my python world!

Τα Unicode string αποθηκεύονται ως 16-bit Unicode.Φυσικά, αυτό επιτρέπει χρήση μεγαλύτερης ποικιλίας διαθέσιμων χαρακτήρων συμπεριλαμβανομένων και συμβόλων από πάρα πολλές ομιλούμενες γλώσσες.

5.1.2 Επεξεργασία και διαμόρφωση string

Για να αποκτήσουμε στοχευμένη πρόσβαση σε ένα σημείο ή κομμάτι το string γράφουμε:
>>> myvar='stringtest' #πρέπει πρώτα να δημιουργήσουμε ένα string για δοκιμές.
>>> myvar[1]
't'
>>> myvar[0]
's'
>>> myvar[0:1]
's'
>>> myvar[0:4]
'stri'
>>> myvar[2:6]
'ring'
>>> myvar[:5]
>>>'strin'

Η μεταβλητή myvar[1] ζητά να αποκτήσουμε πρόσβαση στο στοιχείο με δείκτη 1. Ο αριθμός που βρίσκεται εντός των αγκυλών καλείται *δείκτης* (index), οι τιμές του είναι index=0,1,2,.... συνεπώς αν θέλουμε να εμφανίσουμε το s που είναι το πρώτο γράμμα στη λέξη stringtest πρέπει να γράψουμε myvar[0]. Αν θέλουμε να αποτυπώσουμε πολλά συνεχόμενα γράμματα καταφεύγουμε σε *φάσμα δεικτών* ή όπως επίσημα λέγεται *string slicing*(τεμαχισμός αλφαριθμητικού) π.χ. στο myvar[0:4], θα τυπωθούν όλα τα γράμματα εκτός του τελευταίου δηλαδή η myvar[0:4] τυπώνει ουσιαστικά από το 0 μέχρι το 3. Το ίδιο ισχύει για κάθε περίπτωση. Ας δοκιμάσουμε να γράψουμε με μείον :

>>> myvar[-6:-2]
'ngte'
>>> myvar[-6:-2:2]
'nt'
Όπως φαίνεται μετρά από το τέλος της λέξης, από δεξιά προς αριστερά με το τελευταίο γράμμα της λέξης stringtest να έχει αριθμό index=-1. Αν θέλουμε επιλέγουμε το βήμα όπως επιλέχθηκε να είναι το 2 στην myvar[-6:-2:2].

Γενικά η μορφοποίηση(format) που ακολουθείται παρουσιάζει το συντακτικό:

variable_name[start:stop:step]

Προσέχουμε ότι σε περίπτωση που χρησιμοποιήσουμε το μείον στο πεδίο start θα αρχίσει π.χ. στο παραπάνω παράδειγμα myvar[-6:-2] να μετρά από τέρμα δεξιά προς τα αριστερά 6 θέσεις, βρίσκει ότι πρόκειται για το γράμμα n,το τυπώνει καθώς και όλα όσα βρίσκονται δεξιά του μέχρι την θέση index=-2.

Ένα άλλο παράδειγμα με τις μέρες τις εβδομάδας και πως τις αντιστοιχούμε με index:
```
>>> range(7)
[0, 1, 2, 3, 4, 5, 6]
>>> (Mon, Tue, Wed, Thu, Fri, Sat, Su) =range(7)
>>> Tue
1
>>>Fri
4
```
Η Δευτέρα(Mon) αντιστοιχεί στο 0 οπότε η Κυριακή(Su) αντιστοιχεί στο 6.

Επιπρόσθετα, για την διαμόρφωση ενός string υπάρχει η δυνατότητα να χρησιμοποιήσουμε τον τελεστή % και επίσης να παντρέψουμε 2 ξεχωριστά sting σε ένα:
```
>>> flower1='rose '
>>> flower2= 'poppy'
>>> flower3=flower1 + flower2
>>> price=100
>>>discount=5
>>>print '%s flower costs %d units including discount of=%d percent ' %(flower3, 100, -5)+'(%)'
rose poppy flower costs 100 units including discount of=-5 percent (%)
```

Αυτή η προτελευταία γραμμή χρειάζεται μια επεξήγηση. Το %s δηλώνει να τυπωθεί ένα string,θα τυπωθεί το περιεχόμενο του flower3 που είναι ουσιαστικά η ένωση των λέξεων rose και poppy. Ο τελεστής διαμόρφωσης εξόδου %d units αντιστοιχεί σε ένα ακέραιο και πιο συγκεκριμένα στο 100 και ο τελευταίος τελεστής διαμόρφωσης %d αντιστοιχεί στο -5. Παίζει ρόλο η σειρά τοποθέτησης τους. Επίσης, επειδή θέλω να εμφανίσω πολλά αποτελέσματα μαζί με το που κλείσει ο απόστροφος (') πρέπει να αρχίσει αμέσως ο τελεστής % και μέσα σε παρένθεση τα αποτελέσματα. Αν είχα ένα μόνο αποτέλεσμα δε θα χρειαζόταν οι παρενθέσεις. Ο σταυρός στο τέλος δεν κάνει αριθμητική πρόσθεση αλλά εμφανίζει το σύμβολο του ποσοστού, συνεπώς κάνει ένωση δυο αλφαριθμητικών συνόλων.

Ένας ενδιαφέρον τρόπος αποδοτικής χρήσης του τελεστή % παρουσιάζεται στο ακόλουθο δείγμα κώδικα:

```
>>> days=('Monday','Tuesday')
>>> readme="Welcome.Today is %s and tommorow will be %s."
>>> print readme % days
Welcome.Today is Monday and tommorow will be Tuesday.
>>>
```
Σχήμα 10 – μια πρακτική χρήση του τελεστή %

Ασφαλώς υπάρχουν πολλοί τρόποι που συνδυάζονται με τον τελεστή μορφοποίησης εξόδου (%) και ένας πίνακας, πίνακας 5, με περιεκτική περιγραφή προσφέρεται στην επόμενη σελίδα για ευκολότερη ανάγνωση και γρηγορότερο κατατοπισμό. Ο πίνακας 6 που θα ακολουθήσει αναφέρει τους τελεστές που δρουν σε αλφαριθμητικά και τέλος ο πίνακας 7 τους χαραχτήρες διαφυγής. Οι πίνακες 6 και 7 είναι αυτούσιοι από το manual της Python και μπορούν να βρεθούν από το μενού βοήθεια(F1) στο Python Shell IDLE version 2.7.6 .

Πίνακας 5. Μορφοποίηση εξόδου

Διαμόρφωση συμβόλου	Είδος μετατροπής
%d	Προσημασμένος δεκαδικός ακέραιος
%i	Προσημασμένος δεκαδικός ακέραιος
%u	Μη προσημασμένος δεκαδικός ακέραιος
%f	Αριθμός κινητής υποδιαστολής
%g	Παράγει ίδια αποτελέσματα με %f and %e ,όσο πιο συνοπτικά γίνεται
%G	Παράγει ίδια αποτελέσματα με %F and %E ,όσο πιο συνοπτικά γίνεται
%c	χαρακτήρας
%s	Μετατροπή string διαμέσου της str() πριν την μορφοποίηση
%e	Εκθετικό με μικρό e
%E	Εκθετικό με κεφαλαίο E
%X	Δεκαεξαδικό, αναπαράσταση ακεραίου με κεφαλαίο X
%x	Δεκαεξαδικό, αναπαράσταση ακεραίου με μικρό x
%o	οκταδικό

Πίνακας 6. String special operators.Υποθέτοντας ότι a= 'Hello' και b='Python'.

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give **e**
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give **ell**
in	Membership - Returns true if a character exists in the given string	**H in a** will give 1
not in	Membership - Returns true if a character does not exist in the given string	**M not in a** will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	**print r'\n'** prints \n and **print R'\n'** prints \n
%	Format - Performs String formatting	See at next section

Πηγή: Από το manual της Python έκδοση 2.7.6 στο μενού βοήθεια.

Πίνακας 7. Escape Characters. Δεν τυπώνονται, δρουν σε string με μονά και διπλά quotes.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

Πηγή: Από το manual της Python έκδοση 2.7.6 στο μενού βοήθεια.

5.1.3 Περισσότερα για τροποποίηση string και σύγκριση

Θα ακολουθήσουν αρκετές περιπτώσεις παραδείγματα. Υπάρχει μια ενδιαφέρουσα ενσωματωμένη συνάρτηση, η len(), που έχει την δυνατότητα να μετρά το μήκος ενός string, δηλαδή από πόσους χαρακτήρες αποτελείται, τα κενά προσμετρούνται επίσης!

```
>>> string_1= "RoofTop"
>>> string_2= "RoofTop    "
>>> string_3= "RoofTop\n "
>>> print len(string_1)
7
>>> print len(string_2)
11
>>> print len(string_3)
9
```

Η λογική σύγκριση των strings είναι μια επίσης ενδιαφέρουσα περίπτωση προς διαπραγμάτευση και βασίζεται στους σχεσιακούς τελεστές. Εξυπακούεται ότι η σύγκριση μπορεί να επεκταθεί και σε άλλους τύπους δεδομένων. Για να δοκιμάσουμε αν δυο strings είναι ίσα γράφουμε για παράδειγμα:

```
>>> x='door'
>>> z= 'window'
>>> x==z
False
>>> x is z
False
```

Σε αυτό το παράδειγμα αποκαλύπτονται δυο τρόποι, με το λογικό **is** και το == .Συνήθως το is το χρησιμοποιούμε για να βεβαιώσουμε αν δυο αντικείμενα(objects) έχουν την ίδια ταυτότητα, άρα ταυτίζονται, αλλά δουλεύει και με strings διότι και τα strings στην Python ως αντικείμενα αντιμετωπίζονται. Το σύμβολο == δεν κάνει ανάθεση τιμής αλλά σύγκριση και θέλει προσοχή να μην μπερδευτούμε με αυτή τη λειτουργία. Όπως φαίνεται στο κώδικα πήραμε την ένδειξη ‘False’ διότι το περιεχόμενο των μεταβλητών x και z διαφέρει. Αν θέσουμε z='door' τότε θα πάρουμε ένδειξη True(αληθές). Υπάρχει μια λεπτή διαφοροποίηση μεταξύ του == και του is: το == παράγει πάντα σωστά αποτελέσματα όταν συγκρίνουμε τιμές ενώ το is συγκρίνει εάν δυο μεταβλητές αναφέρονται στο ίδιο αντικείμενο. Για του λόγου του αληθές:

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

Σε αυτή την περίπτωση δείχθηκε ότι οι δυο μεταβλητές είναι *ισοδύναμες* αλλά δεν είναι *ταυτόσημες*, διότι δεν είναι ένα και το αυτό object,αλλά δυο διαφορετικές μεταβλητές που δείχνουν στο ίδιο αντικείμενο. Αν επιθυμούμε να βάλουμε σε αλφαβητική τάξη δυο λέξεις μπορούμε να χρησιμοποιήσουμε ανισώσεις:

```
>>> word1='balloon'
>>> word2='bag'
>>> if word1>word2:
>>>     print  word1+' έπεται του '+ word2
>>> elif word1<word2:
>>>     print word1+' προηγείται του '+ word2
```

Αυτό που χρειάζεται να τονιστεί στις ανισώσεις με αλφαριθμητικά είναι ότι στην Python αν κάποια γράμματα είναι κεφαλαία οπουδήποτε μέσα στις λέξεις τότε συγκρίνονται πρώτα και η ταξινόμηση γίνεται με βάση τα κεφαλαία γράμματα έως ότου εξαντληθούν μετά συνεχίζεται με τα πεζά, αν μόνο η μια από τις δυο λέξεις έχει τουλάχιστον ένα κεφαλαίο γράμμα τότε θα προηγείται πάντα της άλλης λέξης. Αν αυτό αποτελεί δυσάρεστη κατάσταση για τον χρήστη ,μπορεί να μετατρέψει όλα τα γράμματα κεφαλαία ή πεζά στις προς σύγκριση λέξεις, να συγκρίνει και μετά να τις επαναφέρει στην αρχική τους μορφή με κατάλληλο κώδικα. Στην περίπτωση που χρειάζεται να συγκρίνουμε με το τίποτα None, είναι προτιμότερο να χρησιμοποιήσουμε το **is None** κι όχι το == .

Εάν είχαμε Boolean τιμές μπορούμε αντί να γράψουμε:

```
>>> if y==True:
        [code ...]
```

Μπορούμε να γράψουμε απλούστερα:

```
>>> if y :
        [code ...]
```

5.2 Lists

Μια λίστα είναι μια ακολουθία από *τιμές*(values), σε αντίθεση με τα string που ήταν χαρακτήρες. Οι τιμές σε μια λίστα καλούνται *στοιχεία*(items/elements) και μπορούν να είναι οποιουδήποτε τύπου. Μπορούμε να σκεφτούμε ότι μια λίστα είναι η σχέση που έχουν τα στοιχεία της λίστας με τους δείκτες της λίστας που δείχνουν στα στοιχεία της λίστας. Ο πιο απλός τρόπος να δημιουργηθεί μια λίστα είναι να εσωκλείσουμε μέσα σε τετράγωνες αγκύλες τα στοιχεία:
>>> [1, 'hello', 2.5, 4, 9, 'yes']

Αφού τα στοιχεία μιας λίστας μπορεί να είναι οτιδήποτε μπορούμε να τοποθετήσουμε ακέραιους, κινητής υποδιαστολής αριθμούς και αλφαριθμητικά στοιχεία χωρίς πρόβλημα, ο τρόπος που θα αξιοποιηθούν είναι θέμα του χρήστη.

Ασφαλώς μπορούμε να έχουμε *φωλιασμένες λίστες*, δηλαδή λίστα μέσα σε άλλη:
>>> ['yes', 'no', [1,2,3], '10', 'k']

Μια *κενή λίστα* ορίζεται με τις τετράγωνες αγκύλες χωρίς περιεχόμενο:
>>> []

Μπορούμε να αναθέσουμε τις λίστες σε μεταβλητές και να τυπώσουμε :
>>> x=[1, 2, 3, 4, 5, 6]
>>> emtpy=[]
>>> zet=['tree', 'plant', 'flower']
>>>**print** zet, empty, x
['tree', 'plant', 'flower'] [] [1, 2, 3, 4, 5, 6]

Επιπρόσθετα η *τροποποίηση(*mutation*)* των στοιχείων μιας λίστας είναι εφικτή:
#δημιουργία μιας λίστας
>>> y=['yes', 'no', [1,2,3], '10', 'k']
>>> **print** y
['yes', 'no', [1,2,3], '10', 'k']
#αλλαγή δυο στοιχείων σε μια λίστα
>>> y[0]= 'yesyes'
['yesyes', 'no', [1, 2, 3], '10', 'k']
>>> y[4]=3
['yesyes', 'no', [1, 2, 3], '10', 3]
#διαγραφή ενός στοιχείου
>>> **del**(y[4])
>>> **print** y
['yesyes', 'no', [1, 2, 3], '10']
#προσθήκη ενός νέου στοιχείου
>>> y.append(5)
>>> **print** y
['yesyes', 'no', [1, 2, 3], '10', 5]

Η τροποποίηση του στοιχείου 4 έγινε με την πρόταση y[4]=3 και από 'k' άλλαξε στον αριθμό 3. Μετά διαγράφτηκε το 3 με την εντολή del(), οπότε η λίστα προς στιγμή μίκρυνε κατά ένα στοιχείο και τέλος προστέθηκε ένα νέο στοιχείο με την εντολή append() . Επισημαίνεται ότι τα ελληνικά γράμματα δεν αναγνωρίζονται στην γλώσσα.

Ο γνωστός από παλιότερο κεφάλαιο τελεστής in μπορεί να χρησιμοποιηθεί για να αναζητήσουμε αν ένα στοιχείο ανήκει σε μια λίστα ή όχι. Έτσι, για την μεταβλητή y στον από πάνω κώδικα:
>>> 'no' **in** y
>>> True

 Copyright © 2014

5.2.1 Επεξεργασία λίστας και string

Επειδή οι λίστες διαδραματίζουν ένα σημαντικό ρόλο στην Python θα γίνει μια πιο εντατική παράθεση παραδειγμάτων με χρήση ενσωματωμένων(built-in) συναρτήσεων που στην Python καλούνται *μέθοδοι* και θα καλυφθούν σε άλλο κεφάλαιο παρακάτω. Μέχρι στιγμής έχει αποφευχθεί να χρησιμοποιείται ο όρος μέθοδοι. Ο κύριος σκοπός σε αυτή την παράγραφο είναι να δειχθούν μερικοί χρήσιμοι τρόποι για επεξεργασία λιστών αλλά και μετατροπές με λίστες και strings.

```
>>> w=[1 ,2, 3 ,4]+[9, 8, 5, 6]
>>> print w
[1 ,2, 3 ,4, 9, 8, 5, 6]
>>> w+[ '10', 'hot']
[1, 2, 3, 4, 9, 8, 5, 6, '10', 'hot']
>>> w+'hello'
Type Error: can only concatenate list (not "str") to list
```

Επιτρέπεται να ενώσουμε δυο ή περισσότερες λίστες σε μία αλλά δεν επιτρέπεται να ενώσουμε διαφορετικούς τύπους σε έναν. Η τεχνική slicing(τεμαχισμός) φυσικά ισχύει και μπορεί να εφαρμοστεί στις λίστες:

```
>>>x=[12, 4, 54, 6, 8, 7, 0, 9, 32]
>>> y=x[2:6]
>>> y
[ 54, 6, 8, 7, 0]
>>>h=x[2:]
>>> h
[54, 6, 8, 7, 0, 9, 32]
>>>z=x[-4:]
>>> z
[7, 0, 9, 32]
```

Το slicing είναι χρήσιμη τεχνική, υπάρχει και η συνάρτηση split() για χωρισμό strings και μπορεί να εφαρμοστεί όποτε χρειαστεί. Το επόμενο βήμα είναι να δειχθεί πως είναι δυνατόν να γίνει επεξεργασία ενός αλφαριθμητικού(string) με χρήση list(λίστας) και πως επίσης είναι δυνατόν να γίνει μετατροπή της λίστας και πάλι σε αλφαριθμητικό. Για το πρώτο σκέλος της πρότασης η απάντηση είναι με τη βοήθεια της συνάρτησης list() και για το δεύτερο σκέλος γίνεται με τη χρήση της join().

```
#string2list conversion
>>> word='big deal'
>>> msgtext =list(word)
>>> word
'big deal'
>>> msgtext
['b', 'i', 'g', ' ', 'd', 'e', 'a', 'l']
```

Συνεπώς κατορθώσαμε να μετατρέψουμε ένα αλφαριθμητικό σε λίστα και πλέον είναι πολύ εύκολο να το επεξεργαστούμε γράμμα προς γράμμα. Όταν θελήσουμε να το επαναφέρουμε στην αρχική του μορφή θα χρησιμοποιήσουμε μια επιπλέον βοηθητική μεταβλητή για να δράσει πάνω στην εντολή join() με ότι όνομα θέλουμε. Την βοηθητική μεταβλητή θα την βαφτίσουμε στο παρακάτω παράδειγμα 'glue', μπορούμε να την φανταστούμε σαν μια κόλλα που κολλά τα αυτόνομα γράμματα της λίστας σε ένα ενιαίο κείμενο.

```
>>> msgtext
['b', 'i', 'g', ' ', 'd', 'e', 'a', 'l']
>>> glue=''     #δυο μονοί απόστροφοι χωρίς κενό ανάμεσα τους για σωστά αποτελέσματα παρακάτω
>>> glue.join(msgtext)
'big deal'
```

Αν γράφαμε glue=' ' με ένα κενό χαρακτήρα τότε θα παίρναμε αποτέλεσμα: 'b i g   d e a l'. Στο κομμάτι κώδικα με σχόλιο '#string2list conversion' μπορούμε να αντικαταστήσουμε τα γράμματα της λέξης 'deal' με τα γράμματα ας πούμε της λέξης 'story' με τον εξής τρόπο:

```
>>> msgtext =list('story')
```

Παρόλα αυτά, στον κώδικα υπάρχει μια μικρή λεπτομέρεια που εύκολα μπορεί να δημιουργήσει προβλήματα αν δεν γίνει διαπίστωση της. Τι θα συμβεί αν αντί για την πρόταση 'big deal' είχαμε μια αλλαγή για κάποιο λόγο σε 'small deal' ; προφανώς θα είχαμε απώλεια λέξεων διότι το len('big')!=len('small') δηλαδή δεν συμφωνούν τα μήκη τους. Υπάρχουν διαφορετικοί τρόποι για να δοθεί λύση σε ένα τέτοιο πρόβλημα και θα παρουσιαστούν σε μορφή παραδείγματος με κώδικα για καλύτερη κατανόηση στην πράξη.

```
# α΄ τρόπος – απαλοιφή με εντολή del()
>>> msgtext=['b', 'i', 'g', ' ', 's', 't', 'o', 'r', 'y']
>>> msgtext[4:4]=list('hit ')
>>> msgtext
['b', 'i', 'g', ' ', 'h', 'i', 't', ' ', 's', 't', 'o', 'r', 'y']
>>> del msgtext[3:7] #  μοναδικός  κατ' εξαίρεση τρόπος γραφής της εντολής σε αυτή το μορφή.
>>> msgtext
>>> ['b', 'i', 'g', ' ', 's', 't', 'o', 'r', 'y']
```

Βλέπουμε ότι με την εντολή del σε αυτή το μορφή εγγραφής της μπορούμε να επιλέξουμε από ποιο σημείο θα αρχίσει και που θα τελειώσει η διαγραφή. Με την εντολή msgtext[4:4]=list('hit ') επιλέξαμε το σημείο που θα προστεθεί το κείμενο 'hit' σε μορφή λίστας.

```
#β' τρόπος – κενή λίστα []
>>> msgtext=['b', 'i', 'g', ' ', 's', 't', 'o', 'r', 'y']
>>> msgtext[4:4]=list('hit ')
>>> msgtext
['b', 'i', 'g', ' ', 'h', 'i', 't', ' ', 's', 't', 'o', 'r', 'y']
>>> msgtext[3:7]=[]
>>> msgtext
['b', 'i', 'g', ' ', 's', 't', 'o', 'r', 'y']
```

Θυμόμαστε ότι η κενή λίστα ορίζεται ως μια λίστα χωρίς περιεχόμενο ,οπότε αν την εφαρμόσουμε όπως παραπάνω απαλείφει την ανεπιθύμητη λέξη ή κείμενο. Υπάρχουν δυο ακόμα τρόποι, ο ένας με χρήση της γνωστής πλέον συνάρτησης replace() και τέλος με χρήση της συνάρτησης remove() .Μερικές ακόμα πρακτικές και χρήσιμες συναρτήσεις για εφαρμογή σε λίστες είναι η max(),min(),cmp(list1,list2) και len(), δηλαδή εύρεση μέγιστου, ελάχιστου, σύγκριση των στοιχείων δυο λιστών, μήκους(πλήθος τους στην προκείμενη περίπτωση) στοιχείων αντίστοιχα και που η χρήση τους είναι εύκολη.

```
>>> x= [0, -1, 100, 54, 32, 67]
>>> min(x)
>>> -1
>>> max(x)
100
>>> len(x)
6
>>> cmp(x,x)
0
```

Μια χρήσιμη παρατήρηση είναι να επισημανθεί η διαφοροποίηση της append() από την extend(), που παρότι εκτελούν την ίδια λειτουργία διαφοροποιούνται ως προς τα ορίσματα. Η μεν πρώτη δέχεται μόνο ένα όρισμα ενώ η δεύτερη παραπάνω του ενός. Να προστεθούν δυο λίστες (όχι αριθμητικά) με σκοπό να ενωθούν σε μία χρειάζεται ο τελεστής της πρόσθεσης, για να αναπαραχθεί πολλές φορές μια λίστα εφαρμόζουμε τον τελεστή του πολλαπλασιασμού:

```
>>> [0]*4
[0, 0, 0, 0]
```

```
>>> s=[0]
>>> s*4
[0, 0, 0, 0]
# το αποκάτω όμως είναι λάθος , δεν αποτελεί λίστα:
>>> s=0
>>>s*0
0   # το οποίο είναι ακέραιος και όχι λίστα λόγω του εσφαλμένου τρόπου που ορίστηκε το s.
```

5.2.2 Τυπική προσπέλαση βρόχου με λίστα

Αν και έχει γίνει έμμεσα παρουσίαση σε προηγούμενα κεφάλαια για αυτό το θέμα ,αξίζει να αφιερωθεί μια παράγραφος επιπλέον για να ξεκαθαριστούν τυχόν ελλείψεις. Το ακόλουθο παράδειγμα θα βασιστεί στην δοκιμασμένη και αποτελεσματική προσπέλαση λίστας με το συνδυασμό των εντολών **for – in.**

```
>>> for item in mylist:
>>>     print item
```

Αν επιθυμούμε να δούμε μόνο το δείκτη σε μια λίστα βασιζόμαστε στην συνάρτηση range():

```
>>> for index in range(len(mylist):
>>>     print index
```

Αυτή η προσέγγιση μας παρέχει ένα κώδικα γενικό και η *γενικότητα* είναι πολύ θεμιτή γιατί ο κώδικας μπορεί να βρει εφαρμογή για μεγάλο πεδίο εφαρμογών διότι είναι απαλλαγμένος από συγκεκριμένες τιμές που θα περιόριζαν την χρηστικότητα του και την αναπαραγωγή του σε άλλες μελλοντικές εφαρμογές.

Στην περίπτωση που χρειαζόμαστε τον *δείκτη* που δείχνει σε ένα στοιχείο <u>και</u> το *στοιχείο* της λίστας μπορούμε να χρησιμοποιήσουμε την συνάρτηση enumerate:

```
>>> for index, item in enumerate(mylist):
>>>     print item, index
```

Ένα ενδεικτικό παράδειγμα που δημιουργεί μια λίστα και βρίσκει την κυβική τιμή κάθε δείκτη(index) και θα αξιοποιήσει την δεύτερη περίπτωση είναι το ακόλουθο:

```
>>> cube=[ ]
>>> for index in range(9):
        cube.append(index**3)
>>> print cube
[0, 1, 8, 27, 64, 125, 216, 343, 512]
```

Πρώτα δημιουργήσαμε μια κενή λίστα με σκοπό να την γεμίσουμε στην πορεία. Μετά καταφύγαμε στην επαναληπτική δομή for για γέμισμα της λίστας και τέλος η εμφάνιση αποτελεσμάτων. Υπενθυμίζεται ότι ο τελεστής ** δηλώνει ύψωση σε δύναμη, το μέτρημα αρχίζει από το μηδενικό στοιχείο. Ο παραπάνω κώδικας μπορεί να συμπυκνωθεί σε μία γραμμή με τον εξής τρόπο:

```
>>> cube = [index**3 for index in range(9)]
[0, 1, 8, 27, 64, 125, 216, 343, 512]
```

Υπάρχουν αρκετές συναρτήσεις που μπορούν να εφαρμοστούν σε λίστες με βρόχους προκειμένου να απλοποιήσουν την επίλυση ενός προβλήματος. Οι συναρτήσεις μέθοδοι που είναι αρκετά πρακτικές απαριθμούνται σε δέκα και θα παρατεθούν παρακάτω σε μορφή πίνακα με τον όνομα έκαστης, μια σύντομη περιεκτική περιγραφή τι κάνει και ένα ενδεικτικό παράδειγμα εφαρμογή για να φανεί πως θα μπορούσαν να αξιοποιηθούν στην πράξη. Εννοείται ότι οι συναρτήσεις του πίνακα δεν είναι υποχρεωτικό να βρίσκουν εφαρμογή μόνο σε απλές λίστες, μπορούν να επεκταθούν και να αξιοποιηθούν σε ποιο περίπλοκες εφαρμογές.

Πίνακας 8. Χρήσιμες συναρτήσεις μέθοδοι για λίστες

μέθοδος	**Περιγραφή**	Παράδειγμα
append(y)	Προσθήκη στοιχείου στο τέλος μιας λίστας. Δέχεται μία παράμετρο.	mylist=[1,2, 'yes'] mylist.append(11.22) (έξοδος: mylist=[1,2, 'yes', 11.22])
extend(x)	Επέκταση λίστας όπως η append. Δέχεται πολλαπλές παραμέτρους.	face0=['ear', 'lips'] face1=['nose', 'nose'] face2=['hair'] face0,extend(face1,face2) (έξοδος: face0=['ear','lips', 'nose', 'nose', 'hair'])
pop([j])	Αφαίρεση στοιχείου από δεδομένη θέση λίστας, το τυπώνει επίσης. Το [j] δηλώνει ότι μπορεί να παραληφθεί.	number=[1,2,0,5] number.pop(0) (έξοδος: number=[2,0,5] και τυπώνει το 1)
remove(y)	Αφαιρεί το πρώτο σε αύξουσα σειρά y στοιχείο που συναντά στη λίστα.	number=[1, 5, 0, 5] number.remove(5) (έξοδος: number= [1, 0, 5])
count(x)	πόσες φορές εμφανίστηκε το στοιχείο x στην λίστα.	Num=[1,0,0,1,0,0,1] Num.count(1) (έξοδος: 3)
index(x)	Ποιος είναι ο δείκτης που αντιστοιχεί στην πρώτη σε σειρά εμφάνισης τιμή x	number=[2, 5, 0, 5] number.index(5) (έξοδος: 1)
insert(j,y)	Εισαγωγή στοιχείου y σε δεδομένη θέση στην λίστα (παρατήρηση:ισοδύναμα τα a.insert(len(a), x) και a.append(x))	num=[1, 2, 3, 4, 5] num.insert(2,0) (έξοδος: num=[1, 2, 0, 3, 4, 5])
sort()	Ταξινόμηση κατά αύξουσα σειρά. Χωρίς ορίσματα. Τυχόν κεφαλαία γράμματα ταξινομούνται πρώτα.	x=[1, 5, 4, 3, 0, 2] x.sort() (έξοδος: x=[0, 1, 2, 3, 4, 5])
reverse()	Αντιστροφή στην ίδια λίστα. Χωρίς ορίσματα.	x=[1, 5, 4, 3, 0, 2] x.reverse() (έξοδος: x=[5, 4, 3, 2, 1, 0])
sorted(x)	Ίδια λειτουργία με την sort(). Παίρνει όρισμα. Τυχόν κεφαλαία γράμματα ταξινομούνται πρώτα.	x=[1, 5, 4, 3, 0, 2] sorted(x) (έξοδος: x=[0, 1, 2, 3, 4, 5])

5.2.3 Στοίβα και ουρά

Με μια λίστα μπορούμε να φτιάξουμε στοίβα ,από όπου το τελευταίο στοιχείο που τοποθετείται σε μια λίστα είναι αυτό που ανακτάται πρώτο, δηλαδή έχουμε την λογική lifo(last in-first out). Στην κορυφή μιας στοίβας τοποθετούμε ένα στοιχείο με την βοήθεια της append και το ανακτούμε με την βοήθεια της pop.

```
>>> push=[1, 2, 3]
>>> push.append(10)
>>> push.append(11)
```

```
>>> push.append(7)
>>> print push
[1, 2, 3, 10, 11, 7]
>>> push.pop()
7
>>> push.pop()
11
>>> push.pop()
10
```

Πέραν της στοίβας μπορούμε να αξιοποιήσουμε μια λίστα και στην περίπτωση ουράς, δηλαδή η λίστα να λειτουργεί ως μια ουρά 'αναμονής'. Το πρώτο στοιχείο που εισάγεται θα είναι και το πρώτο που θα εξάγεται, δηλαδή λογική fifo(first in-first out). Θα επισημανθεί ότι οι λίστες δεν είναι πολύ αποδοτικές σε σχέση με την στοίβα όταν τις χρησιμοποιούμε για δημιουργία ουράς εξαιτίας του τρόπου που λειτουργεί μια ουρά, για κάθε νέο στοιχείο όλα τα προηγούμενα πρέπει να μετακινούνται κατά μία θέση κάτι που επιβραδύνει την όλη διαδικασία από άποψη ταχύτητας. Στην πράξη για να κατασκευάσουμε μια ουρά θα βασιστούμε σε μια ειδική συνάρτηση την deque που βρίσκεται στο module collections. Η deque έχει σχεδιαστεί για όσο γίνεται πιο γρήγορα appends και pops και από τις δυο άκρες μιας ουράς.

```
>>> from collections import deque
>>> queue= deque(['apples', 'oranges', 'kiwi'])
>>> queue.append('melon')
>>> queue.append('apricot')
>>> queue.append('strawberries')
>>> queue.popleft()
'apples'
>>> queue.popleft()
'oranges'
>>> queue
deque(['kiwi', 'melon', 'apricot', 'strawberries'])
```

5.2.4 Πίνακες με φωλιασμένες λίστες

Είδαμε πώς να φτιάχνουμε μια απλή λίστα και μια απλή φωλιασμένη λίστα. Αν κάθε στοιχείο μιας λίστας είναι φωλιασμένη λίστα τότε μιλάμε για πίνακα διαστάσεων MxN με a_{ij} στοιχεία.

```
Matrix = [
          [a11, a12, a13],
          [a21, a22, a23],
          [a31, a32, a33],
          ]
```

Αυτός είναι ένας πίνακας 3x3,αλλά μπορούμε να κατασκευάσουμε οποιαδήποτε διάσταση επιθυμούμε. Ο συγκεκριμένος πίνακας αν πληκτρολογήσουμε Matrix θα εμφανίσει σε μια γραμμή όλο τον πίνακα όπως μια απλή λίστα. Η συγκεκριμένη λίστα αποτελείται από 3 στοιχεία με δείκτες από το 0 έως το 2.Επίσης, οι στήλες γίνονται γραμμές με τον εξής τρόπο:

```
>>> antimeta8esi = []
>>> for index in range(N):
>>>     andimeta8esi_grammis = []
>>>     for row in matrix:
>>>         andimeta8esi_grammis.append(row[index])
>>>     antimeta8esi.append(andimeta8esi_grammis)
 # ποιο απλά σε μία γραμμή μόνο:
>>> [[row[i] for row in matrix] for i in range(N)]
```

# ακόμα ποιο απλά με χρήση της συνάρτησης zip():
>>> zip(*Matrix) # το αστεράκι υποδηλώνει ότι η μεταβλητή Matrix είναι λίστα

Με ίδιο αποτέλεσμα σε κάθε μια από τις τρεις άνωθεν περιπτώσεις και ίσο με:

```
[
[a11, a21, a31],
[a12, a22, a32 ],
[a13, a23, a33],
]
```

5.3 Tuples

Μια πλειάδα(tuple) είναι μια *σταθερού μήκους*(immutable)ακολουθία από τιμές. Διαθέτει δείκτες όπως και οι λίστες. Το ότι είναι immutable αποτελεί την ειδοποιό διαφορά από τις λίστες. Ενώ στις λίστες χρησιμοποιούμε αγκύλες στις πλειάδες χρησιμοποιούμε παρενθέσεις χωρίς να είναι καταναγκαστικό.

```
>>> t=1, 2, 3, 4, 5          α' τρόπος
>>> t=(1, 2, 3, 4, 5)        β' τρόπος
>>> t=()                     κενή
```

Πλειάδα με ένα στοιχείο δημιουργούμε ως εξής:

```
>>> t=1,      είναι πλειάδα
>>> t= ('1',) είναι πλειάδα
>>> tt=1      δεν είναι πλειάδα
>>> tt= ('1') δεν είναι πλειάδα
```

Όταν έχουμε αμφιβολίες η εντολή type() μπορεί να ξεκαθαρίζει το τοπίο:

```
>>> type(t)
<type 'tuple'>
>>> type(tt)
<type 'int'>
```

Για να μην έχουμε ποτέ αμφιβολίες μπορούμε να βασιστούμε στην ενσωματωμένη συνάρτηση tuple():

```
>>> x=tuple('mystring')
>>> x
('m', 'y', 's', 't', 'r', 'i', 'n', 'g')
```

Παράδειγμα με διάφορες απλές πράξεις και απλοί τρόποι χρήσης:

```
>>> mixed = [(1, 'one'), (2, 'two'), (3, 'three')]  # Μια list με 3 tuples
>>> mixedd = ([1, 'one'], [2, 'two'], [3, 'three'])  # Ένα tuple με 3 lists
>>> nested = ((1, 'one'), (2, 'two'), (3, 'three'))   # Μια tuple  με 3 φωλιασμένες tuples
>>> print mixedd[1]
[2, 'two']
>>> extra=('a', 'b', 'c')
>>> newtuple=mixedd+extra
>>> print newtuple
([1, 'one'], [2, 'two'], [3, 'three'], 'a', 'b', 'c')
>>> print newtuple[2:5]
([3, 'three'], 'a', 'b')
>>> newtuple[4]= '3E'
TypeError: 'tuple' object does not support item assignment
```

Αναθέσαμε μια tuple στην μεταβλητή mixed,τυπώσαμε το στοιχείο που αντιστοιχεί στο δείκτη ένα, ενώσαμε δυο tuples σε μία, ξανατυπώσαμε ένα μέρος μόνο του νέου tuple και δοκιμάσαμε ανεπιτυχώς να αλλάξουμε ένα στοιχείο, αλλά δεν επιτρέπονται αλλαγές στοιχείων εξ ορισμού, διότι

είναι *αδιαίρετη*(immutable). Ένας τρόπος αξιοποίησης μιας πλειάδας στην πράξη είναι ως μια ετερογενή ακολουθία στοιχείων που τα προσπελάζουμε με μια τακτική ανοίγματος πλειάδας(tuple unpacking) που θα δειχθεί, ένας άλλος τρόπος είναι να εκμεταλλευτούμε μια πλειάδα για δεικτοδότηση(indexing), δηλαδή να φανταστούμε την tuple σαν μια δεξαμενή από δείκτες που ο κάθε ένας δείχνει σε κάποιο περιεχόμενο.

```
>>>     #tuple packing παράδειγμα
>>> salutations = 'bye' , 'hi' , 'greetings', 'good day'  #εδώ γίνεται tuple packing
>>> a,b,c,d = salutations  # αριστερό μέλος οι μεταβλητές, δεξί οι τιμές(tuple,list,string): x,y=f,z
# tuple  unpacking επιλέγοντας στοιχεία
>>> print b
'hi'
>>> d   # επιτρέπεται να παραλειφθεί το print.
'good day'
```

5.3.1 Σύνολα

Αν και τα σύνολα(sets) είναι από μόνα τους μια ξεχωριστή κατηγορία τύπου δεδομένων θα παρουσιαστούν εδώ επειδή μοιάζουν πάρα πολύ με τις λίστες και πλειάδες . Ένα σύνολο δημιουργείται με την συνάρτηση set() ή να πληκτρολογήσουμε { } ένα από τα δυο αρκεί.

```
>>> Furniture = ['sofa', 'table', 'chair', 'kitchenette', 'fridge', 'bed']
>>> house = set(Furniture)
>>> print house
set(['sofa', 'fridge', 'bed', 'kitchenette', 'table', 'chair'])
# εντοπισμός στοιχείου αν υπάρχει με τον τελεστή in
>>> 'sofa' in Furniture
True
>>> 'sofa' in house
True
>>> 'desk' in house
False
```

Οι ιδιότητες με τα σύνολα θα παρουσιαστούν με τη μορφή παραδείγματος.

```
>>> cat = set('Hungarian')
>>> dog = set('Phillipines')
>>> cat | dog
set(['a', 'e', 'g', 'P', 'i', 'H', 'l', 'n', 'p', 's', 'r', 'u', 'h'])
>>> cat-dog
set(['a', 'H', 'r', 'u', 'g'])
>>> cat^dog
set(['a', 'e', 'g', 'P', 'h', 'l', 'p', 's', 'r', 'u', 'H'])
>>> cat&dog
set(['i', 'n'])
```

Μερικές επεξηγήσεις θα διαλευκάνουν τι συμβαίνει. Το cat|dog λέει να φτιαχτεί ένα σύνολο από τα γράμματα που απαντώνται είτε στο cat είτε στο dog. Το επόμενο, cat-dog ζητά ένα σύνολο από όλα τα γράμματα που υπάρχουν μόνο στο cat. Παρακάτω το cat^dog αποφέρει σύνολο με γράμματα που δεν είναι και στα δυο και τέλος το cat&dog παράγει την τομή των δυο.

Μιας και έγινε πριν λόγος για τα άγκιστρα { } μπορούμε να γράψουμε σύνολα με αυτά με τον εξής τρόπο:

```
>>> cat = { y for y in 'Hungarian' if y not in 'Hng' }
>>> cat
set(['a', 'i', 'r', 'u'])
```

5.3.2 Ο τελεστής * στην πλειάδα

Το σύμβολο του πολλαπλασιασμού εκτός από πολλαπλασιασμό μπορεί να χρησιμοποιηθεί σε μια πλειάδα για να επιτελέσει διαφορετικό σκοπό, υπερφόρτωση τελεστή. Έτσι, αν το αστεράκι * προηγείται μιας μεταβλητής στις παραμέτρους μιας συνάρτησης, τότε επισημαίνει ότι αυτή η μεταβλητή πρέπει να αντιμετωπίζεται ως πλειάδα. Πράγματι,

```
>>> def paintball(*balls):
>>>     print balls
>>> paintball('red', 'black', 'pink', 'white')
('red', 'black', 'pink', 'white')
```

Ορίσαμε μια συνάρτηση, την paintball, με μία παράμετρο, την πλειάδα balls και ζητήσαμε απλώς να τυπώνει την είσοδο. Με αυτό το μοτίβο ορισμού συνάρτησης δε δώσαμε συγκεκριμένες τιμές, προτιμήσαμε την γενίκευση, την αφαιρετικότητα. Κατόπιν καλέσαμε την συνάρτηση και ότι συγκεκριμένες τιμές ορίσματα δώσουμε γνωρίζει από πριν ο διερμηνευτής ότι θα τα αντιμετωπίσει ως πλειάδα. Η όλη παραπάνω διαδικασία μπορεί να ονομαστεί *συλλογή*. Η συμπληρωματική της διαδικασία θα ονομαστεί *διασπορά*. Οπότε, εάν έχω μια ακολουθία από τιμές και θέλω να τις περάσω ως όρισμα σε μια συνάρτηση μπορώ να χρησιμοποιήσω το αστεράκι * .

```
>>> x = (2,3)
>>> divmod(*x)
(0,2)
# χωρίς την διασπορά θα υπήρχε πρόβλημα όπως φαίνεται από κάτω.
>>> divmod(x)
TypeError: divmod expected 2 arguments, got 1
```

5.3.3 Βρόχος με πλειάδες μέσα σε λίστα

Για την επεξήγηση αυτής της ενότητας θα παρατεθεί πρώτα ο κώδικας και μετά η ανάλυση του.

```
>>> days=('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')
>>> daynumbers = ('1', '2', '3', '4', '5', '6', '7')
>>> week=zip(days,daynumbers)
>>> print week
[('Monday', 1), ('Tuesday', '2'), ('Wednesday', '3'), ('Thursday', '4'), ('Friday', '5'), ('Saturday', '6'),
('Sunday', '7')]
>>> for day, daynumber in week:
>>>     print daynumber,'- ' + day
```

Αποτέλεσμα:
'1' - Monday
'2' - Tuesday
'3' - Wednesday
'4' - Thursday
'5' - Friday
'6' - Saturday
'7' - Sunday

Πριν ξεκινήσει η ανάλυση να τονίστει ότι το 1 είναι αριθμός ενώ το '1' είναι τύπος string και αυτό έχει σημασία γιατί το 1 είναι τύπος int, ενώ το '1' δεν είναι. Ορίστηκαν δυο πλειάδες η days και η daynumbers, μετά έγινε χρήση της συνάρτησης zip() με σκοπό να κατασκευαστεί μια λίστα με περιεχόμενα πλειάδες με αντιστοίχιση ένα προς ένα από κάθε μια πλειάδα. Προέκυψαν επτά πλειάδες στο σύνολο, με την ιδιότητα κάθε μια μέρα της πλειάδας days να ζευγαρώνει με έναν μόνο αριθμό, η σάρωση γίνεται από αριστερά προς τα δεξιά γι αυτό το ένα αντιστοιχεί στην Monday, το δυο

στην Tuesday και ούτω κάθε εξής. Κατόπιν έγινε χρήση της επαναληπτικής μεθόδου **for-in.** Επειδή επιθυμούμε δυο αποτελέσματα και επειδή μιλάμε για πλειάδες μέσα σε μια λίστα διαφοροποιείται η λύση, δε χρησιμοποιήσαμε την enumerate όπως έγινε στον βρόχο με λίστα, εδώ ακολουθήθηκε μια διαφορετική προσέγγιση. Εάν θέλαμε να εμφανίσουμε τις μέρες και τους αριθμητικούς δείκτες που αντιστοιχούν σε κάθε μια μέρα θα χρησιμοποιούσαμε την εντολή enumerate όπως στον βρόχο με λίστα. Εάν θέλουμε να κάνουμε multitasking(πολλαπλή εργασία) , δηλαδή να προσπελάσουμε ταυτόχρονα δυο ή και περισσότερες ακολουθίες τότε ένας κώδικας που μπορεί να το πετύχει αυτό μπορεί να βασιστεί σε ένα συνδυασμό της zip, πλειάδας και στον σχηματισμό for-in, όποτε x1[k]==x2[k] συνεπάγεται True κατάσταση. Για παράδειγμα:

```
>>> for i, j in zip(x1,x2):
        if i==j:
            print "True"
      return False
```

5.4 Dictionary

Ο γενικός τύπος ενός λεξικού είναι:
Όνομα_μεταβλητής={κλειδί_1:τιμή_1, κλειδί_2:τιμή_2,..., κλειδί_Ν:τιμή_Ν}

Ένα λεξικό(dictionary) μοιάζει όπως μια λίστα αλλά διαφέρει κάπως από αυτήν. Ενώ στην λίστα οι δείκτες είναι ακέραιοι, στο λεξικό δεν υπάρχει περιορισμός σε ένα τύπο δεδομένων, μπορούν τα λεξικά να φέρουν σχεδόν όλους τους τύπους δεδομένων. Ένα λεξικό αποτελείται από *κλειδιά* (ή αλλιώς θέσεις) που αντιστοιχούν σε *τιμές*(τιμή κλειδιού) που μπορεί να είναι οποιουδήποτε τύπου δεδομένων. Ένα ζευγάρι κλειδί-τιμή μπορούμε να το ονομάσουμε *αντικείμενο-στοιχείο.* Επιπρόσθετα από τεχνικής άποψης ταξινόμηση γίνεται αυτόματα με κριτήρια του κατασκευαστή της γλώσσας, δεν μπορούμε να το κάνουμε εμείς, οπότε η σειρά εισαγωγής των δεδομένων μπορεί να διαφέρει από αυτό που θα εμφανιστεί στην οθόνη. Άλλωστε άσχετα πως θα αυτοταξινομηθούν τα αντικείμενα-στοιχεία δε θα πρέπει να ανησυχούμε διότι με τα κλειδιά εντοπίζουμε ότι χρειαζόμαστε. Ένα απλό παράδειγμα για το πως χτίζεται ένα λεξικό φαίνεται παρακάτω:

```
>>> family={'father':'Bob','mother':'Tonia','sister':'katrin','brother':'Joe'}
```

Μερικά ακόμα απλά παραδείγματα τρόπου χρήσης του λεξικού:

```
>>> family['mother']
>>> Tonia
>>> myfunction=dict()
>>> print myfunction
{}        # άρα η συνάρτηση dict() δημιουργεί ένα κενό λεξικό
>>> myfunction['one']=1
>>> myfunction['two']=2
>>> print myfunction
{'two': 2, 'one': 1}
>>>c=myfunction.copy()  # c και myfunction είναι ίδια πλέον. Με την myfunction.clear() διαγραφή.
```

5.4.1 Συνάρτηση και λεξικό

Το λεξικό με χρήση συνάρτησης προκύπτει ως εξής:

```
def myfamily(x):
    return family[x]
```

Μετά στο Python Shell γράφουμε:

```
>>> myfamily('father')
'Bob'
>>> myfamily('brother')
'Joe'
```

 Copyright © 2014

Η εντολή len() δίνει τον αριθμό των ζευγαριών κλειδί-τιμή, ο τελεστής **in** μας ενημερώνει αν ένα μέγεθος υπάρχει ως κλειδί λεξικού, γιατί η γνώση μόνο τις τιμής δεν επαρκεί. Συνεπώς προσπαθούμε να επαληθεύσουμε την ύπαρξη κλειδιών. Αν θέλουμε να αναζητήσουμε τιμές τότε θα χρησιμοποιήσουμε την μέθοδο values. Παρακάτω δίνεται ένας απλός κώδικας που καλύπτει τις δυο προαναφερθείσες περιπτώσεις αναζήτησης ύπαρξης κάποιου κλειδιού ή τιμής σε ένα λεξικό.

```
def Deutch2EnglishTranlator(x):
    De2En={'Montag':'Monday', 'Dienstag':'Tuesday', 'Mittwoch':'Wednesday', 'Donnerstag':
'Thurday', 'Freitag':'Friday', 'Samstag':'Saturday', 'Sonntag':'Sunday'}
# αναζήτηση κλειδιού:
>>> 'Sunday' in De2En
False  # προσοχή το Sunday δεν είναι κλειδί αλλά τιμή
>>> 'Sonntag' in De2En
True
# αναζήτηση τιμής:
>>> lookvalue=De2En.values()
>>> 'Sunday' in lookvalue
True
```

Για να δηλώσουμε όμως μια **συνάρτηση ως όρισμα τιμή** μέσα στο λεξικό η διαδικασία διαφοροποιείται. Ένας τρόπος για να πραγματοποιηθεί αυτό είναι με την εξής ακόλουθη μέθοδο(δίχως να σημαίνει ότι δεν υπάρχει και άλλη):

```
def testfunction():
        print 'function as a parameter in a dictionary'

family={'father':'Bob','mother':'Tonia','sister':'katrin','brother':'Joe', 'other': testfunction}
def main(arguments):
        family[arguments]()
```

Σώζουμε τον κώδικα σε ένα script αρχείο και για να δοκιμάσουμε αν πέτυχε πληκτρολογούμε στο Python Shell:

```
>>> main('other')
function as a parameter in a dictionary
```

Αυτό που έγινε είναι ότι περάσαμε μια συνάρτηση σε ένα λεξικό ως τιμή που αντιστοιχεί στο κλειδί 'other'. Η χρησιμότητα του κώδικα είναι μεγάλη γιατί μπορούμε να φτιάξουμε ένα λεξικό με δεκάδες συναρτήσεις και κλειδιά που θα επιλύουν σύνθετα προβλήματα.

5.4.2 Εφαρμογή: μέτρημα στοιχείων

Ένας τρόπος για να μετρήσουμε στοιχεία είναι με μετρητές και πολλές εντολές if μία για κάθε στοιχείο που θα μετρηθεί. Αν υποθέσουμε ότι θέλουμε να μετρήσουμε σε ένα αλφαριθμητικό πόσες φορές εμφανίζεται ο κάθε χαρακτήρας τότε για τα 26 γράμματα του αγγλικού αλφάβητου θα χρειαζόμασταν 26 μετρητές και συνεπώς 26 εντολές if! Φυσικά υπάρχουν και άλλοι τρόποι και μια άλλη προσέγγιση σε μια τέτοια περίπτωση θα μπορούσε να γίνει με χρήση λεξικού. Με το λεξικό μπορούμε να ορίσουμε ως κλειδιά τα 26 γράμματα του αλφάβητου και ως τιμές τους μετρητές αντίστοιχα. Οπότε με ένα λεξικό που θα το αναθέσουμε σε μια μεταβλητή απλοποιούμε τον κώδικα και αποφεύγουμε ένα πρόγραμμα spaghetti .

```
def apari8misi_string(mystring):
   var=dict()
   for character in mystring:
      if character not in var:
         var[character]=1 # νέα εμφάνιση, δημιουργία νέας θέσης στο λεξικό
      else:
         var[character]= var[character] +1
   return var
```

Αν θέλουμε να μετρήσουμε μια γραμμή ολόκληρη μπορούμε να προσθέσουμε μεταξύ της var=dict() και της for :
while mystring!= '\n'.
Σε αυτόν τον κώδικα φτιάξαμε πρώτα ένα κενό λεξικό με την εντολή dict() και κατόπιν με την εντολή for σαρώνουμε το αλφαριθμητικό(αν μιλάγαμε για γραμμή τότε θα σαρώναμε τα αλφαριθμητικά της εκάστοτε γραμμής). Σε κάθε πέρασμα του βρόχου αν ο χαρακτήρας που θα συναντήσει η εντολή for δεν υπάρχει στο string τότε ορίσαμε να δημιουργείται μια νέα θέση στο λεξικό και να αρχίσει να υπολογίζεται, αυτό έγινε με την εντολή var[character]=1. Αν ήδη υπάρχει απλώς αυξάνει τον ήδη υπάρχον μετρητή. Με αυτή την τεχνική δε χρειάζεται να γνωρίζουμε εκ των προτέρων τα γράμματα που περιέχει ένα string. Για επαλήθευση του κώδικα τον καλούμε στο Python Shell:

```
>>> saveme=apari8misi_string('skoulikomirmigkotrypa')
>>> print saveme
{'a': 1, 'g': 1, 'i': 3, 'k': 3, 'm': 2, 'l': 1, 'o': 3, 'p': 1, 's': 1, 'r': 2, 'u': 1, 't': 1, 'y': 1}
```

5.4.3 Αναζήτηση κλειδιών σε λεξικό

Όταν χρειαζόμαστε να βρούμε μια τιμή γνωρίζοντας το κλειδί είναι εύκολο να γίνει όπως έχει δειχθεί. Αν όμως δίνεται μια τιμή και ζητάμε να βρούμε σε ποιο κλειδί αντιστοιχεί και επειδή είναι δυνατόν μια τιμή να αντιστοιχηθεί σε πολλά κλειδιά η απάντηση δεν είναι προφανής. Παρακάτω δίνεται ένας απλός τρόπος εύρεσης κλειδιού και συγκεκριμένα του πρώτου που θα συναντηθεί σε ένα λεξικό:

```
def findKey(dictionary,value):
    for key in dictionary:
        if dictionary[key]==value:
            return key
    raise ValueError
```

Η value error είναι μια ενσωματωμένη εξαίρεση(exception) στην Python μέσα στο module exceptions,παράγει σφάλμα τιμής υποθέτοντας ότι ο τύπος είναι σωστός. Με την εντολή raise είναι που προκαλείται την εξαίρεση(force an error). Ο αλγόριθμος είναι κατασκευασμένος να σαρώνει έως και όλο το λεξικό, να εντοπίζει και να τυπώνει που βρίσκεται το πρώτο κλειδί που θα συναντήσει ακόμη κι αν υπάρχουν και άλλα σε μετέπειτα θέσεις με την ίδια τιμή. Αν δεν υπάρχει αποτέλεσμα τότε παίρνουμε ένα μήνυμα λάθους ValueError, που σημαίνει ότι δεν βρέθηκε ή δεν υπάρχει. Με παράδειγμα αξιοποιώντας τον κώδικα της προηγούμενης ενότητας:

```
# Από όλα τα γράμματα που τελούν ρόλο key και αντιστοιχούν σε τιμή 3 τυπώνει το πρώτο που συναντά
#σαρώνοντας από αριστερά προς τα δεξιά. Το 3 αντιπροσωπεύει ότι τρεις φορές εμφανίστηκε ένα
#γράμμα στην λέξη εισόδου.
>>> apari8misi=apari8misi_string('skoulikomirmigkotrypa')
>>> keyposition=findKey(apari8misi,3)
>>> print keyposition
'i'
```

5.4.4 Ορισμός λίστας ως τιμή σε λεξικό

Στην προηγούμενη ενότητα είδαμε πως αναζητούμε κλειδιά όταν γνωρίζουμε τις τιμές. Εδώ τώρα θα δειχθεί ένας τρόπος να ενσωματώνουμε λίστα ως τιμή σε ένα λεξικό αναπαράγοντας την ίδια λογική με προηγουμένως ως προς το σκεπτικό ανάπτυξης του κώδικα με κάποιες επιπλέον προσθήκες.

Στην επόμενη σελίδα θα παρουσιαστεί μια προτεινόμενη λύση. Ας θεωρήσουμε τον αλγόριθμο apari8misi όπως τον είδαμε προηγουμένως. Σε αυτόν τον κώδικα μετρούσαμε το κάθε κλειδί που ήταν ένα γράμμα πόσες φορές(τιμή κλειδιού στο λεξικό) εμφανιζόταν σε μια λέξη, οπότε τώρα θα κάνουμε το ανάποδο χρησιμοποιώντας μια λίστα για να μπορέσει να δουλέψει.

```
def andistrofi_lexicon(mydictionary):
    andistrofi_lexiko = dict() # δημιουργία μιας κενής λίστας προς γέμισμα πρώτα απ’ όλα.
    for dkey in mydictionary:
        myvalue = mydictionary[dkey]
        if myvalue not in andistrofi_lexiko:
            andistrofi_lexiko[myvalue] = [dkey]
        else:
            andistrofi_lexiko[myvalue].append(dkey)
    return andistrofi_lexiko
```

Κάθε φορά που διατρέχεται ο βρόχος με την εντολή for το εκάστοτε στοιχείο dkey παίρνει ένα κλειδί(λεξικού) από το mydictionary και το myvalue την αντίστοιχη τιμή. Σε περίπτωση που το myvalue δεν έχει ξαναεμφανιστεί(άρα δεν είναι αντίστροφο) δημιουργείται μια λίστα και του ανατίθεται μια τιμή, αν το έχουμε ξανασυναντήσει τότε απλά χρησιμοποιούμε την εντολή append για αντιστοίχηση του αντίστοιχου κλειδιού(λεξικού) στην λίστα. Πιο απλά, O αλγόριθμος δημιουργεί ένα λεξικό με αριθμούς ως κλειδιά και σε κάθε αριθμό κλειδί αντιστοιχεί όλα τα γράμματα μιας λέξης. Έτσι, το ένα θα είναι ένας αριθμός κλειδί στο λεξικό μας που δείχνει ακριβώς ποια γράμματα εμφανίζονται μία φορά σε ένα αλφαριθμητικό, το δυο σημαίνει ότι κάποια γράμματα εμφανίζονται δυο φορές στο αλφαριθμητικό ,το τρία τρεις φορές και ούτω κάθε εξής. Με άλλα λόγια, αν στην ενότητα 5.4.2 γινόταν απαρίθμηση στοιχείων σε αυτή εδώ την ενότητα γίνεται το αντίστροφο. Για του λόγου του αληθές:

```
>>> saveme=apari8misi_string('skoulikomirmigkotrypa')
>>> print saveme
{'a': 1, 'g': 1, 'i': 3, 'k': 3, 'm': 2, 'l': 1, 'o': 3, 'p': 1, 's': 1, 'r': 2, 'u': 1, 't': 1, 'y': 1}
>>> myandistrofi = andistrofi_lexicon('skoulikomirmigkotrypa')
>>> print myandistrofi
{1: ['a', 'g', 'l', 'p', 's', 'u', 't', 'y'], 2:[ 'm', 'r'], 3:[ 'i', 'k', 'o'] }
```

5.4.5 Hash table στην Python

Θα πρέπει να προσέξουμε ότι **οι λίστες δεν μπορούν να οριστούν ως κλειδιά** σε λεξικό. Αν δοκιμάσουμε να ορίσουμε μια λίστα ως κλειδί σε ένα λεξικό τότε θα προκύψει σφάλμα: TypeError: list objects are unhashable.

Το hash(κατακερματισμός) μπορούμε να το σκεπτόμαστε ως μια συνάρτηση που παίρνει μια τιμή οποιουδήποτε τύπου και επιστρέφει έναν ακέραιο. Κατόπιν το λεξικό χρησιμοποιεί αυτόν τον ακέραιο, δηλαδή την *hash τιμή* ,για να αποθηκεύσει και να αναζητήσει ζευγάρια κλειδιών-τιμών σε ένα λεξικό. Επειδή οι λίστες δεν είναι σταθερού μήκους, μπορούν να αυξομειώνονται και αυτό δημιουργεί επιπλοκές. Συνεπώς, όταν δημιουργούμε ένα ζευγάρι κλειδί-τιμή σε λεξικό η γλώσσα κάνει hash το κλειδί, κατόπιν το αποθηκεύει (το κλειδί) σε μια τοποθεσία, δηλαδή αντιστοιχεί το κλειδί σε έναν ακέραιο. Αν τροποποιήσουμε το κλειδί και το ξανακάνουμε hash(νέος ακέραιος) πολύ πιθανόν να αποθηκευτεί σε διαφορετική τοποθεσία. Αυτό σημαίνει ότι μπορεί να έχουμε δυο εγγραφές για το ίδιο κλειδί που θα προκαλέσει conflict(επιπλοκή) προφανώς. Ακόμα χειρότερα, να μην μπορέσει καν να εντοπίσει το κλειδί. Για αυτό το λόγω το κλειδιά πρέπει να είναι *hashable*(που σημαίνει να μπορούμε να χρησιμοποιούμε hash table), ενώ οι λίστες δεν είναι διότι είναι *mutable*(μη σταθερού μήκους).Μπορούμε βέβαια να χρησιμοποιήσουμε πλειάδες αντί για λίστες διότι οι πλειάδες είναι σταθερού μήκους. Η ουσία είναι πως ότι είναι mutable δεν μπορεί να χρησιμοποιηθεί ως κλειδί σε λεξικό, αυτό συμπεριλαμβάνει και τα ίδια τα λεξικά που είναι επίσης mutable,αλλά επιτρέπεται να χρησιμοποιούνται ως τιμές (που αντιστοιχούν σε κλειδιά).

Το Hash table(πίνακας κατακερματισμού) είναι μια ενσωματωμένη δομή δεδομένων στην γλώσσα Python για αποθήκευση συνόλων από στοιχεία. Ουσιαστικά, αυτό που ονομάζεται στην Python λεξικό ή με αλλά λόγια πίνακας συσχετισμού(associative array) είναι ένα hash table στην ορολογία των επιστημόνων της

πληροφορικής, δηλαδή το λεξικό στην Python είναι μια υλοποίηση ενός hash table. Έτσι, ένας απλός πίνακας δεικτοδοτεί(index) τα στοιχεία(elements) του χρησιμοποιώντας ακέραιους. Εντούτοις, αν επιθυμούμε να αποθηκεύσουμε δεδομένα με διαφορετικό χαρακτηριστικό τότε ο πίνακας δεν εξυπηρετεί και χρειάζεται μια κατάλληλη δομή δεδομένων. Για παράδειγμα αν θέλω να πληκτρολογώ το ονοματεπώνυμο των φοιτητών και να παίρνω έξοδο τον αριθμό μητρώο τους δεν βολεύει με απλό πίνακα αλλά με λεξικό βολεύει πολύ. Το πλεονέκτημα του hash table είναι ότι χρειάζεται τον ίδιο χρόνο για εισαγωγή, αναζήτηση και διαγραφή στοιχείων άσχετα πόσο μεγάλος είναι το hash table (το λεξικό στην Python με χρήση του συνδυασμού in-for για βρόχο επανάληψης).Αυτό δε συμβαίνει με τις λίστες που όσο μεγαλώνουν σε έκταση τόσο ποιο πολύ ώρα χρειάζονται για εντοπισμό στοιχείων και προσπέλαση τους.

Η συνάρτηση κατακερματισμού ορίζει έναν αριθμό μοναδικό που χρησιμοποιείται για θέση αποθήκευσης ενός στοιχείου στο πίνακα. Παρόλα αυτά, δεν είναι ένα προς ένα που σημαίνει ότι σε ίσα στοιχεία ανατίθεται πάντοτε ο ίδιος κωδικός ή υπάρχει περίπτωση σε hash table διαφορετικά στοιχεία να έχουν ίδιο κωδικό δηλαδή να έχουμε collision αλλά λύνονται τέτοια προβλήματα με linked lists(συνδεόμενες λίστες). Όλα τα hash tables δεικτοδοτούνται από immutable δεδομένα όπως έχει προαναφερθεί.

Αν και έχουν προαναφερθεί αν όχι άμεσα τουλάχιστον έμμεσα σε προηγούμενη ενότητα, δυο δυνατοί τρόποι προσπέλασης hash table είναι:

A)
```
>>> stdID={'student_A':1010, 'student_B',2100, ' student_C':1503} # (μαθητής : αριθμός μητρώου)
>>> for k in stdID.keys():
        print stdID[k]   # τυπώνει τον αριθμό που σχετίζεται με κάθε κλειδί
```

B)
```
>>> for key,value in stdID.items():
        print key,":", value      # τυπώνει κάθε ζευγάρι κλειδί-τιμή
```

 Copyright © 2014

Κεφάλαιο

6

Θέματα από Python STL και methods

Σε αυτό το κεφάλαιο θα γίνει μια ποιο σαφής διατύπωση του τι είναι methods. Σε προηγούμενα κεφάλαια κάθε φορά που χρησιμοποιήθηκαν είχαν χαρακτηριστεί με το προσωπείο της λέξης συνάρτηση. Στην πραγματικότητα ναι μεν είναι συναρτήσεις αλλά διαφοροποιούνται ελαφρώς από τις κλασικές συναρτήσεις. Επίσης θα γίνει μια αναφορά σε χρήσιμα θέματα αλλά όχι εκτενής παρουσίαση ολόκληρης της στάνταρ βιβλιοθήκης της Python(STL). Σκοπός δεν είναι να καλυφθούν και οι παραμικρές λεπτομέρειες αλλά μια αναφορά σε κάποιες πολύ ενδιαφέρουσες περιπτώσεις από τις δυνατότητες της. Για εξονυχιστική μελέτη της STL(standard library) ενθαρρύνεται η ανάγνωση από την επίσημη ιστοσελίδα της γλώσσας ή/και από το μενού βοήθεια μέσα από το IDLE της Python. Ο λόγος που θα παρουσιαστούν και τα δυο μαζί (STL, methods) είναι επειδή μπορούν να συνδυαστούν αρμονικά.

6.1 Η λεπτή διαφορά μεταξύ τυπικών συναρτήσεων και methods

Οι συναρτήσεις είναι ένα μαθηματικό κατασκεύασμα. Όλες οι μέθοδοι(methods) είναι συναρτήσεις ,αλλά δεν είναι όμως όλες συναρτήσεις μέθοδοι στην γλώσσα Python.Μια συνάρτηση είναι ένα κομμάτι(μπλοκ) κώδικα που μπορεί να δέχεται ή μην δέχεται στην είσοδο ορίσματα (data input) και επιστρέφει ή μπορεί να μην επιστρέφει κάποιο αποτέλεσμα(data output),δηλαδή μια τιμή της συνάρτησης. Μια μέθοδος (method) είναι μια συνάρτηση που ανήκει, είναι κτήμα ενός αντικειμένου ή μονάδα σε κάποια κλάση. Αυτό σημαίνει ότι όταν θέλουμε να αναφερθούμε, χρησιμοποιήσουμε μια μέθοδο αυτό γίνεται δια μέσω του αντικειμένου/κλάσης. Για να καλέσουμε μια μέθοδο θα χρειαστεί να χρησιμοποιήσουμε τον τελεστή τελεία, ενώ μια κλασική συνάρτηση δεν τον χρησιμοποιεί. Μια συνάρτηση ορίζεται με την λέξη κλειδί def και μπορεί να είναι αυτόνομη(stand alone) και να καλείται οπουδήποτε. Εν τέλει, η μέθοδος χρησιμοποιείται και είναι μια έννοια στον αντικειμενοστραφή προγραμματισμό, εξαρτάται από κάτι άλλο δε γίνεται να την καλέσουμε αυτόνομα, δεν είναι αυτοδιάθετη. Αντίθετα μια κλασική συνάρτηση είναι ανεξάρτητη, δεν εξαρτάται από κάποιο αντικείμενο. Από άλλη οπτική γωνία μπορούμε να πούμε ότι η μέθοδος είναι η αντικειμενοστραφής εκδοχή της συνάρτησης στον OOP(object oriented programming).

Συναρτήσεις είδαμε πολλές όπως για παράδειγμα στο κεφάλαιο με τις συναρτήσεις ,καθώς και μερικές methods που παρουσιάστηκαν κατά διαστήματα με το κάλυμμα της λέξης συνάρτησης.

Παράδειγμα τυπικής συνάρτησης:

```
>>> def sum(x,y):
        return x+y
>>> sum(2,3)
5
```

Παράδειγμα method:

```
>>> word= "HeLLo all"
>>> word.upper()
'HELLO ALL'
```

Φυσικά να επισημάνουμε ότι μπορούμε να ορίσουμε συνάρτηση και με την λέξη κλειδί lambda. Όπως:

```
>>> result= lambda x:x**2
>>> result(x) # όπου x η τιμή που πρέπει να ορίσουμε εμείς για να πάρουμε αποτέλεσμα.
```

6.2 Θέματα από Python STL

Κοιτάζοντας μέσα στην STL θα διαπιστώσουμε ότι είναι μακροσκελής. Παρατηρούμε ότι περιέχει πολλά και διάφορα δομικά στοιχεία που απαρτίζουν την γλώσσα. Για παράδειγμα περιλαμβάνει τύπους δεδομένων, ενσωματωμένες συναρτήσεις και exceptions(εξαιρέσεις)- ουσιαστικά τα exceptions είναι objects που δε χρειάζεται να χρησιμοποιήσουμε την εντολή import για να τα καλέσουμε. Παρόλα αυτά, ο μεγαλύτερος όγκος της STL περιλαμβάνει πρακτικά μια συλλογή από modules. Το κάθε module είναι εξειδικευμένο στο να καλύπτει ένα συγκεκριμένο φάσμα εργασιών. Για παράδειγμα έχουμε module για κρυπτογράφηση, κάποιο άλλο που παρέχει το interface για κάποια εφαρμογή(λόγου χάρη για το www), module για αριθμητικές και μαθηματικές πράξεις και τα λοιπά. Επομένως, θα καλυφθούν μερικά από τα προτερήματα της STL αυτά που κρίνονται πολύ συχνά χρήσιμα.

6.2.1 Ενσωματωμένες συναρτήσεις, θεμελιώδεις και μη

Υπάρχει ένα σύνολο από συναρτήσεις που είναι απαραίτητο να γνωρίζουμε πολύ καλά γιατί είναι πάρα πολύ χρήσιμες για να προγραμματίζουμε με ευχέρεια ,δεν πρέπει να τις αγνοούμε και υπάρχει ένα σύνολο από μη θεμελιώδεις συναρτήσεις, που σημαίνει ότι δεν είναι πάρα πολύ σημαντικές σε σχέση με τις θεμελιώδεις. Όλες οι πολύ σημαντικές συναρτήσεις δίνονται σε μορφή πίνακα, στον πίνακα 9, όπως εμφανίζονται στο μενού βοήθεια στο Python shell(IDLE). Ας ενθυμόμαστε ότι ένας γρήγορος τρόπος να λάβουμε πληροφορίες τι κάνει η κάθε μία είναι με τις εντολές: <όνομα συνάρτησης>.__doc__ ,dir και help.

Πίνακας 9. Θεμελιώδεις ενσωματωμένες συναρτήσεις της Python

		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

Πηγή: Από το μενού βοήθεια(F1) ,υποενότητα 'Built-in functions' στο IDLE της Python v2.7.6

Θα γίνει μια ειδική μνεία για τις συναρτήσεις repr() και str() που πρακτικά δίνουν το ίδιο αποτέλεσμα αφού κάνουν την ίδια ακριβώς δουλειά. Και οι δυο επιτρέπουν να μετατρέψουμε μια ποσότητα από αυτό που είναι σε αλφαριθμητικό όπως για παράδειγμα να μετατρέπουν τον ακέραιο τρία στο αλφαριθμητικό τρία,γιατί πολύ απλά ο διερμηνευτής δεν ξέρει να συνδυάζει και να κάνει πράξεις με αλφαριθμητικά και αριθμούς όπως οι int και float. Η διαφορά του repr(object) είναι ότι η str(object) δεν έχει προγραμματιστεί ώστε πάντα να προσπαθεί να επιστρέφει ένα αλφαριθμητικό που θα είναι αποδεκτό για την eval(). Ο στόχος της str() είναι απλώς να επιστρέφει ένα καλό εκτυπώσιμο αποτέλεσμα. Η repr() παρέχει μερικές μικρές επιπλέον δυνατότητες. Έτσι, μια κλάση μπορεί να ελέγχει τι θα επιστρέφει αυτή η συνάρτηση για τα στιγμιότυπα της(η κλάση) με χρήση της μεθόδου __repr__() .

Πλην όμως των ουσιωδών συναρτήσεων υπάρχουν και μερικές λιγότερο σημαντικές ενσωματωμένες συναρτήσεις που μπορούμε να τις αγνοούμε χωρίς σημαντικές επιπτώσεις όταν γράφουμε κώδικα. Αυτές είναι οι: apply, buffer, coerce και intern .

6.2.2 Exceptions

Όταν δοκιμάζουμε έναν αλγόριθμο για τυχόν σφάλματα υπάρχει περίπτωση να συναντήσουμε ένα είδος σφάλματος που λέγεται *runtime error*(σφάλμα χρόνου εκτέλεσης) ή με άλλα λόγια *exception*(εξαίρεση), οι δυο έννοιες είναι ισοδύναμες. Λέγεται runtime error γιατί το σφάλμα εντοπίζεται μονάχα όταν το πρόγραμμα αρχίσει να εκτελείται. Λέγεται exception διότι στηρίζεται στην λογική ότι κάτι το εξαιρετικό και δυσάρεστο συνέβη που σταμάτησε την ομαλή λειτουργία του προγράμματος. Σε απλά προγράμματα είναι μάλλον εξαιρετικά απίθανο να εμφανιστούν. Ένα απλό παράδειγμα exception:

```
def mul(x,y):
    result=x*y
    return result

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

Βλέπουμε ότι ορίσαμε μια απλή συνάρτηση σε μορφή script αρχείου ,μετά δοκιμάσαμε να τυπώσουμε στο IDLE το περιεχόμενο της μεταβλητής result και προέκυψε ένα runtime error,διότι η μεταβλητή είναι τοπική και έχει ακτίνα δράσης μόνο εντός της συνάρτησης ,επειδή δημιουργείται και καταστρέφεται μέσα στην συνάρτηση. Ομοίως και στο επόμενο:

```
>>> en2de={'one': 'ein', 'two': 'zwei', 'three': 'drei'}
>>> print en2de['ten']
KeyError: 'ten'        #σφάλμα
>>> print en2de['ONE']
KeyError: 'ONE'    #σφάλμα
```

6.2.3 Exceptions παραγόμενα από τον χρήστη και στάνταρ exceptions

Στην Python υπάρχουν δυο πολύ χρήσιμα εργαλεία για την αντιμετώπιση μη προβλέψιμων σφαλμάτων: assertions και exceptions που είναι το θέμα αυτής της ενότητας. Πρακτικά ένα exception είναι ένα γεγονός(event) που ανακύπτει κατά την εκτέλεση ενός προγράμματος, που διακόπτει την κανονική ροή εκτέλεσης ενός προγράμματος. Ένα exception είναι ένα αντικείμενο στην Python που αναπαριστά ένα σφάλμα. Θα εξετάσουμε πως μπορούμε να διαχειριστούμε τα exceptions παραγόμενα από τον χρήστη.

Σε ενδεχόμενο απροσδόκητου λάθους τοποθετώντας exception μπορούν να προστατεύουν τον κώδικα και τα scripts από το να διακοπούν και να τερματίσουν. Αν για παράδειγμα ο χρήστης εισάγει δεδομένα από το πληκτρολόγιο που δεν έχει γίνει πρόνοια από τον προγραμματιστή θα προέκυπτε σφάλμα, αλλά για αυτά με τα exceptions αποφεύγουμε με κομψό τρόπο από το να τερματίσει άδοξα το πρόγραμμα. Το πιο απλό συντακτικό είναι το εξής:

```
try:
    μπλοκ κώδικα
except:
    μπλοκ κώδικα σε περίπτωση exception
else:
    μπλοκ κώδικα αν δεν υφίσταται exception.
```

Όποτε έχουμε ένα μπλοκ κώδικα για το οποίο υποπτευόμαστε ότι είναι προβληματικό μπορούμε να το εσωκλείσουμε μέσα στο μοτίβο try-except-else για να το δοκιμάσουμε και να δούμε την εξέλιξη του κώδικα. Αυτό το μοτίβο μας προστατεύει από απρόοπτο τερματισμό, «σπάσιμο», του προγράμματος μας.

Είναι δυνατόν να μην γράψουμε καν την else αν δεν το επιθυμούμε και χωρίς κανένα περεταίρω πρόβλημα.

Παράδειγμα χωρίς πρόβλημα που θα τυπώσει το αποτέλεσμα της άθροισης:

```
try:
    x = 10 + 11.5
except:
    print  'unexpected error, look up the type of your input values. '
else:
    print 'result= ',x
```

Παράδειγμα με πρόβλημα που θα τυπώσει το μπλοκ κώδικα(οσοδήποτε μεγάλο κι αν έχουμε γράψει) και βρίσκεται εντός της except:

```
try:
    x = 10 + 'Python'
except:
    print  'unexpected error, look up the type of your input values. '
else:
    print 'result= ',x
```

Αυτό το μοτίβο try-except-else θα συλλαμβάνει πρακτικά όλα τα είδη exception που μπορούν να συμβούν και είναι γενικού σκοπού μοτίβο. Φυσικά υπάρχουν και ποιο εξειδικευμένα, συγκεκριμένα μοτίβα που βοηθάνε τον προγραμματιστή να εντοπίζει την ρίζα του κακού. Ουσιαστικά πρόκειται για παραλλαγές της βασικής δομής του άνωθεν μοντέλου που παρουσιάστηκε. Έτσι, ένα ποιο συγκεκριμένο μοντέλο ακολουθεί την εξής λογική:

```
try:
     μπλοκ κώδικα
except <περίπτωση 1> :
      μπλοκ κώδικα σε περίπτωση exception
except <περίπτωση 2> :
      μπλοκ κώδικα για αυτή την περίπτωση
              .....
except <περίπτωση κ> :
      μπλοκ κώδικα για αυτή την περίπτωση
else:
      μπλοκ κώδικα αν δεν υφίσταται κανένα  από τα exception.
```

Παράδειγμα:

```
try:
      x = x/y
except ZeroDivisonError:
      print  'ERROR: forbidden operation. Cannot divide by zero'
except OverflowError:
      print  'ERROR: buffer memory is overflown. Try smaller numbers'
else:
      print 'sum result= ',x
```

Από αυτό το πιο αναλυτικό μοτίβο μπορούμε να εξάγουμε τα ακόλουθα συμπεράσματα:

i. Μετά από τις except προτάσεις μπορούμε να συμπεριλάβουμε(προαιρετικά αν και προτείνεται να μην παραλείπεται) την else, η οποία υλοποιείται όταν το κομμάτι κώδικα εντός της try δεν προκαλέσει κανένα exception.
ii. Μια try μπορούμε να την συνοδεύουμε με μία ή και πολλαπλές except για να καλύπτουμε πολλές και διάφορες συγκεκριμένες περιπτώσεις και ανάγκες στον κώδικα μας.

Copyright © 2014

Αν επιθυμούμε λόγω αιτιάσεων του κώδικα που γράφουμε μπορούμε να συμπεριλάβουμε πολλαπλές περιπτώσεις exception σε μια μόνο γραμμή με μια λίστα, μία αρκεί να ικανοποιείται και θα συμβεί:

```
try:
    μπλοκ κώδικα
except (Exception_1[, Exception_2[,...Exception_K]]]):
    μπλοκ κώδικα
else:
    μπλοκ κώδικα αν δεν υφίσταται exception.
```

Στην δομή try-except μπορούμε να προσθέσουμε την finally, try-except-finally ή ποιο απλά χωρίς το except : try-finally, που υλοποιείται ακριβώς εκεί που τελειώνει η except ή πριν από αυτήν και θα περιλαμβάνει κώδικα που θα εκτελεστεί οπωσδήποτε άσχετα αν η try προκάλεσε ένα exception ή όχι.

Παράδειγμα:

```
try:
    if x%2==0:
        print ' x is even'
    else:
        print 'x is odd'
finally:
    z = x + y
    print 'z= ',z
except ValueError, Argument:
    print 'Error: no numbers, improper value: , Argument
```

Τέλος χάριν πληρότητας θα αναφερθεί ότι υπάρχει η συνάρτηση raise που γεννά εξαιρέσεις και που έχει παρουσιαστεί σε προηγούμενο κεφάλαιο με το εξής συντακτικό:
raise [Exception [, ορίσμα [, traceback]]]

Το πεδίο **Exception** αναπαριστά τον τύπο του exception όπως αυτός δίνεται στον πίνακα 10 παρακάτω, το όρισμα(που είναι προαιρετικό και μπορεί να παραληφθεί) είναι η τιμή για το exception.Τέλος, το **traceback** πεδίο σπανιότατα χρησιμοποιείται, είναι προαιρετικό πεδίο και αναπαριστά το traceback αντικείμενο που χρησιμοποιείται για το συγκεκριμένο exception.

Πίνακας 10. Στάνταρ exceptions διαθέσιμα στην Python.

EXCEPTION	Σύντομη περιγραφή
Exception	Βασική κλάση για όλα τα exceptions
ZeroDivisonError	Προκύπτει όταν λαμβάνει χώρα μια διαίρεση ή modulo δια του μηδέν, για όλους τους αριθμητικής τύπους.
SystemExit	Προκύπτει από την sys.exit() συνάρτηση.
StandardError	Βασική κλάση για όλα τα ενσωματωμένα(στην Python) exceptions εκτός των StopIteration και SystemExit.
ImportError	Προκύπτει σε περίπτωση αποτυχίας της import.
OverflowError	Προκύπτει όταν ένας υπολογισμός ξεπεράσει το άνω όριο για έναν αριθμητικό τύπο.
EOFError	Προκύπτει όταν έχει επέλθει το EOF και δεν υφίσταται είσοδος είτε από την συνάρτηση raw_input() είτε input() function.

StopIteration	Προκύπτει όταν η next() method ενός iterator δεν δείχνει σε κάποιο object.
AssertionError	Προκύπτει σε περίπτωση αποτυχίας της δήλωσης Assert.
AttributeError	Προκύπτει σε περίπτωση αποτυχίας αναφοράς χαρακτηριστικής ιδιότητας(attribute reference) ή ανάθεσης(assignment).
FloatingPointError	Προκύπτει όταν αποτύχει ένας υπολογισμός κινητής υποδιαστολή.
ArithmeticError	Βασική κλάση για όλα τα λάθη που προκύπτουν από αριθμητικούς υπολογισμούς.
KeyboardInterrupt	Όταν ο χρήστης διακόπτει την ροή εκτέλεσης προγράμματος(συνήθως πατώντας Ctrl+c).
LookupError	Βασική κλάση για όλα τα λάθη αναζήτησης.
IndexError	Προκύπτει αν ο δείκτης μιας ακολουθίας δεν βρεθεί.
KeyError	Αποτυχία εύρεσης ενός κλειδιού σε ένα λεξικό προκαλεί αυτό το exception.
NameError	Αποτυχία εύρεσης ενός αναγνωριστικού(identifier) σε ένα τοπικό ή καθολικό namespace.
UnboundLocalError	Προκύπτει όταν προσπαθούμε να προσπελάσουμε μια τοπική μεταβλητή(και δεν έχουμε ορίσει τιμή) σε μια συνάρτηση ή method.
RuntimeError	Προκύπτει όταν ένα παραγόμενο σφάλμα δεν ανήκει σε καμία άλλη κατηγορία λαθών.
SystemError	Παρουσιάζεται όταν ο διερμηνευτής εντοπίσει ένα εσωτερικό πρόβλημα, αλλά δεν τερματίζεται(exit) ο διερμηνευτής της Python.
OSError	Προκαλείται από σφάλματα σχετιζόμενα με το λειτουργικό σύστημα (OS).
ValueError	Θα παρουσιαστεί όταν μια ενσωματωμένη συνάρτηση για κάποιο τύπο δεδομένων έχει έγκυρο τύπο ορισμάτων, αλλά τα ορίσματα έχουν καθοριστεί μη έγκυρες τιμές.
TypeError	Θα εμφανιστεί άμα μια μη έγκυρη λειτουργία ή συνάρτηση δοκιμάζεται για ένα συγκεκριμένο τύπο δεδομένων.
IOError	Συμβαίνει εάν μια λειτουργία I/O αποτύχει πχ η print() ή η open() συνάρτηση όταν επιχειρηθεί να ανοιχτεί ένα ανύπαρκτο αρχείο.
NotImplementedError	Συμβαίνει εάν μια αφηρημένη method που χρειάζεται να εφαρμοστεί σε μια κληρονομημένη κλάση δεν πραγματοποιείται τελικά.
IndentationError	Προκύπτει λόγω μη σωστών εσοχών.
SyntaxError	Επέρχεται κατόπιν συντακτικού σφάλματος στην Python.
EnvironmentError	Βασική κλάση για όλα τα exceptions που συμβαίνουν εκτός του περιβάλλοντος της Python.
SystemExit	Ανακύπτει όταν ο διερμηνευτής της Python τερματίζεται με την sys.exit() συνάρτηση. Εάν δεν ρυθμιστεί, αναγκάζει τον διερμηνευτή να τερματίσει(exit).

(ο πίνακας είναι η συνέχεια από την σελίδα 47)

6.2.4 File objects και είσοδος από πληκτρολόγιο

Τα file objects(αντικείμενα αρχεία) για να υλοποιηθούν(εσωτερικά) βασίζονται στο πακέτο 'stdio' της C γλώσσας και μπορούμε να τα καλέσουμε/δημιουργήσουμε χρησιμοποιώντας της ενσωματωμένη εντολή method: open(). Αυτό που κάνει η εντολή μόλις την χρησιμοποιήσουμε είναι να 'επιστρέφει'(δημιουργεί) ένα file object το οποίο θα μπορεί να χρησιμοποιηθεί σε συνδυασμό με άλλα methods και χαρακτηριστικά(attributes) για να επεξεργαστούμε το αρχείο που μόλις προσπελάσαμε. Υπάρχουν βέβαια και άλλοι τρόποι κλήσης τους όπως με τις os.fdopen(), os.popen() και την makefile() method αν μιλάμε για socket objects, ανάλογα τι έργο θέλουμε να επιτελέσουμε χρησιμοποιούμε ότι μας ταιριάζει. Επιπλέον, μπορούμε να χρησιμοποιήσουμε κάποια modules για την διαχείριση των file objects, όπως λόγου χάρη για τα προσωρινά αρχεία μπορούμε να τα δημιουργούμε με την βοήθεια του tempfile module. Για αντιγραφή, μεταφορά και διαγραφή αρχείων ακόμα και καταλόγων(directories) μπορούμε να υπολογίζουμε στο shutil module. Σε περίπτωση αποτυχίας ανοίγματος ενός αρχείου παίρνουμε το 'IOError' exception ως μήνυμα λάθους. Παράδειγμα χρήσης της open():

```
from re import split
def manylines(n):
    fo=open("/Python27/words.txt")
    i = 1
    counts = 0
    for i in range(n):
        line = fo.readline().strip()
        for word in split('\s+', line):
            if 'e' not in word:
                print word
                counts += 1
    fo.close()
    print "number of non 'e' words =", counts
```

Με την open(…) ανοίξαμε ένα αρχείο που ήταν αποθηκευμένο στον σκληρό δίσκο και εκτελέσαμε ένα σάρωμα στο κείμενο και μετά κλείσαμε το words.txt αρχείο με την fo.close(),κατόπιν τυπώσαμε πόσες λέξεις βρέθηκαν που δεν περιέχουν μέσα τους το γράμμα 'e'.Την ανάγνωση γραμμών την καταφέραμε με την εντολή readline. Κάθε φορά που ανοίγεται ένα αρχείο μόλις τελειώσει η επεξεργασία του πρέπει να κλείνει με μια κατάλληλη εντολή όπως η close() για να αποδεσμεύονται οι πόροι τους συστήματος. Το format της close() είναι της μορφής: όνομα_αρχείου.close().Υπάρχει η δυνατότητα να παραλείψουμε εντελώς την close() αν χρησιμοποιήσουμε την δήλωση with σε συνδυασμό με την as ως εξής:

```
with open("words.txt") as myfile:
        for line in myfile:
                print line
```

που ισοδυναμεί με χρήση της εντολής όνομα_αρχείου.close() με το:

```
myfile = open("words.txt")
try:
    for line in myfile:
        print line
finally:
    myfile.close()
```

Να επισημανθεί ότι μπορούμε με την εντολή closed να σιγουρέψουμε αν έκλεισε ένα αρχείο γράφοντας στο παραπάνω κώδικα που χρησιμοποιεί την εντολή with: myfile.closed.Επίσης,η open() δέχεται μέχρι και τρεις παραμέτρους με την πρώτη να είναι υποχρεωτική όπως φαίνεται και στους δυο παραπάνω

 Copyright © 2014

αλγορίθμους, ενώ οι δυο τελευταίες αφορούν τον τρόπο και το buffer και αποτελούν προαιρετικά πεδία. Συνεπώς το πλήρες συντακτικό της είναι:
open(filename,mode,buffer)

Αν τα δυο τελευταία πεδία δεν ορίζονται, τότε η προεπιλογή είναι να ανοίγονται τα αρχεία ως αρχεία ανάγνωσης 'r' (read only). Μπορούμε να εισάγουμε στον διερμηνευτή την ακόλουθη γραμμή εντολής για βοήθεια: print open.__doc__ .Συνηθισμένες τιμές για το mode είναι: 'r' για ανάγνωση, 'w' για εγγραφή ή 'a' για προσθήκη στο τέλος του αρχείου επιπλέον ύλης. Επειδή υπάρχουν αρκετές μέθοδοι(methods) που μπορούμε να χρησιμοποιούμε με τα αρχεία για επεξεργασία αυτών παρατίθεται ένας πίνακας παρακάτω με την ονομασία και σύντομη περιγραφή τους. Ο πίνακας είναι ενδεικτικός και επειδή η γλώσσα εξελίσσεται κάποια από τα αναφερόμενα μπορεί να περιπέσουν σε αχρησία, να καταργηθούν ή να διαφοροποιηθούν. Να επισημανθεί ότι η χρήση των methods γίνεται με το γνωστό τρόπο: filename.method .

```
>>> fo=open('C:\Users\Thanos\Desktop\myword.txt','w')
>>> fo.write('Hi.I just wrote a line in this text stream')
>>> fo.close()
>>> fo=open('C:\Users\Thanos\Desktop\myword.txt','r')
>>> fo.read()
'Hi.I just wrote a line in this text stream'
>>> fo.close()
```

Σχήμα 11. Σωστή αλληλουχία χρήσης των εντολών open, write, read και close.

Πίνακας 11. Επεξεργασία των file objects με methods και χαρακτηριστικά(attributes)

Method ή χαρακτηριστικό	**Περιγραφή**
close()	Κλείσιμο αρχείου. Μπορεί να ξαναχρησιμοποιηθεί. Δεν μπορεί να υποστεί επεξεργασία ένα κλειστό αρχείο χωρίς να ξανά ανοίξει με την open() ,αλλιώς προκύπτει ValueError exception.
flush()	Καθαρισμός της εσωτερικής προσωρινής μνήμης buffer.
tell()	Επιστρέφει την τρέχουσα θέση του αρχείου.
isatty()	Επιστρέφει True εφόσον το αρχείο συνδέεται με μια tty-like συσκευή, αλλιώς επιστρέφει False.
write(*str*)	Εγγραφή ενός αλφαριθμητικού σε ένα αρχείο χωρίς να επιστρέφει κάποια τιμή. Λόγω buffering, μπορεί να μην εμφανιστεί μέχρι να γίνει κλήση των flush() και close().
writelines(sequence)	Εγγραφή πολλών αλφαριθμητικών, η ακολουθία μπορεί να είναι iterable πχ λίστα από αλφαριθμητικά, δεν επιστρέφει τιμή.
truncate([size])	Περικόπτει το μέγεθος του αρχείου σε μικρότερο ξεκινώντας από αριστερά προς τα δεξιά. Αν ορίζεται τιμή στο πεδίο size τότε περιορίζεται το πολύ έως αυτό το μέγεθος. Μιμείται την λειτουργία του πλήκτρου backspace.
fileno()	Επιστρέφει τον ακέραιο 'file descriptor' που χρησιμοποιείται από την υποβαστάζουσα εφαρμογή για να αιτηθεί I/O λειτουργίες. Χρήσιμο για low-level interfaces πχ το fcntl module.
seek(offset[,whence])	Θέτει την θέση ενός αρχείου ως τωρινή ότι κι αν ήταν νωρίτερα. Επιτρεπτές τιμές για το whence: 0,1,2. 0=αναζήτηση από την αρχή του αρχείου,1=από την τωρινή θέση, όποια είναι, 3= από το τέλος του αρχείου.
next()	Επιστρέφει την επόμενη εισαγόμενη γραμμή όταν γίνει χρήση της με ένα αρχείο που λειτουργεί ως iterator σε ένα βρόχο για να καλείται επαναληπτικά. Σε περίπτωση EOF => StopIteration exception. Δεν συνίσταται να συνδυαστεί με άλλες μεθόδους.
read([size])	Ανάγνωση ενός αλφαριθμητικού από ένα ανοικτό αρχείο, αν δεν οριστεί το πεδίο size διαβάζει μέχρι να συναντήσει το EOF,δηλαδή όλο το αρχείο.
readline([size])	Ανάγνωση μιας ολόκληρης γραμμής ενός αρχείου

readlines([sizehint])	Ανάγνωση όλου του κειμένου αν δεν οριστεί περιορισμός. Αν το sizehint δημιουργεί προβλήματα αποφεύγεται.
xreadlines()	Ίδιο με την iter(f)
closed	Χαρακτηριστικό τύπου bool.Υποδεικνύει την τωρινή κατάσταση ενός file object.
encoding	Χαρακτηριστικό .Αποκαλύπτει την κωδικοποίηση του αρχείου. Τυχόν ένδειξη None σημαίνει ότι το σύστημα χρησιμοποιεί default κωδικοποίηση για μετατροπή Unicode strings.
mode	Χαρακτηριστικό .Το I/O mode του αρχείου.
errors	Χαρακτηριστικό .Χειριστής για Unicode errors,μπορεί να συνδυάζεται με το encoding.
newlines	Χαρακτηριστικό .Καταγραφή των τύπων των νέων γραμμών ενώ διαβάζεται ένα αρχείο. Μπορεί να διαχειριστεί τιμές '\r', '\n', '\r\n'. Αρχεία μη ανοιγμένα σε universal read mode επιστρέφεται απάντηση None.
name	Χαρακτηριστικό. Δίνει το όνομα του αρχείο αν έγινε χρήση της open(),ειδάλλως δηλώνει την πηγή του file object σε format <....>
softspace	Χαρακτηριστικό τύπου bool. Δηλώνει αν πρέπει ο κενός χαρακτήρας πρέπει να τυπωθεί πριν από κάποια άλλη τιμή όταν γίνεται χρήση της δήλωσης print. Δεν προορίζεται για να ελέγχει την print αλλά για να καταγράφει την εσωτερική της κατάσταση.

Πηγή: http://docs.python.org/2/library/stdtypes.html#file-objects (συνέχεια από σελίδα 50)

Τέλος, θα δειχθούν οι δυο τρόποι εισαγωγής πληροφοριών από το πληκτρολόγιο. Οι δυο εντολές που πραγματοποιούν εισαγωγή αλφαριθμητικών είναι η input και η raw_input. Στην ποιο πρόσφατη έκδοση της Python 3.x.x ισχύει η input μόνο. Στην input() ότι βρίσκεται εντός των παρενθέσεων η Python 2.7.x το αντιμετωπίζει ως έκφραση(expression) κι όχι ως αλφαριθμητικό(string),συνεπώς αδυνατεί να μετατρέψει η input χαρακτήρες/σύμβολα σε αλφαριθμητικά. Αντίθετα, η raw_input() λαμβάνει την πληροφορία από το πληκτρολόγιο και την μετατρέπει σε αλφαριθμητικό.

A) Ένα παράδειγμα με την input.

```
scratchnotepad.py - C:/Python27/myworkshop/scratchnotepad.py
File Edit Format Run Options Windows Help
userinp= input('what is your name? :')
print userinp
```

```
>>> ================================ RESTART ========================
>>>
what is your name? :Mike

Traceback (most recent call last):
  File "C:/Python27/myworkshop/scratchnotepad.py", line 1, in <module>
    userinp= input('what is your name? :')
  File "<string>", line 1, in <module>
NameError: name 'Mike' is not defined
>>>
```

Σχήμα 12. Λάθος χρήση της input

```
scratchnotepad.py - C:/Python27/myworkshop/scratchnotepad.py
File Edit Format Run Options Windows Help
userinp= input('what is your name? :')
print userinp
```

```
>>> ================================ RESTART =
>>>
what is your name? :'Mike'
Mike
>>>
```

Σχήμα 13.Σωστή χρήση της input

B) Ένα παράδειγμα με την raw_input.

```
scratchnotepad.py - C:/Python27/myworkshop/scratchnotepad.py
File Edit Format Run Options Windows Help
userinp= raw_input('what is your name? :')
print userinp

>>> ================================ RESTART ======
>>>
what is your name? :Mike
Mike
>>>
```

```
scratchnotepad.py - C:/Python27/myworkshop/scratchnotepad.py
File Edit Format Run Options Windows Help
userinp= raw_input('what is your name? :')
print userinp

>>> ================================ RESTART ==
>>>
what is your name? :'Mike'
'Mike'
>>>
```

Σχήμα 14. Η raw_input στην Python έκδοση 2.7.x είναι πιο ευέλικτη από την input.

6.3 Πρόσβαση αρχείων και καταλόγων με modules

Ένα module είναι ένα αρχείο(file) που περιλαμβάνει κώδικα γραμμένο στην Python,λογικά οργανωμένο, μέσα σε ένα module μπορούν, όπως έχουμε δει για παράδειγμα το math module, να περιλαμβάνονται και να ορίζονται συναρτήσεις, κλάσεις και μεταβλητές, μπορεί επίσης να έχει εκτελέσιμο κώδικα. Ένα οποιοδήποτε module είναι ένα αντικείμενο(Python object) με κάποια ονοματισμένα χαρακτηριστικά(attributes) που καθορίζονται με κατάλληλο τρόπο. Είναι χρήσιμα τα modules όπως λόγου χάρη για ομαδοποίηση μπλοκ κώδικα σε modules με συγκεκριμένη θεματολογία το κάθε ένα, άρα ευκολότερα κατανοητά το τι κάνουν και ευκολότερο να τα αναπαράγουμε και να τα συμπεριλαμβάνουμε σε νέα προγράμματα. Ο τυπικός τρόπος για να αποκτήσουμε πρόσβαση και να χρησιμοποιήσουμε κάποιο ή κάποια από τα δομικά μέρη του είναι με την εντολή import και προσπέλαση με τον τελεστή τελεία. Θα ακολουθήσει μια σύντομη περιήγηση (όχι εκτενή παρουσίαση)σε μερικά από τα σημαντικά modules που περιέχει η Python STL και αυτά είναι: os και os.path, sys, time και το shutil module.

6.3.1 Το os module ως διεπαφή του συστήματος

Αυτό το module μπορεί να χρησιμοποιηθεί για την διεκπεραίωση εργασιών όπως η εύρεση του τωρινού επεξεργαζόμενου καταλόγου(directory),για αλλαγή του τωρινού σε λειτουργία καταλόγου, για έλεγχο εάν συγκεκριμένα αρχεία ή κατάλογοι όντως υπάρχουν σε μια συγκεκριμένη τοποθεσία, για την δημιουργία νέων καταλόγων και την προσπέλαση αυτών προκειμένου να πραγματοποιηθούν διάφορες εργασίες που επιθυμεί ο χρήστης είναι μερικά από τα πάρα πολλά που μπορεί να κάνει αυτό το μακροσκελές module. Μια ποιο πρόσφατη βελτιωμένη εκδοχή του είναι το **subprocess** module που μπορεί να αντικαταστήσει επάξια το os module. Οι ακόλουθες σημειώσεις θα επιχειρήσουν να αναδείξουν μάλλον τα ποιο συχνά αναμενόμενα δομικά στοιχεία που απαρτίζουν το os.module. Για μια εξαντλητική λίστα μπορούμε να διατρέξουμε και στην επίσημη ιστοσελίδα των δημιουργών της γλώσσας Python. Αφού εισάγουμε το os module με την εντολή import μπορούμε να *αναζητήσουμε πληροφορίες* για το τρέχον ενεργό κατάλογο:

>>> **import** os
>>>os.getcwd()
'C:\\Python27'
Στον συγκεκριμένο υπολογιστή που έτρεξε η εντολή os.getcwd() το μονοπάτι του τρέχοντος καταλόγου είναι αυτό που προέκυψε. Η δήλωση αυτή είναι χρήσιμη όταν υπάρχει αμφιβολία για το ποιο κατάλογο χρησιμοποιείται σε μια δεδομένη στιγμή. Ένας εναλλακτικός τρόπος είναι επίσης με την:
>>> os.path.abspath('.')
'C:\\Python27'
Ενώ η os.listdir('.') παράγει μια αναλυτικότερη λίστα όλων των συμπεριλαμβανόμενων αρχείων και καταλόγων στο τρέχον κατάλογο.

```
>>> from os import *
>>> getcwd()
'C:\\Python27'
>>> listdir('.')
['code solutions to the book', 'dateutil', 'DLLs',
s', 'LICENSE.txt', 'makeGUI', 'myworkshop', 'NEWS.
ininst.log', 'PyQt-win-gpl-4.10.3', 'python books'
on.exe', 'pythonw.exe', 'README.txt', 'Removepypar
```

Σχήμα 15. Αποτέλεσμα της χρήσης των getcwd() και listdir('.')

Αφού μόλις είδαμε πως μπορούμε να αναζητήσουμε πληροφορίες σε ένα τρέχον ενεργό κατάλογο θα δειχθεί τώρα πώς να *μεταβαίνουμε σε διαφορετικό κατάλογο* με την μέθοδο chdir():
>>> **import** os
>>>os.getcwd()
'C:\\Python27'
>>> os.chdir('C:\Program Files') # εδώ γίνεται η αλλαγή
>>> os.getcwd()
'C:\\Program Files'

Για όσο πλοηγούμαστε εντός του καταλόγου 'Python27' η μετάβαση σε υποκαταλόγους του μπορεί να γίνεται με χρήση του αναγνωριστικού ' ./ ' με τον εξής τρόπο:
>>> os.chdir('./PLOTdata')
>>> os.getcwd()
'C:\\Python27\\PLOTdata'

Για επιστροφή στο προηγούμενο(ένα βήμα πίσω κάθε φορά) γίνεται με την εντολή chdir('../').Μιας και έγινε χρήση των σχετικών μονοπατιών αλλαγής καταλόγου επιτρέπεται η πλοήγηση και με το απόλυτο μονοπάτι(absolute path) όπως έγινε στο προηγούμενο παράδειγμα ('C:\\Program Files'). Για να *δημιουργήσουμε καταλόγους* σε κάποιο σημείο που δεν υπάρχουν θα πρέπει να πληκτρολογήσουμε:
>>> **import** os # αν έχει φορτωθεί η εντολή από πριν, τότε δε χρειάζεται να την ξανακαλέσουμε.
>>> os.mkdir('C:\Program Files\newfile') # absolute path
>>> os.mkdir('./newfile') # relative path

Διαλέγουμε έναν από τους δυο τρόπους: απόλυτο ή σχετικό μονοπάτι. Παρατηρούμε ότι το απόλυτο μονοπάτι αρχίζει πάντα από την αρχή, την βάση των windows, δηλαδή τον επικεφαλή κατάλογο C:\ γι' αυτό λέγεται και απόλυτο μονοπάτι. Αν τυχόν το αρχείο υπάρχει ήδη θα προκύψει μήνυμα λάθους τύπου OSError. Αν δεν είμαστε σίγουροι για την ύπαρξη ή όχι ενός καταλόγου μπορούμε να το σιγουρεύουμε δοκιμάζοντας πρώτα το αν υπάρχει με την εντολή isdir για καταλόγους, isfile για αρχεία(κι όχι την exists γιατί αν βρει το αρχείο ήδη μέσα δε θα δημιουργήσει άλλο) και αν δεν υπάρχει να προχωράμε:

```
>>> import os.path
>>> if os.path.isdir('./newfile'):
        pass
    else:
        os.mkdir('./newfile')
```

Μετέπειτα θα δούμε ότι για να αντλήσουμε (να επιλέξουμε) το όνομα από ένα αρχείο ή κατάλογο από γνωστό δεδομένο μονοπάτι θα χρησιμοποιήσουμε την os.path.basename() ή την os.path.split() που δίνει μια πλειάδα. Παρακάτω δίνεται σε μορφή πίνακα μια λίστα όχι πλήρης αλλά με τα τις πιο συνηθισμένες μεθόδους για το περιβάλλον των windows και σύντομη περιγραφή αυτών.

Πίνακας 12. Το os module. Οι ορθογώνιες αγκύλες δηλώνουν προαιρετικά πεδία.

Μέθοδος	**Περιγραφή**
abort()	Παραγωγή ενός SIGABRT σήμα στο τρέχον process. Στα windows παράγεται exit code 3.
access(path, mode)	Δοκιμή πρόσβασης συγκεκριμένου μονοπατιού με ορίσματα το μονοπάτι και mode να δηλώνει τα access permissions (flags)
getcwd()	Δείχνει το τωρινό ενεργό κατάλογο.
chdir(path)	Αλλαγή τοποθεσίας ενεργού καταλόγου.
listdir(path)	Επιστρέφει μια λίστα με τα περιεχόμενα του καταλόγου.
mkdir(path[,mode])	Δημιουργία καταλόγου.
makedirs(path[,mode])	Δημιουργία καταλόγου, για δημιουργία ενδιάμεσων καταλόγων χρειάζεται να συμπεριληφθεί το lead dir.
times()	Χρονομετρεί και επιστρέφει τους χρόνους επεξεργαστή.
fchdir(fd)	Αλλαγή του τρέχων καταλόγου του file descriptor.
environ	Λεξικό που αντιστοιχεί μεταβλητές περιβάλλοντος με τα pathnames.
strerror(code)	Σφάλμα στον κώδικα
umask(mask)	Θέτει το τωρινό αριθμητικό umask και επιστρέφει το προηγούμενο.
fdopen(fd[,mode[,bufsize]])	Επιστρέφει ένα ανοικτό file object συσχετιζόμενο με έναν file descriptor
chmod(path, mode)	Αλλαγή ρύθμισης(mode) μονοπατιού. Στα Windows το μόνο mode είναι το read-only.
remove(path)	Διαγραφή αρχείων όχι καταλόγων
removedirs(path)	Αναδρομική διαγραφή καταλόγων μόνο
rename(source, destin.)	Απλή μετονομασία αρχείων ή καταλόγων
renames(source, destin.)	Αναδρομική μετονομασία αρχείων ή καταλόγων. Προσπαθεί να δημιουργήσει μονοπάτι.
rmdir(path)	Διαγράφει το μονοπάτι ενός καταλόγου, πρέπει υποχρεωτικά να έχει αδειάσει με μια άλλη εντολή νωρίτερα, ειδάλλως προκύπτει OSError.
stat(path)	Πραγματοποιεί stat κλήση συστήματος στο δοσμένο μονοπάτι.
utime(path,times)	Θέτει την πρόσβαση και τους τροποποιημένους χρόνους.
name	Το όνομα του OS
getpid()	Περισυλλογή του τρέχοντος process αναγνωριστικού(ID)
system(command)	Εκτέλεση εντολής σε ένα subshell. Π.χ. os.system("python.exe")
startfile(path[, operation])	Έναρξη αρχείου ή εφαρμογής, πρακτικά μιμείται το διπλό κλικ.
urandom(n)	Επιστρέφει ένα αλφαριθμητικό με ν τυχαία bytes, κατάλληλο για κρυπτογραφία.

Μια ακόμα ενδιαφέρουσα εφαρμογή που μπορούμε να υλοποιήσουμε με τον παραπάνω πίνακα είναι να ερωτήσουμε αν υπάρχει το αρχείο πριν καν αποκτήσουμε πρόσβαση σε ένα αρχείο για επεξεργασία του.

```
>>> from os import *
>>> from os import F_OK, R_OK, W_OK, X_OK   #flags
'''σχόλια:
F_OK=  αν ένα αρχείο ή κατάλογος υπάρχει              R_OK= αν έχουμε την άδεια για ανάγνωση
W_OK= για γραφή σε μια δοσμένη τοποθεσία               X_OK= για προσπέλαση( execute)
'''
>>> access("c:/python27/wordd.txt ",F_OK)
True
```

```
>>> access("c:/python27/wordd.txt ",R_OK | W_OK)
True
```

Με την stat μπορούμε να λαμβάνουμε λεπτομερείς στατιστικές πληροφορίες για διάφορες λειτουργίες για ένα αρχείο.

```
>>> from os import stat
>>> info=stat('c:/python27/wordd.txt')
>>> print info
nt.stat_result(st_mode=33206, st_ino=0L, st_dev=0, st_nlink=0, st_uid=0, st_gid=0, st_size=65L, st_atime=1387058773L, st_mtime=1387058799L, st_ctime=1387058773L)
>>>info.st_ size   # έτσι εμφανίζονται τα περιεχόμενα της μεταβλητής info.
65
```

Από όπου το st_mode αναπαριστά bits προστασίας(protection bits), st_ino αντιστοιχεί στο inode αριθμό που είναι το file ID; ένας handler(αναγνωριστικό) αρχείων σε περιβάλλον Unix και ο οποίος στα windows πρακτικά δεν υπάρχει με την έννοια που υπάρχει στα Unix, για αυτό το λόγο προκύπτει η ένδειξη μηδέν. Ακόμη, st_dev σημαίνει συσκευή, st_nlink σημαίνει hard link(πραγματικοί σύνδεσμοι), st_uid σημαίνει owner's user ID(ταυτότητα χρήστη), st_gid σημαίνει owner's group ID(ταυτότητα ομάδας), st_size σημαίνει μέγεθος αρχείου σε bytes, st_atime εννοεί τον ποιο πρόσφατο χρόνο προσπέλασης αρχείου(most recent access time), st_mtime αντιστοιχεί στην ποιο πρόσφατη χρονική τιμή μετατροπής(modification time) και st_ctime που ισοδυναμεί με τον χρόνο δημιουργίας αρχείου στα windows(creation time),ενώ στα Linux παραπέμπει στην ποιο πρόσφατη χρονική αλλαγή των metadata.Τέλος, επειδή αυτοί οι αριθμοί μπορεί να είναι τύπου float ή integer μπορούμε να το ελέγξουμε αυτό καλώντας την ακόλουθη συνάρτηση από το os.module:

```
>>> from os import stat_float_times
>>> stat_float_times()
True
```

άρα οι τιμές περιλαμβάνουν τιμές που είναι τύπου float. False θα σήμαινε ότι οι τιμές είναι integers.

6.3.2 To os.path module

Είδαμε ότι για να αποκτήσουμε πρόσβαση στο σύστημα αρχείων(filesystem) χρησιμοποιήσαμε το os module. Για να διαβάσουμε ή να γράψουμε σε ένα απλό text αρχείο χρησιμοποιήσαμε την εντολή open(). Μια προέκταση του os module είναι το os.path module.Η κύρια δραστηριότητα του os.path module είναι για επεξεργασία της διαδρομής ενός μονοπατιού κάποιου αρχείου ή καταλόγου. Βέβαια, διαθέτει συναρτήσεις που στοχεύουν να επεξεργάζονται ονόματα μονοπατιών(pathnames). Η δύναμη αυτού του module γίνεται αναγνωρίσιμη όταν γράφουμε κώδικα που θέλουμε να μπορεί να αλληλεπιδράσει με αρχεία σε διαφορετικές πλατφόρμες. Για σίγουρα σωστή επεξεργασία UNC pathnames προτείνεται η χρήση των συναρτήσεων splitunc() και η ismount().

Με την βοήθεια του module αυτού(os.path) μπορούμε μεταξύ άλλων να ελέγξουμε αν υπάρχει ένα αρχείο, να διακρίνουμε ένα αρχείο από ένα κατάλογο, να επιβεβαιώσουμε αν δυο μεταβλητές δείχνουν ή όχι στον ίδιο κατάλογο, να διαβάσουμε το μέγεθος ενός αρχείου και να ελέγξουμε την διαγραφή κάποιου αρχείου. Θα δειχθεί ένα τυπικό παράδειγμα πως ελέγχουμε την ταυτότητα ενός αρχείου ή καταλόγου.

```
from os.path import isfile, isdir
def check(mypath):
    if isfile(mypath):
        print ' this is a file. '
    elif isdir(mypath):
        print 'this is a directory. '
    else:
        print ' non existent. '
```

Αυτό που συμβαίνει είναι ότι διαβάζει ο κώδικας ένα μονοπάτι που εισάγεται ως όρισμα στο interface της συνάρτησης και αποφαίνεται αν πρόκειται ή όχι για αρχείο ή κατάλογο ή τίποτα από τα δυο.
Σε αυτή την παράγραφο θα γίνει αναφορά πως μπορούμε να δημιουργήσουμε ένα δικό μας μονοπάτι από την πηγή στον προορισμό, τόσο η πηγή όσο και ο προορισμός θα μπορούν να είναι παραμετροποιημένα (custom) από τον χρήστη. Στον κώδικα παρακάτω χτίζεται ένα μονοπάτι με πηγή το '/users' που το αναθέτουμε στη μεταβλητή root και προορισμός το 'frequency spectrum.pdf ' που αντιστοιχεί στην μεταβλητή pdfdocs. Τις μεταβλητές τις χρησιμοποιούμε συνήθως για ευκολότερη παρακολούθηση του κώδικα και για αποθήκευση τιμών που μπορεί μελλοντικά να αλλάξουν.

Σχήμα 16.Δημιουργία ενός αυτοσχέδιου μονοπατιού

```
>>> root='/users'
>>> user='ThanosPC'
>>> mydata='pdfdocs'
>>> custompath= root+ '/' + user + '/' + mydata + '/' + 'frequency spectrum.pdf'
>>> custompath
'/users/ThanosPC/pdfdocs'/'frequency spectrum.pdf'    # χωρίς print είναι αλφαριθμητικό
>>> print custompath
/users/ThanosPC/pdfdocs/frequency spectrum.pdf     # με print εντολή δεν είναι αλφαριθμητικό
```

Επειδή ο διαχωριστής αρχείων '/' (file separator) με τον τρόπο που έχει γραφεί υποθέτει ότι ο κώδικας αναφέρεται σε μονοπάτια για Linux/Unix με αποτέλεσμα να ζημιώνει την φορητότητα του κώδικα σε άλλες πλατφόρμες. Αν λόγου χάρη επιθυμούμε να δημιουργήσουμε μονοπάτι ποιο φορητό που να είναι αξιοποιήσιμο και σε περιβάλλον που χρησιμοποιεί '\' ως file separator, τότε μπορούμε να βασιστούμε στην πιο αξιόπιστη λύση της συνάρτησης join() και παύουμε να ανησυχούμε για τους file separators:

```
>>> from os.path import join
>>> custompath=join(root, user, mydata,'frequency spectrum.pdf')
>>> print custompath
/users\ThanosPC\pdfdocs\frequency spectrum.pdf   #η εντολή join τοποθέτησε αυτόματα τους
                                                 #σωστούς file separators για τρέχον λειτουργικό
                                                 # που τρέχει ο κώδικας που είναι τα windows.
>>> custompath
'/users\\ThanosPC\\pdfdocs\\'frequency spectrum.pdf'    #τα διπλά backslash \\ συμβολίζουν
                                                        #τον backslash χαρακτήρα.
```

Για να αντιμετωπιστεί αυτόματα το πρόβλημα με τον forwardslash '/' στην λέξη /users του από πάνω μονοπατιού χρησιμοποιούμε μια άλλη συνάρτηση, την normpath που θα κάνει αυτόματη μετατροπή των file separators στη σωστή μορφή ανάλογα το λειτουργικό που τρέχει ο κώδικας και αφαιρεί τον τελεστή τελεία '.' που συμβολίζει το τρέχον κατάλογο και την διπλή τελεία που αναπαριστά ένα κατάλογο πίσω από τον τρέχον και τους επιπλέον file separators και που μπορεί να βάλαμε κατά λάθος όπως το τριπλό '///' π.χ. το '/users/ThanosPC//.///pdfdocs/../..//users/./pdfdocs/..' θα μετατραπεί σε: '\users\ThanosPC'. Έτσι, για τον από πάνω κώδικα έχουμε:

```
>>> from os.path import normpath
>>> path=normpath(custompath)
>>> print path
\users\ThanosPC\pdfdocs\frequency spectrum.pdf      #είναι ένα έγκυρο μονοπάτι στα windows.
```

Το επόμενο βήμα είναι αν τυχόν θέλουμε να αντλήσουμε το τελευταίο, πιο ακριανό κομμάτι ενός μονοπατιού πχ το τελευταίο κατάλογο ή αρχείο θα βασιστούμε στις μεθόδους dirname και basename.

```
>>> from os.path import dirname, basename
>>> print path
\users\ThanosPC\pdfdocs\frequency spectrum.pdf
>>> print dirname(path)
\users\ThanosPC\pdfdocs
>>> print basename(path)
frequency spectrum.pdf
```

Η μέθοδος dirname εξάγει από το σύνολο του μονοπατιού τον πιο τελευταίο και πιο ακριανό κατάλογο και τον τυπώνει μαζί με το κομμάτι του μονοπατιού από όπου προέρχεται. Η μέθοδος basename τυπώνει κυριολεκτικά πάντα το τελευταίο κομμάτι του μονοπατιού(το τέρμα δεξιά). Ένας εναλλακτικός τρόπος που ενσωματώνει τις δυο παραπάνω λειτουργίες σε μία συνάρτηση και να πετυχαίνει το ίδιο αποτέλεσμα είναι με την μέθοδο split:

```
>>> from os.path import split
>>> (head,tail)=split(path)          # Για τα head,tail είναι επιλογή του χρήστη οι ονομασίες.
>>> print head
\users\ThanosPC\pdfdocs
>>> print tail
frequency spectrum.pdf
```

Μια άλλη διαθέσιμη μέθοδος που κάνει την ίδια λειτουργία είναι η splitext <u>αφαιρώντας</u> την κατάληξη του ακριανού αρχείου δηλαδή το .pdf στην προκείμενη περίπτωση :

```
>>> from os.path import splitext
>>> path= '\users\ThanosPC\pdfdocs\frequency spectrum.pdf'
>>> (stem,tip)=splitext(path)     # Για τα stem,tip είναι επιλογή του χρήστη οι ονομασίες.
>>> print stem
\users\ThanosPC\pdfdocs\frequency spectrum
>>> print tip
.pdf
```

Κατόπιν για να εντοπίσουμε το ανώτερο επίπεδο, δηλαδή από ποιον σκληρό δίσκο πηγάζει ένα μονοπάτι:

```
>>> from os.path import splitdrive
>>> (OSdrive,tipp)= splitdrive(path)
>>> print OSdrive
            #κενό, καμία ένδειξη γιατί δεν ξεκίνησε το μονοπάτι από τον C: drive ή κάποιον άλλο
            #σκληρό δίσκο, ειδάλλως θα τύπωνε τον σκληρό δίσκο από όπου προήλθε.
>>> print tipp
\users\ThanosPC\pdfdocs\frequency spectrum.pdf
```

Όταν ορίζουμε ένα μονοπάτι και δεν είμαστε σίγουροι αν πρόκειται για απόλυτο ή σχετικό μονοπάτι χρησιμοποιούμε τις μεθόδους abspath, isabs οι οποίες **ελέγχουν μόνο το συντακτικό** για το δοθέν μονοπάτι είναι απόλυτο ή σχετικό αντίστοιχα, αλλά **δεν ελέγχουν αν όντως υπάρχει** αυτό το μονοπάτι ή όχι στο σκληρό δίσκο. Για την επιβεβαίωση της ύπαρξης του μονοπατιού γίνεται χρήση της μεθόδου exists. Για να υποχρεωθεί ένα σχετικό μονοπάτι να μετατραπεί σε απόλυτο θα γίνει χρήση της abspath.

```
>>> from os.path import isabs, abspath, exists
>>> path='pdfdocs/specs.pdf'
>>> isabs(path)
False      #άρα είναι σχετικό το μονοπάτι.
>>> exists(path)
False    #άρα δεν υπάρχει καν αυτό το μονοπάτι στον σκληρό δίσκο που να δείχνει στο αρχείο specs.pdf
```

```
# τώρα θα αναιρέσουμε τα δυο False.
>>> absolute_path=abspath(path)
>>> print absolute_path
C:\Python27\pdfdocs\specs.pdf
>>>exists(absolute_path)
True
>>> isabs(absolute_path)
True
>>> print abspath('pdfdocs/../..') # πηγαίνει δυο βήματα πίσω γιατί έγινε χρήση του /.. δυο φορές.
C:\
```

Τώρα αναιρέσαμε τα δυο False και καταφέραμε να φτιάξουμε ένα κώδικα που φτάνει μέχρι το αρχείο specs.pdf . Αν θέλουμε μπορούμε να το ανοίξουμε και να το διαβάσουμε πλέον με την εντολή open() όπως έχει δειχθεί σε προηγούμενη ενότητα. Σε περίπτωση που δεν υπήρχε στο σκληρό δίσκο του υπολογιστή στη συγκεκριμένη διαδρομή(C:\Python27\pdfdocs\specs.pdf) το αρχείο specs.pdf τότε δε θα παίρναμε True αλλά False. Να επισημανθεί ότι η μέθοδος abspath χρησιμοποιεί εσωτερικά την getcwd() για να χτίσει την σωστή διαδρομή. Πολύ απλά προσθέτει στην αρχή του μονοπατιού pdfdocs\specs.pdf ότι λείπει ώστε να γίνει απόλυτο ,δηλαδή το C:\Python27. Να τονιστεί ότι ο παραπάνω κώδικας παράγει τις δυο True τιμές για τον υπολογιστή που γράφτηκε μόνο. ΑΝ θέλουμε να προκύψουν τα δυο True σε κάποιον άλλο υπολογιστή θα πρέπει επίσης να διαθέτει το αρχείο specs.pdf με το συγκεκριμένο μονοπάτι C:\Python27\pdfdocs\specs.pdf να είναι υπαρκτό ή να το δημιουργήσουμε εμείς χειροκίνητα πριν τρέξουμε τον κώδικα.

Πίνακας 13. Σετ εντολών του os.path για περιβάλλον windows.

Μέθοδος	**Λειτουργία**
abspath(*path*)	Μετατρέπει ένα μονοπάτι σε απόλυτο μονοπάτι με χρήση της getcwd() μεθόδου.
basename(*path*)	Αντλεί το τελικό αρχείο/κατάλογο από το δοσμένο μονοπάτι.
commonprefix(*list*)	Επιστρέφει το μακρύτερο πρόθεμα(κομμάτι μονοπατιού) από όλα τα προθέματα που είναι αποθηκευμένα σε μια λίστα και που είναι κοινά.
dirname(*path*)	Αντλεί από το ζητούμενο μονοπάτι μέχρι το τελευταίο κομμάτι στοιχείο του μονοπατιού χωρίς αυτό.
exists(*path*)	Έλεγχος αν ήδη υπάρχει ένα μονοπάτι στο σκληρό δίσκο. Σε κάποιες πλατφόρμες αν δεν μπορεί να δοθεί άδεια εκτέλεσης της os.stat() για το ζητούμενο αρχείο επιστρέφει False.
lexists(*path*)	Ότι και η exists. Χρήσιμη για πλατφόρμες που δεν διαθέτουν την μέθοδο os.lstat()
expanduser(*path*)	Επιστροφή ορίσματος που ξεκινά με ~ ή ~user αντικατεστημένο με το home directory. Στα windows HOME και USERPROFILE θα εφαρμοστούν αν τεθούν (set).Διαφορετικά γίνεται χρήση των HOMEPATH και HOMEDRIVE. Το ~user προκύπτει αποκόπτοντας το τελευταίο κομμάτι ενός πλήρους μονοπατιού.
expandvars(*path*)	Επιστρέφει το όρισμα με επεκταμένες τις περιβαλλοντικές μεταβλητές. Αλφαριθμητικά τύπου substring με μορφή $name ή ${name} αντικαθιστώνται από την τιμή της περιβάλλοντος μεταβλητής name. Ανύπαρκτες μεταβλητές ή αναφορές σε ανύπαρκτες μεταβλητές παραμένουν απαράλακτες.
getatime(*path*)	Δίνει την ώρα σε δευτερόλεπτα που έγινε τελευταία φορά πρόσβαση στο ζητούμενο μονοπάτι. Μπορεί να είναι integer ή float ο χρόνος. Η τιμή εξαρτάται από το εκάστοτε OS.

getmtime(*path*)	Δίνει την ώρα σε δευτερόλεπτα που έγινε τελευταία φορά μετατροπή(modification) στο ζητούμενο μονοπάτι. Η τιμή μπορεί να είναι integer ή float. Η τιμή εξαρτάται από το εκάστοτε OS.
getctime(*path*)	Δίνει τη χρονική στιγμή δημιουργίας ενός μονοπατιού. Η τιμή εξαρτάται από το εκάστοτε OS.
getsize(*path*)	Επιστρέφει το μέγεθος του μονοπατιού σε bytes. Αν δεν υφίσταται ή δεν επιτρέπεται η πρόσβαση προκύπτει os.error .
isabs(*path*)	Ελέγχει αν το δοθέν μονοπάτι είναι όντως απόλυτο και δίνει True, αλλιώς False.Στα windows γίνεται αφαιρώντας το C:\ ή όποιο άλλο.
isfile(*path*)	Ελέγχει το συντακτικό που δόθηκε και επιστρέφει True αν είναι όντως αρχείο το τελευταίο κομμάτι του path.
isdir(*path*)	Ελέγχει το συντακτικό που δόθηκε και επιστρέφει True αν είναι όντως κατάλογος.
islink(*path*)	Επιστρέφει True εφόσον το path δείχνει σε directory entry(εγγραφή καταλόγου) που είναι συμβολικό link(σύνδεσμος).
ismount(*path*)	Επιστρέφει True αν το όνομα του path είναι σημείο εγκατάστασης(mount point): θέση εντός του file system(σύστημα αρχείων) όπου ένα διαφορετικό file system έχει εγκατασταθεί.
join(*path1*[, *path2*[, ...]])	Δημιουργία μονοπατιού με αυτόματο τρόπο, μέσα στις παρενθέσεις δίνουμε τα ορίσματα μονοπάτια και τα ενώνει σε ένα σώμα.
normcase(*path*)	Στα windows μετατρέπει τα '/' σε '\'. Επίσης για file systems που δεν είναι case-sensitive μετατρέπει όλους τους χαρακτήρες του δοθέντος path σε πεζά.
normpath(*path*)	Προσαρμόζει τα backslash και αφαιρεί περιττά στοιχεία από το δοθέν path ώστε να μπορεί να είναι συμβατό με το εκάστοτε OS.
realpath(*path*)	Επιστρέφει το κανονικό μονοπάτι του δοθέντος path και αφαιρεί ότι συμβολικά link παρεμβαίνουν ,αν υποστηρίζει τέτοια δυνατότητα το file system.
relpath(*path*[, *start*])	Επιστρέφει το σχετικό μονοπάτι είτα από το τρέχον κατάλογο είτε από ένα προαιρετικά ορισμένο σημείο εκκίνησης(κάποιο κατάλογο). Δεν ελέγχει αν όντως υπάρχει, μόνο το συντακτικό της διαδρομής ελέγχεται αν έχει λογική βάση. Ενθαρρύνεται η χρήση της curdir() method.
split(*path*)	Χωρίζει το path σε δυο κομμάτια με την εξής σύνταξη: (head,tail)=split(path). Σε νέα γραμμή γράφουμε ξεχωριστά: print head , print tail.
splitdrive(*path*)	Χωρίζει το path σε δυο κομμάτια (drive, tail). Ίδια μέθοδος συγγραφής με την split.Το drive= C:\ ή όποιο άλλο είναι.
splitext(*path*)	Χωρίζει το path σε δυο κομμάτια (root, ext). Ίδια μέθοδος συγγραφής με την split. Το root =όλο το μονοπάτι εκτός από το τελευταίο στοιχείο του μονοπατιού που έχει και κατάληξη με τελεία πχ abs.pdf , cdfertyfd.txt ,αυτά τα δίνει το ext.
splitunc(*path*)	Χωρίζει το path σε δυο κομμάτια (unc, rest) με unc=UNC mount point και το υπόλοιπο του μονοπατιού βρίσκεται στην rest. Αν το μονοπάτι περιέχει γράμματα από drives (π.χ. C:\) τότε το unc επιστρέφει πάντα άδειο string.
walk(*path*, *visit*, *arg*)	Καλεί την συνάρτηση visit με ορίσματα (arg, dirname, names)για κάθε κατάλογο μέσα σε ένα δέντρο-κατάλογο. Το dirname ορίζει τον επισκεπτόμενο υποκατάλογο. Το names παράγει μια λίστα με τα επισκεπτόμενα αρχεία στο εκάστοτε κατάλογο που τα αντλεί με την μέθοδο os.listdir(dirname)). Η συνάρτηση visit για βελτιστοποίηση μπορεί να τροποποιεί το names για να επηρεάσει το σετ των καταλόγων που επισκέπτεται εκατέρωθεν της dirname π.χ. για αποφυγή επίσκεψης κάποιων από τα παρακλάδια του δέντρου-καταλόγου.
supports_unicode_filenames	Αληθές αν εκάστοτε Unicode strings μπορούν να χρησιμοποιηθούν σαν ονόματα αρχείων.

Πηγή: http://docs.python.org/2/library/os.path.html (συνέχεια από σελίδα 58)

6.3.3 Το sys module

Με την βοήθεια αυτού του module μπορούμε να αποκτάμε πρόσβαση από την γραμμή εντολών(command line)σε συναρτήσεις μεθόδους, μεταβλητές που χρησιμοποιεί ο διερμηνευτής και αλληλεπιδρούν μαζί του προκειμένου να επιτελεστεί μια διεργασία. Μερικές από τις μεθόδους που μπορεί να μας πείσουν να το χρησιμοποιούμε συχνά θα μπορούσαν να είναι η argv, exit, stderr και άλλες. Στο σύνολο του sys.module διαθέτει μια ικανή συλλογή από υπηρεσίες στον χρήστη για αναζήτηση(probing) ή για τροποποίηση των ρυθμίσεων(configuration) του διερμηνευτή που είναι σε runtime mode και με πόρους για αλληλεπίδραση με το OS εκτός του τρέχοντος προγράμματος.

Ένα ενδιαφέρον στοιχείο που περιέχει αυτό το module και χρήζει μελέτης είναι το argv. Είναι συντομογραφία της λέξης arguments vector(διάνυσμα με ορίσματα) και περιέχει όλα τα ορίσματα που περνάνε σε ένα πρόγραμμα. Πρόκειται για μια λίστα, συνεπώς μπορούμε να αξιοποιούμε όλες τις δυνατότητες που μας προσφέρουν οι λίστες. Το πρώτο στοιχείο αυτής της λίστας, δηλαδή το argv[0],είναι πάντα το όνομα του αρχείου το όνομα δηλαδή του προγράμματος μας, τα υπόλοιπα μπορεί να είναι παράμετροι που εισάγουμε εμείς. Αν δεν περαστεί στον διερμηνευτή κανένα όνομα αρχείου τότε επιστρέφει μια κενή λίστα και στην περίπτωση που η εντολή argv εκτελεστεί με την επιλογή -c της γραμμής εντολών τότε το argv[0] θα τεθεί στο αλφαριθμητικό '-c'.

Είναι χρήσιμο το sys.argv γιατί κάποιες φορές αν θελήσουμε να εκτελέσουμε κατά επανάληψη το ίδιο πρόγραμμα με διαφορετικά ορίσματα. Έτσι, με αυτό το ένα εργαλείο θα μπορούσα λόγου χάρη να αποκτήσω πρόσβαση στα περιεχόμενα του τρέχοντος καταλόγου και μερικών ακόμα. Συνεπώς, μπορούμε να πούμε ότι μια βασική του λειτουργία είναι για να αναθέτουμε ορίσματα και να αποκτάμε πρόσβαση σε αυτά. Παρόλα αυτά πολλές εφαρμογές χρησιμοποιούν άλλες βιβλιοθήκες(όχι την sys) όπως η argparse για να αποκτάνε πρόσβαση στα παραπάνω ορίσματα.

Αν πρόκειται για ένα απλό σενάριο μπορούμε να αποφύγουμε την χρήση της argv και να στηριχθούμε στα modules fileinput, argparse, που ενσωματώνουν μέσα τους την sys.argv, την χρησιμοποιούν σιωπηρά χωρίς να γίνεται οπτικά αντιληπτό στον χρήστη, για πρόσβαση στις προαναφερθείσες μεταβλητές. Μπορούμε να πούμε ότι το sys.argv σε μια τέτοια περίπτωση δρα ως ο θεμέλιος λίθος(building block) για άλλα κομμάτια κώδικα. Συνεπώς, αν το επιθυμούμε, για ένα σύνολο από απλά scripts(σενάρια) προγράμματα που δίνονται σε command line(cmd) μπορούμε να χρησιμοποιήσουμε το fileinput module αντί για το sys.argv, ενώ για πιο περίπλοκα scripts το argparse αποφεύγοντας και πάλι να καταφύγουμε κατευθείαν στην sys.argv. Ένα παράδειγμα τι κάνει:

```
>>> import sys
>>> print str(sys.argv)
>>> print 'Plithos orismatwn: ', len(sys.argv)
```

Αν σώσουμε αυτό το απλό κομμάτι κώδικα σε ένα αρχείο script με κάποιο όνομα π.χ. file.py κατόπιν πάμε σε περιβάλλον γραμμής εντολών(cmd) αλλάζουμε το default μονοπάτι έτσι ώστε να δείχνει το τον φάκελο με το αρχείο μας (πχ. cd c:\pthon27) και πληκτρολογούμε χωρίς το δολάριο:

```
$ python file.py 12 –a "hello people"
```

Το αποτέλεσμα που παράγεται θα είναι:

```
['file.py', '12', '–a', 'hello people']
Plithos orismatwn: 4
```

Ένα ακόμα καλό παράδειγμα είναι σε περίπτωση που θέλουμε να μετατρέψουμε σχετικά μονοπάτια σε απόλυτα μονοπάτια(για όποιο OS δεν διαθέτει ήδη μια απλή έτοιμη ενσωματωμένη εντολή για αυτό το σκοπό):

```
>>> import os, sys
>>> print('\n'.join(os.path.abspath(arg) for arg in sys.argv[1:]))
```

Πίνακας 14. Το sys module για περιβάλλον windows, (όχι πλήρης)

Μέθοδος	**Λειτουργία**
argv	Μια λίστα που το πρώτο στοιχείο της είναι το όνομα του αρχείου και τα υπόλοιπα είναι τα δοθέντα ορίσματα.
byteorder	Πληροφόρηση αν το OS είναι Big endian ή Little endian. Χρήσιμο για επικοινωνία μεταξύ διαφορετικών συστημάτων.
builtin_module_names	Μια πλειάδα με μονάχα τα ονόματα όλων των modules που είναι μεταγλωττισμένα(compiled) για τον εκάστοτε συγκεκριμένο διερμηνευτή.
modules	Λεξικό που έχει ως keys-values για κάθε θέση ονόματα modules(keys) με modules(values) που έχουν ήδη φορτωθεί. Διαγραφή μιας θέσης δεν ισοδυναμεί με την reload(). Το modules.keys() δίνει μόνο τα keys και είναι πιο γενική από την builtin_module_names.
sys.call_tracing(*func*, *args*)	Κλήση μιας συνάρτησης με ορίσματα λίστα func(*args) ενόσω το tracing είναι ενεργοποιημένο. Χρήσιμη για εκσφαλμάτωση ξεκινώντας από ένα καθορισμένο σημείο από το χρήστη.
copyright	Αλφαριθμητικό που πληροφορεί τον χρήστη για τα πνευματικά δικαιώματα του διερμηνευτή.
_clear_type_cache()	Άδειασμα της εσωτερικής μνήμης cache. Χρήση για εξειδικευμένες περιπτώσεις.
_current_frames()	Λεξικό με περιεχόμενα τα thread ID με αντίστοιχα ενεργά frames μιας στοίβας τη στιγμή κλήσης της μεθόδου current frames. Χρήσιμο ιδιαίτερα σε περίπτωση αδιεξόδου στο debugging των threads.
dllhandle	Ένας ακέραιος. Ορίζει το αναγνωριστικό(handler) για κάποιο DLL της Python.
displayhook(*value*)	Εκτυπώνει ότι οριστεί ως value. Το αποτέλεσμα περνά ως όρισμα στο sys.stdout και απομνημονεύεται στο __builtin__._
dont_write_bytecode	Για ρύθμιση της παραγωγής αρχείων bytecode.Αν είναι αληθές(τροποποιείται με την επιλογή –B στη γραμμή εντολών) τότε δε θα γραφτούν .pyc ή .pyo αρχεία κατά την εισαγωγή πηγαίων modules.
excepthook(*type*, *value*, *traceback*)	Τυπώνεται ένα δοσμένο traceback και ένα exception στο stderr. Αν το exception δεν συλληφθεί θα κληθεί από τον interpreter η excepthook με τα τρία ορίσματα: exception class, exception instance, and a traceback object.
__displayhook__	Με την εκκίνηση του προγράμματος εδώ βρίσκεται η αρχικοποιημένη τιμή για την μέθοδο displayhook(value)
__excepthook__	Με την εκκίνηση του προγράμματος εδώ βρίσκεται η αρχικοποιημένη τιμή για την μέθοδο excepthook.
exc_info()	Πλειάδα με τρεις τιμές(*type*, *value*, *traceback*)που πληροφορεί για το επίμαχο exception. Αν δεν υφίσταται exception επιστρέφει τρία None.
exc_clear()	Για εξειδικευμένη χρήση, καθαρισμός πληροφοριών σχετικές με το πιο πρόσφατο exception. Αν κατόπιν καλέσουμε την exc_info() θα επιστρέψει τρία None.
exec_prefix	Αλφαριθμητικό που δίνει το συγκεκριμένο πρόθεμα καταλόγου για εγκατεστημένα αρχεία Python που εξαρτώνται από το OS. Προεπιλογή: '/usr/local'
executable	Αλφαριθμητικό που δίνει το απόλυτο μονοπάτι ενός Python εκτελέσιμου αρχείου. Σε περίπτωση αδυναμίας εμφάνισης του επιστρέφει None ή κενό.
exit([*arg*)	Έξοδος από το περιβάλλον της Python. Το προαιρετικό πεδίο [arg] έχει το μηδέν ως προεπιλογή.
exitfunc	Μπορεί να οριστεί από τον χρήστη ή κάποιο πρόγραμμα για καθαρισμό κατά την έξοδο ενός προγράμματος. Για κλήση πολλαπλών συναρτήσεων κατά τον τερματισμό γίνεται χρήση του atexit module.
flags	Κατάσταση των flags της γραμμής εντολών, δεκαέξι στο σύνολο τους:d,3,Q,Qnew,i,i,O ή OO,B,s,S,E,t ή tt,v,U,b,R .

float_info	Μια δομή που περιέχει πληροφορίες χαμηλού επιπέδου σχετικά με τον τύπο του float, την ακρίβεια ψηφίου και ψηφιακή εσωτερική αναπαράσταση του.
float_repr_style	Αλφαριθμητικό που δείχνει πως η repr() method θα δράσει σε float μεγέθη. Δυο δυνατές τιμές: short και legacy.
getcheckinterval()	Εμφανίζει το check interval του διερμηνευτή όπως προκύπτει από την συνάρτηση setcheckinterval()
getdefaultencoding()	Εμφανίζει την τρέχουσα προεπιλεγμένη κωδικοποίηση.
getdlopenflags()	Εμφανίζει την τρέχουσα τιμή των flags της dlopen(). Για περισσότερα συμβουλευόμαστε τα modules dl και DLFCN.
getfilesystemencoding()	Εμφανίζει το όνομα της κωδικοποίσησης για μετατροπή Unicode ονόματα αρχείων σε αρχεία συστήματος συμβατά. Στα windows 7 είναι 'mbcs'.
getrefcount(*object*)	Επιστρέφει μια αρίθμηση αναφοράς ενός αντικειμένου. Η τιμή θα είναι κατά ένα μεγαλύτερη από το αναμενόμενο.
getrecursionlimit()	Επιστροφή του τρέχοντος ορίου αναδρομής, σχετίζεται με τη μέγιστη δυνατή τιμή της στοίβας. Γίνεται set με την setrecursionlimit()
getsizeof(*object*[, *default*])	Επιστρέφει το μέγεθος ενός αντικειμένου σε bytes.
_getframe([*depth*])	Επιστρέφει ένα αντικείμενο frame από την καλούμενη στοίβα. Το depth σημαίνει επιστροφή του αντικειμένου που είναι πιο μέσα από την επιφάνεια της στοίβας.
getprofile()	Επιστρέφει ότι προκύπτει από την setprofile()
gettrace()	Επιστρέφει ότι προκύπτει από την settrace()
getwindowsversion()	Επιστρέφει την έκδοση των windows.
hexversion	Ο αριθμός έκδοσης σε μορφή ακεραίου. Ταιριάζει αρμονικά με την hex().
long_info	Δομή για την εσωτερική αναπαράσταση των ακεραίων
maxint	Μέγιστος υποστηριζόμενος ακέραιος
maxsize	Μέγιστος θετικός ακέραιος, άρα και τις μέγιστες διαστάσεις για λίστες, πλειάδες κλπ
maxunicode	Δίνει το μέγιστο κωδικό αριθμό για Unicode χαρακτήρα. Η μέγιστη τιμή εξαρτάται αν είναι UCS-2 or UCS-4.
meta_path	Λίστα από finder αντικείμενα που διαθέτουν find_module() μεθόδους καλούνται για να φανεί αν ένα από τα αντικείμενα μπορεί να βρει το προς εισαγωγή module. Το find_module() καλείται με τουλάχιστον το απόλυτο όνομα του εισαγόμενου module. Σε περίπτωση μη εύρεσης προκύπτει None, αλλιώς επιστρέφεται ένας loader.
path	Δίνει το μονοπάτι αναζήτησης των modules. Αρχικοποίηση με την περιβαλλοντική μεταβλητή PYTHONPATH. Το path[0] δίνει το κατάλογο που περιέχει το σενάριο που χρησιμοποιήθηκε για να καλέσει τον διερμηνευτή της Python.
setprofile(*profilefunc*)	Θέτει την συνάρτηση προφίλ του συστήματος. Χρήσιμο για χρήση profilers.
settrace(*tracefunc*)	Θέτει την συνάρτηση trace του συστήματος. Για υποστήριξη πολλαπλών νημάτων ο debugger πρέπει να τεθεί για κάθε νήμα προς εκσφαλμάτωση.
stdin, stdout,stderr	Αρχεία αντικείμενα. Το stdin χρησιμοποιείται για όλες τις εισροές με την input και raw_input.εκτός από scripts. Η stdout για έξοδο και εκτύπωση.
tracebacklimit	Αν τεθεί σε ακέραια τιμή, καθορίζει μέγιστο αριθμό πληροφόρησης σε περίπτωση σφαλμάτων. Προεπιλογή το 1000. Το 0 αποκρύπτει τα πάντα.
version	Η εγκατεστημένη έκδοση του διερμηνευτή της Python στον εκάστοτε Η/Υ.
api_version	Η C API έκδοση του διερμηνευτή. Χρήσιμο στο debugging με version conflicts.
winver	Ο αριθμός έκδοσης που χρησιμοποιείται από την registry των windows. Τυπικά είναι οι τρεις πρώτοι χαρακτήρες από το version method.

Πηγή: http://docs.python.org/2/library/sys.html#sys.getdlopenflags (συνέχεια από σελίδα 61)

6.3.4 Το shutil module

Το shutil module είναι συντομογραφία των λέξεων 'shell utilities', προσφέρει έναν αριθμό από υψηλού επιπέδου λειτουργίες για χειρισμό αρχείων και καταλόγων όπως η μεταφορά, και η αντιγραφή αρχείων, η αρχειοθέτηση και η μεταβολή επιτρεπτών ενεργειών(permissions) μεταξύ άλλων. Για επεμβάσεις σε μοναχικά(individual) αρχεία μπορούμε να χρησιμοποιήσουμε το os module επίσης.

Για αντιγραφή περιεχομένων των αρχείων μπορούμε να κάνουμε χρήση της μεθόδου copyfile(), η οποία αντιγράφει τα περιεχόμενα της πηγής στον προορισμό. Σε περίπτωση λάθους ,IOError, αυτό σημαίνει ότι δεν έχουμε άδεια πρόσβασης(permission) για να γράψουμε το αρχείο στον καθορισμένο προορισμό. Εξαιτίας του ότι η συνάρτηση ανοίγει το αρχείο εισόδου (input file) για ανάγνωση, αρχεία ειδικού τύπου δεν μπορούν να αντιγραφούν ως νέα ειδικά αρχεία με αυτή την εντολή. Ακόμη, κάτι πρέπει να θυμόμαστε είναι ότι σε περιβάλλον windows file owners, ACLs και εναλλακτικές ροές δεδομένων δεν αντιγράφονται. Ένα απλό παράδειγμα είναι να υποθέσουμε ότι έχουμε δυο αρχεία txt τα a.txt και b.txt. Το a.txt με το εξής κείμενο:

Login Credentials:
Username: Administrator
Password : Admin1234

Μπορούμε να αντιγράψουμε το a.txt στο b.txt που βρίσκονται στο ίδιο directory(κατάλογο) με μια εντολή μόνο:

```
>>> copyfile("a.txt","b.txt")
```

Σε περίπτωση που δεν υφίσταται το b.txt θα δημιουργηθεί αυτόματα και θα βρίσκεται στον ίδιο κατάλογο με το a.txt (την πηγή). Μετά μπορούμε με την open() να ανοίξουμε το b.txt το και να το διαβάσουμε με την readlines() και να διαγράψουμε την πηγή με την del() όπως έχουμε δει. Ένα ακόμη παράδειγμα σε μορφή συνάρτησης είναι το ακόλουθο:

```
import shutil
 def copypaste(source, destination):
   try:
      shutil.copy(source, destination)
    #περίπτωση πηγή και προορισμός να συμπίπτουν
   except shutil.Error as x:
      print('Error type: %s' % x)
   # αν τυχόν δεν υφίσταται προορισμός
   except IOError as x:
      print('Error type: %s' % x.strerror)
```

Πέρα από την αντιγραφή αρχείων ,είναι δυνατόν η αντιγραφή των καταλόγων με την βοήθεια των κατάλληλων εντολών όπως φαίνεται στον ακόλουθο αλγόριθμο:

```
>>> import shutil
>>> shutil.copytree('C:/Users/mypc/Desktop/a_dir', 'C:/Users/mypc/Desktop/b_dir/a_dir')
>>> shutil.rmtree('C:/Users/mypc/Desktop/a_dir')
```

Τα δυο αρχεία προς επεξεργασία βρίσκονται στην επιφάνεια εργασίας(Desktop) στο παράδειγμα αυτό. Με την copytree(πηγή, προορισμός) αντιγράφτηκε ολόκληρος ο κατάλογος της πηγής εντός του προορισμού ,δηλαδή όλο το a_dir βρίσκεται εντός του b_dir. Κατόπιν, διαγράψαμε το a_dir εντελώς με την εντολή rmtree(target), όπου το target ισούται με το μονοπάτι(path) που βρίσκεται ο στόχος μας δηλαδή το a_dir . Η εντολή move κάνει την ίδια δουλειά με λίγο διαφορετικό συντακτικό στον προορισμό:

```
>>> shutil.move('C:/Users/mypc/Desktop/a_dir', 'C:/Users/mypc/Desktop/b_dir')
```

Οι παραπάνω γραμμές κώδικα μπορούν να γίνουν πιο χρήσιμοι αν τους περικλείσουμε μέσα σε μια συνάρτηση:

```
import shutil
 def copyThisDirectory(source, destination):
   try:
      shutil.copytree(source, destination)
   # Directories are the same
   except shutil.Error as x:
      print('Directory not copied. Error: %s' % x)
   # Any error saying that the directory doesn't exist
   except OSError as x:
      print('Directory not copied. Error: %s' % x)
```

Μια παραλλαγή θα ήταν να αντιγράψουμε μόνο το περιεχόμενο του a_dir που είναι το αρχείο login.txt στον προορισμό που είναι το b_dir με τον εξής τρόπο:
>>> shutil.copy('C:/Users/mypc/Desktop/a_dir/login.txt', 'C:/Users/mypc/Desktop/b_dir')

Παρακάτω, με την μέθοδο copyfileobj(fsrc, fdst[, length]) γίνεται αντιγραφή των αρχείων που μιμούνται αντικείμενα (file-like objects). Το συντακτικό είναι τυπικό με πηγή το fsrc, προορισμό το fdst και μέγεθος του buffer να δίνεται από την τιμή length. Μια αρνητική τιμή σημαίνει αντιγραφή των δεδομένων μονομιάς, διαφορετικά η προεπιλογή είναι να γίνεται ανάγνωση των δεδομένων σε δόσεις για αποφυγή προβλημάτων υπερφόρτωσης με την μνήμη. Ακόμα, αν η τρέχουσα θέση(σημείο εκκίνησης ανάγνωσης) του αντικειμένου fscr της πηγής δεν είναι 0, τότε μόνο τα περιεχόμενα από την τρέχουσα θέση του αρχείου μέχρι το τέλος(ότι υπολείπεται έως το τέλος) θα αντιγραφούν κι όχι ότι βρίσκεται πιο πριν. Στο παρακάτω σχήμα τυπώνονται τα περιεχόμενα του αρχείου: pyparsing-wininst.log(πηγή) στην τυπική έξοδο(προορισμός) που είναι η οθόνη.

```
>>> import sys
>>> copyfileobj(open("pyparsing-wininst.log"), sys.stdout)
*** Installation started 2013/10/23 22:33 ***
Source: C:\Users\Thanos\Downloads\python programs\pyparsing-2.0.1.win32-py2.7.exe
999 Root Key: HKEY_LOCAL_MACHINE
040 Reg DB Value: [Software\Microsoft\Windows\CurrentVersion\Uninstall\pyparsing-py2.7]DisplayName=Pytho
n 2.7 pyparsing-2.0.1
040 Reg DB Value: [Software\Microsoft\Windows\CurrentVersion\Uninstall\pyparsing-py2.7]UninstallString="
C:\Python27\Removepyparsing.exe" -u "C:\Python27\pyparsing-wininst.log"
200 File Copy: C:\Python27\Lib\site-packages\pyparsing-2.0.1-py3.3.egg-info
200 File Copy: C:\Python27\Lib\site-packages\pyparsing.py
200 File Copy: C:\Python27\Lib\site-packages\pyparsing.pyc
200 File Copy: C:\Python27\Lib\site-packages\pyparsing.pyo
```

Σχήμα 17. Η copyfileobj (έγινε χρήση νωρίτερα της εντολής **from** shutil **import** copyfileobj)

Ένα παράδειγμα χρήση της copyfileobj μέσα σε συνάρτηση:

```
import shutil
def CopyThisFile(source, destination, buffer_length=16*1024): # buffer_length μια αυθόρμητη τυχαία
                                                               #τιμή.
   with open(source, 'r') as fsrc:
      with open(destination, 'w') as fdest:
         shutil.copyfileobj(fsrc, fdest, buffer_length)
```

Πρώτα έγινε εισαγωγή με την import του επιθυμητού Module κατόπιν ορίστηκε μια συνάρτηση με όνομα CopyThisFile με παράμετρο στο μέγεθος του buffer αρκετά μεγάλο ώστε να διαβάσει μονομιάς ένα πολύ μεγάλο αρχείο. Κατόπιν με το σχηματισμό with-as ορίστηκε τι ακριβώς αναπαριστούν τα ορίσματα fsrc και fdest. Τυπικά η as είναι ένας τρόπος μετονομασίας ενός αντικειμένου με νέο όνομα.

Παρακάτω δίνεται ένας συγκριτικός πίνακας με τις ιδιότητες και δυνατούς τρόπους χρήσης της αντιγραφής με το shutil module.

Πίνακας 15. Συναρτήσεις για αντιγραφή αρχείων με το shutil module

Συνάρτηση	**αντιγράφει Permissions**	**αντιγράφει Metadata**	**Καθορισμός τιμής Buffer**
shutil.copy	Ναι	Όχι	Όχι
shutil.copy2	Ναι	Ναι	Όχι
shutil.copyfile	Όχι	Όχι	Όχι
shutil.copyfileobj	Όχι	Όχι	Ναι

Μια πιο συγκεντρωτική λίστα με τις διάφορες μεθόδους αυτού του module είναι η ακόλουθη:

copy	copy(source,destination) Αντιγράφει το περιεχόμενο του αρχείου πηγής, δημιουργεί ή ξαναγράφει το αρχείο προορισμού. Σε περίπτωση που ο προορισμός είναι κατάλογος, ο προορισμός είναι ένα αρχείο με το ίδιο basename όπως της πηγής. Αντιγράφονται επίσης τα bits άδειας πρόσβασης, εκτός από την τελευταία φορά πρόσβασης και τροποποίησης
copy2	copy2(source,destination) Επιπλέον της λειτουργίας της copy, αντιγράφει επίσης τις φορές τελευταίας πρόσβασης και τροποποίησης.
copyfile	copyfile(source,destination) Αντιγράφειμόνο τα περιεχόμενα της πηγής(όχι τα permission bits, ούτετους χρόνους τελευταίας πρόσβασης και τροποποίησης), δημιουργώντας ή ξαναγράφοντας τον προορισμό.
copyfileobj	copyfileobj(fscr,fdst, bufsize=τιμή) Αντιγράφει όλα τα byte της πηγής έως την μέγιστη θετική bufsize τιμή αφού πρώτα έχει ανοιχτεί το αρχείο σε read mode.
copymode	copymode(source,destination) Αντιγράφει τα permission bits της πηγής για αρχείο ή κατάλογο και τα δίνει στον προορισμό(αρχείο ή κατάλογο).Επιβάλλεται πηγή και προορισμός να υπάρχουν. Στον προορισμό δεν αλλάζουν τα περιεχόμενα ούτε το στάτους.
copystat	copystat(source,destination) Αντιγράφει από την πηγή τα permission bits και τους χρόνους τελευταίας πρόσβασης και τροποποίησης στον προορισμό. Επιβάλλεται πηγή και προορισμός να υπάρχουν. Στον προορισμό δεν αλλάζουν τα περιεχόμενα ούτε το στάτους.
copytree	copytree(source,destination, symlinks=False) Αντιγράφει ολόκληρο τον κατάλογο δέντρο που αναπαριστά την πηγή source στον προορισμό ο οποίος δεν πρέπει να προϋπάρχει γιατί η εντολή αυτή το δημιουργεί. Στα windows δεν χρησιμοποιείται το όρισμα symlinks.

move	move(source,destination) Μεταφορά αρχείου ή καταλόγου από την πηγή στον προορισμό. Αρχικά, γίνεται απόπειρα με την os.rename().Αν αποτύχει(π.χ. διαφορετικό file system πηγής και προορισμού, προορισμός προϋπάρχει) γίνεται χρήση της copy2() για αρχείο ή copytree για κατάλογο και επιπλέον διαγράφεται η πηγή σε κάθε περίπτωση(γι αυτό προσοχή).
rmtree	rmtree(path[, ignore_errors[, onerror]]) Διαγραφή ολόκληρου του δέντρου καταλόγου όπως ορίζει το path. Αν ignore_errors=True τότε η rmtree αγνοεί τα σφάλματα. Αν το ignore_errors=False και onerror=None τότε η rmtree πρέπει να είναι με δυνατότητα κλήσης(callable) με τρία ορίσματα: την func που προκαλεί(raise) ένα exception(os.remove ή os.rmdir), το path που περνά στην func ως όρισμα και η expc που είναι μια πλειάδα με στοιχεία που παρέχει η sys.exc.info(). Τέλος, αν η onerror προκαλέσει ένα exception, η rmtree τερματίζεται και το εκάστοτε exception διαδίδεται.

6.3.5 Το glob module

Αυτό το module παρέχει την δυνατότητα εύρεσης όλων των μονοπατιών(pathnames) που θα ταιριάζουν με ένα συγκεκριμένο pattern(δηλαδή μοτίβο) που ορίζεται από τον χρήστη ή κάποιο πρόγραμμα σύμφωνα με τους κανόνες που διέπουν ένα Unix shell. Υποστηρίζονται οι στάνταρ wildcards(σχηματο-μορφο-ταιριαστοί χαρακτήρες): ("*, "?" , "[ ]"). Το σύμβολο [ ] δηλώνει εύρος χαρακτήρων. Τεχνικά αυτό γίνεται με χρήση των μεθόδων os.listdir() και fnmatch.fnmatch() σε συνεργασία με ένα subshell(χωρίς να το επικαλούνται).Να επισημανθεί ότι η glob αντιμετωπίζει τα ονόματα αρχείων που ξεκινούν με τελεία ως εδικές περιπτώσεις κάτι που δεν ισχύει π.χ. με την fnmatch.fnmatch(). Επιπλέον, για την περισπωμένη ~ (tilde) και επεκτάσεις μεταβλητών κέλυφους(shell variable expansions) γίνεται χρήση των os.path.expanduser() και os.path.expandvars(). Γόνιμο είναι να προστεθεί πως το * (wild-star) δεν επιδρά μέσα σε έγγραφα αλλά σε αναζήτηση αρχείων και καταλόγων. Τέλος, για ταίριασμα κυριολεκτικών σταθερών(literals) πρέπει να περικλείονται οι χαρακτήρες σε 'αυτάκια' π.χ. '[?]' αντιστοιχεί στον '?' .

```
>>>
>>> import glob,os
>>> os.chdir(r'c:\python27\pyfiles')
>>> glob.glob('*.gif')
['dasdasd.gif', 'NewTest.gif']
```

Σχήμα 18. Εντοπισμός με δυο γραμμές κώδικα και εμφάνιση μέσα στον κατάλογο Pyfiles όλων των αρχείων με κατάληξη gif .

```
>>> glob.glob(r'c:\python27\*.exe')
['c:\\python27\\python.exe', 'c:\\python27\\pythonw.exe', 'c:\\
python27\\Removepyparsing.exe', 'c:\\python27\\w9xpopen.exe']
```

Σχήμα 19. Εντοπισμός με μια γραμμή κώδικα και εμφάνιση μέσα στον κατάλογο Python27 όλων των αρχείων με κατάληξη exe .

Αν και αυτό το module είναι αρκετά απλό εντούτοις είναι πολύ πρακτικό. Μπορεί να φανεί χρήσιμο σε περιστάσεις που θέλουμε λόγου χάρη να αναζητήσουμε για μια λίστα αρχείων στο σύστημα αρχείων(file system) που ικανοποιούν συγκεκριμένα κριτήρια αναζήτησης. Έτσι, αντί να φτιάξουμε ένα ολόκληρο

υποπρόγραμμα που να σαρώνει ένα κατάλογο για να βρει την επιθυμητή πληροφορία(εγγραφή) μπορούμε απλούστατα να βασιστούμε αρκετές φορές στην glob. Παράδειγμα συγκεκριμένων κριτηρίων θα μπορούσαν να είναι όπως φαίνεται στα σχήματα 18 και 19 η αναζήτηση συγκεκριμένων αρχείων με συγκεκριμένα extensions(επεκτάσεις). Άλλο κριτήριο θα μπορούσε να είναι η αναζήτηση ενός αλφαριθμητικού μέσα σε ένα μακροσκελές κείμενο, η αναζήτηση αρχείων με επιθυμητό πρόθεμα, φωτογραφιών και τα λοιπά.

Όταν κάνουμε αναζήτηση αρχείων υπάρχει μια ιδιορρυθμία με τα αρχεία που ξεκινά το όνομα τους με τελεία από τα υπόλοιπα αρχεία. Αυτά τα αρχεία δεν μπορούν να εντοπιστούν γι αυτό όταν χρησιμοποιούμε την glob για εύρεση αρχείων που αρχίζουν με τελεία καλό είναι για σιγουριά να προσθέτουμε και ένα επιπλέον σύμβολο. Έτσι, αν θεωρήσουμε ότι στο μονοπάτι C:/Python27 εντός του φακέλου Python27 υπάρχουν δυο αρχεία το 'ztest.py' και το '.ztest.py' τότε :

```
>>> import os,glob
>>> os.chdir('C:/Python27')
>>> glob.glob('.z*')
['.ztest.txt']
>>> glob.glob('*.txt')
['LICENSE.txt', 'NEWS.txt', 'README.txt', 'Sample.txt', 'words.txt', 'ztest.txt']
# Ενώ:
>>> glob.glob('*.z')
[]
```

Από τα παραπάνω γίνεται εύκολα αντιληπτό ότι η αναζήτηση αρχείων μέσα σε καταλόγους γίνεται ως εξής:

```
# εμφάνιση όλων των φακέλων που βρίσκονται εντός του καταλόγου Tools:
>>> import glob
>>> for item in glob.glob('C:\\Python27\\Tools/*'):
        print item
```

Το αποτέλεσμα είναι:

```
C:\Python27\Tools\i18n
C:\Python27\Tools\pynche
C:\Python27\Tools\Scripts
C:\Python27\Tools\versioncheck
C:\Python27\Tools\webchecker
```

Προφανώς για αναζήτηση μέσα στον υποκατάλογο i18n θα πρέπει να συμπεριληφθεί στον κώδικα :

```
>>> import glob
>>> for item in glob.glob('C:\Python27\Tools\i18n/*'):
        print item
```

Το αποτέλεσμα είναι:

```
C:\Python27\Tools\i18n\makelocalealias.py
C:\Python27\Tools\i18n\msgfmt.py
C:\Python27\Tools\i18n\pygettext.py
```

Αν μέσα σε ένα κατάλογο υπάρχει μόνο ένας υποκατάλογος τότε μπορούμε να περιπατήσουμε εντός αυτού με star wildcard * :

```
>>> for item in glob.glob('C:\Python27\Tools\pynche/*/*'):
        print item
```

Το αποτέλεσμα είναι να εμφανίσει τη διαδρομή για τα δυο διαθέσιμα αρχεία:

```
C:\Python27\Tools\pynche\X\rgb.txt
C:\Python27\Tools\pynche\X\xlicense.txt
```

 Copyright © 2014

Ο wildcard χαρακτήρας '?' δρα για ένα μόνο χαρακτήρα και λειτουργεί ως εξής: Έστω ότι έχουμε ένα κατάλογο με έξι txt αρχεία, τα pass1, pass2, pass_, passw, password και το pas.

```
>>> import glob
>>> for item in glob.glob('C:/Python27/pass?.txt'):
        Print item
```

Αποτέλεσμα εκτύπωσης:

```
C:/Python27/pass1.txt
C:/Python27/pass2.txt
C:/Python27/pass_.txt
C:/Python27/passw.txt
```

Αυτό που κάνει ο αλγόριθμος είναι να πάει να ψάξει στον κατάλογο Python27 και να εκτυπώσει όλα τα αρχεία που έχουν συνολικά μήκος ακριβώς 5 χαρακτήρων, περιλαμβάνουν οπωσδήποτε την *μήτρα* που είναι το 'pass' στην προκείμενη περίπτωση και ένα ακόμη χαρακτήρα οποιοσδήποτε κι αν τύχει να είναι. Με άλλα λόγια όταν θέλουμε να βρούμε κάποιο ή κάποια αρχεία μπορούμε με κατάλληλο τρόπο να ορίζουμε μια μήτρα και στο τέλος της να τοποθετούμε ένα αγγλικό ερωτηματικό, αυτό βοηθά για παράδειγμα σε αρχεία που έχουν παρόμοιο όνομα αλλά επιλέξαμε να τα αποθηκεύουμε με το τελευταίο γράμμα έναν διαφορετικό χαρακτήρα σύμβολο για να διαφοροποιούνται.

Κλείνοντας το θέμα με αυτό το module θα δειχθεί ο τρόπος λειτουργίας για το εύρος χαρακτήρων [ ]κι όχι μόνο έναν απλό χαρακτήρα όπως δείχθηκε παραπάνω με το Αγγλικό ερωτηματικό.

```
>>> import glob
>>> for item in glob.glob('C:/Python27/passfile/*[0-9].*'):
        print item
```

Αποτέλεσμα εκτύπωσης:

```
C:/Python27\passfile\pass1.txt
C:/Python27\passfile\pass2.txt
C:/Python27\passfile\pass3.txt
```

Στο συγκεκριμένο παράδειγμα έγινε χρήση των αριθμών 0-9 εντός των [ ] .Συνεπώς, το εύρος οφείλει να είναι ιεραρχημένο βασιζόμενο στο χαρακτήρα κωδικό για κάθε ψηφίο/γράμμα. Η παύλα μεταξύ του 0 και 9 τονίζει ότι πρόκειται για ένα αδιαίρετο, συνεχόμενο (μη διακοπτόμενο) εύρος ακολουθιακών αριθμών που περιέχονται μέσα στον κατάλογο passfile στα ονόματα των αρχείων.

6.4 Μετρώντας το χρόνο

Σε αυτή την ενότητα θα δοθεί μεγάλο κομμάτι της προσοχής σε δυο χρήσιμα modules που είναι άρρηκτα συνδεδεμένα με το χρόνο, το datetime και το time. Πιο συγκεκριμένα, το time module χρησιμοποιεί συναρτήσεις από την C βιβλιοθήκη με τα ίδια ονόματα. Ας μην ξεχνάμε ότι η Python αναπτύχθηκε με την C γλώσσα. Ο τρόπος ορισμού η/και χρήσης αυτών των συναρτήσεων εξαρτάται από το εκάστοτε OS θα διαφέρουν λίγο πολύ από πλατφόρμα σε πλατφόρμα. Μέσα σε αυτό το module ενυπάρχουν συναρτήσεις για χειρισμό, τροποποίηση και επεξεργασία πληροφορίας του χρόνου. Από την άλλη, το datetime module παρέχει συναρτήσεις για να εξάγουμε χρήσιμες πληροφορίες για το χρόνο (ημερομηνία και ώρα).

Για το time module μια από τις θεμελιώδες συναρτήσεις είναι η time() η οποία επιστρέφει τον αριθμό των δευτερολέπτων ως μια τιμή τύπου float θεωρώντας ένα σημείο εκκίνησης ,ένα σημείο αναφοράς που λέγεται *epoch* και στα windows είναι το έτος 1970(ορίζεται παρακάτω πιο συγκεκριμένα).

```
>>> import time
>>> print 'Current time: ', time.time()
Current time:  1388610845.53
# β' τρόπος είναι με χρήση alias με την βοήθεια της as:
```

```
>>> from time import time as t
>>> print t()
1388610845.53
```

Επειδή το αποτέλεσμα είναι σε μια δεκαδική μορφή που ναι μεν εξυπηρετεί για αποθήκευση και σύγκριση με άλλες τιμές(ημερομηνίες), παρόλα αυτά δεν είναι σε μορφή που να είναι άμεσα αναγνώσιμο το format στον απλό χρήστη. Αυτό μπορεί να διορθωθεί με την βοήθεια της ctime.

```
>>> import time
>>> print 'The time is :', time.ctime()
>>> future = time.time() + 30
>>> print '30 secs later : ', time.ctime(future)
>>> print time.ctime()
Fri Jan 03 20:18:36 2014
```

Αν προσέξουμε τον κώδικα έγινε χρήση της συνάρτησης time.time(). Ας εξετάσουμε λίγο πιο προσεκτικά αυτή την συνάρτηση. Η time.time() παρέχει την τρέχουσα τιμή ώρας χρησιμοποιώντας μια μετρίσιμη έννοια που λέγεται *τικ*(tick) με σημείο εφαρμογής(epoch) την ημερομηνία 1/1/1970 , ώρα 12 το πρωί που ορίζεται και ως *epoch*. Τα *τικ* είναι μετρήσιμες χρονικές στιγμές, όχι απαραίτητα ακέραιοι αριθμοί και τα χρονικά διαστήματα είναι κινητής υποδιαστολής αριθμοί με μονάδες μέτρησης τα δευτερόλεπτα ή κάποια υποδιαίρεση τους. Έτσι ο αριθμός των τικ από το σημείο χρονικής εφαρμογής (epoch, 1/1/1970,ώρα 12:00am) είναι:

```
>>> import time
>>> time.time()
1388779176.905   # αριθμός των τικ από σημείο αναφοράς.
```

Αυτός ο αριθμός δεν είναι στατικός αλλά συνεχώς αυξάνει κάθε δευτερόλεπτο που διανύεται. Το προφανές συμπέρασμα είναι ότι ημερομηνίες πριν την epoch δεν μπορούν να αναπαρασταθούν με αυτή τη μορφή, όπως επίσης το άνω επιτρεπτό όριο(άνω φράγμα) είναι το 2038 για αναπαράσταση με την παραπάνω μορφή, αυτό τουλάχιστον για Unix και Windows. Παρόλα αυτά , αυτή η δεκαδική μορφή(π.χ. 1388779176.905) εξυπηρετεί για υπολογισμούς ημερομηνιών δια μέσω των τικ εντός επιτρεπτών αριθμητικών ορίων όπως μόλις ορίστηκαν.

Ένα άλλο ενδιαφέρον δομικό στοιχείο του time module είναι η δομή struct_time το οποίο δομείται ως μια πλειάδα με εννέα στοιχεία από το 0 έως το 8:

0 tm_year (έτος)	π.χ. 2014
1 tm_mon(μήνας)	1 έως 12
2 tm_mday(μέρα)	1 έως 31
3 tm_hour(ώρα)	0 έως 23
4 tm_min(λεπτά)	0 έως 59
5 tm_sec(δευτερόλεπτα)	0 έως 61 (με 60 ή 61 να είναι leap-seconds)
6 tm_wday(μέρα εβδομάδας)	0 έως 6 (0 είναι η Δευτέρα)
7 tm_yday(μέρα έτους)	1 to 366 (θεωρώντας Ιουλιανή μέρα)
8 tm_isdst	Για DST ώρα

Με βάση αυτό η τρέχουσα τοπική ώρα στην γεωγραφική περιοχή που βρίσκεται σε μια δεδομένη στιγμή ο χρήστης σε μορφή πλειάδας αρχικά και κατόπιν σε φορμαρισμένη(formatted) συμβατική μορφή είναι:

```
>>> import time # αν γράψουμε import time; δεν είναι λάθος γιατί γίνεται χρήση της C library.
>>> localtime = time.localtime(time.time())
>>> print " Current local time :", localtime
Current local time : time.struct_time(tm_year=2014, tm_mon=1, tm_mday=2, tm_hour=22,
tm_min=01, tm_sec=41, tm_wday=4, tm_yday=3, tm_isdst=0)
# Σχόλιο: αυτό το κομμάτι κώδικα θα μπορούσε να γραφτεί εναλλακτικά και με alias:
```

```
# Σχόλιο:from time import time as t
# Σχόλιο:from time import localtime
# Σχόλιο:print localtime(t())
# time.struct_time(tm_year=2014, tm_mon=1, tm_mday=2, tm_hour=22, tm_min=01, tm_sec=41,
#tm_wday=4, tm_yday=3, tm_isdst=0)
# τέλος σχόλιου
>>> formatted_localtime = time.asctime( time.localtime(time.time()) )
>>> print formatted_localtime
Fri Jan 03 22:22:38 2014
>>> from datetime import datetime  # εναλλακτικός τρόπος απεικόνισης τρέχοντος χρόνου
>>> print str(datetime.now())  # χωρίς την εντολή str επιστρέφει πλειάδα.
2014-01-03  22:32:39. 375000
>>> print datetime.now().strftime('date: %d-%m-%Y , time: %H:%M:%S')
>>> date: 03-01-2014 , time: 22:46:22
# η ώρα σε μορφή πλειάδας δίνεται παρακάτω
>>> now= datetime.now()
>>> thetimeis=(now.hour, now.minute, now.second)
>>> print thetimeis
(23, 06, 50)   # (ώρα, λεπτά, δευτερόλεπτα)
```

Για μια ακριβή ρύθμιση παραμέτρων και σειρά εμφάνισης του κάθε αριθμού εξαρτάται ανάλογα με ποια σειρά θα τοποθετηθούν από τον προγραμματιστή οι παράμετροι %d-%m-%Y και %H:%M:%S' αν αλλάξουν θέσεις τότε αλλάζει το αποτέλεσμα. Επισημαίνεται ότι παίζει ρόλο το αν είναι κεφαλαία ή πεζά τα εξής: %d, %H, %S, %I δεν πρέπει να τα αλλάζουμε, τα γράφουμε ως έχουν αλλιώς προκύπτει ValueError. Το %X στη θέση του %d αποδίδει την τρέχουσα ώρα σε 24ωρη μορφή, ενώ για 12h μορφή πληκτρολογούμε %I:%M:%S .

Ενώ σε κάποιες γλώσσες προγραμματισμού μπορεί ενδεχομένως η δημιουργία μιας απλής χρονοκαθυστέρησης να μην είναι μια τόσο εύκολη διαδικασία αλλά ταλαιπωρία, στην Python μια απλή χρονοκαθυστέρηση γίνεται εύκολα με την εντολή sleep(seconds).Έτσι για παράδειγμα μια χρονοκαθυστέρηση του ενός δευτερολέπτου θα γίνει:

```
>>> import time
>>> time.sleep(1) # ένα δευτερόλεπτο delay, απραξία ενός δευτερολέπτου
>>> time.sleep(10) # δέκα δευτερόλεπτα delay.
>>> import operator      # για να γίνει χρήση της συνάρτησης div που διαθέτει το operator module.
>>> time.sleep(operator.div(1.0 , 2)) # μισό δευτερόλεπτο delay
>>> time.sleep(operator.div(1.0 , 1000)) # 1ms delay
>>> time.sleep(operator.div(1.0 , 1000000)) # 1μs delay
>>> time.sleep(0.1) # Delay 0,1 sec. Επιτρέπονται δεκαδικοί αριθμοί.

# με επαναληπτικό βρόχο:
>>> while True:
        print "A recurrent message every one minute, ‘with’ loop ."
        time.sleep(60)
```

Για μέτρηση χρονικού διαστήματος μεταξύ δυο ενεργειών, δηλαδή πόσος χρόνος έχει παρέλθει, μπορούμε να γράψουμε έναν απλό αλγόριθμο όπως αυτός:

```
>>> import time
>>> start = time.time()
    [a block of code here]
>>> stop = time.time()
>>> elapsed_time = stop – start
```

Κάπως έτσι θα μπορούσαμε να εξάγουμε μια καλή εκτίμηση για μέτρηση επίδοσης πχ ενός κομματιού κώδικα που βρίσκεται μεταξύ των start-stop .

Copyright © 2014

ΜΕΡΟΣ II

Case study

- Εντοπισμός και διαχείριση σφαλμάτων
- FSM-Μηχανή πεπερασμένων καταστάσεων
- Εύρεση συντομότερης τυφλής διαδρομής

7.1 Εντοπισμός και διαχείριση σφαλμάτων

Όταν γράφουμε ένα πρόγραμμα ευελπιστούμε να λειτουργήσει χωρίς απρόοπτα. Εντούτοις, όσο μεγαλώνει σε έκταση η εφαρμογή(πρόγραμμα) που αναπτύσσουμε τόσο αυξάνει η πιθανότητα να γίνει πιο επιρρεπής σε λάθη που στον προγραμματισμό καλούνται bugs, η διαδικασία αφαίρεσης τους καλείται debugging. Όταν οι γραμμές του κώδικα είναι λίγες ο εντοπισμός της αιτίας του προβλήματος γίνεται απλούστερος, το αντίθετο είναι χρονοβόρο, κοπιαστικό και μπορεί να προκαλέσει μια πληθώρα από δυσάρεστα συναισθήματα. Τα λάθη μπορούν να προέρχονται κυρίως από τον ίδιο το χρήστη ή από ατυχία του ίδιου του συστήματος πχ υπερφόρτωση της μνήμης.

Τα λάθη θα μπορούσαμε να τα κατηγοριοποιήσουμε σε τρία είδη: syntax errors(συντακτικά), runtime errors(εκτέλεσης) και semantic errors(σημασιολογικά). Τα τελευταία, είναι προβλήματα που προκύπτουν στο πρόγραμμα όταν εκτελείται χωρίς να παράγονται σφάλματα που θα τερματίσουν την ροή εκτέλεσης του κώδικα, παρόλα αυτά παράγεται λογικό λάθος που ναι μεν αδυνατεί να διακρίνει ο διερμηνευτής αλλά μπορεί να διαπιστώσει ο χρήστης με μη αναμενόμενο αποτέλεσμα. Έτσι ο κώδικας συμπεριφέρεται με διαφορετικό τρόπο από τον επιθυμητό. Αυτού του είδους τα λάθη είναι τα πιο ύπουλα γιατί σε ένα πρόγραμμα μεγάλης έκτασης θα χρειαστεί ενδεχομένως περισσότερο χρόνο για να εντοπιστούν και να διορθωθούν από τον χρήστη και επειδή δεν συνηθίζεται να υπάρχει κάποιο traceback βοηθητικό μήνυμα που να μας βοηθά να εντοπιστούν ευκολότερα.

Τα συντακτικά λάθη από μεριάς τους είναι λάθη που μπορεί να τα ανιχνεύσει ο διερμηνευτής, να διακόψει την ροή του προγράμματος και να επιστρέψει ένα μήνυμα λάθους μέσω της traceback. Αυτά τα λάθη παράγονται όταν ο διερμηνευτής της γλώσσας μετατρέπει τον κώδικα από την γλώσσα υψηλού επιπέδου που έχει γραφτεί σε γλώσσα χαμηλού επιπέδου. Συνήθως δηλώνουν τα λάθη ότι πρόκειται για συντακτικά λάθη και συνεπώς σχετίζονται με την παραβίαση των τυπικών κανόνων σύνταξης όπως λόγου χάρη η παράλειψη τοποθέτησης εσοχών στα σωστά σημεία του κώδικα.

Από την άλλη μεριά, τα runtime λάθη παράγονται από τον διερμηνευτή εάν κάτι πάει στραβά καθόσον το πρόγραμμα τρέχει. Συνήθως, σε αρκετές περιπτώσεις αυτά τα μηνύματα λάθους περικλείουν αρκετές πληροφορίες σχετικά με το που βρίσκεται το σφάλμα και ποια ρουτίνα βρισκόταν υπό εκτέλεση εκείνη τη στιγμή.

Το πρώτο στάδιο στο debugging είναι να συμπεράνουμε με ποιο είδος σφάλματος έχουμε να κάνουμε. Αυτό καλείται αναγνώριση και κατανόηση του προβλήματος και είναι σημαντική λειτουργία γιατί αν δεν μπορούμε να αναγνωρίσουμε την ρίζα του προβλήματος θα χάσουμε πολύτιμο χρόνο ψάχνοντας και χρονοτριβώντας. Το επόμενο βήμα είναι να εντοπίσουμε σε ποιο σημείο του κώδικα βρίσκεται το σφάλμα και τέλος να επέμβουμε να το διορθώσουμε. Μετά, έχοντας διορθώσει το πρόβλημα ξαναδοκιμάζουμε τον κώδικα και βλέπουμε αν όντως λύθηκε το πρόβλημα και δεν δημιουργήθηκε στην θέση του κάποιο άλλο πρόβλημα. Είναι δυνατόν όταν επεμβαίνουμε να απαλείψουμε ένα σφάλμα να ανακύπτει ένα άλλο ή ακόμα χειρότερα περισσότερα του ενός ανάλογα την φύση του σφάλματος και την περίπτωση. Μια συνηθισμένη τακτική για την ανίχνευση σφαλμάτων είναι η εισαγωγή πολλών print statements(εντολές εκτύπωσης) γύρω από την περιοχή που παρουσιάζεται το πρόβλημα, προκειμένου να δούμε τι παράγει.

7.1.1 Συντακτικά λάθη

Τα συντακτικά λάθη οφείλονται σε αδυναμία του προγραμματιστή, είναι εύκολη η διόρθωση τους μόλις γίνει αντιληπτό περί τίνος πρόκειται. Αυτό που μπορεί να διαπιστωθεί όμως από τα μηνύματα λάθους είναι ότι δεν είναι απαραίτητα και πολύ βοηθητικά. Έτσι για παράδειγμα όταν προκύπτει ένα Syntax Error: invalid syntax ή ένα Syntax Error: invalid token, συμβάν που μπορεί να είναι συχνό φαινόμενο, βλέπουμε ότι δεν είναι κανένα από τα δυο αρκετά κατατοπιστικό και μπορεί να γεννά αρκετά ερωτηματικά στον χρήστη. Στον αντίποδα, τα συντακτικά μηνύματα λάθους μας πληροφορούν που μέσα στο πρόγραμμα το πρόβλημα προέκυψε. Αν και στη πραγματικότητα μας ενημερώνουν που ο διερμηνευτής εντόπισε το λάθος όχι απαραιτήτως που πραγματικά ακριβώς βρίσκεται το λάθος. Μπορεί λόγου χάρη το μήνυμα λάθους να δηλώνει μια γραμμή και το σφάλμα να βρίσκεται στην προηγούμενη γραμμή.

Φυσικά, αν χτίζεται ο πηγαίος κώδικας σταδιακά, προοδευτικά, δοκιμάζοντας τον κομμάτι κομμάτι πριν εισαχθεί νέο στέλεχος κώδικα, τότε λογικά είναι γνωστό που μπορεί να βρίσκεται το σφάλμα - στο πιο πρόσφατο κομμάτι κώδικα. Αντιθέτως, αν γίνεται χρήση έτοιμου κώδικα που τον έχει φτιάξει κάποιος άλλος τότε είναι σκόπιμο να ελέγχεται χαρακτήρα προς χαρακτήρα ότι ο κώδικας που γράφτηκε είναι ακριβώς ο σωστός και ότι δεν διαφοροποιείται από αβλεψία μας. Φυσικά, αν ο έτοιμος κώδικας περιέχει ούτως ή άλλως κάποιο σφάλμα είναι και αυτό ένα ενδεχόμενο που πρέπει να ληφθεί υπόψιν εφόσον σιγουρευτούμε ότι η προηγούμενη περίπτωση ικανοποιείται.

Μερικά τυπικά λάθη στα οποία μπορεί να υποπέσει κάποιος λόγω αβλεψίας είναι να χρησιμοποιήσει μια λέξη κλειδί ως μεταβλητή όπως αυτές που δίνονται στο κεφάλαιο δύο πίνακας δύο. Ένα άλλο είναι η παράλειψη τοποθέτησης της άνω και κάτω τελείας στην κατάλληλη θέση με τις δηλώσεις if,while,for,def κλπ. Για όσους προέρχονται από προγραμματισμό σε περιβάλλον όπως η C/C++ στην Python δεν τοποθετείται στο τέλος κάθε εντολής το ελληνικό ερωτηματικό. Παρομοίως ο μονός και διπλός αστερίσκος έχει διαφορετική εφαρμογή στην Python. Δεν υπάρχουν pointers στην Python. Στα αλφαριθμητικά πρέπει να τηρείται με συνέπεια το format με τους αποστρόφους. Αυτό σημαίνει ότι αν ξεκινήσουμε με μονό απόστροφο πρέπει να ολοκληρώσουμε το αλφαριθμητικό επίσης με μονό απόστροφο. Δεν παίζει ρόλο αν θα γίνει χρήση μονών ή διπλών αποστρόφων αλλά παίζει ρόλο να ζευγαρώνουν: για κάθε ένα αλφαριθμητικό θα περικλείεται εντός μονών ή διπλών αποστρόφων όχι μείξη. Όμως, επιτρέπεται ένα αλφαριθμητικό να περικλείεται με μονούς αποστρόφους και ένα άλλο με διπλούς. Ένα αλφαριθμητικό μη σωστά τοποθετημένο εντός αποστρόφων πρόκειται να προκαλέσει Syntax error: invalid token. Ειδικά στην περίπτωση με τους τριπλούς αποστρόφους αν τους 'ανοίξουμε' και δεν τους 'κλείσουμε' τότε σημαίνει ότι πρόκειται για σχόλια πολλαπλών γραμμών χωρίς να προκύψει κάποιο μήνυμα σφάλματος και να απορούμε γιατί δεν εκτελείται ο κώδικας.

Επίσης, όταν ανοίγουμε μια παρένθεση ή μια αγκύλη πρέπει επίσης να την κλείνουμε άρα πρέπει να μετράμε με το μάτι πόσες αγκύλες και πόσες παρενθέσεις έχουν ανοιχτεί για να κλείσουν και πρέπει να κλείσουν αλλιώς προκύπτει σφάλμα υπολογισμού γιατί η γλώσσα θεωρεί ότι συνεχίζεται και στην επόμενη γραμμή η τρέχουσα δήλωση. Επίσης, στις διαιρέσεις ποσοτήτων δεν πρέπει να ξεχνάμε να κάνουμε ρητή δήλωση (casting) στις μεταβλητές ή αν πρόκειται για σκέτους αριθμούς να τοποθετούμε και την τελεία με ένα μηδέν μετά τον αριθμό στον αριθμητή ή στον παρονομαστή αλλιώς δε θα προκύψει δεκαδικό αποτέλεσμα ποτέ. Επιπρόσθετα υπάρχει σαφής διάκριση μεταξύ του συμβόλου εκχώρησης τιμής, δηλαδή το μονό ίσον από το διπλό ίσον που πρόκειται για μια λογική πράξη και κάνει σύγκριση της αριστερής ποσότητας με αυτήν που βρίσκεται δεξιά του διπλού ίσον. Αν οι δυο ποσότητες ταυτίζονται το αποτέλεσμα είναι αληθές αλλιώς ψευδές. Σε καμία περίπτωση το διπλό ίσον δε κάνει καταχώρηση τιμής σε μεταβλητή. Σε κάθε περίπτωση πρέπει να τηρούμε την σύμβαση με το ίδιο μήκος πάντοτε για τις εσοχές, γιατί η γλώσσα λειτουργεί με εσοχές κι όχι κάποια άλλα σύμβολα για να αναγνωρίζει διαφορετικά κομμάτια κώδικα ως προς την σύνταξη τους.

Υπάρχει περίπτωση να υποπέσουμε σε σύγχυση όταν κάνουμε διόρθωση λαθών και μετά τρέχουμε ξανά και το πρόβλημα εμμένει. Αν υποθέσουμε ότι τροποποιήσαμε τον αρχικό πηγαίο κώδικα αλλά παραλείψαμε να αποθηκεύσουμε τις αλλαγές τότε οι νέες αλλαγές δε θα γίνουν αντιληπτές από τον διερμηνευτή και δε θα τις λάβει υπόψη του όταν θα ξανατρέξουμε τον κώδικα. Θα συνεχίζει να εκτελείται

ο παλιός κώδικάς πριν γίνουν οι αλλαγές. Το ίδιο μπορεί να συμβεί αν αλλάξουμε το όνομα ενός αρχείου αλλά εξακολουθούμε να τρέχουμε το παλιό αρχείο με το παλιό όνομα. Μπορούμε να εισάγουμε σκόπιμα ένα λάθος στην αρχή του κώδικα για να σιγουρευτούμε ότι ο διερμηνευτής βλέπει το ίδιο πρόγραμμα που βλέπουμε κι εμείς και συνεπώς να επιστρέψει μήνυμα λάθους ανάλογα την περίπτωση με τη σκόπιμο σφάλμα επιδιώκουμε να προκαλέσουμε.

Στην περίπτωση με την εντολή import θα πρέπει να προσέχουμε να μην ονοματίζουμε τα δικά μας modules με ονόματα modules που υπάρχουν στην Python Standard Library γιατί αυτό μπορεί να προκαλέσει σύγκρουση και σφάλματα. Επίσης, όταν έχουμε ένα κομμάτι πηγαίου κώδικα και μετά αποφασίσουμε να καλέσουμε με την εντολή import μια συνάρτηση ή κάτι άλλο θα πρέπει να επανεκκινήσουμε τον διερμηνευτή ή να κάνουμε χρήση της συνάρτησης reload όπως έχει δειχθεί. Δεν έχει νόημα να προσπαθούμε να κάνουμε πολλαπλές φορές import ένα module μία φορά αρκεί και δεν πρόκειται να αλλάξει τίποτα αν προσπαθούμε να το κάνουμε παραπάνω από μία φορά. Μια εφαρμογή δίνεται παρακάτω.

```
1.    def fibonacci((n):
2.     ;ο αναδρομικός τύπος χωρίς χρήση λίστας
3.        if n 0:
4.            return "does "not" exist"
5.        elif n<2:          # isodynamo toy: elif n=0 or n==1:
6.            returnn
7.        else
8.            return fibonacci(n-1)+fibonacci(n-2)
9.     result=fibonacci(6)
10.   print result
```

Αυτός ο κώδικας περιέχει μερικά συντακτικά λάθη που θα είναι ανιχνεύσιμα από τον διερμηνευτή. Στην πρώτη γραμμή άνοιξαν δυο παρενθέσεις αλλά έκλεισε μία μόνο. Στην δεύτερη γραμμή ενώ έπρεπε να είναι σχόλιο δεν έγινε χρήση της δίεσης # που ισοδυναμεί με σχόλιο μιας γραμμής και επίσης τα ελληνικά γράμματα δεν αναγνωρίζονται πρέπει να γίνει χρήση λατινικών χαρακτήρων. Στην τρίτη γραμμή λείπει ο σωστός τελεστής σύγκρισης, ενώ στην γραμμή τέσσερα γίνεται λάθος χρήση αποστρόφων. Ακολούθως, στην γραμμή έξι υπάρχει λάθος γιατί ενώσαμε μαζί την δεσμευμένη λέξη με την μεταβλητή. Ξαναγράφοντας την παραπάνω αναδρομική συνάρτηση χωρίς λάθη εκτελείται ο κώδικας χωρίς πρόβλημα και επιστρέφει την τιμή οκτώ γιατί ζητάμε να υπολογίσει τον έκτο όρο της ακολουθίας στην προτελευταία και στην τελευταία γραμμή του κώδικα ζητάμε να τον εμφανίσει.

```
def fibonacci(n):
# anadromikos typos xoris lista
  if n<0:
    return "does not exist"
  elif n<2:          # isodynamo toy: elif n==0 or n==1:
    return n
  else:
    return fibonacci(n-1)+fibonacci(n-2)
result=fibonacci(6)
print result
```

7.1.2 Σφάλματα runtime

Τα σφάλματα εκτέλεσης ροής δεν γίνονται αντιληπτά πριν εκτελεστεί το πρόγραμμα. Αν πάει κάτι στραβά κατά την εκτέλεση του η Python θα τυπώσει ένα μήνυμα που θα συμπεριλαμβάνει το όνομα του exception, την γραμμή του προγράμματος που έλαβε χώρα το πρόβλημα και ένα traceback μήνυμα. Η τελευταία αναγνωρίζει την συνάρτηση που την δεδομένη στιγμή εκτελείται, κατόπιν την συνάρτηση που

την επικαλέστηκε και συνεχίζει σε αυτό το μοτίβο αναδρομικά μέχρι το σημείο που φτάσαμε στο πρόβλημα. Με άλλα λόγια , η traceback ιχνηλατεί την ακολουθία επίκλησης όλων των συναρτήσεων που μας έφεραν σε αδιέξοδο και άρα σε σφάλμα. Επιπλέον, δηλώνεται και ο αριθμός γραμμής για κάθε μια από τις επικλήσεις των συναρτήσεων.

Σε μια τέτοια περίπτωση, ένα καλό σημείο εκκίνησης για διερεύνηση του προβλήματος είναι να μελετήσουμε το σημείο που στο πρόγραμμα προκλήθηκε το σφάλμα, με σκοπό να λάβουμε μια πρώτη εκτίμηση τι συνέβη και προκάλεσε το πρόβλημα. Μερικά από τα ποιο κοινά λάθη που συναντώνται στην πράξη είναι: NameError, TypeError, KeyError, AttributeError και IndexError. Η γλώσσα Python διαθέτει ένα εκτενές module τον debugger(pdb) για αντιμετώπιση exceptions και ότι άλλο προκύψει, διότι επιτρέπει να επιθεωρήσουμε την κατάσταση του προγράμματος αμέσως πριν την εμφάνιση του σφάλματος.

Νωρίτερα έχει γίνει αναφορά στην εντολή print ότι μπορεί να βοηθήσει στην εκσφαλμάτωση του πηγαίου κώδικα, αρκεί βέβαια να μη το παρακάνουμε και χρησιμοποιούμε υπερβολικά πολλές τέτοιες εντολές γιατί μπορεί να δυσχεράνει την εργασία. Αν γίνει κατάχρηση της print τότε μπορεί να πλημμυρίσει η τυπική έξοδος με πολλές εκτυπώσεις, κάθε μια print και ένα αποτέλεσμα. Σε τέτοιες περιπτώσεις η μία λύση είναι να απλοποιηθεί ο πηγαίος κώδικας με κάθε τρόπο πχ σπάζοντας τον σε μικρότερα και ευκολότερα διαχειρίσημα τεμάχια , ενώ η άλλη λύση είναι να γίνει χρήση λιγότερων και πιο στοχευμένων print δηλώσεων(απλοποίηση της εξόδου). Η απλοποίηση μπορεί να γίνει μικραίνοντας την έκταση των αποτελεσμάτων της εξόδου και περιορισμό εμφάνισης ενός κλάσματος των αποτελεσμάτων. Έτσι, για παράδειγμα αν ζητάμε να εμφανιστεί ένα λεξικό με εκατό στοιχεία μπορούμε να ζητήσουμε να εμφανίσει προσωρινά πολύ λιγότερα, αν πρόκειται για είσοδο δεδομένων με εντολή όπως η input να περιοριστούμε στο να εισάγουμε μια όσο γίνεται πιο απλή, λιγότερο πολυπληθή είσοδο ορισμάτων που προκαλεί ή προκαλούν πρόβλημα και να δούμε με print δήλωση αντίστοιχα για κάθε μία είσοδο τι αποτέλεσμα παράγεται. Κάτι άλλο που μπορεί να γίνει είναι να αφαιρεθεί κώδικας που είναι περιττός και δεν προσφέρει ουσιαστικά στο κτίσιμο μιας εφαρμογής, νεκρός κώδικας και μάλλον περιπλέκει την επίλυση. Κατόπιν, να προσπαθήσουμε να μετατρέψουμε τον κώδικα σε μια τέτοια μορφή που να είναι πιο εύκολα αναγνώσιμος και κατανοητός από οποιονδήποτε όχι μόνο τον συγγραφέα του. Ακόμη, αν ο κώδικας είναι περίπλοκος μπορούμε να μικρύνουμε την έκταση των συναρτήσεων και των φωλιασμένων περιπτώσεων και να χρησιμοποιούμε λιγότερες διακλαδώσεις. Μετά την απλοποίηση του να τεστάρουμε το κάθε μέρος ξεχωριστά. Αν διαπιστώσουμε ότι υπό μια προϋπόθεση μια ρουτίνα λειτουργεί και υπό μια άλλη προϋπόθεση δεν λειτουργεί σωστά τότε αυτό αποτελεί μια καλή ένδειξη για το τι συμβαίνει. Ασφαλώς, αν υπάρχει εμμένουσα αμφιβολία, μπορούμε να ξαναγράψουμε κάποια κομμάτια κώδικα από την αρχή ελπίζοντας ότι θα εντοπίσουμε έτσι την αιτία του προβλήματος όσο ασήμαντη κι αν είναι. Μπορούμε όταν ξαναγράφουμε κάποιο κομμάτι του πηγαίου κώδικα να διαπιστώσουμε ότι σε κάποιο σημείο υπάρχει διαφορετική συμπεριφορά από αυτή που εμείς το προορίσαμε, αυτό είναι μια ένδειξη ότι κάτι δε πάει καλά και εκεί θα βρίσκεται ενδεχομένως το σφάλμα.

Συνεχίζοντας, μια άλλη περίπτωση είναι το πρόγραμμα να σταματήσει να αποκρίνεται, δηλαδή να «κρεμάσει». Αυτό συνεπάγεται ότι έχει περιπέσει σε ατέρμονα βρόχο ή ατέρμονη αναδρομή. Για την περίπτωση του ατέρμονα βρόχου αν νομίζουμε ότι καταλαβαίνουμε ποιος βρόχος προκαλεί το πρόβλημα, θα μπορούσαμε να προσθέσουμε μια εντολή print στο τέλος του βρόχου στη θέση που θα τυπώνονται οι τιμές των μεταβλητών μιας συνθήκης και την τιμή της συνθήκης. Δηλαδή,

```
>>> a,b=8,4
>>> from math import pow
>>> while a<32 and b>0:
        a=math.pow(a,b)
        b=b-1
        print 'a= ', a
        print 'b= ', b
        print ' (a<1024 and b>0)  True or False : \n', (a<1024 and b>0)
```

Μόλις τρέξουμε το πρόγραμμα θα δούμε δυο γραμμές που δίνουν τις τρέχουσες τιμές των μεταβλητών της συνθήκης στην εκάστοτε στιγμή, οι οποίες ανανεώνονται με κάθε νέο πέρασμα και θα δούμε μια τρίτη

γραμμή που θα δίνει την λογική τιμή της ίδιας της συνθήκης, όταν γίνει ψευδής σταματά να εκτελείται ο βρόχος. Στην προκείμενη περίπτωση είναι σωστά κουρδισμένος ο βρόχος και δεν θα υπάρχει πρόβλημα ατέρμονα βρόχου. Το εκτυπώσιμο αποτέλεσμα είναι:

```
a=  16.0
b=  3
 (a<1024 and b>0)  True or False :
True
a=  4096.0
b=  2
 (a<1024 and b>0)  True or False :
False
```

Ένα απλό παράδειγμα ατέρμονα βρόχου είναι το εξής:

```
>>> while True:
        {εντολές}
```

Αν δεν τοποθετήσουμε εντός της while εντολής έναν περιορισμό που να τερματίσει την while κάποια στιγμή, τότε θα μείνει το πρόγραμμα παγιδευμένο μέσα στη συγκεκριμένη while και θα συνεχίζει να εκτελείται για πάντα ή μέχρι να πατήσουμε Ctrl+C. Ιδού ένα πραγματικό παράδειγμα που παραλείψαμε να ορίσουμε έναν περιορισμό για το πότε να τερματίσει και έχει πέσει σε ατέρμονα βρόχο:

```
>>> import time
>>> x=0
>>> while True:
        x+=1
        print x,
        time.sleep(1):
```

Θα συνεχίζει εις αεί να τυπώνει όλους τους ακέραιους αριθμούς οριζοντίως από αριστερά προς τα δεξιά, σε αύξουσα σειρά με καθυστέρηση ενός δευτερολέπτου για τον κάθε ένα.

Για την περίπτωση της ατέρμονης αναδρομής αυτό που γίνεται είναι ότι το πρόγραμμα θα συνεχίσει να κάνει υπολογισμούς, θέλουμε δε θέλουμε, μέχρι να φτάσει σε ένα σημείο κορεσμού που εκεί παράγει το σφάλμα υπέρβασης μέγιστου βάθους αναδρομής(max recursion depth exceeded). Σε μια τέτοια περίπτωση θα πρέπει να είμαστε σε θέση να βρούμε την θεμελιώδη περίπτωση που εκεί θα πρέπει κανονικά να σταματά και να αρχίσει κατόπιν να τυπώνει. Χωρίς την θεμελιώδη περίπτωση θα παγιδευτεί το πρόγραμμα σε ατέρμονα αναδρομή. Αν πάλι υπάρχει αυτή αλλά το πρόγραμμα δεν τα καταφέρνει να την φτάσει τότε μπορούμε να κάνουμε χρήση μιας εντολής print στην αρχή της συνάρτησης που τυπώνει τις παραμέτρους. Με αυτό τον τρόπο όταν ξανατρέξουμε το πρόγραμμα θα παραχθούν μερικές γραμμές εξόδου κάθε φορά που καλείται μια συνάρτηση. Αν οι παράμετροι δεν κατευθύνονται προς την θεμελιώδη περίπτωση(fundamental condition),τότε μπορούμε να σχηματίσουμε μια καλύτερη εικόνα κοιτάζοντας τις print δηλώσεις τι πάει στραβά.

```
import time
def antistrofi_metrisi(n):
    print n
    time.sleep(0.5)
    antistrofi_metrisi (n-1)
antistrofi_metrisi(4)
```

Στην παραπάνω περίπτωση δεν έχουμε εισάγει μια θεμελιώδη περίπτωση οπότε θα προκύψει ατέρμονη αναδρομή. Την μέθοδο sleep την χρησιμοποιούμε για να μπορούμε να βλέπουμε πως εξελίσσεται η ροή του προγράμματος και να κατανοήσουμε τι συμβαίνει σε πραγματικό χρόνο. Ο παρακάτω διορθωμένος κώδικας εκτελεί σωστά την αντίστροφη μέτρηση με την συμπερίληψη της θεμελιώδης περίπτωσης:

```
def countdown(n):
        if n <= 0:        #fundamental condition
                print 'telos anadromis'
        else:
        print n
countdown(n-1)
```

7.1.3 Semantic σφάλματα

Όπως έχει επισημανθεί αυτά τα σφάλματα είναι μάλλον τα δυσκολότερα να αντιμετωπιστούν γιατί ο διερμηνευτής δεν είναι σε θέση να παρέχει απολύτως καμία πληροφόρηση σχετικά με τι πάει στραβά. Δεν είναι συντακτικά λάθη αλλά λογικά λάθη και μόνο αυτός που γράφει τον πηγαίο κώδικα ξέρει τι έργο προορίζεται να επιτελέσει ο κώδικας, ο υπολογιστής δεν μπορεί να σκεφτεί μόνο να εκτελέσει πράξεις ταχύτατα. Άρα, το πώς πρέπει να συμπεριφερθεί ο κώδικας είναι ευθύνη του προγραμματιστή κι όχι της μηχανής. Παράδειγμα με λογικά λάθη:

```
>>> y = x / 2 * math.e                    |          >>> y = x / (2 * math.e)
```

Αν θέλουμε να διαιρέσουμε την ποσότητα x με την ποσότητα 2 * math.e τότε θα πρέπει να μπει όλος ο παρονομαστής εντός παρένθεσης διαφορετικά ο υπολογιστής δεν είναι σε θέση να καταλάβει την διαφορά. Στην έκφραση αριστερά το y είναι το αποτέλεσμα της πράξης αφού πρώτα γίνει η διαίρεση της μεταβλητής x με το 2 και κατόπιν ότι προκύψει από την διαίρεση πολλαπλασιάζεται με το e. Στην έκφραση δεξιά επειδή συμπεριλάβαμε εντός παρένθεσης όλο τον παρονομαστή ,πρώτα θα υπολογιστεί το 2 με το e και έτσι το y θα είναι το αποτέλεσμα της διαίρεσης του x με το αποτέλεσμα του πολλαπλασιασμού των ποσοτήτων 2 και e.

Υπάρχουν περιπτώσεις που μπορεί να έχουμε σχηματίσει μια αρκετά μακροσκελή έκφραση του τύπου: container.listbox(entryfood).boxtype(param).boxcategory(value).setselection(n).Τόσο μακροσκελείς και ακόμα μεγαλύτερες προτάσεις από αυτήν είναι εφικτό να δημιουργηθούν στην Python αλλά δυσχεραίνουν την ανάγνωση τους. Είναι δύσκολο να εκσφαλματωθούν σε περίπτωση που υπάρχει κάποιο λογικό λάθος διότι είναι δυσανάγνωστη έκφραση. Μια λύση είναι να την σπάσουμε σε μικρότερα κομμάτια ποιο εύκολα αναγνώσιμα. Πια πιθανή απλοποίηση είναι η εξής:

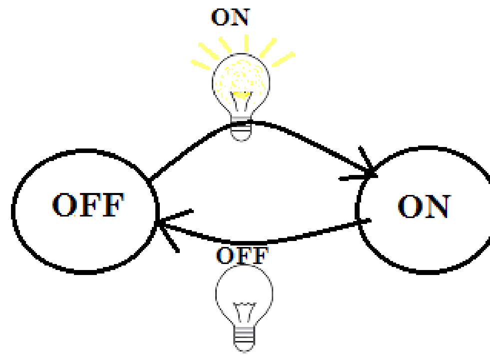
```
mybox = container.listbox(entryfood)
boxstatus= boxtype(param).boxcategory(value)
mybox.boxstatus.setselection(n)
```

7.2 Μελέτη FSM

Μία Μηχανή Πεπερασμένων Καταστάσεων(**F**inite **S**tate **M**achine ή πιο σύντομα FSM), είναι ένα μαθηματικό μοντέλο λογικής, μια διαδικασία με σαφώς καθορισμένα βήματα που αποσκοπεί στην επίλυση ενός προβλήματος με συγκεκριμένο πεπερασμένο τρόπο. Χρησιμοποιείται πολύ τόσο σε προγράμματα για υπολογιστές όσο και σε ακολουθιακά ηλεκτρονικά λογικά κυκλώματα. Η ραχοκοκαλιά μιας FSM αποτελείται από τις *καταστάσεις* και τις *μεταβάσεις.* Το σύνολο των μεταβάσεων δημιουργούν ένα δίκτυο. Μια μετάβαση από την εκάστοτε τωρινή κατάσταση(current state),που είναι και η μοναδική ανά πάσα χρονική στιγμή, σε μια άλλη, γίνεται υπό τον περιορισμό ότι τηρούνται κάποιες προϋποθέσεις. Όταν τηρούνται οι συνθήκες(μπορεί να είναι μία μόνο) για μετάβαση σε μια άλλη κατάσταση λέμε ότι *σκανδαλίζεται*(trigger) και όταν μεταβεί τελικά σε άλλη κατάσταση λέμε ότι *συντελέστηκε*(fired).

Οι καταστάσεις συνδέονται μεταξύ τους με μεταβάσεις. Έτσι , όταν η FSM βρίσκεται στην κατάσταση Α υπό προϋποθέσεις μπορεί να κάνει μετάβαση στην κατάσταση Β, στην κατάσταση Γ κλπ. Είναι πιθανόν όταν η FSM βρίσκεται σε μια κατάσταση να αδυνατεί να μεταβεί σε μια άλλη κατάσταση με ένα βήμα(δηλαδή με μία μόνο μετάβαση) και να χρειάζεται περισσότερα *hops*(άλματα, βήματα) για να φτάσει στην *κατάσταση στόχο*. Επίσης, υπάρχει περίπτωση μια κατάσταση να είναι υπό προϋποθέσεις απαγορευμένη, να μην μπορεί να μεταβεί σε αυτήν, όπως επίσης μπορεί να τύχει η FSM να παγιδευτεί σε μια τάδε κατάσταση και να μην μπορεί να ξεφύγει από αυτήν, εκτός κι αν έχουμε λάβει τα μέτρα μας.

Οι FSM μπορούν να μας δίνουν την ψευδαίσθηση ότι οι μηχανές είναι σε θέση να σκέφτονται όχι μόνο να πράττουν. Στην πράξη έχουν αυτό που λέμε τεχνητή νοημοσύνη (AI) και το πόσο «έξυπνες» είναι εξαρτάται από τον προγραμματιστή της μηχανής. Ένα κλασικό παράδειγμα FSM είναι η μηχανή Turing. Η ιδέα μιας FSM είναι να αναλύσει(να 'σπάσει') ένα πρόβλημα σε μικρά κομμάτια εύκολα διαχειρίσημα και επιλύσιμα, να προδικάσει την συμπεριφορά του συστήματος με όσο γίνεται πιο ντετερμινιστικό τρόπο. Ένα απλό παράδειγμα είναι στο Σχήμα 20 όπου φανταζόμαστε έναν βραχίονα να αναβοσβήνει το φως.

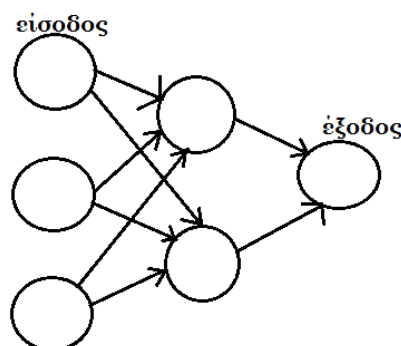


Σχήμα 20. Μια πολύ απλή FSM 2 καταστάσεων: On και Off. Τα βελάκια αναπαριστούν τον διακόπτη που αναβοσβήνει κι άρα τις μεταβάσεις από τη μία σταθερή κατάσταση στην άλλη.

Οι FSM χρησιμοποιήθηκαν και θα συνεχίσουν να χρησιμοποιούνται. Η δημοτικότητα τους έγκειται στο γεγονός ότι συντρέχουν κάποιοι λόγοι που συνδράμουν στην προτίμηση των προγραμματιστών προς αυτές. Μερικοί λόγοι είναι:

- Τείνει να είναι αρκετά εύκολο να γραφτεί κώδικας για FSM από ένα ευρύ κοινό. Υπάρχουν διάφορες προσεγγίσεις για να γραφτεί μια FSM και ο καθένας μπορεί να επιλέξει την μέθοδο που τον εξυπηρετεί καλύτερα.
- Παρουσιάζουν μια αρκετή ευκολία στην εκσφαλμάτωση. Αυτό συμβαίνει γιατί ο κώδικας δομείται από μικρά απλούστερα υποπρογράμματα. Μην ξεχνάμε ότι μια FSM αποτελείται από καταστάσεις και μπορεί να ελεγχθεί αν μια κατάσταση λειτουργεί σωστά και έτσι μπορούμε να ελέγξουμε την κάθε μια ξεχωριστά.
- Επιφέρει ελεγχόμενη κι σίγουρα όχι υπερβολική υπολογιστική επιβάρυνση. Αυτό συμβαίνει γιατί βασίζεται σε μια ακολουθία από εντολές του τύπου if-elif-else οι βασικοί κανόνες είναι ίδιοι και απαράλλακτοι. Δεν χρειάζεται να λύσει δύσκολους αλγόριθμους η CPU, οπότε δεν χρειάζεται αυξημένη υπολογιστική ισχύ.

-Τέλος, οι FSM είναι κοντά στον ανθρώπινο τρόπο σκέψης και είναι ευέλικτες. Μπορούμε να κατακερματίσουμε ένα πρόβλημα σε πολλές καταστάσεις, η κάθε μια κατάσταση θα είναι ένα μέρος της λύσης του συνολικού προβλήματος. Επίσης, μπορεί να γίνει ευκολότερα κατανοητή μια FSM από μη μυημένους στον προγραμματισμό. Η ευελιξία αρμόζει στο γεγονός ότι διαθέτοντας ένα βασικό κορμό μπορούμε να βασιστούμε σε αυτόν και με μικρές ή μεγάλες τροποποιήσεις να διαφοροποιήσουμε τον τρόπο συμπεριφοράς λίγο ή πολύ μιας FSM κι άρα του AI. Τέλος, μπορούμε να επεκταθούμε σε ασαφή λογική και νευρωνικά δίκτυα έχοντας ως ραχοκοκαλιά μια FSM. Με πολύ απλά λόγια αν η FSM προσφέρει το άσπρο και το μαύρο(για μια FSM δυο καταστάσεων που είναι η πιο εύληπτη περίπτωση) τότε η ασαφής λογική και τα νευρωνικά δίκτυα με τις περίπλοκες τεχνικές τους μπορούν να μας προσφέρουν και διαβαθμίσεις μεταξύ του άσπρου και του μαύρου. Για παράδειγμα, ας φανταστούμε μια FSM για ένα κύκλωμα με ένα διακόπτη ON-OFF. Στο ON ανάβει το φως και στο OFF σβήνει. Αν όμως θέλουμε μπορούμε να περιπλέξουμε την λειτουργία του διακόπτη με το να αντικαταστήσουμε τον απλό ON-OFF διακόπτη με ένα ρυθμιζόμενο. Τώρα πια κάνουμε λόγο όχι μόνο για On και OFF αλλά και για το πόσο δυνατό ή άτονο θέλουμε το φως να είναι, δηλαδή έχουμε διαβαθμίσεις μεταξύ των δυο στάνταρ καταστάσεων ON-OFF.



Σχήμα 21. Ένα τεχνητό στέλεχος από νευρωνικό δίκτυο. Κάθε κύκλος αναπαριστά μια κατάσταση, έναν νευρώνα, ενώ τα βελάκια την διασύνδεση κάθε νευρώνα, δηλαδή την διασύνδεση τους.

7.2.1 Ένας αλγόριθμος FSM και η ανάλυση του

Παρακάτω παρατίθεται ένας αλγόριθμος στην βασική του μορφή καθώς και η μεθοδολογία ανάλυσης του όταν συναντάμε τέτοιου είδους πηγαίου κώδικα. Μπορεί να χρησιμοποιηθεί ως βασική δομική μονάδα για ποιο πολύπλοκα προβλήματα. Επίσης, δεν είναι ο μοναδικός τρόπος κατασκευής μιας FSM καθώς υπάρχουν διάφορα μοντέλα ανάπτυξης FSM στην βιβλιογραφία και αναφέρεται στις αναφορές στο τέλος της πτυχιακής αυτής. Η μελέτη, ο τρόπος μελέτης, κατανόησης και ανάλυσης του είναι υπόθεση του εκάστοτε αναγνώστη και προς αυτή την κατεύθυνση κινείται και η ανάλυση που θα παραδοθεί ακολούθως, που είναι και ο σκοπός συγγραφής. Έχει προηγηθεί σκοπίμως μια ανάλυση για το debugging(εκσφαλμάτωση) και τον τρόπο αντιμετώπισης των σφαλμάτων. Αυτό έγινε γιατί όταν γράφουμε κώδικα πολλών γραμμών μοιραία θα υπάρχουν σφάλματα και οι προηγούμενες παράγραφοι σχετικά με την διαχείριση των λαθών αποσκοπούσαν στο να προσφέρουν μια γνώση εκ των προτέρων, να προσφέρουν ένα χέρι βοηθείας όταν τύχει να βρεθούμε σε μια δύσκολη θέση, ένα τέλμα. Τέλος, να τονιστεί ότι ο παρακάτω κώδικας γράφτηκε στην Python version 2.7.6 και συνεπώς δεν εγγυάται ότι θα λειτουργήσει σε νεότερες εκδόσεις. Το ίδιο ισχύει και για την ανάλυση του, η οποία είναι έγκυρη και ισχύει για την άνωθεν έκδοση της Python.

FINITE STATE MACHINE#

```
1.	from random import randint
2.	from time import clock
3.
4.	#-----------------------------------------------------------------------------------------------------------------------
5.	State=type("State",(object,),{})
6.
7.	class LedOn(State):
8.		def Run(self):
9.			print "Led is emitting light!"
10.
11.	class LedOff(State):
12		def Run(self):
13.			print "Led is Off."
14.
15.	#-----------------------------------------------------------------------------------------------------------------------
16.
17.	class Transfer(object):
18.		def __init__(self, gotoState):
19.			self.gotoState=gotoState
20.
21		def Run(self):
22.			print "Changing state..."
23.
24.	#-----------------------------------------------------------------------------------------------------------------------
25.
26.	class myFSM(object):
27.		def __init__(self,MCU):
28.			self.MCU=MCU
29.			self.states={}
30.			self.transfers={}
31.			self.currentState=None
32.			self.trans=None
33.
34.		def SetState(self, Namestate):
35.			self.currentState=self.states[Namestate]
36.
```

```
37.         def Transfer(self,transName):
38.             self.trans=self.transfers[transName]
39.
40.         def Run(self):
41.             if(self.trans):
42.                 self.trans.Run()
43.                 self.SetState(self.trans.gotoState)
44.                 self.trans=None
45.             self.currentState.Run()
46.
47.     #-----------------------------------------------------------------------------------------------------------------
48.
49.     class MCU(object):
50.         def __init__(self):
51.             self.FSM=myFSM(self)
52.             self.LedOn=True
53.
54.     #-----------------------------------------------------------------------------------------------------------------

55.     if __name__=="__main__":
56.         portlight=MCU()
57.
58.         portlight.FSM.states["HIGH"]=LedOn()
59.         portlight.FSM.states["LOW"]=LedOff()
60.         portlight.FSM.transfers["toOn"]=Transfer("HIGH")
61.         portlight.FSM.transfers["toOff"]=Transfer("LOW")
62.
63.         portlight.FSM.SetState("HIGH")
64.
65.         for i in range(12):
66.             launchTime=clock()
67.             timedelay=1
68.             while(launchTime+timedelay>clock()):
69.                 pass
70.             if(randint(0,2)):
71.                 if(portlight.LedOn):
72.                     portlight.FSM.Transfer("toOff")
73.                     portlight.LedOn=False
74.             else:
75.                     portlight.FSM.Transfer("toOn")
76.                     portlight.LedOn=True
77.             portlight.FSM.Run()
```

Οδηγός Ανάλυσης:

Κοιτάζοντας τον κώδικα από 'έξω προς τα μέσα', από πάνω προς τα κάτω, παρατηρούμε ότι απαρτίζεται από 6 δομικές μονάδες(building blocks) που χωρίζονται με την δίεση # και διακεκομμένες γραμμές. Είναι εντελώς προαιρετικά(συνδυασμός δίεσης και παύλες)και σε συνδυασμό με μερικές κενές γραμμές απλώς τοποθετούνται για να κάνουν τον κώδικα πιο ευανάγνωστο και πιο εύληπτο. Επίσης, όταν παραθέτουμε ένα κώδικα πολλών γραμμών η αρίθμηση της κάθε γραμμής είναι πολύ χρήσιμη, όπως λόγου χάρη για προσθήκη αναλυτικών υποσημειώσεων σε ξεχωριστό κείμενο κρατώντας τον κώδικα 'καθαρό' από μακροσκελή σχόλια. Βέβαια σύντομα σχόλια δεν είναι κακό να υπάρχουν ανάμεσα στις γραμμές του κώδικα με σκοπό να αποκαλύπτουν τι θέλει να πετύχει ο αλγόριθμος, ειδικά σε δυσκολονόητα σημεία.

Βέβαια το τι είναι δυσκολονόητο είναι στη διακριτική ευχέρεια του προγραμματιστή που γράφει τον πηγαίο κώδικα να το κρίνει, καθώς και το πόσο αποκαλυπτικός θέλει να είναι.

Επόμενο βήμα, βλέπουμε ότι ο σκελετός του πηγαίου κώδικα βασίζεται σε κλάσεις(κι άρα έχουμε να κάνουμε με OOP), ορισμό και χρήση αρκετών συναρτήσεων, εισαγωγή με την εντολή import δυο μεθόδων, έναν constructor,την συνάρτηση main με τα περιεχόμενα της και την καρδιά της FSM που είναι η δομή if-else. Βέβαια, θα μπορούσαμε να φτιάξουμε την FSM χωρίς κλάσεις, αλλά θα γινόταν πολύ πιο μεγάλη κατάχρηση της δομής if-elif-else, με αποτέλεσμα να οδηγηθούμε σε σπαγγέτι κώδικα που θα είναι πιο δύσκολο να διαβαστεί μετά την πάροδο ενός μεγάλου χρονικού διαστήματος, ακόμα και από τον δημιουργό του. Κατόπιν θα επεξηγηθούν τα εργαλεία που αναφέρθηκαν ,πως λειτουργούν στην γλώσσα Python και δομούν την FSM στο σύνολο της ,προτού ξεκινήσει η γραμμή προς γραμμή ανάλυση.

Μια κλάση(class) είναι βασικά μια δομή της Python που ορίζει μια κατηγορία αντικειμένων και επιπλέον περιγράφει τα κοινά τους χαρακτηριστικά. Στην Python μια κλάση για να χρησιμοποιηθεί πρέπει πρώτα να έχει οριστεί κι άρα να υπάρχει. Το συντακτικό μιας κλάσης παρουσιάζει την ακόλουθη βασική δομή:

```
class όνομα_κλάσης(όρισμα):
        {κώδικας}
```

Μέσα στην κλάση μπορούμε να δημιουργήσουμε συναρτήσεις ή όπως λέγονται methods και ότι άλλο θέλουμε να τοποθετήσουμε ως παράμετρο. Μια κλάση αποτελείται από ορίσματα και χαρακτηριστικά γνωρίσματα(attributes). Σε μια κλάση μπορούμε να έχουμε μεταβλητές κλάσης(class variables) ή/και μεταβλητές στιγμιότυπα(instance variables). Η θέση των μεταβλητών κλάσης είναι στο πεδίο 'όρισμα' ενώ η θέση των μεταβλητών στιγμιότυπα εκεί που λέει κώδικας. Οι μεταβλητές κλάσης έχουν εμβέλεια σε όλο το σώμα της κλάσης, ενώ οι μεταβλητές στιγμιότυπα μόνο μέσα στη συνάρτηση που ορίζονται εντός μιας κλάσης, είναι δηλαδή πιο τοπικές. Οι τελευταίες δημιουργούνται με το συντακτικό: 'self.name=value' από όπου το name και το value τα εισάγει ο χρήστης ή κάποιο πρόγραμμα. Τόσο οι μεταβλητές κλάσης όσο και οι μεταβλητές στιγμιότυπα μπορούν να προσπελαστούν με το εξής συντακτικό: 'self.name'. Μια μεταβλητή στιγμιότυπο κρύβει μέσα της μια μεταβλητή κλάση με το ίδιο όνομα εφόσον την προσπελαύνουμε με αυτό τον τρόπο. Ένα παράδειγμα κλάσης είναι:

```
>>> class Hero:
        def heroname(self,name):
            self.name=name
        def herogender(self,gender):
            self.gender=gender
        def heroage(self,age):
            self.age=age
        def display_all(self):
            print 'Hero name: ', self.name, '\n', 'sex: %s \nage: %d' %(self.gender, self.age)
```

Για να βεβαιωθούμε ότι δημιουργήθηκε η κλάση(ότι ορίστηκε) πληκτρολογούμε το όνομα της κλάσης στον διερμηνευτή και αυτός με την σειρά του τυπώνει ένα μήνυμα που αναφέρει ότι υπάρχει αυτή η κλάση αποθηκευμένη ως αντικείμενο σε κάποια θέση μέσα στην μνήμη(0x0135A260 στην προκείμενη περίπτωση):

```
>>> Hero
<class __main__.Hero at 0x0135A260>
```

Συνεπώς, φτιάξαμε μια κλάση που μπορούμε να την θεωρήσουμε ως ένα *blueprint*(λεπτομερές προσχέδιο). Για να μπορέσουμε να προσπελάσουμε τα περιεχόμενα της κλάσης αυτής πρέπει να δημιουργήσουμε αντικείμενα, επιτρέπεται να δημιουργήσουμε όσα αντικείμενα θέλουμε από το ίδιο blueprint. Ας φτιάξουμε δυο τέτοια:

```
>>> OzWizzard=Hero()
>>> OzFighter=Hero()
```

Έτσι με αυτό τον τρόπο δημιουργήθηκαν δυο αντικείμενα. Τα OzWizzard, OzFighter είναι τα ονόματα των αντικειμένων, μπορούμε να γράψουμε ότι ονόματα επιθυμούμε. Μετά το ίσον δηλώσαμε με ποια κλάση

συσχετίζονται. Είναι σημαντικό μετά το Hero ανάμεσα στην κάθε παρένθεση να μην υπάρχει τίποτα, να είναι κενή. Κατόπιν συνεχίζουμε με τα χαρακτηριστικά του κάθε αντικειμένου:

```
>>> OzWizzard.heroname('Eliud')
>>> OzWizzard.herogender('masculine')
>>> OzWizzard.heroage(31)
```

Τώρα μπορούμε να ζητήσουμε να τυπώσουμε τα αποτελέσματα με τον τρόπο(format) που έχει προσδιοριστεί στην display_all μέθοδο:

```
>>> OzWizzard.display_all() # εκτύπωση θα γίνει γι'αυτό δεν παίρνει ορίσματα εντός παρενθέσεων.
Hero name: Eliud
sex: masculine
age: 31
```

Αν θέλαμε μπορούσαμε να ξεκινήσουμε με μια άδεια κλάση και προοδευτικά να την αναπτύξουμε:

```
>>> class Hero:
        pass

>>> #==============
>>> Hero.name='ADio'
>>> Hero.gender='male'
>>> Hero.age='68'
>>> print  'Hero name:', Hero.name, '\n', 'sex:', Hero.gender, '\n', 'age:', Hero.age
Hero name: ADio
sex: male
age: 68
```

Όταν ορίσαμε το blueprint της κλάσης έγινε χρήση της λέξης self σε αρκετά σημεία. Η self είναι ένας προσωρινός χώρος αποθήκευσης (placeholder) για το εκάστοτε αντικείμενο. Είναι ένα βοηθητικό στοιχείο ή αλλιώς μια βοηθητική 'μεταβλητή' επειδή δεν γνωρίζουμε ή δεν έχουμε αποφασίσει ακόμη να ορίσουμε(συγκεκριμενοποιήσουμε) το όνομα του εκάστοτε αντικειμένου, καθώς και διότι η γενίκευση βοηθά στην αφαιρετικότητα. Άρα όταν ορίσαμε το όνομα OzWizzard, για το class blueprint:Hero στην αρχή της προηγούμενης σελίδας, αυτό που έγινε είναι ότι αποθηκεύτηκε στην self. Όταν ορίσαμε την παράμετρο gender μέσα στην συνάρτηση self.gender έγινε το ίδιο επίσης και ούτω κάθε εξής. Εσωτερικά η Python είναι δομημένη με τέτοιο τρόπο έτσι ώστε να μην μπερδεύεται αν υπάρχουν πολλές ονομασίες που πρέπει να διαχειριστεί. Ακόμη, τονίζεται ότι η self τοποθετείται εξ ορισμού πάντα στην πρώτη θέση στο interface πεδίο, αλλά δεν είναι δεσμευμένη λέξη κλειδί στην Python.

Πέρα από την ανάλυση των κλάσεων απομένει η επεξήγηση της λειτουργίας και χρησιμότητα δυο ακόμα εργαλείων που χρησιμοποιεί ο πηγαίος κώδικας της FSM, αυτά είναι ο constructor και η main συνάρτηση. Ένας constructor προσφέρει με δυο λέξεις αρχικοποίηση ποσοτήτων. Εάν θέλουμε για παράδειγμα ένα σύνολο από περιεχόμενα να εμφανιστούν ή ένα σύνολο από εντολές να περαιωθούν οπωσδήποτε με το που εκκινήσει ένα πρόγραμμα προτού προχωρήσει παραπέρα, τότε ο constructor κάνει αυτήν ακριβώς την δουλειά. Με άλλα λόγια, οποτεδήποτε δημιουργούμε ένα αντικείμενο, συνήθως χρειάζεται να καλέσουμε χαρακτηριστικά τις μεθόδους για να μπορέσουμε να τις χρησιμοποιήσουμε και να πάρουμε αποτελέσματα. Αντιθέτως, όταν χρησιμοποιούμε constructors μόλις αρχικά δημιουργήσουμε ένα αντικείμενο οι τυχόν μέθοδοι που περιέχει ο εκάστοτε constructor καλούνται αυτομάτως, οπουδήποτε βρίσκονται οι εντολές μέσα στον κώδικα πραγματοποιούνται αυτόματα. Δηλαδή, με τον constructor μόλις δημιουργηθεί το αντικείμενο θα προσπελαστούν οι όποιες υπάρχουσες μέθοδοι δηλωμένες εντός του constructor αυτόματα. Το τυπικό του συντακτικό ακολουθεί την εξής μορφή:

```
def __init__ (self):
        {κώδικας}
```

Εκατέρωθεν του init(ολογράφως:initialise)βλέπουμε δυο κάτω παύλες. Ακολουθεί ένα απλό παράδειγμα.

```
class  Display:
    def __init__ (self):
        print 'hello people. This is a constructor and a method at the same time!'
```

Μόλις δημιουργήσουμε το αντικείμενο αμέσως και αυτομάτως θα εκτελεστούν όλες οι εντολές εντός του constructor:

```
>>> message=Display()
 hello people. This is a constructor and a method at the same time!
```

Αντιθέτως, νωρίτερα είδαμε ότι μόλις ορίσαμε ένα συνηθισμένο αντικείμενο π.χ. το OzWizzard με την εντολή OzWizzard=Hero() που δεν συνδέεται με constructor τίποτα δεν εκτελείται αυτομάτως.

Αφού έγινε επεξήγηση για τον constructor απομένει να γίνει μια προσέγγιση για την περίπτωση της main. Όπως για παράδειγμα έχουμε στην γλώσσα C μία κύρια συνάρτηση main για κάθε πρόγραμμα, κάτι αντίστοιχο μπορεί να υπάρξει και στην γλώσσα Python έτσι ώστε να διαθέτει, να εξομοιώνει την ύπαρξη μιας κύριας συνάρτησης. Η μόνη διαφορά από άλλες γλώσσες προγραμματισμού είναι ότι στην Python δεν υπάρχει main που να εκτελείται αυτομάτως(όχι με τον τρόπο που υπάρχει στις άλλες γλώσσες προγραμματισμού, στην Python αυτή υπονοείται ως όλος ο κώδικας που είναι στο top level) με το που ζητήσουμε να εκτελεστεί ο πηγαίος κώδικας μας, αλλά πρέπει να την κατασκευάσουμε εμείς χειροκίνητα με μια συνθήκη if-else, η οποία if-else είναι ο πιο προτιμώμενος από τους προγραμματιστές(όχι όμως και μοναδικός) τρόπος για αυτό το σκοπό. Προτιμάται αυτός ο τρόπος με if-else χάριν απλότητας και αμεσότητας κατανόησης όταν παρέλθει ένα μεγάλο χρονικό διάστημα από την τελευταία φορά που είδαμε τον κώδικα και μοιραία δε θα θυμόμαστε σχεδόν τίποτα.

Ο πιο κλασικός τρόπος σύνταξής της παρουσιάζει την ακόλουθη μορφή:

```
if __name__ == "__main__":
    {κώδικας}
elif:
    {κώδικας}
else:
    {κώδικας}
```

Το αν θα συμπεριλάβουμε ή όχι την elif και την else εξαρτάται από το τι θέλουμε να κάνει ο κώδικας μας, πάντως δεν είναι υποχρεωτικά και μπορούν να παραλειφθούν. Η __name__ είναι μια ενσωματωμένη μεταβλητή μέσα στην καρδιά της Python, χρησιμοποιείται ως έχει με δυο κάτω παύλες δεξιά και αριστερά της λέξης name και λαμβάνει την τιμή(evaluate) στο όνομα του εκάστοτε παρόντος(current) module. Αντιθέτως, αν το module εκτελεστεί απευθείας από το command line, τότε η μεταβλητή __name__ αντιστοιχίζεται με το αλφαριθμητικό == "__main__", δηλαδή η λογική πράξη == τίθεται(set) αληθές και εκτελείται ο κώδικας που ακολουθεί κατόπιν. Αυτό σημαίνει ότι υπάρχουν δυο δυνατοί τρόποι: ένα script αρχείο(ένα module) να εκτελείται απευθείας από command line ή να εισαχθεί (import) μέσα από κάτι άλλο πχ ένα άλλο module(script μέσα σε script με χρήση της εντολής import). Στην δεύτερη περίπτωση θα γίνουν import οι ορισμοί από τις διάφορες συναρτήσεις και κλάσεις του module αλλά σε αντίθεση με την πρώτη περίπτωση ο κώδικας που θα βρίσκεται εντός της main() δεν θα εκτελεστεί. Ένα πρακτικό παράδειγμα που αποκρυσταλλώνει το άνωθεν κείμενο είναι το ακόλουθο:

```
1.  # this script save as example1.py
2.  def test1():
3.      print "test1 function is inside example1.py"
4.  print "top level inside example1.py"
5.  #==================================================
6.  if __name__ == "__main__":
7.      print  "example1 trexei apeftheias"
8.  else:
9.      print "example1 egine import mesa se ena allo script arxeio"
```

Αυτό το script το σώζουμε ως example1.py. Αναπαριστά την πρώτη περίπτωση με την απευθείας κλήση του module και γράφουμε και σώζουμε ένα ακόμα script, το example2.py που αυτή τη φορά θα κάνει import το example1.py:

```
1.	# this script save as example2.py
2.	import example1
3.	print "top level inside example2.py"
4.	example1.test1()
5.	#=====================================================
6.	if __name__ == "__main__":
7.		print "example2.py ekteleitai apeftheias"
8.	else:
9.		print  "example2.py egine import mesa se kapoio allo module"
```

Αν τρέξουμε το example1.py τότε θα προκύψει το ακόλουθο αποτέλεσμα:
top level inside example1.py
example1 trexei apeftheias

Αν τρέξουμε το example2.py τότε θα προκύψει το ακόλουθο αποτέλεσμα:
top level inside example1.py
example1 egine import mesa se ena allo script arxeio
top level inside example2.py
test1 function is inside example1.py
example2.py ekteleitai apeftheias

Από όπου παρατηρούμε ότι όταν το module example1.py φορτώνεται(loaded) η μεταβλητή του __name__ ισούται με "example1" αντί για __main__ .

Το ίδιο παράδειγμα, ίδια λογική, με πιο απλοποιημένο κώδικα:

```
#save this as file2
print 'now we called %s'  % __name__
if __name__ == "__main__":
	print  and… again in %s' % __name__
```

```
#save as file1
import file2
```

Αν τρέξουμε το file1.py τότε θα προκύψει το ακόλουθο αποτέλεσμα:
now we called file2

Αν τρέξουμε το file2.py τότε θα προκύψει το ακόλουθο αποτέλεσμα:
now we called __main__
and… again __main__

Συνοψίζοντας το τι συμβαίνει στα δυο παραπάνω παραδείγματα βλέπουμε ότι όταν ο διερμηνευτής διαβάζει ένα script αρχείο θα εκτελέσει όλο τον κώδικα που θα βρει μέσα σε αυτό το αρχείο. Πριν εκτελέσει τον κώδικα θα ορίσει μερικές ειδικές μεταβλητές. Συνεπώς όταν ο διερμηνευτής τρέχει το script αρχείο(ή αλλιώς module) ως ένα κύριος πρόγραμμα(main program) θέτει την ειδική μεταβλητή __name__ να ισούται με "__main__" . Αν όμως αυτό το script αρχείο γίνει import ως περιεχόμενο(κομμάτι) ενός άλλου αρχείου τότες η __name__ θα τεθεί και θα ισούται με το όνομα του εκείνου module. Συνεπώς με τον να κάνουμε τον έλεγχο για την main() με την εντολή, μπορούμε να ελέγχουμε εμείς πότε θα εκτελεστεί απευθείας το module ως ένα πρόγραμμα και πότε θα θέλουμε απλώς να το κάνουμε import και από εκεί και πέρα να χρησιμοποιούμε όποιες από τις μεθόδους(συναρτήσεις) που αυτό τυχόν διαθέτει με την ίδια λογική που κάνουμε με το module math, os, sys και τα λοιπά. Συμπερασματικά ,βασιζόμαστε στην χρήση της **if** __name__ == "__main__": για να αποτρέψουμε συγκεκριμένο κώδικα από το να εκτελεστεί όταν το module εισαχθεί(imported).

 Copyright © 2014

7.2.2 Γραμμή προς γραμμή ανάλυση του κώδικα

Μια FSM(Finite State Machine)απαρτίζεται από συγκεκριμένο και περιορισμένο αριθμό καταστάσεων εξ' ου και η λέξη Finite(πεπερασμένος), το State δηλώνει ότι αποτελείται από καταστάσεις και τι τιμές(πως είναι) σε μια ακριβώς δεδομένη χρονική στιγμή. Το Machine δηλώνει κάτι το αυτοματοποιημένο. Άρα, έχοντας πλέον αποκρυπτογραφήσει όλες τις δομικές μονάδες από τις οποίες χτίζεται ο κώδικας της FSM και έχοντας αναλύσει πως λειτουργούν και συνεργάζονται αρμονικά τα διάφορα εργαλεία του πηγαίου κώδικα, μπορούμε να προχωρήσουμε στην ανάλυση του, έχοντας υπόψιν το σενάριο ότι πρόκειται για ένα LED συνδεμένο σε μια θύρα I/O.

Στην γραμμή ένα και δυο εισάγουμε(κάνουμε χρήση της import) δυο μεθόδους: την randit από το module random, η οποία απλά παράγει έναν τυχαίο ακέραιο και την clock από το time module με σκοπό να καταγράφουμε το πέρασμα του χρόνου του προγράμματος. Με τις δυο αυτές μεθόδους(randit και random) θα περιηγούμαστε μεταξύ των διάφορων καταστάσεων, αρκετά αργά σε πραγματικό χρόνο ώστε να είναι οπτικά αντιληπτό σε εμάς, μιας και η CPU μπορεί να εκτελέσει πολύ πιο γρήγορα τον κώδικα χρειαζόμαστε έναν τεχνητό τρόπο να επιβραδύνουμε την εμφάνιση των αποτελεσμάτων ώστε να γίνονται αντιληπτές οι μεταβάσεις και η κατάσταση της FSM σε μια δεδομένη χρονική στιγμή.

Στην γραμμή πέντε ακολουθεί η επόμενη σημαντική δήλωση. Εδώ ορίζεται μια μεταβλητή αντικείμενο με όνομα State. Η χρησιμότητα της έγκειται στο γεγονός ότι για να έχουμε καταστάσεις θα πρέπει να δημιουργήσουμε μια οντότητα(entity),ένα περίγραμμα που θα αναπαριστά την οντότητα κατάσταση. Με την βοήθεια αυτής της βασικής κλάσης καταστάσεων(state base class) θα μπορέσουμε να φτιάξουμε τις δυο καταστάσεις λειτουργίας του LED:HIGH και LOW.

Στις γραμμές επτά έως εννέα δημιουργούμε την ευσταθή κατάσταση HIGH με όνομα κλάσης LedOn, στις γραμμές έντεκα έως δεκατρία φτιάχνουμε την ευσταθή κατάσταση LOW με όνομα κλάσης LedOff. Σε κάθε μια από τις δυο κλάσεις ορίσαμε από μια μέθοδο, την Run, που δεν παίρνει ορίσματα εισόδου(εκτός από την υποχρεωτική self) και το μόνο που κάνει είναι να ενημερώνει τον χρήστη σε τι κατάσταση βρίσκεται η FSM σε μια δεδομένη χρονική στιγμή. Θα μπορούσαμε να προσθέσουμε μέσα σε αυτές τις δυο κλάσεις και άλλες μεθόδους που να προσφέρουν διάφορες άλλες λειτουργίες και χαρακτηριστικά στην FSM ανάλογα με τις απαιτήσεις της εκάστοτε εφαρμογής.

Κατόπιν, στις γραμμές δεκαεπτά έως είκοσι δύο φτιάξαμε το σώμα, την οντότητα, μιας βασικής κλάσης με όνομα Transfer που θα αναπαριστά τις μεταβάσεις, τα βελάκια, από την μια κατάσταση σε μια άλλη. Η κλάση Transfer απαρτίζεται από δυο μεθόδους: μία για μετάβαση σε κατάσταση (__init__ ()) και μία για εκτύπωση στην οθόνη(Run()) του τι κάνει εκείνη τι στιγμή η FSM. Η Transfer(object) κλάση θα κληρονομήσει στοιχεία από την object κλάση. Επίσης, βλέπουμε ότι η μια μέθοδος, η init, είναι ένας constructor με δυο ορίσματα. Αυτό που ζητά η κλάση μόλις δημιουργηθεί είναι ότι περνάμε μια μεταβλητή την toState ως αλφαριθμητικό οποιαδήποτε κι αν είναι η κατάσταση στην οποία μεταβαίνει η FSM.

Μετέπειτα, στις γραμμές 26 μέχρι 45 έχουμε την γένεση της FSM οντότητας. Μέσα στο σώμα της ορίζονται τέσσερις μέθοδοι. Η πρώτη, η __init__ , κάνει αρχικοποίηση γιατί από κάπου πρέπει να αρχίσει να λειτουργεί. Η παράμετρος MCU στον constructor θα περαστεί ως όρισμα μόλις δημιουργηθεί ένα αντικείμενο. Ακόμη, βλέπουμε ότι χρειάζεται ένας χώρος όπου θα αποθηκεύονται όλες οι ευσταθείς καταστάσεις γι αυτό το σκοπό δημιουργούμε ένα λεξικό το self.states ως κενό αφού θα γεμίσει με περιεχόμενα στην πορεία. Ότι κάναμε για τις καταστάσεις το ίδιο κάνουμε για τις μεταβάσεις: δημιουργούμε επίσης ένα κενό λεξικό που εκεί μέσα θα αποθηκεύονται όλες οι μεταβάσεις. Ομοίως ορίζονται δυο ακόμα μεταβλητές: self.currentState=None και self.trans=None με αρχικοποίηση το τίποτα, διότι θα πάρουν τιμές αργότερα και μπορούμε να τους φανταστούμε κάτι σαν δείκτες τρέχουσας θέσης. Η μεν είναι για να δεσμεύσει μια θέση στη μνήμη όπου θα αποθηκεύεται η τρέχουσα κατάσταση, ενώ η δε χρειάζεται για να αποθηκεύεται η πιο πρόσφατη μετάβαση. Παρακάτω, η μέθοδος SetState είναι μια μέθοδος που θα κοιτάξει για ένα οποιοδήποτε αλφαριθμητικό εισάγουμε εμείς(Namestate) εντός του

λεξικού εκείνης της δεδομένης της ευσταθούς κατάστασης(self.states) και σώζεται ως στιγμιότυπο, όπως επίσης θα θέσει την currentState μεταβλητή ίση με την εισαγόμενη τιμή. Η Τρίτη σε σειρά μέθοδος η Transfer εφαρμόζει την ίδια λογική αλλά για τις μεταβάσεις αυτή την φορά. Τέλος, η μέθοδος Run στην γραμμή σαράντα λέει ότι αν υπάρχει μια μετάβαση (if self.trans) αποθηκευμένη μέσα στην self.trans της γραμμής 32 τότε θα εκτελέσουμε την μετάβαση εκείνη, θα θέσουμε την κατάλληλη κατάσταση που γίνεται η μετάβαση και θα επαναφέρουμε σε κενό περιεχόμενο την self.trans για επόμενη χρήση. Τέλος θα γίνει η εκτέλεση της με την εντολή self.currentState.Run() δηλαδή θα της δοθεί πνοή.

Στις γραμμές 49 μέχρι 52 φτιάχνουμε την κλάση MCU που εκεί μέσα θα αποθηκευτούν όλα τα χαρακτηριστικά και ιδιότητες όπως και η ίδια η FSM. Ας πάρουμε μια αναλογία από έναν τελεστικό ενισχυτή ,τότε όλα τα τεχνικά χαρακτηριστικά του θα αποθηκευόντουσαν σε αυτήν την κλάση ,μέγιστη θερμοκρασία λειτουργίας, ονομαστική τάση/ρεύμα λειτουργίας και οτιδήποτε άλλο. Άλλη αναλογία είναι από τους εικονικούς χαρακτήρες σε βιντεοπαιχνίδια. Σε μια τέτοια περίπτωση αυτή εδώ η κλάση θα κρατούσε πληροφορίες για οπλισμό, ρουχισμό χαρακτήρα και άλλα. Συνεχίζοντας ,στον constructor που ορίζεται μέσα της δημιουργούμε ένα στιγμιότυπο της FSM,αυτής που δημιουργήσαμε στις γραμμές 26 μέχρι 45.Επίσης ορίζεται ως αληθές και η ιδιότητα της FSM στην γραμμή 52,γιατί από κάπου πρέπει να ξεκινήσει.

Τελικά, στις γραμμές 55 μέχρι 77 δημιουργούμε το πρόγραμμα που θα καλεί, τρέξει αυτήν την FSM. Στην γραμμή 56 δημιουργούμε ένα στιγμιότυπο της MCU κλάσης, πρακτικά ένα αντικείμενο είναι με άλλα λόγια. Στην γραμμή 58 ορίζουμε την ενεργή περίπτωση που ανάβει το LED και την συσχετίζουμε(το σύμβολο ίσον) με την ευσταθή κατάσταση(κλάση) LedOn που γεννήθηκε νωρίτερα. Οπότε το στιγμιότυπο της ευσταθής κατάστασης LedOn που δημιουργήσαμε εδώ θα αποθηκευτεί στο λεξικό states της FSM (FSM.states).Ομοίως αντίστοιχα πράγματα στην γραμμή 59 για την περίπτωση σβηστού Led, και των μεταβάσεων για τις γραμμές 60-61. Κατόπιν, στην γραμμή 63 θέτουμε ένα σημείο εκκίνησης της FSM σε HIGH γιατί από κάπου πρέπει να ξεκινήσει να λειτουργεί. Στην συνέχεια στις γραμμές 65 και μετά αρχίζει ένας επαναληπτικός βρόχος 12 συνολικά επαναλήψεων που θα τυπώσει 12 αποτελέσματα στην οθόνη. Με άλλα λόγια ότι βρίσκεται από την γραμμή 65 και μετά είναι το κυρίως πρόγραμμα που διατρέχει ότι κώδικα γράψαμε στις γραμμές 55 έως 63.

Στην γραμμή 66 ορίσαμε μια μεταβλητή για να καταγράφουμε τον χρόνο εκκίνησης και θέτουμε το χρονικό διάστημα timedelay ίσο με ένα δευτερόλεπτο. Μετέπειτα στην γραμμή 68 ορίζουμε μια φωλιασμένη επαναληπτική δομή με μια συνθήκη, η οποία ορίζει ότι όσο η συνάρτηση clock() είναι μικρότερη από το άθροισμα των startTime και timedelay εκτελείται ότι κώδικας βρίσκεται μέσα στην while. Με άλλα λόγια δεν θέλουμε η clock() να ξεπεράσει το ένα δευτερόλεπτο και θα είναι αληθής. Η εντολή pass που βρίσκεται μέσα στην while λέει να συνεχίζει να εκτελείται επαναληπτικά η while. Αν όμως η clock() έχει τύχει να ξεπεράσει το ένα δευτερόλεπτο τότε προχωράμε στην γραμμή 70 που είναι μια συνθήκη if που καλεί μέσα στην παρένθεση την randint, η οποία παράγει έναν τυχαίο ακέραιο αριθμό μεταξύ 0 και 2.Επείδη το αποτέλεσμα του randint μπορεί να είναι το 0 ή 1, αν είναι το μηδέν θα παρακάμψει, δεν θα εκτελέσει ότι βρίσκεται στο μέσα στο σώμα της δομής if-else, αλλά θα εκτελεστεί η γραμμή 77 μονάχα, άρα δεν πρόκειται να αλλάξει ευσταθείς καταστάσεις.

Από την άλλη μεριά, αν η randint ισούται με ένα τότε θα προκαλέσει μια μετάβαση στην αντίθετη κατάσταση από αυτή που βρίσκεται η FSM. Δηλαδή, αν το φως είναι μέχρι στιγμής αναμμένο (γραμμή 71) τότε θα σβήσει με τις γραμμές 72-73. Αν το φως είναι σβηστό(γραμμή 74) τότε θα ανάψει με τις γραμμές κώδικα 75-76.

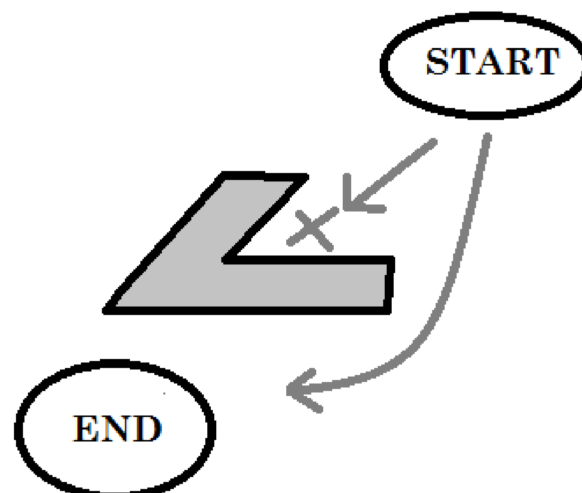
 Copyright © 2014

7.3 Εύρεση συντομότερης τυφλής διαδρομής

Σε αυτή την ενότητα θα παρουσιαστεί ένας πιο έξυπνος αλγόριθμος που μπορεί να προσθέσει περισσότερη νοήμον τεχνητή νοημοσύνη σε ένα σύστημα και να το κάνει να συμπεριφέρεται εξυπνότερα από ότι μια σκέτη FSM. Αν η FSM προσφέρει τη δημιουργία καταστάσεων και τις μεταβάσεις από μια κατάσταση σε μια άλλη όταν ικανοποιούνται οι απαραίτητες συνθήκες, τότε ο ακόλουθος αλγόριθμος μπορεί να βρει το πιο σύντομο μονοπάτι - διαδρομή από μια κατάσταση αφετηρία σε μια κατάσταση τερματισμό. Πρόκειται για ένα πάρα πολύ χρήσιμο αλγόριθμο με πλήθος δυνατών εφαρμογών που θα μπορεί να βρει εφαρμογή όπως λόγου χάρη σε βιντεοπαιχνίδια και σε έξυπνα ηλεκτρονικά συστήματα.

Ο τρόπος με τον οποίο λειτουργεί αυτός ο αλγόριθμος είναι ο εξής: έχουμε μια κατάσταση αφετηρία, μια κατάσταση τερματισμός, το ερώτημα είναι πως θα φτάσουμε από την αφετηρία στον στόχο, πως θα υπολογίζουμε τον πρόοδο από την αφετηρία στον τερματισμό ,πως θα επιλέγεται το συντομότερο μονοπάτι και πως θα δημιουργείται το εκάστοτε επόμενο βήμα με κατεύθυνση προς τον στόχο. Για έναν άνθρωπο αυτό το πρόβλημα είναι εύκολο να λυθεί απλώς πηγαίνοντας πχ με τα πόδια αν μιλάγαμε για την περίπτωση κάλυψης οδικής διαδρομής, αλλά για τον υπολογιστή που δεν έχει αληθινή νοημοσύνη ένα τέτοιο πρόβλημα είναι άθλος, διότι δεν γνωρίζει η μηχανή πώς να το πράξει, δεν είναι σε θέση να σκέφτεται, πρέπει κάποιος άνθρωπος να την προγραμματίσει πώς να το κάνει. Ο αλγόριθμος στην προκείμενη περίπτωση είναι «κουρδισμένος» για κάθε νέο του βήμα να ανακαλύπτει το επόμενο βήμα προς το τέρμα. Ο αλγόριθμος δεν εγγυάται πάντα 100% επιτυχία ,μπορεί να υπάρξει περίπτωση να αποτύχει, όπως και οι άνθρωποι μπορούν να αποτύχουν σε κάτι που κάνουν.

Η λογική που κρύβεται πίσω από αυτό τον αλγόριθμο λέει ότι πρώτα δημιουργούμε μια λίστα από απογόνους-πιθανά μονοπάτια από την εκάστοτε τρέχουσα θέση για όλα τα επόμενα δυνατά βήματα-διαδρομές προς τον στόχο. Κατόπιν αποθηκεύουμε σε μια ταξινομημένη λίστα(priorityQueue) όλους τους συνδυασμούς βημάτων που οδηγούν στο επιθυμητό αποτέλεσμα. Η ταξινόμηση γίνεται μετρώντας απόσταση μεταξύ του σημείου που βρίσκεται σε μια δεδομένη στιγμή και του τερματισμού-στόχου. Το κοντινότερο βήμα προς τον στόχο επιλέγεται και έρχεται πρώτο στην ταξινόμηση άρα έχει προτεραιότητα. Μετά στο επόμενο βήμα ξανά η ίδια διαδικασία μέχρις ότου φτάσει ο αλγόριθμος στον τερματισμό. Επισημάνεται ότι η λίστα δεν απορρίπτει καμία επιλογή(βήμα) κρατά αποθηκευμένες όλες τις δυνατές επιλογές. Συνεπώς , αν ένα μονοπάτι δεν δίνει λύση, μπορεί να πειραματιστεί ο αλγόριθμος με τα υπόλοιπα μονοπάτια(ως μονοπάτι νοείται ένα σύνολο από διαδοχικά βήματα). Το ενδιαφέρον με αυτό τον αλγόριθμο είναι ότι αν βρεθεί σε κάποιο σημείο σε ένα αδιέξοδο μπορεί να επιστρέψει προς τα πίσω, δηλαδή να «κοιτάξει» το ιστορικό των προηγούμενων βημάτων, να γυρίσει δηλαδή προς τα πίσω σε μια προσπάθεια να πάει τριγύρω από ένα εμπόδιο.



Σχήμα 22. Παράκαμψη εμποδίου

#The Blind Pathfinder

```
1   from Queue import PriorityQueue
2
3   class State(object):
4       def __init__(self,value,elder,begin=0,target=0):
5           self.offsprings=[]
6           self.elder=elder
7           self.value=value
8           self.distance=0
9           if elder:
10              self.path=elder.path[:]
11              self.path.append(value)
12              self.begin=elder.begin
13              self.target=elder.target
14          else:
15              self.path=[value]
16              self.begin=begin
17              self.target=target
18
19      def GetDistance(self):
20          pass
21      def CreateOffsprings(self):
22           pass
23
24  class subState (State):
25      def __init__(self,value,elder,begin=0,target=0):
26          super(subState,self).__init__(value,elder,begin,target)
27          self.distance=self.GetDistance()
28
29      def GetDistance(self):
30          if self.value==self.target:
31              return 0
32          dist=0
33          for i in range(len(self.target)):
34              letter=self.target[i]
35              distance+=abs(i-self.value.index(letter))
36          return distance
37
38      def CreateOffsprings(self):
39          if not self.offsprings:
40              for i in xrange(len(self.target)-1):
41                  val=self.value     # make a copy of val of line 42.
42                  val=val[:i]+val[i+1]+val[i]+val[i+2:]
43                  offspring=subState(val,self)
44                  self.offsprings.append(offspring)
45
46  class Pathfinder_Solver:
47      def __init__(self,begin,target):
48          self.path=[]
49          self.visitedQueue=[]
50          self.priorityQueue=PriorityQueue()
51          self.begin=begin
52          self.target=target
53
```

```
54        def Solve(self):
55            beginState=subState(self.begin,0,self.begin,self.target)
56            count=0
57            self.priorityQueue.put((0,count,beginState))
58            while(not self.path and self.priorityQueue.qsize()):
59                closestoffspring=self.priorityQueue.get()[2]
60                closestoffspring.CreateOffsprings()
61                self.visitedQueue.append(closestoffspring.value)
62                for offspring in closestoffspring.offsprings:
63                    if offspring.value not in self.visitedQueue:
64                        count +=1
65                        if not offspring.distance:
66                            self.path=offspring.path
67                            break
68                        self.priorityQueue.put((offspring.distance,count,offspring))
69            if not self.path:
70                print 'Error: Stoxos: '+self.target+' den vrethike'
71            return self.path
72    #______________________________________________________________________________
73    #______________________________________________________________________________
74    if __name__ == "__main__":
75        START= 'ytrweq'
76        STOP='qwerty'
77        print 'ekinisi anazitisi diadromis:'
78        P=Pathfinder_Solver(START,STOP)
79        P.Solve()
80        for i in xrange(len(P.path)):
81            print '%d) ' %i + P.path[i]
```

Ανάλυση κώδικα:

Αρχικά, στη γραμμή ένα εισάγουμε από το module Queue μια δομή δεδομένων την PriorityQueue, η οποία θα οργανώνει-ιεραρχεί τα στοιχεία σε ένα σύνολο από στοιχεία. Μπορούμε να την φανταστούμε σαν ένα λεξικό που έχει ως περιεχόμενα του λίστες. Στις γραμμές τρία έως είκοσι δύο δημιουργούμε μια κλάση με όνομα State που είναι ο σκελετός των καταστάσεων, μια θεμελιώδη κλειδί κλάση(base class). Αυτό που κάνει είναι ότι θα χρησιμοποιηθεί για να αποθηκεύσει όλα τα υποχρεωτικά βήματα που θα διατρέχει ο αλγόριθμος. Αποτελείται από τρεις συναρτήσεις που ορίζονται στις γραμμές 4, 19 και 21. Στην γραμμή 4 έχουμε έναν constructor που κάνει αρχικοποίηση τιμών. Βλέπουμε ότι για την πρώτη αρχική κατάσταση(για να πάρει μπροστά ο κώδικας) από κάπου πρέπει να αρχίσει, συνεπώς το όρισμα του begin πρέπει να είναι ίσο με μηδέν- ομοίως και ο τερματισμός γιατί δεν υπάρχουν προηγούμενα βήματα-δεν έχουμε elder. Στα επόμενα βήματα μετά την εκκίνηση κάθε απόγονος βήμα θα μπορεί να αντλεί την απαραίτητη πληροφορία από την μεταβλητή elder που συμβολίζει το παρελθοντικό βήμα, μετά την πρώτη φορά, η μόνη πληροφορία που θα χρειαζόμαστε είναι οι δυο παράμετροι elder και η τιμή value του τρέχοντος απόγονου(value, elder). Μέσα στον constructor η τοπική μεταβλητή self.offsprings=[ ] είναι μια λίστα που σταδιακά θα γεμίζει, αναπαριστά όλες τις επόμενες αμέσως πιο γειτονικές δυνατές επιλογές βήματα όταν βρισκόμαστε στο ι-οστό βήμα. Ο αλγόριθμος σε παρακάτω γραμμές έχει λάβει τα μέτρα του ώστε να μην τίθεται θέμα να υπάρξουν διπλά αντίγραφα μιας ίδιας διαδρομής (spam). Επίσης ορίζεται θέση αποθήκευσης του εκάστοτε elder , της τιμής value που θα αναπαριστά το εκάστοτε απόγονο και την εκάστοτε απόσταση από τερματισμό με την μεταβλητή self.distance. Την self.distance εδώ απλώς της δίνουμε ύπαρξη και την αρχικοποιούμε στο μηδέν αλλά στην πορεία σε παρακάτω γραμμές θα πάρει τιμή. Στην γραμμή εννέα δημιουργούμε μια συνθήκη if, η οποία λέει ότι αν υπάρχει elder τότε με το σύμβολο [:] δημιουργούμε ένα αντίγραφο του elder.path(είναι λίστα) και το αποθηκεύουμε ως το τρέχον self.path. Το σύμβολο [:] είναι σημαντικό γιατί χωρίς αυτό θα καταστρέφαμε το όποιο παρελθοντικό μονοπάτι είχε διανυθεί μέχρι στιγμής, θα χάνονταν τα προηγούμενα βήματα πάνω στα οποία χτίζονται τα επόμενα. Με την εντολή στη γραμμή 11 αποθηκεύουμε την τρέχουσα τιμή βήμα στο χτισμένο μέχρι στιγμής μονοπάτι. Επιπλέον, έτσι θα γνωρίζει που βρίσκεται σε μια δεδομένη στιγμή.

Επίσης, στις γραμμές 12 και 13 δεσμεύουμε δυο μεταβλητές για αποθήκευση της κατάστασης εκκίνησης (self.begin) και της κατάστασης τερματισμού(self.target). Οι γραμμές 14 έως 17 λένε ότι αν δεν υπάρχει elder (δηλαδή παρελθόν),θα ξεκινήσουμε ένα νέο μονοπάτι που θα είναι μια λίστα από αντικείμενα(self.path, γραμμή 15), ξεκινώντας με την όποια τρέχουσα τιμή(γραμμή 15, η [value]).Οπότε σε αυτή την else περίπτωση θα πρέπει επίσης να δημιουργηθούν οι δυο καταστάσεις self.begin και self.target. Τέλος, ορίζουμε δυο κενές συναρτήσεις, οι γραμμές 19 και 21, που θα συγκεκριμενοποιηθούν αργότερα στην υποκλάση subState.

Εν συνέχεια, στις γραμμές 24 μέχρι 44 ορίζεται η subState κλάση με περιεχόμενα έναν constructor και δυο μεθόδους. Ο constructor παίρνει ορίσματα τα self, value, elder, begin, target. Βλέπουμε ότι εδώ στον constructor στην γραμμή 26 αρχικοποιούμε τις τιμές της βασικής κλάσης που είναι η State κλάση. Επίσης, αντικαθιστούμε την τιμή της μεταβλητής self.distance με αυτή που θα παράγει η μέθοδος GetDistance(). Η συνάρτηση GetDistance() στο σώμα της κάνει έναν έλεγχο αν έχουμε ήδη φτάσει στο τέρμα(γραμμή 30) για να μην κάνει άδικα υπολογισμούς. Αν δεν συμβαίνει αυτό, τότε θα αρχίσει να κάνει υπολογισμούς στις επόμενες γραμμές(32-36). Με την επαναληπτική δομή for(γραμμή 33) θα διατρέχουμε κάθε letter προς τον στόχο (γραμμή 34) και η απόσταση ισούται με την δήλωση στην γραμμή 35 όταν ολοκληρωθεί η for. Επομένως, συλλέγουμε το τρέχον letter και στην επόμενη γραμμή παίρνουμε τον δείκτη index του τρέχοντος letter στην τρέχουσα τιμή value και την αφαιρούμε αυτή την τιμή από αυτή που βρισκόμαστε στη δεδομένη στιγμή. Άρα, θα προκύψει η απόσταση που βρίσκεται το letter από την θέση τερματισμού(target) κι έτσι η γραμμή 27 συλλέγει το αποτέλεσμα που επιστρέφει η γραμμή 36.

Παρακάτω στις γραμμές 38-44 ορίζουμε μια μέθοδο που θα παράγει τους απογόνους βήματα προς τον τέρμα goal. Αν δεν υπάρχουν απόγονοι(γραμμή 39) εκτελείται το σώμα του επαναληπτικού βρόχου for που ακολουθεί και θα γεννήσει τους απογόνους διαδρομές. Αυτό που κάνει η for είναι να εξετάσει εξαντλητικά κάθε δυνατό συνδυασμό των γραμμάτων στη γραμμή 42. Αυτό το πετυχαίνουμε ανταλλάζοντας θέση του δεύτερου γράμματος val[i+1] με το πρώτο val[i] για κάθε δυνατό διαθέσιμο ζευγάρι γραμμάτων. Η val[:i] συμβολίζει την αρχή τι υπήρχε μέχρι στιγμής, ενώ η val[i+2:] συμβολίζει το τέλος των γραμμάτων κι έτσι έχουμε μια νέα αλληλουχία γραμμάτων που αποθηκεύεται στην μεταβλητή val. Στην γραμμή 43 είναι η τελική δημιουργία του πιο πρόσφατου απόγονου(άρα το τρέχον βήμα). Εδώ, περνάμε ως ορίσματα την val που μόλις δημιουργήθηκε στην προηγούμενη γραμμή για να αποθηκευτεί στον απόγονο offspring και η self είναι για αποθήκευση του elder για το τρέχον απόγονο(σχέση πατέρας- γιος είναι ένας άλλος τρόπος να το σκεφτόμαστε). Η γραμμή 44 απλά προσθέτει αυτόν τον πιο πρόσφατο απόγονο στην λίστα με όλους τους απογόνους, από τον πρώτο μέχρι και τον πιο πρόσφατο.

Στην γραμμή 46 αρχίζουν τα πιο ενδιαφέρονται. Εδώ δημιουργούμε την κλάση Pathfinder_Solver με περιεχόμενα έναν constructor και μια έξυπνα κατασκευασμένη μέθοδο. Η γραμμή 48 περιέχει την μεταβλητή που θα αποθηκευτεί μέσα της το τελικό αποτέλεσμα, την τελική λύση από την αρχή μέχρι το τέλος, το ένα και μοναδικό μονοπάτι(ακολουθία βημάτων) που λύνει το πρόβλημα. Η επόμενη γραμμή αποθηκεύει όλες τις ενδιάμεσα επισκεπτόμενες περιπτώσεις(βήματα απογόνους) έτσι ώστε να αποφύγουμε να επισκεφτούμε εις διπλούν κάποιον απόγονο και συνεπώς να γλιτώσουμε από ατέρμονα βρόχο. Στις γραμμές 51 και 52 απλά σώζουμε της μεταβλητές της κλάσης που βλέπουμε στην γραμμή 47. Η γραμμή 54 είναι μια μέθοδος που θα λύσει το πρόβλημα εύρεσης της διαδρομής. Στην γραμμή 55 έχουμε το σημείο εφαρμογής, εκκίνηση, το μηδέν στην παρένθεση σημαίνει ότι ακόμα δεν υπάρχει elder(πρόγονος) αφού από αυτό το σημείο θα αρχίσει. Στη γραμμή 56, η μεταβλητή count, έχουμε αρχικοποίηση μετρητή που παρακάτω θα αυξάνει με σκοπό να απαριθμεί τους απογόνους και δημιουργεί μια τιμή που δείχνει σε πιο υπ' αριθμόν απόγονο είμαστε σε μια δεδομένη στιγμή. Στην γραμμή 57 γίνεται χρήση της PriorityQueue με ορίσματα το 0 που είναι ο αριθμός προτεραιότητας(αν δεν υπάρχουν στοιχεία στην PriorityQueue, που δεν υπάρχουν, μπορούμε να βάλουμε ότι αριθμό θέλουμε χωρίς πρόβλημα), η μεταβλητή count και η beginState που γεννήθηκε στην γραμμή 55. Η εντολή put απλά λέει πρόσθεσε ότι έχει μέσα η πλειάδα στην PriorityQueue.

Κατόπιν, στην γραμμή 58 και μετά είναι η καρδιά της επίλυσης. Η while λέει ότι αν το self.path είναι άδειο και η PriorityQueue έχει μέγεθος, έχει κάτι μέσα της τότε να εκτελεστεί ο κώδικας μέσα στο σώμα της while. Στην γραμμή 59, η get() και το [2] λένε να επιλεγεί το δεύτερο στοιχείο, δηλαδή η παράμετρος beginState της γραμμής 57. Συνεπώς, στην γραμμή 59 εισάγουμε μια νέα μεταβλητή και στην επόμενη γραμμή καλούμε την συνάρτηση για δημιουργία απογόνων. Στην ακριβώς επόμενη γραμμή θα

προσθέσουμε αυτό το πιο πρόσφατο απόγονο στην επισκεπτόμενη ουρά αναμονής(visitedQueue, αυτό που κάνει είναι να καταγράφει όλους τους απογόνους που επισπευτήκαμε). Κατόπιν, διατρέχουμε όλους τους απογόνους που γεννηθήκαν σε αυτή την εκάστοτε δεδομένη κατάσταση(γραμμή 62) , αν ο απόγονος που δημιουργήθηκε δεν το επισκέφτηκε η αναζήτηση(γραμμή 63), τότε αυξάνει ο μετρητής κατά ένα και στις γραμμές 65-66 αν η απόσταση δεν είναι μηδέν και δεν υπάρχει άλλος απόγονος τότε θέτει ότι δηλώνει η γραμμή 66 και είναι η τελική λύση το όλου προβλήματος. Η εντολή break τερματίζει την for, αν τερματιστεί η for τότε πάμε πίσω στην while όπου το όρισμα self.path της while έχει τεθεί και τερματίζει έτσι η while. Αν δεν έχει βρεθεί μονοπάτι, τότε οι γραμμές 69-70 επιστρέφουν ένα μήνυμα λάθους ότι δε βρέθηκε μονοπάτι-διαδρομή που να οδηγεί στο τέρμα(target) κι άρα στην λύση. Η γραμμή 71 επιστρέφει το τελικό αποτέλεσμα για να εκτυπωθεί πιο κάτω στην main() όταν καλέσουμε την συνάρτηση Pathfinder_Solver.Solve().

Τέλος, έχουμε την συνάρτηση main(). Για δοκιμή του αλγόριθμου στο σύνολο του, ορίζουμε ως αρχική κατάσταση προς λύση την ανακατεμένη λέξη 'ytrweq' και ζητάμε από τον αλγόριθμο να προσπαθήσει να βρει (σε πόσα βήματα)και να τυπώσει όλα τα ενδιάμεσα βήματα μέχρι τον συντομότερο δρόμο τερματισμό, που είναι η λέξη στόχος(target): 'qwerty'. Στην γραμμή 78-79 εκκινούμε την επίλυση καλώντας από την κλάση Pathfinder_Solver την συνάρτηση Solve. Η εντολή for στην γραμμή 80-81 χρειάζεται για να τυπώσει το τελικό αποτελέσματα: το αρχικό σημείο, το τελικό σημείο και τα ενδιάμεσα βήματα. Το τελικό αποτέλεσμα είναι:

```
>>>
ekinisi anazitisi diadromis:
0) ytrweq
1) ytwreq
2) ywtreq
3) ywrteq
4) ywretq
5) ywertq
6) wyertq
7) weyrtq
8) werytq
9) wertyq
10) wertqy
11) werqty
12) weqrty
13) wqerty
14) qwerty
>>>
```

Σχήμα 23. Δοκιμή αλγόριθμου για είσοδο: ytrweq.

Συμπεράσματα μέρους Ι και μέρους ΙΙ

Μέρος Ι:

Ξεκινάμε με μια γενική εισαγωγή στο πρώτο κεφάλαιο για τις γλώσσες προγραμματισμού και ιδιαίτερα την Python,γίνεται μια αναφορά σε τέσσερα διαφορετικά μοντέλα προγραμματισμού, εξηγούνται μερικοί λόγοι για τους οποίους η γλώσσα Python αυξάνει σε δημοτικότητα και διαθεσιμότητα από ένα ευρύ κοινό, που αποφασίζει να επενδύσει σε χρόνο, κόπο και χρήμα για να μάθει να προγραμματίζει στην Python.

Κατόπιν, στα επόμενα κεφάλαια αρχίζει να γίνεται μια πιο στοχευμένη περιοδεία σε διάφορες δυνατότητες που προσφέρει η γλώσσα Python. Στο δεύτερο κεφάλαιο δίνεται έμφαση στους διάφορους τύπους μεταβλητών και αριθμών. Ξεχωριστό αυτοτελές κεφάλαιο αφιερώνεται για τις συναρτήσεις που είναι το σημείο κλειδί σε κάθε γλώσσα προγραμματισμού. Συνεπώς, για τις συναρτήσεις παρουσιάζεται ένας τρόπος καλής και ορθής χρήσης τους για την συγκεκριμένη γλώσσα προγραμματισμού. Εν συνέχεια, σε επόμενο κεφάλαιο γίνεται μια ιδιαίτερη μνεία για τις επαναληπτικές δομές και γενικότερα για τις εντολές ελέγχου ροής προγράμματος. Ειδική παράγραφος αφιερώνεται για την συνάρτηση lambda.

Ακολούθως, στο κεφάλαιο πέντε παρουσιάζονται οι βασικοί τύποι δεδομένων που βρίσκουν εφαρμογή στην γλώσσα προγραμματισμού Python όπως είναι η λίστα, το λεξικό και η πλειάδα. Αυτά είναι βασικότατα εργαλεία που η κατανόηση τους είναι απαραίτητη για ανάπτυξη εφαρμογών. Ειδική αναλυτική ενότητα γίνεται για το hashtable και πως βρίσκει εφαρμογή στην Python. Επιπλέον, αφιερώνεται ένα κεφάλαιο παρουσιάζοντας σε ικανό βάθος και έκταση έξι σημαντικά modules,δηλαδή αυτόνομα αρχεία που περιέχουν συναρτήσεις που ονομάζονται μέθοδοι.

Διαβάζοντας το παρόν σύγκραμα ο αναγνώστης με παρελθόν στην C/C++ θα διαπιστώσει ότι στην Python δεν υπάρχουν pointers, η κάθε εντολή σε κάθε γραμμή δεν τερματίζεται με ερωτηματικό αλλά αντί αυτού γίνεται χρήση εσοχών. Ακόμη ο τρόπος με τον οποίο γίνεται χρήση της συνάρτησης main() είναι διαφορετικός στην γλώσσα Python σε σχέση με άλλες γλώσσες προγραμματισμού ,γιατί στην Python δεν εκτελείται η main() αυτομάτως με τον τρόπο που γίνεται στις άλλες γλώσσες προγραμματισμού, αλλά πρέπει να ακολουθηθούν κάποια διαδοχικά βήματα. Τέλος, ένα ακόμη ιδιαίτερα αντιληπτό χαρακτηριστικό στην Python είναι ότι τα πάντα είναι αντικείμενα, ότι και να χρησιμοποιήσουμε είναι αντικείμενο: μεταβλητή, πλειάδα, κλάση κλπ.

Μέρος ΙΙ:

Στο μέρος δυο στρέφεται η προσοχή αρχικά στον εντοπισμό και αντιμετώπισης σφαλμάτων. Η ανίχνευση, η κατανόηση της αιτίας του προβλήματος και η διόρθωση ενός σφάλματος είναι μια διαδικασία που μπορεί να καταβάλει σωματικά και νοητικά τον προγραμματιστή. Αυτό το κομμάτι του debugging είναι ένα ιδιαίτερο θέμα που αξίζει να χρήζει διερεύνησης. Είναι πολύ πιθανόν να υπάρξουν στιγμές αδυναμίας εύρεσης του προβλήματος όσο απλό κι αν είναι. Παράγοντες που μπορούν να επηρεάσουν μεταξύ άλλων είναι η διάθεση, η κόπωση, η καθαρότητα του μυαλού, το φαινομενικά σωστό κατά την άποψη του προγραμματιστή νοητικό μοντέλο στο οποίο βασίζεται ο κώδικας του κι η αδυναμία αντίληψης ότι πιθανόν να είναι εσφαλμένο σε κάποιο σημείο.

Στη συνέχεια, έχοντας προϊδεαστεί για τους πιθανούς κινδύνους που μπορούν να οδηγήσουν σε λάθη εστιάζουμε την προσοχή μας σε ένα κώδικα πολλών γραμμών, μια FSM. Η πιο τυπική χρήση μιας FSM είναι ως μια πολύ απλή τεχνητή νοημοσύνη, έναν αλγόριθμο, μια διαδικασία που με προδιαγεγραμμένα βήματα καθορίζει την συμπεριφορά ενός συστήματος, τι μπορεί να πράξει. Δεν είναι αυτοσκοπός η ανακάλυψη του κώδικα καθώς στην βιβλιογραφία υπάρχουν αρκετές πηγές. Εντούτοις, η έμφαση δίνεται στην ανάλυση του κώδικα. Όταν συναντάμε έναν πηγαίο κώδικα πολλών γραμμών πρέπει να είμαστε σε θέση να έχουμε ένα σύστημα, μια μέθοδο να προσεγγίσουμε, να διαβάσουμε και να κατανοήσουμε τον κώδικα. Συνεπώς ,γίνεται μια προσπάθεια να δειχθεί ένας προτεινόμενος τρόπος προσέγγισης μακροσκελών προγραμμάτων. Τέλος το ίδιο ισχύει και για τον αλγόριθμο εύρεσης συντομότερου μονοπατιού.

Copyright © 2014

Παραπομπές

pdf/ebooks/βιβλία:
- Data structures and algorithms in Python by Michael T. Goodrich,Roberto, Tamassia,Michael, H.Goldwasser
- Invent Your Own Computer Games with Python, second edition by Al Sweigart
- Think python by Allen Downey
- Σημειώσεις θεωρίας προγραμματισμού Β' εξαμήνου για την C γλώσσα του Δρ. Πετράκη Νικόλαου
- Programming Game AI by Example by Mat Buckland
- Dive into Python by Mark Pilgrim
- Game Engine Architecture by Jason Gregory
- Learn Python The Hard Way, 3rd Edition by Zed. A. Shaw
- Artificial Intelligence for Games by John Funge
- Η γλώσσα C++ σε βάθος, Νίκος Μ. Χατζηγιάννης

Ιστοσελίδες:
http://www.python.org/
http://www.wxpython.org/
http://www.khanacademy.org/science/computer-science
http://forums.devshed.com/python-programming-11/ -forum
http://pymotw.com/2/sys/index.html
http://www.lightbird.net/py-by-example
http://www.pythoncentral.io

Video στο διαδίκτυο:
http://www.youtube.com/watch?v=RrPZza_vZ3w - Learn Python Through Public Data Hacking
http://www.youtube.com/watch?v=tKTZoB2Vjuk - Google Python Class Day 1 Part 1
http://www.youtube.com/watch?v=EPYupizJYQI - Google Python Class Day 1 Part 2
http://www.youtube.com/watch?v=haycL41dAhg - Google Python Class Day 1 Part 3
http://www.youtube.com/show/python101 - Python 101: episodes 1-11
http://www.youtube.com/watch?v=ob4faIum4kQ - A* Algorithm
http://www.youtube.com/watch?v=4Mf0h3HphEA&list=PLEA1FEF17E1E5C0DA -ALL 43 Python
http://www.youtube.com/watch?v=E_kZDvwofHY-Advanced Python or Understanding Python
http://www.youtube.com/watch?v=E45v2dD3IQU - Finite-State Machines (FSM)

MIT video:
http://www.youtube.com/watch?v=bX3jvD7XFPs&list=PL5F90ACB29A774A5F
http://www.youtube.com/watch?v=SLvTCHhu5SE&list=PL5F90ACB29A774A5F
http://www.youtube.com/watch?v=ggxY20cXql8&list=PL5F90ACB29A774A5F

Δεύτερη σελίδα - αρχική πηγή προέλευσης της εικόνας με το φίδι πριν υποστεί επεξεργασία η εικόνα:
https://www.google.cz/search?q=python+picture&newwindow=1&tbm=isch&tbo=u&source=univ&sa=X&ei=hn2OUtYI7pTtBo3xgcgK&ved=0CCsQsAQ&biw=1280&bih=683#facrc=_&imgdii=_&imgrc=3AHEcxHu-6vzNM%3A%3BzS56iFiOCi7kBM%3Bhttp%253A%252F%252Fi.dailymail.co.uk%252Fi%252Fpix%252F2011%252F12%252F03%252Farticle-2069512-0F09244F00000578-972_634x526.jpg%3Bhttp%253A%252F%252Fwww.dailymail.co.uk%252Fnews%252Farticle-2069512%252F%3B634%3B526