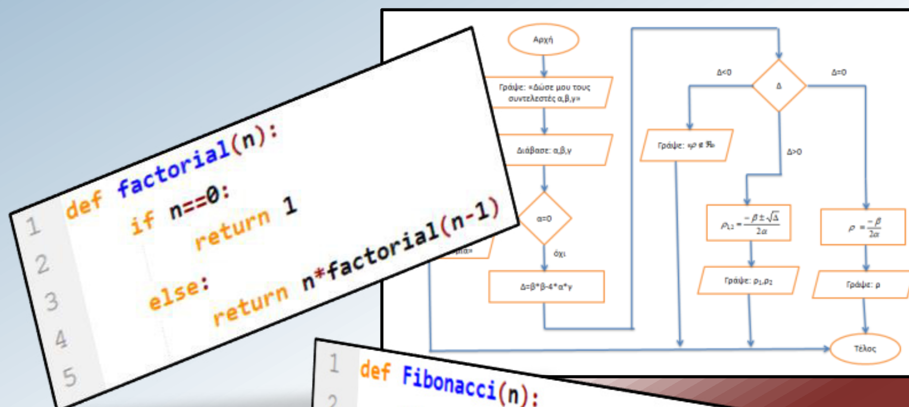
Γεώργιος Μανής
Επίκουρος Καθηγητής
Τμ. Μηχ. Η/Υ & Πληροφορικής
Πανεπιστημίου Ιωαννίνων

Εισαγωγή στον Προγραμματισμό με Αρωγό τη Γλώσσα Python



Ελληνικά Ακαδημαϊκά Ηλεκτρονικά Συγγράμματα και Βοηθήματα
www.kallipos.gr
HEALLINK
Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών
Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης
ΕΣΠΑ 2007-2013
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

ΓΕΩΡΓΙΟΣ ΜΑΝΗΣ
Επίκουρος Καθηγητής Πανεπιστημίου Ιωαννίνων
Τμ. Μηχ. Η/Υ και Πληροφορικής

Εισαγωγή στον Προγραμματισμό με Αρωγό τη Γλώσσα Python

Ελληνικά Ακαδημαϊκά Ηλεκτρονικά
Συγγράμματα και Βοηθήματα
www.kallipos.gr

Εισαγωγή στον Προγραμματισμό με Αρωγό τη Γλώσσα Python

Συγγραφή

Γεώργιος Μανής

Κριτικός αναγνώστης

Κωνσταντίνος Μπλέκας

Συντελεστές έκδοσης

Γλωσσική Επιμέλεια: Ελένη Ψαθά

ISBN: 978-960-603-415-2

Copyright © ΣΕΑΒ, 2015

Το παρόν έργο αδειοδοτείται υπό τους όρους της άδειας Creative Commons Αναφορά Δημιουργού - Μη Εμπορική Χρήση - Παρόμοια Διανομή 3.0. Για να δείτε ένα αντίγραφο της άδειας αυτής επισκεφτείτε τον ιστότοπο https://creativecommons.org/licenses/by-nc-sa/3.0/gr/

ΣΥΝΔΕΣΜΟΣ ΕΛΛΗΝΙΚΩΝ ΑΚΑΔΗΜΑΪΚΩΝ ΒΙΒΛΙΟΘΗΚΩΝ

Εθνικό Μετσόβιο Πολυτεχνείο

Ηρώων Πολυτεχνείου 9, 15780 Ζωγράφου

www.kallipos.gr

.

Ευχαριστίες:

Η συγγραφή ενός βιβλίου είναι μία χρονοβόρα διαδικασία και δύσκολη. Για να γραφεί, θα πρέπει ο συγγραφέας να μπορέσει να στριμώξει μέσα στις άλλες υποχρεώσεις του, χωρίς να τις αμελήσει, τον χρόνο που απαιτείται για να το ολοκληρώσει. Δεν θα το κατάφερνα χωρίς η σύζυγός μου, η **Σύλβια**, να με στηρίξει και να αναλάβει ένα ακόμα μεγαλύτερο φορτίο από τις οικογενειακές μου υποχρεώσεις. Την ευχαριστώ και γι’ αυτό.

Θα ήθελα, επίσης, να ευχαριστήσω τον φίλο και κριτικό αναγνώστη του βιβλίου, **Κωνσταντίνο Μπλέκα**, ο οποίος, με τις εύστοχες παρατηρήσεις του, έκανε το βιβλίο αυτό πιο κατανοητό και περισσότερο πλήρες.

Επίσης, την **Ελένη Ψαθά** που έκανε τη γλωσσική επιμέλεια.

Θα ήταν παράλειψη να μην κάνω μνεία στο έργο **Κάλλιπος**, το οποίο χρηματοδότησε τη συγγραφή.

- - -

Το βιβλίο αυτό το αφιερώνω στα παιδιά μου, τη **Μάνια** και τον **Ανδρέα**, εννέα και επτά χρονών σήμερα, αντίστοιχα.

Περιεχόμενα

1 Εισαγωγή **11**
1.1 Οι γλώσσες προγραμματισμού 11
1.2 Η γλώσσα προγραμματισμού Python 15

2 Τα πρώτα βήματα **19**
2.1 Το πρώτο μας πρόγραμμα 20
2.2 Η δομή ενός προγράμματος 23
2.3 Εκτέλεση σε ένα περιβάλλον εκτέλεσης 25
2.4 Αποσφαλμάτωση κώδικα 29

3 Εισαγωγή στους αλγορίθμους - διαγράμματα ροής **31**
3.1 Διαγράμματα ροής 31
3.2 Τα πρώτα αλγοριθμικά μας βήματα 36
3.2.1 Ρίζες τριωνύμου 36
3.2.2 Μέγιστος τριών αριθμών 39
3.2.3 Παραγοντικό 41

4 Μεταβλητές και εκφράσεις **47**
4.1 Μεταβλητές στις διάφορες γλώσσες προγραμματισμού 47
4.2 Μεταβλητές στην Python 49
4.3 Εκφράσεις 53
4.4 Λογικές εκφράσεις 56
4.5 Είσοδος και έξοδος 57
4.5.1 Η εντολή input 57
4.5.2 Η εντολή print 58

5 Δομές δεδομένων **61**
5.1 Εγγραφές 62
5.2 Πλειάδες 63

5.3 Αλφαριθμητικά 64
5.4 Λίστες 65
5.5 Σύνολα 69
5.6 Ακολουθίες 73
5.7 Λεξικά 76
5.8 Συνοπτικοί πίνακες λειτουργιών στις βασικότερες δομές δεδομένων 79

6 Οι δομές ελέγχου ροής 83

6.1 Η ακολουθία εντολών 83
6.2 Η δομή της απόφασης 85
6.2.1 Η δομή if-elif-else 85
6.2.2 Παράδειγμα με τη δομή if-elif-else: το τριώνυμο 90
6.2.3 Η απόφαση με πολλαπλές επιλογές 92
6.3 Δομή επανάληψης 93
6.3.1 Η δομή for 93
6.3.2 Παραδείγματα με τη δομή for: εσωτερικό γινόμενο, λίστα ακέραιων, προπαίδεια 95
6.3.3 Η δομή while 97
6.3.4 Παράδειγμα με τη δομή while: ρίζες πολυωνύμου με τη μέθοδο της διχοτόμησης 99
6.3.5 Η δομή do while 101

7 Συναρτήσεις 104

7.1 Κάτι σαν κοινός παρονομαστής 104
7.2 Ορισμός και κλήση μιας συνάρτησης στην Python 105
7.3 Επιστροφή αποτελέσματος 107
7.4 Παράμετροι 109
7.5 Παραδείγματα συναρτήσεων 110
7.5.1 Απόλυτη τιμή 110
7.5.2 Παραγοντικό 111
7.5.3 Ύψωση σε δύναμη 112
7.5.4 Λίστα θετικών και αρνητικών αριθμών 113
7.5.5 Λίστα από λίστες 114
7.5.6 Άλλες συναρτήσεις 115
7.6 Πέρασμα παραμέτρων με τιμή και με αναφορά 117
7.7 Συναρτήσεις και διαδικασίες 120
7.8 Ευελιξία περάσματος παραμέτρων στην Python 121

8 Προγραμματίζοντας αλγορίθμους έξυπνα και δημιουργικά 125
8.1 Παράδειγμα 1: πρώτοι αριθμοί 126
8.2 Παράδειγμα 2: δυαδικοί αριθμοί 134

9 Αναδρομή 139
9.1 Αναδρομικές συναρτήσεις και μαθηματικά 139
9.2 Απόδειξη με επαγωγή 140
9.3 Αναδρομικές συναρτήσεις στην Πληροφορική 142
9.4 Το παραγοντικό και η ύψωση σε δύναμη ως αναδρομικές συναρτήσεις 143
9.5 Η ακολουθία των αριθμών Fibonacci 144
9.6 Εκτέλεση μια αναδρομικής συνάρτησης βήμα προς βήμα 147
9.7 Οι επιπτώσεις της αναδρομής στον χρόνο εκτέλεσης 148

10 Αναζήτηση, ταξινόμηση 151
10.1 Αναζήτηση 152
10.1.1 Σειριακή αναζήτηση 152
10.1.2 Δυαδική αναζήτηση 155
10.2 Ταξινόμηση 157
10.2.1 Ταξινόμηση φυσαλίδας 157
10.2.2 Ταξινόμηση με επιλογή 160
10.2.3 Ταξινόμηση δύο ταξινομημένων πινάκων με συγχώνευση 161
10.2.4 Ταξινόμηση με τη χρήση κάδων 163
10.2.5 Γρήγορη ταξινόμηση 164

11 Οι πίνακες ως δομή δεδομένων 167
11.1 Από τις λίστες στους πίνακες 168
11.2 Βασικές πράξεις πινάκων σε μονοδιάστατους πίνακες 168
11.3 Βασικές πράξεις πινάκων σε πολυδιάστατους πίνακες 171
11.4 Άλλες συναρτήσεις πάνω σε πίνακες 173
11.5 Συσχέτιση, συνέλιξη, αυτοσυσχέτιση 176

12 Είσοδος και έξοδος δεδομένων σε αρχεία 184
12.1 Άνοιγμα και κλείσιμο αρχείου 185
12.2 Λειτουργίες εγγραφής 186
12.3 Λειτουργίες ανάγνωσης 188
12.4 Λειτουργίες επανάληψης σε αρχεία 190
12.5 Εγγραφή και ανάγνωση ολόκληρων δομών σε δυαδικά αρχεία . 192

13 Φτιάχνοντας παιχνίδια **195**
13.1 Κρεμάλα 195
13.2 Το Παιχνίδι της Ζωής 200

14 Συμβουλές προς έναν νέο προγραμματιστή **208**

Βιβλιογραφία **213**

Ευρετήριο όρων **215**

Πίνακας Ακρονύμων

GPL	General Public Licence
RAM	Random Access Memory
ROM	Read Only Memory
ΔΡΠ	Διάγραμμα Ροής Προγράμματος

Κεφάλαιο 1: Εισαγωγή

Η **Επιστήμη της Πληροφορικής (Computer Science)** αποτελεί σήμερα μία από τις σημαντικότερες και πλέον εξελισσόμενες επιστήμες. Οι εφαρμογές της έχουν εισχωρήσει σε όλα τα πεδία της ερευνητικής δραστηριότητας, χρησιμοποιούνται καθημερινά από επιχειρήσεις και επαγγελματίες και έχουν διεισδύσει στα σπίτια μας προσφέροντάς μας λύσεις σε καθημερινές μας ανάγκες αλλά και διασκέδαση. Η γνώση βασικών λειτουργιών και η χρήση ηλεκτρονικών υπολογιστών είναι πια απαραίτητες σε κάθε άνθρωπο, ο οποίος δεν θέλει να μείνει πίσω από τις τεχνολογικές εξελίξεις αλλά και επιθυμεί επαγγελματική επιτυχία.

Για τον μέσο χρήστη ο τρόπος με τον οποίο έχει δομηθεί το οικοδόμημα που χρησιμοποιεί - το οποίο ξεκινάει από το υλικό, συνεχίζει στο λειτουργικό σύστημα και γενικότερα στο λογισμικό συστήματος και καταλήγει στο λογισμικό εφαρμογών - δεν έχει και μεγάλη σημασία, αφού αυτός χρησιμοποιεί μόνο την κορυφή του παγόβουνου. Για τον προγραμματιστή, όμως, είναι σημαντικό. Ο προγραμματιστής αλληλεπιδρά με το υλικό, αναπτύσσει το λογισμικό σε κάθε του επίπεδο, το τροποποιεί, το αναβαθμίζει, το συντηρεί. Το σημαντικότερο εργαλείο και μέσο έκφρασης είναι οι **γλώσσες προγραμματισμού (programming languages)**.

1.1 Οι γλώσσες προγραμματισμού

Οι γλώσσες προγραμματισμού αποτελούν το μέσο επικοινωνίας ανάμεσα στον προγραμματιστή εφαρμογών και στον υπολογιστή. Όμως ο τρόπος με τον οποίο σκέφτεται ο άνθρωπος είναι πολύ διαφορετικός και πιο πολύπλοκος από τις εντολές που μπορεί να εκτελέσει ένας ηλεκτρονικός υπολογιστής.

Όσο και εάν αυτό φαίνεται παράξενο, ο ηλεκτρονικός υπολογιστής στηρί-

ζεται σε πολύ απλές ιδέες, και οι πράξεις που μπορεί να κάνει και οι εντολές που έχει τη δυνατότητα να εκτελέσει είναι στοιχειώδεις σε σχέση με την πολυπλοκότητα ενός ζωντανού οργανισμού, ακόμα και όταν μιλάμε για τα τελευταίας τεχνολογίας συστήματα. Οι εντολές αυτές καλούνται **γλώσσα μηχανής (machine language)** και είναι η γλώσσα με την οποία προγραμματίζεται ο ηλεκτρονικός υπολογιστής.

Η γλώσσα, όμως, αυτή αποτελείται από στοιχειώδεις εντολές κάτι που κάνει τον προγραμματισμό σε αυτή δύσκολο, επίπονο και μακριά από τον ανθρώπινο τρόπο σκέψης. Η ανάπτυξη εφαρμογών σε γλώσσα μηχανής θα ήταν υπερβολικά χρονοβόρα και το οικονομικό κόστος ανάπτυξης υπερβολικά υψηλό. Το πρόβλημα αυτό έρχονται να λύσουν οι γλώσσες υψηλού επιπέδου, που είναι οι γνωστές γλώσσες προγραμματισμού.

Οι **γλώσσες προγραμματισμού υψηλού επιπέδου (high level programming languages)** αποτελούν ένα ενδιάμεσο στρώμα ανάμεσα στον προγραμματιστή εφαρμογών και στη γλώσσα μηχανής. Οι μεταβλητές, οι εντολές και οι δομές που χρησιμοποιούν είναι πιο εύχρηστες από τις αντίστοιχες της γλώσσας μηχανής και κάνουν τον προγραμματισμό ευκολότερο, πιο ευχάριστο και αποδοτικό, μειώνοντας ταυτόχρονα το κόστος ανάπτυξης και αποσφαλμάτωσης.

Έχουν προταθεί και χρησιμοποιηθεί κατά καιρούς πολλές γλώσσες προγραμματισμού, όπως οι σχετικά παλαιότερες αλλά όχι τελείως παροπλισμένες FORTRAN (1957), BASIC (1964) και PASCAL (1970). Η γλώσσα C (1969) αποτέλεσε έναν σταθμό στην ιστορία των γλωσσών προγραμματισμού, έγινε ιδιαίτερα δημοφιλής και ποτέ δεν παροπλίστηκε. Εξελίξεις της όπως η C++ (1980) , η C# (2000) και η Java (1995) έχουν κατακτήσει σημαντικό μέρος της αγοράς και εντάσσονται στις πιο δημοφιλείς και περισσότερο χρησιμοποιούμενες στην ανάπτυξη εφαρμογών σήμερα γλώσσες.

Κάθε γλώσσα προγραμματισμού δεν είναι τίποτε άλλο παρά ένα ακόμα πρόγραμμα το οποίο δέχεται σαν είσοδο το δικό μας πρόγραμμα και το μετασχηματίζει σε ένα πρόγραμμα σε γλώσσα μηχανής ώστε να μπορεί να το εκτελέσει ο υπολογιστής. Η επιστήμη των μεταφραστών έχει προχωρήσει τόσο, που τα αποτελέσματα της μεταγλώττισης και η βελτιστοποίηση που επιτυγχάνουν στον τελικά παραγόμενο κώδικα μηχανής είναι εντυπωσιακά.

Επίσης, η ευκολία που παρέχει ο προγραμματισμός σε γλώσσα υψηλού επιπέδου είναι πολύ σημαντική, αφού οι εντολές της γλώσσας μηχανής είναι τόσο στοιχειώδεις σε σχέση με τις δομές μιας γλώσσας υψηλού επιπέδου που θα ήταν άσκοπο ή και αδύνατον από άποψη χρόνου και κόστους να κατασκευαστούν με αυτόν τον τρόπο μεγάλες εφαρμογές. Σε γλώσσα μηχανής γράφονται συνήθως μόνο τμήματα κώδικα που είναι σχετικά μικρά σε μέγεθος

και είναι καίριο να εκτελούνται γρήγορα (π.χ. κάποια τμήματα ενός λειτουργικού συστήματος).

Ο χρόνος και ο τρόπος που επιλέγεται για τον μετασχηματισμό ενός προγράμματος από γλώσσα υψηλού επιπέδου σε γλώσσα μηχανής χωρίζουν τις γλώσσες προγραμματισμού σε δύο κύριες κατηγορίες: τους **διερμηνευτές (interpreters)** και τους **μεταγλωττιστές (compilers)**. Οι διερμηνευτές μετασχηματίζουν σε γλώσσα μηχανής τα προγράμματα μία προς μία εντολή κατά τη διάρκεια της εκτέλεσής τους, ενώ παρέχουν και το περιβάλλον μέσα στα οποίο θα εκτελεστεί το πρόγραμμα που παράχθηκε.

Αυτό σημαίνει ότι ο διερμηνευτής θα μετασχηματίσει την πρώτη εντολή του προγράμματος και θα την εκτελέσει, στη συνέχεια θα κάνει το ίδιο και με τη δεύτερη και με την τρίτη έως ότου τελειώσει το πρόγραμμα. Οι μεταγλωττιστές ακολουθούν διαφορετική διαδικασία. Το πρόγραμμα μεταγλωττίζεται ολόκληρο και αποθηκεύεται στον δίσκο. Στη συνέχεια μπορεί να εκτελεστεί κατευθείαν από τον υπολογιστή.

Καθεμία από τις δύο τεχνικές έχει τα πλεονεκτήματά της. Περισσότερο διαδεδομένοι είναι οι μεταγλωττιστές, αφού έχουν σημαντικά πλεονεκτήματα σε σχέση με τους διερμηνευτές. Το σημαντικότερο από αυτά είναι ότι τα προγράμματα που έχουν μετασχηματιστεί από μεταγλωττιστές τρέχουν γρηγορότερα από τα αντίστοιχα των διερμηνευτών, αφού η μεταγλώττιση δεν γίνεται κατά την εκτέλεση της εφαρμογής ώστε να την καθυστερεί και μάλιστα σημαντικά.

Επίσης, η μεταγλώττιση γίνεται μόνο μία φορά και αποθηκεύεται στον δίσκο και όχι κάθε φορά που θα τρέξει το πρόγραμμα όπως γίνεται με τους διερμηνευτές. Ακόμα, το πρόγραμμα που μετασχηματίζεται από έναν μεταγλωττιστή είναι ανεξάρτητο από αυτόν στο τέλος της μεταγλώττισης, ενώ ο διερμηνευτής πρέπει να παρέχει και το περιβάλλον για την εκτέλεση του προγράμματος.

Ένα πλεονέκτημα των διερμηνευτών είναι ότι υπάρχει μία περισσότερο διαδραστική σχέση ανάμεσα στον προγραμματιστή και στην εκτέλεση, αφού ο προγραμματιστής μπορεί να εκτελέσει εντολές οι οποίες δεν εντάσσονται σε ένα πρόγραμμα, αλλά κατά την εκτέλεση του προγράμματος μπορεί να διακόψει την εφαρμογή, να εκτελέσει πάλι εντολές, να ελέγξει τιμές μεταβλητών και στη συνέχεια να συνεχίσει την εκτέλεση.

Ένας από τους λόγους που η Java έγινε τόσο δημοφιλής είναι ότι στηρίζεται σε διερμηνευτή, αλλά η φιλοσοφία διαφέρει από αυτήν των υπόλοιπων διερμηνευόμενων γλωσσών. Η Java μεταφράζεται σε χρόνο μετάφρασης σε μία γλώσσα η οποία δεν είναι κατανοητή στον υπολογιστή (δεν είναι γλώσσα

μηχανής, αλλά δεν έχει σύνθετες δομές) και λέγεται Java byte code. Για να εκτελεστεί πρέπει σε χρόνο εκτέλεσης να μεταφραστεί σε κώδικα μηχανής με τη βοήθεια της **εικονικής μηχανής της Java (Java Virtual Machine)**, ένα λογισμικό που τρέχει στη μηχανή που τελικά εκτελείται το πρόγραμμα. Πρόκειται για μια ενδιάμεση λύση η οποία έχει το μειονέκτημα της μετάφρασης σε χρόνο εκτέλεσης, αλλά ο κώδικας ο οποίος παράγεται είναι ανεξάρτητος από το υλικό (τον υπολογιστή).

Αντιπροσωπευτικές γλώσσες που χρησιμοποιούν την τεχνική των διερμηνευτών είναι η BASIC, η Java, η Prolog και η Python, ενώ η τεχνική των μεταγλωττιστών ακολουθείται μεταξύ άλλων από τις C, C++, C#, Pascal και FORTRAN.

Ένας άλλος διαχωρισμός των γλωσσών προγραμματισμού μπορεί να γίνει με βάση τη φιλοσοφία που ακολουθείται στον προγραμματισμό. Η BASIC είναι από τις πρώτες γλώσσες που προτάθηκαν και χρησιμοποιούν απλές δομές. Οι γλώσσες αυτές βρίσκονται κοντύτερα στη γλώσσα μηχανής από ό,τι οι γλώσσες που βρίσκονται σε άλλες κατηγορίες, οι εντολές τους δηλαδή αποσυντίθενται σχετικά εύκολα σε εντολές γλώσσας μηχανής. Στη συνέχεια εμφανίστηκε **ο δομημένος προγραμματισμός (structured programming)** ή **διαδικασιακός προγραμματισμός (structured programming)** με κύριους εκπροσώπους τη C και την Pascal. Στον δομημένο προγραμματισμό τόσο η διάρθρωση του προγράμματος όσο και οι δομές των δεδομένων είναι πιο πολύπλοκες και πλησιάζουν περισσότερο στον τρόπο σκέψης και έκφρασης του ανθρώπου, κάτι που έκανε τον τρόπο αυτόν προγραμματισμού να επικρατήσει.

Κάποια στιγμή εμφανίστηκαν οι γλώσσες που **βασίζονται σε αντικείμενα (object based)**. Γρήγορα αντικαταστάθηκαν από τις **αντικειμενοστραφείς γλώσσες (object oriented)**. Τα **αντικείμενα (objects)** είναι δομές που προσομοιάζουν τα αντικείμενα του πραγματικού κόσμου. Για παράδειγμα, σε ένα παιχνίδι με τράπουλα θα παρασταθούν με αντικείμενα τα τραπουλόχαρτα και οι παίκτες (τουλάχιστον). Τα αντικείμενα αποτελούνται από **δεδομένα** (π.χ. ένα αντικείμενο - φύλλο μπορεί να έχει για δεδομένα το είδος του και τον αριθμό του) και **μεθόδους** για να προσπελάζονται αυτά τα δεδομένα (π.χ. ένας παίκτης μπορεί να έχει τη μέθοδο **μέτρησε τα σπαθιά που έχεις**). Κάθε αντικείμενο ανήκει σε μία **κλάση (class)**, είναι κάτι σαν ο τύπος δεδομένων του αντικειμένου, μια περιγραφή του, αν προτιμάτε. Το βασικότερο χαρακτηριστικό των αντικειμενοστραφών γλωσσών προγραμματισμού είναι η **κληρονομικότητα (inheritance)**. Η κληρονομικότητα είναι η δυνατότητα να φτιάχνεις κλάσεις οι οποίες κληρονομούν χαρακτηριστικά από άλλες κλάσεις. Έτσι, μπορείς να φτιάξεις μία νέα κλάση επεκτείνοντας μία άλλη, χωρίς φυσικά να καταστρέψεις την αρ-

χική κλάση. Αυτό έχει πολλά πλεονεκτήματά και συνέβαλε καθοριστικά στην καθιέρωση των αντικειμενοστραφών γλωσσών προγραμματισμού.

Να αναφέρουμε, τέλος, για πληρότητα ότι υπάρχουν οι **γλώσσες παράλληλου προγραμματισμού (languages for parallel programming)** που χρησιμοποιούνται σε **παράλληλες αρχιτεκτονικές (parallel architectures)**. Αυτές είναι συνήθως προεκτάσεις γλωσσών που ήδη υπάρχουν ώστε να υποστηρίξουν τις λειτουργίες του παράλληλου προγραμματισμού (π.χ. Parallel C, Parallel FORTRAN) ή άλλες γλώσσες σχεδιασμένες γι' αυτόν τον σκοπό. Επίσης υπάρχουν γλώσσες **προσανατολισμένες σε προγραμματισμό στο διαδίκτυο (languages for web development)**, όπως η html, η javascript και η php.

Στο πλαίσιο του βιβλίου αυτού κάνουμε τα πρώτα βήματα στον προγραμματισμό. Μάλιστα, από τις διαφορετικές φιλοσοφίες στον τρόπο σχεδίασης και υλοποίησης των εφαρμογών θα επιλέξουμε τον δομημένο προγραμματισμό ως καταλληλότερο για ένα εισαγωγικό μάθημα. Η φιλοσοφία της BASIC είναι ίσως ευκολότερη και θα έκανε κανείς τη σκέψη ότι τα πρώτα βήματα ενός προγραμματιστή θα έπρεπε να ακολουθήσουν τον πιο βατό δρόμο.

Όμως, ακόμα κι εάν εξαιρέσουμε το γεγονός ότι πρακτικά μόνο ο δομημένος και ο αντικειμενοστραφής προγραμματισμός υφίστανται σήμερα, έχουν εκφραστεί σοβαρές επιφυλάξεις για το πόσο ορθά οδηγεί τη σκέψη ενός μαθητευόμενου στον χώρο της Πληροφορικής η φιλοσοφία της BASIC. Άλλωστε, ακόμα και οι νεότερες εκδόσεις των γλωσσών αυτών έχουν στραφεί προς τον δομημένο και τον αντικειμενοστραφή προγραμματισμό.

1.2 Η γλώσσα προγραμματισμού Python

Ο σκοπός του βιβλίου δεν είναι να μάθουμε κάποια συγκεκριμένη γλώσσα προγραμματισμού, αλλά να κάνουμε μία εισαγωγή στη φιλοσοφία και στην τεχνολογία του προγραμματισμού. Για τον σκοπό αυτόν χρειαζόμαστε μια απλή γλώσσα, η οποία όμως να μην είναι παροπλισμένη ή να μην έχει αμφισβητηθεί. Η Python είναι μία γλώσσα που μπορεί να χρησιμοποιηθεί για τον σκοπό αυτόν, είναι απλή, ή τουλάχιστον δεν είναι απαραίτητο να εμβαθύνει κανείς σε μεγάλο βαθμό για να αναπτύξει εφαρμογές. Είναι πολύ δημοφιλής γλώσσα σήμερα, χρησιμοποιείται στην ανάπτυξη εμπορικών εφαρμογών και γίνεται κάθε μέρα όλο και περισσότερο δημοφιλής.

Κύριος στόχος της είναι η αναγνωσιμότητα του κώδικα και η ευκολία χρήσης της. Διακρίνεται για την ευκολία και την ταχύτητα εκμάθησής της. Έχει πολλές βιβλιοθήκες που διευκολύνουν ιδιαίτερα αρκετές συνηθισμένες εργασίες. Αρχικά, η Python ήταν γλώσσα σεναρίων που χρησιμοποιούνταν στο λει-

τουργικό σύστημα **Amoeba**, ικανή και για κλήσεις συστήματος. Η Python 2.0 κυκλοφόρησε το 2000. Το 2008 κυκλοφόρησε η έκδοση 3.0. Πολλά από τα καινούργια χαρακτηριστικά αυτής της έκδοσης έχουν μεταφερθεί στις εκδόσεις 2.6 και 2.7, που είναι προς τα πίσω συμβατές. Η Python 3 είναι ιστορικά η πρώτη γλώσσα προγραμματισμού που σπάει την προς τα πίσω συμβατότητα με προηγούμενες εκδόσεις.

Είναι σημαντικό να αναφέρουμε ότι η Python είναι **γλώσσα ανοικτού κώδικα (open source)**, ο κώδικάς της δηλαδή είναι διαθέσιμος σε όλους και χωρίς κόστος. Το λογισμικό ανοικτού κώδικα μπορεί ο καθένας ελεύθερα να το χρησιμοποιεί, να το αντιγράφει, να το διανέμει και να το τροποποιεί ανάλογα με τις ανάγκες του. Σήμερα λειτουργεί ένα παγκόσμιο ανοικτό δίκτυο προγραμματιστών, οι οποίοι παράλληλα αναπτύσσουν και διορθώνουν τον κώδικα των προγραμμάτων, κυκλοφορώντας ταχύτατα νέες βελτιωμένες εκδόσεις λογισμικού. Το διαδίκτυο αποτελεί το βασικό τρόπο πρόσβασης στο διαθέσιμο ελεύθερο λογισμικό. Η εξάπλωση του λογισμικού ανοικτού κώδικα έχει στηριχθεί στην ευρεία χρήση του διαδικτύου και η διαδικασία ανάπτυξης και λειτουργίας του διαδικτύου βασίζεται, κατά κύριο λόγο, σε λογισμικό ανοικτού κώδικα. Το λογισμικό ανοιχτού κώδικα κερδίζει διαρκώς νέους φίλους παγκοσμίως στην εκπαίδευση, στη δημόσια διοίκηση, στις επιχειρήσεις, εμφανίζονται νέα εργαλεία, αξιόπιστα, σταθερά στη λειτουργία τους, απαλλαγμένα από τα σημαντικά κόστη απόκτησης και συνεχούς αναβάθμισης που απαιτεί το εμπορικό λογισμικό. Όλο και πιο πολλοί πόροι διατίθενται στην τεχνική υποστήριξη με σημαντικά οφέλη για την τοπική και εθνική οικονομία.

Ας γυρίσουμε όμως στην Python. Απόρροια των πλεονεκτημάτων του ανοικτού λογισμικού είναι και ο μεγάλος αριθμός βιβλιοθηκών για Python, ελεύθερων στο διαδίκτυο και φυσικά ανοικτού κώδικα. Οι διαθέσιμες βιβλιοθήκες καλύπτουν ένα πολύ μεγάλο εύρος αναγκών. Η Python έχει χρησιμοποιηθεί ευρέως για την ανάπτυξη εφαρμογών ανοικτού λογισμικού και είναι μία από τις γλώσσες που χρησιμοποιούνται περισσότερο από την κοινότητα ανοικτού λογισμικού. Επιπλέον, στηρίζει αλλά και στηρίζεται από την ανάπτυξη των τεχνολογιών του διαδικτύου νέας γενιάς.

Η διαχείρισή της γίνεται από τον μη κερδοσκοπικό οργανισμό **Python Software Foundation**. Ο κώδικας διανέμεται με την άδεια **Python Software Foundation License** η οποία είναι συμβατή με την **GPL (General Public Licence)**. Κατά την GPL οι χρήστες μπορούν να τρέξουν ένα πρόγραμμα για οποιονδήποτε λόγο, να μελετήσουν τη λειτουργία ενός προγράμματος και να το τροποποιήσουν, να διανείμουν αντίγραφα του προγράμματος έτσι ώστε να βοηθήσουν τον πλησίον, να βελτιώσουν το πρόγραμμα και να προσφέρουν τις βελτιώσεις

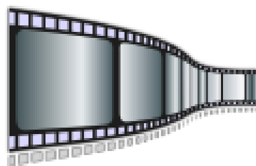
στο κοινό έτσι ώστε να ωφεληθεί ολόκληρη η κοινότητα.

Στο βιβλίο αυτό επιλέξαμε να χρησιμοποιήσουμε τη νεότερη δυνατή έκδοση, την έκδοση 3. Μοιάζει να είναι μία επιλογή μάλλον προφανής, αλλά δεν είναι έτσι. Ενώ η Python 3 έχει κυκλοφορήσει από το 2008, μέχρι σήμερα δεν έχει αντικαταστήσει την Python 2. Ο λόγος δεν είναι ότι η Python 3 δεν είναι καλύτερη από την Python 2. Είναι! Όμως υπάρχει ένας μεγάλος όγκος από βιβλιοθήκες και λογισμικό ήδη διαθέσιμος για την Python 2, ο οποίος δεν έχει μεταφερθεί στην Python 3. Έτσι, η Python 2 παραμένει ζωντανή και ίσως περισσότερο χρησιμοποιούμενη από την Python 3. Εμείς θα χρησιμοποιήσουμε τη νεότερη έκδοση, όμως το καλό είναι ότι οι διαφορές τους είναι πολύ λίγες όσον αφορά αυτά που εμείς θα χρησιμοποιήσουμε. Το να προγραμματίσετε σε Python 2, ενώ έχετε μάθει Python 3, είναι εύκολο.

Το βιβλίο αυτό είναι ένα ηλεκτρονικό βιβλίο. Ακολουθώντας την εξέλιξη της τεχνολογίας και μάλιστα, ας παινευτούμε λίγο γι' αυτό ως άνθρωποι στον χώρο της Πληροφορικής!, της σύγχρονης τάσης για τη μετάδοση και εμπέδωση της γνώσης, σας δίνει τη δυνατότητα, όχι μόνο να το διαβάσετε στην οθόνη του υπολογιστή σας, του tablet σας ή του κινητού σας, αλλά χρησιμοποιώντας ήχο και εικόνα, διαδραστικότητα και τεχνολογία διαδικτύου κάνει την προσπάθειά σας περισσότερο ευχάριστη και την απόκτηση γνώσεων ευκολότερη και πιο αποδοτική. Θα βρείτε μέσα σε αυτό έγχρωμες εικόνες, συνδέσμους που θα σας παραπέμψουν στο διαδίκτυο κάνοντας μόνο ένα κλικ επάνω τους (εδώ πρέπει να έχετε σύνδεση στο διαδίκτυο για να μπορέσει να γίνει αυτό), παρουσιάσεις με εικόνα και ήχο που θα μπορέσετε να παρακολουθήσετε όσες φορές θέλετε και με όποια ταχύτητα εσείς επιθυμείτε.

Ελπίζω να γίνει πράγματι ένας αρωγός στην προσπάθειά σας, όπως υπόσχεται και ο τίτλος του.

Αν θέλετε να δείτε ένα οπτικό καλωσόρισμα στο βιβλίο, κάντε κλικ στην Ταινία 1.1, που ακολουθεί.



Ταινία 1.1: Οπτικό καλοσώρισμα.
`http://repfiles.kallipos.gr/file/3153`

Περισσότερο υλικό σχετικό με αυτό το κεφάλαιο μπορείτε να διαβάσετε στο πρώτο κεφάλαιο των βιβλίων [1][2][3].

Βιβλιογραφία

1. Brian Heinold (2012). **Introduction to Programming Using Python.** Publisher: Mount St. Mary's University, Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.

2. Ellis Horowitz (1993). **Βασικές Αρχές Γλωσσών Προγραμματισμού.** 2η έκδοση, Εκδόσεις Κλειδάριθμος.

3. Αχιλλέας Καμέας (2000). **Τεχνικές Προγραμματισμού.** Τόμος Β. ΠΛΗ-10, Ελληνικό Ανοικτό Πανεπιστήμιο.

Κεφάλαιο 2:
Τα πρώτα βήματα

Και τώρα ας κάνουμε το πρώτο μας βήμα. Για εσάς, που νιώθετε ότι αυτό είναι πραγματικά το πρώτο βήμα στον προγραμματισμό, ίσως να είναι και το πιο σημαντικό. Θα δούμε πώς σκεφτόμαστε για να φτιάξουμε το πιο απλό πρόγραμμα και θα το δούμε να εκτελείται. Αν σας φαίνεται υπερβολικό το ότι αυτό το βήμα είναι πολύ σημαντικό, τότε, όταν θα φτάσετε στο τέλος του βιβλίου, αναρωτηθείτε ποιο πράγματι ήταν το σημαντικότερο από όλα τα βήματα που κάνατε.

Ο προγραμματισμός δεν είναι απλό πράγμα. Είναι πολύ απλό! Και ευχάριστο. Όταν ξεπεράσετε τις πρώτες δυσκολίες και καταλάβετε πώς σκεφτόμαστε και με ποιον τρόπο από ένα σύνολο από στοιχειώδεις λειτουργίες φτάνουμε σε ένα οικοδόμημα που λειτουργεί στον υπολογιστή και είναι χρήσιμο και λειτουργικό, θα πιάσετε τον εαυτό σας να περνάει πολλές ώρες γράφοντας κώδικα, χωρίς να καταλαβαίνει πότε πέρασε η ώρα ή πότε ξημέρωσε ...

Έτσι λοιπόν, πρώτα απ' όλα πρέπει να σκεφτούμε τι είναι αυτό που θέλουμε να φτιάξουμε. Πρέπει να το σκεφτούμε καλά, να μη μας ξεφύγει τίποτα, κάθε ανάγκη μας πρέπει από τώρα να αποτυπωθεί στο χαρτί, πριν αρχίσουμε τη διαδικασία ανάπτυξης του κώδικα. Αφού σιγουρευτούμε ότι έχουμε καλά στο μυαλό μας τι ακριβώς θέλουμε να φτιάξουμε, σε όλη του τη λεπτομέρεια, μπορούμε να προχωρήσουμε στο επόμενο βήμα. Εκεί πρέπει από ένα σύνολο δεδομένων λειτουργιών να επιλέξουμε τις κατάλληλες και να τις συνδυάσουμε, ώστε να κάνουν αυτό που θέλουμε.

Θυμίζει λίγο τη μαγειρική, όπου ανακατεύουμε υλικά, για να φτιάξουμε κάτι πολύ νόστιμο, αλλά περισσότερο θυμίζει τις κατασκευές που κάνουν τα μικρά παιδιά με τα τουβλάκια. Σκέφτονται τι θα φτιάξουν και σιγά σιγά το κατασκευάζουν με μικρά δομικά στοιχεία. Ακόμα περισσότερο μου θυμίζει τη διαδικασία με την οποία θα μάθω σε ένα ρομποτάκι να κινείται. Πρώτα θα αποφα-

σίσω πού θέλω να πάει, μετά θα σχεδιάσω τη διαδρομή που θα κάνει και μετά θα αρχίσω και θα του λέω:

> **Κούνησε το δεξί σου πόδι κατά ένα βήμα,**
> **κούνησε το αριστερό σου πόδι κατά μισό βήμα,**
> **στρίψε 90ο δεξιά, κ.ο.κ.**

Μόλις σχεδιάσουμε τη διαδρομή με τέτοια λεπτομέρεια, έχουμε τελειώσει και μπορούμε να διασκεδάσουμε βλέποντας το ρομποτάκι να φτάνει στο στόχο του.

Παρακάτω στο κεφάλαιο αυτό θα κάνουμε δύο πράγματα: Πρώτα θα σχεδιάσουμε το πρώτο μας πρόγραμμα. Όχι, δεν θα είναι το **hello world**, έκπληξη! Μετά θα τρέξουμε το πρόγραμμά μας σε ένα περιβάλλον Python.

2.1 Το πρώτο μας πρόγραμμα

Ας ξεκινήσουμε με βάση τις ελάχιστες γνώσεις που έχουμε αυτήν τη στιγμή και ας κάνουμε μία προσπάθεια να λύσουμε ένα απλό πρόβλημα: ας υποθέσουμε ότι θέλουμε να κατασκευάσουμε ένα πρόγραμμα το οποίο να κάνει πρόσθεση δύο αριθμών. Θα φανταζόμασταν ότι, όταν εκτελείται το πρόγραμμα, θα εμφανίζεται στην οθόνη κάτι τέτοιο:

> **Δώσε μου τον πρώτο αριθμό: 5**
> **Δώσε μου τον δεύτερο αριθμό: 8**
> **Το άθροισμα είναι: 13**

Τους αριθμούς 5 και 8 τους εισάγει ο χρήστης του προγράμματος από το πληκτρολόγιο. Με τον όρο **χρήστης (user)** εννοούμε αυτόν που τελικά χρησιμοποιεί το πρόγραμμα, π.χ. τον πωλητή σε ένα μαγαζί, τον υπάλληλο σε μία επιχείρηση που νοικιάζει CD, το παιδί που παίζει παιχνίδια. Ο προγραμματιστής πρέπει να αναλύσει το παραπάνω πρόβλημα, να βρει μία λύση γι' αυτό, να εξετάσει διαφορετικές λύσεις που ενδεχομένως είναι καλύτερες από αυτήν που βρήκε, και στη συνέχεια, αφού βεβαιωθεί ότι η λύση του είναι ορθή και η καλύτερη δυνατή, να αναπτύξει τον κώδικα. Τέλος, πρέπει να ελέγξει ότι το πρόγραμμά του δεν έχει σφάλματα, που πηγάζουν από την κωδικοποίηση αλλά και από τη σχεδίαση, τα οποία δεν μπόρεσε να προβλέψει νωρίτερα, αλλά και να διορθώσει ό,τι πιθανά λάθη υπάρχουν.

Η τυποποίηση της διαδικασίας που ακολουθείται για την ανάπτυξη ενός προγράμματος αποτελεί έναν μεγάλο κλάδο της Πληροφορικής ο οποίος ονομάζεται **Τεχνολογία Λογισμικού (Software Engineering)**. Εμείς, στο πλαίσιο ενός

εισαγωγικού βιβλίου, δεν θα ασχοληθούμε παρά μόνο πολύ επιφανειακά με θέματα Τεχνολογίας Λογισμικού.

Ο προγραμματιστής, λοιπόν, θα σκεφτεί κάπως έτσι για να λύσει το πρόβλημα με την πρόσθεση:

> **Αρχικά πρέπει να ζητήσω από τον χρήστη να μου δώσει τους δύο αριθμούς.**
> **Στη συνέχεια πρέπει να τους προσθέσω, για να βρω το αποτέλεσμα.**
> **Τέλος, πρέπει να τυπώσω το αποτέλεσμα.**

Τώρα ας αναλύσουμε περισσότερο το πρόβλημα:

> **Ζητάω τον πρώτο αριθμό.**
> **Ζητάω τον δεύτερο αριθμό.**
> **Προσθέτω τους δύο αριθμούς.**
> **Τυπώνω το αποτέλεσμα.**

Στη συνέχεια, ας προσθέσουμε τα μηνύματα που θέλουμε να εμφανίσουμε στην οθόνη:

> **Τύπωσε ”Δώσε μου τον πρώτο αριθμό”.**
> **Ζήτησε τον πρώτο αριθμό.**
> **Τύπωσε ”Δώσε μου τον δεύτερο αριθμό:”.**
> **Ζήτησε τον δεύτερο αριθμό.**
> **Πρόσθεσε τους δύο αριθμούς.**
> **Τύπωσε: ”Το άθροισμα είναι:”.**
> **Τύπωσε το άθροισμα των δύο αριθμών.**

Ο αλγόριθμος (μα τι είναι αυτό;) είναι σχεδόν έτοιμος. Του λείπει όμως κάτι σημαντικό ακόμα. Θα μιλήσουμε λίγο τώρα για την έννοια της **μεταβλητής (variable)** και με την ευκαιρία και για τις **σταθερές (constants)**. Οι σταθερές είναι αριθμοί οι οποίοι δεν χρειάζεται να αλλάξουν κατά την εκτέλεση ενός προγράμματος. Σε ένα πρόβλημα γεωμετρίας είναι πολύ πιο εύχρηστο να ονομάσουμε **pi** τον αριθμό 3.141592... και να χρησιμοποιούμε αυτό κάθε φορά που θέλουμε να αναφερθούμε στη σταθερά αυτή. Αντίθετα, οι μεταβλητές μπορούν να αλλάξουν κατά τη διάρκεια της εκτέλεσης ενός προγράμματος. Ίσως η λέξη **μεταβλητή** να μην είναι η καταλληλότερη για να περιγράψει αυτό που ονομάζουμε στον προγραμματισμό **μεταβλητή**. Ουσιαστικά, πρόκειται για κάποιον χώρο τον οποίο δεσμεύουμε για να αποθηκεύουμε πληροφορίες.

Όπως σε ένα μουσικό επανεγγράψιμο CD ή άλλο σύγχρονο αποθηκευτικό μέσο μπορούμε να γράψουμε ένα τραγούδι, να το ακούσουμε όσες φορές θέλουμε, να το κρατήσουμε όσο καιρό θέλουμε και στο τέλος να γράψουμε κάποιο άλλο τραγούδι από πάνω σβήνοντας έτσι το παλιό. Σε μια μεταβλητή δεν γράφουμε τραγούδια (ή τουλάχιστον όχι μόνο τραγούδια, αν θέλουμε να είμαστε ακριβείς). Στο πρόγραμμά μας διακρίνουμε τρεις μεταβλητές: τους δύο αριθμούς που θα προστεθούν και το αποτέλεσμα.

Ας ονομάσουμε τον πρώτο αριθμό **ΑΡΙΘΜΟ1** τον δεύτερο **ΑΡΙΘΜΟ2** και το αποτέλεσμα **ΑΠΟΤΕΛΕΣΜΑ**. Γενικά, μπορούμε να δώσουμε ό,τι ονόματα θέλουμε στις μεταβλητές μας, αρκεί να μην αποτελούν λέξεις που έχουν δεσμευτεί από τη γλώσσα προγραμματισμού που χρησιμοποιούμε. Για παράδειγμα, δεν μπορούμε να ονομάσουμε μία μεταβλητή **for** στη C ή στην Python, αφού υπάρχει εντολή μ' αυτό το όνομα και, φυσικά, κάτι τέτοιο θα προκαλούσε σύγχυση. Θα μπορούσαμε επίσης να ονομάσουμε **A** τη μεταβλητή **ΑΡΙΘΜΟΣ1**, **B** τη μεταβλητή **ΑΡΙΘΜΟΣ2** και **X** τη μεταβλητή **ΑΠΟΤΕΛΕΣΜΑ**. Κάτι τέτοιο όμως δεν θα ήταν και η καλύτερη επιλογή. Στις μεταβλητές μας πρέπει να δίνουμε ονόματα περιγραφικά, τα οποία όσο το δυνατόν εκφράζουν καλύτερα τον ρόλο της μεταβλητής στο πρόγραμμά μας.

Είναι σίγουρο ότι βλέποντας ένα (δικό μας) πρόγραμμα πολλών χιλιάδων γραμμών, δύο χρόνια αφότου το ολοκληρώσαμε, η λέξη **ΑΠΟΤΕΛΕΣΜΑ** θα μας θυμίσει πολύ περισσότερα από ό,τι ένα απλό **X** και θα μας διευκολύνει πολύ περισσότερο να καταλάβουμε τι είχαμε στο μυαλό μας όταν κατασκευάζαμε το πρόγραμμα. Οι μεταβλητές πρέπει να μας βοηθούν να κατανοήσουμε τον κώδικα και όχι ο κώδικας να μας βοηθάει να καταλάβουμε τις μεταβλητές.

Το ίδιο διευκολύνεται και ένας τρίτος ο οποίος προσπαθεί να κατανοήσει τον κώδικά μας (ο δύστυχος). Ας προσθέσουμε τώρα στο πρόγραμμά μας και τις μεταβλητές. Θα γίνει κάπως έτσι:

```
Τύπωσε ”Δώσε μου τον πρώτο αριθμό:”
Ζήτησε τον ΑΡΙΘΜΟ1
Τύπωσε ”Δώσε μου τον δεύτερο αριθμό:”
Ζήτησε τον ΑΡΙΘΜΟ2
ΑΠΟΤΕΛΕΣΜΑ = ΑΡΙΘΜΟΣ1 + ΑΡΙΘΜΟΣ2
Τύπωσε ”Το άθροισμα είναι:”
Τύπωσε ΑΠΟΤΕΛΕΣΜΑ
```

Τώρα βρισκόμαστε πολύ κοντά σε κάτι που είναι σε θέση να καταλάβει ένας υπολογιστής. Πρακτικά έχουμε ήδη τελειώσει. Αν φτάσουμε στο σημείο αυτό, η κωδικοποίηση του προγράμματος σε μια γλώσσα υψηλού επιπέδου είναι μία

απλή διαδικασία. Πρέπει απλά να χρησιμοποιήσουμε τις κατάλληλες εντολές που θα υλοποιήσουν το παραπάνω σχέδιο κώδικα.

Η Python διαθέτει την εντολή **input** για είσοδο δεδομένων, την εντολή **print** για έξοδο δεδομένων (εμφάνιση στην οθόνη) και, φυσικά, υποστηρίζει αριθμητικές πράξεις μεταξύ μεταβλητών. Αν και είναι νωρίς ακόμα, νομίζω ότι δεν θα δυσκολευτείτε να καταλάβετε τη λειτουργία του κώδικα στο Σχήμα 2.1.

```
1  number1 = int(input('Δώσε μου τον πρώτο αριθμό: '))
2  number2 = int(input('Δώσε μου τον δεύτερο αριθμό: '))
3  result = number1 + number2
4  print ('Το αποτέλεσμα είναι: ',result)
```

Σχήμα 2.1: Πρόσθεση δύο αριθμών.

Στη γραμμή 1 ζητάμε την τιμή για τη μεταβλητή **number1**. Αν και είναι πολύ νωρίς για να πούμε λεπτομέρειες, αυτό γίνεται με την εντολή **input**, η οποία διαβάζει κάτι από το πληκτρολόγιο και στη συνέχεια με την **int** το μετατρέπει σε ακέραιο. Όμοια, στη γραμμή 2 δίνουμε τον δεύτερο αριθμό στη μεταβλητή **number2**. Στη συνέχεια προσθέτουμε τις **number1** και **number2** και το αποτέλεσμα το βάζουμε στη μεταβλητή **result**. Στη γραμμή 4 τυπώνουμε το αποτέλεσμα.

Παρατηρήστε ότι το **Το αποτέλεσμα είναι:** το οποίο υπάρχει ανάμεσα σε δύο εισαγωγικά θα τυπωθεί ως έχει στην οθόνη. Όταν όμως φτάσουμε στη μεταβλητή **result**, θα εμφανιστεί στην οθόνη η τιμή της μεταβλητής (το άθροισμα των δύο τιμών που θα δώσουμε εμείς και όχι η λέξη **result**). Αλλά νομίζω ότι μπήκαμε σε αρκετή λεπτομέρεια και όλα αυτά θα τα δούμε αργότερα. Ας σταματήσουμε, λοιπόν, εδώ και ας πανηγυρίσουμε, διότι μόλις ολοκληρώσαμε το πρώτο μας πρόγραμμα.

2.2 Η δομή ενός προγράμματος

Αφού καταφέραμε και υλοποιήσαμε το πρώτο μας πρόγραμμα, ας δούμε ένα πρόγραμμα στη γενικότερη του μορφή. Σε κάθε γλώσσα προγραμματισμού και σε κάθε διαφορετικό προγραμματιστικό μοντέλο η μορφή που έχει ένα πρόγραμμα είναι διαφορετική. Εμείς θα επιχειρήσουμε να μην μιλήσουμε για κάποια γλώσσα συγκεκριμένα, αλλά ούτε και να γενικεύσουμε τόσο ώστε να γίνει ενότητα αυτή κουραστική. Θα μιλήσουμε βέβαια για διαδικασιακό προγραμματισμό, όχι δηλαδή για γλώσσες με αντικείμενα.

Συνήθως, στην αρχή ενός προγράμματος δηλώνουμε τις βιβλιοθήκες που θα χρησιμοποιηθούν. Οι **βιβλιοθήκες (libaries)** είναι έτοιμα υποπρογράμματα τα οποία αν τα δηλώσουμε μπορούμε να τα καλέσουμε και να τα χρησιμοποιήσουμε μέσα στο δικό μας πρόγραμμα. Για παράδειγμα, η γλώσσα C δεν υποστηρίζει την τετραγωνική ρίζα, αλλά αν γράψουμε στην αρχή του προγράμματος:

```
#include<math.h>
```

Τότε, αν κληθεί η συνάρτηση **SQRT** όπως φαίνεται παρακάτω, θα επιστρέψει στο x την τετραγωνική ρίζα του **y**:

```
x=SQRT(y)
```

Στη συνέχεια, δηλώνουμε τις μεταβλητές που θα χρησιμοποιήσουμε και τι τύπος είναι αυτές. Αυτό χρειάζεται κυρίως στους μεταγλωττιστές, ενώ οι διερμηνευτές δεν το απαιτούν. Έτσι, στην Python δεν χρειάζεται να ορίσουμε μεταβλητές, ενώ αντίθετα στη C αυτό είναι απαραίτητο. Η **y** στο παραπάνω παράδειγμα, αφού το παράδειγμα είναι σε C, πρέπει να έχει δηλωθεί ως ακέραια ή πραγματική μεταβλητή, ενώ η **x** ως πραγματική. Θα γράφαμε κάτι τέτοιο:

```
int y;
double y;
```

Στη συνέχεια, δηλώνουμε τα δικά μας υποπρογράμματα. Ας υποθέσουμε ότι η **SQRT** δεν υπήρχε στη C, αλλά εμείς τη χρειαζόμασταν. Δεν θα είχαμε άλλη επιλογή από το να τη φτιάξουμε μόνοι μας. Στην Python χρησιμοποιούμε την **def** για τον σκοπό αυτόν. Λεπτομέρειες θα δούμε σε αντίστοιχο κεφάλαιο παρακάτω στο βιβλίο, εδώ απλά ας δούμε τη γενική μορφή μίας συνάρτησης, ας την ονομάσουμε **my_sqrt**:

```
def my_sqrt(y):
...
```

Ορίσαμε ότι η συνάρτησή μας θα λέγεται **my_sqrt** και ότι θα παίρνει μία παράμετρο ως είσοδο, την **y**. Εκεί που βρίσκονται οι τρεις τελείες θα τοποθετηθεί ο κυρίως κώδικας της συνάρτησης που θα υπολογίζει και θα επιστρέφει το αποτέλεσμα.

Τέλος, ακολουθεί το κυρίως πρόγραμμα. Το κυρίως πρόγραμμα δεν έχει μία συγκεκριμένη δομή, αλλά εξαρτάται από την κάθε εφαρμογή. Μία συνήθης δομή είναι αυτή που ακολουθούμε στο παράδειγμα της ενότητας 2.1, όπου τοποθετήσαμε στην αρχή την είσοδο των δεδομένων, μετά την επεξεργασία τους και στο τέλος την εμφάνιση των αποτελεσμάτων στην οθόνη.

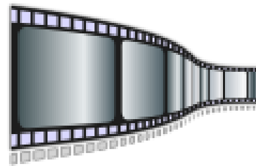
2.3 Εκτέλεση σε ένα περιβάλλον εκτέλεσης

Και τώρα είμαστε έτοιμοι να το τρέξουμε. Για να γίνει όμως αυτό, χρειαζόμαστε έναν διερμηνευτή της Python. Υπάρχουν πολλοί διερμηνευτές, ελεύθεροι στο διαδίκτυο, ανάλογα με το λειτουργικό σύστημα στο οποίο εργάζεστε και τις προτιμήσεις σας. Μπορείτε να κατεβάσετε όποιον θέλετε και να τον εγκαταστήσετε. Επίσης πρέπει να επιλέξετε και την έκδοση της Python που επιθυμείτε. Δείτε σχετικά με τις εκδόσεις της Python στο προηγούμενο κεφάλαιο.

Εδώ, και σε όλο το υπόλοιπο βιβλίο, θα χρησιμοποιήσουμε την τελευταία έκδοση της Python, η οποία είναι διαθέσιμη την ώρα που γράφεται αυτό το βιβλίο για περιβάλλον Microsoft Windows. Πρόκειται για την έκδοση 3.4, την οποία μπορείτε να κατεβάσετε από τη διεύθυνση:

`http://www.python.org`

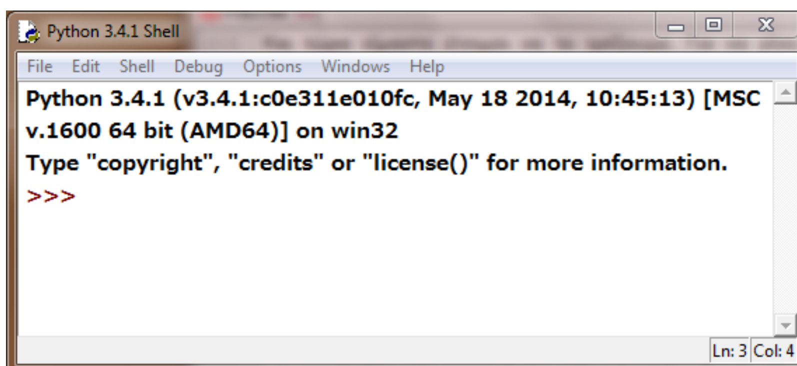
Εκεί θα βρείτε ένα ολόκληρο περιβάλλον, το **IDLE**, το οποίο μπορείτε να χρησιμοποιήσετε για να γράψετε, να αποσφαλματώσετε και να εκτελέσετε το πρόγραμμά σας. Αν θέλετε να το εγκαταστήσουμε μαζί, κάντε κλικ στο εικονίδιο που ακολουθεί και δείτε την Ταινία 2.1.



Ταινία 2.1: Εγκατάσταση του IDLE.

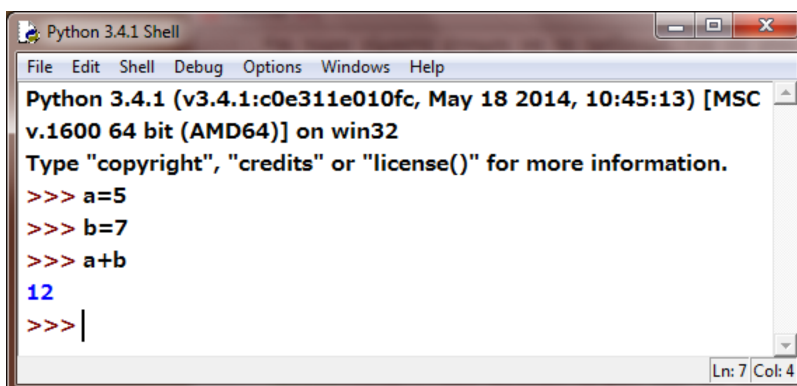
`http://repfiles.kallipos.gr/file/9280`

Η παρουσίαση του περιβάλλοντος ανάπτυξης δεν είναι μέσα στους σκοπούς αυτού του βιβλίου. Θα σας δείξει σύντομα πώς θα γράψετε, θα αποσφαλματώσετε και θα τρέξετε ένα πρόγραμμα μέσα στο περιβάλλον αυτό, αλλά πρέπει να ασχοληθείτε μόνοι σας, να το ψάξετε αρκετά και να το μάθετε καλά. Αναζητήστε τις πολλές επιλογές που υπάρχουν στα μενού, θα σας φανούν σίγουρα χρήσιμες.



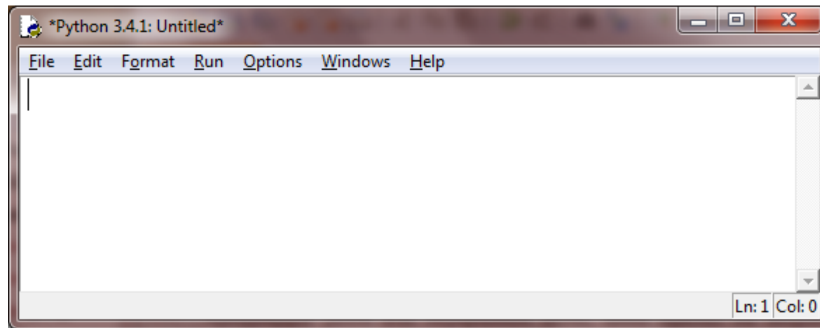
Σχήμα 2.2: Ο διερμηνευτής της Python.

Αφού, λοιπόν, εγκαταστήσετε το περιβάλλον στον υπολογιστή σας, εκτελέστε το πρόγραμμα **IDLE (Python GUI)** από το μενού εκκίνησης προγραμμάτων των Windows. Θα εμφανιστεί ένα παράθυρο σαν αυτό του Σχήματος 2.2, το οποίο είναι ο διερμηνευτής της Python. Εκεί μπορείτε να πληκτρολογείτε εντολές. Για παράδειγμα, αν πληκτρολογήσετε **a=5**, **b=7** και **a+b** θα πάρετε το αποτέλεσμα **12**. Δείτε το Σχήμα 2.3, όπου φαίνεται η εκτέλεση αυτών των εντολών. Αλλά ας επιστρέψουμε πάλι στο IDLE.

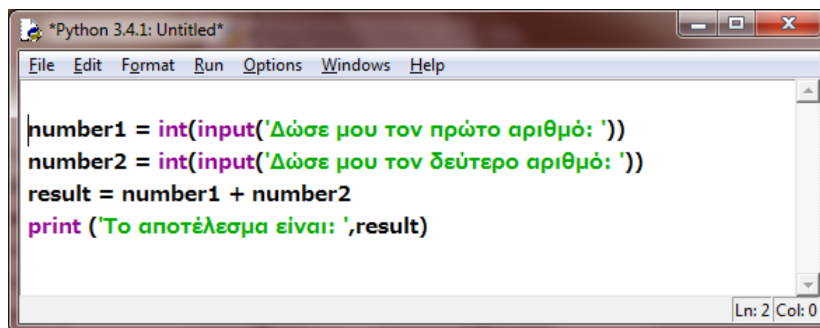


Σχήμα 2.3: Παράδειγμα χρήσης του διερμηνευτή.

Στο παράθυρο αυτό, από το μενού **File** επιλέξτε **New File**. Αυτό θα έχει ως αποτέλεσμα να ανοίξει ένα νέο παράθυρο, αυτό που φαίνεται στο Σχήμα 2.4. Αυτό το παράθυρο είναι ένας εκδότης κειμένου στον οποίο θα πληκτρολογήσουμε το πρόγραμμά μας. Όταν αυτό γίνει, το παράθυρο θα μοιάζει σαν αυτό που φαίνεται στο Σχήμα 2.5. Πληκτρολογήστε το, λοιπόν, και μετά αποθηκεύστε το στον δίσκο με κάποιο όνομα (έστω **add.py**).



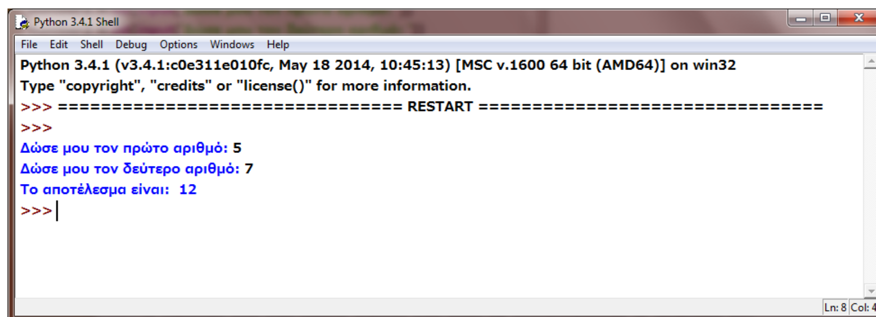
Σχήμα 2.4: Ο εκδότης κειμένου του IDLE.



Σχήμα 2.5: Το πρόγραμμα της άθροισης στον εκδότη κειμένου.

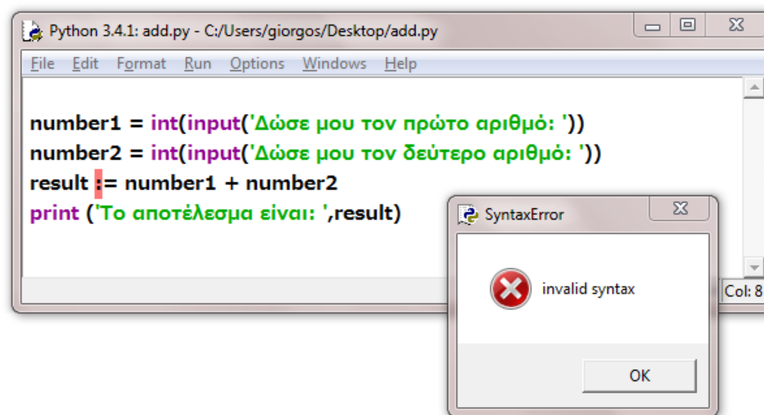
Στη συνέχεια, από το μενού **Run** επιλέξτε **Run Module**. Αυτό θα έχει ως αποτέλεσμα την εκτέλεση του προγράμματος **add.py** στο παράθυρο του διερμηνευτή. Το αποτέλεσμα θα είναι κάτι σαν αυτό που φαίνεται στο Σχήμα 2.6. Αρχικά, θα μας ζητηθεί να δώσουμε τον πρώτο αριθμό, στη συνέχεια τον δεύτερο και τέλος θα τυπωθεί το αποτέλεσμα στην οθόνη, με το κατάλληλο μήνυμα, όπως ακριβώς το είχαμε σχεδιάσει και κωδικοποιήσει νωρίτερα.

Ας κάνουμε τώρα μία αλλαγή στο πρόγραμμά μας, ένα τεχνητό λάθος, για να δούμε τι θα συμβεί. Έστω ότι αντικαθιστούμε το σύμβολο **=** στην τρίτη γραμμή με το σύμβολο **:=**. Το σύμβολο **:=** χρησιμοποιείται από πολλές γλώσσες ως σύμβολο εκχώρησης,όχι όμως από την Python. Προφανώς, τώρα υπάρχει ένα συντακτικό λάθος το οποίο αναμένουμε από την Python να αναγνωρίσει.



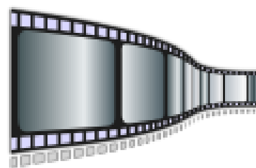
Σχήμα 2.6: Η εκτέλεση της άθροισης στον διερμηνευτή.

Πράγματι, λαμβάνουμε ένα μήνυμα ότι υπάρχει ένα συντακτικό λάθος και το σημείο στο οποίο έγινε το λάθος κοκκινίζει. Δείτε το Σχήμα 2.7, στο οποίο φαίνονται το μήνυμα λάθους και η σημείωση του λάθους πάνω στον κώδικα.



Σχήμα 2.7: Ένα παράδειγμα συντακτικού λάθους.

Θέλετε να πειραματιστούμε λίγο περισσότερο με το περιβάλλον της Python; Κάντε κλικ στην Ταινία 2.2 και ακολουθήστε με.



Ταινία 2.2: Το περιβάλλον της Python.

`http://repfiles.kallipos.gr/file/9281`

Ας σταματήσουμε, όμως, εδώ. Είδαμε πολύ λίγο το περιβάλλον εκτέλεσης IDLE. Συνεχίστε το μόνοι σας και προσπαθήστε να εξοικειωθείτε με αυτό. Τον σκοπό αυτού του κεφαλαίου τον πετύχαμε. Ξεκινήσαμε από το μηδέν, πήραμε ένα πρόβλημα, καταγράψαμε τις απαιτήσεις μας, σχεδιάσαμε τη λύση, γράψαμε τον κώδικα και το είδαμε να εκτελείται και να λειτουργεί σωστά. Τώρα μπορούμε να πανηγυρίσουμε!

2.4 Αποσφαλμάτωση κώδικα

Στην προηγούμενη ενότητα είδαμε, μεταξύ άλλων, και ένα παράδειγμα **αποσφαλμάτωσης κώδικα (code debugging)**. Αν και ο όρος **αποσφαλμάτωση κώδικα** είναι αρκετά περιγραφικός και εύστοχος στα ελληνικά, ο αντίστοιχος αγγλικός μοιάζει περίεργος και ακατανόητος. Ακούγονται διάφορες ερμηνείες για την προέλευση του όρου αυτού, το μόνο σίγουρο είναι ότι είναι πολύ παλιός και χρονολογείται τουλάχιστον στα τέλη του 19ου αιώνα.

Σφάλματα (bugs) ονομάζουμε λάθη που υπάρχουν στον κώδικά μας. Τα λάθη αυτά μπορεί να είναι **συντακτικά λάθη (syntax errors)** ή **λογικά λάθη (logic errors)**. Τα συντακτικά οφείλονται σε λάθος χρήση της γλώσσας. Είδαμε παραπάνω, στο Σχήμα 2.7, ένα τέτοιο. Ο διερμηνευτής (ή ο μεταγλωττιστής) έχει τη δυνατότητα να τα εντοπίσει και να μας ζητήσει να τα διορθώσουμε. Αντίθετα, τα λογικά σφάλματα ο διερμηνευτής (ή ο μεταγλωττιστής) δεν μπορεί να τα εντοπίσει. Ένα λογικό λάθος συμβαίνει όταν το πρόγραμμά μας είναι ορθό, τηρεί δηλαδή τους κανόνες της γλώσσας, μπορεί να εκτελεστεί χωρίς κανένα πρόβλημα, αλλά κάνει διαφορετικό πράγμα από αυτό που θέλουμε εμείς.

Στο παράδειγμα του Σχήματος 2.5, αν αντί για τη γραμμή:

```
result=number1+number2
```

γράφαμε:

```
result=number1-number2
```

η Python δεν θα είχε κανένα πρόβλημα. Πώς, άλλωστε, να φανταστεί ότι εμείς θέλαμε να κάνουμε πρόσθεση και όχι αφαίρεση. Το λάθος αυτό είναι ένα παράδειγμα λογικού λάθους. Δεν μπορεί να το βρει κανείς άλλος εκτός από εμάς. Τα λογικά λάθη είναι δύσκολο να βρεθούν, χρειάζεται συστηματική και επίμονη προσπάθεια. Είναι ίσως το πιο δύσκολο, επίπονο και πιο βαρετό σημείο στην ανάπτυξη ενός προγράμματος.

Η σημασία αλλά και η δυσκολία της διαδικασίας αυτής έχει οδηγήσει στη συστηματικοποίησή της. Υπάρχουν **εργαλεία αποσφαλμάτωσης (debuggers)**

που βοηθούν στη δουλειά αυτή. Έχουν αναπτυχθεί μεθοδολογίες και εργαλεία που βοηθούν στην εφαρμογή τους. Σε κάθε περίπτωση όμως, η διαίσθηση του προγραμματιστή και η ικανότητά του να αποσφαλματώνει ένα πρόγραμμα είναι το βασικότερο συστατικό της επιτυχίας.

Περισσότερο υλικό σχετικό με αυτό το κεφάλαιο μπορείτε να διαβάσετε στο δεύτερο κεφάλαιο των βιβλίων [1][2][3] και στο κεφάλαιο του [4] **τα πρώτα βήματα**.

Βιβλιογραφία:

1. Brian Heinold (2012).**Introduction to Programming Using Python**. Publisher: Mount St. Mary's University, Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.
2. Αχιλλέας Καμέας (2000). **Τεχνικές Προγραμματισμού**. Τόμος Β. ΠΛΗ-10, Ελληνικό Ανοικτό Πανεπιστήμιο.
3. Eric Roberts. (2004). **Η Τέχνη και Επιστήμη της C**. Μετάφραση: Γιώργος Στεφανίδης, Παναγιώτης Σταυρόπουλος, Αλέξανδρος Χατζηγεωργίου, Εκδόσεις Κλειδάριθμός.
4. C. H. Swaroop (2015). **A Byte of Python**. Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο, Μετάφραση: ubuntu-gr.org Team.

Κεφάλαιο 3: Εισαγωγή στους αλγορίθμους - διαγράμματα ροής

Αλγόριθμος (algorithm) λέγεται μία πεπερασμένη διαδικασία καλά ορισμένων βημάτων που ακολουθείται για τη λύση ενός προβλήματος. Το **διάγραμμα ροής προγράμματος (flowchart)**, σε συντομογραφία **ΔΡΠ**, μπορεί να το συναντήσετε σπανιότερα και ως **λογικό διάγραμμα**. Πρόκειται για μία σχηματική παράσταση του αλγορίθμου. Η έννοια του αλγορίθμου είναι πολύ βασική στην Πληροφορική, ενώ το διάγραμμα ροής ένα πολύ χρήσιμο περιγραφικό εργαλείο. Ένα διάγραμμα ροής αποτελείται από σύμβολα και λέξεις οι οποίες είναι σε θέση να περιγράψουν με λεπτομέρεια κάθε αλγόριθμο, κάθε διαδικασία δηλαδή επίλυσης οποιουδήποτε προβλήματος.

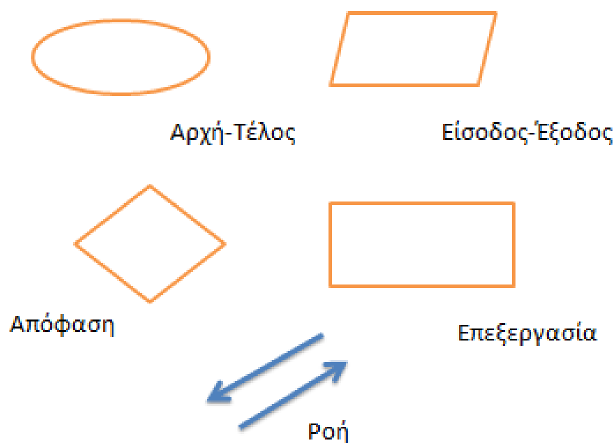
3.1 Διαγράμματα ροής

Η κατασκευή ενός διαγράμματος ροής προηγείται της φάσης της κωδικοποίησης. Με τον όρο **κωδικοποίηση (coding)** εννοούμε την ανάπτυξη του κώδικα. Το διάγραμμα ροής:

- δίνει τη δυνατότητα να μελετήσουμε καλά ένα πρόβλημα και να εμβαθύνουμε σε αυτό,
- επιτρέπει να σχεδιάσουμε μέρος του προγράμματος ή το πρόγραμμα στην ολότητά του,
- δίνει μία χρήσιμη οπτική αναπαράσταση του αλγορίθμου,
- επιτρέπει να διαπιστώσουμε εγκαίρως, αν ο τρόπος που έχουμε σκεφτεί για να λύσουμε το πρόβλημα είναι δόκιμος και να βεβαιωθούμε για την ορθότητα της λύσης,

- δίνει τη δυνατότητα να αποτυπώσουμε τις σκέψεις μας για να μπορέσουμε να εργαστούμε με άλλους συνεργάτες επί της λύσης, πριν καν ακόμα ξεκινήσουμε την επίπονη διαδικασία της κωδικοποίησης και
- επιτρέπει να συγκρίνουμε διαφορετικές μεταξύ τους λύσεις και να επιλέξουμε την καλύτερη, αρκετά νωρίς, αφού οι αλλαγές κατά την κωδικοποίηση είναι πολύ πιο ακριβές και επίπονες από ό,τι οι αλλαγές κατά τη σχεδίαση ενός λογισμικού.

Ας εμβαθύνουμε, λοιπόν, στα διαγράμματα ροής. Η σχηματική αναπαράσταση ενός αλγορίθμου πρέπει να είναι σε θέση να περιγράψει με απόλυτη ακρίβεια και λεπτομέρεια τα βήματα που θα ακολουθηθούν για τη λύση ενός προβλήματος, δηλαδή έναν αλγόριθμο. Για τον σκοπό αυτόν έχουμε επιλέξει κάποια θεμελιώδη δομικά στοιχεία τα οποία θεωρούμε ικανά και ευέλικτα αρκετά ώστε να μπορούν να περιγράψουν τις διαδικασίες που θα ακολουθήσουμε. Έχουμε, επίσης, επιλέξει κάποια ακόμα δομικά εργαλεία τα οποία θα μας δώσουν ακόμα μεγαλύτερη εκφραστικότητα και ευελιξία στην περιγραφή μας. Καθένα από αυτά τα δομικά στοιχεία έχουμε επιλέξει να το συμβολίσουμε με κάποιο γεωμετρικό σχήμα, μέσα στο οποίο μπορούμε να σημειώσουμε οδηγίες/περιγραφές ώστε να κάνουμε τον συμβολισμό ακόμα πιο ευέλικτο και να αυξήσουμε την εκφραστική του δύναμη.



Σχήμα 3.1: Τα βασικότερα σύμβολα που χρησιμοποιούνται στα διαγράμματα ροής.

Τα βασικότερα σύμβολα που χρησιμοποιούνται σε ένα διάγραμμα ροής είναι η **έλλειψη**, το **ορθογώνιο**, το **πλάγιο παραλληλόγραμμο**, ο **ρόμβος** και τα

βέλη. Τα σχήματα αυτά φαίνονται στο Σχήμα 3.1.

Τα **ορθογώνια παραλληλόγραμμα** συμβολίζουν επεξεργασία. Τα **πλάγια παραλληλόγραμμα** είσοδο και έξοδο δεδομένων, ενώ οι **ρόμβοι** αποτελούν σημεία στα οποία πρέπει να ληφθεί κάποια απόφαση. Με τα **βέλη** απεικονίζουμε τη ροή του προγράμματος. Πέρα από αυτά τα σύμβολα, υπάρχουν μερικά ακόμα, δευτερεύουσας σημασίας, αλλά επίσης χρήσιμα. Ας μείνουμε όμως τώρα σε αυτά και ας τα δούμε αναλυτικά ένα ένα.

Τα **βέλη** τα χρησιμοποιούμε για να δείξουμε ότι η ροή εκτέλεσης ενός προγράμματος μεταφέρεται από το ένα σχήμα σε ένα άλλο. Όταν, δηλαδή, εκτελεστεί μία εντολή, το βελάκι που ξεκινάει από αυτό το σχήμα κατευθύνεται προς το σχήμα το οποίο πρέπει να εκτελεστεί στη συνέχεια. Από κάποιο σχήμα μπορεί να μην ξεκινά κανένα βέλος (συμβαίνει όταν τερματίζεται το πρόγραμμα), μπορεί να ξεκινά ένα βέλος, το οποίο καταλήγει στο επόμενο σχήμα στο οποίο θα περάσει η εκτέλεση, ή περισσότερα βέλη όταν πρέπει να λάβουμε κάποια απόφαση.

Η **έλλειψη** χρησιμοποιείται για να σημειωθεί η αρχή και το τέλος του αλγορίθμου. Ανάλογα με το αν συμβολίζουμε αρχή ή τέλος σημειώνουμε μέσα τη λέξη **ΑΡΧΗ** ή **ΤΕΛΟΣ** αντίστοιχα μέσα την έλλειψη. Φυσικά, κάθε πρόγραμμα έχει μία μόνο αρχή, αλλά δεν πειράζει αν τερματίζεται σε δύο διαφορετικά σημεία, ανάλογα με τη διαδρομή που θα ακολουθήσει η ροή εκτέλεσης. Από κάθε έλλειψη που συμβολίζει αρχή προγράμματος ξεκινάει ακριβώς ένα βέλος ροής, ενώ σε κάθε έλλειψη που συμβολίζει τέλος μπορούν να καταλήγουν ένα ή περισσότερα τέτοια βέλη. Φυσικά, από μία έλλειψη που συμβολίζει τέλος δεν μπορεί να ξεκινάει κανένα βέλος ροής. Θα δούμε παραδείγματα για τα βέλη και τις ελλείψεις λίγο παρακάτω, αφού δούμε πρώτα πώς συμβολίζουμε την είσοδο και πώς την έξοδο δεδομένων.

Η διαδικασία εισόδου και εξόδου στοιχείων συμβολίζεται με ένα **πλάγιο παραλληλόγραμμο**. Μέσα στο πλάγιο παραλληλόγραμμο διευκρινίζουμε αν πρόκειται για είσοδο, αν πρόκειται για έξοδο και ποια μεταβλητή είναι αυτή που συμμετέχει. Αν, δηλαδή, θέλουμε να διαβάσουμε τη μεταβλητή **Α**, θα πρέπει μέσα στο πλάγιο παραλληλόγραμμο να γράψουμε κάτι σαν:

ΔΙΑΒΑΣΕ Α

Αν θέλουμε να τυπώσουμε στην οθόνη ένα μήνυμα το οποίο μας πληροφορεί για το αποτέλεσμα μιας πράξης, μπορούμε να γράψουμε:

ΤΥΠΩΣΕ “ΑΠΟΤΕΛΕΣΜΑ=”,Χ

όπου η λέξη **ΑΠΟΤΕΛΕΣΜΑ** θα τυπωθεί ως έχει (διότι είναι μέσα σε εισαγωγικά), ενώ το **Χ** θα αποτιμηθεί και θα εμφανιστεί η τιμή που έχει. Στο πα-

ράδειγμα της πρόσθεσης, είσοδο δεδομένων έχουμε στην αρχή του προγράμματος, όπου και πρέπει να δοθούν από τον χρήστη στον υπολογιστή οι δύο αριθμοί που θα προστεθούν. Σε εκείνο το σημείο έχουμε και έξοδο δεδομένων, αφού για κάθε έναν από τους δύο αριθμούς που ζητούνται προηγείται ένα μήνυμα διευκρινιστικό προς τον χρήστη:

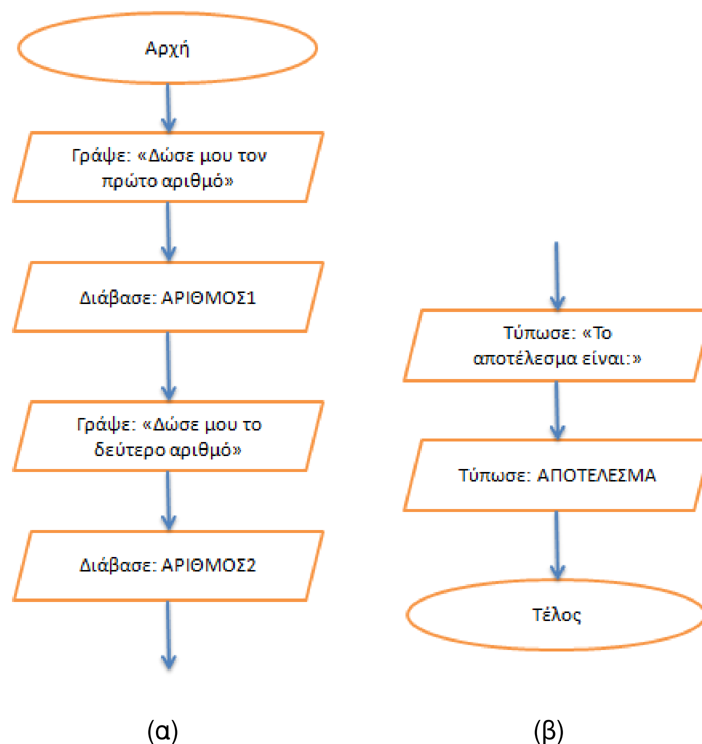
Δώσε μου τον πρώτο αριθμό

και

Δώσε μου τον δεύτερο αριθμό.

Έξοδο δεδομένων έχουμε στο τέλος του προγράμματος, όπου υπάρχει η εμφάνιση του αποτελέσματος στην οθόνη συνοδευόμενο πάλι από το κατάλληλο βοηθητικό μήνυμα:

Το αποτέλεσμα είναι:



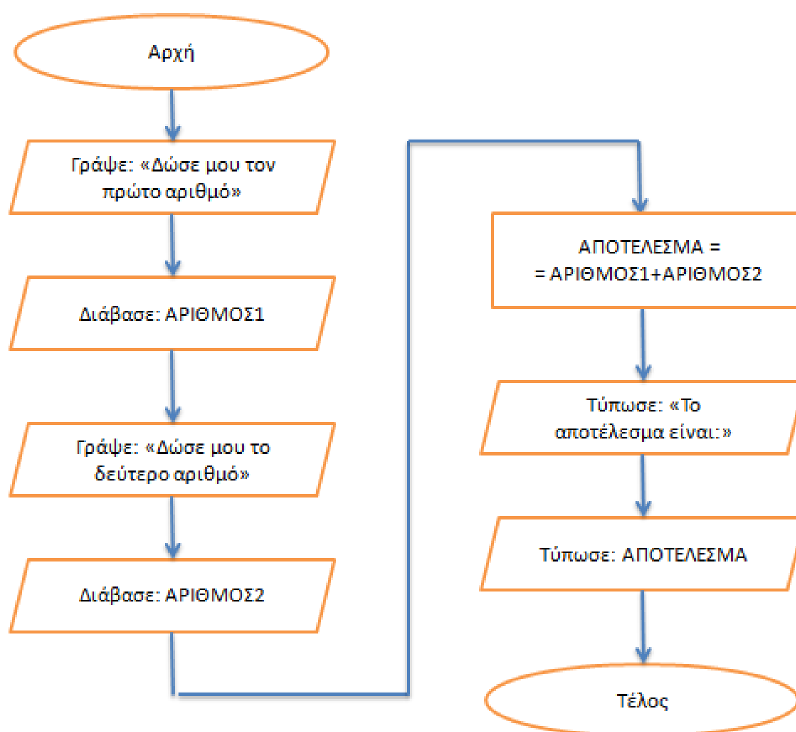
Σχήμα 3.2: Η είσοδος (α) και η έξοδος (β) του προγράμματος της πρόσθεσης.

Τα μέρη του διαγράμματος ροής προγράμματος για το πρόβλημα της πρόσθεσης που αναλογούν στην αρχή και στο τέλος του προγράμματος και στην

είσοδο και έξοδο δεδομένων φαίνονται στο Σχήμα 3.2. Στο Σχήμα 3.2(α) φαίνεται η έλλειψη στην οποία είναι σημειωμένη η αρχή του προγράμματος καθώς και τέσσερα παραλληλόγραμμα, δύο που τυπώνουν μηνύματα στην οθόνη, έτσι ώστε να γίνει πιο κατανοητό στον χρήστη τι του ζητείται, και δύο ώστε να γίνει η είσοδος από το πληκτρολόγιο για τις μεταβλητές **ΑΡΙΘΜΟΣ1** και **ΑΡΙΘΜΟΣ2**.

Στο Σχήμα 3.2(β) φαίνονται τα αντίστοιχα δύο πλάγια παραλληλόγραμμα για την εμφάνιση του αποτελέσματος στην οθόνη και του αντίστοιχου μηνύματος πριν από αυτό. Η έλλειψη στο τέλος υποδηλώνει το τέλος του προγράμματος.

Το **ορθογώνιο παραλληλόγραμμο** χρησιμοποιείται για να περιγραφεί κάποια διαδικασία. Στο παράδειγμα θα συμβολιστεί με ορθογώνιο παραλληλόγραμμο η πρόσθεση των δύο αριθμών, η οποία θα τοποθετηθεί ανάμεσα στην είσοδο και στην έξοδο των δεδομένων. Το διάγραμμα ροής προγράμματος για το πρόβλημα της πρόσθεσης εικονίζεται ολοκληρωμένο στο Σχήμα 3.3.



Σχήμα 3.3: Ολοκληρωμένο το διάγραμμα ροής της πρόσθεσης.

Ο **ρόμβος** χρησιμοποιείται για να συμβολιστούν σημεία στα οποία πρέπει να ληφθεί κάποια απόφαση. Σημεία, δηλαδή, στα οποία το πρόγραμμα έχει

περισσότερους από έναν δρόμο να επιλέξει ανάλογα με το αν ισχύουν ή όχι κάποιες συνθήκες. Έτσι, από ένα ρόμβο μπορούν να ξεκινούν περισσότερα από ένα βέλη και να κατευθύνονται σε περισσότερα του ενός σχήματα, ανάλογα φυσικά με το πόσες επιλογές έχει η εντολή απόφασης που θέλουμε να συμβολίσουμε.

Στο απλό πρόγραμμα της πρόσθεσης που έχουμε φτιάξει μέχρι τώρα δεν απαιτείται κάπου να ληφθεί μία απόφαση, ώστε να χρησιμοποιηθεί ένας ρόμβος. Παρακάτω θα δούμε ένα άλλο πρόβλημα, την εύρεση των ριζών ενός τριωνύμου, στο οποίο χρησιμοποιείται εντολή απόφασης και εμφανίζεται στο διάγραμμα ροής ένας ρόμβος.

Περισσότερα για διαγράμματα ροής μπορείτε να βρείτε στο κεφάλαιο 2 του βιβλίου [1] και στη συνέχεια θα σας συνέστηνα την αναζήτηση στο διαδίκτυο.

3.2 Τα πρώτα αλγοριθμικά μας βήματα

3.2.1 Ρίζες τριωνύμου

Πριν, όμως, προχωρήσουμε στο να δούμε πώς ένα διάγραμμα ροής μεταφράζεται σε κώδικα ή, ακόμα πιο σωστά, ποια είναι τα δομικά στοιχεία που χρησιμοποιούμε ώστε να κτίσουμε ένα πρόγραμμα σε μία διαδικασιακή γλώσσα, ας δούμε κάποια λίγο μεγαλύτερα παραδείγματα.

Ας ξεκινήσουμε με ένα πρόγραμμα που δέχεται σαν είσοδο ένα τριώνυμο:

$$f(x) = \alpha x^2 + \beta x + \gamma$$

το οποίο αντιστοιχεί στη λύση της δευτεροβάθμιας εξίσωσης:

$$\alpha x^2 + \beta x + \gamma = 0$$

Επιθυμούμε το πρόγραμμα που θα φτιάξουμε να αποφασίζει αν το τριώνυμο έχει, πόσες και ποιες πραγματικές ρίζες. Στην περίπτωση που το τριώνυμο έχει πραγματικές ρίζες, τότε αυτές υπολογίζονται και τυπώνονται στην οθόνη. Στην περίπτωση που το τριώνυμο δεν έχει πραγματικές ρίζες, τότε τυπώνεται ένα μήνυμα που μας ειδοποιεί ότι δεν υπάρχουν πραγματικές ρίζες.

Είσοδος στοιχείων απαιτείται κατά την εκκίνηση του προγράμματος όπου και ζητείται το τριώνυμο, οι συντελεστές **α**, **β** και **γ** δηλαδή. Έτσι, στην αρχή του διαγράμματος ροής υπάρχουν τρία πλάγια παραλληλόγραμμα. Για να είναι το πρόγραμμα φιλικό ως προς τον χρήστη, πριν από κάθε ερώτηση για τις

τιμές των **α**, **β** και **γ** πρέπει να τυπώνεται ένα μήνυμα στην οθόνη που να πληροφορεί τον χρήστη τι ακριβώς του ζητείται να κάνει. Συνεπώς, πριν από κάθε πλάγιο παραλληλόγραμμο για την εισαγωγή μίας τιμής, πρέπει να υπάρχει και ένα άλλο πλάγιο παραλληλόγραμμο που να τυπώνει ένα μήνυμα της μορφής:

Δώσε μου τον συντελεστή α:

ή

Δώσε μου τον συντελεστή β:

ή

Δώσε μου τον συντελεστή γ:

Στο τέλος του προγράμματος πρέπει να τυπώνεται το αποτέλεσμα. Έτσι, σε κάποιο σημείο του διαγράμματος ροής πρέπει να υπάρχουν τα κατάλληλα πλάγια παραλληλόγραμμα για τα αποτελέσματα:

Το τριώνυμο δεν έχει πραγματικές ρίζες

και

Το τριώνυμο έχει μία πραγματική διπλή ρίζα την:

και

Το τριώνυμο έχει δύο πραγματικές ρίζες τις:

Επίσης, αν διαπιστωθεί ότι η εξίσωση είναι πρωτοβάθμια πρέπει να τυπωθεί το κατάλληλο μήνυμα:

Η εξίσωση είναι πρωτοβάθμια

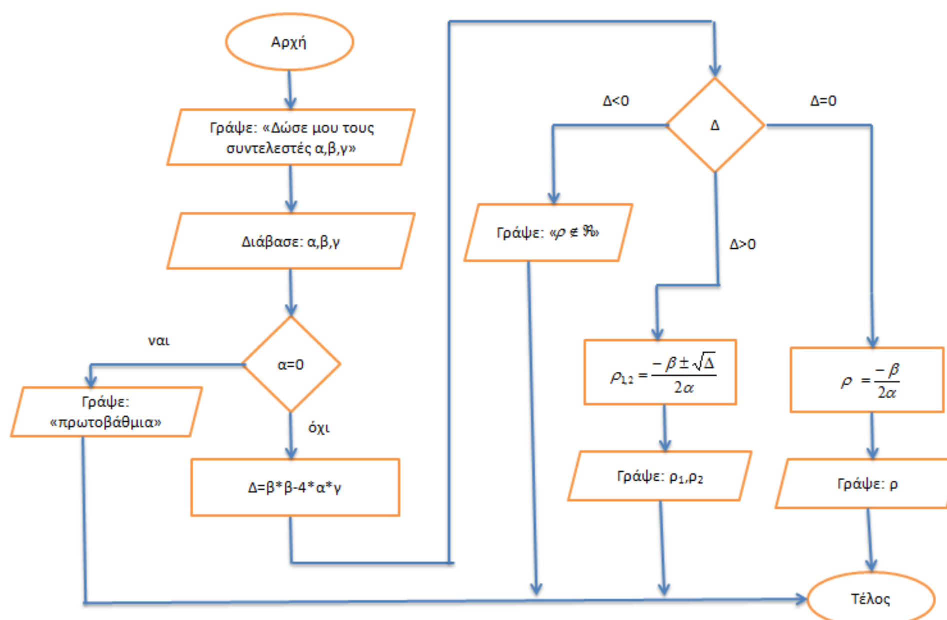
αν και αυτό μαθηματικά δεν είναι απόλυτα σωστό, αλλά ας κρατήσουμε το πρόγραμμα σχετικά απλό. Μπορεί κανείς να παρατηρήσει ότι σε ένα τόσο απλό πρόγραμμα και για μια τόσο απλή διαδικασία απαιτούνται τόσα πολλά πλάγια παραλληλόγραμμα. Το ίδιο πρόβλημα υπάρχει με όλα τα σχήματα του διαγράμματος ροής. Για να μειώσουμε την έκτασή του συνηθίζουμε να ομαδοποιούμε λειτουργίες περισσοτέρων σχημάτων σε ένα, όταν αυτό δεν αποβαίνει σε βάρος της περιγραφής του αλγορίθμου. Στο συγκεκριμένο παράδειγμα μπορούμε να χρησιμοποιήσουμε ένα ή δύο πλάγια παραλληλόγραμμα αντί για έξι στην αρχή του προγράμματος, ενώ από ένα μπορεί να χρησιμοποιηθεί και για καθεμία από τις πιθανές περιπτώσεις αποτελέσματος.

Στο παράδειγμα του τριωνύμου χρησιμοποιείται και ο ρόμβος. Ο αλγόριθμος ξεκινάει ζητώντας από τον χρήστη να του δώσει τους τρεις συντελεστές

του τριωνύμου. Αυτό γίνεται με τα δύο πλάγια παραλληλόγραμμα μετά ακριβώς από την έλλειψη που υποδηλώνει την αρχή του προγράμματος. Στη συνέχεια ακολουθεί ο ρόμβος, ο οποίος χρησιμοποιείται για να λάβουμε κάποια απόφαση. Στο σημείο αυτό εξετάζει εάν το **a**, ισούται με μηδέν ή όχι. Από τον ρόμβο αυτόν ξεκινάνε δύο βέλη, ένα για την περίπτωση που το **a**, είναι πράγματι ίσο με μηδέν και ένα για την περίπτωση που δεν είναι. Το πρόγραμμα ακολουθεί τελείως διαφορετική διαδρομή σε κάθε περίπτωση. Στην πρώτη, αποφασίζει ότι το τριώνυμο πρακτικά είναι πρώτου βαθμού και εμφανίζει το κατάλληλο μήνυμα λίγο πριν τερματίσει την εκτέλεση.

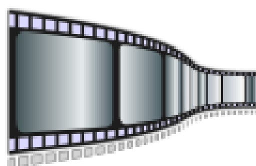
Πρέπει εδώ να σημειώσουμε ότι θα μπορούσαμε να συνεχίσουμε την εκτέλεση και να βρούμε τις ρίζες, αφού και στην περίπτωση αυτήν υπάρχει μεθοδολογία λύσης. Ο μόνος λόγος που δεν το κάναμε ήταν για να μη μεγαλώσει πολύ το διάγραμμα ροής. Στην περίπτωση, τώρα, που το α είναι διαφορετικό του μηδενός, ακολουθούμε το βέλος που πηγαίνει προς τα κάτω. Εκεί, υπολογίζεται η διακρίνουσα μέσα στο ορθογώνιο παραλληλόγραμμο. Στη συνέχεια, ακολουθεί ρόμβος, δηλαδή εντολή απόφασης, για να ελεγχθεί εάν η διακρίνουσα είναι θετική, αρνητική ή μηδέν, και φυσικά ακολουθείται και ο ανάλογος δρόμος.

Αν, λοιπόν, η διακρίνουσα είναι αρνητική, η ροή ελέγχου κατευθύνεται μέσα από το αριστερό βέλος και στο πλάγιο παραλληλόγραμμο που μας πληροφορεί ότι δεν υπάρχουν πραγματικές ρίζες. Στην περίπτωση που η διακρίνουσα είναι θετική, τότε, μέσα από το βέλος που ξεκινάει από το κάτω μέρος του ρόμβου η εκτέλεση πηγαίνει στο ορθογώνιο παραλληλόγραμμο που υπολογίζει τις δύο ρίζες και στη συνέχεια στο πλάγιο παραλληλόγραμμο που μας πληροφορεί γι' αυτές. Τέλος, αν η διακρίνουσα είναι μηδέν, θα ακολουθηθεί η τρίτη επιλογή, θα κατευθυνθούμε στο ορθογώνιο παραλληλόγραμμο που θα υπολογίσει τη μοναδική ρίζα και στη συνέχεια στο πλάγιο που θα την τυπώσει. Και οι τρεις δρόμοι συγκλίνουν στο τέλος στην έλλειψη, που υποδηλώνει τον τερματισμό του προγράμματος. Το διάγραμμα ροής για το τριώνυμο φαίνεται στο Σχήμα 3.4. Αποτελεί ένα πολύ χαρακτηριστικό παράδειγμα λογικών αποφάσεων που λαμβάνονται κατά την εκτέλεση ενός προγράμματος.



Σχήμα 3.4: Διάγραμμα ροής για την επίλυση του τριωνύμου.

Δείτε βήμα βήμα την εκτέλεση του αλγορίθμου στην Ταινία 3.1.



Ταινία 3.1: Επίλυση του τριωνύμου.

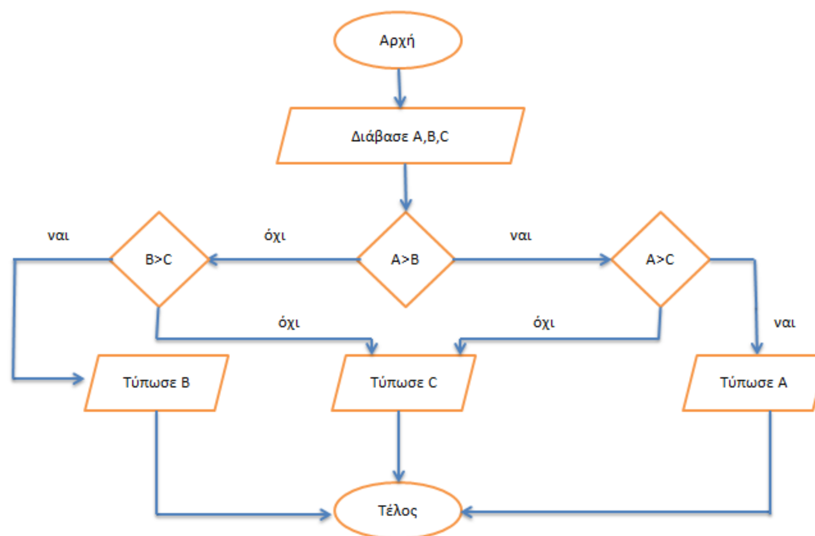
`http://repfiles.kallipos.gr/file/22380`

3.2.2 Μέγιστος τριών αριθμών

Ας δούμε λίγα ακόμα ενδιαφέροντα διαγράμματα ροής προγράμματος. Στο Σχήμα 3.5 φαίνεται ένα διάγραμμα ροής στο οποίο τρεις αριθμοί συγκρίνονται έτσι ώστε να βρεθεί ποιος από τους τρεις είναι ο μεγαλύτερος. Δεν μας ενδιαφέρει να διαπιστώσουμε ότι, για παράδειγμα, ο δεύτερος αριθμός είναι ο μεγαλύτερος, αλλά ποια τιμή έχει ο μεγαλύτερος από τους τρεις αριθμούς. Δηλαδή, αν έχουμε σαν είσοδο τους αριθμούς 3,8,5, η απάντηση είναι 8.

Το διάγραμμα ροής ξεκινάει εισάγοντας τους τρεις αριθμούς, έστω **A**,**B** και **C**. Στη συνέχεια, εξετάζουμε ποιος από τους δύο πρώτους (**A** ή **B**) είναι ο μεγα-

λύτερος. Αν ο **A** είναι ο μεγαλύτερος, τότε αποκλείουμε την περίπτωση ο **B** να είναι ο μεγαλύτερος από τους τρεις. Έτσι, συνεχίζουμε την αναζήτηση ανάμεσα στους **A** και **C**. Όμοια, αν ο **B** ήταν ο μεγαλύτερος, τότε θα αποκλείαμε την περίπτωση ο **A** να ήταν ο μεγαλύτερος από τους τρεις. Έτσι, συνεχίζουμε την αναζήτηση ανάμεσα στους **B** και **C**. Η δεύτερη σύγκριση ανάμεσα στους δύο πιθανούς αριθμούς, αφού αποκλείστηκε ο ένας από τους τρεις, μας φανερώνει τον μέγιστο, ο οποίος και τυπώνεται.



Σχήμα 3.5 Διάγραμμα ροής για την εύρεση του μέγιστου τριών αριθμών.

Παρατηρήστε ότι το διάγραμμα ροής προγράμματος επιστρέφει σωστό αποτέλεσμα στην περίπτωση που ο μεγαλύτερος αριθμός εμφανίζεται σε δύο ή και στις τρεις από τις μεταβλητές **A**,**B**,**C**. Ας δούμε γιατί. Έστω **A=B** και **A>C** και φυσικά **B>C**. Η πρώτη σύγκριση θα μας οδηγήσει στον ρόμβο **B>C** και στη συνέχεια θα τυπωθεί το **B**. Δεν μας πειράζει ότι το αποτέλεσμα τυπώθηκε από το **B**. Θα μπορούσε να είχε τυπωθεί και από το **A**, αν είχαμε σχεδιάσει διαφορετικά το διάγραμμα. Σε κάθε περίπτωση όμως και ανεξάρτητα από το αν θα τυπωθεί το αποτέλεσμα από το **A** ή το **B**, ο αριθμός που θα τυπωθεί θα έχει τη μέγιστη τιμή, την τιμή δηλαδή που αναζητούσαμε. Σημειώνουμε πάλι ότι το διάγραμμα ροής θα ήταν διαφορετικό αν δεν μας ένοιαζε να τυπωθεί η μέγιστη τιμή, αλλά αν μας ένοιαζε και από ποια ή από ποιες από τις **A**,**B** και **C** προήλθε αυτή η μέγιστη τιμή.

3.2.3 Παραγοντικό

Στο Σχήμα 3.6 παρουσιάζεται το διάγραμμα ροής προγράμματος για τον υπολογισμό του παραγοντικού. Για να υπολογίσουμε το παραγοντικό του **x** θα πρέπει να υπολογίσουμε την παράσταση:

$$f(x) = 1 * 2 * 3 * ... * x$$

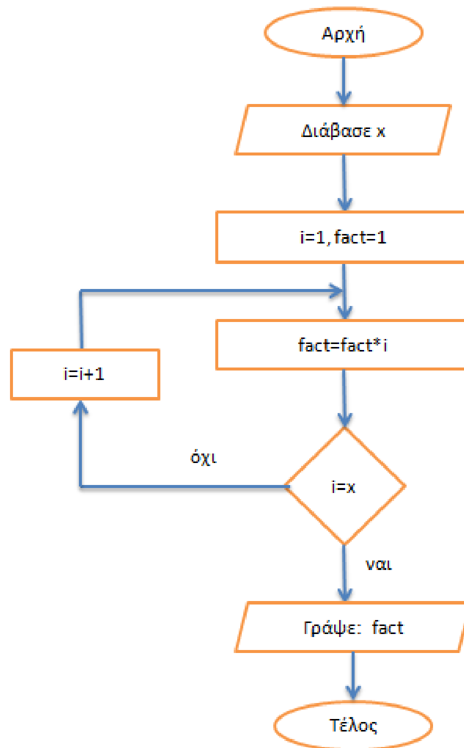
Χρειαζόμαστε, δηλαδή, μια μεταβλητή που θα ξεκινήσει από την τιμή 1, θα αυξάνεται σε κάθε βήμα κατά 1 και θα καταλήγει να έχει την τιμή **x**. Ας ονομάσουμε τη μεταβλητή αυτήν **i**. Σε κάθε βήμα θα πρέπει μία άλλη μεταβλητή να πολλαπλασιάζει την τιμή που είχε στο προηγούμενο βήμα με την τιμή της **i**. Ας την ονομάσουμε **fact**. Δείτε στη μέση του Σχήματος 3.6 την αύξηση των μεταβλητών **i** και **fact** κατά 1 και **i** φορές αντίστοιχα.

Παρατηρήστε, επίσης, στην αρχή του διαγράμματος την αρχικοποίηση των μεταβλητών αυτών σε 1. Το **i** είναι προφανές γιατί πρέπει να αρχικοποιηθεί σε 1, αφού είπαμε ότι θέλουμε να υπολογίσουμε την παράσταση

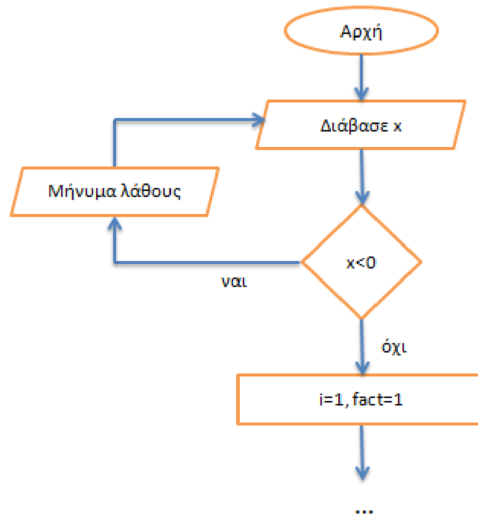
$$f(x) = 1 * 2 * 3 * ... * x$$

Το **fact** πρέπει και αυτό να αρχικοποιηθεί σε 1, και ο λόγος είναι ότι θέλουμε να το αρχικοποιήσουμε στο ουδέτερο στοιχείο του πολλαπλασιασμού. Σκεφτείτε τι θα γινόταν αν το αρχικοποιούσαμε στο 0. Το **fact** θα είχε πάντοτε την τιμή 0, αφού με ό,τι και να το πολλαπλασιάζαμε το αποτέλεσμα θα ήταν 0. Τώρα, σχεδόν έχουμε περιγράψει όλο το διάγραμμα, εκτός από το ρόμβο που συγκρίνει το **i** με το **x**.

Μετά την αύξηση της τιμής του **fact**, το **i** ελέγχεται για να διαπιστωθεί εάν έχουν γίνει όλες οι επαναλήψεις που απαιτούνται. Εάν έχουν γίνει, δηλαδή αν οι δύο μεταβλητές έχουν την ίδια τιμή, τότε ο υπολογισμός του παραγοντικού έχει τελειώσει και τυπώνουμε το αποτέλεσμα. Εάν όχι, οδηγούμε τον έλεγχο στο σημείο που αυξάνεται το **i** κατά 1 και εκτελούμε ακόμα μία επανάληψη.



Σχήμα 3.6: Διάγραμμα ροής για την εύρεση του παραγοντικού



Σχήμα 3.7: Έλεγχος ορθής εισόδου για το διάγραμμα του παραγοντικού.

Το διάγραμμα ροής προγράμματος για τον υπολογισμό του παραγοντικού φαίνεται να λειτουργεί. Για να είμαστε όμως περισσότερο ορθοί, θα πρέπει να ελέγξουμε ότι η τιμή που δόθηκε από τον χρήστη για τη μεταβλητή **x** είναι νόμιμη. Το παραγοντικό ορίζεται για αριθμούς μεγαλύτερους ή ίσους από το 0. Έτσι, εάν από τον χρήστη δοθεί αρνητικός αριθμός, θα πρέπει να εμφανιστεί κάποιο κατάλληλο μήνυμα λάθους και να επιστρέψει ο έλεγχος στην εντολή που ζητά να δοθεί τιμή για το **x**. Έτσι, όσο ο χρήστης δίνει αρνητικές τιμές για το **x**, πρέπει να του εμφανίζεται το μήνυμα λάθους και να του ζητείται να δώσει εκ νέου τιμή. Το τμήμα του διαγράμματος ροής που υλοποιεί την παραπάνω λειτουργία φαίνεται στο Σχήμα 3.7.

Όλα τώρα φαίνεται να λειτουργούν σωστά. Ή μήπως όχι; Αναφέραμε προηγουμένως ότι το παραγοντικό ορίζεται για αριθμούς μεγαλύτερους ή ίσους από το μηδέν. Εμείς στον συλλογισμό που κάναμε για το διάγραμμα ροής δεν εξετάσαμε καθόλου τι γίνεται για **x=0** και μάλιστα αρχικοποιήσαμε την τιμή του **fact** σε 1. Είναι σωστό αυτό που κάναμε ή πρέπει να διορθώσουμε κάτι;

Καταρχήν ας θυμηθούμε ότι το παραγοντικό του 0 είναι ίσο με 1 (0!=1). Ας δούμε τι θα βγάλει το διάγραμμα ροής προγράμματος αν δώσουμε στο **x** την τιμή 0. Το **fact** θα αρχικοποιηθεί σε 1, θα εκτελεστεί μία φορά η

fact=fact*i

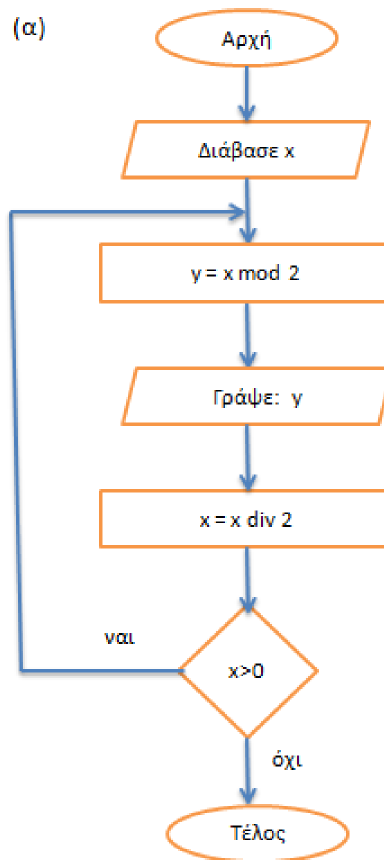
χωρίς να αλλάξει η τιμή του **fact**, αφού το **i** ισούται με 1, και δεν θα γίνει άλλη επανάληψη, αφού ο ρόμβος θα οδηγήσει την εκτέλεση έξω από το βρόχο (το **i** ισούται με **x**). Τώρα έχουμε βεβαιωθεί ότι όλα έχουν πάει καλά και μπορούμε να θεωρήσουμε ότι ολοκληρώθηκε το διάγραμμα ροής.

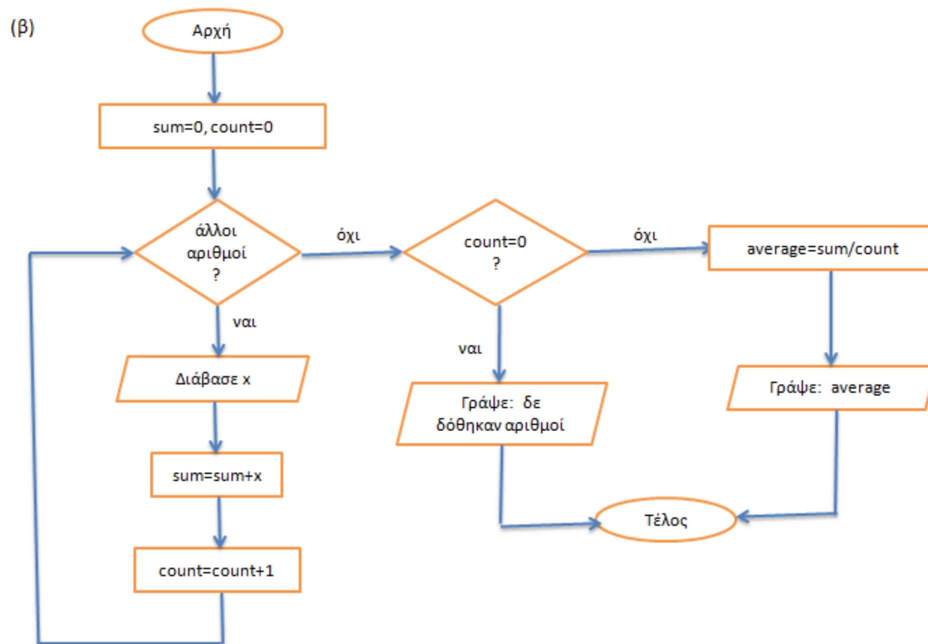
Ασκήσεις που μπορείτε να κάνετε μόνοι σας

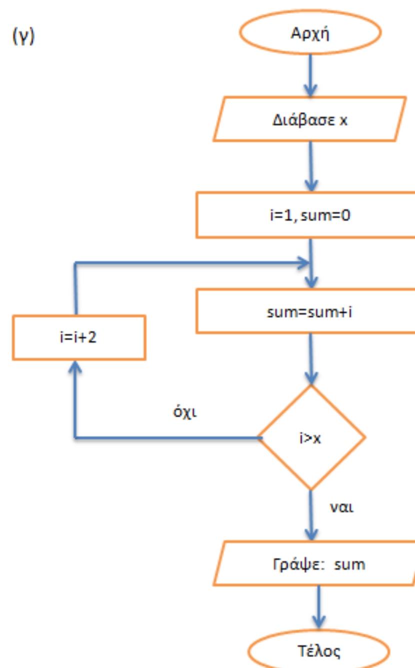
- Να κατασκευασετε το διάγραμμα ροής για τη λύση της πρωτοβάθμιας εξίσωσης.
- Να τροποποιήστε το διάγραμμα ροής του τριωνύμου ώστε να συνεχίζει τη λύση στην περίπτωση που το α ισούται με μηδέν αλλά και να βρίσκει και να τυπώνει τις μιγαδικές ρίζες όταν η διακρίνουσα είναι αρνητική.
- Σε αντιστοιχία με το διάγραμμα ροής του παραγοντικού να φτιάξετε ένα αντίστοιχο διάγραμμα ροής που να υπολογίζει την ύψωση ενός αριθμού σε ακέραια δύναμη. Μην ξεχάσετε να ελέγξετε ότι το πρόγραμμά σας λειτουργεί και για αρνητικές

τιμές του εκθέτη, καθώς και ότι δεν δημιουργούνται προβλήματα για μηδενικές τιμές τόσο του εκθέτη όσο και της βάσης.

- Να υλοποιήσετε διάγραμμα ροής το οποίο να βρίσκει τον μέγιστο τεσσάρων αριθμών.
- Να εξηγήσετε τι κάνει καθένα από τα διαγράμματα των Σχημάτων 3.8α, 3.8β και 3.8γ.







Σχήμα 3.8: Διαγράμματα ροής για άσκηση

Βιβλιογραφία

1. Αχιλλέας Καμέας (2000). **Τεχνικές Προγραμματισμού.** Τόμος Β. ΠΛΗ-10, Ελληνικό Ανοικτό Πανεπιστήμιο.

Κεφάλαιο 4: Μεταβλητές και εκφράσεις

Έχουμε ήδη μία αίσθηση του τι είναι **μεταβλητή**. Τις έχουμε ήδη χρησιμοποιήσει στο παράδειγμα του τριωνύμου ώστε να κρατήσουμε τις τιμές για τα **α**, **β** και **γ**, και έχουμε ήδη μιλήσει για τα κουτάκια αυτά που τα γεμίζουμε, χρησιμοποιούμε το περιεχόμενο τους ή το αντικαθιστούμε αν θέλουμε.

Η **μεταβλητή**, λοιπόν, είναι ένα συμβολικό όνομα μιας περιοχής της μνήμης στην οποία μπορούμε να γράψουμε και να ανακτήσουμε δεδομένα μέσω του συμβολικού αυτού ονόματος. Η πράξη με την οποία αναθέτουμε τιμές στις μεταβλητές λέγεται **εκχώρηση**. Τις μεταβλητές μπορούμε να τις χρησιμοποιήσουμε μέσα σε παραστάσεις (π.χ. μαθηματικές), να αποτιμηθούν και να υπολογιστεί μέσα από αυτήν την παράσταση μία νέα τιμή η οποία και μπορεί να εκχωρηθεί σε μία νέα μεταβλητή. Τις παραστάσεις αυτές θα τις λέμε **εκφράσεις**. Πριν δούμε όμως περισσότερα για τις εκφράσεις, ας δούμε λίγο περισσότερα για τις μεταβλητές και τη χρήση τους μέσα στα προγράμματα.

4.1 Μεταβλητές στις διάφορες γλώσσες προγραμματισμού

Στις περισσότερες γλώσσες προγραμματισμού, μία μεταβλητή έχει κάποιον τύπο ο οποίος και δεν αλλάζει. Συνηθισμένοι απλοί τύποι μεταβλητών είναι:

- οι **ακέραιες (integer)** μεταβλητές
- οι **πραγματικές (real)** μεταβλητές
- οι **χαρακτήρες (characters)**
- και οι **λογικές (boolean)** μεταβλητές.

Σύνθετοι τύποι μεταβλητών μπορεί να είναι:

- τα **αλφαριθμητικά (strings)**

- οι **πίνακες (arrays)**
- και σε κάποιες γλώσσες **οι λίστες (lists)**
- ή άλλες πιο σύνθετες δομές

Οι απλοί τύποι μεταβλητών, στις περισσότερες γλώσσες προγραμματισμού, και ιδιαίτερα σε γλώσσες που ακολουθούν τη φιλοσοφία των μεταφραστών, καταλαμβάνουν συγκεκριμένο χώρο στη μνήμη. Σε κάποιο χώρο της μνήμης έχει οριστεί μία περιοχή στην οποία θα αποθηκευτούν οι μεταβλητές ενός προγράμματος. Πρόκειται για μια απλοποιημένη θεώρηση, όχι μακριά όμως από την πραγματικότητα. Στην αρχή της περιοχής αυτής βρίσκεται ο χώρος που έχει δεσμευτεί για την πρώτη μεταβλητή, αμέσως μετά ο χώρος που έχει δεσμευτεί για τη δεύτερη κ.ο.κ. Να σημειώσουμε εδώ ότι ο χώρος που απαιτείται για την αποθήκευση μιας μεταβλητής εξαρτάται από τον τύπο της μεταβλητής και την αρχιτεκτονική του συστήματος. Για το λόγο αυτό πρέπει ο τύπος τους να δηλώνεται με σαφήνεια στο πρόγραμμα και δεν είναι δυνατόν να αλλάξει σε καμία φάση της μεταγλώττισης ή της εκτέλεσης.

Σχήμα 4.1: Παράδειγμα αποθήκευσης μεταβλητών (γλώσσα C).

Ας δούμε ένα παράδειγμα. Στη C ένας ακέραιος μπορεί να είναι (ανάλογα με την αρχιτεκτονική 2 ή 4 bytes, έστω 2 στο παράδειγμα μας), ένας ακέραιος μεγάλης ακρίβειας 4 bytes, ένας πραγματικός 4 bytes, ένας χαρακτήρας 1 byte και ένας πραγματικός διπλής ακρίβειας 8 bytes. Αν δηλωθούν με τη σειρά, το **i** σαν ακέραιος, το **L** σαν ακέραιος μεγάλης ακρίβειας, **r1** και **r2** σαν πραγματικοί, το **c** σαν χαρακτήρας, και το **D** σαν πραγματικός διπλής ακρίβειας, η εικόνα του τμήματος της μνήμης που κρατάει τις παραπάνω μεταβλητές θα ήταν όπως αυτή που φαίνεται στο Σχήμα 4.1, όπου κάθε κουτάκι αντιστοιχεί σε ένα byte, και η μνήμη μεγαλώνει από τα αριστερά στα δεξιά. Συνολικά, δηλαδή, θα χρειαζόντουσαν 23 bytes για να αποθηκευτούν όλες αυτές οι μεταβλητές στη μνήμη.

Σε κάποιες γλώσσες, όπως για παράδειγμα στη C, υπάρχει και η έννοια του **προσήμου (sing)**. Μία μεταβλητή, δηλαδή, μπορεί να δηλωθεί σαν **προσημασμένη (singed)** ή **μη προσημασμένη (unsinged)**. Αν μία ακέραια μεταβλητή δηλωθεί ως **μη προσημασμένη**, τότε ο χώρος της μνήμης που θα χρειαζόταν για να αναπαρασταθεί το πρόσημο μπορεί να χρησιμοποιηθεί ώστε να αναπαραστήσουμε μεγαλύτερους αριθμούς. Για παράδειγμα, αν έχουμε στη διάθεση

μας 8 bit, τότε μπορούμε να αναπαραστήσουμε 256 ακέραιους αριθμούς: από το -128, έως το 127. Αν όμως γνωρίζουμε ότι ο αριθμός που θέλουμε να αναπαραστήσουμε είναι θετικός, τότε μπορούμε να τον δηλώσουμε ως **μη προσημασμένο**. Στην περίπτωση αυτή, οι 256 αριθμοί που μπορούμε να αναπαραστήσουμε είναι από το 0 μέχρι το 255 (διευκρίνιση: πρόκειται για παράδειγμα, κανένας υπολογιστής δεν χρησιμοποιεί μόνο 8 bit για την αναπαράσταση αριθμών).

4.2 Μεταβλητές στην Python

Στην Python τα πράγματα είναι πολύ πιο απλά. Εκμεταλλευόμενοι το γεγονός ότι η Python είναι γλώσσα **διερμηνεύσιμη (interpreted)**, ακολουθεί δηλαδή τη φιλοσοφία των διερμηνευτών, υπάρχει πολύ μεγαλύτερη ευελιξία από ό,τι στις γλώσσες των που υλοποιούνται με τη φιλοσοφία των μεταφραστών. Οι μεταβλητές στην Python:

- δεν χρειάζεται να δηλωθούν σε κανένα σημείο του προγράμματος
- έχουν τύπο, αλλά ο τύπος αποκτάται όταν τους πρωτοεκχωρηθεί τιμή
- ο τύπος τους μπορεί να αλλάξει κατά την εκτέλεση του προγράμματος

Οι βασικοί τύποι των μεταβλητών της Python είναι οι ακόλουθοι:

- **αριθμητικές (numerical)** μεταβλητές
- **λογικές (boolean)** μεταβλητές
- **αλφαριθμητικά (strings)**
- **λίστες (lists)**
- **πλειάδες (tuples)**
- **λεξικά (dictionaries)**

Ενώ οι αριθμητικές μεταβλητές χωρίζονται σε:

- **ακέραιες** μεταβλητές
- μεταβλητές **ακέραιες υψηλής ακρίβειας**
- **πραγματικές** μεταβλητές
- **μιγαδικές** μεταβλητές

Με τη βοήθεια του διερμηνευτή της Python ας παίξουμε λίγο με τις μεταβλητές και ας δούμε τι ακριβώς συμβαίνει με τους τύπους, τις εκχωρήσεις αλλά και το χώρο που αυτές καταλαμβάνουν στη μνήμη. Είναι και μια ευκαιρία να εξοικειωθούμε λίγο περισσότερο με τον διερμηνευτή.

```
>>> import sys
>>> a=1
>>> a
1
>>> type(a)
<class 'int'>
>>> sys.getsizeof(a)
28
>>>
```

Σχήμα 4.2: Ακέραιες μεταβλητές.

Στο Σχήμα 4.2 αρχικοποιούμε τη μεταβλητή **a** στην τιμή 1. Αμέσως μετά ζητάμε να μας πληροφορήσει ο διερμηνευτής για την τιμή της **a**, οπότε παίρνουμε την απάντηση **1**. Στη συνέχεια ρωτάμε ποιος είναι ο τύπος του **a**. Η απάντηση είναι ότι το **a** είναι μεταβλητή τύπου **int (ακέραιου)**. Η **a** έγινε τύπου **ακέραιου** ακριβώς όταν της εκχωρήθηκε η τιμή 1.

Τέλος, ρωτάμε πόσο χώρο καταλαμβάνει η **a** στη μνήμη. Για το σκοπό αυτό χρησιμοποιούμε τη συνάρτηση **getsizeof** η οποία μας επιστρέφει αυτό ακριβώς το πράγμα. Προσοχή όμως, η **getsizeof** δεν είναι ενσωματωμένη συνάρτηση της Python και για να τη χρησιμοποιήσουμε θα πρέπει να κάνουμε **import** το **module sys**. Δείτε πως γίνεται αυτό στο Σχήμα 4.2 Το αποτέλεσμα μας πληροφορεί ότι για τη μεταβλητή **a** χρειαζόμαστε 28 bytes.

Όχι και λίγα, έτσι; Θυμηθείτε ότι είχαμε χρειαστεί μόλις 23 bytes στη C για να αποθηκεύσουμε εκείνες τις πέντε μεταβλητές (δες στο Σχήμα 4.1). Έτσι, η ευκολία που μας παρέχει η Python και η ευελιξία της δεν είναι χωρίς τίμημα. Μεταφράζεται και σε κόστος χώρου στην αποθήκευση αλλά και σε κόστος χρόνου, διότι η πρόσβαση αλλά γενικότερα η διαχείριση των μεταβλητών γίνεται πολύ πιο ακριβή.

Πρέπει επίσης να σημειώσουμε ότι ο χώρος που καταλαμβάνει μία μεταβλητή στη μνήμη εξαρτάται και από την υλοποίηση της Python. Δεν είναι δηλαδή ίδιος για όλους τους διερμηνευτές της γλώσσας. Δεν είναι επίσης ίδιος για όλες τις τιμές που παίρνει η μεταβλητή. Ας συνεχίσουμε λίγο τις ερωταπαντήσεις με τον διερμηνευτή. Στο Σχήμα 4.3 αρχικά αλλάζουμε την τιμή της **a** και προσθέτουμε 10 μηδενικά μετά τον άσσο, ώστε να γίνει ένας πολύ μεγάλος ακέραιος. Παρότι ο τύπος της μεταβλητής εξακολουθεί να είναι **int**, ο

χώρος που καταλαμβάνει είναι τώρα 32 bytes.

```
>>> a=10000000000
>>> type(a)
<class 'int'>
>>> sys.getsizeof(a)
32
>>> a=3.14
>>> type(a)
<class 'float'>
>>> sys.getsizeof(a)
24
>>> a=30000000000.14
>>> type(a)
<class 'float'>
>>> sys.getsizeof(a)
24
>>>
```

Σχήμα 4.3: Χώρος που καταλαμβάνουν στη μνήμη ακέραιοι αριθμοί διαφορετικού μεγέθους.

Στη συνέχεια εκχωρούμε στην **a** έναν πραγματικό αριθμό. Ενώ σε άλλες γλώσσες αυτό θα ήταν τελείως παράνομο, στην Python επιτρέπεται χωρίς να δημιουργείται κανένα πρόβλημα. Τώρα παρατηρούμε ότι η μεταβλητή έγινε τύπου **float**. Ο χώρος που καταλαμβάνει στη μνήμη έγινε τώρα 24 bytes. Ακόμα και όταν μεγαλώσαμε αρκετά τον αριθμό ο χώρος στη μνήμη δεν άλλαξε. Μη σας παραξενεύει, έχει να κάνει με τον τρόπο που κωδικοποιείται ο αριθμός στη μνήμη.

```
>>> a='1'
>>> type(a)
<class 'str'>
>>> sys.getsizeof(a)
50
>>> a='12'
>>> sys.getsizeof(a)
51
>>> a='123'
>>> sys.getsizeof(a)
52
>>>
```

Σχήμα 4.4: Παράδειγμα συμβολοσειρών.

Ας συνεχίσουμε ρίχνοντας και μια ματιά στα αλφαριθμητικά. Ας αφήσουμε απ' έξω τις λίστες και γενικότερα τις πιο πολύπλοκες δομές. Θα τις δούμε έτσι

και αλλιώς αναλυτικότερα παρακάτω, σε επόμενο κεφάλαιο, αφού παρουσιάζουν ιδιαίτερο ενδιαφέρον.

Τα **αλφαριθμητικά** είναι μεταβλητές οι οποίες έχουν τη δυνατότητα να αποθηκεύσουν συμβολοσειρές που μπορεί να αποτελούνται από γράμματα, αριθμούς ή και άλλα σύμβολα. Τις συμβολοσειρές τις περικλείουμε μέσα σε εισαγωγικά (μονά ή διπλά) ώστε να ξεχωρίσουν από οτιδήποτε άλλο. Για παράδειγμα, το **’test’** (το οποίο είναι ακριβώς το ίδιο με το **”test”**) είναι μία συμβολοσειρά που αποτελείται από το χαρακτήρα **t**, στη συνέχεια τον χαρακτήρα **e**, στην συνέχεια τον χαρακτήρα **s** και τελειώνει με τον χαρακτήρα **t**. Η Python δεν καταλαβαίνει τίποτε παραπάνω από αυτό που περιέγραψα. Αν όμως χρησιμοποιήσουμε το **test** χωρίς εισαγωγικά, τότε η Python αντιλαμβάνεται τη μεταβλητή **test**, θα την αποτιμήσει και θα χρησιμοποιήσει την τιμή της. Ένα σχόλιο πάνω σε αυτό είχαμε δει, δειλά δειλά και πρώιμα, στο παράδειγμα του τριωνύμου στο κεφάλαιο με τους αλγορίθμους και τα διαγράμματα ροής. Στο Σχήμα 4.4 αλλάζουμε και πάλι τον τύπο της **a**, αφού τώρα του εκχωρούμε ένα αλφαριθμητικό. Παρατηρήστε ότι για ένα αλφαριθμητικό μήκους 1 χρειάζονται 50 bytes για αποθήκευση στην Python που χρησιμοποιώ, ενώ για κάθε έναν χαρακτήρα που προσθέτουμε ο χώρος αποθήκευσης αυξάνεται κατά 1.

```
>>> b=True
>>> type(b)
<class 'bool'>
>>> a=12
>>> x=a<6
>>> x
False
>>> a=5+3j
>>> type(a)
<class 'complex'>
>>> a
(5+3j)
>>>
```

Σχήμα 4.5: Παράδειγμα λογικών και μιγαδικών μεταβλητών.

Οι λογικές μεταβλητές είναι μεταβλητές οι οποίες παίρνουν δύο τιμές **True** και **False**. Έτσι, είναι δυνατόν να ορίσουμε απευθείας **x=True**, ή να γίνει αυτό μέσα από μία λογική σύγκριση, π.χ. **x=a<6**, όπου το **x** θα πάρει την τιμή **True** αν πράγματι το **a** είναι μικρότερο του 6 και **False** σε κάθε άλλη περίπτωση. Θα δούμε τη χρησιμότητά των λογικών μεταβλητών και εκφράσεων παρακάτω όταν και θα σχηματίσουμε εκφράσεις με αυτές, αλλά και στο επόμενο κεφάλαιο, στις δομές ελέγχου.

Οι μιγαδικές μεταβλητές παίρνουν τιμές μιγαδικές. Είναι, δηλαδή, νόμιμο να ορίσουμε ότι **a=5+3j** όπως και ότι **b=2-3j** και η Python το αντιλαμβάνεται με τον τρόπο που επιθυμούμε. Μπορούμε δηλαδή να κάνουμε με αυτές οτιδήποτε μπορούμε να κάνουμε και με μία αριθμητική μεταβλητή, αρκεί βέβαια μαθηματικά να έχει νόημα. Θα δούμε περισσότερα πάλι παρακάτω στις αριθμητικές εκφράσεις. Στο μεταξύ, στο Σχήμα 4.5 δείτε παραδείγματα ορισμού τόσο λογικών μεταβλητών, όσο και μιγαδικών.

4.3 Εκφράσεις

Οι **εκφράσεις** μιας γλώσσας είναι συνδυασμός τελεστών και μεταβλητών, η αποτίμηση των οποίων δημιουργεί μία νέα τιμή η οποία μπορεί να εκχωρηθεί σε μία μεταβλητή ή να χρησιμοποιηθεί σαν μεταβλητή.

Οι τελεστές ανάλογα με τις μεταβλητές πάνω στις οποίες εφαρμόζονται αποκτούν και διαφορετική σημασία. Κάποιος τελεστής μπορεί ακόμα να έχει νόημα να εφαρμοστεί σε μία μεταβλητή κάποιου τύπου αλλά να μην έχει νόημα να εφαρμοστεί στις μεταβλητές άλλου τύπου.

Οι σημαντικότεροι τελεστές που εφαρμόζονται πάνω σε αριθμητικές μεταβλητές είναι οι ακόλουθοι:

- **μοναδιαίοι:** +, -
- **προσθετικοί** +, -
- **πολλαπλασιαστικοί** *, /, //
- διάφοροι άλλοι

Οι **μοναδιαίοι** τελεστές είναι αυτοί που χρησιμοποιούμε σαν πρόσημα. Οι **προσθετικοί** είναι αυτοί που χρησιμοποιούμε για τις πράξεις της πρόσθεσης και της αφαίρεσης. Οι **πολλαπλασιαστικοί** χρησιμοποιούνται για τον πολλαπλασιασμό και τη διαίρεση. Το σύμβολο // υποδηλώνει ακέραια διαίρεση (στην Python 3) Στους διάφορους ανήκουν μεταξύ άλλων το υπόλοιπο διαίρεσης (συμβολίζεται με **%** στην Python, είναι διαδεδομένος συμβολισμός και χρησιμοποιείται από πολλές γλώσσες) και η ύψωση σε δύναμη (συμβολίζεται με **).

Για την προτεραιότητα των πράξεων ακολουθείται η συνήθης πρακτική. Μεγαλύτερη προτεραιότητα έχουν οι τελεστές υπολοίπου **%** και ύψωσης σε δύναμη **. Μετά ακολουθούν οι πολλαπλασιαστικοί τελεστές. Τέλος, με μικρότερη προτεραιότητα έχουμε τους προσθετικούς (συμπεριλαμβανομένων των μοναδιαίων, δηλαδή των προσήμων). Η αποτίμηση των τελεστών με την ίδια

προτεραιότητα γίνεται από τα αριστερά στα δεξιά (έχει σημασία αυτό στην ακέραια διαίρεση). Αν θέλουμε να αλλάξουμε τη σειρά εκτέλεσης των πράξεων μπορούμε να το κάνουμε χρησιμοποιώντας παρενθέσεις, ακριβώς όπως στα μαθηματικά, οι οποίες έχουν και τη μέγιστη προτεραιότητα.

```
>>> a=3.16
>>> b=2.1
>>> c=3*(a+b)**2+5
>>> c
88.0028
>>> 3/5
0.6
>>> 3//5
0
>>> 5//2*3
6
>>> 5*2//3
3
```

Σχήμα 4.6: Παραδείγματα αριθμητικών εκφράσεων.

Ας δούμε τώρα παραδείγματα για όλα αυτά στο Σχήμα 4.6. Στην αρχή, ορίζουμε δύο μεταβλητές **a** και **b** και από αυτές υπολογίζουμε τη **c**. Θα υπολογίσουμε, δηλαδή, την παράσταση:

$$c = 3(a+b)^2 + 5$$

Πρώτα εκτελείται η πράξη μέσα στην παρένθεση (μέγιστη προτεραιότητα), στη συνέχεια η ύψωση σε δύναμη (αμέσως μικρότερη προτεραιότητα), μετά ο πολλαπλασιασμός με το 3 και τέλος η πρόσθεση του 5 (η πιο μικρή προτεραιότητα από όλες).

Στο ίδιο Σχήμα ακολουθούν παραδείγματα με την ακέραια διαίρεση. Το **3/5** θα δώσει 0.6 που είναι το αποτέλεσμα της διαίρεσης του 3 με το 5. Αντίθετα, το **3//5** θα δώσει αποτέλεσμα 0, αφού το πηλίκο της διαίρεσης του 3 με το 5 είναι μηδέν. Το υπόλοιπο της διαίρεσης του 3 με το 5 (που είναι 0) θα το παίρναμε γράφοντας **3\\5**. Στα δύο τελευταία παραδείγματα του Σχήματος παρατηρήστε ότι στην έκφραση **5//2*3** εκτελείται πρώτα η ακέραια διαίρεση, ενώ στην έκφραση **5*2//3** εκτελείται πρώτα ο πολλαπλασιασμός.

```
>>> s1='1234'
>>> s2='abc'
>>> s3=s1+2*s2+'!!!'
>>> s3
'1234abcabc!!!'
>>>
```

Σχήμα 4.7: Παραδείγματα εκφράσεων με αλφαριθμητικά.

Πάμε τώρα να δούμε ένα παράδειγμα με τα αλφαριθμητικά. Δεν θα επεκταθούμε πολύ διότι σε επόμενο κεφάλαιο θα δούμε αναλυτικά τα αλφαριθμητικά μαζί με άλλες ακολουθίες (ακολουθίες; δεν ξέρουμε τι είναι ακόμα). Θα ασχοληθούμε μόνο με ένα παράδειγμα στο οποίο οι τελεστές αλλάζουν λειτουργικότητα όταν εφαρμόζονται σε διαφορετικού τύπου δεδομένα. Στο Σχήμα 4.7 παρατηρήστε ότι στο σύμβολο **+** έχει πάρει το ρόλο της παράθεσης (τοποθετεί το δεύτερο όρισμα μετά το πρώτο) ενώ το σύμβολο * επαναλαμβάνει το δεύτερο του τελούμενο (το **s1** στο παράδειγμα) όσες φορές λέει το αριστερό του (το 2 στο παράδειγμα).

```
>>> a=3+5j
>>> b=2-3j
>>> a+b
(5+2j)
>>> a*b
(21+1j)
>>> c=4-j
Traceback (most recent call last):
  File "<pyshell#150>", line 1, in <module>
    c=4-j
NameError: name 'j' is not defined
>>> c=4-1j
>>>
```

Σχήμα 4.8: Παραδείγματα εκφράσεων με μιγαδικούς αριθμούς.

Στους μιγαδικούς αριθμούς τα σύμβολα αυτά αποκτούν πάλι διαφορετική λειτουργικότητα, αφού μετατρέπονται σε σύμβολα πράξεων μιγαδικών αριθμών. Δείτε ένα παράδειγμα πρόσθεσης και πολλαπλασιασμού στο Σχήμα 4.8.

Στο ίδιο Σχήμα δείτε και ένα παράδειγμα σφάλματος το οποίο με την πρώτη ματιά ίσως μας παραξενέψει. Γιατί δεν καταλαβαίνει το **4-j** και θέλει να το γράψουμε **4-1j**, το οποίο βέβαια είναι τουλάχιστον άκομψο; Η απάντηση είναι απλή. Ο διερμηνευτής δεν είναι σε θέση να διακρίνει αν γράφοντας **4-j** εμείς εννοούμε ότι το **j** είναι το σύμβολο που χρησιμοποιούμε για το φανταστικό μέρος

ενός μιγαδικού, άρα ολόκληρος ο **4-j** είναι ένας μιγαδικός αριθμός, ή εάν εμείς απλά θέλουμε να αφαιρέσουμε τη μεταβλητή **j** από το **4**. Εδώ, δηλαδή, έχουμε αμφισημία. Τι είναι καλύτερο; Κάτι λιγότερο κομψό, ή κάτι διφορούμενο; Ρητορική ερώτηση.

4.4 Λογικές εκφράσεις

Μία λογική μεταβλητή μπορεί να πάρει τις τιμές **True** (σημαίνει **αληθής**, ότι η λογική παράσταση που αποτιμήθηκε σε αυτό ισχύει) ή **False** (σημαίνει **ψευδής**, ότι η λογική παράσταση που αποτιμήθηκε σε αυτό δεν ισχύει). Μία λογική έκφραση είναι, σε αναλογία με την αριθμητική έκφραση, ένας συνδυασμός λογικών μεταβλητών και λογικών τελεστών. Οι λογικοί τελεστές είναι διαφορετικοί από τους αριθμητικούς, μιας που ο σκοπός τους είναι να φτάσουμε σε μία απόφαση (**True** ή **False**) και όχι να υπολογίσουμε μία ποσότητα (αν βέβαια θέλουμε να είμαστε αυστηροί, και αυτό ποσότητα είναι, αφού είναι δυνατόν να εκχωρείται - αλλά δεν θέλουμε να είμαστε αυστηροί).

Για παράδειγμα, η παράσταση **A>B** είναι μία λογική έκφραση που παίρνει την τιμή **True** αν το **A** είναι μεγαλύτερο του **B** και **False** σε κάθε άλλη περίπτωση. Ακολουθον οι πιο συχνά χρησιμοποιούμενοι τελεστές για τον σχηματισμό λογικών εκφράσεων, όπως χρησιμοποιούνται στη γλώσσα Python. Παρόμοιοι τελεστές υπάρχουν και στις άλλες γλώσσες.

- **==**: ισότητα
- **<**: μικρότερο
- **>**: μεγαλύτερο
- **<=**: μικρότερο ή ίσο
- **>=**: μεγαλύτερο ή ίσο
- **!=**: διάφορο

Οι λογικές εκφράσεις που σχηματίζονται με τους τελεστές αυτούς είναι μάλλον απλές και πολλές φορές υπάρχει η ανάγκη για κάτι πιο πολύπλοκο. Έτσι, υπάρχουν ακόμα τελεστές οι οποίοι μπορούν να χρησιμοποιηθούν για λογική σύζευξη, λογική διάζευξη ή λογική άρνηση, όπως ακριβώς χρησιμοποιούνται στην καθημερινή μας ζωή.

- **not**: **not A==True** εάν **A==False**
- **and**: **A and B==True** εάν **A==True** και **B==True**
- **or**: **A or B==True** εάν **A==True** ή **B==True**

Ισχύουν και πάλι κανόνες προτεραιότητας με το **not** να έχει τη μεγαλύτερη, το **and** να ακολουθεί και το **or** να έχει τη μικρότερη. Με τις παρενθέσεις μπορούμε να μεταβάλλουμε την προτεραιότητα όπως επιθυμούμε. Ας δούμε δύο παραδείγματα:

- **A<4 and B>3** είναι αληθές όταν και το **A** είναι μικρότερο του 4 αλλά και το **B** μεγαλύτερο του 3, και τα δύο δηλαδή πρέπει να ισχύουν.
- **not(A<4 or B>3)** η έκφραση μέσα στην παρένθεση είναι αληθής όταν ή το **A** είναι μικρότερο του 4 ή το **B** μεγαλύτερο του 3, αρκεί δηλαδή να ισχύει το ένα από τα δύο. Όμως το **not** αντιστρέφει ότι ισχύει μέσα στην παρένθεση.

4.5 Είσοδος και έξοδος

Κάθε γλώσσα προγραμματισμού παρέχει εντολές με τις οποίες ένα πρόγραμμα επικοινωνεί με τον έξω κόσμο. **Είσοδο δεδομένων** έχουμε όταν ο υπολογιστής ζητάει από τον χρήστη να του δώσει τιμή για κάποια μεταβλητή (π.χ. έναν από τους συντελεστές του τριωνύμου). **Έξοδο δεδομένων** έχουμε όταν ο υπολογιστής πληροφορεί για κάτι τον χρήστη (π.χ. τυπώνει ένα μήνυμα στην οθόνη ή την τιμή μιας μεταβλητής).

Η είσοδος και η έξοδος δεδομένων έχουν περιγραφεί σχετικά αναλυτικά στο παράδειγμα της πρόσθεσης αλλά και στο παράδειγμα του τριωνύμου. Οι εντολές εισόδου-εξόδου διαφέρουν συνήθως αρκετά από γλώσσα σε γλώσσα, αλλά η γενικότερη φιλοσοφία δεν μπορεί παρά να είναι ίδια. Ας δούμε τι συμβαίνει στην Python.

4.5.1 Η εντολή input

Στην Python η είσοδος δεδομένων από το πληκτρολόγιο γίνεται με την **input**. Στο Σχήμα 4.9 έχουμε ένα παράδειγμα χρήσης της **input**. Ας βασιστούμε σε αυτό για να καταλάβουμε τη χρήση της. Αρχικά γράφουμε:

```
x=input('Give me the value of x: ')
```

Ας δούμε λίγο τι σημαίνει αυτό. Ο διερμηνευτής θα εμφανίσει στην οθόνη το μήνυμα **Give me the value of x:** και θα περιμένει από εμάς να δώσουμε τιμή για το **x**. Ας δώσουμε **34**.

```
>>> x=input('Give me the value of x: ')
Give me the value of x: 34
>>> x
'34'
>>> 2*x
'3434'
>>> x=int(input('Give me the value of x: '))
Give me the value of x: 34
>>> x
34
>>> 2*x
68
>>> x=int(input('Give me the value of x: '))
Give me the value of x: 3.4
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    x=int(input('Give me the value of x: '))
ValueError: invalid literal for int() with base 10: '3.4'
>>>
```

Σχήμα 4.9: Παραδείγματα χρήσης της input.

Παρατηρώντας όμως στο Σχήμα 4.9 τι μας επέστρεψε ο διερμηνευτής όταν του ζητήσαμε την τιμή του **x**, θα δούμε ότι αυτό που μας επέστρεψε είναι κλεισμένο μέσα σε εισαγωγικά. Αν μάλιστα του ζητήσουμε να μας επιστρέψει το διπλάσιό του, θα μας επιστρέψει **3434**. Τώρα ξέρουμε τι συμβαίνει. Το **34** διαβάστηκε σαν συμβολοσειρά. Η λύση είναι να μετατρέψουμε τη συμβολοσειρά σε ακέραιο. Αυτό γίνεται ως εξής:

```
x=int(input('Give me the value of x: '))
```

Δείτε στο ίδιο σχήμα ότι αυτό έχει τα αναμενόμενα αποτελέσματα, αφού όταν του ζητήσουμε το **x** μάς απαντά πράγματι **34**, αλλά και μπορεί να το χρησιμοποιήσει σαν ακέραιο, διπλασιάζοντάς το όταν το πολλαπλασιάζουμε με το 2.

Αν βέβαια αντί για **34** δώσουμε **3.4**, τότε έχουμε πρόβλημα, αφού ο διερμηνευτής θα διαβάσει μεν τη συμβολοσειρά **3.4**, αλλά όταν πάει να την μετατρέψει σε ακέραιο θα αποτύχει.

Όλα τα παραπάνω ισχύουν στην Python 3. Η Python 2 είναι αρκετά διαφορετική στο σημείο αυτό. Προσέξτε λίγο μην μπερδευτείτε.

4.5.2 Η εντολή print

Η εμφάνιση αλφαριθμητικών και τιμών μεταβλητών στην οθόνη γίνεται με την **print**. Εδώ οι επιλογές μας είναι πολύ περισσότερες. Θα δούμε με ποιο

τρόπο είναι δυνατόν να εμφανίσουμε απλά στην οθόνη αποτελέσματα. Από όλους τους τρόπους που μπορεί να το πετύχει αυτό η Python, εμείς θα χρησιμοποιήσουμε έναν που μοιάζει περισσότερο με τον τρόπο που υποστηρίζει την εμφάνιση δεδομένων στην οθόνη η γλώσσα C. Εάν γράψουμε, λοιπόν **print(s)**, θα τυπωθεί στην οθόνη η τιμή του **s**.

Αν τώρα θέλουμε περισσότερη ευελιξία στο τι τυπώνουμε, μπορούμε να χρησιμοποιήσουμε την print με κάποιες παραμέτρους. Έτσι, ο παρακάτω κώδικας:

```
print("The root is x=%g"%x)
```

σημαίνει ότι θα τυπωθεί το μήνυμα

```
The root is x=...
```

όπου οι τελείες θα αντικατασταθούν από την τιμή της μεταβλητής **x**. Το σύμβολο **%g** σημαίνει ότι στο σημείο εκείνο αναμένεται να εμφανιστεί ένας πραγματικός αριθμός, και μάλιστα, όπως δηλώνεται παρακάτω με το σύμβολο **%x**, πρόκειται για την τιμή της μεταβλητής **x** που είναι πραγματική. Αν το **x** ήταν ακέραιος, η **print** θα λειτουργούσε κανονικά, αφού το σύνολο των ακέραιων είναι υποσύνολο του συνόλου των πραγματικών, αν όμως ήταν συμβολοσειρά, θα εμφανιζόταν μήνυμα λάθους.

Εκτός από το σύμβολο **%g**, έχουμε τα σύμβολα **%d** για ακέραιο και **%s** για συμβολοσειρά.

Οι απαραίτητοι έλεγχοι συμβατότητας των συμβόλων αυτών με τις μεταβλητές που τους αντιστοιχίζονται γίνονται πάντοτε και προκύπτουν οι αναγκαίες στρογγυλοποιήσεις ή τα μηνύματα σφαλμάτων όπου είναι απαραίτητο.

Στο Σχήμα 4.10 μπορείτε να δείτε ένα παράδειγμα χρήσης της **print**. Ορίζονται το πραγματικό και το φανταστικό μέρος ενός μιγαδικού αριθμού και τυπώνεται ο μιγαδικός αριθμός και ο συζυγής του.

```
>>> xReal=3
>>> xIm=2
>>> print("The roots are x1=%g+%gi
              and x2=%g-%gi"%(xReal,xIm,xReal,xIm))
The roots are x1=3+2i and x2=3-2i
```

Σχήμα 4.10: Παράδειγμα χρήσης της print.

Αν θέλετε να διαβάσετε περισσότερο υλικό που σχετίζεται με το κεφάλαιο αυτό, μπορείτε να ανατρέξετε στο κεφάλαιο 2 του βιβλίου [1], στο κεφάλαιο 4 του βιβλίου [2] και στο κεφάλαιο 3 του βιβλίου [3].

Ασκήσεις που μπορείτε να κάνετε μόνοι σας

- Ένα πρόγραμμα μαθητολογίου κρατά πληροφορίες για κάθε μαθητή (όνομα, έτος γέννησης, διεύθυνση κλπ) καθώς και για την επίδοσή του σε διάφορα μαθήματα. Σχεδιάστε στο μυαλό σας πώς θα έπρεπε να είναι ένα πρόγραμμα μαθητολογίου και ορίστε τις μεταβλητές που θα χρειαστείτε για να κρατήσετε πληροφορίες που αφορούν έναν μαθητή καθώς και τον τύπο των μεταβλητών αυτών (π.χ. η χρονολογία γέννησης είναι ακέραιος αριθμός).
- Αποτιμήστε τη λογική έκφραση: **y= (not A) and B or C** για κάθε πιθανό συνδυασμό τιμών των λογικών μεταβλητών **A**, **B** και **C** και συγκεντρώστε τα αποτελέσματά σας σε έναν πίνακα. Για να βοηθηθείτε στη σκέψη σας, σημειώστε ότι ο πίνακας πρέπει να έχει 8 γραμμές και τέσσερις στήλες. Κάθε στήλη αντιστοιχεί σε μία από τις μεταβλητές **A**, **B**, **C** και **y**.
- Για την ίδια έκφραση, κατασκευάστε ένα διάγραμμα ροής το οποίο να δέχεται ως είσοδο τις τιμές των **A**, **B** και **C** και να υπολογίζει την τιμή του **y**.
- Γράψτε σε γλώσσα Python το τμήμα του παραπάνω διαγράμματος ροής που αντιστοιχεί στην είσοδο και στην έξοδο των δεδομένων.

Βιβλιογραφία

1. Allen B. Downey (2012). **Think Python**. Publisher: O'Reilly Media.
2. Ellis Horowitz (1993). **Βασικές Αρχές Γλωσσών Προγραμματισμού**. 2η έκδοση, Εκδόσεις Κλειδάριθμος.
3. Cody Jackson (2011). **Learning to Program Using Python**. Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.

Κεφάλαιο 5: Δομές δεδομένων

Η έννοια της **δομής δεδομένων (data structure)** είναι πολύ ευρεία στην επιστήμη της Πληροφορικής. Με τον όρο δομή δεδομένων εννοούμε τον τρόπο αποθήκευσης δεδομένων, οργανωμένα και βασιζόμενα σε πιο απλούς τύπους δεδομένων. Σκοπός είναι να επιτύχουμε όσο το δυνατόν καλύτερο τρόπο οργάνωσης και αποθήκευσής τους, ώστε αυτά να μπορούν να χρησιμοποιηθούν αποδοτικά..

Αν και δεν μπορούμε να πούμε ότι οι περισσότερο γνωστές δομές δεδομένων υποστηρίζονται από όλες τις γλώσσες προγραμματισμού, οι περισσότερες γλώσσες έχουν επιλέξει να υποστηρίξουν ένα ικανό σύνολο δομών ώστε να διευκολύνουν τον προγραμματιστή.

Γνωστές δομές δεδομένων είναι οι **εγγραφές (records)**, οι **πίνακες (arrays)**, οι **λίστες (lists)** και τα **δέντρα (trees)**. Τα **αλφαριθμητικά (strings)** μπορούν επίσης να ενταχθούν στην κατηγορία αυτήν, αφού αποτελούν μία οργάνωση χαρακτήρων.

Η Python βασίζεται πολύ σε λίστες. Με έκπληξη θα διαπίστωνε κανείς ότι η Python δεν υποστηρίζει πίνακες. Υποκαθιστά τους πίνακες με δομές που βασίζονται στις λίστες. Αν κανείς θελήσει να χρησιμοποιήσει πίνακες, χωρίς ουσιαστικά να διαχειρίζεται λίστες, θα πρέπει να καταφύγει σε βιβλιοθήκες. Ο μεγάλος αριθμός βιβλιοθηκών που είναι διαθέσιμος για την Python μάς λύνει πολλές φορές τα χέρια. Ακόμα, η Python υποστηρίζει **πλειάδες (tuples)**, **σύνολα (sets)** και **λεξικά (dictionaries)**.

Θα τα δούμε όλα αναλυτικά. Αλλά κατά την προσφιλή μας συνήθεια, δεν θα περιοριστούμε στις δομές τις Python. Θα τις παρουσιάσουμε έστω και εν συντομία όλες και θα εστιάσουμε σε αυτές που υποστηρίζει η Python, μελετώντας τις και εμπεδώνοντάς τις με παραδείγματα.

5.1 Εγγραφές

Μια από τις πιο απλές δομές δεδομένων είναι οι εγγραφές. Ουσιαστικά, πρόκειται για μια ομαδοποίηση απλούστερων δομών που έχουν μεταξύ τους μια λογική σύνδεση. Για παράδειγμα, σε μια γλώσσα όπου έχουμε δομές για ακέραιους, πραγματικούς και συμβολοσειρές, μπορούμε σε μία εφαρμογή μαθητολογίου να συνδυάσουμε τις δομές αυτές και να φτιάξουμε μια νέα δομή που θα την ονομάζουμε **student** και θα αποτελείται από έναν ακέραιο, ο οποίος θα είναι ο κωδικός του φοιτητή, μία συμβολοσειρά που θα είναι το ονοματεπώνυμο, μία συμβολοσειρά που θα είναι η διεύθυνση κατοικίας του, έναν ακέραιο, που θα είναι η τάξη του, έναν πίνακα από 10 ακέραιους, που θα είναι οι βαθμοί του στα δέκα μαθήματα που διδάσκεται, και έναν πραγματικό, που θα είναι ο μέσος όρος του.

```
1  Type
2      student = Record
3      code: Integer;
4      nam: String[25];
5      address: String[25];
6      marks: Array[1..10] of Integer;
7      average: Real;
8      End;
9
10 Var
11     s: student;
12
13 Begin
14     Readln(s.code);
15     Writeln(s.code);
16 End.
```

Σχήμα 5.1: Παράδειγμα εγγραφών στην Pascal.

Η Python δεν υποστηρίζει εγγραφές. Ο πιο εύκολος τρόπος για να τις υλοποιήσει κανείς είναι μέσα από το μηχανισμό των κλάσεων στον αντικειμενοστραφή προγραμματισμό. Αφού η Python δεν υποστηρίζει εγγραφές, θα χρησιμοποιήσουμε Pascal για να δούμε ένα παράδειγμα.

Στο Σχήμα 5.1 φαίνεται ένα τέτοιο παράδειγμα για το πρόγραμμα με το μαθητολόγιο που αναφέραμε. Ορίζουμε μια εγγραφή, τη **student**, και μέσα σε αυτήν τα πεδία **code**, που είναι ένας ακέραιος αριθμός, για να κρατά τον κωδικό του μαθητή, δύο συμβολοσειρές, τις **nam** και **address**, μήκους 25 χαρακτήρων η καθεμία, για να αποθηκευτεί το ονοματεπώνυμο και η διεύθυνσή του μαθητή αντίστοιχα, έναν πίνακα 10 ακέραιων, που τον ονομάσαμε **marks**, για τη βαθμολογία του (δεν είπαμε τι είναι ο πίνακας, όποιον δυσκολεύει η έννοια ας την

αγνοήσει προς το παρόν), και τέλος έναν πραγματικό αριθμό, τον **average**, για τον μέσο όρο της βαθμολογίας.

Παρακάτω δηλώνουμε ότι το **s** είναι τύπου **student**. Αυτό σημαίνει ότι μπορούμε να διαβάσουμε και να γράψουμε τα **s.code**, **s.nam** κ.ο.κ. σαν να ήταν απλές μεταβλητές. Με την ομαδοποίησή τους στο **s** τα οργανώσαμε σε μία ομπρέλα, αφού συνδέονται λογικά με κοινό παρονομαστή τον μαθητή. Στο παράδειγμα, απλά διαβάζουμε και γράφουμε στο **s.code**, ενώ στην πραγματικότητα έχουμε πλήρη ευελιξία με όλα τα πεδία για να τα χειριστούμε όπως επιθυμούμε.

Η **student** όχι μόνο αποτελείται από πιο απλές δομές, αλλά μπορεί να συμμετέχει και στη δημιουργία περισσότερο σύνθετων, πιο πολύπλοκων εγγραφών για παράδειγμα.

5.2 Πλειάδες

Οι πλειάδες συναντώνται στην Python. Δεν υποκαθιστούν τις εγγραφές, αλλά μοιάζουν λίγο με αυτές. Ενώ οι εγγραφές είναι τύποι δεδομένων μέσα από τους οποίους δηλώνουμε μεταβλητές και τις μεταβλητές αυτές στη συνέχεια χρησιμοποιούμε, οι πλειάδες είναι μία ομαδοποίηση μεταβλητών ή δομών, χωρίς όμως να ανήκουν σε κάποιο τύπο δεδομένων.

```
>>> x=333,'Γιώργος Μανής','Νεοκαισάρεια',4,3.3
>>> x
(333, 'Γιώργος Μανής', 'Νεοκαισάρεια', 4, 3.3)
>>> x[1]
'Γιώργος Μανής'
>>> y=333,
>>> y
(333,)
>>> x[1]='σφάλμα'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    x[1]='σφάλμα'
TypeError: 'tuple' object does not support item assignment
>>>
```

Σχήμα 5.2: Παραδείγματα με πλειάδες.

Ίσως με ένα παράδειγμα να αποδειχθεί πολύ ευκολότερο από ό,τι μοιάζει να είναι παραπάνω. Ένας μαθητής, λοιπόν, θα μπορούσε να είναι η πλειάδα που φαίνεται στο Σχήμα 5.2. Αποτελείται από τον κωδικό, το ονοματεπώνυμο, τη διεύθυνση, την τάξη και τον μέσο όρο της βαθμολογίας (ο οποίος είναι και μικρός, πρέπει να διαβάζει μάλλον λίγο περισσότερο).

Εκχωρήσαμε την πλειάδα στη μεταβλητή **x**, όπως θα μπορούσαμε να κάνουμε και σε μία εγγραφή. Γράφοντας **x[1]** μπορούμε να προσπελάσουμε το στοιχείο της πλειάδας που βρίσκεται στη θέση 1. Η αρίθμηση ξεκινάει από το 0. Έτσι, **x[0]** είναι ο κωδικός, **x[1]** το ονοματεπώνυμο, **x[2]** η διεύθυνση κ.ο.κ.

Αν τώρα θέλουμε να ορίσουμε μία πλειάδα με ένα μόνο στοιχείο, πρέπει μετά το στοιχείο αυτό να βάλουμε ένα κόμμα, όπως φαίνεται στη μεταβλητή **y** του Σχήματος 5.2. Αν δεν βάζαμε το κόμμα μετά το 333, ο διερμηνευτής θα εκλάμβανε το **y** ως ακέραια μεταβλητή και όχι σαν πλειάδα.

Τέλος, δείτε τι θα συμβεί εάν προσπαθήσω να τροποποιήσω το όνομα του μαθητή. Ο διερμηνευτής δεν με αφήνει να το κάνω και μου επιστρέφει μήνυμα λάθους.

Ο λόγος είναι ότι οι πλειάδες δεν τροποποιούνται! Παράξενο; Ναι, αλλά έτσι είναι. Αν θέλουμε να φτιάξουμε μία δομή που να έχει λειτουργία παρόμοια με αυτήν των πλειάδων αλλά να μπορούμε να τροποποιούμε τα στοιχεία της, πρέπει να καταφύγουμε στις λίστες. Θα τις δούμε παρακάτω, αλλά ας μιλήσουμε πρώτα για αλφαριθμητικά.

5.3 Αλφαριθμητικά

Αλφαριθμητικά (strings) λέγονται ακολουθίες από χαρακτήρες, ψηφία ή άλλα σύμβολα. Για να ξεχωρίσουν από τις μεταβλητές, τις λέξεις-κλειδιά της γλώσσας και άλλα σύμβολα, τα περικλείουμε μέσα σε εισαγωγικά. Στην Python είναι επιτρεπτό να χρησιμοποιήσουμε τόσο τα μονά (**'...'**) όσο και τα διπλά εισαγωγικά (**"..."**) για το σκοπό αυτό.

Παραδείγματα από αλφαριθμητικά φαίνονται στο Σχήμα 5.3. Στα **s** και **t** εκχωρούνται αντίστοιχα οι τιμές **abc** και **123**. Οι διάφοροι τελεστές που είδαμε προηγουμένως αποκτούν άλλη έννοια τώρα. Το σύμβολο **+** γίνεται παράθεση, οπότε η πράξη **s+t** θα επιστρέψει το αλφαριθμητικό **s** ακολουθούμενο από το **t** (**abc123**), ενώ η έκφραση **3*t+2*s** θα κάνει παράθεση το **t** με τον εαυτό του τρεις φορές, το **s** με τον εαυτό του δύο φορές και στη συνέχεια θα κάνει παράθεση αυτών των δύο (**123123123abcabc**). Η προτεραιότητα των πράξεων ισχύει, όπως και η δυνατότητα για χρήση παρενθέσεων.

```
>>> s='abc'
>>> t='123'
>>> s+t
'abc123'
>>> 3*t+2*s
'123123123abcabc'
>>> s[1]
'b'
>>> s[1]='x'
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    s[1]='x'
TypeError: 'str' object does not support item assignment
>>> s='this is new'
>>> s
'this is new'
>>>
```

Σχήμα 5.3: Παραδείγματα με αλφαριθμητικά.

Τους χαρακτήρες ενός αλφαριθμητικού μπορούμε να τους ανακτήσουμε όπως ακριβώς ανακτούμε στοιχεία μέσα από πλειάδες. Αν, λοιπόν, ζητήσουμε τον χαρακτήρα **s[1]**, θα πάρουμε τον δεύτερο χαρακτήρα του **s**, δηλαδή το **b**. Τα αλφαριθμητικά όμως, όπως και οι πλειάδες, αφού δημιουργηθούν, δεν μπορούν να αλλάξουν. Αν, δηλαδή, προσπαθήσουμε να τροποποιήσουμε ένα αλφαριθμητικό, αυτό δεν είναι επιτρεπτό. Δείτε παρακάτω το ίδιο παράδειγμα στο Σχήμα 5.3, όπου παίρνουμε ένα μήνυμα λάθους όταν το επιχειρούμε, το οποίο μας λέει ότι δεν επιτρέπεται να μεταβληθεί η τιμή του αλφαριθμητικού (**'str' object does not support item assignment**). Μπορούμε, φυσικά, να εκχωρήσουμε μια νέα τιμή σε ένα αλφαριθμητικό, χωρίς να δημιουργείται κάποιο πρόβλημα. Το παλιό αλφαριθμητικό καταστρέφεται και ένα νέο παίρνει τη θέση του, όπως φαίνεται στο τέλος του ίδιου σχήματος.

Έχουμε μιλήσει αρκετά για τα αλφαριθμητικά σε διάφορα σημεία ως τώρα, οπότε μάλλον δεν χρειάζεται να επιμείνουμε άλλο. Θα επιστρέψουμε όμως λίγο αργότερα στο κεφάλαιο αυτό, εκεί που θα μιλήσουμε για τις ακολουθίες, αφού τα αλφαριθμητικά εντάσσονται στη γενικότερη κατηγορία των ακολουθιών.

5.4 Λίστες

Οι λίστες είναι η σημαντικότερη από τις σύνθετες δομές δεδομένων της Python. Όχι μόνο διότι χρησιμεύει σε πολλές εφαρμογές αλλά και επειδή η

φιλοσοφία της Python βασίζεται αρκετά σε αυτές. Λίστες με παρόμοια λειτουργικότητα βρίσκει κανείς σε πολλές γλώσσες. Όσες γλώσσες δεν υποστηρίζουν λίστες συνήθως παρέχουν μηχανισμούς για την υλοποίησή τους, όπως κάνει η C και η Pascal, στις οποίες οι λίστες μπορούν να υλοποιηθούν με τη χρήση δεικτών.

Στην Python τα πράγματα είναι πολύ απλά και ο προγραμματιστής δεν χρειάζεται να υλοποιήσει τίποτα, μόνο να επιλέξει το πώς θα τις χρησιμοποιήσει.

```
>>> x=[1,2,3,'ok',(2,3)]
>>> x[1]
2
>>> x[3]
'ok'
>>> x[4]
(2, 3)
>>> x[0]='yes!!'
>>> x
['yes!!', 2, 3, 'ok', (2, 3)]
>>> len(x)
5
>>> x[5]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    x[5]
IndexError: list index out of range
>>>
```

Σχήμα 5.4: Βασικές λειτουργίες σε μία λίστα.

Όπως και οι πλειάδες, έτσι και οι λίστες αποτελούνται από μία σειρά από στοιχεία, καθένα από τα οποία μπορεί να ανήκει σε διαφορετικό τύπο δεδομένων. Στο παράδειγμα στο Σχήμα 5.4 βλέπουμε τη δημιουργία μιας λίστας, της **x**, η οποία αποτελείται από τους ακέραιους 1,2 και 3, από το αλφαριθμητικό **ok** και την πλειάδα **(2,3)**. Βλέπουμε ότι τα στοιχεία μιας λίστας μπορεί να είναι οτιδήποτε.

Όπως και στις πλειάδες, αν γράψουμε **x[1]**, θα πάρουμε το δεύτερο στοιχείο στη σειρά, που εδώ είναι το 2. Το **x[3]**, δηλαδή το τέταρτο στοιχείο, είναι το αλφαριθμητικό **ok**, ενώ το πέμπτο, **x[4]**, μας επιστρέφει την πλειάδα **(2,3)** ολόκληρη. Αν τώρα θελήσουμε να αντικαταστήσουμε ένα στοιχείο της λίστας, π.χ. το πρώτο, τότε γράφοντας **x[0]='yes!!'**, το πρώτο στοιχείο της λίστας θα γίνει **yes!!**.

Η συνάρτηση **len** επιστρέφει το μέγεθος της λίστας. Η λίστα του παραδείγματος έχει μέγεθος 5, δηλαδή ξεκινάει από το 0 και φτάνει στο 4. Αν, δηλαδή,

ζητήσουμε την τιμή του **x[5]**, δεν θα πάρουμε τίποτα, ή μάλλον θα πάρουμε ένα μήνυμα λάθους.

```
>>> x=[1,2,3,'ok',(2,3),[1,2,3],2]
>>> del x[2]
>>> x
[1, 2, 'ok', (2, 3), [1, 2, 3], 2]
>>> x.append('new')
>>> x
[1, 2, 'ok', (2, 3), [1, 2, 3], 2, 'new']
>>> x.count(2)
2
>>> x.count((2,3))
1
>>> x.index([1,2,3])
4
>>> x.index(2)
1
>>> x.insert(3,'inserted')
>>> x
[1, 2, 'ok', 'inserted', (2, 3), [1, 2, 3], 2, 'new']
>>>
```

Σχήμα 5.5: Μέθοδοι σε λίστες.

Πάμε τώρα στα παραδείγματα του Σχήματος 5.5. Παρατηρήστε ότι στη λίστα **x**, εκτός των στοιχείων που είχε προηγουμένως, προσθέσαμε τη λίστα **[1,2,3]** και τον ακέραιο 2. Ο διερμηνευτής δεν διαμαρτυρήθηκε, άρα επιτρέπεται μία λίστα να περιέχει και άλλες λίστες, κάτι αρκετά χρήσιμο, όπως θα δούμε αργότερα όταν μιλήσουμε για πίνακες. Στη συνέχεια πληκτρολογούμε **del x[2]** και το στοιχείο στη θέση 2 της λίστας (το τρίτο στοιχείο θυμίζω) αφαιρείται από τη λίστα. Η λίστα μικραίνει κατά ένα στοιχείο και όλα τα στοιχεία δεξιά από το στοιχείο που αφαιρέθηκε έρχονται μία θέση αριστερά.

Στη συνέχεια καλούμε κάποιες άλλες συναρτήσεις πάνω στη λίστα. Αν θέλουμε να είμαστε ακριβείς, πρέπει να τις πούμε **μεθόδους (methods)**. Ας μην αναλύσουμε όμως τι είναι η μέθοδος, έχει να κάνει με τον αντικειμενοστραφή προγραμματισμό. Ας τις λέμε μεθόδους και ας γνωρίζουμε μόνο ότι εφαρμόζουν πάνω στη λίστα κάποιες λειτουργίες με βάση τις παραμέτρους τους.

Η μέθοδος **append** προσθέτει ένα στοιχείο στο τέλος της λίστας. Έτσι, αν θέλουμε να προσθέσουμε το στοιχείο **new** στο τέλος της λίστας θα γράψουμε **x.append('new')**. Το **new** θα προστεθεί στο τέλος της λίστας, όπως φαίνεται στο Σχήμα 5.5. Το μέγεθος της λίστας θα αυξηθεί κατά 1.

Η μέθοδος **count** μετράει πόσες φορές το στοιχείο που της δίνεται σαν παράμετρος εμφανίζεται στη λίστα. Στο παράδειγμα ζητάμε να μετρηθεί πόσες

φορές υπάρχει ο ακέραιος 2 στη λίστα. Το αποτέλεσμα που μας επιστρέφεται είναι 2, διότι βρήκε το 2 μία φορά στη θέση 1 της λίστας και μία φορά στη θέση 6. Παρατηρήστε ότι το 2 εμφανίζεται και μέσα στην πλειάδα **(2,3)** αλλά και στη λίστα **[1,2,3]**.

Επειδή, όμως, το στοιχείο με το οποίο συγκρίνεται το 2 είναι η πλειάδα ή η λίστα και όχι ο ακέραιος 2, η Python δεν το συνυπολογίζει στο μέτρημα. Γενικά, σταματάει τον έλεγχο στο πρώτο επίπεδο της λίστας και δεν εισέρχεται αναδρομικά μέσα σε κάθε στοιχείο της για να συνεχίσει την αναζήτηση.

Η μέθοδος **index** επιστρέφει τη θέση του πρώτου στοιχείου που εμφανίζεται στη λίστα και είναι ίσο με αυτό που της έχει δοθεί σαν παράμετρος. Και πάλι η αναζήτηση μένει στο πρώτο επίπεδο και δεν προχωράει αναδρομικά στα στοιχεία της λίστας. Έτσι, όταν θα ζητήσουμε το **index** για το στοιχείο **[1,2,3]**, τότε θα το αναζητήσει και θα το βρει στη θέση 4, αριθμός που θα επιστραφεί σαν αποτέλεσμα.

Αν, τώρα, ρωτούσαμε για το **index** του στοιχείου 2, τότε θα μας επιστρεφόταν σαν αποτέλεσμα η θέση του πρώτου 2 μέσα στην λίστα, η θέση 1 δηλαδή. Το άλλο 2 το οποίο είναι στη θέση 5 (6ο στη λίστα) δεν θα βρεθεί.

Η μέθοδος **insert** εισάγει ένα στοιχείο σε μία θέση της λίστας που της δίνεται σαν παράμετρος. Τα στοιχεία της λίστας που βρίσκονται μετά από αυτό μετακινούνται μία θέση προς τα δεξιά και το μέγεθος της λίστας μεγαλώνει κατά 1. Στο παράδειγμα εισάγεται το **inserted** στη θέση 3, δηλαδή ακριβώς μετά το **ok**. Τα στοιχεία από τη θέση 3 της παλιάς λίστας και έπειτα "παραμέρισαν κάνοντας χώρο" για να μπει το νέο στοιχείο στη θέση που ζητήθηκε.

Στο Σχήμα 5.6 φαίνονται μερικά ακόμα παραδείγματα μεθόδων που εφαρμόζονται σε λίστες.

Η **pop** αφαιρεί το τελευταίο στοιχείο και το επιστρέφει σαν αποτέλεσμα. Στο παράδειγμα το τελευταίο στοιχείο της λίστας είναι το **new**. Η **pop** το αφαιρεί από τη λίστα και το επιστρέφει στον διερμηνευτή. Στο παράδειγμα αυτό η **pop** δεν πήρε κάποια παράμετρο. Όταν βάζουμε στην **pop** παράμετρο, τότε η **pop** αφαιρεί από τη λίστα, όχι το τελευταίο στοιχείο, αλλά αυτό που βρίσκεται στη θέση που δόθηκε στην παράμετρο. Έτσι, στο παράδειγμα δόθηκε σαν παράμετρος το 3, οπότε και αφαιρέθηκε από τη λίστα και επιστράφηκε στον διερμηνευτή το **inserted**, που ήταν στη θέση 3 της λίστας.

```
>>> x
[1, 2, 'ok', 'inserted', (2, 3), [1, 2, 3], 2, 'new']
>>> x.pop()
'new'
>>> x
[1, 2, 'ok', 'inserted', (2, 3), [1, 2, 3], 2]
>>> x.pop(3)
'inserted'
>>> x
[1, 2, 'ok', (2, 3), [1, 2, 3], 2]
>>> x.remove(2)
>>> x
[1, 'ok', (2, 3), [1, 2, 3], 2]
>>> x.reverse()
>>> x
[2, [1, 2, 3], (2, 3), 'ok', 1]
>>> y=['a','b','c','d']
>>> x.extend(y)
>>> x
[2, [1, 2, 3], (2, 3), 'ok', 1, 'a', 'b', 'c', 'd']
>>> y
['a', 'b', 'c', 'd']
```

Σχήμα 5.6: Μέθοδοι σε λίστες (συνέχεια).

Παρόμοια λειτουργία έχει και η **remove**, στην οποία δίνεται σαν παράμετρος, όχι η θέση του στοιχείου αλλά το ίδιο το στοιχείο. Έτσι, στο παράδειγμα, δόθηκε σαν παράμετρος το 2, και αφαιρέθηκε από τη λίστα το 2 που βρισκόταν στην θέση 1. Προσέξτε ότι στη λίστα υπήρχε και δεύτερο 2, επιλέχτηκε όμως το πρώτο στη σειρά από τα δύο για να αφαιρεθεί.

Η **reverse** αντιστρέφει τη σειρά των στοιχείων της λίστας στην οποία εφαρμόζεται. Δείτε ένα παράδειγμα χρήσης της στο Σχήμα 5.6.

Η **extend** εφαρμόζεται σε μία λίστα και παίρνει ως παράμετρο μία άλλη λίστα. Προσθέτει στο τέλος της λίστας στην οποία εφαρμόζεται τα στοιχεία της λίστας που δόθηκε ως παράμετρος. Παρατηρήστε στο Σχήμα 5.6 ότι η λίστα **x** στην οποία εφαρμόστηκε η μέθοδος έχει τροποποιηθεί, ενώ η λίστα **y** που χρησιμοποιήθηκε ως παράμετρος παρέμεινε αμετάβλητη.

5.5 Σύνολα

Τα **σύνολα (sets)** είναι μη διατεταγμένες συλλογές από στοιχεία, καθένα από τα οποία εμφανίζεται μόνο μία φορά στη συλλογή. Χρησιμοποιούνται για να παρασταθούν δομές, όπως αυτές των συνόλων έτσι όπως τις ξέρουμε από

τα μαθηματικά.

Τα στοιχεία ενός συνόλου τα συμβολίζουμε μέσα σε άγκιστρα. Ορίζουμε ότι το σύνολο **A** έχει τα στοιχεία **1,2,3** γράφοντας:

A={1,2,3}

Δείτε το παράδειγμα στο Σχήμα 5.7. Ένα σύνολο διαφέρει από μία λίστα στο ότι στα στοιχεία του δεν υπάρχει η έννοια της διάταξης. Έτσι, τα σύνολα:

A={1,2,3}, B={3,1,2}

είναι μεταξύ τους ίσα, γι' αυτό και ο έλεγχος ισότητας επιστρέφει **True** (δείτε παρακάτω στο ίδιο σχήμα).

```
>>> A={1,2,3}
>>> B={3,1,2}
>>> A==B
True
>>> 1 in A
True
>>> 14 in A
False
>>> len(A)
3
>>> C={1,2,3,4}
>>> C.issuperset(B)
True
>>> B.issuperset(C)
False
>>> B.issubset(C)
True
>>>
```

Σχήμα 5.7: Παραδείγματα με σύνολα.

Οι πιο συνηθισμένες ερωτήσεις που κάνουμε για ένα σύνολο είναι εάν ένα στοιχείο ανήκει σε ένα σύνολο ή όχι και ποιος είναι ο πληθάριθμός του (πληθικός αριθμός). Για το πρώτο υπάρχει ο τελεστης **in** και για το δεύτερο η συνάρτηση **len**. Στο παράδειγμά μας, αν ρωτήσουμε εάν το 1 ανήκει στο σύνολο **A** (**1 in A**), θα πάρουμε θετική απάντηση, ενώ αν ρωτήσουμε αν το 14 ανήκει στο **A**, θα πάρουμε αρνητική. Ο πληθάριθμός του **A** είναι φυσικά 3 και για να τον πάρουμε θα γράψουμε **len(A)**. Επίσης, πολλές φορές ρωτάμε αν ένα σύνολο είναι υποσύνολο ή υπερσύνολο ενός άλλου συνόλου.

Ας ορίσουμε, λοιπόν, ένα ακόμα σύνολο, το:

C={1,2,3,4}

και ας κάνουμε κάποιες ερωτήσεις για να δούμε τι απαντήσεις θα πάρουμε. Αν ρωτήσουμε:

C.issuperset(B)

θα πάρουμε την απάντηση **True**, αφού πράγματι το **C** είναι υπερσύνολο του **B**. Αν ρωτήσουμε:

B.issuperset(C)

θα πάρουμε την απάντηση **False**, αφού μόλις είπαμε ότι το **B** δεν είναι υπερσύνολο του **C**. Αντίθετα, αν ρωτήσουμε

B.issubset(C)

τώρα θα πάρουμε θετική απάντηση.

```
>>> A={1,2,3}
>>> B={1,4,5}
>>> C={2,3,4,5,6}
>>> A.intersection(B)
{1}
>>> A.intersection(C)
{2, 3}
>>> A.union(B)
{1, 2, 3, 4, 5}
>>> C.difference(A)
{4, 5, 6}
>>> C.symmetric_difference(A)
{1, 4, 5, 6}
>>>
```

Σχήμα 5.8: Πράξεις στα σύνολα.

Οι πιο συνήθεις μαθηματικές πράξεις που ορίζονται σε ένα σύνολο είναι η ένωση, η τομή, η διαφορά και η συμμετρική διαφορά δύο συνόλων. Γι' αυτές υπάρχουν οι **union**, **intersection**, **difference** και **symmetric_difference** αντίστοιχα. Πάμε στο Σχήμα 5.8 για να δούμε παραδείγματα. Ας θεωρήσουμε τα σύνολα:

A={1,2,3}, B={1,4,5}, C={2,3,4,5,6}

Το

A.intersection(B)

θα μας δώσει την τομή των δύο συνόλων. Τα μόνο κοινό τους στοιχείο είναι το 1, άρα θα επιστρέψει το μονοσύνολο **{1}**. Το

A.intersection(C)

θα δώσει την τομή των **A** και **C**, δηλαδή το **{2,3}**. Το

A.union(B)

θα δώσει την ένωσή τους, το σύνολο δηλαδή που αποτελείται από τα στοιχεία και του **A** και του **B**. Αυτό είναι το **{1,2,3,4,5}**. Το

C.difference(A)

θα μας δώσει τη διαφορά τους, τα στοιχεία, δηλαδή, του **C** που δεν ανήκουν στο **A** (το σύνολο **{4,5,6}**), ενώ τέλος το

C.symmetric_difference(A)

τη συμμετρική διαφορά τους, τα στοιχεία δηλαδή του **A** που δεν ανήκουν στο **C** και τα στοιχεία του **C** που δεν ανήκουν στο **A**. Το σύνολο αυτό, όπως φαίνεται και στο Σχήμα 5.8, είναι το **{1,4,5,6}**.

```
>>> A={'a','b','c'}
>>> A.add('new')
>>> A
{'b', 'a', 'new', 'c'}
>>> t={1,2,3}
>>> A.update(t)
>>> A
{'c', 1, 2, 3, 'b', 'new', 'a'}
>>> A.remove(2)
>>> A
{'c', 1, 3, 'b', 'new', 'a'}
>>> A.pop()
'c'
>>> A
{1, 3, 'b', 'new', 'a'}
>>> A.clear()
>>> A
set()
>>>
```

Σχήμα 5.9: Προσθαφαίρεση στοιχείων σε σύνολα.

Σε ένα σύνολο μπορούμε να προσθέσουμε και να αφαιρέσουμε στοιχεία. Ας χρησιμοποιήσουμε σαν βάση το Σχήμα 5.9. Ας ορίσουμε, λοιπόν, το σύνολο:

A{'a','b','c'}

και ας του προσθέσουμε το στοιχείο **new**. Θα χρειαστούμε την **add** και θα την εφαρμόσουμε πάνω στο **A** με παράμετρο το **new**. Το **new** θα προστεθεί στο **A**. Αν τυπώσουμε το **A**, μπορεί να δούμε το **new** να εμφανίζεται σε οποιαδήποτε θέση. Αυτό συμβαίνει διότι στα σύνολα δεν υπάρχει η έννοια της διάταξης, όπως αναφέραμε και παραπάνω. Μπορούμε επίσης να προσθέσουμε πάνω από ένα στοιχείο προσθέτοντας στοιχεία σε ένα σύνολο από ένα άλλο σύνολο. Για τη δουλειά αυτήν υπάρχει η **update**, με την οποία στο παράδειγμα προσθέτουμε τα στοιχεία του συνόλου **t={1,2,3}** στο σύνολο **A**.

Για την αφαίρεση ενός στοιχείου υπάρχει η **remove** και η **discard**. Η διαφορά τους είναι ότι η πρώτη δημιουργεί μήνυμα λάθους αν το στοιχείο που θέλουμε να αφαιρέσουμε δεν υπάρχει στο σύνολο. Στο παράδειγμα αφαιρούμε με τη **remove** το 2 από το **A**. Η **pop** αφαιρεί και επιστρέφει ένα τυχαίο στοιχείο του συνόλου (και σφάλμα αν το σύνολο είναι το κενό). Στο παράδειγμα αφαιρεί το **c** από το σύνολο **A** και το επιστρέφει στον διερμηνευτή. Τέλος, η **clear** αφαιρεί όλα τα στοιχεία ενός συνόλου και το κάνει ίσο με το κενό σύνολο.

5.6 Ακολουθίες

Οι **ακολουθίες (sequences)** είναι σειρές από δεδομένα τα οποία είναι αριθμημένα, δηλαδή προσπελάζονται μέσα από δείκτες. Κάθε, λοιπόν, στοιχείο μιας ακολουθίας έχει έναν αύξοντα αριθμό, ο οποίος είναι και το κλειδί για να φτάσουμε σε αυτόν.

Οι ακολουθίες δεν είναι ξεχωριστός τύπος δεδομένων. Οι ακολουθίες ομαδοποιούν τις πλειάδες, τα αλφαριθμητικά και τις λίστες ομογενοποιώντας τον τρόπο προσπέλασης στα δεδομένα που έχουν οργανωθεί με αυτά. Μπορούμε να το φανταστούμε σαν μία ομπρέλα που τα καλύπτει όλα και προσφέρει έναν κοινό τρόπο πρόσβασης σε αυτά, όπου αυτό έχει νόημα.

Χωρίζονται σε **μεταβλητές (mutable)** ακολουθίες και **αμετάβλητες (immutable)**. **Μεταβλητές** είναι αυτές που αφότου δημιουργηθούν, επιτρέπεται να τροποποιηθούν. Αμετάβλητες είναι αυτές που, αφότου δημιουργηθούν, απαγορεύεται να αλλάξει το περιεχόμενό τους. Μεταβλητές είναι οι λίστες. Όπως έχουμε δει, μία λίστα μπορεί να τροποποιηθεί αφού φτιαχτεί. Αμετάβλητες είναι τα αλφαριθμητικά και οι πλειάδες. Όπως πάλι είδαμε, αφού δημιουργηθούν, δεν μπορούν να μεταβληθούν. Έτσι, ό,τι θα λέμε παρακάτω θα αφορά και τις λίστες και τις πλειάδες αλλά και τα αλφαριθμητικά στην περίπτωση που δεν υπάρχει τροποποίηση στοιχείων, αλλά θα αφορά μόνο τις λίστες στην περίπτωση που τα στοιχεία της ακολουθίας τροποποιούνται.

```
>>> L=['a','b','c','d','e','f','g']
>>> L[0:3]
['a', 'b', 'c']
>>> L[:3]
['a', 'b', 'c']
>>> L[3:]
['d', 'e', 'f', 'g']
>>> L[3:6]
['d', 'e', 'f']
>>> L[:]
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>>
```

Σχήμα 5.10: Παραδείγματα τεμαχισμού ακολουθιών.

Έστω, λοιπόν, ότι έχουμε την ακολουθία **s**. Το **s[1]** επιστρέφει το δεύτερο στοιχείο της ακολουθίας, εφόσον υπάρχει, και το **s[1]=...** τροποποιεί το δεύτερο στοιχείο της ακολουθίας, εφόσον υπάρχει αλλά και εφόσον αναφερόμαστε σε μεταβλητές ακολουθίες.

Η σημαντικότερη πράξη στις ακολουθίες είναι ο **τεμαχισμός (slicing)**, είτε μιλάμε για αναζήτηση είτε για αντικατάσταση στοιχείων. Στο Σχήμα 5.10 φαίνονται διάφορα παραδείγματα τεμαχισμού. Έστω ότι έχουμε τη λίστα:

L=['a','b','c','d','e','f','g']

και έστω ότι θα ζητήσουμε το **L[0:3]**. Τότε θα μας επιστραφεί μία λίστα η οποία θα ξεκινάει από το στοιχείο 0 της **L** και θα φτάνει μέχρι (όχι μέχρι και) το στοιχείο 3. Άρα η λίστα που θα μας επιστραφεί θα είναι η **['a','b','c']**. Το αποτέλεσμα θα ήταν το ίδιο αν είχαμε ζητήσει τη λίστα **L[:3]**. Η παράληψη του πρώτου αριθμού πριν την άνω και κάτω τελεία σημαίνει από την αρχή της ακολουθίας.

Το ίδιο φυσικά συμβαίνει και εάν παραλείψουμε τον δεύτερο αριθμό. Το **L[3:]** θα ξεκινήσει από το **d** και θα φτάσει στο τέλος, άρα η λίστα που θα επιστραφεί θα είναι η **['d','e','f','g']**. Προσέξτε ότι το **L[3:6]** δεν επιστρέφει και το έκτο στοιχείο της λίστας, άρα δεν επιστρέφει την **['d','e','f','g']**, όπως ίσως βιαστικά θα υποθέταμε, αλλά την **['d','e','f']**. Παρατηρήστε επίσης ότι η **L[:]** θα επιστρέψει όλη τη λίστα.

Αν θέλουμε, μπορούμε να χρησιμοποιήσουμε και **βήμα (step)** (Σχήμα 5.11), δηλαδή να μην παίρνουμε τα στοιχεία με τη σειρά, αλλά με κάποιον άλλο, αυστηρά ορισμένο, τρόπο. Αν ζητήσουμε τη λίστα **L[0:6:2]**, θα πάρουμε μία λίστα που ξεκινάει από το στοιχείο 0 της **L**, δηλαδή το **a**, στη συνέχεια αφήνει το στοιχείο **b** και πηγαίνει στο στοιχείο **c**. Αυτό το κάνει διότι έχει δηλωθεί, ακριβώς μετά το **0:6** που ορίζει τα όρια της αρχικής μέσα από τα οποία θα προέλ-

θουν τα στοιχεία της νέας λίστας, τον αριθμό 2. Αν είχε δηλωθεί εκεί ο αριθμός 3, θα έπαιρνε ένα στοιχείο και θα παρέλειπε δύο, κ.ο.κ.

```
>>> L=['a','b','c','d','e','f','g']
>>> L[0:6:2]
['a', 'c', 'e']
>>> L[0::2]
['a', 'c', 'e', 'g']
>>> L[::2]
['a', 'c', 'e', 'g']
>>> L[5:1:-2]
['f', 'd']
>>> L[::-3]
['g', 'd', 'a']
>>> L[1:5:-2]
[]
>>>
```

Σχήμα 5.11: Βήμα στον τεμαχισμό.

Έτσι, η λίστα που θα μας επιστραφεί σαν αποτέλεσμα θα είναι η **['a','c','e']**. Τα **b**,**d**,**f** παρελήφθησαν λόγω του βήματος, ενώ το **g** διότι ζητήσαμε μέχρι το στοιχείο **x** στη θέση 6. Αν θέλαμε και το τελευταίο στοιχείο, θα έπρεπε να ζητήσουμε τη λίστα **L[0::2]** ή την **L[::2]**.

Φυσικά, το βήμα μπορεί να είναι και αρνητικό. Το **L[5:1:-2]** θα ξεκινήσει από το στοιχείο στη θέση 5, δηλαδή το **f**, θα κινηθεί προς τα πίσω στη λίστα, θα παραλείψει το **e**, αφού το βήμα υποδηλώνει ότι παίρνουμε ένα στοιχείο και ένα αφήνουμε, θα πάρει το **d**, θα αφήσει το **c** λόγω βήματος, αλλά δεν θα πάρει και το **b**, αφού η λίστα πηγαίνει μέχρι το στοιχείο στη θέση 1 και όχι μέχρι και αυτό.

```
>>> L=['a','b','c','d','e','f','g']
>>> L[-1]
'g'
>>> L[-4:-2]
['d', 'e']
>>>
```

Σχήμα 5.12: Αναφορά σε στοιχείο της ακολουθίας μετρώντας από το τέλος.

Το **L[::-3]** θα δώσει τη λίστα **['g','d','a']**, όπως περιμένατε, και το **L[1:5:-2]** την κενή λίστα, αφού δεν είναι δυνατόν να κινηθείς από το στοιχείο στη θέση 1 προς το στοιχείο στη θέση 5 κινούμενος με φορά από το τέλος της λίστας προς την αρχή της.

Κάτι άλλο πολύ χρήσιμο είναι η δυνατότητα να έχει κανείς πρόσβαση σε μια ακολουθία μετρώντας στοιχεία από το τέλος προς την αρχή. Παραδείγματα βλέπουμε στο Σχήμα 5.12. Αν ζητήσουμε το στοιχείο **L[-1]**, ο διερμηνευτής θα μας απαντήσει **g**. Με το -1, δηλαδή, θα καταλάβει το πρώτο στοιχείο από το τέλος. Όμοια φυσικά, αν του ζητήσουμε το **L[-4:-2]**, θα ξεκινήσει από το τέταρτο στοιχείο από το τέλος και θα φτάσει μέχρι το δεύτερο στοιχείο από το τέλος. Μην ξεχνιόμαστε, μέχρι το δεύτερο από το τέλος, όχι μέχρι και το δεύτερο από το τέλος. Έτσι, το αποτέλεσμα θα είναι **['d','e']**.

```
>>> L=['a','b','c','d','e','f','g']
>>> L[2:4]=['new1','new2','new3']
>>> L
['a', 'b', 'new1', 'new2', 'new3', 'e', 'f', 'g']
>>> L[6:6]=['here']
>>> L
['a', 'b', 'new1', 'new2', 'new3', 'e', 'here', 'f', 'g']
>>>
```

Σχήμα 5.13: Αντικατάσταση και εισαγωγή σε ακολουθία.

Τέλος, ας δούμε και λίγο μερικές αντικαταστάσεις. Αν και μιλάμε γενικά για ακολουθίες, αυτή η πράξη έχει νόημα στις λίστες που είναι μεταβλητές δομές δεδομένων. Ο τεμαχισμός μπορεί να χρησιμοποιηθεί ώστε να δηλωθεί ένα τμήμα ακολουθίας που θέλουμε να αντικατασταθεί. Για παράδειγμα (δες στο Σχήμα 5.13), αν θέλουμε να αντικατασταθούν τα στοιχεία στις θέσεις 2 και 3 με τα στοιχεία **new1**, **new2**, **new3**, μπορούμε να γράψουμε το εξής:

L[2:4]=['new1','new2','new3']

Παρατηρήστε ότι αντικαταστήσαμε δύο στοιχεία με τρία. Τι θα γινόταν τώρα αν προσπαθούσαμε να αντικαταστήσουμε μηδέν στοιχεία με ένα, αν γράφαμε δηλαδή:

L[6:6]=['here']

Θα παρενέβαλλε στην έκτη θέση το στοιχείο **here**.

5.7 Λεξικά

Μία πολύ χρήσιμη δομή δεδομένων την οποία βρίσκει κανείς σε λίγες γλώσσες, μέσα σε αυτές και η Python, είναι τα λεξικά. Τα λεξικά θυμίζουν λίγο τις

λίστες, έχουν όμως την εξής ουσιώδη διαφορά: Ενώ στις λίστες η προσπέλαση των στοιχείων γίνεται με δείκτες, στα λεξικά γίνεται με λέξεις-κλειδιά. Ενώ οι δείκτες είναι ακέραιοι αριθμοί οι οποίοι ξεκινάνε από τη μηδέν, είναι διατεταγμένοι και συνεχόμενοι, οι λέξεις-κλειδιά μπορούν να είναι οτιδήποτε.

Ας δούμε ένα παράδειγμα στο οποίο φαίνεται η χρησιμότητα των λεξικών. Έστω ότι επιθυμούμε να φτιάξουμε κάτι σαν τηλεφωνικό κατάλογο, ο οποίος να έχει για κάθε φίλο μας καταχωρημένο το τηλέφωνό του. Αν θεωρήσουμε απίθανο και αγνοήσουμε την πιθανότητα δύο φίλοι μας να έχουν το ίδιο ονοματεπώνυμο, η λέξη-κλειδί που χαρακτηρίζει κάθε φίλο μας θα είναι το ονοματεπώνυμό του. Ας θεωρήσουμε, κιόλας, ότι αρκεί το μικρό του όνομα, για απλότητα. Έτσι, όπως μπορούμε να δούμε στο Σχήμα 5.14, μπορούμε να ορίσουμε ένα λεξικό με τρεις εγγραφές ως εξής:

```
tel={'giorgos':'26542', 'petros':'25421', 'andreas':'56254'}
```

.

Και εάν θέλουμε να ρωτήσουμε για το τηλέφωνο του Γιώργου, δεν έχουμε παρά να γράψουμε:

```
tel['giorgos']
```

```
>>> tel={'giorgos':'26542','petros':'25421','andreas':'56254'}
>>> tel['giorgos']
'26542'
>>> tel.update({'sylvia':'56541'})
>>> tel
{'andreas': '56254', 'sylvia': '56541', 'petros': '25421',
                                          'giorgos': '26542'}
>>> tel['sylvia']='unknown'
>>> tel
{'andreas': '56254', 'sylvia': 'unknown', 'petros': '25421',
                                           'giorgos': '26542'}
>>> x=tel.pop('giorgos')
>>> tel
{'andreas': '56254', 'sylvia': 'unknown', 'petros': '25421'}
>>> x
'26542'
```

Σχήμα 5.14: Παραδείγματα με λεξικά.

Τώρα, αν θέλουμε να προσθέσουμε μία εγγραφή στον τηλεφωνικό μας κατάλογο, ας πούμε το τηλέφωνο της Σύλβιας, θα πρέπει να χρησιμοποιήσουμε την **update** ως εξής:

tel.update({'sylvia':'56541'})

Τυπώνοντας όλο το λεξικό με την **print**, θα δείτε ότι η Σύλβια έχει προστεθεί στο λεξικό το οποίο έχει τώρα τέσσερις εγγραφές.

Η τροποποίηση της τιμής μίας εγγραφής στο λεξικό γίνεται όπως στις λίστες. Γράφοντας:

tel['sylvia']='unknown'

στη θέση του τηλεφώνου της Σύλβιας θα υπάρχει η ένδειξη **unknown**.

Η διαγραφή μιας εγγραφής από ένα λεξικό γίνεται με την **pop**. Παρακάτω, στο Σχήμα 5.14, φαίνεται ένα παράδειγμα χρήσης της. Στο παράδειγμα η **pop** καλείται με παράμετρο το **giorgos**, οπότε αφαιρεί από το λεξικό την εγγραφή αυτήν και ταυτόχρονα επιστρέφει το τηλέφωνο του Γιώργου, το οποίο καταχωρείται στη μεταβλητή **x**.

Υπάρχουν κάποιες ακόμα μέθοδοι οι οποίες μπορούν να εφαρμοστούν σε λεξικά, αλλά δεν είναι μέσα στους σκοπούς του βιβλίου να μπούμε σε τέτοια λεπτομέρεια. Αντίθετα, είναι πολύ ενδιαφέρον να δούμε μία περισσότερο πολύπλοκη δόμηση δεδομένων με τη χρήση λεξικών. Ας θεωρήσουμε ότι μας ενδιαφέρει να κρατήσουμε για κάθε φίλο μας περισσότερη πληροφορία από το τηλέφωνό του, για παράδειγμα και το mail του. Θα φτιάξουμε τότε ένα λεξικό από λεξικά για τον σκοπό αυτόν.

```
>>> giorgosData={'tel':'26542','mail':'giorgos@gr'}
>>> petrosData={'tel':'25421','mail':'petros@gr'}
>>> andreasData={'tel':'56254','mail':'andreas@gr'}
>>> D={'giorgos':giorgosData,'petros':petrosData,
                                'andreas':andreasData}
>>> D['petros']['mail']
'petros@gr'
>>>
```

Σχήμα 5.15: Λεξικά μέσα σε λεξικά.

Στο Σχήμα 5.15 ορίζουμε τρία λεξικά, τα **giorgosData**, **petrosData** και το **andreasData**, καθένα από τα οποία έχει δύο εγγραφές: το τηλέφωνο και το mail. Στη συνέχεια φτιάχνουμε το λεξικό **D**, το οποίο έχει μέσα του τα **giorgosData**,

petrosData και **andreasData**. Τώρα, για να έχουμε πρόσβαση στο mail του Πέτρου, θα πρέπει να γράψουμε:

D['petros']['mail']

Πολύ πιο κομψό από το να οργανώναμε τον τηλεφωνικό μας κατάλογο με λίστες, ενώ θα έπρεπε να προσπελάσουμε την πληροφορία με δείκτες οι οποίοι θα ήταν ακέραιοι αριθμοί.

Λειτουργίες σε συμβολοσειρές	
s.count(x)	μετράει τις εμφανίσεις του x στο s
s.find(x)	βρίσκει την θέση της πρώτης εμφάνισης του x στο s
s.index(x)	όπως το find αλλά βγάζει error αν δεν υπάρχει το x
s.ljust(len)	επιστρέφει συμβολοσειρά μήκους len η οποία περιέχει την s με αριστερή στοίχιση
s.rjust(x)	επιστρέφει συμβολοσειρά μήκους len η οποία περιέχει την s με δεξιά στοίχιση
s.replace(old,new)	επιστρέφει συμβολοσειρά στην οποία έχει αντικαταστήσει στην s την υποσυμβολοσειρά old με την new
s.split([delim])	επιστρέφει λίστα των λέξεων μέσα στη συμβολοσειρά προαιρετικά ορίζονται οι διαχωριστές στη delim
s.splitlines()	δημιουργεί λίστα με βάση τις αλλαγές γραμμών που υπάρχουν στην s
s.strip([w])	επιστρέφει συμβολοσειρά από την οποία έχουν αφαιρεθεί από την αρχή και το τέλος οι λευκοί χαρακτήρες της s ή αν οριστεί η συμβολοσειρά w, αυτοί που υπάρχουν στην w
s.lower()	επιστρέφει συμβολοσειρά με όλα τα γράμματα της s πεζά
s.upper()	το ίδιο με την lower αλλά με γράμματα κεφαλαία
s.swapcase()	επιστρέφει συμβολοσειρά όπου τα πεζά γράμματα γίνονται κεφαλαία και αντίστροφα
s.title()	επιστρέφει συμβολοσειρά με γράμματα τίτλου: κάθε λέξη ξεκινάει με κεφαλαίο, τα υπόλοιπα πεζά

Πίνακας 5.1: Συνοπτικός πίνακας λειτουργιών σε αλφαριθμητικά.

5.8 Συνοπτικοί πίνακες λειτουργιών στις βασικότερες δομές δεδομένων

Παρακάτω μπορείτε να βρείτε συνοπτικά και οργανωμένες σε πίνακες τις σημαντικότερες λειτουργίες της Python επάνω στις σημαντικότερες δομές

δεδομένων που είδαμε στο κεφάλαιο αυτό. Φυσικά, ούτε ο κατάλογος των λειτουργιών είναι πλήρης αλλά ούτε και η περιγραφή τους. Το πρώτο θα ήταν κουραστικό και σίγουρα έξω από τους σκοπούς του βιβλίου. Το δεύτερο θα καταργούσε τον χαρακτήρα του συνοπτικού. Για περισσότερες λεπτομέρειες φυσικά μπορείτε να ανατρέξετε στο κείμενο. Για ακόμα περισσότερες στο διαδίκτυο, όπου μία αναζήτηση θα σας πάει γρήγορα στη λειτουργία που θέλετε και θα σας δωσει όση λεπτομέρεια χρειάζεστε.

Λειτουργίες σε λίστες	
len(L)	επιστρέφει τον αριθμό των στοιχείων της L
L.index(x)	επιστρέφει τη θέση του x στη λίστα
L.count(x)	μετράει τις εμφανίσεις του x στην L
L.insert(p,x)	εισάγει το στοιχείο x στη θέση p (μεταφέροντας τα στοιχεία μετά τη θέση p κατά μία θέση)
L.remove(x)	αφαιρεί από την L το στοιχείο x
L.pop([p])	αφαιρεί και επιστρέφει ένα τυχαίο στοιχείο από την L αν δηλωθεί το p, αφαιρεί και επιστρέφει το στοιχείο στη θέση p
L.append(x)	προσθέτει το x στο τέλος της L
L.extend(L)	προσθέτει τη λίστα A στο τέλος της L
L.reverse()	αντιστρέφει τη λίστα (το πρώτο στοιχείο τελευταίο και το τελευταίο πρώτο)
L.sort()	ταξινομεί τη λίστα L

Πίνακας 5.2: Συνοπτικός πίνακας λειτουργιών σε λίστες.

Περισσότερο υλικό και παραδείγματα που αφορούν δομές δεδομένων που χρησιμοποιεί η Python μπορείτε να διαβάσετε στα κεφάλαια 3,5,7,9 του βιβλίου [1], στα κεφάλαια 8,10,11,12 του βιβλίου [2], στα κεφάλαια 10,11,14 του βιβλίου [3], στο κεφάλαιο 9 του βιβλίου [4] και στο κεφάλαιο **δομές δεδομένων** του βιβλίου [5].

Λειτουργίες σε σύνολα	
x in A	ανήκει το x στο S;
len(A)	πληθάριθμος του S
A.subset(B)	είναι το A υποσύνολο του B;
A.superset(B)	είναι το A υπερσύνολο του B;
A.union(B)	A ένωση B
A.intersection(B)	A τομή B
A.difference(B)	Διαφορά του A από το B
A.symmetric_difference(B)	Συμμετρική διαφορά των A και B
A.copy()	Δημιουργεί αντίγραφο του A
A.update(B)	προσθέτει τα στοιχεία του B στο A
A.add(x)	προσθέτει το x στο A
A.remove(x)	αφαιρεί το x από το A
A.discard(x)	αφαιρεί το x από το A, αν υπάρχει
A.pop()	αφαιρεί και επιστρέφει ένα στοιχείο από το A
A.clear()	αφαιρεί όλα τα στοιχεία από το A

Πίνακας 5.3: Συνοπτικός πίνακας λειτουργιών σε σύνολα.

Λειτουργίες σε λεξικά	
len(d)	επιστρέφει τον αριθμό των στοιχείων του d
del d[key]	αφαιρεί το key από το λεξικό d
key in d	ελέγχει αν το d έχει κλειδί x
d[key]=x	βάζει στο κλειδί key την τιμή x
d.copy()	δημιουρεί αντίγραφο του λεξικού
d.clear()	αφαιρεί όλα τα στοιχεία του λεξικού
d.has_key(key)	ελέγχει αν το d έχει κλειδί key
d.pop(key)	αφαιρεί το key από το d και το επιστρέφει
d.popitem()	αφαιρεί ένα τυχαίο στοιχείο από το d και το επιστρέφει
d.update(t)	ενημερώνει το d με βάση τα στοιχεία του t

Πίνακας 5.4: Συνοπτικός πίνακας λειτουργιών σε λεξικά.

Βιβλιογραφία

1. Jennifer Campel, Paul Gries, Jason Montojo, Greg Wilson (2019). Practical Programming, **An Introduction to Computer Science**

Using Python. Publisher: The Pragmatic Bookself.

2. Allen B. Downey (2012). **Think Python**. Publisher: O'Reilly Media.
3. Brian Heinold (2012). **Introduction to Programming Using Python**. Publisher: Mount St. Mary's University, Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.
4. Cody Jackson (2011). **Learning to Program Using Python**. Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.
5. C. H. Swaroop (2015). **A Byte of Python**. Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο, Μετάφραση: ubuntu-gr.org Team.

Κεφάλαιο 6: Οι δομές ελέγχου ροής

Κάθε γλώσσα προγραμματισμού έχει τις δικές της προγραμματιστικές δομές. Οι βασικές όμως προγραμματιστικές δομές δεν διαφέρουν σημαντικά από γλώσσα σε γλώσσα. Κατά την προσφιλή μας συνήθεια θα μιλήσουμε γενικά, αλλά θα χρησιμοποιήσουμε την Python ώστε να γίνουμε πιο κατανοητοί, αλλά και για να δούμε εφαρμογές. Οι δομές της Python θα παρουσιαστούν με την απαιτούμενη ακρίβεια ώστε να μπορείτε να τις χρησιμοποιήσετε σε όλα σας τα προγράμματα.

Οι τρεις βασικές δομές ελέγχου ροής προγράμματος είναι η δομή της ακολουθίας εντολών, η δομή της απόφασης και η δομή της επανάληψης. Κάποιες από αυτές αντιστοιχούν σε περισσότερες από μία εντολές. Ας δούμε μία μία τις κατηγορίες αυτές, ξεκινώντας με τη δομή της ακολουθίας εντολών, η οποία είναι και η απλούστερη.

6.1 Η ακολουθία εντολών

Οι εντολές σε ένα πρόγραμμα εκτελούνται σειριακά η μία μετά την άλλη ξεκινώντας από την πρώτη. Τελειώνει η εκτέλεση μίας εντολής και ακολουθεί η εκτέλεση της επόμενης. Κάθε εντολή θα εκτελεστεί ακριβώς μία φορά, χωρίς δηλαδή να παραλειφθεί καμία από αυτές ή να επιστρέψουμε για να ξαναεκτελέσουμε κάποια.

Ας φέρουμε στο μυαλό μας ένα ρομποτάκι. Ας σκεφτούμε πώς θα το προγραμματίζαμε για να μεταβεί από το σημείο **A** του Σχήματος 6.1 στο σημείο **T**. Θα του λέγαμε λοιπόν:

Πήγαινε 7 τετράγωνα κάτω.

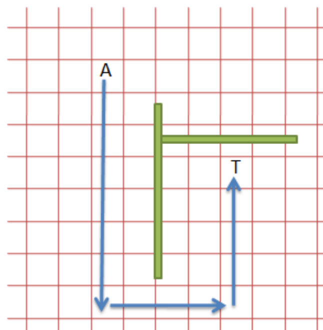
Αφού το έκανε αυτό θα του λέγαμε:

Πήγαινε 4 τετράγωνα δεξιά.

και τέλος

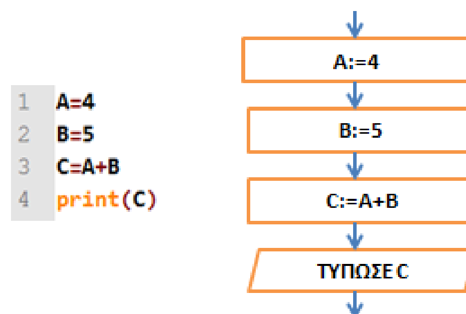
Πήγαινε 4 τετράγωνα επάνω.

Καθεμία εντολή εκτελείται αφού τελειώσει η προηγούμενη και όλες οι εντολές θα εκτελεστούν ακριβώς μία φορά αρχίζοντας από την πρώτη και τερματίζοντας στην τελευταία.



Σχήμα 6.1: Ακολουθιακές εντολές σε ένα ρομπότ.

Ας πάμε στο Σχήμα 6.2 σε ένα παράδειγμα εκτέλεσης ακολουθίας εντολών με Python. Η μεταβλητή **A** παίρνει τιμή 4, στη συνέχεια η μεταβλητή **B** τιμή 5, τις προσθέτουμε και το αποτέλεσμα το εκχωρούμε στη μεταβλητή **C**, την οποία και τυπώνουμε. Δείτε και το διάγραμμα ροής που αντιστοιχεί σε αυτήν την ακολουθία εντολών.



Σχήμα 6.2: Παράδειγμα ακολουθίας εντολών.

6.2 Η δομή της απόφασης

Στη δομή της απόφασης έχουμε μία περισσότερο πολύπλοκη δομή, στην οποία ο έλεγχος του προγράμματος καλείται να επιλέξει ανάμεσα σε δύο ή και περισσότερες διαφορετικές διαδρομές ανάλογα με το αν ισχύει ή όχι κάποια ή κάποιες συνθήκες. Για την υλοποίηση της απόφασης, κάθε γλώσσα μπορεί να έχει μία ή περισσότερες δομές. Η πιο συνηθισμένη είναι η δομή **if**, ενώ υπάρχει συνήθως και μία δομή για πολλαπλή απόφαση. Η Python έχει την **if-elif-else** για να υλοποιήσει την απόφαση, αλλά δεν έχει δομή για την πολλαπλή απόφαση και χρησιμοποιεί την **if-elif-else** για τον σκοπό αυτόν. Παρακάτω θα δούμε την **if-elif-else**, τη **switch** από τη C και πώς θα υλοποιήσει ισοδύναμα τη **switch** η Python.

6.2.1 Η δομή if-elif-else

Η λογική απόφαση υλοποιείται με τη δομή ελέγχου **if-elif-else**. Η δομή της στην απλούστερη μορφή της περιέχει μία μόνο if και είναι αυτή που φαίνεται παρακάτω:

```
if condition:
    statement_1
    statement_2
    ...
    statement_N
```

Με το που θα περάσει ο έλεγχος στη δομή **if**, θα ελεγχθεί η συνθήκη **condition**. Αυτή είναι μία λογική έκφραση η οποία μπορεί να αποτιμηθεί σε **True** ή **False**, ανάλογα με το αν είναι αληθής ή όχι. Αν η συνθήκη **condition** ισχύει, τότε εκτελούνται οι **statement_1, statement_2, ..., statement_N**, ενώ όταν δεν ισχύει, τότε οι εντολές αυτές δεν εκτελούνται.

Παρατηρήστε ότι οι εντολές **statement_1, statement_2, ..., statement_N** είναι στοιχισμένες μεταξύ τους, λίγο πιο μέσα από τη στοίχιση της **if**. Με αυτόν τον τρόπο ομαδοποιούνται οι **statement_1, statement_2, ..., statement_N** σε μία δομή ακολουθίας εντολών, όπως την είχαμε δει παραπάνω. Δεν υπάρχει άλλος τρόπος ομαδοποίησής τους, όπως άγκιστρα ή λέξεις κλειδιά που χρησιμοποιούν οι άλλες γλώσσες.

```
1 if condition:
2     statement1
3     statement2
4 statementN+1
5 statementN+2
6 statementN+3
```

Σχήμα 6.3: Παράδειγμα στοίχισης εντολών στην if.

Στο παράδειγμα του Σχήματος 6.3 οι εντολές **statement1**, **statement2** θα εκτελεστούν μόνο εάν ισχύει η συνθήκη **condition**, ενώ οι εντολές **statementN+1**, **statementN+2**, **statementN+3** θα εκτελεστούν άσχετα με το εάν έχει προηγηθεί η εκτέλεση των **statement1**, **statement2**, δηλαδή άσχετα από το αποτέλεσμα της αποτίμησης της συνθήκης **condition**.

```
1 x=int(input('Δώσε μου έναν αριθμό: '))
2 if (x>0):
3     print ('Ο αριθμός είναι θετικός')
4 if (x<0):
5     print ('Ο αριθμός είναι αρνητικός')
6 if (x==0):
7     print ('Ο αριθμός δεν είναι θετικός, ούτε αρνητικός')
```

Σχήμα 6.4: Πρόσημο αριθμού.

Το πρόγραμμα του Σχήματος 6.4 ελέγχει εάν ένας αριθμός είναι θετικός, αρνητικός ή μηδέν. Στην αρχή ζητείται ένας ακέραιος αριθμός. Ας θεωρήσουμε ότι ο χρήστης πραγματικά δίνει έναν ακέραιο αριθμό, τότε ο έλεγχος πηγαίνει στην πρώτη **if** όπου και ελέγχεται εάν ο **x** είναι θετικός ή όχι. Αν πράγματι είναι θετικός, τότε εκτελείται η πρώτη από τις print η οποία μας πληροφορεί ότι ο **x** είναι θετικός. Στη συνέχεια, ο έλεγχος μεταφέρεται στη δεύτερη **if**, όπου αντίστοιχα ελέγχεται εάν ο αριθμός είναι αρνητικός. Εάν πράγματι είναι, η δεύτερη **print** μάς τυπώνει το ανάλογο μήνυμα. Τέλος, εκτελείται και η τρίτη **if**, για την περίπτωσή που το **x** είναι μηδέν.

Παρατηρήστε ότι μόνο μία από τις τρεις συνθήκες μπορεί κάθε φορά να γίνει αληθής. Γι' αυτό φροντίσαμε εμείς. Δεν θα ήταν σωστό, άλλωστε, το πρόγραμμα, αν μπορούσε να συμβεί κάτι διαφορετικό. Παρατηρήστε, επίσης, ότι και οι τρεις συνθήκες ελέγχονται, απλά οι δύο απορρίπτονται και η μία γίνεται δεκτή. Αν θέλαμε να μας βοηθήσει η Python να κάνουμε κάτι καλύτερο από το να παραθέσουμε μία σειρά από **if**, θα έπρεπε να χρησιμοποιήσουμε μια πιο πολύπλοκη μορφή της **if** με τα **elif** και το **else**. Θα τη δούμε και αυτήν.

Οι λογικές εκφράσεις που χρησιμοποιήθηκαν στο παράδειγμα αυτό είναι μάλλον απλές, ενώ πολλές φορές υπάρχει η ανάγκη για πιο πολύπλοκες συνθήκες, με τη χρήση των τελεστών **and**, **or** και **not**, για λογική σύζευξη, λογική διάζευξη και λογική άρνηση αντίστοιχα. Στο Σχήμα 6.5 φαίνεται ένα πρόγραμμα που ελέγχει εάν ένας αριθμός βρίσκεται στο κλειστό διάστημα **[0,1]**.

```
1 x=float(input('Δώσε μου έναν πραγματικό αριθμό: '))
2 if (x>=0 and x<=1):
3     print ('Ο αριθμός βρίσκεται στο διάστημα [0,1]')
4 if (x<0 or x>1):
5     print ('Ο αριθμός δεν βρίσκεται στο διάστημα [0,1]')
```

Σχήμα 6.5: Αριθμός σε κλειστό διάστημα.

Ας δούμε τώρα μία γενικότερη μορφή της **if**, η οποία θα έκανε τα δύο προηγούμενα παραδείγματα και πιο σύντομα και πιο καλά δομημένα αλλά και πιο γρήγορα στην εκτέλεσή τους. Η μορφή της **if-else** είναι η ακόλουθη:

```
if condition:
  statement_if_1
  statement_if_2
  ...
  statement_if_N
else:
  statement_else_1
  statement_else_2
  ...
  statement_else_N
```

Στη δομή αυτήν, εάν ισχύει η συνθήκη **condition**, θα εκτελεστούν οι εντολές που βρίσκονται ανάμεσα στο **if** και στο **else**, δηλαδή οι **statement_if_1, statement_if_2, ..., statement_if_N**, ενώ εάν η συνθήκη condition δεν ισχύει, τότε θα εκτελεστούν οι εντολές που βρίσκονται αμέσως μετά το **else**, δηλαδή οι **statement_else_1**, **statement_else_2, ..., statement_else_N**.

Ο κώδικας του παραδείγματος στο Σχήμα 6.5 μπορεί τώρα να γραφεί όπως φαίνεται στο Σχήμα 6.6, όπου δεν γίνεται ξανά έλεγχος σε κάποια δεύτερη **if** συνθήκη αν ο πρώτος έλεγχος αποτύχει, αλλά απευθείας πηγαίνουμε στις εντολές **statement_else_1**, **statement_else_2, ..., statement_else_N** οι οποίες και εκτελούνται. Αυτό, πέρα από πιο κομψό, είναι και πιο γρήγορο, αφού δεν χρειάζεται να ελεγχθούν δύο λογικές συνθήκες οι οποίες μάλιστα είναι συμπληρωματικές μεταξύ τους.

```
1 x=float(input('Δώσε μου έναν πραγματικό αριθμό: '))
2 if (x>=0 and x<=1):
3     print ('Ο αριθμός βρίσκεται στο διάστημα [0,1]')
4 else:
5     print ('Ο αριθμός δεν βρίσκεται στο διάστημα [0,1]')
```

Σχήμα 6.6: Αριθμός σε κλειστό διάστημα, έκδοση με else.

Ας απλοποιήσουμε λίγο τους συμβολισμούς και ας θεωρήσουμε ότι μία σειρά από μία ή περισσότερες εντολές μπορεί να συμβολιστεί απλά με το **statements**. Έτσι, ο ορισμός της **if-else** μπορεί να γραφεί πιο σύντομα:

```
if condition:
    statements_if
else:
    statements_else
```

Θα χρησιμοποιούμε παρακάτω σε όλο το κεφάλαιο αυτόν τον συμβολισμό και για συντομία αλλά και πάλι για λόγους κομψότητας. Ας δούμε λοιπόν, με το συμβολισμό αυτό και την τελευταία μορφή της **if**, αυτή στην οποία μπορεί να χρησιμοποιηθεί και το **elif**. Η μορφή της δομής **if-elif-else** είναι η ακόλουθη:

```
if condition_if:
    statements_if
elif condition_elif_1:
    statements_elif_1
elif condition_elif_2:
    statements_elif_2
...
elif condition_elif_N:
    statements_elif_N
else:
    statements_else
```

Στη δομή αυτήν ελέγχεται αν η συνθήκη **condition_if** ισχύει, και εάν ισχύει, εκτελούνται οι εντολές **statements_if**. Εάν όμως η συνθήκη δεν ισχύει, τότε ελέγχεται η συνθήκη **condition_elif_1**. Εάν αυτή ισχύει, τότε εκτελούνται οι εντολές **statements_elif_2**. Αν ούτε και αυτή δεν ισχύει, τότε συνεχίζουμε τους ελέγχους. Ελέγχονται με τον ίδιο τρόπο όσες συνθήκες **condition_elif** χρειαστεί, έως ότου μία από αυτές γίνει αληθής και εκτελεστούν οι αντίστοιχες **statements_elif**. Εάν καμία από αυτές δεν αποτιμηθεί σε αληθής, τότε εκτελούνται οι εντολές **statements_else**.

Το παράδειγμα που ελέγχει εάν ο αριθμός **x** είναι θετικός, αρνητικός ή μηδέν μπορεί να γραφεί κομψότερα όπως φαίνεται στο Σχήμα 6.7. Ο κώδικας αυτός είναι επίσης γρηγορότερος από τον αντίστοιχο κώδικα του Σχήματος 6.6, αφού όταν μία συνθήκη γίνει αληθής, καμία άλλη συνθήκη της δομής δεν θα ελεγχθεί από εκεί και κάτω.

```
1  x=int(input('Δώσε μου έναν αριθμό: '))
2  if (x>0):
3      print ('Ο αριθμός είναι θετικός')
4  elif (x<0):
5      print ('Ο αριθμός είναι αρνητικός')
6  else:
7      print ('Ο αριθμός δεν είναι θετικός, ούτε αρνητικός')
```

Σχήμα 6.7: Πρόσημο αριθμού, έκδοση με if-elif-else.

```
1  x=int(input('Δώσε μου έναν αριθμό: '))
2  if (x>0):
3      print ('Ο αριθμός είναι θετικός')
4  else:
5      if (x<0):
6          print ('Ο αριθμός είναι αρνητικός')
7      else:
8          print ('Ο αριθμός δεν είναι θετικός, ούτε αρνητικός')
```

Σχήμα 6.8: Πρόσημο αριθμού, έκδοση με φωλιασμένα if-else.

Σημειώστε ότι είναι δυνατόν να έχουμε φωλιασμένες δομές **if**. Δηλαδή, καθένα από τα **statements** που αναφέρουμε παραπάνω μπορεί να είναι μία ακόμα δομή **if**. Έτσι, το πρόγραμμα που ελέγχει εάν ένας αριθμός βρίσκεται στο διάστημα **[0,1]** μπορεί να γραφεί όπως φαίνεται στο Σχήμα 6.8, αν και η λύση του Σχήματος 6.7 είναι περισσότερο κομψή. Σε άλλες περιπτώσεις όμως η εμφώλευση είναι πολύ χρήσιμη αλλά και μοναδική επιλογή.

```
1  from math import sqrt
2
3  a=float(input('Δώσε μου το α:' ))
4  b=float(input('Δώσε μου το β:' ))
5  c=float(input('Δώσε μου το γ:' ))
6
7  if a==0:
8      if b!=0:
9          x=-c/b
10         print('Υπάρχει μία ρίζα η', x)
11     elif c==0:
12         print('Κάθε πραγματικός αριθμός αποτελεί λύση')
13     else:
14         print('Κανένας πραγματικός αριθμός δεν αποτελεί λύση')
15 else:
16     D = b**2-4*a*c
17     if D>0:
18         x1=(-b+sqrt(D))/(2*a)
19         x2=(-b-sqrt(D))/(2*a)
20         print('Το τριώνυμο έχει δύο πραγματικές ρίζες, τις:',
21               x1,'και',x2)
22     elif D==0:
23         x=-b/(2*a)
24         print('Το τριώνυμο έχει μία πραγματική ρίζα, την', x)
25     else:
26         c1=-b/(2*a)+(sqrt(-D))*1j
27         c2=-b/(2*a)-(sqrt(-D))*1j
28         print('Το τριώνυμο έχει δύο μιγαδικές ρίζες, τις:',
29               c1,'και', c2)
```

Σχήμα 6.9: Ο κώδικας που υλοποιεί το τριώνυμο.

6.2.2 Παράδειγμα με τη δομή if-elif-else: το τριώνυμο

Ένα πολύ καλό παράδειγμα για να κατανοήσουμε τη δομή **if-then-else** είναι το τριώνυμο. Το έχουμε χρησιμοποιήσει και στο κεφάλαιο με τα διαγράμματα ροής, θα χρησιμοποιήσουμε το ίδιο και εδώ στην Python, τόσο για να το δούμε σε πραγματικό κώδικα όσο και γιατί αποτελεί ένα πολύ καλό μέσο για να κατανοήσουμε πλήρως τη δομή, αφού έχει δύο δομές **if-elif-else** φωλιασμένες μέσα σε μία δομή **if-else**. Ο κώδικας φαίνεται στο Σχήμα 6.9.

Αρχικά δηλώνουμε στη γραμμή 1 ότι θα χρειαστούμε την τετραγωνική ρίζα

(**sqrt**) από τη βιβλιοθήκη των μαθηματικών (**math**). Αν και δεν έχουμε μιλήσει για τις βιβλιοθήκες, νομίζω ότι δεν θα δυσκολευτείτε να καταλάβετε κάτι εδώ. Ίσως απλά να νιώσετε κάποια έκπληξη που η τετραγωνική ρίζα δεν είναι ενσωματωμένη στην Python, αλλά έτσι είναι.

Μετά ζητάμε τους τρεις συντελεστές του τριωνύμου, τα **a**, **b** και **c**. Η πρώτη απόφαση που πρέπει να πάρουμε είναι εάν το **a** είναι μηδέν ή όχι. Στην περίπτωση που το **a** είναι μηδέν, η λύση είναι πολύ διαφορετική από την περίπτωση που το **a** δεν είναι μηδέν, αφού τότε εκφυλίζεται σε πρωτοβάθμια. Στις γραμμές 7 και 15 η ροή του προγράμματος πρέπει να αποφασίσει ποια από τις δύο διαδρομές πρέπει να ακολουθήσει. Αν το **a** είναι μηδέν, θα εκτελέσει τις γραμμές 7-14, όπου υπάρχει η λύση για την πρωτοβάθμια εξίσωση, ενώ αν το **a** δεν είναι μηδέν, θα εκτελέσει τον κώδικα στις γραμμές 16-27, στις οποίες βρίσκεται η λύση για τη δευτεροβάθμια.

Ας ξεκινήσουμε λοιπόν με την πρώτη περίπτωση, όπου η λύση εκφυλίζεται σε πρωτοβάθμια. Και εδώ η ροή εκτέλεσης πρέπει να εξετάσει τρεις υποπεριπτώσεις:

- αν το **b** είναι διαφορετικό του μηδενός, η εξίσωση έχει μία πραγματική λύση,
- αν το **b** είναι μηδέν και το **c** είναι μηδέν, τότε κάθε πραγματικός αριθμός είναι λύση,
- αν το **b** είναι μηδέν και το **c** είναι διαφορετικό του μηδενός, δεν υπάρχει πραγματικός αριθμός που είναι λύση.

Έτσι, στις γραμμές 8, 11 και 13 χρησιμοποιείται η **if-elif-else** για να διακρίνει τις τρεις αυτές περιπτώσεις. Αφού ο έλεγχος της ροής επιλέξει έναν από τους τρεις δρόμους, τότε υπολογίζεται η αντίστοιχη λύση και τυπώνεται το κατάλληλο μήνυμα. Στην πρώτη υποπερίπτωση το **b** είναι διαφορετικό του μηδενός και υπολογίζεται μια πραγματική λύση. Το **elif** που ακολουθεί θα εκτελεστεί μόνο αν το **if** αποτύχει, δηλαδή το **b** είναι ίσο με μηδέν.

Άρα, στον έλεγχο που κάνει η **elif** δεν χρειάζεται να ελεγχθεί το **b**, το οποίο είναι σίγουρα μηδέν. Έτσι, ελέγχεται μόνο το **c**. Σε κάθε άλλη περίπτωση θα εκτελεστεί το **else**. Και ποια είναι η κάθε άλλη περίπτωση; Μόνο μία έμεινε, το **c** να είναι διάφορο του μηδενός.

Δεν χρειάζεται πάλι να πούμε ότι το **b** είναι μηδέν, αφού αλλιώς δεν θα βρισκόμασταν εδώ και θα είχε γίνει αληθής η πρώτη συνθήκη της **if-elif-else**. Ούτε όμως και ότι το **a** είναι μηδέν, αλλιώς δεν θα είχαμε μπει καθόλου σε αυτήν την **if-elif-else**. Η μορφή αυτή της **if** είναι πολύ ευέλικτη και μπορεί να

εκφράσει με κομψότητα και ακρίβεια, χωρίς φλυαρία αυτό που θέλουμε. Αρκεί να τη χρησιμοποιήσουμε σωστά.

Ας επιστρέψουμε τώρα στην εξωτερική **if** και ας πάμε στην **else** της που βρίσκεται στη γραμμή 15. Εκεί πηγαίνει ο έλεγχος όταν το **a** είναι διάφορο του μηδενός. Ας μη δούμε αναλυτικά την **if-elif-else** που είναι φωλιασμένη μέσα της, αφού τώρα πια πρέπει να μπορούμε και μόνοι μας να το κάνουμε. Την έχουμε συζητήσει και στο κεφάλαιο με τα διαγράμματα ροής.

Ας μείνουμε μόνο στις γραμμές 26-27 όπου υπολογίζονται οι μιγαδικές ρίζες. Έχουμε μιλήσει για μιγαδικούς αριθμούς στο κεφάλαιο με τις δομές δεδομένων της Python. Παρατηρήστε πόσο κομψός γίνεται ο κώδικας με τη χρήση των μιγαδικών μεταβλητών, αφού ο συμβολισμός των μεταβλητών στον κώδικά μας μοιάζει με αυτόν που θα χρησιμοποιούσαμε αν γράφαμε τη μεταβλητή στο χαρτί. Μη σας παραξενέψει το **1j**, ανατρέξτε πίσω στο βιβλίο στους μιγαδικούς αριθμούς να δείτε γιατί το γράφουμε έτσι. Το ζητά η γλώσσα για να αποφύγουμε αμφισημίες και τη σύγχυση με τη μεταβλητή **j**.

6.2.3 Η απόφαση με πολλαπλές επιλογές

Η απόφαση με πολλαπλές επιλογές σαν δομή εμφανίζεται σε πολλές γλώσσες, όπως η Pascal, η Ada, η C, η C++ η C# και η Java, και εμφανίζεται ως **switch**, **case**, **select** ή **inspect**. Η Python δεν την υποστηρίζει και είμαστε υποχρεωμένοι να χρησιμοποιήσουμε την **if-elif-else** για τον σκοπό αυτόν. Εμείς θα δούμε εδώ τη **switch**, η οποία είναι η δομή που χρησιμοποιεί η C.

Και θα πάμε απευθείας σε ένα παράδειγμα. Νομίζουμε ότι είναι αρκετά απλή και ότι αυτό αρκεί, μετά και την εμπειρία μας με τη **if-elif-else**. Στο Σχήμα 6.10 υλοποιούμε έναν πίνακα επιλογών όπου ο χρήστης έχει ήδη επιλέξει μία από αυτές η οποία και έχει αποθηκευτεί στη μεταβλητή **menu**. Η μεταβλητή **menu** δηλώνεται αμέσως μετά τη **switch** σαν τη μεταβλητή η οποία θα συγκριθεί παρακάτω με τη μεταβλητή ή τη σταθερά που ακολουθεί κάθε ένα από τα **case**. Έτσι, αν η μεταβλητή **menu** ισούται με 1 θα εκτελεστεί ο κώδικας που ακολουθεί το πρώτο **case**. Αν η **menu** ισούται με 2 θα εκτελεστεί ο κώδικας που ακολουθεί το δεύτερο **case** κ.ο.κ. Αν το **menu** δεν ισούται με καμία από τις μεταβλητές ή σταθερές που ακολουθούν το κάθε **case**, τότε θα εκτελεστεί ο κώδικας που ακολουθεί τη **default**. Το ισοδύναμο πρόγραμμα σε γλώσσα Python χρησιμοποιώντας τη δομή **if-elif-else** φαίνεται στο Σχήμα 6.11.

```
1  switch(menu)
2  {
3      case 1: printf("selection 1");
4              break;
5      case 2: printf("selection 2");
6              break;
7      case 3: printf("selection 3");
8              break;
9      default: printf("not a valid selection");
10 }
```

Σχήμα 6.10: Παράδειγμα με τη δομή switch.

```
1 if menu==1:
2   print('selection 1')
3 elif menu==2:
4   print('selection 2')
5 elif menu==3:
6   print('selection 3')
7 else:
8   print('not a valid selection')
```

Σχήμα 6.11: Ισοδύναμο σε Python με τη δομή πολλαπλής επιλογής.

6.3 Δομή επανάληψης

Θα μπορούσαμε να πούμε ότι οι τρεις πιο χαρακτηριστικές δομές επανάληψης είναι η **for**, η **while** και η **do while**. Η **for** και η **while** υποστηρίζονται από την Python και θα τις δούμε εκεί. Η **do while** δεν υποστηρίζεται, οπότε θα δούμε ένα παράδειγμά της σε γλώσσα C. Οι δομές που υλοποιούν κάποιας μορφής επανάληψη λέγονται **βρόχοι (loops)** και εάν μέσα σε ένα βρόχο βρίσκονται ένας ή περισσότεροι άλλοι βρόχοι, τότε τους αποκαλούμε **φωλιασμένους βρόχους (nested loops)**.

6.3.1 Η δομή for

Ορίζει μία ακολουθία εντολών που επαναλαμβάνεται τόσες φορές όσες καθορίζεται από τις παραμέτρους της εντολής. Τη δομή **for** τη χαρακτηρίζει

μία μεταβλητή (συνήθως τη λέμε **μεταβλητή επανάληψης (iteration variable)** η οποία αρχικοποιείται σε κάποια τιμή και σε κάθε επανάληψη αυξάνει την τιμή της σταθερά, τόσο όσο δηλώνεται από ένα προαιρετικό μέρος της εντολής, το **βήμα**. Εάν το βήμα δεν δηλωθεί, τότε θεωρείται ότι είναι ίσο με 1. Εάν το βήμα έχει αρνητική τιμή, τότε σε κάθε επανάληψη ο μετρητής μειώνεται κατά όσο ορίζει το βήμα. Η έξοδος από τη δομή γίνεται όταν ο μετρητής φτάσει σε προκαθορισμένη τιμή. Στην Python έχουμε μεγάλη ευελιξία στη χρήση της **for**. Μία σύνταξη που σε εισαγωγικό πλαίσιο μας αρκεί είναι η ακόλουθη:

```
for i in range([A,]B,step)
  statements
```

όπου **i** η μεταβλητή της επανάληψης, **A** η αρχική τιμή της (οι αγκύλες σημαίνουν ότι είναι προαιρετική, αν δεν δηλωθεί θεωρείται ότι είναι μηδέν), **step** το βήμα και **statements** οι εντολές που αποτελούν το σώμα της επανάληψης, οι οποίες είναι και στοιχισμένες μεταξύ τους και λίγο πιο δεξιά από τη στοίχιση της **for**.

Ας δούμε δύο παραδείγματα: Ο πρώτος βρόχος στο Σχήμα 6.12 θα τυπώσει τους αριθμούς από 1 έως 10, ενώ ο δεύτερος μόνο τους περιττούς.

```
1 for i in range(1,11):
2   print(i)
3
4 for i in range(1,11,2):
5   print(i)
6
7 for i in range(0,5):
8   for j in range (0,5):
9     print (i,j)
```

Σχήμα 6.12: Παραδείγματα με τη δομή for.

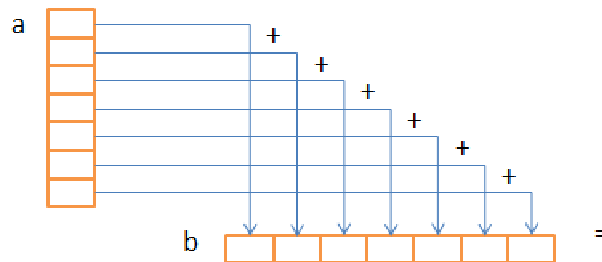
Θα παρατηρήσατε βέβαια ότι ο μετρητής (**i** στην περίπτωσή μας) αρχικοποιήθηκε στο 1, που είναι η πρώτη από τις τιμές στην παρένθεση του **range**, και έφτασε έως την τιμή 11, που είναι η δεύτερη από τις τιμές στην παρένθεση του **range**, χωρίς όμως να πάρει την τιμή αυτή. Στο δεύτερο βρόχο, η τρίτη τιμή μέσα στο range είναι το βήμα. Αυτό είναι που μας εξαίρεσε τους άρτιους αριθμούς.

Το τρίτο παράδειγμα στο τέλος του Σχήματος 6.12 δείχνει έναν φωλιασμένο βρόχο που τυπώνει όλα τα διατεταγμένα ζεύγη ακέραιων από **(0,0)** έως

(4,4) και μάλιστα με την ακόλουθη σειρά: **(0,0), (0,1), (0,2), (0,3), (0,4), (1,0), (1,1), (1,2) ... (4,4)**. Αλλά ένα λίγο μεγαλύτερο παράδειγμα θα δούμε στην επόμενη ενότητα.

6.3.2 Παραδείγματα με τη δομή for: εσωτερικό γινόμενο, λίστα ακέραιων, προπαίδεια

Το **εσωτερικό γινόμενο** δύο διανυσμάτων δίνεται από το άθροισμα των γινομένων των στοιχείων των διανυσμάτων που βρίσκονται στην ίδια θέση μέσα στα δύο διανύσματα (δείτε στο Σχήμα 6.13).



Σχήμα 6.13: Σχηματική παράσταση για το εσωτερικό γινόμενο δύο διανυσμάτων.

Πιο μαθηματικά, για τα διανύσματα :

$$\vec{a} = (a_1, a_2, \ldots\ a_N),\ \ \vec{b} = (b_1, b_2, \ldots\ b_N)$$

μεγέθους **N**, το εσωτερικό γινόμενο δίνεται από τον τύπο:

$$innerProduct(\vec{a}, \vec{b}) = \sum_{i=1}^{N} a_i b_i$$

Στον κώδικα στο Σχήμα 6.14 το εσωτερικό γινόμενο των **a** και **b** υπολογίζεται με δύο τρόπους: Έναν πιο παραδοσιακό και έναν που χρησιμοποιεί περισσότερο κλήσεις και μεθόδους της Python. Αυτό δεν σημαίνει ότι ο δεύτερος τρόπος είναι καλύτερος από τον πρώτο.

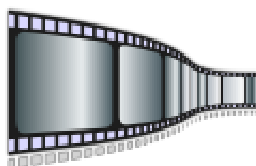
```
1  a=[1,3,5,7,9]
2  b=[2,4,6,8,0]
3  c=0
4  for i in range(len(a)):
5      c+=a[i]*b[i]
6  print (c)
7
8  C=[]
9  for i,j in zip(a,b):
10     C.append(i*j)
11 c=sum(C)
12 print (c)
```

Σχήμα 6.14: Κώδικας για το εσωτερικό γινόμενο δύο διανυσμάτων.

Με τον πρώτο τρόπο (γραμμές 3-6) η μεταβλητή **c** λειτουργεί ως άθροισμα-συσσωρευτής μέσα σε μία δομή επανάληψης **for**. Η **c** αρχικοποιείται στο μηδέν και κάθε γινόμενο **a[i]b[i]** που υπολογίζεται μέσα στη **for** συσσωρεύεται στο **c**, προστίθεται δηλαδή στην παλιά του τιμή και δημιουργεί την καινούργια. Στο τέλος των επαναλήψεων όλα τα **N** γινόμενα έχουν προστεθεί στο **c** και το αποτέλεσμα έχει υπολογιστεί.

Κατά τον δεύτερο τρόπο χρησιμοποιούμε τη **for** και την **zip** ώστε σε κάθε επανάληψη να έχουμε πρόσβαση με το **i** στα στοιχεία του **a** και με το **j** στα αντίστοιχα στοιχεία του **b**, μέσα στην ίδια επανάληψη. Τα γινόμενα **i*j** κρατιούνται σε μία λίστα. Στο τέλος της επανάληψης υπολογίζεται το άθροισμα των στοιχείων αυτής της λίστας, το οποίο είναι και το ζητούμενο εσωτερικό γινόμενο.

Αν θέλετε να δείτε ένα ακόμα παράδειγμα χρήσης του **for**, κάντε κλικ στην Ταινία 6.1 και δείτε πώς από μία λίστα ακέραιων αριθμών θα φτιάξουμε δύο λίστες, η μία να περιέχει τους περιττούς αριθμούς της αρχικής λίστας και η δεύτερη τους άρτιους.



Ταινία 6.1: Λίστα περιττών και άρτιων.

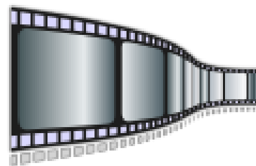
http://repfiles.kallipos.gr/file/9283

Ο κώδικας που το κάνει αυτό φαίνεται στο Σχήμα 6.15.

```
1 odd=[]
2 even=[]
3 for i in L:
4     if i%2==0:
5         even.append(i)
6     else:
7         odd.append(i)
```

Σχήμα 6.15: Διαχωρισμός μιας λίστας ακέραιων σε λίστες άρτιων και περιττών.

Επίσης, αν θέλετε να δείτε ένα παράδειγμα με φωλιασμένους βρόχους, κάντε κλικ στην Ταινία 6.2, για να δείτε πώς θα φτιάξετε ένα πρόγραμμα που να τυπώνει την προπαίδεια.



Ταινία 6.2: Προπαίδεια.

http://repfiles.kallipos.gr/file/9284

Ο κώδικας που το κάνει αυτό φαίνεται στο Σχήμα 6.16.

```
for i in range(11):
  for j in range(11):
    print ('{0:2d}x{1:2d}={2:3d}'.format(i,j,i*j),
                                          end='    ')
  print('')
```

Σχήμα 6.16: Η προπαίδεια.

Πολλά ακόμα παραδείγματα με **for** και φωλιασμένους βρόχους θα δούμε στο κεφάλαιο με τα μαθηματικά προβλήματα με τους πίνακες.

6.3.3 Η δομή while

Η δομή **while** είναι πολύ κοντινή σαν δομή με την επαναληπτική δομή **for**. Η λειτουργία της είναι να εκτελεί κάποιες εντολές για όσο διάστημα μια συνθήκη είναι αληθής. Η σύνταξή της είναι η ακόλουθη:

while (condition):
  statements

όπου η **condition** είναι μία συνθήκη και το **statements** μία ακολουθία εντολών. Όταν ο έλεγχος του προγράμματος εισέρχεται στη **while**, τότε ελέγχεται εάν η συνθήκη **condition** ισχύει ή όχι. Εάν η συνθήκη δεν ισχύει, τότε ο έλεγχος φεύγει από τη **while**. Αν ισχύει, τότε εκτελούνται οι εντολές **statements** και μετά ο έλεγχος μεταφέρεται στην αρχή της δομής **while**, όπου και ξαναελέγχεται η συνθήκη. Οι **statements** δηλαδή θα εκτελούνται για όσο καιρό η συνθήκη **condition** ισχύει.

```
1  i=1
2  while i<11:
3    print(i)
4    i+=1
5
6  i=1
7  while i<11:
8    print(i)
9    i+=2
10
11 i=1
12 while i<5:
13   j=1
14   while j<5:
15     print (i,j)
16     j+=1
17   i+=1
```

Σχήμα 6.17: Παραδείγματα με τη while.

Στο Σχήμα 6.17 φαίνεται ο κώδικας του Σχήματος 6.12 υλοποιημένος με τη δομή **while**. Οτιδήποτε μπορούμε να φτιάξουμε με for μπορούμε να το κάνουμε και με τη **while**. Αρκεί να αρχικοποιήσουμε εμείς τη μεταβλητή του βρόχου και να φροντίσουμε και την αύξηση του βήματος μόνοι μας. Νομίζω δεν θα σας δυσκολέψει να το καταλάβετε μόνοι σας παρατηρώντας τους δύο κώδικες. Ας δούμε μαζί κάτι πιο δύσκολο, την εύρεση μίας ρίζας ενός πολυωνύμου με τη μέθοδο της διχοτόμησης.

6.3.4 Παράδειγμα με τη δομή while: ρίζες πολυωνύμου με τη μέθοδο της διχοτόμησης

Ας δούμε ένα ενδιαφέρον παράδειγμα από τα μαθηματικά: Θα υπολογίζουμε τη ρίζα ενός πολυωνύμου, έστω 4ου βαθμού, όταν γνωρίζουμε ότι αυτή βρίσκεται μέσα σε ένα συγκεκριμένο διάστημα και ότι η ρίζα αυτή είναι η μοναδική μέσα στο διάστημα αυτό. Θα χρησιμοποιήσουμε για τον σκοπό αυτόν τη μέθοδο της διχοτόμησης.

Σύμφωνα με τη μέθοδο αυτήν, εκτελώντας συνεχείς επαναλήψεις, περιορίζουμε συνέχεια και βηματικά το διάστημα μέσα στο οποίο θεωρούμε ότι βρίσκεται η ρίζα. Όταν φτάσουμε σε ένα διάστημα τόσο μικρό που να θεωρούμε ότι έχουμε προσεγγίσει ικανοποιητικά τη λύση, σταματάμε τις επαναλήψεις, θεωρούμε ότι βρήκαμε το αποτέλεσμα και το επιστρέφουμε.

```
1  p=[]
2  for i in range(4,-1,-1):
3      coeff=float(input('Συντελεστής του x^'+str(i)+':'))
4      p.insert(0,coeff)
5
6  a=float(input('Αρχή του διαστήματος: '))
7  b=float(input('Τέλος του διαστήματος: '))
8  error=float(input('Επιτρεπτό σφάλμα: '))
9
10 while abs(a-b)>error:
11     m=(a+b)/2
12     p_a=p[0]+p[1]*a+p[2]*a**2+p[3]*a**3+p[4]*a**4
13     p_m=p[0]+p[1]*m+p[2]*m**2+p[3]*m**3+p[4]*m**4
14     if p_a*p_m<=0:
15         b=m
16     else:
17         a=m
18     print (a,b,p_a,p_m)
19
20 print ('Η ρίζα είναι η: ',(a+b)/2)
```

Σχήμα 6.18: Εύρεση ρίζας πολυωνύμου με τη μέθοδο της διχοτόμησης.

Η κεντρική ιδέα του αλγορίθμου βασίζεται στο γεγονός ότι, αν γνωρίζουμε ότι σε ένα διάστημα **(a,b)** υπάρχει για το πολυώνυμα **p(x)** ακριβώς μία ρίζα, τότε το γινόμενο **p(a)p(b)** είναι αρνητικό, βασιζόμενοι στο ότι η συνάρτηση είναι

συνεχής στο διάστημα **(α,β)**. Ας δούμε τις υπόλοιπες λεπτομέρειες πάνω στον κώδικα, ο οποίος φαίνεται στο Σχήμα 6.18.

Αρχικά, πρέπει να εισάγουμε τα δεδομένα. Χωρίς βλάβη της γενικότητας και για απλοποίηση, θεωρήσαμε ότι ο βαθμός του πολυωνύμου είναι 4. Έτσι ζητάμε έναν έναν τους συντελεστές του πολυωνύμου χρησιμοποιώντας μία επανάληψη **for**. Τα όριά της είναι από 4 έως -1, ώστε η μεταβλητή της επανάληψης να πάρει τις τιμές **5,4,3,2,1,0**. Μην ξεχνάτε ότι την τελευταία τιμή (το -1) δεν την παίρνει. Επίσης θα παρατηρήσατε ότι το βήμα το θέσαμε -1, κάτι απαραίτητο ώστε οι τιμές της μεταβλητής της επανάληψης να μειώνονται κατά 1 σε κάθε της βήμα. Μετά την εισαγωγή κάθε τιμής, η τιμή αυτή τοποθετείται στην πρώτη θέση της λίστας **p**. Δηλαδή, μετά το τέλος των επαναλήψεων ο σταθερός όρος του πολυωνύμου θα βρίσκεται στη θέση 0 της λίστας, ο συντελεστής του **x** στη θέση 1 κ.ο.κ.

Στη συνέχεια ζητάμε τις τιμές της αρχής και του τέλους του διαστήματος μέσα στο οποίο ξέρουμε ότι υπάρχει η ρίζα αλλά και της επιτρεπτής τιμής του σφάλματος.

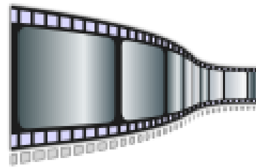
Έπειτα, αρχίζουν οι επαναλήψεις. Όσο το διάστημα **(a,b)** είναι μεγαλύτερο του επιτρεπόμενου σφάλματος (έλεγχος από το **while** στη γραμμή 10), υπολογίζουμε το μέσο του διαστήματος (γραμμή 11) και τις τιμές του πολυωνύμου στην αρχή και στη μέση του διαστήματος (γραμμές 12-13). Εάν το γινόμενο των δύο αυτών τιμών είναι μικρότερο ή ίσο του μηδενός, τότε η ρίζα βρίσκεται στο πρώτο μισό του **(a,b)**.

Άρα ”πετάμε” το δεξί μισό του διαστήματος **(a,b)**, κάνοντας το **b** ίσο με **m** (γραμμή 15). Η επόμενη επανάληψη θα γίνει για το διάστημα **(a,b)**, όπου όμως το **b** θα έχει τη νέα του τιμή. Σε αντίθετη περίπτωσή, που το γινόμενο των δύο τιμών είναι μεγαλύτερο του μηδενός, το αριστερό διάστημα θα πρέπει να ”πεταχτεί”, με τρόπο ακριβώς όμοιο, θέτοντας **a** ίσο με **m** (γραμμές 16-18). Το μόνο που απέμεινε είναι να τυπώσουμε το αποτέλεσμα, που μπορεί να είναι οποιοσδήποτε αριθμός μέσα στο τελευταίο διάστημα **(a,b)** όπως διαμορφώθηκε με το πέρας των επαναλήψεων. Από όλες αυτές τις τιμές θα προτιμήσουμε να επιστρέψουμε ως λύση το μέσο του διαστήματος (γραμμή 20).

Ελπίζουμε να μην ήταν δύσκολο. Ας δούμε και ένα παράδειγμα με μία Ταινία. Στην Ταινία 6.3 θα χωρίσουμε έναν θετικό ακέραιο αριθμό στα ψηφία του. Στο Σχήμα 6.19 φαίνεται ο κώδικας που το κάνει αυτό.

```
1 x=int(input('Δώσε μου τον αριθμό:'))
2 while x>0:
3     print (x%10)
4     x=x//10
```

Σχήμα 6.19: Διάσπαση ενός αριθμού στα ψηφία του.



Ταινία 6.3: Διάσπαση αριθμού στα ψηφία του.

http://repfiles.kallipos.gr/file/22386

6.3.5 Η δομή do while

Μία ακόμα ενδιαφέρουσα δομή, η οποία δεν υπάρχει στην Python, είναι η **do while**. Δανειζόμαστε τον όρο από τη C, αφού σε διαφορετικές γλώσσες έχει διαφορετικά ονόματα. Η **do while** δεν διαφέρει πολύ από τη **while**. Η βασική της διαφορά είναι ότι ο έλεγχος της συνθήκης γίνεται στην αρχή και όχι στο τέλος. Αυτό σημαίνει και ότι οι εντολές μέσα στο βρόχο θα εκτελεστούν τουλάχιστον μία φορά και μετά θα ελεγχθεί η συνθήκη για το αν θα συνεχιστεί η επανάληψη ή όχι. Αυτό την κάνει περισσότερο κατάλληλη για κάποιες ανάγκες από ό,τι η **do while**. Φυσικά ό,τι μπορεί να κάνει κανείς με την **do while** μπορεί να το κάνει και με τη **while**.

```
1 x=int(input('Δώσε μου έναν θετικό αριθμό: '))
2 while x<=0:
3     print('Ο αριθμός δεν είναι θετικός, ξαναπροσπάθησε.')
4     x=int(input('Δώσε μου έναν θετικό αριθμό: '))
```

Σχήμα 6.20: Αμυντικός προγραμματισμός με τη χρήση της while.

Ας δούμε ένα παράδειγμα: Την τεχνική αυτή τη λέμε αμυντικό προγραμματισμό, αφού τη χρησιμοποιούμε για να ελέγξουμε αν κάποιος έχει δώσει επιτρεπτές τιμές σε μία εντολή εισόδου, π.χ. την input. Αν θελήσουμε να κάνουμε ένα πρόγραμμα που θα ζητά από τον χρήστη ένα θετικό αριθμό και θα

ελέγχει αν πράγματι ο αριθμός που του έδωσε ο χρήστης είναι θετικός, τότε θα γράψουμε τον κώδικα του Σχήματος 6.20.

Ο κώδικας αυτός ζητά, αρχικά, από τον χρήστη έναν θετικό αριθμό και στη συνέχεια ελέγχει με το **while** αν ο αριθμός που δόθηκε είναι θετικός ή όχι. Αν είναι θετικός, τότε δεν μπαίνει μέσα στο **while**. Αν όμως δεν είναι θετικός, τότε μπαίνει μέσα στο **while** και τυπώνει ένα μήνυμα σφάλματος και ξαναζητάει τον αριθμό. Παρατηρήστε ότι, αν ο χρήστης ξαναδώσει λάθος αριθμό, το πρόγραμμα δεν θα βγει έξω από το **while**, οπότε θα ζητήσει εκ νέου να δοθεί θετικός αριθμός, επαναλαμβάνοντας τις δύο τελευταίες γραμμές. Αν δοθεί θετικός αριθμός, τότε το σώμα του **while** δεν θα ξαναεκτελεστεί και το πρόγραμμα θα προχωρήσει παρακάτω έχοντας έναν θετικό αριθμό στη θέση του **x**.

```
1  do
2  {
3      printf("Δώσε μου έναν θετικό αριθμό: ");
4      scanf("%d",&x);
5      if (x<=0) printf("Λάθος, ξαναπροσπάθησε. ");
6  } while (x<=0);
```

Σχήμα 6.21: Αμυντικός προγραμματισμός με τη χρήση της do while.

Αν τώρα θέλαμε να κάνουμε το ίδιο ακριβώς πρόγραμμα με την **do while**, θα μπορούσαμε να δομήσουμε τον κώδικα πιο καλά. Το ίδιο πρόγραμμα σε γλώσσα C με την **do while** φαίνεται στο Σχήμα 6.21. Μέσα στον βρόχο πια, και όχι πριν από αυτόν, εκτελείται η εντολή εισόδου (**scanf** στο σχήμα). Όταν υπάρξει πρόβλημα, εμφανίζεται ένα μήνυμα λάθους και ο έλεγχος δεν φεύγει από τον βρόχο, αλλά πηγαίνει πάλι στην αρχή του για να εκτελέσει την ίδια διαδικασία.

Αν χρειάζεστε περισσότερο υλικό σχετικό με εντολές ελέγχου ροής εκτέλεσης της Python μπορείτε να ανατρέξετε στο κεφάλαιο 7 του βιβλίου [1], στο κεφάλαιο 5 του βιβλίου [2], στο κεφάλαιο 4 του βιβλίου [3], στα κεφάλαια 4,5 του βιβλίου [4], στο κεφάλαιο 3 του βιβλίου [5] και στο κεφάλαιο 4 του βιβλίου [6].

Ασκήσεις που μπορείτε να κάνετε μόνοι σας

- Να φτιάξετε ένα πρόγραμμα σε Python που να υπολογίζει την πρώτη παράγωγο ενός πολυωνύμου. Θα δέχεται δηλαδή σαν

είσοδο τους συντελεστές του πολυωνύμου και θα υπολογίζει και θα τυπώνει τους συντελεστές της πρώτης παραγώγου.

- Με βάση την μέθοδο της διχοτόμησης, να υλοποιήσετε τη μέθοδο **Newton-Raphson** για τον υπολογισμό της τετραγωνικής ρίζας.
- Να υλοποιήσετε ένα πρόγραμμα με το οποίο να ελέγχεται εάν ένα έτος είναι δίσεκτο ή όχι. Ένα έτος είναι δίσεκτο εάν διαιρείται με το 4. Εξαιρούνται όσα διαιρούνται με το 100. Από την εξαίρεση εξαιρούνται όσα διαιρούνται με το 400. Το 1996, για παράδειγμα, είναι δίσεκτο διότι διαιρείται με το 4. Το 1900 δεν είναι δίσεκτο, διότι διαιρείται με το 4 αλλά και με το 100. Το 2000 είναι δίσεκτο, διότι, παρόλο που διαιρείται με το 100, διαιρείται και με το 400.
- Να φτιάξετε ένα πρόγραμμα το οποίο να δέχεται σαν είσοδο μία χρονική στιγμή **A** του 24ώρου (ώρα, λεπτά, δευτερόλεπτα) και μία χρονική περίοδο **B** (ώρα, λεπτά, δευτερόλεπτα) που ξεκινά από την **A**. Βρείτε ποια χρονική στιγμή του 24ωρου τελειώνει η χρονική περίοδος **B**. Αν δηλαδή έχουμε την χρονική στιγμή 20:04:50 και θέλουμε να προσθέσουμε την περίοδο 5:04:40, το αποτέλεσμα πρέπει να είναι 01:09:30.

Βιβλιογραφία

1. Allen B. Downey (2012). **Think Python**. Publisher: O'Reilly Media.
2. Brian Heinold (2012). **Introduction to Programming Using Python**. Publisher: Mount St. Mary's University, Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.
3. Ellis Horowitz (1993). **Βασικές Αρχές Γλωσσών Προγραμματισμού**. 2η έκδοση, Εκδόσεις Κλειδάριθμος.
4. Cody Jackson (2011). **Learning to Program Using Python**. Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.
5. Αχιλλέας Καμέας (2000). **Τεχνικές Προγραμματισμού**. Τόμος Β. ΠΛΗ-10, Ελληνικό Ανοικτό Πανεπιστήμιο.
6. Eric Roberts. (2004). **Η Τέχνη και Επιστήμη της C**. Μετάφραση: Γιώργος Στεφανίδης, Παναγιώτης Σταυρόπουλος, Αλέξανδρος Χατζηγεωργίου, Εκδόσεις Κλειδάριθμός.

Κεφάλαιο 7: Συναρτήσεις

7.1 Κάτι σαν κοινός παρονομαστής

Ας ξεκινήσουμε δικαιολογώντας τον τίτλο **κοινός παρονομαστής**. Φανταστείτε ότι έχετε ένα μεγάλο πρόγραμμα στο οποίο σε διαφορετικά του σημεία γίνεται η ίδια λειτουργία, ή αν προτιμάτε επαναλαμβάνεται ο ίδιος (ακριβώς;) κώδικας. Ένα πρόγραμμα, για παράδειγμα, χρειάζεται να υπολογίσει σε διαφορετικά μέρη τον δεκαδικό λογάριθμο της κυβικής ρίζας ενός αριθμού. Θεωρήστε, μάλιστα, ότι δεν υπάρχει κάποια υποστήριξη από τη γλώσσα προγραμματισμού για τον υπολογισμό της κυβικής ρίζας ή του δεκαδικού λογαρίθμου. Στα δύο παραδείγματα αυτά ο κώδικας δεν είναι πολύ μεγάλος, δεν είναι πάντως και πολύ μικρός. Σε άλλες περιπτώσεις, όμως, ο κώδικας αυτός μπορεί να είναι και ιδιαίτερα μεγάλος.

Θα ήταν λοιπόν πολύ φυσικό να αναζητήσουμε έναν τρόπο ώστε να βγάλουμε αυτόν τον κώδικα ως κοινό παρονομαστή και να τον τοποθετήσουμε σε κάποιο σημείο του προγράμματός μας και να τον καλούμε από εκεί όσες φορές χρειαστεί και σε όσα σημεία του προγράμματος υπάρχει η ανάγκη να το κάνουμε. Αν, λοιπόν, για παράδειγμα, χρειαζόμαστε να τυπώνεται στην οθόνη το:

University of Ioannina, Dept. of Computer Science & Engineering

σε διάφορα σημεία, θα μπορούσαμε να φτιάξουμε μία συνάρτηση που κάνει κάτι τέτοιο και να την καλούμε από όποιο σημείο του προγράμματός μας επιθυμούμε. Το ίδιο και αν εμφανιζόταν σε πολλά σημεία ο υπολογισμός του αθροίσματος **1+2+3+...+10**.

Συναρτήσεις υποστηρίζουν οι περισσότερες γλώσσες προγραμματισμού (εντάξει, ας πούμε όλες, δεν είναι σημαντικό το σφάλμα που κάνουμε) με τη

μία ή την άλλη μορφή. Κάποιες από αυτές, όπως η Pascal, κάνουν διαχωρισμό ανάμεσα σε συναρτήσεις και διαδικασίες. Διαδικασίες θεωρούμε τις συναρτήσεις που δεν επιστρέφουν τίποτε. Στα παραδείγματά μας δηλαδή ο υπολογισμός της κυβικής ρίζας και του λογαρίθμου θα ορίζονταν ως συναρτήσεις διότι επιστρέφουν σαν αποτέλεσμα σε αυτόν που τις κάλεσε έναν πραγματικό αριθμό η καθεμία, κάτι δηλαδή το οποίο μπορεί να αποθηκευτεί σε μία απλή μεταβλητή ή και σε έναν πιο σύνθετο τύπο δεδομένων. Αντίθετα, η συνάρτηση που τυπώνει στην οθόνη το μήνυμα:

University of Ioannina, Dept. of Computer Science & Engineering

δεν επιστρέφει τίποτα (όχι δεν κάνει τίποτα). Στη C όπως και στην Pyhton δεν γίνεται διάκριση ανάμεσα στις διαδικασίες και στις συναρτήσεις, αλλά όπως θα δούμε παρακάτω αυτή δεν είναι και απαραίτητη.

7.2 Ορισμός και κλήση μιας συνάρτησης στην Python

Σε κάθε συνάρτηση δίνουμε ένα όνομα. Στην Python μία συνάρτηση ορίζεται με τη λέξη-κλειδί **def**, η οποία ακολουθείται από το όνομα της συνάρτησης. Το όνομα της συνάρτησης είναι αυτό το οποίο θα χρησιμοποιήσουμε για να απευθυνθούμε αργότερα σε αυτήν, για να την καλέσουμε δηλαδή. Μετά το όνομά της, και μέσα σε δύο παρενθέσεις, ακολουθούν οι παράμετροί της. Δεν μιλήσαμε καθόλου για το τι είναι μία παράμετρος. Θα μιλήσουμε βέβαια αναλυτικά σε λίγο. Τις παραμέτρους ακολουθεί ο κυρίως κώδικας της συνάρτησης. Κατά την προσφιλή μας συνήθεια στην Python, πριν ξεκινήσουν κάποιες ομαδοποιημένες εντολές, όπως ο κώδικας μιας συνάρτησης, βάζουμε μία άνω και κάτω τελεία. Ο κώδικας ακολουθεί στοιχισμένος κάποιες θέσεις προς τα μέσα, όπως ακριβώς κάνουμε όταν θέλουμε να ομαδοποιήσουμε κώδικα στην Python. Μέσα στον κώδικα μπορεί να υπάρχει και η εντολή **return**, που χρησιμοποιείται για να επιστρέψει η συνάρτηση το αποτέλεσμά της.

Ας δούμε τις συναρτήσεις που συζητήσαμε παραπάνω γραμμένες σε Python. Στο Σχήμα 7.1 φαίνεται η υλοποίηση της **univ**, η οποία απλά εμφανίζει ένα μήνυμα στην οθόνη, και η υλοποίησή της **sum10**, η οποία υπολογίζει και επιστρέφει το άθροισμα **1+2+3+...+10**. Σίγουρα καμία από τις δύο δεν είναι και τόσο χρήσιμη, αλλά είναι πολύ καλά παραδείγματα για να καταλάβουμε κάποια απλά πράγματα πριν πάμε σε περισσότερο σύνθετα.

```
1  def univ():
2      print('University of Ioannina')
3      print('Dept. of Computer Science & Engineering')
4
5  def sum10():
6      the_sum=0
7      for i in range(1,11):
8          the_sum+=i
9      return the_sum
10
11 univ()
12 x=sum10()
13 print(x)
14 y=3*sum10()-(4+sum10()/2)
```

Σχήμα 7.1: Υλοποίηση της univ και της sum10.

Έτσι, στη γραμμή 1 δηλώνουμε (με το **def**) ότι θα ορίσουμε τη συνάρτηση **univ**. Στη συνέχεια ακολουθούν δύο παρενθέσεις, οι οποίες δεν έχουν τίποτε μέσα τους. Αυτό σημαίνει ότι η συνάρτηση δεν έχει καθόλου παραμέτρους. Ακολουθεί άνω και κάτω τελεία και ο κώδικας στοιχισμένος προς τα μέσα. Εκεί υπάρχουν δύο **print** (γραμμές 2 και 3) που και θα εμφανίσουν το ζητούμενο στην οθόνη.

Παρακάτω εμφανίζεται η συνάρτηση **sum10**. Όμοια δηλώνουμε το όνομά της και δεν χρησιμοποιούμε παραμέτρους. Ο κώδικας αποτελείται από έναν βρόχο ο οποίος υπολογίζει το **1+2+3+...+10**. Στο τέλος (γραμμή 9) υπάρχει η εντολή **return**, η οποία δηλώνει ότι από όλους τους υπολογισμούς που έγιναν και από όλες τις μεταβλητές που χρησιμοποιήθηκαν, αυτή που ενδιαφέρει αυτόν που κάλεσε τη συνάρτηση είναι η μεταβλητή **the_sum**.

Στο τέλος του Σχήματος 7.1 φαίνεται ο τρόπος με τον οποίο καλούμε τις δύο αυτές συναρτήσεις. Είναι διαφορετικός για τη μία και διαφορετικός για την άλλη, και αυτό οφείλεται στη λειτουργία τους. Στη γραμμή 11 καλείται η **univ**, ενώ αμέσως πιο κάτω η **sum10**. Η διαφορά τους είναι ότι, επειδή η **sum10** επιστρέφει κάποιο αποτέλεσμα, μπορούμε να τη χρησιμοποιήσουμε ως έκφραση και να εκχωρήσουμε το αποτέλεσμά της σε μία μεταβλητή. Ή να αποτελέσει μέρος μιας περισσότερο σύνθετης έκφρασης χρησιμοποιώντας την όπως θα χρησιμοποιούσαμε μία απλή μεταβλητή. Στη γραμμή 14 φαίνεται μία εκχώρηση στη μεταβλητή **y**, η οποία περιέχει δύο κλήσεις της **sum10**.

7.3 Επιστροφή αποτελέσματος

Είδαμε ότι η επιστροφή αποτελέσματος μίας συνάρτησης γίνεται μέσα από τη **return**. Εξετάσαμε όμως μία απλή περίπτωση στην οποία επιστρέφουμε μόνο έναν αριθμό. Η Python μάς δίνει μεγάλη ευελιξία στο τι θα επιστρέψουμε σαν αποτέλεσμα. Ας υποθέσουμε ότι θέλουμε να μας επιστραφεί κάτι πιο πολύπλοκο: μία λίστα, για παράδειγμα, η οποία θα περιέχει τους ακέραιους αριθμούς από το 1 μέχρι το 100 οι οποίοι διαιρούνται με το 2 και το 3 αλλά όχι με το 5. Στο Σχήμα 7.2 φαίνεται ο ορισμός της συνάρτησης. Νομίζω ότι δεν υπάρχει λόγος να περιγράψουμε ακριβώς το πώς λειτουργεί ο βρόχος, είναι πια φαντάζομαι καθαρό μετά τα κεφάλαια 5 και 6. Δείτε, όμως, ότι στο τέλος επιστρέφεται μία λίστα η οποία μπορεί να χρησιμοποιηθεί κανονικά από το πρόγραμμα που κάλεσε τη συνάρτηση. Στη γραμμή 8, το αποτέλεσμα της συνάρτησης εκχωρείται στη μεταβλητή **the_list**, η οποία στη γραμμή 9 τυπώνεται, ενώ στη γραμμή 10 προσπελάζεται και τυπώνεται το πρώτο της στοιχείο.

```
1  def list_2_3_not5():
2      L=[]
3      for x in range(1,101):
4          if x%2==0 and x%3==0 and x%5!=0:
5              L.append(x)
6      return L
7
8  the_list=list_2_3_not5()
9  print (the_list)
10 print (the_list[0])
```

Σχήμα 7.2: Λίστα με τους ακέραιους αριθμούς από το 1 μέχρι το 100 οι οποίοι διαιρούνται με το 2 και το 3 αλλά όχι με το 5.

Αν θέλουμε τώρα να επιστρέψουμε περισσότερα από ένα πράγματα, Αν θέλουμε, ας πούμε, να επιστρέψουμε μία λίστα με τα τέλεια τετράγωνα από το 1 μέχρι και το 100 αλλά και ταυτόχρονα τη θέση στη λίστα του τέλειου τετραγώνου που είναι κοντύτερα στο 50. Ο κώδικας φαίνεται στο Σχήμα 7.3. Στη γραμμή 1 δηλώνεται το όνομα της συνάρτησης, η οποία δεν έχει παραμέτρους. Στη γραμμή 2 ορίζεται η λίστα **L**. Αν διαβάσουμε τη γραμμή 2 λέει ότι η λίστα που δημιουργείται αποτελείται από **x*x** (τέλεια τετράγωνα), όταν το **x** παίρνει τιμές στο διάστημα από 1 έως και 10. Στην αμέσως από κάτω γραμμή ορίζεται μία νέα λίστα, η **T**, η οποία αποτελείται από τις απόλυτες τιμές των διαφορών του **e** με το 50, όταν το **e** παίρνει μία προς μία τις τιμές των στοιχείων της λί-

στας **L**. Αυτό που επιστρέφεται σαν αποτέλεσμα φαίνεται στη γραμμή 4. Εκεί επιστρέφονται δύο πράγματα: Το πρώτο είναι η λίστα **L**, εύκολο. Για το δεύτερο, εφαρμόζουμε στη λίστα **T** τη μέθοδο **index**, η οποία επιστρέφει τη θέση στη λίστα του στοιχείου που υπάρχει μέσα στην παρένθεση. Επειδή η λίστα **T** περιέχει τις αποστάσεις κάθε στοιχείου του **L** από το 50, αν βάλουμε σαν παράμετρο της **index** το **min(T)** που δίνει το ελάχιστο στοιχείο της λίστας **T**, θα έχουμε το ζητούμενο αποτέλεσμα.

```
 1 def pSquare50():
 2     L=[x*x for x in range(1,11)]
 3     T=[abs(e-50) for e in L]
 4     return L,T.index(min(T))
 5
 6 def pSquare50_2():
 7     L=[]
 8     for x in range(1,11):
 9         L.append(x*x)
10     T=[]
11     for e in L:
12         T.append(abs(e-50))
13     m=min(T)
14     pos=T.index(m)
15     return L,pos
16
17 A,a = pSquare50()
```

Σχήμα 7.3: Παράδειγμα με τέλεια τετράγωνα.

Αν σας ήταν δύσκολο, είναι ίσως διότι χρησιμοποιήσαμε λίγο τις ιδιαιτερότητες της Python για να γίνει μικρότερος ο κώδικας. Αν προτιμάτε μια πιο συμβατική υλοποίηση, δείτε τη συνάρτηση **pSquare50_2** που ακολουθεί την **pSquare50** στο ίδιο Σχήμα (7.3). Καλό είναι όμως να συνηθίζετε σιγά σιγά και τον πρώτο τρόπο λύσης.

Πάμε να δούμε τώρα πώς καλούμε μία συνάρτηση που επιστρέφει δύο πράγματα. Για να κρατήσουμε το αποτέλεσμα χρειαζόμαστε δύο μεταβλητές: μία για τη λίστα και μία για τον ακέραιο. Οι δύο αυτές μεταβλητές θα τοποθετηθούν πριν το σύμβολο της εκχώρησης, χωρισμένες μεταξύ τους με κόμματα και με τη σειρά που τις επιστρέφει η συνάρτηση. Δείτε στη γραμμή 17 του Σχήματος 7.3 πώς η **pSquare50** θα δώσει τιμές στις **A** και **a**.

7.4 Παράμετροι

Στην αρχή του κεφαλαίου, όταν αναφέραμε ότι σε μερικά σημεία ο κώδικας επαναλαμβάνεται ο ίδιος, βάλαμε τη λέξη ακριβώς μέσα σε παρένθεση ακολουθούμενη από ένα ερωτηματικό. Ας επιστρέψουμε στο παράδειγμα του αθροίσματος της **sum10**. Αν σε κάποιο σημείο του κώδικα του προγράμματος χρειαζόταν να υπολογίσουμε το άθροισμα **1+2+...+10** και σε κάποιο σημείο άλλο το άθροισμα **1+2+...+20**, ο κώδικας σε αυτά τα δύο σημεία θα ήταν σχεδόν ίδιος αλλά όχι ακριβώς ίδιος. Θα ήταν λοιπόν πολύ καλή ιδέα να φτιάχναμε μία συνάρτηση που να υπολογίζει γενικά το άθροισμα **1+2+...+N** και κάθε φορά που την καλούμε να ορίζουμε για ποιο **N** θέλουμε να υπολογιστεί. Αυτό το **N** είναι που το ονομάζουμε παράμετρο.

Κάποια από τα προβλήματα που έχουμε ήδη συζητήσει πιθανόν να έχει νόημα να γίνουν συναρτήσεις. Για παράδειγμα, αν υλοποιούσαμε το τριώνυμο σαν συνάρτηση, θα βάζαμε σαν παραμέτρους τους συντελεστές **α**,**β** και **γ**, ενώ το πόσες, ποιες και τι ρίζες παίρνουμε θα μπορούσε να επιστραφεί σαν αποτέλεσμα με τη **return**. Σε μία συνάρτηση που υπολογίζει την κυβική ρίζα ενός αριθμού, παράμετρος είναι ο αριθμός του οποίου η κυβική ρίζα θέλουμε να υπολογιστεί, ενώ η κυβική ρίζα θα επιστραφεί με **return**. Όμοια και στη συνάρτηση του λογαρίθμου. Στην ύψωση ενός αριθμού σε δύναμη οι παράμετροι είναι ο αριθμός και η δύναμη στην οποία θα υψωθεί. Μπορείτε να σκεφτείτε πολλά τέτοια παραδείγματα, θα δούμε μερικά ακόμα παρακάτω, όταν θα παρουσιάσουμε μερικά παραδείγματα ολοκληρωμένων συναρτήσεων.

Πριν πάμε όμως σε περισσότερο ενδιαφέροντα παραδείγματα, ας δούμε πώς θα γραφτεί η **sum10** όταν, αντί για το άθροισμα **1+2+...+10**, υπολογίζει το άθροισμα **1+2+...+N**, ή μάλλον γενικότερα, το **a+(a+1)+(a+2)+...+(b-2)+(b-1)+b**. Το άθροισμα, δηλαδή, όλων των αριθμών στο κλειστό διάστημα **[a,b]**. Ο κώδικας φυσικά μοιάζει με αυτόν του Σχήματος 7.1. Μέσα στην παρένθεση όμως πρέπει να ορίσουμε δύο παραμέτρους: την αρχή και το τέλος του διαστήματος, τα **a** και **b** δηλαδή. Στον κώδικα οι επαναλήψεις του βρόχου θα ξεκινούν από το **a** και θα φτάνουν στο **b**, άρα το **range** θα έχει σαν πρώτο όρισμα μέσα στην παρένθεση το **a** και σαν δεύτερο το **b+1** (μην ξεχνάτε, πηγαίνει έως **b+1**, δηλαδή έως και **b**). Η συνάρτηση ολοκληρωμένη φαίνεται στο Σχήμα 7.4.

```
def sum10(a,b):
    the_sum=0
    for i in range(a,b+1):
        the_sum+=i
    return the_sum
```

Σχήμα 7.4: Παράδειγμα περάσματος παραμέτρου.

7.5 Παραδείγματα συναρτήσεων

Στο κεφάλαιο αυτό θα δούμε παραδείγματα από πέντε συναρτήσεις, τρεις από τα μαθηματικά και δύο με λίστες. Θα δούμε και τρεις ταινίες, καθεμία από τις οποίες θα παρουσιάσει και μία συνάρτηση. Από τα μαθηματικά θα δούμε τον υπολογισμό της απόλυτης τιμής, του παραγοντικού και της ύψωσης σε δύναμη. Παρότι αυτές τις συναρτήσεις μπορεί κανείς να τις βρει έτοιμες στην Python ή ενσωματωμένες ή σε βιβλιοθήκες, αποτελούν πολύ καλά παραδείγματα για να κατανοήσετε τη λογική των συναρτήσεων. Θα τις υλοποιήσουμε λοιπόν για εκπαιδευτικούς λόγους. Οι δύο συναρτήσεις που θα δούμε και χρησιμοποιούν λίστες είναι στην ίδια λογική. Παρουσιάζουν και αυτές εκπαιδευτικό ενδιαφέρον, ενώ δεν είμαι καθόλου σίγουρος ότι θα τις χρειαστείτε ως έχουν για να τις ενσωματώσετε σε κάποιον κώδικά σας. Τέλος, θα δούμε σε ταινίες την εναλλαγή των τιμών δύο μεταβλητών, τη μέτρηση φωνηέντων μέσα σε μία λίστα συμβολοσειρών και από ένα σύνολο σημείων του επιπέδου την εύρεση του σημείου που βρίσκεται κοντύτερα στην αρχή των αξόνων.

7.5.1 Απόλυτη τιμή

Η απόλυτη τιμή ενός αριθμού είναι ο αριθμός χωρίς το πρόσημο. Αν θέλουμε να είμαστε περισσότερο σωστοί στον ορισμό μας, η απόλυτη τιμή ενός αριθμού είναι ο ίδιος αριθμός αν αυτός δεν είναι αρνητικός ή ο αντίθετός του αν αυτός είναι αρνητικός. Τον ορισμό αυτόν θα τον χρησιμοποιήσουμε και στον κώδικα που θα φτιάξουμε. Θέλουμε, λοιπόν, μια συνάρτηση με μία είσοδο, τον αριθμό του οποίου την απόλυτη τιμή θέλουμε να υπολογίσουμε, και μία έξοδο, την απόλυτη τιμή που θα υπολογίσουμε. Ο κώδικας μέσα στη συνάρτηση θα αποτελείται κυρίως από μία δομή απόφασης (**if-else**), η οποία θα εξετάζει αν ο αριθμός που δόθηκε σαν είσοδος στη συνάρτηση (α) είναι αρνητικός, οπότε και θα επιστρέφει τον αντίθετό του, ή αν (β) είναι θετικός ή μηδέν, οπότε θα επιστρέφει τον ίδιο τον αριθμό. Ο κώδικας είναι απλός. Νομίζω δεν θα σας δυσκολέψει. Μπορείτε να τον δείτε αναλυτικά στο Σχήμα 7.5.

```
1 def absolute(x):
2     if x<0:
3         return -x
4     else:
5         return x
```

Σχήμα 7.5: Απόλυτη τιμή.

7.5.2 Παραγοντικό

Τον υπολογισμό του παραγοντικού τον έχουμε συζητήσει αναλυτικά σε προηγούμενο κεφάλαιο. Εδώ θα δούμε το πώς θα φτιάξουμε μία συνάρτηση που να το υπολογίζει. Είναι εύκολο να σκεφτούμε ότι χρειάζεται μόνο μία παράμετρος μέσω της οποίας θα εισάγεται στη συνάρτηση ο αριθμός του οποίου το παραγοντικό θέλουμε να υπολογίσουμε. Χρειαζόμαστε, επίσης, και μία μεταβλητή που θα επιστρέφει το αποτέλεσμα.

Όμως, μία συνάρτηση, εκτός από το να υπολογίζει σωστά το αποτέλεσμα όταν της δοθούν σωστές παράμετροι, πρέπει να αντιδρά σωστά και να προστατεύει κάποιον τρίτο από το να δώσει εσφαλμένες τιμές στις παραμέτρους. Στο παραγοντικό, για παράδειγμα, η συνάρτηση πρέπει να ελέγχει αν ο αριθμός του οποίου το παραγοντικό της ζητήθηκε να υπολογίσει είναι θετικός ή μηδέν. Αν είναι αρνητικός, τότε το παραγοντικό δεν μπορεί να υπολογιστεί, οπότε και πρέπει να πληροφορήσει αυτόν που την κάλεσε ότι κάτι δεν πάει καλά.

Δείτε τον κώδικα στο Σχήμα 7.6. Με το που ξεκινάει η συνάρτηση ελέγχει αν ο **x** είναι αρνητικός. Αν πράγματι είναι αρνητικός, επιστρέφει μια συμβολοσειρά με την τιμή **error: negative number**. Θα μπορούσε να είναι και κάτι πιο περιγραφικό, αν θα θέλατε. Αν όλα πήγαν καλά, δηλαδή στις παραμέτρους δόθηκε θετικός αριθμός ή μηδέν, υπολογίζεται το παραγοντικό και επιστρέφεται το αποτέλεσμα. Δείτε τον κώδικα στο Σχήμα 7.6 και, αν σας δυσκολεύει κάτι, επιστρέψτε στο κεφάλαιο με τα διαγράμματα ροής (Κεφάλαιο 3), όπου και είχαμε αναλύσει το συγκεκριμένο πρόβλημα.

```
1 def factorial(x):
2     if x<0:
3         return 'error: negative number'
4     else:
5         fact=1
6         for i in range(1,x+1):
7             fact=fact*i
8     return fact
```

Σχήμα 7.6: Παραγοντικό.

7.5.3 Ύψωση σε δύναμη

Η ύψωση σε δύναμη είναι ένα κλασικό και ενδιαφέρον θέμα. Πριν ξεκινήσουμε τον σχεδιασμό και την υλοποίηση της συνάρτησης, καλό είναι να ρίξουμε μια ματιά στη θεωρία. Όλοι γνωρίζουμε ότι η ύψωση του **a** στη **x** σημαίνει **a*a*...*a**, **x** φορές. Αυτό είναι η απλή περίπτωση όμως, στην οποία το **x** είναι ένας θετικός ακέραιος αριθμός. Ας θυμηθούμε τι συμβαίνει και στις υπόλοιπες περιπτώσεις, αλλά ας περιορίσουμε το πρόβλημά, για απλότητα, μόνο σε ακέραιες τιμές του **x**.

Σύμφωνα με τους παραπάνω τύπους, όταν το **x** είναι θετικός αριθμός, τότε πολλαπλασιάζουμε το **a** με τον εαυτό του **x** φορές. Όταν το **x** είναι αρνητικός αριθμός, τότε το **a** μεταφέρεται στον παρονομαστή και το πρόσημο του **x** γίνεται θετικό. Φυσικά, αν το **a** είναι μηδέν, τότε το **a** δεν μπορεί να μεταφερθεί στον παρονομαστή, αφού διαίρεση με το μηδέν δεν ορίζεται. Έτσι, και η ύψωση σε δύναμη σε αυτήν την περίπτωση δεν ορίζεται. Τέλος, αν ο εκθέτης είναι μηδέν το αποτέλεσμα της ύψωσης σε δύναμη είναι πάντοτε 1. Πάμε τώρα να υλοποιήσουμε τη συνάρτηση.

Η συνάρτησή μας θα έχει μία δομή απόφασης με τέσσερα σκέλη, ένα για καθεμία από τις παραπάνω περιπτώσεις. Τα **a** και **x** είναι οι παράμετροι που δίνονται σαν είσοδο στη συνάρτηση. Στο Σχήμα 7.7 φαίνεται ο κώδικας που την υλοποιεί. Μόνο που η δομή απόφασης έχει πέντε και όχι τέσσερις περιπτώσεις. Την πρώτη περίπτωση από τις περιπτώσεις δεν την συζητήσαμε. Εκεί κάνουμε έναν έλεγχο για να διαπιστώσουμε αν το **x** είναι πράγματι ακέραιος. Μπορεί η ύψωση σε μη ακέραια δύναμη να έχει νόημα, εμείς όμως στον κώδικα είπαμε ότι δεν θα λάβουμε υπόψη μας την περίπτωση αυτήν. Είναι καλό, αν όχι απαραίτητο, να κάνουμε έναν έλεγχο στην αρχή της συνάρτησης αν ο **x** είναι ακέραιος και να επιστρέφουμε ένα κατάλληλο μήνυμα σε αυτόν που μας

κάλεσε όταν ο **x** δεν είναι ακέραιος. Για να το πετύχουμε αυτό, ελέγχουμε αν το ακέραιο μέρος του **x**, το **int(x)** δηλαδή, είναι ίσο με το **x**. Υπάρχουν και άλλοι τρόποι να το ελέγξουμε αυτό.

Στη συνέχεια, μπαίνουμε στο κυρίως πρόγραμμα. Μέσα σε αυτό γίνεται ο έλεγχος (α) αν το **x** είναι μηδέν, οπότε και επιστρέφεται το 1 σαν αποτέλεσμα, (β) αν το **x** είναι θετικός, οπότε και με έναν βρόχο πολλαπλασιάζουμε το **a** με τον εαυτό του **x** φορές, (γ) αν το **a** είναι μηδέν και ταυτόχρονα το **x** αρνητικό, όπου και η πράξη δεν ορίζεται και επιστρέφουμε μήνυμα λάθους και, τέλος, (δ) η εναπομείνουσα περίπτωση στην οποία το **x** είναι αρνητικό αλλά το **a** δεν είναι μηδέν, οπότε η πράξη ορίζεται και επιστρέφουμε το αποτέλεσμα του πολλαπλασιασμού του **1/a** με τον εαυτό του **-x** φορές.

```
1  def power(a,x):
2      if int(x)!=x:
3          return 'error: not supported'
4      elif x==0:
5          return 1
6      elif x>0:
7          p=1
8          for i in range(x):
9              p*=a
10         return p
11     elif a==0 and x<0:
12         return 'error: division by zero'
13     else:
14         p=1
15         for i in range(-x):
16             p/=float(a)
17         return p
```

Σχήμα 7.7: Ύψωση σε δύναμη.

7.5.4 Λίστα θετικών και αρνητικών αριθμών

Στο παράδειγμα αυτό θα φτιάξουμε μία συνάρτηση η οποία θα δέχεται σαν είσοδο μία λίστα πραγματικών αριθμών και θα τη χωρίζει σε μία λίστα θετικών αριθμών και σε μία λίστα αρνητικών αριθμών, ενώ θα μετράει πόσες φορές εμφανίζεται στη δοθείσα λίστα το μηδέν.

Στην είσοδο έχουμε μία παράμετρο, τη λίστα, και στην έξοδο τρεις μεταβλητές, δύο λίστες και έναν ακέραιο αριθμό. Ο κώδικας, ο οποίος φαίνεται στο Σχήμα 7.8, αποτελείται από έναν βρόχο που προσπελάζει ένα ένα τα στοιχεία της αρχικής λίστας και για καθένα από αυτά εισέρχεται σε μία δομή απόφασης με την οποία εισάγει το στοιχείο στη λίστα των θετικών αριθμών, στη λίστα των αρνητικών αριθμών ή αυξάνει κατά ένα τον μετρητή των μηδενικών, ανάλογα με το αν το στοιχείο είναι θετικός, αρνητικός αριθμός ή μηδέν. Η εντολή **return** ολοκληρώνει τη συνάρτηση επιστρέφοντας τα τρία αυτά αποτελέσματα.

```
1  def pos_and_neg(L):
2      pos=[]
3      neg=[]
4      zero=0
5      for x in L:
6          if x>0:
7              pos.append(x)
8          if x<0:
9              neg.append(x)
10         if x==0:
11             zero+=1
12     return pos,neg,zero
```

Σχήμα 7.8: Λίστα θετικών και αρνητικών αριθμών.

7.5.5 Λίστα από λίστες

Στη συνάρτηση αυτή θα φτιάξουμε μία λίστα παίρνοντας δεδομένα από τρεις λίστες. Οι τρεις αυτές λίστες θα έχουν τα ονόματα, τα επίθετα και τις ηλικίες κάποιων ατόμων. Θα υποθέσουμε ότι το ι-οστό στοιχείο της πρώτης λίστας αντιστοιχεί με το ι-οστό στοιχείο της δεύτερης και το ι-οστό στοιχείο της τρίτης λίστας. Σκοπός μας είναι να φτιάξουμε μια νέα λίστα, κάθε στοιχείο της οποίας θα είναι μία λίστα η οποία θα περιέχει το επίθετο, το όνομα και την ηλικία ενός ατόμου.

Ο κώδικας φαίνεται στο Σχήμα 7.9. Στην αρχή του κώδικα γίνεται ένας έλεγχος ότι και οι τρεις λίστες έχουν τον ίδιο αριθμό στοιχείων. Χρησιμοποιούμε τη **len** για τον σκοπό αυτόν η οποία μας δίνει το πλήθος των στοιχείων μιας λίστας. Παρατηρήστε ότι η Pyton μάς επιτρέπει να τσεκάρουμε την

ισότητα τριών πραγμάτων χωρίς να καταφύγουμε αναγκαστικά σε συγκρίσεις ανά δύο.

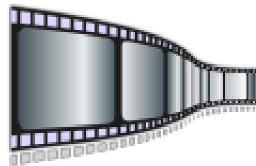
Αν τώρα οι λίστες έχουν ίδιο αριθμό στοιχείων, τότε ένας βρόχος, χρησιμοποιώντας την εντολή **zip**, προσπελάζει παράλληλα τα στοιχεία των τριών λιστών. Στην πρώτη επανάληψη του βρόχου το **x** παίρνει το πρώτο στοιχείο από την πρώτη λίστα, το **y** το πρώτο στοιχείο από τη δεύτερη λίστα και το **z** το πρώτο στοιχείο από την τρίτη λίστα. Στη δεύτερη επανάληψη το **x** παίρνει το δεύτερο στοιχείο από την πρώτη λίστα, το **y** το δεύτερο στοιχείο από τη δεύτερη λίστα κ.ο.κ. Στο τέλος της κάθε επανάληψης μια νέα λίστα δημιουργείται με το ι-οστο όνομα, επίθετο και την ι-οστή ηλικία. Η νέα αυτή λίστα προστίθεται στο τέλος της λίστας που κατασκευάζουμε και που τελικά θα επιστραφεί σαν αποτέλεσμα.

```
1 def lists_to_list(name,lastName,age):
2     if not(len(name)==len(lastName)==len(age)):
3         return 'error'
4     else:
5         Person=[]
6         for x,y,z in zip(name,lastName,age):
7             Person.append([x,y,z])
8         return Person
```

Σχήμα 7.9: Λίστα από λίστες.

7.5.6 Άλλες συναρτήσεις

Ας δούμε μερικές συναρτήσεις σε μικρές ταινίες. Κάντε κλικ παρακάτω για να δείτε την Ταινία 7.1, που εξηγεί πώς μία συνάρτηση αντιμεταθέτει τις τιμές δύο μεταβλητών. Ο κώδικάς της φαίνεται στο Σχήμα 7.10.



Ταινία 7.1: Εναλλαγή των τιμών δύο μεταβλητών.

`http://repfiles.kallipos.gr/file/22388`

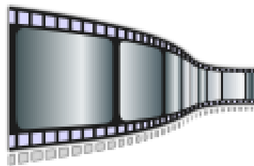
```python
def dummy_swap(x,y):
    t=x
    x=y
    y=t
    return x,y

def swap(x,y):
    return y,x
```

Σχήμα 7.10: Εναλλαγή των τιμών δύο μεταβλητών.

Στο Σχήμα 7.11 φαίνεται ο κώδικας μιας συνάρτησης η οποία επιλέγει από μία λίστα τη συμβολοσειρά που έχει περισσότερα φωνήεντα. Κάνετε κλικ και στην Ταινία 7.2 που ακολουθεί και θα δείτε πώς λειτουργεί.

```python
def vowel_count(s):
    vowels={'a','e','i','o','u'}
    max_vow=0
    for word in s:
        count=0
        for letter in word:
            if letter in vowels:
                count+=1
        if count>max_vow:
            max_vow=count
    return max_vow
```

Σχήμα 7.11: Φωνήεντα σε λίστα από συμβολοσειρές.



Ταινία 7.2: Φωνήεντα σε λίστα από συμβολοσειρές.

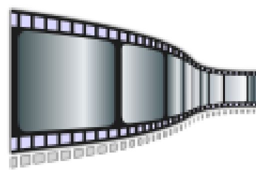
http://repfiles.kallipos.gr/file/22391

Τέλος, στο Σχήμα 7.12 φαίνεται ο κώδικας μίας συνάρτησης που βρίσκει μέσα από ένα σύνολο σημείων στο επίπεδο το σημείο που είναι εγγύτερα στην

αρχή των αξόνων, ενώ στην Ταινία 7.3 μπορείτε να δείτε πώς σκεφτήκαμε για να την υλοποιήσουμε. Στο παράδειγμα αυτό βλέπουμε μία συνάρτηση η οποία καλείται μέσα από μία άλλη. Ορίζουμε ως σημείο μία πλειάδα μεγέθους 2 και χρησιμοποιούμε σύνολα αυτών των σημείων.

```
1  import math
2
3  def distance(p):
4      return math.sqrt(p[0]*p[0]+p[1]*p[1])
5
6  def min_from_origin(S):
7      if len(S)==0:
8          return 'error: no points in set'
9      min_p=S.pop()
10     min_dist=distance(min_p)
11     for p in S:
12         if distance(p)<min_dist:
13             min_dist=distance(p)
14             min_p=p
15     return min_p
16
17 S={(3,4),(2,3),(1,1),(5,5),(1,2)}
18 print (min_from_origin(S))
```

Σχήμα 7.12: Απόσταση από την αρχή των αξόνων.



Ταινία 7.3: Απόσταση από την αρχή των αξόνων.

http://repfiles.kallipos.gr/file/22392

7.6 Πέρασμα παραμέτρων με τιμή και με αναφορά

Ο τρόπος περάσματος των παραμέτρων μπορεί να διαφέρει από γλώσσα σε γλώσσα. Για την ακρίβεια, το πιο συνηθισμένο είναι μια γλώσσα να υποστη-

ρίζει **πέρασμα παραμέτρων με τιμή (call by value)** και **πέρασμα παραμέτρων με αναφορά (call by reference)**. Ο τρόπος περάσματος που παρουσιάσαμε παραπάνω στην Python ανήκει στην κατηγορία **πέρασμα με τιμή**. Ας δούμε όμως τι είναι αυτοί οι δύο τρόποι περάσματος.

Όταν έχουμε πέρασμα παραμέτρων με τιμή, κατά τη δημιουργία του περιβάλλοντος για την εκτέλεση της κληθείσας συνάρτησης, η τιμή της παραμέτρου αντιγράφεται από τη στοίβα της καλούσας συνάρτησης στη στοίβα της κληθείσας. Δημιουργείται δηλαδή ένα τοπικό αντίγραφο της μεταβλητής της καλούσας στον χώρο της κληθείσας. Στη συνέχεια, ο έλεγχος μεταβαίνει στην κληθείσα, όπου η τοπική αυτή μεταβλητή μπορεί να διαβαστεί αλλά και να τροποποιηθεί. Μετά την ολοκλήρωση της εκτέλεσης της κληθείσας συνάρτησης, η στοίβα της καταστρέφεται (παύει να είναι έγκυρη για την ακρίβεια), οπότε και το τοπικό αντίγραφο της μεταβλητής χάνεται με τον τερματισμό της μαζί με ό,τι αλλαγές έχει πιθανόν υποστεί. Άρα, ό,τι συνέβη στη μεταβλητή που αντιγράφηκε στο χώρο της κληθείσας δεν επηρεάζει την αρχική μεταβλητή στον χώρο της καλούσας.

Μπορεί βέβαια η προηγούμενη περιγραφή να είναι (λίγο πολύ ...) ακριβής, δεν είναι όμως σίγουρα και απόλυτα κατανοητή. Θα δούμε καλύτερα τι σημαίνει πέρασμα παραμέτρου με τιμή χρησιμοποιώντας το παράδειγμα του Σχήματος 7.13.

```
1 def inc(x):
2     print(x)
3     x=x+1
4     print(x)
5
6 a=3
7 print(a)
8 inc(a)
9 print(a)
```

Σχήμα 7.13: Πέρασμα παραμέτρου με τιμή.

Ο κώδικας του Σχήματος 7.13 υποτίθεται ότι φτιάχνει μια συνάρτηση η οποία αυξάνει κατά 1 την τιμή της παραμέτρου **x**. Για να βεβαιωθούμε, τυπώνουμε σε διάφορα μέρη την τιμή της **x**, αλλά από την εκτέλεση τα συμπεράσματα είναι διαφορετικά από ό,τι αναμέναμε. Εκτελώντας τον παραπάνω κώδικα παίρνουμε:

```
3
3
4
3
```

Για να δούμε τι έγινε. Το **a** αρχικά γίνεται 3. Το τυπώνουμε και παίρνουμε στην οθόνη το 3 από την **print** που ακολουθεί την εκχώρηση. Μετά καλούμε τη συνάρτηση. Το **x** παίρνει ό,τι τιμή έχει το **a**, άρα γίνεται 3, κάτι που επιβεβαιώνεται και από το πρώτο **print** στην πρώτη γραμμή της συνάρτησης. Μετά αυξάνουμε το **x** κατά 1 και το ξανατυπώνουμε παίρνοντας στην οθόνη το 4, όπως αναμενόταν. Τώρα τερματίζεται η συνάρτηση. Η τιμή του **x**, δηλαδή το 4, δεν επιστρέφεται πίσω στο **a**, άρα η τιμή του **a** παραμένει 3, κάτι που βλέπουμε και ως αποτέλεσμα της τελευταίας **print**. Η C, η Pascal και η Python υποστηρίζουν πέρασμα παραμέτρων με τιμή.

Η C και η Pascal υποστηρίζουν και το πέρασμα παραμέτρων με αναφορά. Η Pascal το υποστηρίζει με άμεσο τρόπο, ενώ η C μέσα από έναν μηχανισμό δεικτών. Στο πέρασμα παραμέτρου με αναφορά η τιμή της τοπικής μεταβλητής (του **x** στο παράδειγμά μας) επιστέφεται πίσω (στο **a** στο παράδειγμά μας). Έτσι, στην οθόνη θα εμφανίζονταν:

```
3
3
4
4
```

Ας δούμε με λίγο περισσότερο λεπτομέρεια το γιατί. Κατά τη δημιουργία του περιβάλλοντος για την εκτέλεση της κληθείσας συνάρτησης, η τιμή της παραμέτρου δεν αντιγράφεται από τη στοίβα της καλούσας στη στοίβα της κληθείσας, αλλά ένας δείκτης προς τη μεταβλητή στην καλούσα συνάρτηση δημιουργείται και αποθηκεύεται στον χώρο της κληθείσας, στη στοίβα της δηλαδή. Στη συνέχεια ο έλεγχος μεταβαίνει στην κληθείσα, όπου η μεταβλητή στον χώρο της καλούσας μπορεί να διαβαστεί αλλά και να τροποποιηθεί, μέσα από τον δείκτη αυτόν. Μετά την ολοκλήρωση της εκτέλεσης της κληθείσας συνάρτησης, η στοίβα της καταστρέφεται (παύει να είναι έγκυρη), οπότε και ο δείκτης στη μεταβλητή χάνεται με τον τερματισμό της συνάρτησης. Ό,τι αλλαγές όμως έχουν πιθανόν γίνει στην τιμή της μεταβλητής δεν χάνονται, αφού αυτές μέσα από τον δείκτη έχουν ουσιαστικά γίνει στη μεταβλητή στον χώρο της καλούσας. Άρα, πρακτικά, ό,τι αλλαγή έγινε στην τιμής της μεταβλητής διατηρήθηκε και μετά το πέρας της συνάρτησης και επηρέασε την αρχική μεταβλητή.

Ο μηχανισμός αυτός με τον δείκτη κρύβεται από τις περισσότερες γλώσσες, για παράδειγμα από την Pascal και την Java ή την C++. Στη C, όχι μόνο δεν κρύβεται, αλλά είναι υποχρεωμένος ο προγραμματιστής να φροντίσει για τις λεπτομέρειες του μηχανισμού αυτού και να δημιουργήσει και να διαχειριστεί αυτός τον δείκτη. Αυτό δεν είναι ό,τι πιο εύχρηστο, και πολλούς προγραμματιστές στην αρχή τούς δυσκολεύει ή τούς ξενίζει. Όμως, φροντίζοντας εσύ για όλες τις λεπτομέρειες, έχεις πραγματικά συναίσθηση του τι συμβαίνει μέσα στο πρόγραμμά σου, αλλά και περισσή ευελιξία. Τα πράγματα στην Pascal είναι πολύ πιο εύκολα, αφού εκεί ο προγραμματιστής πρέπει απλά να δηλώσει ποιο τρόπο περάσματος προτιμά. Στην επόμενη ενότητα, στην οποία θα μιλήσουμε για συναρτήσεις και διαδικασίες, θα δούμε ένα παράδειγμα περάσματος παραμέτρων με αναφορά σε γλώσσα Pascal.

7.7 Συναρτήσεις και διαδικασίες

Μία **διαδικασία (procedure)** διαφέρει από μία **συνάρτηση (function)** στο ότι, ενώ η συνάρτηση υπολογίζει και επιστρέφει μια τιμή ή ένα σύνολο από τιμές, η διαδικασία δεν επιστρέφει τίποτε. Αυτό, βέβαια, δεν σημαίνει ότι η διαδικασία δεν υπολογίζει ή δεν κάνει τίποτα ή ότι δεν είναι χρήσιμη σε αυτόν που την καλεί. Η διαδικασία μπορεί να κάνει κάτι στο οποίο δεν χρειάζεται να επιστρέψει ένα αποτέλεσμα (δείτε την **univ** παραπάνω), ή μπορεί να επιστρέψει κάτι που υπολόγισε μέσα από μία παράμετρο που περνιέται με αναφορά. Θα λέγαμε, λοιπόν, ότι μια διαδικασία κάνει ακριβώς ό,τι και μία συνάρτηση, αλλά διαφέρει ο τρόπος με τον οποίο αντλούμε από αυτήν τα αποτελέσματα.

```
1  procedure swap(var c1,c2:integer);
2  var c:integer;
3  begin
4    c:=c1;
5    c1:=c2;
6    c2:=c;
7  end;
```

Σχήμα 7.14: Διαδικασία εναλλαγής τιμών σε Pascal.

Για να είμαστε περισσότερο ακριβείς σε κάτι που αναφέραμε παραπάνω, η Python υποστηρίζει τις διαδικασίες, αρκεί να μην πρέπει να επιστρέψουν κάτι στο κυρίως πρόγραμμα. Η **univ**, την οποία υλοποιήσαμε στο Σχήμα 7.1, αποτελεί μια διαδικασία, αφού δεν περιέχει κάποια εντολή **return** για να επιστρα-

φεί το αποτέλεσμα. Η κλήση της γίνεται χωρίς να βρίσκεται στο δεξί μέλος μιας εκχώρησης, σε αντίθεση με τη **sum10**, η οποία μπορεί να αποτιμηθεί και η κλήση της να αντικατασταθεί με το αποτέλεσμά της μέσα σε μία έκφραση.

Θα προτιμήσουμε να δείξουμε ένα παράδειγμα διαδικασίας χρησιμοποιώντας πάλι Pascal. Στο παράδειγμα αυτό θα δούμε μία διαδικασία στην οποία το πέρασμα παραμέτρων γίνεται με αναφορά. Στο Σχήμα 7.14 φαίνεται ο κώδικας σε Pascal της γνωστής μας από τα παραπάνω διαδικασίας που εναλλάσσει τις τιμές δύο μεταβλητών. Εκεί δηλώνεται η **swap** σαν διαδικασία και στη συνέχεια ακολουθούν δύο παράμετροι, οι **a** και **b**, που περνάνε και οι δύο με αναφορά. Οι παράμετροι αυτές παίρνουν τιμές από τις **x** και **y** του κυρίως προγράμματος. Οι τιμές των παραμέτρων αυτών εναλλάσσονται μέσα στον κώδικα της **swap**. Με το πέρας της συνάρτησης, η εναλλαγή στις τιμές των δύο μεταβλητών δεν χάνεται, αφού το πέρασμα έχει γίνει με αναφορά και οι νέες τιμές των **a** και **b** περνούν στις μεταβλητές **x** και **y** του κυρίως προγράμματος αντίστοιχα.

7.8 Ευελιξία περάσματος παραμέτρων στην Python

Η Python, όπως πολλές σύγχρονες γλώσσες, παρέχει μεγάλη ευελιξία στον τρόπο που μπορούμε να περνάμε παραμέτρους στις συναρτήσεις. Με την ευελιξία αυτή πετυχαίνει, όχι μόνο να διευκολύνει σε πολλές περιπτώσεις τον προγραμματιστή και να τον καθοδηγεί στο να γράφει καλύτερο κώδικα, αλλά και να κάνει τον κώδικά της περισσότερο ευανάγνωστο.

```
1  def sentence(the_subject,the_verb,the_object):
2      return the_subject+' '+the_verb+' '+the_object
3
4  print(sentence('Peter','reads','a book'))
5
6  print(sentence(the_subject='Peter',the_verb='reads',
7                                  the_object='a book'))
8  print(sentence(the_verb='reads',the_object='a book',
                                   the_subject='Peter'))
```

Σχήμα 7.15: Χρήση ονομάτων για τις παραμέτρους.

Θα μιλήσουμε γι' αυτό, όπως κάνουμε συνήθως, οδηγούμενοι από παραδείγματα. Έστω ότι έχουμε τη συνάρτηση **sentence** που φαίνεται στο Σχήμα

7.15. Η συνάρτηση αυτή παίρνει ένα υποκείμενο, ένα ρήμα και ένα αντικείμενο και φτιάχνει μία πρόταση. Ο προφανής τρόπος να την καλέσουμε είναι να τοποθετήσουμε μέσα στην παρένθεση τρεις συμβολοσειρές. Αν δώσουμε τα **Peter**, **reads**, a **book** με την σειρά αυτή, το **Peter** θα αντιστοιχιστεί με το υποκείμενο, το **reads** με το ρήμα και το a **book** με το αντικείμενο, ακολουθώντας τη σειρά με την οποία έχουν δοθεί οι παράμετροι στον ορισμό και στην κλήση της συνάρτησης. Δράττομαι της ευκαιρίας να πω ότι οι πρώτοι καλούνται τυπικές παράμετροι και οι δεύτεροι πραγματικές παράμετροι.

Κάνοντας χρήση των δυνατοτήτων της Python και εκμεταλλευόμενοι την βασική της φιλοσοφική προσέγγιση για καλύτερο και περισσότερο ευανάγνωστο κώδικα, μπορούμε, αντί να καλέσουμε απλά τη συνάρτηση με τις τρεις παραμέτρους της, να επαναλάβουμε την ονομασία τους. Έτσι η κλήση (βλέπε παρακάτω στο Σχήμα 7.15) μπορεί να γίνει και ως:

sentence(the_subject='Peter',the_verb='reads',the_object='a book')

Ο κώδικας είναι σαφώς πιο ευανάγνωστος και δεν μας αναγκάζει να ανατρέχουμε στον ορισμό της συνάρτησης στην περίπτωση που ένα μήνα ή τρία χρόνια μετά χρειαστεί να ξαναδιαβάσουμε τον κώδικα. Αλλά μας δίνει και μία ακόμα δυνατότητα: δεν είναι πια απαραίτητο οι πραγματικές παράμετροι να ακολουθούν τη σειρά των πραγματικών, αφού είναι απόλυτα σαφές ποια τιμή αντιστοιχεί σε ποια παράμετρο και δεν καθορίζεται πια αυτό από τη σειρά με την οποία αυτές δηλώθηκαν και κλήθηκαν. Και στις τρεις κλήσεις της **sentence** στο Σχήμα 7.15 θα επιστρέψουν το ίδιο πράγμα και στην οθόνη θα εμφανιστεί για καθεμία το μήνυμα:

Peter reads a book

Η ευελιξία που έχουμε δεν σταματάει εδώ. Η Python μάς δίνει τη δυνατότητα να αντιστοιχίσουμε κάποια προκαθορισμένη αρχική τιμή σε μία παράμετρο, η οποία δεν θα ληφθεί καθόλου υπόψη αν κατά την κλήση της συνάρτησης δοθεί κάποια τιμή στην παράμετρο (όπως έχει συμβεί σε όλα τα παραδείγματα μέχρι τώρα). Αν όμως κατά την κλήση της συνάρτησης η παράμετρος αυτή παραλειφθεί, τότε αυτή θα αρχικοποιηθεί στην προκαθορισμένη αρχική τιμής. Παρατηρήστε ότι με τον τρόπο αυτόν μπορούμε να φτιάξουμε συναρτήσεις οι οποίες δεν είναι απαραίτητο να έχουν σταθερό αριθμό από ορίσματα.

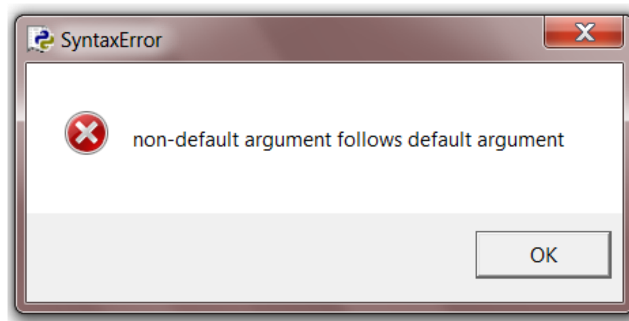
Στο παράδειγμά στο Σχήμα 7.16, στην πρώτη κλήση της **sentence_2**, το αντικείμενο που θα τυπωθεί θα είναι το **a book**, παρότι η προκαθορισμένη αρχική τιμή ήταν το **the newspaper**, αφού κατά την κλήση της τής δόθηκε η τιμή a **book**. Αντίθετα, η δεύτερη κλήση θα χρησιμοποιήσει σαν αντικείμενο το **the**

newspaper, αφού κατά την κλήση της συνάρτησης η παράμετρος **the_object** παραλείφθηκε και δεν της δόθηκε έτσι τιμή. Το ίδιο θα συμβεί και στην τελευταία κλήση. Εκεί ακολουθείται η σειρά με την οποία εμφανίζονται οι τυπικοί και οι πραγματικοί παράμετροι.

```
1  def sentence_2(the_subject,the_verb,the_object='the newspaper'):
2      return the_subject+' '+the_verb+' '+the_object
3
4  print(sentence_2(the_verb='reads',the_object='a book',
5                                   the_subject='Peter'))
6  print(sentence_2(the_verb='reads',the_subject='Peter'))
7
8  print(sentence_2('Peter','reads'))
```

Σχήμα 7.16: Προκαθορισμένη αρχική τιμή και μεταβλητός αριθμός παραμέτρων.

```
1  def sentence_3(the_subject,the_verb='reads',the_object):
2      return the_subject+' '+the_verb+' '+the_object
```



Σχήμα 7.17: Προκαθορισμένη αρχική τιμή και θέση παραμέτρων.

Το γεγονός ότι η παράμετρος στην οποία δόθηκε η προκαθορισμένη αρχική τιμή ήταν η τελευταία διευκολύνει στο να αντιστοιχιστούν οι παράμετροι. Στο Σχήμα 7.17 βλέπουμε μία περίπτωση που δεν το ακολουθούμε αυτό και η προκαθορισμένη αρχική τιμή δίνεται στην προτελευταία παράμετρο. Η Python μάς διαμαρτύρεται θυμίζοντάς μας ότι ακόμα και η ευελιξία πρέπει να έχει τα όριά της.

Περισσότερο υλικό σχετικό με αυτό τις συναρτήσεις της Python η συναρτήσεις γενικότερα μπορείτε να διαβάσετε στο κεφάλαιο 3 του βιβλίου [1], στο

κεφάλαιο 4 του βιβλίου [2], στο κεφάλαιο 6 του βιβλίου [3], στα κεφάλαια 7,8 του βιβλίου [4] και στο κεφάλαιο 5 του βιβλίου [5].

Βιβλιογραφία

1. Allen B. Downey (2012). **Think Python**. Publisher: O'Reilly Media.
2. John Guttag (2015). **Υπολογισμοί και Προγραμματισμός Με Την Python**. Μετάφραση: Παναγιώτης Καναβός, Επιμέλεια: Γεώργιος Μανής, Εκδόσεις Κλειδάριθμος.
3. Brian Heinold (2012). **Introduction to Programming Using Python**. Publisher: Mount St. Mary's University, Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.
4. Cody Jackson (2011). **Learning to Program Using Python**. Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.
5. Eric Roberts. (2004). **Η Τέχνη και Επιστήμη της C**. Μετάφραση: Γιώργος Στεφανίδης, Παναγιώτης Σταυρόπουλος, Αλέξανδρος Χατζηγεωργίου, Εκδόσεις Κλειδάριθμός

Κεφάλαιο 8: Προγραμματίζοντας αλγορίθμους έξυπνα και δημιουργικά

Η συνεχής βελτίωση του **υλικού (hardware)** τις τελευταίες δεκαετίες έχει σαν αποτέλεσμα την ύπαρξη πολύ ισχυρών επεξεργαστών. Αν και σε λίγα χρόνια, ίσως, το πολύ ισχυρό να έχει τελείως διαφορετική έννοια. Δεν πρέπει να παραγνωρίσουμε ότι μέσα στα κινητά μας τηλεφωνά βρίσκονται επεξεργαστές πολλές τάξεις μεγέθους ισχυρότεροι από αυτούς που χρησιμοποιούνταν, όχι και τόσες πολλές δεκαετίες πριν. Και να μην ξεχνάμε, επίσης, ότι οι επεξεργαστές αυτοί κατοικούσαν μέσα σε υπολογιστικά συστήματα που ο χώρος που καταλάμβαναν μετριόταν σε δωμάτια.

Η εξέλιξη του υλικού προσφέρει ταχύτητα και επιλογές, δεν εξαλείφει όμως την ανάγκη ανάπτυξης γρήγορου λογισμικού. Η κατασκευή ενός επεξεργαστή δύο φορές πιο γρήγορου από τον προκάτοχό του ίσως κάνει την εκτέλεση ενός προγράμματος δύο φορές πιο γρήγορη. Η εύρεση ενός αλγορίθμου που λύνει ένα πρόβλημα σε **logN** βήματα αντί σε **N** βήματα μπορεί να λύσει ένα πρόβλημα ακόμα και εκατομμύρια φορές γρηγορότερα. Θα δούμε κάποια τέτοια παραδείγματα στο κεφάλαιο 10, όπου θα μιλήσουμε για αναζήτηση και ταξινόμηση.

Στη συνέχεια θα δούμε κάποια παραδείγματα μικρών αλγορίθμων. Θα παρατηρήσουμε ότι, αν σκεφτόμαστε έξυπνα, μπορούμε να φτιάξουμε πολύ καλύτερους αλγορίθμους. Η αναζήτηση της βέλτιστης λύσης ενός προβλήματος δεν είναι μόνο χρήσιμη αλλά αποτελεί μία πρόκληση που πάντα κεντρίζει το ενδιαφέρον.

Παρακάτω θα δούμε δύο ομάδες ενδιαφέροντων, έξυπνων και γρήγορων αλγορίθμων: αλγορίθμων για τον υπολογισμό πρώτων αριθμών και αλγορίθμων σχετικών με μετατροπές σε συστήματα αρίθμησης.

8.1 Παράδειγμα 1: πρώτοι αριθμοί

Οι **πρώτοι αριθμοί (prime numbers)** είναι αυτοί που διαιρούνται ακριβώς μόνο με τον εαυτό τους και τη μονάδα. Το 1 δεν θεωρείται πρώτος. Οι πρώτοι αριθμοί μέχρι το 100 είναι οι ακόλουθοι :

> **2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.**

Δεν υπάρχει κάποιος μαθηματικός τύπος για να παράγουμε την ακολουθία αυτήν. Το αν ένας αριθμός είναι πρώτος ή όχι πρέπει να προκύψει από ελέγχους. Ο εύκολος τρόπος για να το κάνουμε αυτό είναι να ελέγξουμε έναν έναν κάθε αριθμό αν είναι πρώτος ή όχι. Η συνάρτηση στο Σχήμα 8.1 μάς τυπώνει όλους τους πρώτους αριθμούς μικρότερους ή ίσους του **N**, ελέγχοντας για κάθε **x** στο διάστημα **[2,N]** αν υπάρχει έστω και ένας από τους αριθμούς στο διάστημα **[2,x-1]** που να διαιρεί ακριβώς τον **x**.

```
1 def printPrimes_1(N):
2     for x in range (2,N+1):
3         prime=True
4         for t in range (2,x):
5             if x%t==0:
6                 prime=False
7         if prime:
8             print(x)
```

Σχήμα 8.1: Πρώτοι αριθμοί - έκδοση 1.

Το **x** παίρνει τιμές από το 2 μέχρι και το **N**, θα ελέγξουμε δηλαδή όλους τους αριθμούς αυτούς αν είναι πρώτοι ή όχι. Μέσα στο σώμα του βρόχου, σε κάθε επανάληψη, θα ελέγξουμε από έναν αριθμό. Αρχικοποιούμε τη μεταβλητή **prime** σε **True**, υποθέτοντας αρχικά ότι ο αριθμός είναι πραγματικά πρώτος. Στη συνέχεια, με τον εσωτερικό βρόχο και τη μεταβλητή **t** διατρέχουμε μία μία τις τιμές από 2 μέχρι **x** και ελέγχουμε αν μία από αυτές διαιρεί ακριβώς το **x**. Στην περίπτωση αυτήν αλλάζουμε την τιμή της **prime** σε **False**. Μόλις τελειώσουν όλοι οι έλεγχοι για τον αριθμό **x**, τότε, αν το **prime** έχει παραμείνει **True**, έχουμε διαπιστώσει ότι ο **x** είναι ένας πρώτος αριθμός και τον τυπώνουμε.

Ο κώδικας αυτός λειτουργεί καλά. Όμως, σίγουρα δεν είναι ο καλύτερος δυνατός. Το πρώτο πράγμα που μπορούμε να κάνουμε για να τον βελτιώσουμε είναι, όταν σιγουρευτούμε ότι ένας αριθμός δεν είναι πρώτος, να σταματάμε

τους ελέγχους. Έτσι, μόλις η μεταβλητή **prime** γίνει **False**, πρέπει να σταματήσουμε και τις επαναλήψεις. Μπορούμε, λοιπόν, να μετατρέψουμε την εσωτερική **for** σε μία **while**, από την οποία θα βγαίνουμε όταν τελειώσουν οι επαναλήψεις ή αν η **prime** γίνει **False**. Ο κώδικας φαίνεται στο Σχήμα 8.2.

```
1  def printPrimes_2(N):
2      for x in range (2,N+1):
3          prime=True
4          t=2
5          while t<x and prime==True:
6              if x%t==0:
7                  prime=False
8              t=t+1
9          if prime:
10             print(x)
```

Σχήμα 8.2: Πρώτοι αριθμοί - έκδοση 2.

Στο Σχήμα 8.3 φαίνεται ένας κώδικας που κάνει ακριβώς το ίδιο πράγμα. Χρησιμοποιεί τη δομή **for-else** της Python και μας επιτρέπει να κάνουμε ακριβώς ό,τι κάνει και ο κώδικας στο Σχήμα 8.2, αλλά σε πολύ μικρότερο χώρο. Μοιάζει και πιο δομημένος, αλλά γούστα είναι αυτά. Δεν έχουμε δει τη δομή **for-else** και είναι μία καλή ευκαιρία να τη δούμε τώρα.

Η **for-else** είναι μια μορφή της **for** η οποία συνδυάζεται με την εντολή **break**. Η **break**, όταν εκτελεστεί, προκαλεί έξοδο από τη δομή **for** μέσα στην οποία βρίσκεται. Η **else** τοποθετείται στο τέλος της **for** στοιχισμένη ακριβώς κάτω από αυτήν. Ο κώδικας που τοποθετείται μέσα στην **else** θα εκτελεστεί μονάχα όταν βγούμε κανονικά από τη **for**, όχι δηλαδή αν βγούμε μέσω της **break**.

Θα δούμε σαν παράδειγμα τον κώδικα της **printPrimes_3**. Όταν η έξοδος από το εσωτερικό **for** γίνει λόγω του **break**, αυτό σημαίνει ότι βγήκαμε από το βρόχο διότι **x%t==0**, δηλαδή διότι αποφασίσαμε ότι ο αριθμός δεν είναι πρώτος. Άρα δεν πρέπει να κάνουμε τίποτε, και πράγματι στην περίπτωση αυτήν ο κώδικας μέσα στο **else** [δηλαδή το **print(x)**] δεν εκτελείται. Αντίθετα, αν βγούμε από τον βρόχο διότι τελειώσαν οι επαναλήψεις, αυτό σημαίνει ότι για κανένα **t** δεν ίσχυσε το **x%t==0**, άρα ο αριθμός είναι πρώτος. Στην περίπτωση αυτή θα εκτελεστεί ο κώδικας μέσα στο **else** [δηλαδή το **print(x)**] πληροφορώντας μας ότι βρέθηκε ένας πρώτος αριθμός.

```
1 def printPrimes_3(N):
2     for x in range (2,N+1):
3         for t in range (2,x):
4             if x%t==0:
5                 break
6         else:
7             print (x)
```

Σχήμα 8.3: Πρώτοι αριθμοί - έκδοση 3.

Ας βελτιώσουμε λίγο ακόμα τον κώδικά μας. Παρατηρήστε ότι, αν **x=ab**, τότε, αν **a<=sqrt(x)**, ισχύει ότι **b>=sqrt(x)**. Δεν είναι δυνατόν, δηλαδή, να πολλαπλασιάσουμε δύο αριθμούς μικρότερους από την τετραγωνική ρίζα του **x** και να μας δώσουν **x**, αλλά ούτε και δύο μεγαλύτερους. Έτσι, η αναζήτηση για διαιρέτες του **x** μπορεί να σταματήσει στην τετραγωνική ρίζα του **x**. Στον κώδικα στο Σχήμα 8.4 παρατηρήστε ότι ο εσωτερικός βρόχος σταματάει στο **int(sqrt(x))**. Το **+1** έχει να κάνει με το ότι θέλουμε οι επαναλήψεις να συμπεριλάβουν και την τετραγωνική ρίζα του **x**.

```
1 def printPrimes_4(N):
2     for x in range (2,N+1):
3         for t in range (2,int(sqrt(x))+1):
4             if x%t==0:
5                 break
6         else:
7             print (x)
```

Σχήμα 8.4: Πρώτοι αριθμοί - έκδοση 4.

Ας εκμεταλλευτούμε κάτι ακόμα. Ο αριθμός 2 είναι ο μόνος άρτιος πρώτος αριθμός. Άρα δεν χρειάζεται να ελέγξουμε ούτε αυτόν ούτε κανέναν άλλο άρτιο. Μπορούμε, να τυπώσουμε τον 2 και να προσπεράσουμε με το κατάλληλο βήμα όλους τους υπόλοιπους άρτιους. Έτσι, στο Σχήμα 8.5 τυπώνουμε τον 2 στην αρχή και ο εξωτερικός βρόχος ξεκινάει από το 3 και έχει βήμα 2, δηλαδή **range(3,N+1,2)**.

```
1 def printPrimes_5(N):
2     print (2)
3     for x in range (3,N+1,2):
4         for t in range (2,int(sqrt(x))+1):
5             if x%t==0:
6                 break
7         else:
8             print (x)
```

Σχήμα 8.5: Πρώτοι αριθμοί - έκδοση 5.

Το ίδιο ακριβώς μπορούμε να κάνουμε και με τον εσωτερικό βρόχο. Τα πολλαπλάσια των άρτιων αριθμών είναι άρτιοι αριθμοί. Αφού γνωρίζουμε ότι οι άρτιοι δεν είναι πρώτοι (χειριστήκαμε διαφορετικά το 2), δεν χρειάζεται να ελέγχουμε και το εάν ένας άρτιος αριθμός διαιρεί το **x**, το οποίο δεν είναι πια άρτιος λόγω της βελτίωσης που κάναμε στο Σχήμα 8.5. Έτσι, το **t** παίρνει πια τιμές από το σύνολο **range(3,int(sqrt(x))+1,2)**. Ο κώδικας, όπως έχει τροποποιηθεί τώρα, φαίνεται στο Σχήμα 8.6.

```
1 def printPrimes_6(N):
2     print (2)
3     for x in range (3,N+1,2):
4         for t in range (3,int(sqrt(x))+1,2):
5             if x%t==0:
6                 break
7         else:
8             print (x)
```

Σχήμα 8.6: Πρώτοι αριθμοί - έκδοση 6.

Στην έβδομη έκδοση του αλγορίθμου (Σχήμα 8.7) δεν επιδιώκουμε κάποια βελτίωση αλλά τοποθετούμε τους πρώτους αριθμούς που βρίσκουμε σε μία λίστα, έστω την **L**, και τους τυπώνουμε όλους μαζί στο τέλος. Θα χρησιμοποιήσουμε τη λίστα αυτή στο Σχήμα 8.8, όπου ο εσωτερικός βρόχος παίρνει τιμές από τη λίστα αυτήν. Τι κάνουμε δηλαδή; Για να ελέγξουμε αν κάποιος αριθμός είναι πρώτος, ελέγχουμε αν τον διαιρεί έστω και ένας από τους πρώτους που έχουμε ήδη βρει. Είναι σωστό; Ναι! Αν ένας αριθμός δεν είναι πρώτος, μπορεί να αναλυθεί σε γινόμενο πρώτων αριθμών μικρότερων από αυτόν. Πριν χρειαστεί να ελεγχθεί εάν ο αριθμός διαιρεί τον **x**, θα έχει ήδη βρεθεί ένας πρώτος που τον διαιρεί.

```
1 def printPrimes_7(N):
2     list=[2]
3     for x in range (3,N+1,2):
4         for t in range (3,int(sqrt(x))+1,2):
5             if x%t==0:
6                 break
7         else:
8             list.append(x)
9     print (list)
```

Σχήμα 8.7: Πρώτοι αριθμοί - έκδοση 7.

```
1 def printPrimes_8(N):
2     list=[2]
3     for x in range (3,N+1,2):
4         for t in list:
5             if x%t==0:
6                 break
7         else:
8             list.append(x)
9     print (list)
```

Σχήμα 8.8: Πρώτοι αριθμοί - έκδοση 8.

Και για τη λίστα **L**, όπως είδαμε και σε προηγούμενες εκδόσεις του αλγορίθμου, δεν απαιτείται να κάνουμε έλεγχο για το αν κάποιο στοιχείο της διαιρεί το **x**, παρά μόνο όταν αυτό είναι μικρότερο από την τετραγωνική ρίζα του **x**. Ένας κομψός τρόπος να το δηλώσουμε είναι αυτός που φαίνεται στο Σχήμα 8.9, όπου η λίστα **L** αντικαθίσταται από τη λίστα:

[i for i in L if i<=sqrt(x)]

Τι σημαίνει αυτό; Η Python μάς δίνει έναν κομψό τρόπο να επιλέξουμε στοιχεία από μία λίστα. Ο παραπάνω κώδικας διαβάζεται ως εξής:

Η λίστα αποτελείται από όλα τα **i** που ανήκουν στην **L** για τα οποία ισχύει ότι είναι μικρότερα ή ίσα με την τετραγωνική ρίζα του **x**.

```
1 def printPrimes_9(N):
2     L=[2]
3     for x in range (3,N+1,2):
4         for t in [i for i in L if i<=sqrt(x)]:
5             if x%t==0:
6                 break
7         else:
8             L.append(x)
9     print (list)
```

Σχήμα 8.9: Πρώτοι αριθμοί - έκδοση 9.

Αλλά και εδώ έχουμε ένα αδύνατο σημείο. Η λίστα **L** συμβαίνει να είναι ταξινομημένη. Αυτό σημαίνει ότι, αν βρω ένα **i** για το οποίο ισχύει ότι **i<=sqrt(x)**, τότε για όλα τα επόμενα **i** που θα ακολουθήσουν θα ισχύει το ίδιο. Ο κώδικας του Σχήματος 8.9 δεν θα αφήσει να εκτελεστεί κανένα από τα **if x%t==0** για τα οποία δεν ισχύει το **i<=sqrt(x)**, ο έλεγχος όμως για **i<=sqrt(x)** θα γίνει για όλη τη λίστα.

Αν θέλουμε να το σταματήσουμε αυτό και να κάνουμε την εκτέλεση πιο γρήγορη, θα πρέπει να εγκαταλείπουμε τις επαναλήψεις όταν **t>sqrt(x)**, όπως φαίνεται στο Σχήμα 8.10. Το πρόβλημα που δημιουργείται τώρα είναι ότι, βγαίνοντας από τον βρόχο, δεν ξέρουμε αν βγήκαμε λόγω της συνθήκης **x%t==0** (όπου ο αριθμός δεν είναι πρώτος) ή λόγω της συνθήκης **t>sqrt(x)** (όπου ο αριθμός είναι πρώτος). Χρειάζεται, λοιπόν, ένας έλεγχος στο τέλος, και εάν βγήκαμε λόγω της συνθήκης **t>sqrt(x)**, τότε τοποθετούμε στην **L** τον πρώτο αριθμό που βρήκαμε, δηλαδή τον **x**.

```
1 def printPrimes_10(N):
2     L=[2]
3     for x in range (3,N+1,2):
4         for t in L:
5             if x%t==0 or t>sqrt(x):
6                 break
7         if t>sqrt(x):
8             L.append(x)
9     print (list)
```

Σχήμα 8.10: Πρώτοι αριθμοί - έκδοση 10.

Πόσο άξιζε ο κόπος που κάναμε; Πόσο πιο γρήγορη είναι η έκδοση 10 από την έκδοση 1; Τοποθετήσαμε έναν μετρητή σε καθεμία από τις εκδόσεις ο

οποίος μετρά πόσες φορές εκτελέστηκε ο έλεγχος **x%t==0**, ο βασικός και συχνότερα επαναλαμβανόμενος έλεγχος του αλγορίθμου, ο οποίος ουσιαστικά μετράει πόσες φορές χρησιμοποιήθηκε η πράξη υπόλοιπο διαίρεσης από κάθε αλγόριθμο, μία καλή προσέγγιση της πολυπλοκότητας και του του χρόνου εκτέλεσής του. Τα αποτελέσματα φαίνονται στον Πίνακα 8.1 και είναι εντυπωσιακά. Σαν είσοδος δόθηκε το **10000**, ζητήθηκε δηλαδή να υπολογιστούν οι πρώτοι αριθμοί μέχρι το 10000.

Εκδοση	Βήματα
1	$50x10^6$
2&3	$6x10^6$
4	$117x10^3$
5	$112x10^3$
6&7	$56x10^3$
8	$771x10^3$
9&10	$40x10^3$

Πίνακας 8.1: Αριθμός φορών που χρησιμοποιήθηκε το υπόλοιπο της διαίρεσης σε κάθε έκδοση του αλγορίθμου.

Με κόκκινο χρώμα φαίνεται ότι, από την πρώτη έως την τελευταία έκδοση του αλγορίθμου, καταφέραμε να τον κάνουμε χίλιες φορές πιο γρήγορο. Ένας αλγόριθμος χίλιες φορές πιο γρήγορος σημαίνει ότι, αν ο ένας κάνει ένα δευτερόλεπτο για να τελειώσει, ο άλλος κάνει 15 με 20 λεπτά.

Στις εκδόσεις 2 και 3 καταφέραμε να ρίξουμε τον χρόνο εκτέλεσης 10 φορές κάτω. Άλλες 50 φορές τον ρίξαμε στην έκδοση 4. Δύο φορές κάτω τον ρίξαμε στην έκδοση 6, ενώ τελικά φτάσαμε μόλις (!) στις 40.000 επαναλήψεις στις τελευταίες εκδόσεις. Άξιζε τον κόπο.

Το πρόβλημα των πρώτων αριθμών το έχουν εξετάσει και οι αρχαίοι Έλληνες. Ο Ερατοσθένης ο Κυρηναίος, αρχαίος Έλληνας μαθηματικός, γεωγράφος, αστρονόμος, γεωδαίτης, ιστορικός και φιλόλογος, πρότεινε μία μέθοδο για τον υπολογισμό των πρώτων αριθμών που έμεινε γνωστή ως το **κόσκινο του Ερατοσθένη (sieve of Eratosthenes)**.

Σύμφωνα με αυτήν, για να υπολογίσουμε τους πρώτους αριθμούς έως τον αριθμό **N**, φτιάχνουμε μία λίστα και τοποθετούμε σε αυτήν ταξινομημένους όλους τους αριθμός από το 2 μέχρι το **N**. Ξεκινάμε με τον πρώτο αριθμό της λίστας, το 2, και αφαιρούμε από τη λίστα όλα τα πολλαπλάσιά του. Μετά πηγαί-

νουμε στον επόμενο αριθμό που βρίσκεται στη λίστα, ο οποίος είναι και αυτός πρώτος. Αφαιρούμε από τη λίστα όλα τα πολλαπλάσιά του. Συνεχίζουμε μέχρι να φτάσουμε στο σημείο όπου ο εξεταζόμενος αριθμός **α** είναι μεγαλύτερος από την τετραγωνική ρίζα του **N**. Είχαμε εξηγήσει γιατί, όταν συζητούσαμε τον κώδικα του Σχήματος 8.4. Στο τέλος, όλοι οι αριθμοί που έχουν μείνει στη λίστα είναι οι πρώτοι αριθμοί από το 1 μέχρι το **N**.

Ο κώδικας που το υλοποιεί φαίνεται στο Σχήμα 8.11. Αρχικά τοποθετούμε τους αριθμούς στη λίστα **x**. Για να είναι πιο εύκολη η πρόσβαση στα στοιχεία της λίστας και το στοιχείο **i** να βρίσκεται στη θέση **i** της λίστας, τοποθετούμε σε αυτήν και το 0 και το 1. Το 1 δεν είναι πρώτος, το αντικαθιστούμε αμέσως με 0. Όπου στον πίνακα υπάρχει 0 σημαίνει ότι σε εκείνο το σημείο δεν υπάρχει πρώτος αριθμός. Στη συνέχεια ξεκινάμε με το 2, όλα τα πολλαπλάσιά του είναι στις άρτιες θέσεις της λίστας. Τοποθετώντας το 2 στην μεταβλητή **i** και χρησιμοποιώντας τη μεταβλητή **j** ως δείκτη, ο οποίος σε κάθε βήμα αυξάνεται κατά 1, παράγουμε όλους τους άρτιους αριθμούς στις θέσεις των οποίων αντικαθιστούμε τον αριθμό που υπάρχει με 0, αφού έχουμε ήδη προαποφασίσει ότι αυτοί δεν είναι πρώτοι. Ο βρόχος σταματάει όταν **i*j>N**, δηλαδή όταν τελειώσει η λίστα.

```
 1  def eratosthenes(N):
 2      x=[i for i in range(0,N+1)]
 3      x[1]=0
 4      i=2
 5      while x[i]**2<=N:
 6          j=2
 7          while i*j<=N:
 8              x[i*j]=0
 9              j=j+1
10          i=i+1
11          while x[i]==0:
12              i=i+1
13      while 0 in x:
14          x.remove(0)
15      print (x)
```

Σχήμα 8.11: Το ”κόσκινο του Ερατοσθένη”.

Στη συνέχεια πρέπει ό,τι κάναμε με το 2 να το κάνουμε με τον επόμενο πρώτο αριθμό. Δεν θα χρειαστεί να πάμε μακριά, διότι είναι το 3 ο αμέσως

επόμενος αριθμός στη λίστα. Θα επαναλάβουμε την ίδια διαδικασία που κάναμε με το 2 και με το 3. Όταν τελειώσουμε με το 3, θα μεταφερθούμε μία ακόμα θέση στη λίστα για να βρούμε τον επόμενο πρώτο αριθμό που θα πάρει τη θέση του 3. Εκεί όμως αντί για το 4 βρίσκεται το 0. Το 4 έχει αντικατασταθεί με 0 όταν σημειώναμε ότι όλα τα πολλαπλάσια του 2 δεν είναι πρώτοι. Θα το αγνοήσουμε και θα προχωρήσουμε στο 5. Ο βρόχος αυτός, ο οποίος είναι ο περισσότερο εξωτερικός βρόχος του κώδικα του Σχήματος 8.11, θα τερματιστεί όταν φτάσουμε στο σημείο όπου ο εξεταζόμενος αριθμός θα είναι μεγαλύτερος από την τετραγωνική ρίζα του **N**.

8.2 Παράδειγμα 2: δυαδικοί αριθμοί

Μπορεί για εμάς να είναι πολύ βολικό να χρησιμοποιούμε δέκα ψηφία για να σχηματίσουμε αριθμούς, αυτό όμως δεν είναι παρά μία σύμβαση που κάναμε. Και ο λόγος δεν είναι άλλος από το ότι οι άνθρωποι έχουν δέκα δάκτυλα. Θα μπορούσαμε αντί για το δέκα να έχουμε σαν βάση κάποιον άλλο αριθμό. Μην ξεχνάμε ότι παλαιότερα κάποιοι χρησιμοποιούσαν τις ντουζίνες (δωδεκάδες) και τα καταφέρνανε μια χαρά. Σήμερα χρησιμοποιούμε το 24 σαν βάση για να χωρίσουμε το ημερονύχτιο σε ώρες. Και δεν υπάρχει κανένα πρόβλημα με αυτό. Την ώρα τη χωρίζουμε σε 60 λεπτά, κάθε λεπτό σε 60 δευτερόλεπτα και κάθε δευτερόλεπτο σε 100 εκατοστά. Πάλι κανένα πρόβλημα (ο γιος μου, από την άλλη, βρήκε πολύ παράξενη αυτήν την ιδέα όταν του το εξήγησα για πρώτη φορά).

Μπορούμε να μετρήσουμε με οποιαδήποτε βάση θέλουμε. Και με το 7 και με το 2 και με το 16. Το πρώτο, το 7, το είπα τυχαία. Τα άλλα δύο όχι. Με το δυαδικό σύστημα αναπαριστά αριθμούς ο υπολογιστής. Εκείνος έχει δύο δάκτυλα: περνάει και δεν περνάει ρεύμα. Το δεκαδικό δεν τον βολεύει, μπορεί όμως όταν επικοινωνεί με εμάς να μετατρέπει τους αριθμούς σε δεκαδικό και μετά να μας τους δίνει. Κάτι ενδιάμεσο είναι το δεκαεξαδικό. Η μετατροπή από το δυαδικό στο δεκαεξαδικό είναι ευκολότερη από τη μετατροπή στο δεκαδικό, και για εμάς το δεκαεξαδικό είναι περισσότερο ευανάγνωστο από ό,τι το δυαδικό.

Παρακάτω θα δούμε κάποιους μικρούς αλγορίθμους υλοποιημένους σε μορφή συνάρτησης που σχετίζονται με τα συστήματα αρίθμησης και τις μετατροπές σε αυτά.

Ο πρώτος από αυτούς είναι η μετατροπή δεκαδικού αριθμού σε δυαδικό. Η μετατροπή ενός δεκαδικού αριθμού, έστω του **x**, σε έναν δυαδικό γίνεται ως εξής:

- Διαιρούμε τον αριθμό **x** με το 2 και σημειώνουμε το πηλίκο και το υπόλοιπο.
- Θέτουμε το **x** να είναι ίσο με το πηλίκο που βρήκαμε.
- Επαναλαμβάνουμε τα βήματα αυτά όσο το πηλίκο είναι μεγαλύτερο του μηδέν.
- Ο δυαδικός αριθμός που ψάχνουμε αποτελείται από τα υπόλοιπα των διαιρέσεων ξεκινώντας από το τελευταίο και καταλήγοντας στο πρώτο.

Ο κώδικας που αντιστοιχεί στον αλγόριθμο αυτόν φαίνεται στο Σχήμα 8.12. Σαν είσοδος στη συνάρτηση **dec2bin** δίνεται ένας θετικός ακέραιος αριθμός. Κανονικά πρέπει να γίνει έλεγχος ότι πράγματι ο αριθμός που δώσαμε είναι τέτοιος, αλλά ας το αγνοήσουμε αυτό.

```
1  def dec2bin(x):
2      r=[]
3      while x>0:
4          y=x%2
5          r=r+[y]
6          x=x//2
7      return(r)
```

Σχήμα 8.12: Μετατροπή δεκαδικού σε δυαδικό.

Το αποτέλεσμα θα αποθηκευτεί σε μία λίστα, ας την ονομάσουμε **r** και ας την αρχικοποιήσουμε, όπως είναι φυσιολογικό, ώστε να είναι κενή. Στη συνέχεια εκτελούμε τις επαναλήψεις του βρόχου, μέσα στον οποίο γίνεται η διαίρεση του αριθμού **x**. Μέσω της μεταβλητής **y** το υπόλοιπο σημειώνεται στη λίστα **r**, ενώ το πηλίκο τοποθετείται στη μεταβλητή **x** αντικαθιστώντας την παλιά τιμή της. Ο βρόχος ολοκληρώνεται όταν η συνθήκη στη **while** πάψει να ισχύει, δηλαδή όταν το **x** γίνει μηδέν. Το αποτέλεσμα βρίσκεται στη λίστα **r** και από αυτήν επιστρέφεται. Στη θέση 0 της λίστας **r** βρίσκεται το λιγότερο σημαντικό ψηφίο του αριθμού.

Η μετατροπή από δυαδικό σε δεκαδικό είναι πολύ πιο εύκολη. Έστω ότι έχουμε τον δυαδικό αριθμό:

$$d = d_n d_{n-1} \dots d_2 d_1 d_0$$

όπου d_i είναι ένα δυαδικό ψηφίο. Ο αντίστοιχος δεκαδικός αριθμός δίνεται από τον τύπο:

$$D = \sum_{i=0}^{n} d_i 2^i$$

Άρα, χρειαζόμαστε έναν βρόχο ο οποίος σε κάθε του βήμα θα κάνει τους επιμέρους πολλαπλασιασμούς των ψηφίων d_i με το $2i$ και θα αθροίζει τα επιμέρους γινόμενα. Το άθροισμα αυτό είναι και το αποτέλεσμα που επιστρέφει η συνάρτηση.

Ο κώδικας φαίνεται στο Σχήμα 8.13. Το **x** είναι η λίστα με τα ψηφία του δυαδικού αριθμού που δίνεται σαν είσοδος. Κανονικά πρέπει να γίνει έλεγχος ότι η λίστα αποτελείται πράγματι από δυαδικά ψηφία (δηλαδή από 0 ή 1). Η μεταβλητή **result** αθροίζει τα επιμέρους γινόμενα **x[i]*2**i**, ενώ ο βρόχος **for** εκτελείται μέχρι να επισκεφτούμε όλα τα στοιχεία της λίστας. Στην τελευταία γραμμή επιστρέφεται το αποτέλεσμα.

```
1 def bin2dec(x):
2     result=0
3     for i in range(len(x)):
4         result+=x[i]*2**i
5     return result
```

Σχήμα 8.13: Μετατροπή δυαδικού σε δεκαδικό.

Θα δούμε, τέλος, ακόμα μία συνάρτηση η οποία υπολογίζει και επιστρέφει το συμπλήρωμα ως προς 2 **(complement of 2)** ενός δυαδικού αριθμού. Το συμπλήρωμα ως προς 2 ενός δυαδικού αριθμού χρησιμεύει στην αρχιτεκτονική υπολογιστών για την αναπαράσταση αρνητικών αριθμών. Σύμφωνα με αυτό, όταν το περισσότερο σημαντικό ψηφίο ενός αριθμού με **n** ψηφία είναι 0, τότε ο αριθμός αυτός είναι θετικός και το μέτρο του δίνεται από τα υπόλοιπα **n-1** ψηφία. Όταν, όμως, το περισσότερο σημαντικό ψηφίο του αριθμού είναι 1, τότε ο αριθμός αυτός είναι αρνητικός και για να βρούμε το μέτρο του αριθμού πρέπει να υπολογίσουμε το συμπλήρωμα ως προς 2 των ψηφίων του. Έτσι, παίρνουμε το συμπλήρωμα του αριθμού ως προς 1, δηλαδή αντικαθιστούμε κάθε 1 με 0 και κάθε 0 με 1 και στο τέλος σε ό,τι βρούμε προσθέτουμε το 1. Αν προκύψει κρατούμενο στο αριστερότερο ψηφίο, το αγνοούμε.

Ένας έξυπνος, γρήγορος και εύκολα υλοποιήσιμος τρόπος για να το κάνουμε αυτό είναι να συμπληρώσουμε ως προς 1 όλα τα ψηφία που βρίσκονται αριστερά από το δεξιότερο 1. Αν δηλαδή πρέπει να βρούμε το συμπλήρωμα ως προς 2 του δυαδικού αριθμού **11001110**, τότε, αφού το δεξιότερο 1 βρίσκεται στη δεύτερη από δεξιά θέση, θα συμπληρώσουμε τον αριθμό **110011** και θα αφήσουμε τα τελευταία δύο ψηφία, τα 10, ως έχουν. Το αποτέλεσμα δηλαδή θα είναι **00110010**.

Να αναφέρουμε, ώστε να γίνει λίγο πιο κατανοητός ο τρόπος αποθήκευσης

ενός αρνητικού αριθμού σε μορφή συμπληρώματος ως προς 2, ότι ο αριθμός που συμβολίζεται με **11001110** είναι αρνητικός και το μέτρο του είναι αυτό που βρήκαμε παραπάνω, δηλαδή το **00110010** ή, αν προτιμάτε, το 50. Ο αριθμός **11001110**, δηλαδή, είναι το -50.

Η υλοποίηση της συνάρτησης φαίνεται στο Σχήμα 8.14. Χρησιμοποιούμε μία μεταβλητή, την **i**, η οποία διατρέχει όλη τη λίστα **x** που δίνεται σαν είσοδος. Ο πρώτος βρόχος, το **while**, εκτελείται όσο το **x[i]** είναι μηδέν και μεταφέρει τα μηδενικά αυτά στη λίστα **y**, που θα αποτελέσει και το αποτέλεσμα όταν ολοκληρωθεί η εκτέλεση της συνάρτησης. Παρατηρήστε ότι, αν η μεταβλητή **i** πάρει τιμή μεγαλύτερη από το μήκος της λίστας, τότε η εκτέλεση εξέρχεται από τον βρόχο και δεν μπαίνει σε καμία από τις **if** και **while** που ακολουθούν.

```
1  def compl2(x):
2      y=[]
3      i=0
4      while i<len(x) and x[i]==0:
5          y.append(0)
6          i=i+1
7      if i<len(x):
8          y.append(1)
9          i=i+1
10     while i<len(x):
11         if x[i]==0:
12             y.append(1)
13         else:
14             y.append(0)
15         i=i+1
16     return y
```

Σχήμα 8.14: Υπολογισμός συμπληρώματος ως προς 2.

Όταν τώρα η ροή της εκτέλεσης βγει έξω από το πρώτο **while** (και δεν έχουν τελειώσει τα στοιχεία της λίστας **x**), τότε αυτό σημαίνει ότι βρήκαμε το δεξιότερο 1. Ελέγχουμε, λοιπόν, με ένα **if** ότι πράγματι δεν έχουν τελειώσει τα στοιχεία της λίστας **x** και περνάμε, στην περίπτωση αυτήν, το 1 στη λίστα **y**. Τώρα, όσα ψηφία μάς έχουν απομείνει θα πρέπει να αντιστραφούν, τα 0 να γίνουν 1 και τα 1 να γίνουν 0. Αυτό το κάνει το τελευταίο **while**, το οποίο θα τερματιστεί όταν τελειώσουν όλα τα στοιχεία της **x**. Με την τελευταία εντολή της συνάρτησης, όπως συνήθως συμβαίνει, επιστρέφουμε το αποτέλεσμα.

Μικρός και έξυπνος αλγόριθμός σαν όλους που είδαμε στο κεφάλαιο αυτό.

Περισσότερους τέτοιους αλγόριθμους μπορείτε να βρείτε στο κεφάλαιο 6 του βιβλίου [1].

Ασκήσεις που μπορείτε να κάνετε μόνοι σας:

- **Μέγιστος κοινός διαιρέτης** δύο ακέραιων αριθμών είναι ο μεγαλύτερος ακέραιος αριθμός που διαιρεί και τους δύο ακριβώς. Ένας αλγόριθμος για να βρεθεί ο μέγιστος κοινός διαιρέτης είναι ο αλγόριθμος του Ευκλείδη, αρχαίου Έλληνα μαθηματικού. Ο αλγόριθμος ξεκινά με ένα ζεύγος θετικών ακεραίων των οποίων τον μέγιστο κοινό διαιρέτη θέλουμε να υπολογίσουμε. Σε κάθε του βήμα αντικαθιστά τον μεγαλύτερο από τους δύο αριθμούς με τη διαφορά του από τον μικρότερο. Η διαδικασία επαναλαμβάνεται έως ότου οι αριθμοί γίνουν ίσοι. Όταν οι αριθμοί γίνουν ίσοι, τότε αυτός ο αριθμός είναι και ο μέγιστος κοινός διαιρέτης του αρχικού ζεύγους. Υλοποιήστε μία συνάρτηση σε γλώσσα Python η οποία να υπολογίζει με τον αλγόριθμο του Ευκλείδη τον μέγιστο κοινό διαιρέτη δύο αριθμών.
- Με παρόμοιο τρόπο μπορούμε να υπολογίσουμε τον μέγιστο κοινό διαιρέτη χρησιμοποιώντας το υπόλοιπο της διαίρεσης. Σκεφτείτε πώς, και συμπληρώστε τον κώδικα που ακολουθεί:

```
def gcd(a, b):
while b: a, b = b, ...
return a
```

- Να γράψετε έναν αλγόριθμο που να αναλύει έναν αριθμό σε γινόμενο πρώτων παραγόντων.
- Να γράψετε ένα πρόγραμμα που να ελέγχει αν δύο αριθμοί είναι φιλικοί. Δύο αριθμοί είναι φιλικοί, αν ο καθένας ισούται με το άθροισμα όσων διαιρούν τον άλλο. Δύο γνωστοί φιλικοί αριθμοί είναι οι 220 και 284.

Βιβλιογραφία

1. Eric Roberts. (2004).**Η Τέχνη και Επιστήμη της C**. Μετάφραση: Γιώργος Στεφανίδης, Παναγιώτης Σταυρόπουλος, Αλέξανδρος Χατζηγεωργίου, Εκδόσεις Κλειδάριθμός

Κεφάλαιο 9: Αναδρομή

Ο τρόπος με τον οποίο σκεφτήκαμε και σχεδιάσαμε τις συναρτήσεις στο προηγούμενο κεφάλαιο ακολουθούσε τη φιλοσοφία του προγραμματισμού που είχαμε αναπτύξει σε όλο το προηγούμενο βιβλίο. Και πώς θα μπορούσε να μην ήταν έτσι άλλωστε;

Στο κεφάλαιο αυτό θα αναζητήσουμε κάτι διαφορετικό. Θα δούμε έναν τρόπο να υλοποιούμε συναρτήσεις χρησιμοποιώντας **αναδρομή (recursion)**. Η αναδρομή είναι κάτι το οποίο έχουμε δει στα μαθηματικά στο σχολείο. Θυμηθείτε τις **αναδρομικές ακολουθίες (recursive sequences)** στις οποίες ορίζαμε τον πρώτο όρο της ακολουθίας και τον ν-οστό όρο με βάση τον (ν-1)-οστό όρο. Το ίδιο θα κάνουμε και εδώ, μελετώντας και περιγράφοντας λειτουργίες που μπορούν να γραφούν αναδρομικά με έναν αρκετά διαφορετικό και ιδιαίτερα ενδιαφέροντα τρόπο.

Το πρώτο πράγμα που θα αναρωτηθεί κανείς είναι γιατί να το κάνουμε αυτό. Είναι καλύτερος τρόπος προγραμματισμού; Είναι γρηγορότερος ο κώδικας που παράγεται; Είναι ευκολότερος ο κώδικας που αναπτύσσουμε; Οι απαντήσεις προφανώς δεν μπορούν να δοθούν μονολεκτικά ή έστω σύντομα. Ένα μπορεί να ειπωθεί σύντομα: είναι κάτι διαφορετικό και έχει τα πλεονεκτήματά του και τα μειονεκτήματά του. Αν μη τι άλλο, είναι ένας πολύ πρωτότυπος τρόπος να γράφεις κώδικα. Ας τον μελετήσουμε με προσοχή.

9.1 Αναδρομικές συναρτήσεις και μαθηματικά

Στα μαθηματικά υπάρχουν πολλές περιγραφές που μπορούν να γίνουν με τρόπο αναδρομικό. Να ξεκινήσουμε με τις αναδρομικές ακολουθίες που αναφέραμε και πιο πάνω. Ας υποθέσουμε ότι θέλουμε να περιγράψουμε την ακολουθία των ακέραιων αριθμών που διαιρούνται ακριβώς με το 3. Θα πρέπει

να ορίσουμε τον πρώτο όρο της ακολουθίας, ο οποίος θα είναι το 3 (ή το 0 αν προτιμάτε να ξεκινήσουμε από εκεί), καθώς και να περιγράψουμε το πώς προκύπτει ο κάθε επόμενος όρος από τον προηγούμενό του. Ο κάθε επόμενος όρος, λοιπόν, προκύπτει αν προσθέσουμε στον προηγούμενό του το 3. Έτσι, με έναν πιο μαθηματικό συμβολισμό, η ακολουθία δίνεται από τους ορισμούς:

$$\alpha_1 = 3$$

$$\alpha_v = \alpha_{v-1} + 3$$

Ένα παράδειγμα προβλήματος που έχουμε εξετάσει και που μπορεί να περιγραφεί αναδρομικά είναι ο υπολογισμός του παραγοντικού. Το παραγοντικό ενός αριθμού i προκύπτει αν το παραγοντικό του προηγούμενού του αριθμού, του **i-1**, το πολλαπλασιάσουμε με το **i**. Χρειάζεται να ορίσουμε και το παραγοντικό του πρώτου όρου της σειράς, του 0 στη συγκεκριμένη περίπτωση. Το παραγοντικό του 0 ορίζεται ως 1 Έτσι, το πρόβλημα ολοκληρωμένο με μαθηματικούς συμβολισμούς περιγράφεται ως εξής:

$$f_0 = 1$$

$$f_v = f_{v-1} * v$$

Ένα παρόμοιο παράδειγμα είναι και η ύψωση σε δύναμη, όπου ο κάθε επόμενος όρος προκύπτει με τον πολλαπλασιασμό του προηγούμενου όρου με τη βάση, τον αριθμό δηλαδή που υψώνουμε σε δύναμη. Θα δούμε παρακάτω μια ταινία γι' αυτό. Ας δούμε όμως πρώτα ένα ακόμα μαθηματικό πρόβλημα που η αναδρομική του υλοποίηση παρουσιάζει μεγάλο ενδιαφέρον: την ακολουθία Fibonacci.

Στην ακολουθία **Fibonacci** κάθε όρος προκύπτει ως το άθροισμα των δύο προηγούμενών του όρων. Δεν θα δυσκολευτεί κανείς να φανταστεί ότι, για να οριστεί η ακολουθία, δεν αρκεί να ορίσουμε τον πρώτο της όρο μόνο, αλλά είναι απαραίτητο να ορίσουμε τους δύο πρώτους όρους. Ένας εύκολος τρόπος να καταλάβουμε γιατί είναι να σκεφτούμε ότι, αν δεν ορίσουμε τους δύο πρώτους όρους, δεν είναι δυνατόν να υπολογίσουμε τον τρίτο, ο οποίος βασίζεται, όπως και κάθε άλλος, στους δύο προηγούμενους. Συνήθως ορίζουμε τον πρώτο όρο σε 0 ή 1 και τον δεύτερο σε 1.

9.2 Απόδειξη με επαγωγή

Στα σχολικά μας χρόνια είχαμε μάθει να αποδεικνύουμε θεωρήματα με επαγωγή. Η φιλοσοφία της απόδειξης με επαγωγή πάλι είναι πολύ κοντινή σε ό,τι

συζητούμε, είναι μάλλον οι δύο όψεις του ίδιου νομίσματος. Θα χρησιμοποιήσω ένα απλοποιημένο πρόβλημα στο οποίο θα αποδείξουμε επαγωγικά ότι όλα τα ντόμινο που τοποθετούμε σε σειρά το ένα μετά το άλλο θα πέσουν κάτω! Και δεν θα το κάνω μόνο επειδή μου αρέσει να βλέπω ντόμινο να πέφτουν με τάξη κάτω το ένα μετά το άλλο, αλλά και γιατί με τον τρόπο αυτόν λειτουργεί μια αναδρομική συνάρτηση σαν αυτές που θα υλοποιήσουμε σε λίγο.

Τι κάνουμε λοιπόν όταν τοποθετούμε τα ντόμινο; Βάζουμε κάθε ντόμινο τόσο μακριά από το προηγούμενό του έτσι ώστε να είμαστε σίγουροι ότι, αν πέσει το προηγούμενο ντόμινο, τότε θα πέσει και αυτό. Αν το εξασφαλίσουμε αυτό, είμαστε σίγουροι ότι, αν εμείς ρίξουμε το πρώτο ντόμινο, τότε θα πέσουν όλα ανεξαιρέτως τα ντόμινο.

Το ίδιο ακριβώς κάνουμε και στην απόδειξη με επαγωγή. Δεχόμαστε ότι κάτι ισχύει για **v** και με αυτό δεδομένο αποδεικνύουμε ότι ισχύει και για **v+1**. Πρέπει πάλι να ρίξουμε το πρώτο ντόμινο, να εξετάσουμε τι γίνεται για **v=1** (ή για κάποιο άλλο **v** που έχει νόημα για το πρόβλημα που πάμε να λύσουμε).

Θα δείξουμε λοιπόν ότι:

$$1 + 2 + 3 + \ldots + v = \frac{v(v+1)}{2}$$

Έστω ότι ονομάζουμε **T(v)** το άθροισμα:

$$T(v) = 1 + 2 + 3 + \ldots + v$$

Τότε διπλασιάζοντας τα δύο μέλη της εξίσωσης μπορούμε να έχουμε:

$$2T(v) = 1 + 2 + 3 + \ldots + (v-1) + v + v + (v-1) + (v-2) + \ldots + 2 + 1$$

Προσθέτοντας τους όρους του δεξιού μέλους της εξίσωσης που απέχουν **v** όρους μεταξύ τους παίρνουμε:

$$2T(v) = (v+1) + (v+1) + ... + (v+1) = v(v+1)$$

και άρα:

$$T(v) = \frac{v(v+1)}{2}$$

Ας το δείξουμε τώρα και αναδρομικά. Για **v=1** έχουμε:

$$1 = \frac{1 \times 2}{2} = 1$$

άρα ισχύει. Έστω ότι ισχύσει για **v**, θα δείξουμε ότι ισχύει και για **v+1**, ότι δηλαδή:

$$1 + 2 + 3 + \ldots + v + (v + 1) = \frac{(v + 1)(v + 2)}{2}$$

Πράγματι έχουμε:

$$1 + 2 + 3 + \ldots + v + (v + 1) = \frac{v(v + 1)}{2} + (v + 1) =$$
$$= \frac{v(v + 1) + 2(v + 1)}{2} = \frac{(v + 1)(v + 2)}{2}$$

άρα η αρχική μας υπόθεση ισχύει.

9.3 Αναδρομικές συναρτήσεις στην Πληροφορική

Πώς θα μπορούσε η Πληροφορική να διαφέρει από τα μαθηματικά; Με τρόπο ανάλογο με ό,τι είδαμε παραπάνω, σε πολλές γλώσσες προγραμματισμού, σχεδόν σε όλες που υποστηρίζουν το διαδικασιακό μοντέλο, ο προγραμματιστής μπορεί να υλοποιήσει κάποιους υπολογισμούς αναδρομικά.

Η φιλοσοφία φυσικά δεν αλλάζει. Θα αλλάξουμε λίγο τη διατύπωση που κάναμε παραπάνω με μία ισοδύναμη. Αντί να λέμε ότι θεωρούμε ότι κάτι ισχύει για **v** και πάμε να αποδείξουμε με βάση αυτό ότι ισχύει και για **v+1**, θα πούμε ότι θα φτιάξουμε μία συνάρτηση που να υπολογίζει κάτι για την τιμή **v**, θεωρώντας ότι η συνάρτηση λειτουργεί **για v-1**. Αυτό είναι πιο κοντά στη διατύπωση που είχαμε κάνει στα ντόμινο, ότι, αν πέσει το προηγούμενο, πρέπει να πέσει και το επόμενο.

Να μην να ξεχάσουμε ότι πρέπει να πέσει και το πρώτο ντόμινο και ότι αυτό πρέπει να γίνει με διαφορετικό τρόπο από ό,τι θα πέσουν τα υπόλοιπα ντόμινο. Αντίστοιχα, πρέπει να γράψουμε τη συνάρτηση ώστε να λειτουργεί και για **v=1** (ή κάτι άλλο, ανάλογα με το πρόβλημα), ξεχωριστά από τα υπόλοιπα **v**. Η συνταγή λοιπόν έχει ως εξής:

- Φτιάχνουμε τη συνάρτηση να λειτουργεί για μια αρχική τιμή.
- Υποθέτουμε ότι (με κάποιο μαγικό τρόπο) η συνάρτηση λειτουργεί αν κληθεί με παράμετρο το **v-1**.
- Υλοποιούμε τη συνάρτηση ώστε να λειτουργεί αν κληθεί με παράμετρο **v**, επιτρέποντας να καλέσουμε μέσα σε αυτήν τη συνάρτηση με παράμετρο **v-1**.

Μένει να δούμε λίγο το μαγικό κομμάτι, αλλά θα το κάνουμε αργότερα. Πριν χαλάσουμε τη μαγεία, ας δούμε λίγες αναδρομικές συναρτήσεις με τη βοήθεια της Python.

9.4 Το παραγοντικό και η ύψωση σε δύναμη ως αναδρομικές συναρτήσεις

Η μαθηματική αναδρομική περιγραφή του παραγοντικού ορίζει τον πρώτο όρο n=0, το οποίο εξ' ορισμού είναι ίσο με 1, ενώ κάθε άλλο όρο τον ορίζει σαν το γινόμενο του παραγοντικού του προηγούμενου όρου και του n. Είδαμε παραπάνω τον μαθηματικό ορισμό. Το ίδιο ακριβώς κάνει και ο κώδικας του Σχήματος 9.1.

```
1  def factorial(n):
2      if n==0:
3          return 1
4      else:
5          return n*factorial(n-1)
```

Σχήμα 9.1: Συνάρτηση που υπολογίζει το παραγοντικό ενός αριθμού.

Στην πρώτη γραμμή ορίζεται το όνομα της συνάρτησης, ας την πούμε **factorial**. Το κυρίως σώμα της συνάρτησης αποτελείται από μία εντολή απόφασης η οποία διαχωρίζει δύο περιπτώσεις: αν το **n** είναι ίσο με μηδέν και αν το **n** δεν είναι ίσο με μηδέν. Στην πρώτη περίπτωση επιστρέφει το 1 σαν αποτέλεσμα, ενώ στη δεύτερη χρησιμοποιεί την κλήση **factorial(n-1)**, για την οποία έχουμε δεχτεί ότι λειτουργεί, υπολογίζει το παραγοντικό του **n-1**, το πολλαπλασιάζει με το **n** και βρίσκει το παραγοντικό του **n**, το οποίο και επιστρέφει σαν αποτέλεσμα.

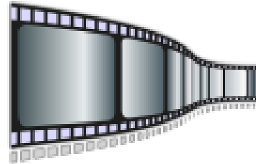
```
1  def factorial_2(n):
2      if n<0:
3          return 'error: negative value'
4      elif n==0:
5          return 1
6      else:
7          return n*factorial(n-1)
```

Σχήμα 9.2: Ολοκληρωμένη συνάρτηση για το παραγοντικό.

Αν θέλουμε να είμαστε περισσότερο σωστοί ως προγραμματιστές, πρέπει να κάνουμε κάτι για την περίπτωση που η συνάρτηση κληθεί με αρνητικό αριθμό σαν όρισμα. Δεν είναι δύσκολο, η δομή απόφασης πρέπει να έχει τώρα

τρία σκέλη, ένα για **n<0**, οπότε θα επιστρέφει μήνυμα λάθους, και τα άλλα δύο για **n=0** και **n>0 (else)**, τα οποία δεν αλλάζουν. Στο Σχήμα 9.2 φαίνεται ο ολοκληρωμένος κώδικας.

Για την ύψωση ενός αριθμού σε δύναμη παρακολουθήστε την Ταινία 9.1. Ο κώδικάς της φαίνεται στο Σχήμα 9.3.



Ταινία 9.1: Ύψωση αριθμού σε δύναμη.

http://repfiles.kallipos.gr/file/22389

```
1 def power(a,n):
2     if n==0:
3         return 1
4     elif n>0:
5         return a*power(a,n-1)
6     elif n<0 and a==0:
7         return 'error: not defined'
8     else:
9         return power(a,n+1)/a
```

Σχήμα 9.3: Ύψωση αριθμού σε δύναμη.

9.5 Η ακολουθία των αριθμών Fibonacci

Κάθε όρος της ακολουθίας των αριθμών **Fibonacci** αποτελείται από το άθροισμα των δύο προηγούμενων όρων. Φυσικά, για να έχει νόημα κάτι τέτοιο πρέπει να οριστούν ξεχωριστά οι δύο πρώτοι όροι. Έτσι, αν ο πρώτος όρος είναι το 0 και ο δεύτερος το 1, τότε η μαθηματική περιγραφή της ακολουθίας Fibonacci είναι η ακόλουθη:

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_v = fib_{v-1} + fib_{v-2}$$

Θα ξεκινήσουμε με μία επαναληπτική υλοποίηση της ακολουθίας. Ας υποθέσουμε λοιπόν ότι θέλουμε να φτιάξουμε μία συνάρτηση που επιστρέφει τον ν-οστό όρο της ακολουθίας Fibonacci. Μία πρώτη προσέγγιση θα ήταν να χρησιμοποιήσουμε μία λίστα όπου και θα αποθηκεύουμε τους όρους της ακολουθίας. Αυτό δεν είναι δύσκολο. Ο κώδικας φαίνεται στο Σχήμα 9.4. Οι δύο πρώτοι όροι επιστρέφονται χωρίς να χρειαστεί κάποιος υπολογισμός, ενώ, αν ζητείται όρος μεγαλύτερος από τον δεύτερο, τότε οι δύο πρώτοι όροι τοποθετούνται στην αρχή μίας λίστας. Για τον τρίτο έως και τον ν-οστό όρο, κάθε όρος υπολογίζεται σαν το άθροισμα των δύο προηγούμενων όρων στη λίστα και τοποθετείται στη θέση του στο τέλος της λίστας για να χρησιμοποιηθεί σε υπολογισμούς που θα ακολουθήσουν ή να επιστραφεί με τη **return**, αν είναι ο τελευταίος όρος που υπολόγισε ο βρόχος, δηλαδή το ζητούμενο αποτέλεσμα.

```
1  def Fibonacci_2(n):
2      if n<0:
3          return 'error: negative value'
4      elif n==1:
5          return 0
6      elif n==2:
7          return 1
8      else:
9          a=0
10         b=1
11         for i in range(3,n+1):
12             c=a+b
13             a=b
14             b=c
15         return c
```

Σχήμα 9.4: Αριθμοί Fibonacci με χρήση λίστας.

Η λύση αυτή είναι εύκολη και φαντάζομαι αρκετά κατανοητή. Έχει όμως ένα σημαντικό μειονέκτημα: αποθηκεύει όλους τους αριθμούς της ακολουθίας, χωρίς κάτι τέτοιο να είναι απαραίτητο. Η επόμενή μας προσέγγιση, η οποία φαίνεται στο Σχήμα 9.5, δεν απαιτεί τη χρήση λίστας. Χρησιμοποιεί τρεις μεταβλητές, μία για τον όρο που θα υπολογιστεί τώρα, μία για τον προηγούμενό του (αυτόν που υπολογίστηκε στην προηγούμενη επανάληψη) και μία για τον προ-προηγούμενό του. Πρέπει φυσικά να φροντίσουμε ώστε, όταν ένας όρος υπολογιστεί, για να είμαστε έτοιμοι για τον υπολογισμό του επόμενου

όρου, ο μέχρι τώρα προηγούμενος όρος πρέπει να πάρει τη θέση του προ-προηγούμενου και ο νέος όρος πρέπει να πάρει τη θέση του προηγούμενου.

```
1  def Fibonacci_1(n):
2      if n<0:
3          return 'error: negative value'
4      elif n==1:
5          return 0
6      elif n==2:
7          return 1
8      else:
9          L = [0,1]
10         for i in range(3,n+1):
11             L.append(L[-1]+L[-2])
12         print (L)
13         return L[-1]
```

Σχήμα 9.5: Αριθμοί Fibonacci με τη χρήση τριών μεταβλητών.

```
1  def Fibonacci_3(n):
2      if n<0:
3          return 'error: negative value'
4      elif n==1:
5          return 0
6      elif n==2:
7          return 1
8      else:
9          a=0
10         b=1
11         for i in range(3,n+1):
12             b=a+b
13             a=b-a
14         return b
```

Σχήμα 9.6: Αριθμοί Fibonacci με τη χρήση δύο μεταβλητών.

Αν το καταλάβαμε ως εδώ πήγαμε πολύ καλά. Αν θέλετε να το προχωρήσετε λίγο περισσότερο προσπαθήστε να καταλάβετε τον κώδικα του Σχήματος 9.6, όπου εμφανίζεται μία παραλλαγή της προηγούμενης λύσης στην

οποία απαιτούνται μόνο δύο μεταβλητές, αντί για τρεις, για να κρατηθούν οι απαραίτητοι όροι της ακολουθίας.

Ο σκοπός της υποενότητας αυτής είναι να φτάσουμε στην αναδρομική υλοποίηση. Είχαν όμως τόσο ενδιαφέρον οι παραπάνω μη αναδρομικές υλοποιήσεις που δεν μπόρεσα να αντισταθώ στο να τις αναλύσω λίγο περισσότερο. Η αναδρομική υλοποίηση προκύπτει φυσικά από τον μαθηματικό αναδρομικό ορισμό. Ο κώδικας φαίνεται στο Σχήμα 9.7. Αρχικά φροντίζουμε ώστε η συνάρτηση να επιστρέφει σωστό αποτέλεσμα για τους δύο πρώτους όρους. Για κάθε άλλο όρο η συνάρτηση καλεί αναδρομικά δύο φορές τον εαυτό της και εκφράζει τον νέο όρο σαν άθροισμα των δύο προηγούμενων όρων.

```
1  def Fibonacci(n):
2      if n<0:
3          return 'error: negative value'
4      elif n==1:
5          return 0
6      elif n==2:
7          return 1
8      else:
9          return Fibonacci(n-1)+Fibonacci(n-2)
```

Σχήμα 9.7: Αναδρομικός υπολογισμός της ακολουθίας Fibonacci.

Δεν ξέρω ποιος κώδικας από τους τέσσερις σας δυσκόλεψε περισσότερο. Εμένα πιο εύκολος μου φαίνεται ο τελευταίος. Αλλά, όπως θα δούμε παρακάτω στην υποενότητα 9.7, η ευκολία αυτή δεν είναι χωρίς τίμημα.

9.6 Εκτέλεση μια αναδρομικής συνάρτησης βήμα προς βήμα

Επειδή, δυστυχώς, μαγεία υπάρχει μόνο στον κόσμο των παραμυθιών, πρέπει κάπου εδώ να απομυθοποιήσουμε το γιατί η υπόθεση ότι η συνάρτηση υπολογίζει σωστά κάποιους προηγούμενους όρους μάς επιτρέπει να υπολογίζουμε σωστά τον επόμενο όρο. Θα το δούμε καλύτερα με τη βοήθεια της Ταινίας 9.2, αλλά ας παρακολουθήσουμε την εκτέλεση ενός παραδείγματος υπολογισμού του παραγοντικού αναδρομικά. Έστω ότι θέλουμε να υπολογίσουμε το παραγοντικό του 5. Έχουμε λοιπόν:

factorial(5)=5*factorial(4)

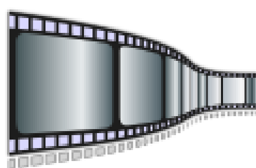
σύμφωνα με τον αναδρομικό ορισμό και τον κώδικα που γράψαμε, αφού η συνάρτηση θα καλέσει αναδρομικά τον εαυτό της για να υπολογίσει το **factorial(4)** και ό,τι βρει θα το πολλαπλασιάσει με το 5. Όμως η **factorial(4)**, πάλι για να υπολογιστεί, θα καλέσει την **factorial(3)**. Έτσι έχουμε:

factorial(5)=5*factorial(4)=5*4*factorial(3)=

=5*4*3*factorial(2)=5*4*3*2*factorial(1)=%*4*3*2*1*factorial(0)

Όμως, η **factorial(0)** ορίζεται με διαφορετικό τρόπο και δεν απαιτείται αναδρομική κλήση για να εκτελεστεί, αφού στον κώδικα της συνάρτησης έχει οριστεί να επιστρέφει 1. Αν στην παραπάνω σχέση αντικαταστήσουμε το **factorial(0)** με το 1, έχουμε φτάσει στο αποτέλεσμά μας (το οποίο εδώ είναι το 120).

Παρακολουθήστε την Ταινία 9.2, όπου τα παραπάνω αναπαρίστανται με τον τρόπο που πραγματικά γίνονται οι υπολογισμοί μέσα στον υπολογιστή με τη χρήση της στοίβας εκτέλεσης.



Ταινία 9.2: Εκτέλεση μιας αναδρομικής συνάρτησης βήμα προς βήμα.

http://repfiles.kallipos.gr/file/22390

9.7 Οι επιπτώσεις της αναδρομής στον χρόνο εκτέλεσης

Είμαι σίγουρος ότι κατάφερα τουλάχιστον να σας πείσω ότι η αναδρομή είναι ένας τελείως διαφορετικός τρόπος να γράφουμε συναρτήσεις. Σίγουρα δεν σας έπεισα για την αναγκαιότητα να γράφουμε κώδικα με αυτόν τον τρόπο. Από την άλλη, είμαι σίγουρος ότι δεν έχετε καμία αμφιβολία ότι, αν έχουμε τον ορισμό ενός προβλήματος με τρόπο αναδρομικό, είναι πολύ εύκολο να γράψουμε την αναδρομική συνάρτηση που το υλοποιεί. Σε κάποιες περιπτώσεις, όπως στους αριθμούς Fibonacci, ίσως να το θεωρήσατε και εσείς πιο εύκολο. Πιστέψτε με όμως, υπάρχουν παραδείγματα που η αναδρομική υλοποίηση είναι τόσο ευκολότερη από μία μη αναδρομική, που δεν διανοούμαστε καν να πληρώσουμε το κόστος. Τέτοιοι αλγόριθμοι είναι ο αλγόριθμος **γρήγορης ταξινόμησης (quicksort)**, που θα δούμε σε επόμενο κεφάλαιο, αλλά και διάφορες

λειτουργίες πάνω σε δέντρα, οι οποίες όμως δεν αποτελούν αντικείμενο αυτού του βιβλίου. Πληροφορίες γύρω από τα δέντρα αλλά και κώδικες που υλοποιούν λειτουργίες σε αυτά μπορείτε να αναζητήσετε σε κάποιο βιβλίο δομών δεδομένων.

Τα πράγματα, όπως στις περισσότερες περιπτώσεις, έχουν περισσότερες από μία πλευρές. Για την ευκολία που μας παρέχει πολλές φορές η αναδρομή, πληρώνουμε τίμημα: η αναδρομή είναι αρκετά ακριβή διαδικασία και ο κώδικας που παράγεται είναι πολύ πιο αργός από μια αντίστοιχη επαναληπτική υλοποίηση. Ενώ σε μία επαναληπτική υλοποίηση κάθε βήμα είναι απλά μερικές πράξεις μέσα σε έναν βρόχο, στην αναδρομική υλοποίηση κάθε βήμα αποτελεί και μία κλήση συνάρτησης, κάτι που φυσικά είναι πολύ πιο ακριβό. Όσοι είδατε την Ταινία 9.2 ίσως να συνειδητοποιείτε περισσότερο το απαιτούμενο κόστος. Αν παρατηρήσουμε, μάλιστα, καλύτερα τον υπολογισμό των αριθμών Fibonacci, θα δούμε ότι εμφανίζεται ακόμα ένας λόγος που κάνει την εκτέλεση της συνάρτησης ακόμα πιο αργή. Ας δούμε τα πρώτα βήματα του υπολογισμού του **Fibonacci(10)**:

Fibonacci(10)=Fibonacci(9)+Fibonacci(8)=
=Fibonacci(8)+Fibonacci(7)+Fibonacci(7)+Fibonacci(6)= ...

Το **Fibonacci(10)** απαιτεί τον υπολογισμό του **Fibonacci(9)** και **Fibonacci(8)**. Το **Fibonacci(9)** θα εκκινήσει τον υπολογισμό του **Fibonacci(8)** και **Fibonacci(7)**. Άρα το **Fibonacci(8)** θα υπολογιστεί δύο φορές. Αν πάμε χαμηλότερα στο δέντρο που δημιουργείται, θα δούμε ότι κάποιοι αριθμοί Fibonacci θα υπολογιστούν πολλές φορές από διαφορετικές κλήσεις της συνάρτησης. Η επίδραση που έχει αυτό στην ταχύτητα είναι καταστροφική. Δοκιμάστε αν θέλετε να συγκρίνετε τους χρόνους εκτέλεσης των συναρτήσεων στα Σχήματα 9.4 έως 9.7. Μη βάλετε μεγάλες τιμές στην αναδρομική συνάρτηση, δοκιμάστε να τις αυξήσετε σιγά σιγά. Γύρω στο 30 με 40 αρχίζει να αποκτάει το πείραμα μεγάλο ενδιαφέρον.

Λύσεις βέβαια υπάρχουν, όπως το να θυμόμαστε τους αριθμούς που έχουν ήδη υπολογιστεί και να κάνουμε μία διαφορετική υλοποίηση που να μην απαιτεί τον επανυπολογισμό τους. Αυτό όμως που θέλω να σας δείξω είναι ότι σε κάθε περίπτωση υπάρχει ένα τίμημα το οποίο, ανάλογα με την εφαρμογή και τις απαιτήσεις μας, πρέπει να αποφασίσουμε αν θέλουμε να το πληρώσουμε ή όχι.

Περισσότερο υλικό σχετικό με αναδρομή μπορείτε να διαβάσετε στο κεφάλαιο 5 του βιβλίου [1], στο κεφάλαιο 4 του βιβλίου [2], στο κεφάλαιο 15 του βιβλίου [3] και στο κεφάλαιο 8 του βιβλίου [4].

Ασκήσεις που μπορείτε να κάνετε μόνοι σας

- Υλοποιήστε αναδρομικά τον υπολογισμό του αθροίσματος:

$$1 + 2 + 3 + ... + v = (v + 1)/2$$

- Υλοποιήστε αναδρομικά το άθροισμα των στοιχείων μιας λίστας. Σκεφτείτε ότι το άθροισμα των στοιχείων μιας λίστας είναι το άθροισμα του πρώτου στοιχείου της λίστας και το άθροισμα των υπόλοιπων στοιχείων της λίστας.
- Φτιάξτε ένα διάγραμμα που να συγκρίνει τον χρόνο εκτέλεσης για τους τέσσερις κώδικες των Σχημάτων 9.4 έως 9.7.
- Προσπαθήστε να γράψετε μία αναδρομική συνάρτηση που να υπολογίζει τον ν-οστο όρο Fibonacci, χωρίς όμως να χρειάζεται να επαναλαμβάνει υπολογισμούς. Βάλτε και αυτήν τη συνάρτηση στο διάγραμμα της προηγούμενης άσκησης.
- Γράψτε έναν αναδρομικό αλγόριθμο για τον υπολογισμό του μέγιστου κοινού διαιρέτη δύο αριθμών.

Βιβλιογραφία

1. Allen B. Downey (2012). **Think Python**. Publisher: O'Reilly Media.
2. John Guttag (2015). **Υπολογισμοί και Προγραμματισμός Με Την Python**. Μετάφραση: Παναγιώτης Καναβός, Επιμέλεια: Γεώργιος Μανής, Εκδόσεις Κλειδάριθμος.
3. Brian Heinold (2012). **Introduction to Programming Using Python**. Publisher: Mount St. Mary's University, Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.
4. Cody Jackson (2011). **Learning to Program Using Python**. Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.

Κεφάλαιο 10: Αναζήτηση, ταξινόμηση

Η **αναζήτηση (search)** και η **ταξινόμηση (sort)** είναι δύο προβλήματα με ιδιαίτερο ενδιαφέρον στον χώρο της Πληροφορικής, τα οποία χρησιμοποιούνται σε πληθώρα διαφορετικών εφαρμογών. Ο λόγος που αφιερώνουμε ένα ολόκληρο κεφάλαιο (και όχι μικρό) σε αυτές τις δύο έννοιες είναι ότι, πέρα από τις εφαρμογές τους, η μελέτη τους αποτελεί ένα πολύ καλό μέσο εκπαιδευτικά, στη φάση που αυτήν τη στιγμή βρίσκεστε. Τους λόγους μπορεί κανείς εύκολα να τους δει:

- Η δυσκολία τους δεν είναι ούτε μικρή ούτε και πολύ μεγάλη. Είναι λίγο αυξημένη σε σχέση με ό,τι έχετε δει μέχρι τώρα, χωρίς από την άλλη να είναι κάτι πολύ δυσκολότερο.
- Για την ακρίβεια, η δυσκολία των υπαρχόντων αλγορίθμων είναι αυξανόμενη, ξεκινάει δηλαδή από απλούς αλγορίθμους, των οποίων η δυσκολία σταδιακά αυξάνεται, μέχρις ότου φτάσουμε στους αναδρομικούς αλγορίθμους, που παρουσιάζουν τη μεγαλύτερη ίσως δυσκολία
- Οι διαφορετικοί μεταξύ τους αλγόριθμοι που έχουν προταθεί και χρησιμοποιούνται μας δείχνουν πως ένα πρόβλημα μπορεί να αντιμετωπιστεί με πολλούς διαφορετικούς τρόπους.
- Τέλος, είναι μια πολύ καλή ευκαιρία να δούμε εισαγωγικά την έννοια της **πολυπλοκότητας (complexity)** ενός αλγορίθμου και να μπορέσουμε να συγκρίνουμε θεωρητικά αλγορίθμους μεταξύ τους με βάση τον αριθμό των βημάτων που θα κάνει ο καθένας για να λύσει ένα πρόβλημα.

Έχουν προταθεί πολλοί και διαφορετικοί αλγόριθμοι, τόσο για την ταξινόμηση όσο και για την αναζήτηση, για διαφορετικές παραλλαγές του προβλήμα-

τος. Εμείς εδώ θα ασχοληθούμε με τους περισσότερο σημαντικούς. Θα δούμε τη **σειριακή αναζήτηση (serial search)** και μία παραλλαγή της που μπορεί να εφαρμοστεί αν τα στοιχεία πάνω στα οποία γίνεται η αναζήτηση είναι ταξινομημένα. Θα δούμε και τη **δυαδική αναζήτηση (binary search)**, η οποία είναι ο γρηγορότερος τρόπος για να κάνει κανείς αναζήτηση, εφαρμόζεται, όμως, μόνο αν τα στοιχεία πάνω στα οποία γίνεται η αναζήτηση είναι ταξινομημένα.

Όσον αφορά την ταξινόμηση, θα δούμε την **ταξινόμηση με φυσαλίδα (bubble sort)**, την **ταξινόμηση με επιλογή (selection sort)**, την **ταξινόμηση δύο ήδη ταξινομημένων πινάκων με συγχώνευση (merge sort)**, την **ταξινόμηση ακέραιων αριθμών σε ένα ορισμένο εύρος με τη χρήση κάδων (bucket sort)** και τη **γρήγορη ταξινόμηση (quick sort)**, η οποία είναι πράγματι γρήγορη, γρηγορότερη από κάθε άλλη, και θα τη δούμε στην αναδρομική της μορφή.

Επίσης, ενδιαφέρον παρουσιάζουν η αναζήτηση και ταξινόμηση με τη χρήση δεντρικών δομών, αλλά η μελέτη τους είναι έξω από τους σκοπούς ενός εισαγωγικού στον προγραμματισμό βιβλίου. Μπορείτε να τους αναζητήσετε σε κάποιο βιβλίο δομών δεδομένων.

10.1 Αναζήτηση

10.1.1 Σειριακή αναζήτηση

Ο πιο απλός αλγόριθμος αναζήτησης είναι αυτός της **σειριακής αναζήτησης (serial search)**. Ας υποθέσουμε ότι έχουμε ένα διατεταγμένο σύνολο από αντικείμενα (σε ένα διατεταγμένο σύνολο η σειρά των στοιχείων του έχει σημασία) και, χωρίς βλάβη της γενικότητας, ας θεωρήσουμε ότι πρόκειται για ακέραιους αριθμούς. Την ίδια υπόθεση θα κάνουμε για όλους τους αλγορίθμους που θα εξετάσουμε στο κεφάλαιο αυτό.

Θεωρούμε ότι το σύνολο αυτό είναι μη ταξινομημένο και θέλουμε να ελέγξουμε αν κάποιος αριθμός είναι μέλος του ή όχι. Δυστυχώς, το γεγονός ότι τα στοιχεία του συνόλου δεν είναι ταξινομημένα δεν μας αφήνει κάποιο περιθώριο να κάνουμε κάτι περισσότερο έξυπνο από το να κοιτάξουμε ένα ένα όλα τα στοιχεία του συνόλου. Αν είμαστε τυχεροί και το στοιχείο βρίσκεται στο διατεταγμένο σύνολο και μάλιστα στις πρώτες θέσεις, εύκολα και γρήγορα θα δώσουμε την απάντηση στο ερώτημά μας. Αν βρίσκεται στις τελευταίες θέσεις, τότε θα μπορέσουμε πάλι να απαντήσουμε, λιγότερο γρήγορα αυτήν τη φορά. Αν ο αριθμός δεν υπάρχει καθόλου στο σύνολο, τότε αναγκαστικά πρέπει να φτάσουμε στο τέλος του και, αφού ελέγξουμε όλα τα στοιχεία του, τότε και μόνο τότε, θα είμαστε σε θέση να γνωρίζουμε ότι ο αριθμός δεν ανήκει

στο σύνολο.

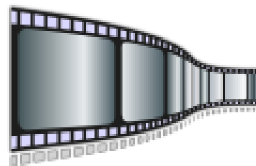
Αν και η Python έχει σύνολα, θα χρησιμοποιήσουμε μονοδιάστατες λίστες ή, που είναι το ίδιο στη συγκεκριμένη περίπτωση, μονοδιάστατους πίνακες, για να τοποθετήσουμε εκεί και να χειριστούμε κατάλληλα τα στοιχεία του συνόλου. Ο κώδικας που αντιστοιχεί στον αλγόριθμο που περιγράφηκε παραπάνω φαίνεται στο Σχήμα 10.1.

```
1 def serial_search(A,item):
2     for x in A:
3         if x==item:
4             return 'found'
5     return 'not found'
```

Σχήμα 10.1: Σειριακή αναζήτηση.

Στο σχήμα αυτό μπορείτε να παρατηρήσετε ότι χρησιμοποιείται ένας βρόχος ο οποίος ελέγχει ένα ένα τα στοιχεία του πίνακα και, αν κάποιο στοιχείο ισούται με τον προς αναζήτηση αριθμό, τότε επιστρέφει την τιμή **found**. Αν το πρόγραμμα βγει από τον βρόχο, σημαίνει ότι το στοιχείο δεν βρέθηκε, οπότε η συνάρτηση επιστρέφει την τιμή **not found**.

Για ένα παράδειγμα εκτέλεσης κάντε κλικ στην Ταινία 10.1.



Ταινία 10.1: Παράδειγμα εκτέλεσης του αλγορίθμου σειριακής αναζήτησης.

http://repfiles.kallipos.gr/file/22393

Παρατηρήστε ότι, για να αποφασίσουμε αν ένας δεδομένος αριθμός ανήκει ή όχι σε ένα μη ταξινομημένο σύνολο αριθμών, πρέπει στη χειρότερη περίπτωση να κάνουμε τόσα βήματα όσα και το μέγεθος του πίνακα. Στο συγκεκριμένο βιβλίο δεν θα ασχοληθούμε σε βάθος με την έννοια της πολυπλοκότητας των αλγορίθμων. Ας αρκεστούμε στο να συμβολίσουμε την πολυπλοκότητα ενός αλγορίθμου ο οποίος κάνει το πολύ **N** βήματα για να απαντήσει σε ένα πρόβλημα σαν **O(N)**.

Εδώ, όπως και σε κάθε πρόβλημα αναζήτησης ή ταξινόμησης, κάθε βήμα είναι και μία σύγκριση. Σε κάποιον άλλο αλγόριθμο το βήμα μπορεί να είναι κάτι

άλλο. Έτσι λοιπόν, ο αλγόριθμος της σειριακής αναζήτησης έχει πολυπλοκότητα O(N). Στη συνέχεια, θα τον συγκρίνουμε με βάση την πολυπλοκότητα με τους υπόλοιπους αλγορίθμους αναζήτησης.

Μία παραλλαγή του προβλήματος της αναζήτησης είναι η περίπτωση που ο πίνακας μέσα στον οποίο αναζητούμε το στοιχείο είναι ταξινομημένος. Φυσικά ο αλγόριθμος που συζητήθηκε παραπάνω μπορεί να χρησιμοποιηθεί και εδώ. Όμως, στην περίπτωση αυτήν μπορούμε να κάνουμε κάτι καλύτερο.

Αν το στοιχείο υπάρχει ήδη στον πίνακα, ο αλγόριθμός θα κάνει τα ίδια ακριβώς βήματα με τον προηγούμενό του. Στην περίπτωση όμως που το στοιχείο δεν υπάρχει στον πίνακα, τότε χρειάζεται να συνεχίσουμε την αναζήτηση μόνο έως το σημείο του πίνακα στο οποίο βρίσκονται στοιχεία μικρότερα (αν ο πίνακας έχει αύξουσα διάταξη) ή μεγαλύτερα (στην περίπτωση που είναι ταξινομημένος σε φθίνουσα διάταξη) από το προς αναζήτηση στοιχείο.

Πάλι βέβαια, ο αλγόριθμος για να απαντήσει αν ένας αριθμός ανήκει ή όχι στον πίνακα πρέπει στη χειρότερη περίπτωση να κάνει **N** βήματα. Άρα η πολυπλοκότητα του αλγορίθμου εξακολουθεί να είναι **O(N)**. Ο μέσος χρόνος όμως κατά τον οποίο ο αλγόριθμος θα αποκριθεί είναι μικρότερος και μάλιστα ο μισός του χειρότερου. Ο κώδικας ενός τέτοιου αλγορίθμου φαίνεται στο Σχήμα 10.2. Η διαφοροποίηση με τον προηγούμενό αλγόριθμο βρίσκεται στη γραμμή 5, όπου, αφού ένα στοιχείο διαπιστωθεί ότι δεν ανήκει στον πίνακα, τότε ελέγχεται μήπως το προς αναζήτηση στοιχείο είναι μικρότερο από το στοιχείο του πίνακα με το οποίο το συγκρίνουμε. Αν ισχύει κάτι τέτοιο, μπορούμε να συμπεράνουμε ότι το στοιχείο αυτό δεν θα υπάρχει ούτε παρακάτω, μια και όλα τα στοιχεία του πίνακα που βρίσκονται μετά από αυτό είναι ακόμα μεγαλύτερα.

```
1  def serial_search_2(A,item):
2      for x in A:
3          if x==item:
4              return 'found'
5          if x>item:
6              return 'not found'
7      return 'not found'
```

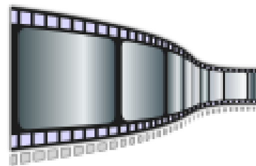
Σχήμα 10.2: Σειριακή αναζήτηση σε ταξινομημένο πίνακα.

Σταματάμε, λοιπόν, εκεί την αναζήτηση και επιστρέφουμε **not found**. Πότε όμως η εκτέλεση φτάνει στη γραμμή 7 και γιατί εκεί επιστρέφεται **not found**; Για να συμβεί κάτι τέτοιο πρέπει να μην έχει εκτελεστεί κανένα από τα δύο προηγούμενα **return** (του **if** και του **elif**) αλλά και να έχουν τελειώσει όλες οι

επαναλήψεις του βρόχου. Αυτό θα συμβεί όταν το προς αναζήτηση στοιχείο είναι μεγαλύτερο από κάθε στοιχείο του πίνακα.

Ο αλγόριθμος που περιγράφηκε εκμεταλλεύεται πράγματι την ταξινόμηση του πίνακα, γι' αυτό και παρουσιάζει μικρότερο μέσο χρόνο απόκρισης από τον προηγούμενό αλγόριθμο. Παρόλο που τα καταφέρνει πιο καλά από τον σειριακό αλγόριθμο, δεν τα καταφέρνει τόσο καλά όσο θα μπορούσε. Η δυαδική αναζήτηση που θα παρουσιαστεί στην επόμενη υποενότητα είναι πολύ πιο γρήγορη.

Ένα παράδειγμα εκτέλεσης μπορείτε να δείτε στην Ταινία 10.2.



Ταινία 10.2: Παράδειγμα εκτέλεσης του αλγορίθμου της σειριακής αναζήτησης σε ταξινομημένο πίνακα.
`http://repfiles.kallipos.gr/file/22394`

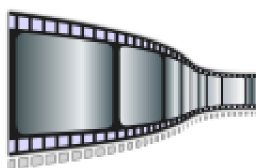
10.1.2 Δυαδική αναζήτηση

Η **δυαδική αναζήτηση (binary search)** είναι ο γρηγορότερος αλγόριθμος αναζήτησης. Προϋποθέτει ότι ο πίνακας στον οποίο θα αναζητήσουμε το στοιχείο είναι ταξινομημένος. Η ιδέα είναι η ακόλουθη:

- Επιλέγουμε το μεσαίο στοιχείο του πίνακα. Αν ο πίνακας έχει άρτιο αριθμό στοιχείων, επιλέγουμε ένα εκ των δύο στοιχείων που βρίσκονται πλησιέστερα στη μέση του πίνακα.
- Αν ο προς αναζήτηση αριθμός είναι ίσος με το στοιχείο που επιλέξαμε, τότε βρήκαμε το στοιχείο που ζητούσαμε και σταματάμε την αναζήτηση.
- Αν ο προς αναζήτηση αριθμός είναι μικρότερος από το στοιχείο που επιλέξαμε, τότε συνεχίζουμε την αναζήτηση στο αριστερό μέρος του πίνακα, αφού όλα τα στοιχεία στο δεξί μέρος είναι μεγαλύτερα από τον προς αναζήτηση αριθμό.
- Αν ο προς αναζήτηση αριθμός είναι μεγαλύτερος από το στοιχείο που επιλέξαμε, τότε συνεχίζουμε την αναζήτηση στο δεξί μέρος του πίνακα, αφού όλα τα στοιχεία στο αριστερό μέρος είναι μικρότερα από τον προς αναζήτηση αριθμό.

- Συνεχίζουμε την ίδια διαδικασία μέχρις ότου βρεθεί το στοιχείο που ψάχνουμε ή μέχρις ότου, συνεχίζοντας την παραπάνω διαδικασία, φτάσουμε σε ένα σημείο που το μέρος του πίνακα στο οποίο πρέπει να ψάξουμε να βρούμε τον αριθμό δεν έχει κανένα στοιχείο. Στην τελευταία περίπτωση συμπεραίνουμε ότι ο αριθμός που αναζητούμε δεν βρίσκεται στον πίνακα.

Ας δούμε στην Ταινία 10.3 πώς λειτουργεί ο αλγόριθμος αυτός και ας συνειδητοποιήσουμε για ποιο λόγο είναι ο καλύτερος αλγόριθμος αναζήτησης.



Ταινία 10.3: Παράδειγμα εκτέλεσης του αλγορίθμου δυαδικής αναζήτησης.

http://repfiles.kallipos.gr/file/22395

```
1  def binary_search(A,item):
2      first = 0
3      last = len(A)-1
4      found = False
5      while first<=last and not found:
6          mid = (first + last)//2
7          if A[mid] == item:
8              return(mid)
9          else:
10             if item < A[mid]:
11                 last = mid-1
12             else:
13                 first = mid+1
14     return mid
```

Σχήμα 10.3: Αλγόριθμος δυαδικής αναζήτησης.

Ας δούμε και τον κώδικα που υλοποιεί τον αλγόριθμο και περιγράφεται στα παραπάνω βήματα. Φαίνεται στο Σχήμα 10.3, ακολουθεί την περιγραφή που

κάναμε παραπάνω, αλλά δώστε λίγο περισσότερη προσοχή, γιατί δεν είναι πολύ απλός.

Εδώ αποκτά ιδιαίτερο ενδιαφέρον να δούμε πόσα βήματα χρειάζεται ο αλγόριθμος της δυαδικής αναζήτησης για να φτάσει στην απάντηση. Σε κάθε βήμα του γίνεται και μία σύγκριση. Με τη σύγκριση αυτήν αποκλείουμε τα μισά από τα στοιχεία του πίνακα. Με την επόμενη σύγκριση έχουμε αποκλείσει τα μισά από όσα είχαν μείνει. Η διαδικασία αυτή συνεχίζεται μέχρις ότου βρούμε το στοιχείο ή μέχρις ότου αποφασίσουμε ότι αυτό δεν υπάρχει στον πίνακα. Τι σας θυμίζει από τα μαθηματικά αυτό; Αν σε κάθε βήμα πετάω τα μισά στοιχεία έξω, πόσα βήματα χρειάζεται να κάνω για να τα πετάξω όλα;

Μην σας κρατάω σε αγωνία, αν δεν το έχετε ήδη σκεφτεί. Η απάντηση είναι ο δυαδικός λογάριθμος. Αν έχουμε **N** στοιχεία και σε κάθε βήμα μάς απομένουν τα μισά, τότε σε $\log_2(N)$ βήματα τα στοιχεία αυτά θα έχουν τελειώσει. Λέμε ότι ο αλγόριθμος αυτός έχει λογαριθμική πολυπλοκότητα και τον συμβολίζουμε με **O(logN)**.

10.2 Ταξινόμηση

10.2.1 Ταξινόμηση φυσαλίδας

Η **ταξινόμηση φυσαλίδας (bubble sort)** θυμίζει λίγο τη διαδικασία με την οποία, όταν έχουμε μέσα σε μία στήλη ελαφριές και βαριές φυσαλίδες, οι ελαφριές σιγά σιγά θα ανέβουν προς τα πάνω και οι βαριές θα κινηθούν προς τα κάτω. Σε κάθε βήμα μία φυσαλίδα που έχει μία βαρύτερη από πάνω της θα αλλάξει θέση με αυτήν. Όταν πια καμιά φυσαλίδα δεν έχει ακριβώς από πάνω της μία βαρύτερη, τότε έχει επέλθει η ισορροπία, ή στην περίπτωση της ταξινόμησης αριθμών, η ταξινόμηση έχει ολοκληρωθεί.

Έχουμε διάφορους τρόπους για να υλοποιήσουμε με κώδικα την παραπάνω σκέψη. Θα δούμε δύο εκδοχές, την πιο απλή και μια περισσότερο έξυπνη. Η πιο απλή φαίνεται στο Σχήμα 10.4, όπου έχουμε έναν πίνακα **A** με ακέραιους αριθμούς (χωρίς βλάβη της γενικότητας) τους οποίους η συνάρτηση **bubble_sort** θέλει να ταξινομήσει και να επιστρέψει ως αποτέλεσμα. Ο κώδικας αποτελείται από δύο βρόχους.

```
1 def bubble_sort(A):
2     N = len(A)
3     for i in range(N-1):
4         for j in range(N-i-1):
5             if A[j]>A[j+1]:
6                 A[j],A[j+1] = A[j+1],A[j]
```

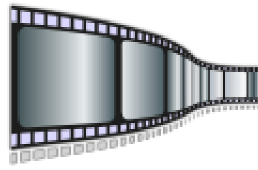
Σχήμα 10.4: Ταξινόμηση φυσαλίδας.

Ας ξεκινήσουμε από τον εσωτερικό και ας ξεχάσουμε αρχικά ότι φτάνει έως το **N-i-1** και ας θεωρήσουμε ότι φτάνει μέχρι το **N-1**. Αν εκτελέσουμε αυτόν τον βρόχο, θα συμβούν **N-1** συγκρίσεις και το πρώτο στοιχείο θα συγκριθεί με το δεύτερο, το δεύτερο με το τρίτο και το προτελευταίο με το τελευταίο. Για όσες από αυτές τις **N-1** συγκρίσεις βρεθεί μεγαλύτερος αριθμός να είναι κάτω από μικρότερο, θα γίνει εναλλαγή. Με το πέρας του βρόχου ο πίνακας θα είναι περισσότερο ταξινομημένος, όχι όμως και ταξινομημένος. Όμως, το μεγαλύτερο στοιχείο θα έχει τοποθετηθεί στην τελευταία θέση του πίνακα, εκεί που είναι και η θέση του δηλαδή. Σκεφτείτε λίγο γιατί. Όπου και να βρίσκεται στον πίνακα το μεγαλύτερο στοιχείο, μόλις για πρώτη φορά συγκριθεί θα μετακινηθεί μία θέση κάτω. Μα και στην επόμενη σύγκριση πάλι θα συμμετέχει, αφού έχει πάει από τη θέση **i** στη θέση **i+1**.

Η μετακίνηση του μεγαλύτερου στοιχείου θα σταματήσει μόνο στην τελευταία σύγκριση. Την επόμενη φορά, λοιπόν, που θα εκτελεστεί ο εσωτερικός βρόχος, δεν χρειάζεται να ελεγχθεί το τελευταίο στοιχείο, αφού αυτό είναι στη θέση του. Θα χρειαστεί να πάμε τις συγκρίσεις μέχρι τη θέση **N-2**. Την τρίτη φορά μέχρι τη θέση **N-3** κ.ο.κ. Γι' αυτό και ο εσωτερικός βρόχος δεν πηγαίνει μέχρι **N-1** στον κώδικα του Σχήματος 10.4 αλλά μέχρι **N-i-1**.

Πάμε τώρα στον εξωτερικό βρόχο. Πόσες φορές πρέπει να εκτελεστεί ο εσωτερικός βρόχος προκειμένου να ταξινομηθεί ο πίνακας, αφού τουλάχιστον ένα στοιχείο πάει στη θέση του σε κάθε επανάληψη του εσωτερικού βρόχου; Θα λέγαμε **N** φορές, αφού έχουμε **N** στοιχεία. Μα γιατί λέει ο κώδικας **N-1**; Για σκεφτείτε λίγο, είναι δυνατόν τα **N-1** στοιχεία να είναι στη θέση τους και ένα να βρίσκεται σε λάθος; Όχι. Μάλλον ο κώδικας έχει δίκιο.

Ένα παράδειγμα εκτέλεσης του αλγορίθμου της ταξινόμησης φυσαλίδας μπορείτε να δείτε στην Ταινία 10.4.



Ταινία 10.4: Παράδειγμα εκτέλεσης του αλγορίθμου ταξινόμησης φυσαλίδας.

http://repfiles.kallipos.gr/file/22396

```
1 def bubble_sort_2(A):
2     N = len(A)
3     for i in range(N-1):
4         changed = False
5         for j in range(N-i-1):
6             if A[j]>A[j+1]:
7                 A[j],A[j+1] = A[j+1],A[j]
8                 changed = True
9         if not changed: return
```

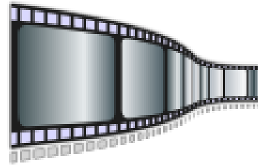
Σχήμα 10.5: Βελτιωμένος αλγόριθμος ταξινόμησης φυσαλίδας.

Ας δούμε κάτι περισσότερο έξυπνο: Ας αναρωτηθούμε αν όλα τα βήματα της ταξινόμησης είναι πάντοτε απαραίτητα. Έστω ότι έχουμε να ταξινομήσουμε τον πίνακα:

A = [2 4 7 6 9]

Με την πρώτη επανάληψη του εσωτερικού βρόχου ο πίνακας θα έχει ταξινομηθεί και όλες οι υπόλοιπες επαναλήψεις θα είναι περιττές. Ας κάνουμε κάτι γι' αυτό. Στο Σχήμα 10.5 ελέγχουμε αν σε κάποιο πέρασμα δεν έχει γίνει καμία εναλλαγή στοιχείων. Αν πράγματι δεν έχει γίνει καμία εναλλαγή στοιχείων, αυτό σημαίνει ότι ο πίνακας έχει ήδη ταξινομηθεί. Χρησιμοποιείται για τον σκοπό αυτόν η μεταβλητή **changed**, η οποία ακριβώς πριν την εκτέλεση του εσωτερικού βρόχου αρχικοποιείται σε **False**. Αν κατά τη διάρκεια της εκτέλεσης του εσωτερικού βρόχου γίνει μία εναλλαγή, τότε γίνεται **True**. Στο τέλος της εκτέλεσης του εσωτερικού βρόχου ελέγχουμε αν η τιμή της **changed** είναι **False**. Αν πράγματι είναι **False**, αυτό σημαίνει ότι δεν έγινε καμία εναλλαγή στο πέρασμα αυτό, ο πίνακας έχει ήδη ταξινομηθεί, οπότε και η εκτέλεση δεν χρειάζεται να συνεχιστεί. Ο αλγόριθμος τερματίζεται επιστρέφοντας τον ταξινομημένο πίνακα ως αποτέλεσμα.

Δείτε ένα παράδειγμα εκτέλεσης του αλγορίθμου αυτού στην Ταινία 10.5.



Ταινία 10.5: Παράδειγμα εκτέλεσης του βελτιωμένου αλγορίθμου φυσαλίδας.

http://repfiles.kallipos.gr/file/22397

Ο αλγόριθμος αυτός χρειάζεται, στη χειρότερη περίπτωση, **N-1** περάσματα για τον εξωτερικό βρόχο και για καθένα από τα περάσματα αυτά **N-1-i** συγκρίσεις στον εσωτερικό βρόχο. Αφού μιλάμε για πολυπλοκότητα, δεν χρειάζεται να είμαστε τόσο ακριβείς, αρκεί να πούμε ότι χρειαζόμαστε περίπου **N** περάσματα λόγω του εξωτερικού βρόχου και **N** συγκρίσεις μέσα στον εσωτερικό βρόχο, άρα συνολικά απαιτούνται **NxN** συγκρίσεις. Ο αλγόριθμος λέγεται ότι έχει τετραγωνική πολυπλοκότητα και συμβολίζεται με **O(N^2)**.

10.2.2 Ταξινόμηση με επιλογή

Η **ταξινόμηση με επιλογή (selection sort)** βασίζεται σε μία διαφορετική ιδέα. Στο πρώτο βήμα αναζητούμε το στοιχείο εκείνο του πίνακα με τη μικρότερη τιμή, αυτό δηλαδή που πρέπει να τοποθετηθεί πρώτο στον ταξινομημένο πίνακα. Όταν το βρούμε, το τοποθετούμε στη θέση του, δηλαδή πρώτο στον πίνακα. Το στοιχείο που βρισκόταν μέχρι τώρα πρώτο στον πίνακα το τοποθετούμε στη θέση που βρισκόταν μέχρι τώρα αυτό που εντοπίσαμε ως μικρότερο στοιχείο. Πρακτικά εναλλάξαμε το στοιχείο που βρισκόταν πρώτο στον πίνακα με το μικρότερο στον πίνακα στοιχείο.

Έτσι, με την ολοκλήρωση του πρώτου βήματος το μικρότερο στοιχείο βρέθηκε στην πρώτη θέση, ενώ ο υπόλοιπος πίνακας είναι φυσικά αταξινόμητος. Στο επόμενο βήμα βρίσκουμε το μικρότερο στοιχείο του πίνακα που έχει μείνει αταξινόμητος και το τοποθετούμε με την ίδια διαδικασία στην πρώτη θέση του αταξινόμητου πίνακα, δηλαδή στη δεύτερη θέση ολόκληρου του πίνακα. Παρατηρούμε ότι με την ολοκλήρωση του δεύτερου βήματος του αλγορίθμου τα δύο μικρότερα στοιχεία βρίσκονται στη θέση τους. Συνεχίζουμε με την ίδια λογική έως ότου όλα τα στοιχεία βρεθούν στη θέση τους.

Δείτε τον κώδικα στο Σχήμα 10.6.

```
1 def selection_sort(A):
2     for i in range(len(A)-1):
3         x = A.index(min(A[i:]))
4         A[i],A[x] = A[x],A[i]
```

Σχήμα 10.6: Αλγόριθμος ταξινόμησης με επιλογή.

Ας μετρήσουμε τώρα συγκρίσεις, όπως κάναμε σε όλες τις προηγούμενες περιπτώσεις. Η επιλογή ενός στοιχείου ως μικρότερου απαιτεί το να διατρέξουμε όλο τον μέχρι στιγμής μη ταξινομημένο πίνακα. Αυτά είναι περίπου **N** βήματα (για την ακρίβεια λιγότερα και εξαρτάται από το πόσα στοιχεία έχουν ήδη τοποθετηθεί στη θέση τους, αλλά δεν πειράζει, μας αρκεί να το εκφράσουμε στο περίπου, ως συνάρτηση του **N**). Οι επαναλήψεις του εξωτερικού βρόχου είναι πάλι περίπου **N**, αφού σε κάθε βήμα βάζουμε στη θέση τους ένα στοιχείο. Συμπεραίνουμε, λοιπόν, ότι έχουμε πάλι έναν τετραγωνικό αλγόριθμο, δηλαδή πολυπλοκότητα **O(N^2)**.

10.2.3 Ταξινόμηση δύο ταξινομημένων πινάκων με συγχώνευση

Θα εξετάσουμε τώρα ένα λίγο διαφορετικό πρόβλημα: Έχουμε δύο πίνακες οι οποίοι είναι ήδη ταξινομημένοι. Θέλουμε να τους συνενώσουμε σε έναν πίνακα ο οποίος φυσικά θα είναι και αυτός ταξινομημένος.

Η εύκολη λύση θα ήταν να τοποθετήσουμε τα στοιχεία των δύο πινάκων σε έναν και να τα ταξινομήσουμε χρησιμοποιώντας έναν από τους αλγορίθμους που έχουμε ήδη αναφέρει. Αυτό θα λειτουργούσε και θα έδινε αποτέλεσμα σε χρόνο **O(N^2)**, όπως είδαμε.

Αυτό που δεν εκμεταλλευτήκαμε είναι ότι οι δύο πίνακες είναι ήδη ταξινομημένοι. Η ιδιότητα αυτή μας επιτρέπει να κατασκευάσουμε έναν αλγόριθμο ο οποίος λύνει το πρόβλημά μας σε χρόνο **O(N)**. Η διαφορά στον χρόνο εκτέλεσης είναι τεράστια.

Ο αλγόριθμος που θα εξετάσουμε φαίνεται στο Σχήμα 10.7. Χρησιμοποιεί δύο δείκτες, ο ένας (το **i**) διατρέχει τον πρώτο πίνακα, τον **A**. Ο δεύτερος (το **j**) διατρέχει τον δεύτερο πίνακα, τον **B**. Αρχικά τα **i** και **j** δείχνουν τα πρώτα στοιχεία των **A** και **B**. Συγκρίνουμε τα στοιχεία αυτά και όποιο από τα δύο είναι μικρότερο το τοποθετούμε στον **C** και προχωράμε τον αντίστοιχο δείκτη (**i** ή **j**) κατά 1, δείχνοντας έτσι ότι το στοιχείο αυτό έχει ήδη χρησιμοποιηθεί και ταξινομηθεί. Στο επόμενο βήμα, ουσιαστικά επαναλαμβάνουμε το πρώτο βήμα εξετάζοντας με τον ίδιο τρόπο τα στοιχεία που δείχνουν οι δείκτες **i** και **j**. Το στοιχείο που κάθε φορά επιλέγεται τοποθετείται κατάλληλα στον **C**. Οι

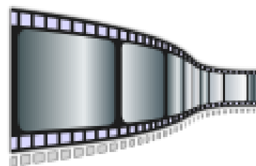
επαναλήψεις τελειώνουν όταν όλα τα στοιχεία ενός εκ των **A** ή **B** εξαντληθούν. Στην περίπτωση αυτήν, τοποθετούμε τα εναπομείναντα στοιχεία του πίνακα, του οποίου τα στοιχεία δεν εξαντλήθηκαν, στο τέλος του **C**.

```
1  def merge_sort(A,B):
2      i=0;
3      j=0;
4      C=[]
5      while i<len(A) and j<len(B):
6          if A[i]<B[j]:
7              C.append(A[i])
8              i=i+1
9          else:
10             C.append(B[j])
11             j=j+1
12     if i==len(A):
13         while j<len(B):
14             C.append(B[j])
15             j=j+1
16     else:
17         while i<len(A):
18             i=i+1
19     return C
```

Σχήμα 10.7: Αλγόριθμος ταξινόμησης με συγχώνευση.

Παρατηρήστε ότι τα βήματα που κάναμε ήταν όσα και ο συνολικός αριθμός των στοιχείων. Άρα η πολυπλοκότητα του αλγορίθμου είναι **O(N)**. Εκμεταλλευόμενοι ότι οι δύο πίνακες ήταν αρχικά ταξινομημένοι καταφέραμε να βρούμε το ζητούμενο αποτέλεσμα πολύ γρήγορα.

Ένα παράδειγμα εκτέλεσης μπορεί κανείς να παρακολουθήσει στην Ταινία 10.6.



Ταινία 10.6: Παράδειγμα εκτέλεσης του αλγορίθμου ταξινόμησης με συγχώνευση.

http://repfiles.kallipos.gr/file/22398

10.2.4 Ταξινόμηση με τη χρήση κάδων

Θα δούμε ακόμα μία διαφορετική περίπτωση ταξινόμησης, στην οποία τα δεδομένα μας πάλι διαφέρουν και πρέπει να βρούμε κάποιο τρόπο ώστε να εκμεταλλευτούμε πλήρως αυτά που μας δίνονται και να πετύχουμε τον αλγόριθμο με τη μικρότερη δυνατή πολυπλοκότητα. Τώρα μας δίνεται ένας μη ταξινομημένος πίνακας με αριθμήσιμα στοιχεία για τα οποία ορίζεται κάποια σχέση διάταξης και τα οποία είναι κάτω και άνω φραγμένα. Για να το απλοποιήσουμε λίγο και χωρίς βλάβη της γενικότητας, ας θεωρήσουμε ότι μιλάμε για έναν πίνακα με θετικούς ακέραιους αριθμούς. Το γεγονός ότι μπορούμε εξαρχής να γνωρίζουμε ότι οι αριθμοί αυτοί πηγαίνουν από το μηδέν έως κάποιον μέγιστο αριθμό, μας δίνει τη δυνατότητα να κάνουμε κάτι έξυπνο και πολύ γρήγορο.

Ας το δούμε με ένα παράδειγμα: Ας υποθέσουμε ότι έχουμε να ταξινομήσουμε ακέραιους αριθμούς από το 0 μέχρι το 100. Μπορούμε να θεωρήσουμε 101 κάδους, έναν για κάθε αριθμό από το 0 μέχρι το 100. Κάνοντας ένα πέρασμα του πίνακα, τοποθετούμε τον αριθμό στον κατάλληλο κάδο. Στη συνέχεια, περνώντας έναν έναν τους κάδους βάζουμε τους αριθμούς πίσω στον πίνακα. Επειδή όμως θα ξεκινήσουμε από τον κάδο με τα 0 και θα προχωρήσουμε κάδο κάδο μέχρι τον κάδο με τα 100, ο νέος πίνακας που θα προκύψει θα είναι ταξινομημένος. Μάλιστα χρειαστήκαμε **2*N** περάσματα, ένα για να βάλουμε τους αριθμούς στους κάδους και ένα για να τους βάλουμε πίσω στον πίνακα. Πολύ μικρή πολυπλοκότητα, μόνο **O(N)**, πολύ γρήγορος αλγόριθμός.

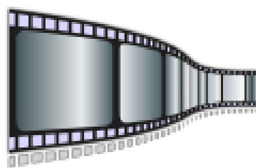
```
1 def bucket_sort(A):
2     bucket=[0 for x in range(max(A)+1)]
3     C=[]
4     for x in A:
5         bucket[x]=bucket[x]+1
6     for i in range(max(A)):
7         for j in range(bucket[i]):
8             C.append(i)
9     return C
```

Σχήμα 10.8: Αλγόριθμος ταξινόμησης με χρήση κάδων.

Ο κώδικας που τον υλοποιεί φαίνεται στο Σχήμα 10.8. Αρχικοποιούμε έναν πίνακα όσο μεγάλο χρειαζόμαστε (όσο το μεγαλύτερο στοιχείο του πίνακα συν μία θέση για το μηδέν). Στη συνέχεια επισκεπτόμαστε ένα ένα τα στοιχεία του πίνακα και τα βάζουμε στον αντίστοιχο κάδο. Ο κάθε κάδος αντιστοιχεί σε μία

θέση του πίνακα **bucket**, όπου υπάρχει ένας μετρητής που κρατάει την πληροφορία πόσα στοιχεία υπάρχουν αυτήν τη στιγμή μέσα στον κάδο. Ο διπλός βρόχος που ακολουθεί δεν πρέπει να ξεγελάει. Μπορεί να θυμίζει πολυπλοκότητα **O(N^2)** αλλά δεν είναι έτσι. Παρατηρήστε ότι το **C.append(i)** θα εκτελεστεί **N** φορές. Ο εξωτερικός βρόχος επισκέπτεται έναν έναν τους κάδους, ενώ ο εσωτερικός τοποθετεί ένα ένα τα στοιχεία του κάδου πίσω στον πίνακα. **N** στοιχεία θα τοποθετηθούν, **O(N)** η πολυπλοκότητα.

Δείτε στην Ταινία 10.7 ένα παράδειγμα εκτέλεσης.



Ταινία 10.7: Παράδειγμα εκτέλεσης αλγορίθμου ταξινόμησης με τη χρήση κάδων.

http://repfiles.kallipos.gr/file/22399

10.2.5 Γρήγορη ταξινόμηση

Η γρήγορη ταξινόμηση δεν είναι τυχαίο που ονομάστηκε έτσι. Εφαρμόζεται σε οποιονδήποτε πίνακα, δεν χρειάζεται δηλαδή να είναι ταξινομημένος ή να ισχύει οποιαδήποτε ειδική συνθήκη, όπως στους αλγορίθμους με συγχώνευση και με τη χρήση κάδων.

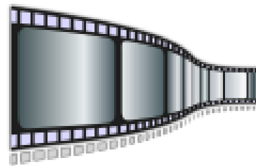
Την ταχύτητά του την οφείλει στη χαμηλή πολυπλοκότητα. Ενώ όλοι οι άλλοι γενικοί αλγόριθμοι ταξινόμησης είχαν πολυπλοκότητα **O(N^2)**, αυτός καταφέρνει να ταξινομήσει έναν πίνακα σε **O(NlogN)**. Και το καταφέρνει χρησιμοποιώντας αναδρομή, κάτι που έχουμε εξετάσει σε προηγούμενο κεφάλαιο.

Βασίζεται στην τεχνική διαίρει και βασίλευε. Σε κάθε βήμα επιλέγει ένα τυχαίο σημείο και με βάση αυτό χωρίζει τον πίνακα σε δύο υποπίκακες. Ο πρώτος έχει όλα τα στοιχεία που είναι μικρότερα από το τυχαίο σημείο που επιλέχθηκε και ο δεύτερος όλα τα μεγαλύτερα. Στη συνέχεια, αναδρομικά, ταξινομεί τους δύο υποπίνακες. Έτσι, σε κάθε βήμα σπάει έναν πίνακα σε δύο υποπίνακες με μισό περίπου μέγεθος ο καθένας, δημιουργεί δηλαδή δύο προβλήματα, το καθένα με μέγεθος το μισό του αρχικού. Όπως και στη δυαδική αναζήτηση που είχαμε δει παραπάνω στο κεφάλαιο αυτό, ο αριθμός των φορών που μπορεί να διασπαστεί ο πίνακας μέχρι να γίνει ένας πίνακας με ένα στοιχείο είναι **logN**. Γι' αυτόν τον λόγο η πολυπλοκότητα της γρήγορης ταξινόμησης είναι **O(NlogN)**.

Ο κώδικας που υλοποιεί τον αλγόριθμο φαίνεται στο Σχήμα 10.9 και ένα παράδειγμα εκτέλεσης μπορείτε να δείτε στην Ταινία 10.8.

```
1  def quick_sort(A):
2      less = []
3      equal = []
4      greater = []
5      if len(A) > 1:
6          pivot = A[0]
7          for x in A:
8              if x < pivot:
9                  less.append(x)
10             if x == pivot:
11                 equal.append(x)
12             if x > pivot:
13                 greater.append(x)
14         return quick_sort(less)+quick_sort(equal)
15                                +quick_sort(greater)
16     else: return A
```

Σχήμα 10.9: Αλγόριθμος γρήγορης ταξινόμησης.



Ταινία 10.8: Παράδειγμα εκτέλεσης αλγορίθμου γρήγορης ταξινόμησης.
http://repfiles.kallipos.gr/file/22400

Περισσότερο υλικό σχετικό με αναζήτηση και ταξινόμηση μπορείτε να διαβάσετε στο κεφάλαιο 11 του βιβλίου [1], στο κεφάλαιο 10 του βιβλίου [2], στο κεφάλαιο 7 του βιβλίου [3] και στο κεφάλαιο 12 του βιβλίου [4].

Ασκήσεις που μπορείτε να κάνετε μόνοι σας

- Υλοποιήστε τον αλγόριθμο ταξινόμησης με εισαγωγή. Σύμφωνα με αυτόν, ξεκινάμε με έναν άδειο πίνακα. Εισάγουμε το πρώτο

στοιχείο στην πρώτη του θέση. Το δεύτερο στοιχείο το εισάγουμε στην πρώτη θέση, αν είναι μικρότερο από το πρώτο, και στη δεύτερη, αν είναι μεγαλύτερο. Συνεχίζουμε έτσι, ώστε σε κάθε βήμα να εισάγουμε και ένα νέο στοιχείο έτσι ώστε ο πίνακας να είναι πάντοτε ταξινομημένος. Σε κάθε βήμα, δηλαδή, βρίσκουμε τη θέση στην οποία πρέπει να μπει το καινούργιο στοιχείο, μεταφέρουμε όλα τα μεγαλύτερά του στοιχεία μία θέση δεξιότερα και τοποθετούμε το νέο στοιχείο στην κενή θέση που δημιουργήσαμε. Όταν εισάγουμε και το τελευταίο στοιχείο, ο αλγόριθμος τερματίζεται.

- Γράψτε ένα πρόγραμμα που ταξινομεί έναν πίνακα δύο διαστάσεων λεξικογραφικά. Αυτό σημαίνει ότι ταξινομεί τις γραμμές του με βάση το πρώτο στοιχείο κάθε γραμμής. Αν τα πρώτα στοιχεία δύο γραμμών είναι ίσα μεταξύ τους, τότε ελέγχει το δεύτερο στοιχείο της γραμμής. Αν και αυτά είναι ίσα, πηγαίνει στο τρίτο κ.ο.κ. Αν όλα τα στοιχεία δύο γραμμών είναι ανά δύο (αντίστοιχα) ίσα μεταξύ τους, τότε φυσικά δεν έχει νόημα ποια από τις δύο γραμμές θα τοποθετηθεί πρώτη.
- Γράψτε το πρόγραμμα της σειριακής αναζήτησης χρησιμοποιώντας αναδρομή.
- Υλοποιήστε τη συνάρτηση **merge_sort** έτσι που να συγχωνεύει τρεις ήδη ταξινομημένους πίνακες.

Βιβλιογραφία

1. Jennifer Campel, Paul Gries, Jason Montojo, Greg Wilson (2019). **Practical Programming, An Introduction to Computer Science Using Python**. Publisher: The Pragmatic Bookself.
2. John Guttag (2015). **Υπολογισμοί και Προγραμματισμός Με Την Python**. Μετάφραση: Παναγιώτης Καναβός, Επιμέλεια: Γεώργιος Μανής, Εκδόσεις Κλειδάριθμος.
3. Αχιλλέας Καμέας (2000). **Τεχνικές Προγραμματισμού**. Τόμος Β. ΠΛΗ-10, Ελληνικό Ανοικτό Πανεπιστήμιο.
4. Eric Roberts. (2004). **Η Τέχνη και Επιστήμη της C**. Μετάφραση: Γιώργος Στεφανίδης, Παναγιώτης Σταυρόπουλος, Αλέξανδρος Χατζηγεωργίου, Εκδόσεις Κλειδάριθμός.

Κεφάλαιο 11: Οι πίνακες ως δομή δεδομένων

Οι **πίνακες (arrays)** είναι μία πολύτιμη δομή δεδομένων, ένα πολύ χρήσιμο εργαλείο το οποίο συναντά κανείς σε οποιαδήποτε γλώσσα προγραμματισμού έχει νόημα να το υποστηρίξει. Το γεγονός αυτό μας προκαλεί ακόμα μεγαλύτερη έκπληξη όταν διαπιστώνουμε ότι η Python δεν τους υποστηρίζει. Είναι, όμως, έτσι; Ή μήπως τελικά τους υποστηρίζει;

Η πραγματικότητα είναι ότι στην Python δεν υπάρχει τύπος πίνακα όπως υπάρχουν ενσωματωμένες δομές για λίστες ή συμβολοσειρές ή ακόμα και λεξικά. Χρησιμοποιώντας, όμως, τις λίστες με κατάλληλο τρόπο, είναι δυνατόν να φτιάξουμε πίνακες, όποιων διαστάσεων επιθυμούμε, και να τους χρησιμοποιήσουμε χωρίς ιδιαίτερη δυσκολία.

Στο κάτω κάτω, ακόμα κι αν δεν μας αρέσει και τόσο η ιδέα να χρησιμοποιήσουμε τις λίστες για να αναπαραστήσουμε πίνακες, δεν πρέπει να ξεχνάμε το μεγάλο όπλο της Python, τις βιβλιοθήκες. Αν ψάξουμε λίγο στις μηχανές αναζήτησης, θα βρούμε περισσότερες από μία βιβλιοθήκες της Python, ελεύθερες στο διαδίκτυο, οι οποίες θα ικανοποιήσουν απόλυτα και την πιο απαιτητική εφαρμογή. Αναζητήστε τις **NumPy** και **SciPy**, που είναι οι περισσότερο γνωστές, και θα βρείτε εύκολα το πώς λειτουργούν.

Εδώ δεν θα δούμε τους πίνακες μέσα από κάποια βιβλιοθήκη. Ο λόγος είναι εκπαιδευτικός. Με το να γνωρίζουμε το πώς λειτουργεί το λογισμικό που χρησιμοποιούμε έχουμε πάντοτε κέρδος. Η υλοποίηση των πράξεων των πινάκων με απλές δομές είναι πολύ χρήσιμη εκπαιδευτικά και έχετε πολλά να κερδίσετε παρακολουθώντας την. Αφού οι αλγόριθμοι και οι δομές αυτές γίνουν κτήμα σας, τότε χρησιμοποιήστε, αν θέλετε, μία από τις διαθέσιμες βιβλιοθήκες, όποια σας ταιριάζει καλύτερα.

11.1 Από τις λίστες στους πίνακες

Η έννοια του **μονοδιάστατου πίνακα (one dimensional array)** σχετίζεται άμεσα με την έννοια του **διανύσματος (vector)** στα μαθηματικά, αφού με έναν μονοδιάστατο πίνακα μπορούμε να αναπαραστήσουμε ένα διάνυσμα τιμών.

Ένας μονοδιάστατος πίνακας μπορεί κάλλιστα να αναπαρασταθεί με μία απλή λίστα. Αν έχουμε, για παράδειγμα, τη λίστα:

```
L=[2,4,6,8,9]
```

αυτό είναι ισοδύναμο με έναν μονοδιάστατο πίνακα 5 θέσεων, αφού το πρώτο του στοιχείο προσπελάζεται ως **L[0]** το δεύτερο ως **L[1]** κ.ο.κ. Το μόνο που ίσως ξενίζει λίγο είναι ο τρόπος με τον οποίο θα αρχικοποιηθεί ο πίνακας, ο οποίος είναι ο ακόλουθος: Πρακτικά αρχικοποιούμε μία λίστα με μηδενικά (ή ό,τι άλλο νομίζουμε) **N=5** θέσεων.

```
A=[0 for i in range(N)]
```

Στους δισδιάστατους πίνακες τα παραπάνω προσαρμόζονται ανάλογα. Ένας δισδιάστατος πίνακας είναι μία λίστα από λίστες. Ένας πίνακας **NxN** ορίζεται ως εξής και αυτό ξενίζει, ίσως, ακόμα περισσότερο:

```
A=[[0 for i in range(N)] for j in range(N)]
```

δηλαδή αποτελείται από μία λίστα **N** θέσεων, η οποία περιέχει μέσα της **N** λίστες, **N** θέσεων η καθεμία. Η προσπέλαση των στοιχείων του πίνακα γίνεται όπως αναμένουμε, το στοιχείο στη θέση **0,0** είναι το **A[0][0]**, ενώ το στοιχείο στην τελευταία γραμμή και τελευταία στήλη είναι το **A[N-1][N-1]**.

Όσα περιγράφηκαν για τους δισδιάστατους πίνακες γενικεύονται και σε περισσότερες διαστάσεις, υποστηρίζοντας με αυτόν τον τρόπο πίνακες μεγαλύτερων διαστάσεων.

Οι πίνακες, αφού στην ουσία είναι λίστες, διατηρούν όλες τις ιδιότητες των λιστών. Για παράδειγμα, για το πέρασμα παραμέτρων σε συναρτήσεις ισχύει ακριβώς ό,τι έχουμε δει στο αντίστοιχο κεφάλαιο.

11.2 Βασικές πράξεις πινάκων σε μονοδιάστατους πίνακες

Από τις βασικές πράξεις που ορίζονται σε μονοδιάστατους πίνακες θα επιλέξουμε τρεις για να δούμε από πιο κοντά. Στο Σχήμα 11.2 φαίνονται τρεις

συναρτήσεις: μία για την πρόσθεση (**v_add**), μία για τον βαθμωτό πολλαπλασιασμό (**s_mult**) και μία για το εσωτερικό γινόμενο (**inner_prod**). Το εσωτερικό γινόμενο το έχουμε δει αρκετά αναλυτικά στο κεφάλαιο 6, όταν είχαμε δει τη δομή **for**, αλλά το συμπεριλάβαμε διότι είναι ένα πολύ χαρακτηριστικό παράδειγμα.

Άλλωστε, λογικά, οι βρόχοι δεν πρέπει να σας δυσκολέψουν στη φάση αυτήν που βρίσκεστε τώρα. Δείτε τους προσεκτικά. Είναι υλοποιημένοι σε μορφή συνάρτησης. Θα επιμείνουμε περισσότερο σε αυτό. Θα σχολιάσουμε κυρίως τις παραμέτρους και τους ελέγχους που κάνουμε στην αρχή για τις εισόδους. Στην πρόσθεση, για παράδειγμα, οι δύο πίνακες πρέπει να έχουν την ίδια διάσταση και, φυσικά, οι αριθμοί να είναι πραγματικοί.

Το γεγονός ότι είμαστε μέσα σε μία συνάρτηση δεν μας εξασφαλίζει ότι ως παράμετροι θα δοθούν δύο πίνακες ίδιων διαστάσεων ώστε να είναι μαθηματικά επιτρεπτή η πράξη. Αν εμείς είμαστε οι δημιουργοί άλλα και οι χρήστες της συνάρτησης, τότε μπορούμε να προσέξουμε τι πίνακες θα δώσουμε σαν είσοδο, ώστε αυτοί να έχουν πάντοτε το ίδιο μέγεθος και να μη βρεθεί ποτέ η συνάρτησή μας στη δύσκολη θέση να πρέπει να προσθέσει πίνακες διαφορετικών διαστάσεων. Αν, όμως, τη συνάρτηση τη φτιάχνουμε για να τη δώσουμε και να χρησιμοποιηθεί από άλλους, τότε πρέπει να λάβουμε όλα τα μέτρα για πιθανές εσφαλμένες παραμέτρους στην είσοδο.

Στον κώδικα του Σχήματος 11.1, στη συνάρτηση **v_add** γίνεται έλεγχος αν οι δύο πίνακες έχουν τις ίδιες διαστάσεις. Αν δεν γινόταν αυτός ο έλεγχος, τότε, σύμφωνα με τον κώδικα, θα δημιουργούνταν ο πίνακας **z**, διάστασης ίδιας με του **x**, θα εκτελούνταν ο βρόχος με τόσες επαναλήψεις όσα και τα στοιχεία του **x** και, εάν τα στοιχεία του **y** ήταν περισσότερα από αυτά του **x**, τότε τα περίσσια στοιχεία θα αγνοούνταν. Δεν το θέλουμε αυτό. Αν τα στοιχεία του **y** ήταν λιγότερα από αυτά του **x**, θα παίρναμε μήνυμα λάθους όταν τα στοιχεία του **y** τελείωναν. Με τον έλεγχο που έχουμε βάλει, εξασφαλίζουμε ότι όλα θα γίνουν νόμιμα και σύμφωνα με τα μαθηματικά.

Το δεύτερο που πρέπει να ελέγξουμε είναι εάν τα στοιχεία των δύο πινάκων είναι πραγματικοί αριθμοί (το σύνολο των ακεραίων είναι υποσύνολο του συνόλου των πραγματικών). Η Python έχει τη συνάρτηση **isinstance(x,type)** για τη δουλειά αυτήν, η οποία ελέγχει αν το **x** είναι τύπου **type**. Πάντως, ακόμα κι αν δεν κάνουμε αυτόν τον έλεγχο δεν θα δημιουργηθεί μεγάλο πρόβλημα, αφού, αν προσπαθήσουμε να προσθέσουμε πράγματα που δεν προστίθενται, η Python θα διαμαρτυρηθεί και θα επιστρέψει μήνυμα λάθους. Λιγότερο κομψό από αυτό που θα επιστρέφαμε εμείς, αλλά, σε κάθε περίπτωση, δεν θα κάνει κάτι παράνομο. Για τον λόγο αυτόν δεν έχουμε ενσωματώσει στην **v_add** αυ-

τόν τον έλεγχο. Για να δούμε λίγο τη χρήση της **isinstance(x,type)**, στο Σχήμα 11.2 παραθέτουμε τη συνάρτηση **is_integer**, που ελέγχει αν όλα τα στοιχεία ενός πίνακα είναι ακέραιοι αριθμοί.

```
1  def v_add(x,y):
2      Nx=len(x)
3      Ny=len(y)
4      if Nx!=Ny: return 'error'
5      z=[0 for i in range(Nx)]
6      for i in range(Nx):
7          z[i]=x[i]+y[i]
8      return z
9
10 def s_mult(a,x):
11     N=len(x)
12     z=[0 for i in range(N)]
13     for i in range(N):
14         z[i]=a*x[i]
15     return z
16
17 def inner_prod(x,y):
18     Nx=len(x)
19     Ny=len(y)
20     if Nx!=Ny: return 'error'
21     z=0
22     for i in range(Nx):
23         z+=x[i]*y[i]
24     return z
```

Σχήμα 11.1: Πράξεις με μονοδιάστατους πίνακες.

```
1  def is_integer(x):
2      for i in x:
3          if not isinstance(i,int):
4              return False
5      return True
```

Σχήμα 11.2: Έλεγχος αν όλα τα στοιχεία ενός πίνακα είναι ακέραιοι.

Παρακάτω, στο Σχήμα 11.1 φαίνονται δύο ακόμα συναρτήσεις, μία για τον βαθμωτό πολλαπλασιασμό (**s_mult**) και μία για το εσωτερικό γινόμενο (**inner_prod**).

Θέλω μόνο να σταθώ λίγο στο εσωτερικό γινόμενο. Δεν νομίζω να σας δυσκολέψει το ότι, ενώ έχουμε δύο πίνακες, μας αρκεί μία μεταβλητή για να τους διασχίσουμε. Θέλουμε να πολλαπλασιάσουμε το πρώτο στοιχείο του πρώτου πίνακα με το πρώτο στοιχείο του δεύτερου, το δεύτερο στοιχείο του πρώτου πίνακα με το δεύτερο στοιχείο του δεύτερου κ.ο.κ. Το ίδιο, άλλωστε, είχε συμβεί και στην πρόσθεση των πινάκων. Το τονίζω εδώ για να το χρησιμοποιήσουμε παρακάτω, στον πολλαπλασιασμό πινάκων, διότι, για κάποιο λόγο τον οποίο δεν είμαι σίγουρος ότι ξέρω, στον πολλαπλασιασμό πινάκων αρκετοί δυσκολεύονται με αυτό το σημείο.

11.3 Βασικές πράξεις πινάκων σε πολυδιάστατους πίνακες

Πολυδιάστατοι είναι πίνακες (**multi-dimensional arrays**) που η διάστασή τους είναι μεγαλύτερη του 1. Χωρίς βλάβη της γενικότητας θα μιλήσουμε στην ενότητα αυτήν για δισδιάστατους πίνακες. Για πίνακες μεγαλύτερων διαστάσεων αρκεί να γενικεύσετε αυτά που θα πούμε και δεν θα έχετε κανένα πρόβλημα να τους χρησιμοποιήσετε. Στα Σχήματα 11.3 και 11.4 φαίνονται οι κώδικες για την πρόσθεση **m_add** και τον πολλαπλασιασμό **m_mult** πινάκων δύο διαστάσεων αντίστοιχα.

```
1  def m_add(x,y):
2      N1=len(x)
3      M1=len(x[0])
4      N2=len(y)
5      M2=len(y[0])
6      if N1!=N2: return 'error'
7      if M1!=M2: return 'error'
8      for i in range (N1):
9          if len(x[i])!=M1: return 'error'
10         if len(y[i])!=M2: return 'error'
11     z=[[ 0 for i1 in range (M1)] for i2 in range (N1)]
12     for i in range (N1):
13         for j in range (M1):
14             print i,j
15             z[i][j]=x[i][j]+y[i][j]
16     return z
```

Σχήμα 11.3: Πρόσθεση δισδιάστατων πινάκων.

Η πρόσθεση πινάκων είναι μια απλή γενίκευση αυτής των μονοδιάστατων. Χρειαζόμαστε δύο και όχι έναν βρόχο (με μεταβλητές βρόχου τις **i** και **j** στον συγκεκριμένο κώδικα), φωλιασμένους, ώστε να καλύψουμε τις δύο διαστάσεις των πινάκων, ενώ σε κάθε βήμα αυτού του φωλιασμένου βρόχου γίνεται η πρόσθεση των στοιχείων **i**,**j** κάθε πίνακα και προκύπτει το στοιχείο **i**,**j** του καινούργιου πίνακα. Δεν νομίζω ότι σας δυσκολεύει, οπότε δεν θα το σχολιάσω περισσότερο.

Θα μείνω, πάλι, στους ελέγχους στην αρχή του κώδικα. Θα θεωρήσουμε ότι πλήρεις έλεγχοι είναι αναγκαίοι και ότι η συνάρτηση θα δοθεί προς χρήση σε τρίτους. Θα περιοριστούμε και πάλι στον έλεγχο διαστάσεων. Κάθε πίνακας έχει δύο διαστάσεις. Η πρώτη είναι ο αριθμός των λιστών που περιέχονται μέσα στην κύρια λίστα. Μπορούμε να τον πάρουμε χρησιμοποιώντας τη **len** στους πίνακες **x** και **y**. Αν εκχωρήσουμε τις δύο αυτές διαστάσεις στις μεταβλητές **N1** και **N2**, θα πρέπει φυσικά να ισχύει **N1=N2**.

Καθένας από αυτούς τους πίνακες έχει μέσα του **N1** λίστες (ή **N2**, είναι μεταξύ τους ίσα), οι οποίες πρέπει να έχουν ακριβώς τον ίδιο αριθμό στοιχείων μέσα τους. Τίποτε δεν απαγορεύει στον χρήστη της συνάρτησης να δώσει ως είσοδο μία λίστα που μέσα της έχει τρεις λίστες: η πρώτη με τρία στοιχεία, η δεύτερη με δύο και η τρίτη με οκτώ. Εμείς οφείλουμε να το ελέγξουμε και να επιστρέψουμε μήνυμα λάθους σε μία τέτοια περίπτωση. Έτσι, στη συνέχεια των ελέγχων στο Σχήμα 11.4 εκχωρούμε στις μεταβλητές **M1** και **M2** το μέγεθος της πρώτης από τις εσωτερικές λίστες των πινάκων **x** και **y** αντίστοιχα. Με έναν βρόχο ελέγχουμε ότι καθεμία από τις υπόλοιπες εσωτερικές λίστες έχει μέγεθος ίσο με την πρώτη. Φυσικά, απαιτείται και ο έλεγχος αν **M1=M2**, διότι στην περίπτωση που δεν ισχύει πάλι, πρέπει να επιστρέψουμε μήνυμα λάθους για να είμαστε συνεπείς με τα μαθηματικά.

Στον πολλαπλασιασμό πινάκων πρέπει, για μια ακόμα φορά, να ελέγξουμε ότι κάθε είσοδος είναι πραγματικά ένας πίνακας, δηλαδή ότι οι εσωτερικές λίστες έχουν όλες τον ίδιο αριθμό στοιχείων. Οι υπόλοιποι έλεγχοι έχουν να κάνουν με τον ορισμό της πράξης του πολλαπλασιασμού πινάκων. Θυμίζω ότι, εάν έχω την πράξη **C=AxB**, οι γραμμές του **A** πρέπει να είναι ίσες με τις στήλες του **B**, δηλαδή **M1=N2** στον κώδικα του Σχήματος 11.4. Ο πίνακας που προκύπτει πρέπει να είναι διαστάσεων **N1xM2**. Έτσι, ορίζουμε τον πίνακα **z** να έχει διάσταση **N1xM2**.

```
1  def m_mult(x,y):
2      N1=len(x)
3      M1=len(x[0])
4      N2=len(y)
5      M2=len(y[0])
6      if M1!=N2: return 'error'
7      for i in range(N1):
8          if len(x[i])!=M1: return 'error'
9      for i in range(N2):
10         if len(y[i])!=M2: return 'error'
11     z=[[ 0 for i1 in range (M2)] for i2 in range (N1)]
12     for i in range (N1):
13         for j in range (M2):
14             for k in range (N2):
15                 z[i][j]+=x[i][k]*y[k][j]
16     return z
```

Σχήμα 11.4: Πολλαπλασιασμός πινάκων.

Έπειτα ακολουθούν οι βρόχοι. Εδώ έχουμε φωλιασμένους βρόχους σε τρία επίπεδα. Θα χρησιμοποιήσω το συμπέρασμα που βγάλαμε όταν συζητούσαμε το εσωτερικό γινόμενο παραπάνω στο κεφάλαιο αυτό, ότι, για να υπολογίσουμε το εσωτερικό γινόμενο, μας αρκεί ένας απλός βρόχος. Αφού κάθε στοιχείο του πίνακα **z** είναι το αποτέλεσμα ενός εσωτερικού γινομένου, θα χρησιμοποιήσουμε έναν διπλό βρόχο για να διανύει τα στοιχεία του πίνακα **z** και έναν απλό μέσα σε αυτόν που θα βρίσκει το εσωτερικό γινόμενο των στηλών και των γραμμών που ορίζονται από τις δύο μεταβλητές βρόχων των δύο εξωτερικότερων δομών **for**.

11.4 Άλλες συναρτήσεις πάνω σε πίνακες

Ας ολοκληρώσουμε τις συναρτήσεις που θα υλοποιήσουμε στο κεφάλαιο αυτό με την αναστροφή ενός πίνακα, τον έλεγχο αν είναι μοναδιαίος και τον έλεγχο αν είναι συμμετρικός. Την αναστροφή του πίνακα θα τη δούμε μέσα από το Σχήμα 11.5, ενώ για τις άλλες δύο μπορείτε να βρείτε δύο μικρές ταινίες που πιθανόν να σας κάνουν την κατανόηση περισσότερο ευχάριστη.

```
1  def m_transpose(x):
2      N=len(x);
3      M=len(x[0])
4      for i in range (N):
5          if len(x[i])!=M: return 'error'
6      y=[[ 0 for i1 in range (N)] for i2 in range (M)]
7      for i in range (M):
8          for j in range (N):
9              y[i][j]=x[j][i]
10     return y
```

Σχήμα 11.5: Αναστροφή πίνακα.

Στην αναστροφή ενός πίνακα οι γραμμές γίνονται στήλες και οι στήλες γραμμές. Θα δούμε δύο τρόπους για να το κάνουμε αυτό. Στο Σχήμα 11.5 φαίνεται ο κώδικας μίας συνάρτησης που επιστρέφει τον ανάστροφο ενός πίνακα. Αφού εξάγει τις δύο διαστάσεις του πίνακα και τις τοποθετήσει στις μεταβλητές **N** και **M** και κάνει τους απαιτούμενους ελέγχους, δημιουργεί έναν πίνακα **y** διάστασης **MxN**. Με έναν διπλό βρόχο και χρησιμοποιώντας ως μεταβλητές βρόχου τις **i** και **j**, διανύει τον πίνακα **y** και σε κάθε θέση **i**,**j** του **y** τοποθετεί την τιμή που υπάρχει στη θέση **j**,**i** του πίνακα **x**. Στο τέλος επιστρέφει τον πίνακα **y** σαν αποτέλεσμα.

```
1  for i in range(N):
2      for j in range(i):
3          A[i][j],A[j][i]=A[j][i],A[i][j]
```

Σχήμα 11.6: Αναστροφή ενός πίνακα επί τόπου.

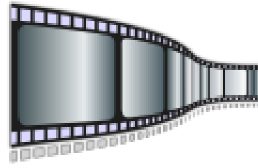
Στο Σχήμα 11.6 βλέπουμε πώς μπορεί να γίνει μία αναστροφή πίνακα, επί τόπου, δηλαδή χωρίς να χρησιμοποιήσουμε δεύτερο πίνακα. Για είναι επιτρεπτή η επί τόπου αναστροφή ενός πίνακα πρέπει αυτός να είναι τετραγωνικός, δηλαδή οι δύο διαστάσεις του να είναι μεταξύ τους ίσες (αφού οι γραμμές πρέπει να γίνουν στήλες και οι στήλες γραμμές). Μας δίνεται ο τετραγωνικός πίνακας A με διάσταση **NxN**. Η κύρια διαγώνιος δεν θα αλλάξει. Αυτό που θα αλλάξει είναι τα στοιχεία από τα δύο τριγωνάκια που απομένουν. Θα διατρέξουμε το ένα από τα δύο τριγωνάκια με έναν κατάλληλο βρόχο και θα εναλλάσσουμε το κάθε στοιχείο που επισκεπτόμαστε (**i**,**j**) με το κατάλληλο στοιχείο (**j**,**i**) από το άλλο τριγωνάκι. Μένει μόνο να δούμε πώς θα ορίσουμε τα όρια των βρόχων ώστε να επισκεφτούμε ακριβώς τα στοιχεία ενός εκ των δύο τριγώνων.

Ο εξωτερικός από τους δύο βρόχους δεν μπορεί παρά να πηγαίνει από το 0 έως το **N**, αλλιώς θα ήταν αδύνατον να προσπελάσουμε κάποια από τα στοιχεία του πίνακα. Ο εσωτερικός βρόχος θα πρέπει για την πρώτη γραμμή να μην προσπελάσει κανένα στοιχείο, για τη δεύτερη να προσπελάσει ένα, για την τρίτη δύο και για την τελευταία όλα εκτός από το τελευταίο. Αν αναλογιστούμε ότι οι πίνακες ξεκινούν από το στοιχείο **0,0** και φτάνουν στο **N-1,N-1**, ενώ τα στοιχεία της κύριας διαγωνίου είναι τα **i,i**, τότε είναι φανερό ότι στην πρώτη γραμμή πρέπει να φτάσουμε έως τη στήλη 0 (άρα να μην πάρουμε κανένα στοιχείο), στη δεύτερη έως τη στήλη 1 (άρα να πάρουμε ένα στοιχείο), στην τρίτη έως τη στήλη 2 κ.ο.κ, ενώ στην τελευταία έως τη στήλη **N-1**, δηλαδή την τελευταία, δηλαδή να αφήσουμε μόνο ένα στοιχείο. Δεν είναι πολύ εύκολο, είναι ενδιαφέρον και βέβαια, όπως πάντα, έχει πλάκα.

```
1  def m_isUnary(x):
2      N=len(x)
3      for i in range (N):
4          if len(x[i])!=N: return 'error'
5      for i in range (N):
6          for j in range (N):
7              if i==j and x[i][j]!=1: return False
8              if i!=j and x[i][j]!=0: return False
9      return True
10
11 def m_isSymmetric(x):
12     N=len(x)
13     for i in range (N):
14         if len(x[i])!=N: return 'error'
15     for i in range (N):
16         for j in range (i+1,N):
17             if x[i][j]!=x[j][i]: return False
18     return True
```

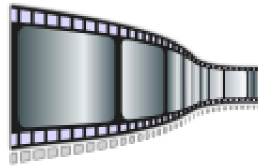
Σχήμα 11.7: Άλλες συναρτήσεις με πίνακες.

Για τις άλλες δύο συναρτήσεις, τον έλεγχο συμμετρικότητας και τον έλεγχο αν ένας πίνακας είναι μοναδιαίος, παρακολουθήστε τις Ταινίες 11.1 και 11.2 αντίστοιχα. Ο κώδικας των δύο αυτών συναρτήσεων φαίνεται στο Σχήμα 11.7.



Ταινία 11.1: Έλεγχος αν ένας πίνακας είναι μοναδιαίος.

http://repfiles.kallipos.gr/file/22401



Ταινία 11.2: Έλεγχος αν ένας πίνακας είναι συμμετρικός.

http://repfiles.kallipos.gr/file/22402

11.5 Συσχέτιση, συνέλιξη, αυτοσυσχέτιση

Η **συσχέτιση (correlation)** ή **ετεροσυσχέτιση (cross-correlation)**, η **συνέλιξη (convoution)** και η **αυτοσυσχέτιση (autocorrelation)** είναι τρεις πράξεις που βρίσκουν εφαρμογή σε πολλές περιοχές, όπως για παράδειγμα στην επεξεργασία σήματος. Ο τρόπος υπολογισμού τους είναι παρόμοιος.

Η **συσχέτιση** δίνεται από τον ακόλουθο τύπο. Ουσιαστικά πρόκειται για τον υπολογισμό ενός κυλιόμενου εσωτερικού γινομένου.

$$r_{xy}(k) = \sum_{n=-\infty}^{n=+\infty} x(n)y(n+k)$$

Αντίστοιχα η **αυτοσυσχέτιση** δίνεται από τον τύπο που ακολουθεί. Πρόκειται για τη συσχέτιση ενός σήματος με τον εαυτό του.

$$r_{xx}(k) = \sum_{n=-\infty}^{n=+\infty} x(n)x(n+k)$$

Τέλος, η **συνέλιξη** δίνεται από τον παρακάτω τύπο. Προσέξτε ότι το δεύτερο σήμα πρακτικά το διαπερνούμε με την αντίθετη φορά από ό,τι στη συσχέτιση.

$$(x * y)(k) = \sum_{n=-\infty}^{n=+\infty} x(n)y(k-n)$$

Ας ξεκινήσουμε με τη συσχέτιση. Θα θεωρήσουμε ότι, όταν δύο σήματα δεν έχουν το ίδιο μέγεθος, τότε το μικρότερο θα συμπληρωθεί με μηδενικά στο τέλος έως ότου αποκτήσει το μέγεθος του άλλου.

Έστω λοιπόν ότι έχουμε δύο σήματα, τα **x** και **y**, και τα έχουμε τοποθετήσει σε δύο πίνακες:

x=[2,3,4,5,6]

και

y=[2,3,4]

Το σήμα **y** θα συμπληρωθεί με μηδενικά και θα γίνει:

y=[2,3,4,0,0]

Η συσχέτιση των δύο σημάτων θα δώσει:

xcorr(x,y)=[0,0,8,18,29,38,47,28,12]

x: | 2 | 3 | 4 | 5 | 6 | → 2x0+3x0+4x0+5x0+6x0=0
pad: | 0 | | | | |

x: | 2 | 3 | 4 | 5 | 6 | → 2x0+3x0+4x0+5x0+6x0=0
pad: | 0 | 0 | | | |

x: | 2 | 3 | 4 | 5 | 6 | → 2x4+3x0+4x0+5x0+6x0=8
pad: | 4 | 0 | 0 | | |

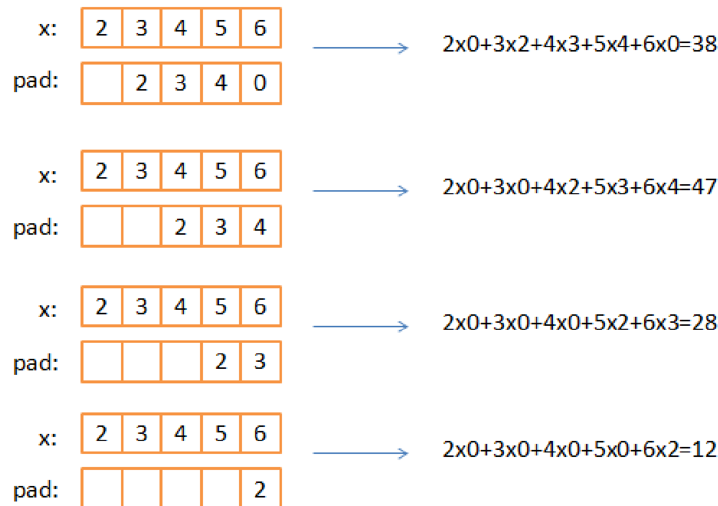
x: | 2 | 3 | 4 | 5 | 6 | → 2x3+3x4+4x0+5x0+6x0=18
pad: | 3 | 4 | 0 | 0 | |

x: | 2 | 3 | 4 | 5 | 6 | → 2x2+3x3+4x4+5x0+6x0=29
pad: | 2 | 3 | 4 | 0 | 0 |

Σχήμα 11.8: Υπολογισμός συσχέτισης.

Ο υπολογισμός των πέντε πρώτων από τους παραπάνω αριθμούς φαίνεται στο Σχήμα 11.8, ενώ των τεσσάρων τελευταίων στο Σχήμα 11.9. Στα σχήματα

αυτά εμφανίζονται δύο πίνακες. Ο πρώτος είναι ο πίνακας **x** όπως τον έχουμε ορίσει παραπάνω. Ο δεύτερος προκύπτει από τον πίνακα **y** μετακινώντας κάθε φορά τα στοιχεία του μία θέση δεξιά. Για τον λόγο αυτόν ας τον ονομάσουμε **pad**.



Σχήμα 11.9: Υπολογισμός συσχέτισης (συνέχεια).

Στο πρώτο βήμα, ο πίνακας **pad** αποτελείται από το πρώτο στοιχείο του **y**, το οποίο τυχαίνει να είναι το μηδέν, τοποθετημένο στην πρώτη θέση του. Τα υπόλοιπα στοιχεία του μπορούμε να τα θεωρούμε μηδενικά. Στο δεύτερο βήμα, ο πίνακας **pad** αποτελείται από τα δύο πρώτα στοιχεία του **y** τοποθετημένα στις δύο πρώτες θέσεις του. Σε κάθε βήμα μετακινούμε τα στοιχεία του πίνακα κατά μία θέση δεξιότερα, έως ότου και το τελευταίο στοιχείο του **y** εξέλθει από το **pad**.

```
1 def product(a,b):
2     result=0
3     for i in range(len(a)):
4         result=result+a[i]*b[i]
5     return result
```

Σχήμα 11.10: Συνάρτηση υπολογισμού εσωτερικού γινομένου.

Σε κάθε βήμα υπολογίζουμε το εσωτερικό γινόμενο των πινάκων x και pad. Για τον σκοπό αυτόν φτιάχνουμε μία συνάρτηση η οποία θα βοηθήσει στο

να μειώσουμε τον υπόλοιπο κώδικα. Η συνάρτηση αυτή φαίνεται στο Σχήμα 11.10. Παρόμοια συνάρτηση για το εσωτερικό γινόμενο έχουμε συζητήσει σε προηγούμενο κεφάλαιο, αλλά έτσι και αλλιώς δεν νομίζω να σας δυσκόλευε.

```
1  def xcorr(x,y):
2    if len(x)>len(y):
3      for i in range(len(x)-len(y)):
4        y.append(0)
5    if len(y)>len(x):
6      for i in range(len(y)-len(x)):
7        x.append(0)
8    pad=[0 for i in range(len(x))]
9    result=[]
10   for i in range(len(x)):
11     pad.pop(-1)
12     pad.insert(0,y[-(i+1)])
13     result.append(product(pad,x))
14   for i in range(len(x)-1):
15     pad.pop(-1)
16     pad.insert(0,0)
17     result.append(product(pad,x))
18   return result
```

Σχήμα 11.11: Η συνάρτηση της συσχέτισης.

Η κύρια συνάρτηση, η **xcorr**, φαίνεται στο Σχήμα 11.11. Στις πρώτες τρεις γραμμές ελέγχεται εάν ο πίνακας **x** έχει περισσότερα στοιχεία από τον **y**, και στην περίπτωση αυτήν προστίθενται στο τέλος του **y** όσα μηδενικά χρειάζονται ώστε οι δύο πίνακες να αποκτήσουν το ίδιο μέγεθος. Στις τρεις επόμενες γραμμές γίνονται οι αντίστοιχες ενέργειες για την περίπτωση που ο **y** έχει περισσότερα στοιχεία από τον **x**. Σε κάθε περίπτωση, μετά την εκτέλεση των γραμμών αυτών, οι δύο πίνακες έχουν το ίδιο μέγεθος.

Στη συνέχεια ορίζεται ο πίνακας **pad** και τα στοιχεία του αρχικοποιούνται σε μηδέν, καθώς και η λίστα **result**, η οποία αρχικοποιείται σε κενή. Η λίστα **result** θα περιέχει στο τέλος το αποτέλεσμα, όλα τα εσωτερικά γινόμενα που θα υπολογιστούν, με τη σωστή σειρά. Κάθε φορά που θα υπολογίζεται ένα, θα τοποθετείται στο τέλος της λίστας με μία κλήση της **append** και πιο συγκεκριμένα: **result.append(product(pad,x))**.

Το κύριο μέρος της συνάρτησης αποτελείται από δύο βρόχους. Ο πρώτος βρόχος κάνει **len(x)** επαναλήψεις και εκτελεί τα **len(x)** πρώτα βήματα. Σε

κάθε βήμα προετοιμάζουμε κατάλληλα το **pad**. Αρχικά ξεκινάει έχοντας όλα τα στοιχεία του μηδενικά. Σε κάθε βήμα βγάζουμε ένα στοιχείο από τα δεξιά (**pad.pop(-1)**) και προσθέτουμε ένα στοιχείο αριστερά (**pad.insert(0,y[-(i+1)])**), και πιο συγκεκριμένα το i-οστό από δεξιά, όπου **i** το τρέχον βήμα. Μην μπερδευτείτε, το - μπροστά σημαίνει ότι μετράμε από το τέλος του πίνακα, ενώ το +1 υπάρχει εκεί ώστε την πρώτη φορά που το **i** είναι 0 η παράσταση να είναι ίση με 1.

Ο δεύτερος βρόχος κάνει περίπου την ίδια δουλειά. Αυτό που τον διαφοροποιεί είναι ότι τα στοιχεία που γίνονται **insert**, που εισάγονται δηλαδή στα αριστερά του πίνακα, δεν προέρχονται από τον **x** αλλά είναι μηδενικά: **pad.insert(0,0)**.

Στο τέλος επιστρέφουμε το αποτέλεσμα με την **return**.

Ο υπολογισμός της αυτοσυσχέτισης μπορεί να γίνει με τον ίδιο τρόπο. Ευκολότερα, όμως, μπορούμε να χρησιμοποιήσουμε τη συνάρτηση που μόλις φτιάξαμε καλώντας την με την ίδια είσοδο και για τις δύο παραμέτρους της. Και αυτό διότι ισχύει ότι:

acorr(x)=xcorr(x,x)

Ο κώδικας φαίνεται στο Σχήμα 11.12.

```
1 def acorr(x):
2   return xcorr(x,x)
```

Σχήμα 11.12: Η συνάρτηση της αυτοσυσχέτισης.

Τέλος, θα φτιάξουμε και μια συνάρτηση που υπολογίζει τη συνέλιξη. Εδώ, θα φτιάξουμε λίγο διαφορετικό κώδικα. Όταν έχουμε δύο σήματα διαφορετικού μεγέθους, δεν θα συμπληρώσουμε με μηδενικά, αλλά θα 'σύρουμε' το ένα από τα δύο, με τον ίδιο τρόπο που κάναμε στην συσχέτιση, υπολογίζοντας για όλες τις δυνατές θέσεις το εσωτερικό του γινόμενο με το άλλο. Δείτε ένα παράδειγμα υπολογισμού στο Σχήμα 11.13.

x:	2	3	4	5	6	→ 2x2+3x0+4x0+5x0+6x0=4
pad:	2					
x:	2	3	4	5	6	→ 2x3+3x2+4x0+5x0+6x0=12
pad:	3	2				
x:	2	3	4	5	6	→ 2x4+3x3+4x2+5x0+6x0=25
pad:	4	3	2			
x:	2	3	4	5	6	→ 2x0+3x4+4x3+5x2+6x0=34
pad:		4	3	2		
x:	2	3	4	5	6	→ 2x0+3x0+4x4+5x3+6x2=43
pad:			4	3	2	
x:	2	3	4	5	6	→ 2x0+3x0+4x0+5x4+6x3=38
pad:				4	3	
x:	2	3	4	5	6	→ 2x0+3x0+4x0+5x0+6x4=24
pad:					4	

Σχήμα 11.13: Παράδειγμα υπολογισμού της συνέλιξης.

Ο κώδικας της συνάρτησης φαίνεται στο Σχήμα 11.14. Στις πρώτες έξι γραμμές αποφασίζεται ποιο σήμα είναι το μεγαλύτερο και το ονομάζουμε **large**. Το άλλο το ονομάζουμε **small**. Όταν τα δύο σήματα έχουν το ίδιο μήκος, τυχαία το ένα από τα δύο (το δεύτερο) παίρνει την ονομασία **large** και το άλλο **small**.

Οι τρεις βρόχοι που ακολουθούν φροντίζουν για την ολίσθηση του μικρού σήματος επάνω στο μεγάλο. Ο πρώτος, κατά τρόπο παρόμοιο με αυτόν που είδαμε στη συσχέτιση, εισάγει ένα ένα τα στοιχεία του **small** στο **pad** και εκτελεί τα εσωτερικά γινόμενα. Ο δεύτερος είναι κάτι που δεν χρειάστηκε να γίνει στη συσχέτιση. Φροντίζει για τη μετακίνηση του **pad** προς τα δεξιά, πάνω στο **large**, έως ότου φτάσουμε στον τρίτο βρόχο, ο οποίος αναλαμβάνει το τελευταίο κομμάτι, που, όπως και στη συσχέτιση, εισάγει μηδενικά στο pad για να

υπολογιστεί το τελευταίο κομμάτι των υπολογισμών.

```
1  def conv(x,y):
2      if len(x)>len(y):
3          small=y
4          large=x
5      else:
6          small=x
7          large=y
8      pad = [0 for i in range(len(small))]
9      result=[]
10     for i in range(len(small)):
11         pad.pop(-1)
12         pad.insert(0,small[i])
13         result.append(product(pad,large[:len(pad)]))
14     for i in range(len(pad),len(large)-len(pad)):
15         result.append(product(pad,large[i:i+len(pad)]))
16     for i in range(1,len(small)):
17         pad.pop(-1)
18         pad.insert(0,0)
19         result.append(product(pad,large[-len(pad):]))
20     return(result)
```

Σχήμα 11.14: Η συνάρτηση της συνέλιξης.

Ασκήσεις που μπορείτε να κάνετε μόνοι σας

- Να γράψετε μία συνάρτηση που να υπολογίζει αν ένας πίνακας είναι άνω τριγωνικός.
- Να φτιάξετε μία συνάρτηση η οποία παίρνει σαν είσοδο έναν τετραγωνικό πίνακα 3x3 και υπολογίζει την ορίζουσά του.
- Να γράψετε μία συνάρτηση που να ταξινομεί τις γραμμές ενός δισδιάστατου πίνακα με βάση το πρώτο στοιχείο της γραμμής. Αν τα πρώτα στοιχεία δύο γραμμών είναι ίσα, τότε να ελέγχει τα δεύτερα στοιχεία, αν και αυτά είναι ίσα, να ελέγχει τα τρίτα κ.ο.κ.
- Να γράψετε μία συνάρτηση που να επιστρέφει την κατ' απόλυτη τιμή μικρότερη διαφορά που μπορούμε να βρούμε αφαι-

ρώντας δύο στοιχεία ενός δισδιάστατου πίνακα **NxN**. Να σχηματίσετε, δηλαδή, όλα τα πιθανά ζεύγη στοιχείων ενός πίνακα και να βρείτε ποια από αυτά έχουν τη μικρότερη κατ' απόλυτη τιμή διαφορά και να την επιστρέψετε σαν αποτέλεσμα της συνάρτησης.

- Δοκιμάστε να γράψετε μία συνάρτηση που να υπολογίζει τον αντίστροφο ενός πίνακα.

Κεφάλαιο 12: Είσοδος και έξοδος δεδομένων σε αρχεία

Τα δεδομένα που επεξεργαζόμαστε, καθώς και ο κώδικας που τρέχουμε, βρίσκονται αποθηκευμένα στη μνήμη **RAM (Random Access Memory)** του υπολογιστή. Τα δεδομένα διατηρούνται εκεί όσο ο υπολογιστής λειτουργεί. Όταν τον σβήσουμε, ό,τι βρίσκεται στη μνήμη RAM χάνεται.

Δεδομένα και προγράμματα υπάρχουν αποθηκευμένα και στη μνήμη **ROM (Read Only Memory)**, η οποία, όμως, έχει διαφορετικό σκοπό. Τα περιεχόμενά της τα έχει γράψει ο κατασκευαστής του υπολογιστή και πρόκειται για αποθηκευμένα δεδομένα και προγράμματα τα οποία έχουν να κάνουν με βασικές λειτουργίες και, φυσικά, δεν επιτρέπεται να τροποποιηθούν. Εκεί, οι πληροφορίες διατηρούνται όταν σβήσουμε τον υπολογιστή, αλλά ούτε τις τοποθετήσαμε εμείς, ούτε μπορούμε να τις τροποποιήσουμε.

Αν, λοιπόν, επιθυμούμε μια πληροφορία να διατηρηθεί αφότου σβήσουμε τον υπολογιστή, η αποθήκευση πρέπει να γίνει σε κάποια άλλη μονάδα, εσωτερική στον υπολογιστή ή εξωτερική, όπως ο σκληρός δίσκος, ένας **οπτικός δίσκος (CD-ROM)**, μία **μνήμη USB (USB stick)** ή κάποια **κάρτα μνήμης (memory card)**. Η τεχνολογία τρέχει τόσο πολύ (και) σε αυτό το σημείο, που τα μέσα αποθήκευσης πληθαίνουν μέρα με τη μέρα, αλλά να σημειώσουμε ότι γίνονται όλο και πιο γρήγορα και πιο αξιόπιστα. Τα μέσα αποθήκευσης εξελίσσονται τόσο γρήγορα, που σε πολύ λίγο καιρό αυτή η παράγραφος θα χρειάζεται ενημέρωση.

Τα **αρχεία (files)** αποτελούν δομημένα βασικά στοιχεία πληροφορίας τα οποία βρίσκονται στα μέσα αποθήκευσης μιας υπολογιστικής μονάδας. Η μορφή τους ποικίλλει ανάλογα με τον τύπο τους και θα μπορούσε να είναι: δεδομένα, κείμενο (text), εικόνα (image), ήχος (sound), video ή κάποια άλλη πολυμεσική

πληροφορία (multimedia). Ακόμα και ένα πρόγραμμα το οποίο θα αποθηκεύσουμε στον δίσκο θεωρείται ένα αρχείο, είτε είναι σε μορφή πηγαίου κώδικα είτε είναι μεταφρασμένο σε γλώσσα μηχανής. Όλα αυτά θα μείνουν στο μέσο αποθήκευσης και αφού σβήσει ο υπολογιστής, και θα παραμείνουν εκεί μέχρι να τα ζητήσουμε ξανά.

Δεν είναι, όμως, μόνο αυτός ο λόγος για τον οποίο γράφουμε πληροφορία στα αρχεία. Μέσα από αυτά μπορούμε να τη μοιραστούμε με άλλους, να τη στείλουμε σε διαφορετικούς υπολογιστές ή και σε άλλες συσκευές που έχουν τη δυνατότητα και τη νοημοσύνη να την καταλάβουν, ή απλά να την παρουσιάσουν στον χρήστη τους.

Η Python, όπως και όλες οι γλώσσες προγραμματισμού για τις οποίες αυτό έχει νόημα, υποστηρίζει αρχεία. Τα περισσότερα από αυτά που θα δούμε ακολουθούν τη φιλοσοφία που θα συναντούσε κανείς στις περισσότερες γλώσσες προγραμματισμού.

12.1 Άνοιγμα και κλείσιμο αρχείου

Είναι το πρώτο και το τελευταίο πράγμα που κάνουμε με ένα αρχείο. Λέγοντας **άνοιγμα (open)** εννοούμε ότι πληροφορούμε την Python ότι θέλουμε να χρησιμοποιήσουμε το συγκεκριμένο αρχείο. Εκείνη το βρίσκει στο μέσο αποθήκευσης και το κάνει διαθέσιμο σε εμάς. Κάνοντας **κλείσιμο (close)** ενός αρχείου πληροφορούμε την Python ότι το αρχείο δεν μας χρειάζεται άλλο, έχουμε διαβάσει ή τροποποιήσει ό,τι χρειαζόμασταν, και μπορεί να θεωρήσει ότι δεν το θέλουμε πια να είναι ανοικτό.

Το άνοιγμα ενός αρχείου γίνεται με την κλήση της **open**. Η **open** παίρνει σαν παράμετρο το όνομα ή την πλήρη διαδρομή του αρχείου, πώς, δηλαδή, αυτό ονομάζεται και πού βρίσκεται στο μέσο αποθήκευσης. Στη δεύτερη παράμετρο δηλώνουμε τον τρόπο με τον οποίο θέλουμε να ανοίξουμε το αρχείο και οι επιλογές μας είναι οι εξής:

- **'r'** (άνοιγμα για ανάγνωση -**read**): Επιθυμούμε μόνο να διαβάσουμε πληροφορία από το αρχείο
- **'w'** (άνοιγμα για εγγραφή - **write**): Επιθυμούμε να γράψουμε πληροφορία στο αρχείο. Ό,τι πληροφορία υπήρχε στο αρχείο πριν το ανοίξουμε χάνεται. Αν το όνομα του αρχείου που δώσαμε δεν υπάρχει στο μέσο αποθήκευσης, τότε θα δημιουργηθεί ένα καινούργιο αρχείο.
- **'a'** (άνοιγμα για πρόσθεση - **append**): Επιθυμούμε να γράψουμε

πληροφορία στο αρχείο χωρίς να χαθεί η πληροφορία που υπήρχε εκεί πριν το ανοίξουμε. Η νέα πληροφορία θα γραφεί μετά την παλιά. Αν το αρχείο δεν υπάρχει στο μέσο αποθήκευσης, τότε θα δημιουργηθεί ένα νέο.

Η κλήση **open** επιστρέφει σαν αποτέλεσμα έναν δείκτη στο αρχείο. Εμάς (σε αυτό τουλάχιστον το βιβλίο) δεν μας νοιάζει ακριβώς τι είναι αυτό, αλλά πώς θα το χρησιμοποιήσουμε. Ο δείκτης αυτός θα χρησιμοποιηθεί από εδώ και πέρα όποτε θέλουμε να προσπελάσουμε το αρχείο, για μια πράξη ανάγνωσης ή εγγραφής, για παράδειγμα, ή για να κλείσουμε ένα αρχείο.

Για να κλείσουμε το αρχείο, απλά καλούμε την **close** εφαρμόζοντάς την στον δείκτη του αρχείου.

Η σύνταξη της **open** είναι η ακόλουθη:

fp = open(file_name,mode)

όπου:
fp: ο δείκτης που θα δημιουργηθεί για το αρχείο,
file_name: το όνομα ή η πλήρης διαδρομή για το αρχείο, και
mode: το αν θα το ανοίξουμε για διάβασμα, για γράψιμο ή για πρόσθεση

Η σύνταξη της **close** είναι η ακόλουθη:

fp.close()

όπου:
fp: ο δείκτης στο αρχείο.

12.2 Λειτουργίες εγγραφής

Η εγγραφή σε ένα αρχείο γίνεται με την **write**. Η **write** εφαρμόζεται στον δείκτη του αρχείου και παίρνει σαν παράμετρο μια συμβολοσειρά, την οποία και γράφει στο αρχείο. Η σύνταξή της ακολουθεί:

fp.write(s)

όπου:
fp: ο δείκτης στο αρχείο, και
s: η συμβολοσειρά που θα γραφεί στο αρχείο.

Αν έχουμε μία λίστα από συμβολοσειρές, τότε μπορούμε να χρησιμοποιήσουμε την **writelines** η οποία θα γράψει μία μία όλες τις συμβολοσειρές της

λίστας στο αρχείο, με τη σειρά που αυτές συναντώνται στη λίστα. Η σύνταξή της ακολουθεί:

fp.writelines(L)

όπου:
fp: ο δείκτης στο αρχείο, και
L: η λίστα με τις συμβολοσειρές που θα γραφούν στο αρχείο

```
1  s='This is a test file\n'
2  t='to test python files\n'
3  f=open ('testfile','w')
4  f.write(s)
5  f.write(t)
6  f.close()
7
8  s='This is another test file\n'
9  t='to test python files\n'
10 lst=[]
11 lst.append(s)
12 lst.append(t)
13 f=open ('testfile','a')
14 f.writelines(lst)
15 f.close()
```

Σχήμα 12.1: Παράδειγμα με εγγραφές σε ένα αρχείο.

Στο Σχήμα 12.1 φαίνεται ένα παράδειγμα στο οποίο κάνουμε κάποιες εγγραφές σε ένα αρχείο. Έστω ότι στις μεταβλητές **s** και **t** έχουμε εκχωρημένες τις τιμές

s='This is a test file\n'

και

t='to test python files\n'

οι οποίες και θέλουμε να αποτελέσουν τις δύο πρώτες γραμμές του αρχείου με το όνομα **testfile**. Στην αρχή ανοίγουμε το αρχείο **testfile** για εγγραφή ('**w**') με την **open**. Μετά καλούμε την **f.write()** δύο φορές, τη μία με παράμετρο **s** για να γράψει στην πρώτη γραμμή το **'This is a test file\n'** και τη δεύτερη με την παράμετρο **t** για να γράψει τη γραμμή **'to test python files\n'**. Προσέξτε τα

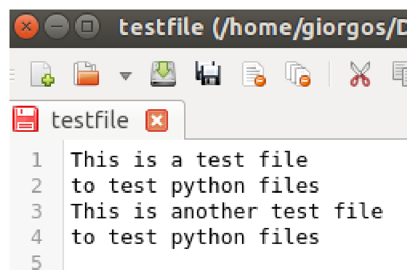
σύμβολα **\n** στο τέλος κάθε συμβολοσειράς, που υποδηλώνουν αλλαγή γραμμής. Αν δεν υπήρχαν, οι δύο φράσεις θα ήταν η μία δίπλα στην άλλη στην ίδια γραμμή του αρχείου.

Στη συνέχεια κλείνουμε το αρχείο εφαρμόζοντας την **close** πάνω στην **f**, δηλαδή γράφοντας **f.close()**.

Ας υποθέσουμε τώρα ότι μια άλλη μέρα θέλουμε να συνεχίσουμε να γράφουμε παρακάτω στο ίδιο αρχείο. Έστω ότι οι δύο γραμμές που θέλουμε να γράψουμε βρίσκονται στη λίστα **lst**.

Θα ανοίξουμε το ίδιο αρχείο, το **testfile**, για πρόσθεση (**'a'**) πάλι με την **open**. Αυτήν τη φορά, όμως, θα καλέσουμε τη **writelines**, μια που όλα είναι έτοιμα μέσα στη λίστα **lst**.

Αφού κλείσουμε το αρχείο, μπορούμε με έναν εκδότη κειμένου να δούμε τι περιέχει μέσα. Θα πάρουμε την εικόνα του Σχήματος 12.2.



Σχήμα 12.2: Το αρχείο που φτιάξαμε σε έναν εκδότη κειμένου.

12.3 Λειτουργίες ανάγνωσης

Οι κυριότερες λειτουργίες ανάγνωσης στην Python είναι οι **read**, **readline** και **readlines**. Καθεμίας η σύνταξη και περιγραφή ακολουθούν:

Η **read** διαβάζει **N** χαρακτήρες από το αρχείο. Αν το **N** παραλειφθεί σαν παράμετρος, τότε διαβάζει όλο το αρχείο. Η σύνταξή της είναι η ακόλουθη:

fp.read(N)

όπου:

fp: ο δείκτης στο αρχείο, και

N: ο αριθμός των χαρακτήρων που θα διαβαστούν

Η **readine** διαβάζει μία γραμμή από το αρχείο. Η σύνταξή της είναι η ακόλουθη:

fp.readline()

όπου:
fp: ο δείκτης στο αρχείο

Η **readines** διαβάζει όλο το αρχείο και το τοποθετεί γραμμή γραμμή σε μία λίστα. Η σύνταξή της είναι η ακόλουθη:

fp.readlines()

όπου:
fp: ο δείκτης στο αρχείο.

```
1  f=open ('testfile','r')
2  print (f.read())
3  f.close
4
5  f=open ('testfile','r')
6  print(f.read(6))
7  print (f.read())
8  f.close
9
10 f=open ('testfile','r')
11 print(f.readline())
12 f.close
13
14 f=open ('testfile','r')
15 print(f.readlines())
16 f.close
```

Σχήμα 12.3: Παράδειγμα αναγνώσεων από ένα αρχείο.

Στο Σχήμα 12.3 χρησιμοποιούμε τις παραπάνω λειτουργίες για να διαβάσουμε την πληροφορία που γράψαμε στο αρχείο **testfile** προηγουμένως. Αρχικά, δοκιμάζουμε την **read**. Ανοίγουμε το αρχείο και καλούμε την print με παράμετρο την **f.read()**. Η **read** θα διαβάσει όλο το αρχείο **f**, δηλαδή το **testfile**, και η **print** θα το τυπώσει στην οθόνη.

Στο δεύτερο παράδειγμα στο ίδιο σχήμα εφαρμόζουμε πρώτα τη **read** στο **f** με παράμετρο 6. Με αυτόν τον τρόπο θα τυπωθούν οι 6 πρώτοι χαρακτήρες του αρχείου, δηλαδή το

This i

Όταν στη συνέχεια καλέσουμε την **read** χωρίς παράμετρο, θα διαβαστεί και θα τυπωθεί ολόκληρο το υπόλοιπο αρχείο, δηλαδή από τον χαρακτήρα **s** και κάτω. Συνολικά θα δούμε τυπωμένο το ακόλουθο:

```
This i
s a test file
to test python files
This is another test file
to test python files
```

Στο τρίτο παράδειγμα χρησιμοποιούμε τη **readline**. Αυτή αναμένουμε να διαβάσει μόνο μία γραμμή. Καλώντας την τυπώνεται μόνο η πρώτη γραμμή του αρχείου, δηλαδή:

```
This is a test file
```

Στο τελευταίο παράδειγμα δοκιμάζουμε τη **readlines**. Καλώντας τη διαβάζεται, γράφεται σε μία λίστα όλο το αρχείο και στη συνέχεια τυπώνεται η λίστα στην οθόνη μέσα από την **print** . Το αποτέλεσμα που θα εμφανιστεί στην οθόνη είναι το:

```
['This is a test file\n', 'to test python files\n', 'This is another test file\n',
'to test python files\n']
```

Το σύμβολο **\n** υποδηλώνει το τέλος μιας γραμμής κειμένου.

12.4 Λειτουργίες επανάληψης σε αρχεία

Ένα από τα σημεία της Python που παρουσιάζει ιδιαίτερο ενδιαφέρον είναι ο τρόπος που υποστηρίζει κάποιες δομές επανάληψης. Θυμίζουμε λίγο τη δομή **for**, η οποία μπορεί να διαπεράσει με λίγο κώδικα όλα τα στοιχεία μιας λίστας. Στο Σχήμα 12.4 φαίνεται ένα παράδειγμα στο οποίο διαπερνάται η λίστα **L** και τυπώνονται όλα τα στοιχεία της που διαιρούνται ακριβώς με το 5.

```
1 for x in L
2   if x%5==0:
3     print(x)
```

Σχήμα 12.4: Παράδειγμα διαπέρασης λίστας με τη for.

Ανάλογες πολύ βολικές δομές υπάρχουν και για τις λειτουργίες στα αρχεία. Στο Σχήμα 12.5 φαίνονται δύο παραδείγματα επανάληψης με τη **read**.

Στο πρώτο διαβάζονται ένας ένας οι χαρακτήρες του αρχείου. Ο πρώτος έξω από τον βρόχο, οι υπόλοιποι μέσα. Κάθε χαρακτήρας τυπώνεται με την **print** , η οποία έχει στο τέλος της το **end="** ώστε να μην αλλάζει γραμμή αφού τυπωθεί ο χαρακτήρας. Ο βρόχος του **while** σταματάει να εκτελείται μόλις το **char** πάρει την τιμή **eof**, στο τέλος αρχείου δηλαδή. Μπορείτε να το διαβάσετε και έτσι: όσο αυτό που διαβάζουμε είναι χαρακτήρας, μένουμε μέσα στον βρόχο. Μόλις διαβαστεί κάτι άλλο (το eof στην περίπτωσή μας), βγαίνουμε από τον βρόχο.

```
1  f=open ('testfile','r')
2  char=f.read(1)
3  while char:
4      print(char,end='')
5      char=f.read(1)
6  f.close
7
8  f=open ('testfile','r')
9  line=f.readline()
10 while line:
11     print(line,end='')
12     line=f.readline()
13 f.close
```

Σχήμα 12.5: Παράδειγμα δομής επανάληψης με while για διάβασμα από αρχείο.

Το ίδιο ακριβώς κάνει και το επόμενο παράδειγμα και με την ίδια λογική, μόνο που χρησιμοποιεί τη **readline**. Εδώ το end=" το χρησιμοποιούμε διότι στο τέλος κάθε γραμμής του αρχείου υπάρχει ένας χαρακτήρας αλλαγής γραμμής. Επειδή το **print** θα προκαλούσε μία αλλαγή γραμμής ακόμα, χρησιμοποιούμε το **end="** για να μη συμβεί αυτό.

Στο Σχήμα 12.6 φαίνονται τα ίδια παραδείγματα, αλλά με τη χρήση της **for**. Παρατηρήστε πόσο πιο όμορφο είναι. Είναι σαν να μιλάμε: "Για κάθε χαρακτήρα στο αρχείο (ή για κάθε γραμμή στο αρχείο) κάνε τα εξής". Στο τελευταίο από αυτά, μάλιστα, δεν χρησιμοποιούμε καθόλου τις κλήσεις **read** και **readlines**, αντίθετα έχουμε το πιο φιλικό:

for line in f:

```
1  f=open ('testfile','r')
2  for char in f.read():
3      print(char,end='')
4  f.close
5
6  f=open ('testfile','r')
7  for line in f.readlines():
8      print(line,end='')
9  f.close
10
11 f=open ('testfile','r')
12 for line in f:
13     print (line,end='')
14 f.close()
```

Σχήμα 12.6: Παράδειγμα δομής επανάληψης με for για διάβασμα από αρχείο.

12.5 Εγγραφή και ανάγνωση ολόκληρων δομών σε δυαδικά αρχεία

Τέλος, θα δούμε πώς είναι δυνατόν να γράψουμε μια ολόκληρη δομή σε ένα αποθηκευτικό μέσο. Θα δούμε εδώ και μία άλλη μορφή αρχείων, τα δυαδικά. Είναι λίγο καταχρηστική η διάκριση σε δυαδικά και μη αρχεία, αφού όλα τα αρχεία τελικά εγγράφονται σε δυαδική μορφή στο αποθηκευτικό μέσο. Αυτό που θέλουμε να δείξουμε εδώ και να διαφοροποιήσουμε από τα αρχεία κειμένου είναι ότι η κωδικοποίηση της πληροφορίας εδώ είναι διαφορετική και δεν είναι δυνατόν να δούμε το περιεχόμενο ενός αρχείου απλά με έναν εκδότη κειμένου.

Για τα αρχεία αυτής της μορφής θα χρησιμοποιήσουμε μία βιβλιοθήκη, την **pickle**. Στο Σχήμα 12.7 φαίνεται το παράδειγμα στο οποίο θα βασιστούμε. Θα γράψουμε ένα αρχείο το οποίο θα έχει μέσα του δύο λίστες, τις οποίες και στη συνέχεια θα διαβάσουμε. Οι λίστες βρίσκονται στις μεταβλητές **x** και **y**. Ανοίγουμε ένα νέο αρχείο, έστω το **testfile**, για εγγραφή. Παρατηρήστε όμως ότι η δεύτερη παράμετρος της **open** είναι **wb** και όχι **w**, όπως αναμενόταν. Αυτό γίνεται για να δηλώσουμε ότι το αρχείο είναι σε δυαδική μορφή.

Στη συνέχεια καλούμε την **pickle.dump** δύο φορές. Τη μία για να γράψουμε τη λίστα **x** και την άλλη για την **y**. Είναι απλό. Χωρίς να ανησυχούμε, η Python θα κωδικοποιήσει και θα τοποθετήσει τις δύο λίστες μέσα στο αρχείο. Κλεί-

νουμε το αρχείο και τελειώσαμε με τις εγγραφές. Αν επιχειρήσουμε να ανοίξουμε το αρχείο με έναν εκδότη κειμένου, θα δούμε κάτι ακαταλαβίστικα σύμβολα στην οθόνη, οπότε μην το κάνετε.

Η διαδικασία ανάγνωσης είναι το ίδιο απλή και φαίνεται στη συνέχεια στο ίδιο σχήμα. Θα ανοίξουμε το αρχείο, πάλι χρησιμοποιώντας το **b**, δηλαδή η δεύτερη παράμετρος της **open** θα είναι τώρα **rb**. Στη συνέχεια θα καλέσουμε την **pickle.load** δύο φορές, με την πρώτη θα διαβαστεί η πρώτη λίστα που αποθηκεύτηκε και με τη δεύτερη η δεύτερη. Τόσο απλό. Κλείνουμε το αρχείο και τελειώσαμε.

```
1  import pickle
2
3  x=[1,2,3,4,5]
4  y=['a','b','c']
5
6  f=open ('testfile','wb')
7  pickle.dump(x,f)
8  pickle.dump(y,f)
9  f.close()
10
11 f=open ('testfile','rb')
12 a=pickle.load(f)
13 b=pickle.load(f)
14 print(a)
15 print(b)
16 f.close()
```

Σχήμα 12.7: Εγγραφή και ανάγνωση σε δυαδικά αρχεία.

Περισσότερο υλικό σχετικό με αναγνώσεις και εγγραφές σε αρχεία μπορείτε να διαβάσετε στο κεφάλαιο 8 του βιβλίου [1], στο κεφάλαιο 14 του βιβλίου [2] και στο κεφάλαιο 13 του βιβλίου [3].

Βιβλιογραφία

1. Jennifer Campel, Paul Gries, Jason Montojo, Greg Wilson (2019). **Practical Programming, An Introduction to Computer Science Using Python**. Publisher: The Pragmatic Bookself.
2. Allen B. Downey (2012). **Think Python**. Publisher: O'Reilly Media.

3. Brian Heinold (2012). **Introduction to Programming Using Python**. Publisher: Mount St. Mary's University, Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.

Κεφάλαιο 13: Φτιάχνοντας παιχνίδια

Σε αυτό το κεφάλαιο θα διασκεδάσουμε λίγο. Όχι ότι περάσαμε άσχημα στα προηγούμενα, αλλά νομίζω ότι εδώ θα περάσουμε ακόμα καλύτερα. Θα συνδυάσουμε ”το τερπνόν μετά του ωφελίμου” και θα φτιάξουμε δύο παιχνίδια: την Κρεμάλα και το Παιχνίδι της Ζωής. Μη βιαστείτε να υποθέσετε ότι είναι δύσκολο. Δεν είναι πολύ εύκολο, αλλά θα δείτε, θα τα καταφέρουμε μια χαρά.

13.1 Κρεμάλα

Το πρώτο παιχνίδι που θα φτιάξουμε είναι η Κρεμάλα. Νομίζω δεν χρειάζεται να πω τους κανόνες του παιχνιδιού, θα μου ήταν απίστευτο αν άκουγα ότι κάποιος δεν το έχει παίξει ποτέ. Στη δική μας έκδοση ο υπολογιστής θα επιλέγει μία λέξη μέσα από ένα σύνολο από διαθέσιμες λέξεις και στη συνέχεια ο παίκτης θα προσπαθεί να μαντέψει τη λέξη αυτή, με τη γνωστή διαδικασία, δίνοντας ένα ένα κάποια γράμματα και έχοντας περιθώριο να κάνει λάθος σε έξι μόνο επιλογές του.

Πιστεύω ότι το να φτιάξουμε την Κρεμάλα θα αποδειχθεί το ίδιο διασκεδαστικό με το παίξιμο του παιχνιδιού, αλλά πολύ περισσότερο ενδιαφέρον. Θα ήταν λάθος να ξεκινούσαμε να γράφουμε κώδικα χωρίς να μελετήσουμε λίγο το πρόβλημα: τι θα θέλουμε να κάνουμε και πώς θα το σχεδιάσουμε. Σίγουρα θα πληρώναμε αυτήν τη βιασύνη μας με βήματα που θα κάναμε προς τα πίσω και που τελικά θα μας κόστιζαν περισσότερο σε χρόνο και σε διάθεση.

Αρχικά, χρειαζόμαστε ένα σύνολο μέσα από το οποίο θα επιλέξουμε τη λέξη που θα ζητήσουμε από τον παίκτη να μαντέψει. Οι λέξεις θα μπορούσαν να βρίσκονται μέσα σε ένα αρχείο ή μέσα σε μία δομή ενσωματωμένη στο πρόγραμμα. Θα χρησιμοποιήσουμε εδώ τη δεύτερη λύση, μία λίστα από συμβολο-

σειρές. Καθεμία από τις συμβολοσειρές αυτές θα είναι και μία πιθανή επιλογή λέξης.

Στη συνέχεια πρέπει να επιλέξουμε τυχαία μία λέξη από τη λίστα. Ο καλύτερος και ευκολότερος τρόπος είναι να αναζητήσουμε μία συνάρτηση από μία έτοιμη βιβλιοθήκη της Python. Ένα από τα μεγαλύτερα πλεονεκτήματα της Python είναι ο πολύ μεγάλος αριθμός από ευέλικτες και αξιόπιστες βιβλιοθήκες οι οποίες είναι διαθέσιμες και ελεύθερες στο διαδίκτυο. Δεν θα δυσκολευτείτε, με τις κατάλληλες λέξεις-κλειδιά σε μία μηχανή αναζήτησης, να φτάσετε στη βιβλιοθήκη **random**.

Εκεί θα βρείτε έτοιμες συναρτήσεις, όπως η **randint(a,b)**, η οποία επιστρέφει έναν ακέραιο αριθμό στο κλειστό διάστημα **[a,b]**, η **random()**, που επιστρέφει έναν πραγματικό αριθμό στο διάστημα **[0,1)**, η **suffle(x)**, η οποία αλλάζει τυχαία τη σειρά των στοιχείων της ακολουθίας **x** (τα ανακατεύει), η **gauss(mu, sigma)**, που επιστρέφει έναν τυχαίο αριθμό με βάση την Γκαουσιανή κατανομή (υπάρχουν συναρτήσεις και για άλλες κατανομές), αλλά και η **choice**, όπως και άλλες πολλές.

Για τη δουλειά που θέλουμε η καταλληλότερη συνάρτηση που μας δίνει η **random** είναι η **choice**. Αυτή δέχεται σαν είσοδο μία ακολουθία (λίστα στην περίπτωσή μας) και επιλέγει και επιστρέφει ένα τυχαίο στοιχείο της. Ας αρχίσουμε να κοιτάμε σιγά σιγά κώδικα και ας δούμε τι σημαίνουν τα παραπάνω σε επίπεδο προγράμματος. Στο Σχήμα 13.1 αρχικά φαίνεται ο κώδικας στον οποίο ορίζεται η λίστα (**lexeis**) με τις λέξεις, και χρησιμοποιείται η **choice**, ώστε να επιλέξει και να τοποθετήσει μία λέξη από τη λίστα στη μεταβλητή **lexi**.

```
1 import random
2
3 lexeis = ['λεμονι','τσαντα','καδρο','ροδα',
4 'βιολι','φως','υπολογιστης','κασετινα','καρεκλα','διακοπτης']
5
6 lexi = random.choice(lexeis)
7 grammata=list(lexi)
8 othoni = ['_']*len(lexi)
```

Σχήμα 13.1: Επιλογή της λέξης και αρχικοποιήσεις.

Στη συνέχεια θα κάνουμε δύο πράγματα. Το πρώτο είναι να σπάσουμε τη λέξη σε γράμματα. Αυτό δεν είναι δύσκολο, αφού η Python μάς παρέχει έτοιμη

τη **list** η οποία κάνει αυτήν ακριβώς τη δουλειά. Αν δεν μας την έδινε, θα έπρεπε να τη φτιάξουμε, δεν είναι δύσκολο. Καλούμε, λοιπόν, τη **list** ως εξής:

grammata=list(lexi)

Αν η **lexi** ήταν βιολί, τότε το **grammata** θα γινόταν ίσο με **['β','ι','ο','λ','ι']**.

Η λίστα **grammata** αποτελεί εσωτερική πληροφορία του προγράμματος. Εννοώ ότι ο χρήστης δεν τη βλέπει (φυσικά). Αυτό που θα δει ο χρήστης είναι μία άλλη πληροφορία που φανερώνει ποια γράμματα έχει ήδη βρει από τη λέξη, ενώ στη θέση αυτών που δεν έχει βρει θα εμφανίζεται μία παύλα. Αρχικά, και αφού ο χρήστης δεν έχει ακόμα προσπαθήσει να βρει κάποιο γράμμα, η πληροφορία που θα του δίνεται θα είναι μια σειρά από παύλες, τόσες όσα τα γράμματα της λέξης. Ας φτιάξουμε, λοιπόν, μία λίστα με τόσες παύλες όσες και τα γράμματα της λέξης. Ας ονομάσουμε τη λίστα αυτή **othoni**. Γράφουμε:

othoni = ['_']*len(lexi)

όπου θυμίζω ότι η συνάρτηση **len** μάς δίνει τον αριθμό των στοιχείων μιας λίστας. Δείτε τον κώδικα που έχουμε γράψει μέχρι στιγμής στο Σχήμα 13.1.

Φτάσαμε τώρα στο σημείο να έχουμε όλες τις λίστες που χρειαζόμαστε για να αναπαραστήσουμε την πληροφορία που θέλουμε. Τώρα πρέπει, σιγά σιγά, να αρχίζουμε να εμφανίζουμε πράγματα στην οθόνη για τον χρήστη. Πρώτα απ' όλα ας εμφανίσουμε τη λίστα **othoni**, τη λίστα που αρχικά έχει τις παύλες, θυμίζω.

```
1 def emfanisi(s):
2     for x in s:
3         print(x,' ',end="")
4     print ()
```

Σχήμα 13.2: Συνάρτηση εμφάνισης λίστας με τα γράμματα ή τις παύλες στην οθόνη.

Επειδή σε περισσότερα από ένα σημεία του προγράμματος θα χρειαστεί να κάνουμε το ίδιο πράγμα, μας συμφέρει να φτιάξουμε μία συνάρτηση που θα κάνει αυτήν τη δουλειά. Τη συνάρτηση αυτή θα την καλέσουμε όπου χρειαστεί να εμφανίσουμε το περιεχόμενο της λίστας **othoni** στον χρήστη. Αν και δεν έχουμε δει τον κώδικα ακόμα παρακάτω, δεν είναι δύσκολο να σκεφτεί κανείς ότι η λίστα **othoni** θα πρέπει να εμφανίζεται στην αρχή στον παίκτη, καθώς και μετά από κάθε προσπάθειά του να μαντέψει κάποιο γράμμα. Αλλά ας μην

τρέχουμε, ας φτιάξουμε πρώτα τη συνάρτηση. Ο κώδικάς της φαίνεται στο Σχήμα 13.2.

Της δίνουμε το όνομα **emfanisi** και σαν παράμετρο μία λίστα, έστω την **s**. Ο κυρίως κώδικάς της αποτελείται από έναν βρόχο **for**, με τον οποίο επισκεπτόμαστε κάθε στοιχείο της λίστας και το εμφανίζουμε στην οθόνη. Το **end=""** στο τέλος της **print** υποδεικνύει ότι, αφού τυπωθεί το **x**, δεν πρέπει να γίνει αλλαγή γραμμής. Έτσι, όλη η λίστα θα εμφανιστεί σε μία γραμμή. Το τελευταίο και μόνο του **print** είναι για να γίνει αλλαγή γραμμής μετά την εμφάνιση ολόκληρης της λίστας.

Στη συνέχεια πρέπει να ζητήσουμε το πρώτο γράμμα από τον χρήστη. Αυτό θα το κάνουμε χρησιμοποιώντας αμυντικό προγραμματισμό. Θα προστατεύσουμε δηλαδή τον χρήστη από εσφαλμένες εισόδους αλλά και το ίδιο το πρόγραμμά μας, αφού είναι υποχρεωμένο να "αντιδράσει" κατάλληλα σε οτιδήποτε του δώσει ο χρήστης, ο οποίος θα μπορούσε να είναι ένα μικρό παιδί. Έτσι, πρέπει να απαγορεύσουμε εισόδους που έχουν περισσότερα από ένα γράμματα, εισόδους που δεν έχουν κανένα γράμμα (αν ο χρήστης πατήσει απλά το **enter**) ή ξένους χαρακτήρες, κεφαλαία γράμματα ή ψηφία, οτιδήποτε δηλαδή δεν είναι μικρό γράμμα του ελληνικού αλφαβήτου. Φυσικά, αν διαγνωστεί μία λάθος είσοδος, θα πρέπει να εμφανίζεται το κατάλληλο μήνυμα και να ζητείται από τον χρήστη να δώσει ένα νέο γράμμα. Ο κώδικας αυτός φαίνεται στο Σχήμα 13.3. και η βασική του δομή είναι ένα **while** μέσα από το οποίο θα βγούμε μόνο με σωστή είσοδο. Σε κάθε λάθος περίπτωση δεν επιτρέπεται η έξοδος από αυτό.

```
1  gramma = input('Δώσε μου ένα γράμμα: ')
2  while len(gramma)>1 or len(gramma)==0 or gramma
3              not in 'αβγδεζηθικλμνξοπρστυφχψως':
4      if len(gramma)>1:
5          print('Δώσε μου το πολύ ένα γράμμα')
6      elif len(gramma)==0:
7          print('Δε μου έδωσες κανένα γράμμα')
8      else:
9          print('Δώσε μου μικρό ελληνικό γράμμα')
       gramma = input('Προσπάθησε πάλι: ')
```

Σχήμα 13.3: Αμυντικός προγραμματισμός για την εισαγωγή ενός μικρού ελληνικού γράμματος.

Όλα αυτά δεν πρέπει να ήταν δύσκολα. Πάμε τώρα να δούμε το κύριο μέρος του αλγορίθμου, για να δούμε ότι ούτε αυτό είναι δύσκολο.

Θα χρησιμοποιήσουμε δύο λογικές μεταβλητές, την **kerdizo** και τη **xano**, τις οποίες θα αρχικοποιήσουμε στο **False**, αφού στην αρχή ούτε έχω κερδίσει ούτε έχω χάσει. Οι μεταβλητές αυτές θα παραμείνουν **False** έως ότου κάποιες συνθήκες μάς δείξουν ότι κερδίσαμε ή ότι χάσαμε, οπότε και η αντίστοιχη μεταβλητή θα γίνει **True**. Πότε θα χάσουμε; Όταν συμπληρωθούν έξι λανθασμένες επιλογές γραμμάτων. Δημιουργούμε, λοιπόν, μία μεταβλητή, ας την ονομάσουμε **lathi**, την οποία αρχικοποιούμε στο μηδέν. Όταν γίνει ίση με 6, η μεταβλητή **xano** θα γίνει **True**. Η μεταβλητή **kerdizo** θα γίνει **True** όταν κανένα στοιχείο της λίστας **othoni** δεν είναι ίσο με '_'.

Ο κύριος βρόχος του προγράμματος εκτελείται όσο οι μεταβλητές **kerdizo** και **xano** είναι ίσες με **True**, όσο δηλαδή ούτε έχουμε κερδίσει ούτε έχουμε χάσει. Μέσα στον βρόχο αυτόν, ζητούμε ένα νέο γράμμα (με αμυντικό προγραμματισμό όπως συζητήσαμε παραπάνω) και εάν το γράμμα υπάρχει στη λέξη, εκτελούμε τον κώδικα του Σχήματος 13.4.

```
1  if gramma in lexi:
2      while gramma in grammata:
3              thesi = grammata.index(gramma)
4              grammata[thesi]='_'
5              othoni[thesi]=gramma
6      if '_' not in othoni:
7          kerdizo=True
8  else:
9      lathi+=1
10     print('Δεν υπάρχει το γράμμα αυτό στη λέξη')
11     if lathi==6:
12         xano=True
13 emfanisi(othoni)
```

Σχήμα 13.4: Μετά την επιτυχή επιλογή γράμματος.

Ο κώδικας αυτός αποτελείται από έναν βρόχο **while**. Ο λόγος που χρειαζόμαστε το **while** είναι ότι περισσότερα από ένα ίδια γράμματα μπορεί να υπάρχουν στη λέξη. Έτσι, για όσο συναντούμε γράμματα ίδια με αυτό που δόθηκε από τον χρήστη, βρίσκουμε τη θέση του γράμματος στη λέξη, σβήνουμε το γράμμα από τη θέση αυτήν από τη λίστα **grammata** (αντικαθιστώντας το με '_' αλλά μπορούσαμε να το αντικαταστήσουμε με οτιδήποτε δεν είναι ελληνικό μικρό γράμμα) και αντικαθιστούμε στην ίδια θέση της λίστας **othoni** την παύλα

με το γράμμα. Αφού αντικαταστήσουμε όλα τα ίδια γράμματα στη λέξη, ελέγχουμε αν η λέξη βρέθηκε, αν δηλαδή δεν έμειναν άλλες παύλες στην **othoni**, και ενημερώνουμε κατάλληλα τη μεταβλητή **kerdizo**.

Εάν, τώρα, το γράμμα δεν υπάρχει στη λέξη, τότε αυξάνουμε τη μεταβλητή **lathi** κατά 1 και ελέγχουμε μήπως φτάσαμε στα 6 λάθη, οπότε και έχουμε χάσει στο παιχνίδι.

Στο τέλος του κυρίως βρόχου, είτε έχουμε χάσει είτε έχουμε κερδίσει, θα ξαναεμφανίσουμε στην οθόνη το περιεχόμενο της λίστας **othoni**. Αν έχουμε βρει τη λέξη, η λέξη θα εμφανιστεί ολόκληρη. Αν όχι, θα εμφανιστεί με τις παύλες στα γράμματα που δεν έχουμε ακόμα βρει.

Όλος ο κώδικας της Κρεμάλας φαίνεται στο Σχήμα 13.5. Το μόνο κομμάτι του κώδικα που δεν έχουμε συζητήσει είναι το τελευταίο **if**. Εκεί φτάνουμε όταν **kerdizo==True** ή **xano==True**. Θα ελέγξουμε για ποιον από τους δύο λόγους φτάσαμε εκεί και θα τυπώσουμε το ανάλογο μήνυμα.

Τελειώσαμε. Προσθέστε περισσότερες λέξεις και ξεκινήστε. Μην παραμελήσετε όμως τη συνέχεια της μελέτης σας.

13.2 Το Παιχνίδι της Ζωής

Φαντάζομαι ότι και αυτό το ξέρετε, αλλά αυτήν τη φορά θα δώσω τους κανόνες. Ένας οργανισμός αποτελείται από μία επιφάνεια με **NxM** κύτταρα. Σε κάθε βήμα του παιχνιδιού υπολογίζουμε τα κύτταρα τα οποία θα υπάρχουν στην επόμενη γενιά. Το αν ένα κύτταρο θα υπάρχει στην επόμενη γενιά εξαρτάται από τον αριθμό των γειτονικών του κυττάρων που έχει ο οργανισμός ζωντανά σε αυτήν τη γενιά. Πιο συγκεκριμένα, οι κανόνες που καθορίζουν αν ένα κύτταρο θα είναι ζωντανό ή όχι στην επόμενη γενιά είναι οι ακόλουθοι:

- κάθε ζωντανό κύτταρο με λιγότερο από δύο ζωντανούς γείτονες πεθαίνει από έλλειψη γειτόνων,
- κάθε ζωντανό κύτταρο με δύο ή τρεις ζωντανούς γείτονες επιβιώνει και στην επόμενη γενιά,
- κάθε ζωντανό κύτταρο με περισσότερους από τρεις ζωντανούς γείτονες πεθαίνει από υπερπληθυσμό,
- κάθε νεκρό κύτταρο με ακριβώς τρεις ζωντανούς γείτονες μετατρέπεται στην επόμενη γενιά σε ζωντανό κύτταρο.

```
import random

def emfanisi(s):
    for x in s:
        print(x,' ',end="")
    print ()

lexeis = ['λεμονι','τσαντα','καδρο','ροδα',
'βιολι','φως','υπολογιστης','κασετινα','καρεκλα','διακοπτης']

lexi = random.choice(lexeis)
grammata=list(lexi)
othoni = ['_']*len(lexi)
emfanisi(othoni)

lathi = 0
xano = False
kerdizo = False
while xano==False and kerdizo==False:
    gramma = input('Δώσε μου ένα γράμμα: ')
    while len(gramma)>1 or len(gramma)==0
                or gramma not in 'αβγδεζηθικλμνξοπρστυφχψως':
        if len(gramma)>1:
            print('Δώσε μου το πολύ ένα γράμμα')
        elif len(gramma)==0:
            print('Δε μου έδωσες κανένα γράμμα')
        else:
            print('Δώσε μου μικρό ελληνικό γράμμα')
        gramma = input('Προσπάθησε πάλι: ')
    if gramma in lexi:
        while gramma in grammata:
                thesi = grammata.index(gramma)
                grammata[thesi]='_'
                othoni[thesi]=gramma
        if '_' not in othoni:
            kerdizo=True
    else:
        lathi+=1
        print('Δεν υπάρχει το γράμμα αυτό στη λέξη')
        if lathi==6:
            xano=True
    emfanisi(othoni)

if xano==True:
    print('Λυπάμαι, έχασες!!!')
    print('Η λέξη ήταν',lexi)
else:
    print('Μπράβο, κέρδισες!!!')
```

Σχήμα 13.5: Ολοκληρωμένο το πρόγραμμα της Κρεμάλας.

Έτσι, ο παίκτης πρέπει να κατασκευάζει έναν αρχικό οργανισμό με κύτταρα και να παρακολουθεί την εξέλιξή του σε κάθε βήμα. Ας δούμε πώς θα το κάνουμε αυτό. Είναι μάλλον δυσκολότερο από την Κρεμάλα, αφού θα χρειαστούμε έναν βρόχο, με ένα **while** το οποίο έχει μέσα του φωλιασμένα τέσσερα **for** και τα οποία έχουν φωλιασμένα μέσα τους τρία **if**. Πάρτε βαθιά ανάσα και ξεκινάμε.

```
1 M=18
2 N=11
3 kyttara=[[' ' for i in range(N)] for j in range(M)]
4 nea_genia=[[' ' for i in range(N)] for j in range(M)]
```

Σχήμα 13.6: Ορισμός των πινάκων που απαιτούνται.

Αρχικά πρέπει να ορίσουμε δύο πίνακες δύο διαστάσεων. Ο πρώτος, ας τον ονομάσουμε **kyttara** και ας θεωρήσουμε ότι θέλουμε να έχει διαστάσεις **MxN**, θα αντιστοιχίζει κάθε στοιχείο του σε ένα κύτταρο και θα έχει την πληροφορία αν είναι ζωντανό ('**X**') ή όχι (' '). Ο δεύτερος πίνακας, ας τον ονομάσουμε **nea_genia**, πρέπει και αυτός να έχει τις ίδιες διαστάσεις **MxN**, και θα χρησιμοποιηθεί ώστε να υπολογιστεί τι μορφή θα έχει ο οργανισμός στην επόμενη γενιά, ποια κύτταρα δηλαδή θα επιβιώσουν και ποια όχι. Ο ορισμός αυτών των πινάκων φαίνεται στο Σχήμα 13.6.

```
1 C=int(input('Με πόσα κύτταρα θέλεις να ξεκινήσεις; '))
2 for i in range(C):
3     print('Κύτταρο ',i+1,':')
4     x=int(input('Οριζόντια θέση:'))
5     x=x-1
6     y=int(input('Κατακόρυφη θέση:'))
7     y=y-1
8     kyttara[x][y]='X'
```

Σχήμα 13.7: Περιγραφή του αρχικού οργανισμού.

Στη συνέχεια θα ζητήσουμε από τον χρήστη να περιγράψει τον αρχικό οργανισμό. Να δώσει δηλαδή τις συντεταγμένες κάθε ζωντανού κυττάρου. Ο κώδικας φαίνεται στο Σχήμα 13.7 και είναι μάλλον εύκολος για να τον σχολιάσουμε στη φάση αυτήν. Παρατηρήστε μόνο την αύξηση και μείωση των δεικτών των πινάκων κατά ένα, έτσι ώστε, όταν γίνεται πρόσβαση στους πίνακες, να θεωρούμε ότι αυτοί ξεκινούν από τη θέση μηδέν, αλλά όταν επικοινωνούμε

με τον χρήστη, να λέμε το κύτταρο στη θέση **1,1** και όχι το κύτταρο στη θέση **0,0** που θα ξένιζε κάποιους.

Θα φτιάξουμε και μία συνάρτηση που να εμφανίζει τον πίνακα με τον οργανισμό στην οθόνη, ας την ονομάσουμε **emfaniseOrganismo**. Οι πρώτες γραμμές που τυπώνουν πολλές φορές μία κενή γραμμή είναι για να καθαρίσει η οθόνη και να υπάρχει μόνο η απεικόνιση του οργανισμού σε αυτήν. Η υπόλοιπη λογική είναι ίδια με αυτήν της Κρεμάλας. Χρειαζόμαστε έναν φωλιασμένο βρόχο για να διατρέξουμε τις γραμμές και τις στήλες. Κάθε κύτταρο τυπώνεται χωρίς αλλαγή γραμμής, αλλά όταν τελειώσει μία σειρά του πίνακα, τότε αλλάζουμε και γραμμή, καλώντας μία κενή **print**. Τυπώνουμε και κάποιες οριζόντιες γραμμές πάνω και κάτω από τον πίνακα, όπως και κάθετες δεξιά και αριστερά, έτσι, απλά για ομορφιά. Στο τέλος βάζουμε μία εντολή **input**, η οποία επιτρέπει τη συνέχιση της εκτέλεσης αφού πατηθεί το πλήκτρο **enter**. Η συνάρτηση ολοκληρωμένη φαίνεται στο Σχήμα 13.8.

```
1  def emfasiseOrganismo():
2      for i in range(30):
3          print()
4      print('-------------')
5      for i in range(M):
6          print('|',end='')
7          for j in range(N):
8              print(kyttara[i][j],end='')
9          print('|')
10     print('-------------')
11     print()
12     print()
13     input('Με Enter συνεχίζουμε στο επόμενο βήμα...')
```

Σχήμα 13.8: Εμφάνιση του οργανισμού στην οθόνη.

Με το που θα ολοκληρωθεί η είσοδος, δηλαδή η περιγραφή του αρχικού οργανισμού, θα πρέπει να τον τυπώσουμε στην οθόνη. Καλούμε, λοιπόν, τη συνάρτηση **emfaniseOrganismo**. Μετά ακολουθεί το κύριο μέρος του προγράμματος, το οποίο είναι μία πολύ μεγάλη φωλιασμένη δομή. Το κύριο έργο της είναι να υπολογίσει την επόμενη γενιά με βάση την τρέχουσα.

Έξω έξω χρειάζεται ένας διπλός βρόχος που μας βοηθάει να επισκεφτούμε ένα ένα τα κύτταρα. Για κάθε κύτταρο πρέπει να μετρήσουμε τα γειτονικά του. Άρα θα αρχικοποιήσουμε σε αυτό το βάθος φωλιάσματος έναν μετρητή, ας τον

ονομάσουμε **counter**, τον οποίο θα αυξάνουμε κατά ένα για κάθε ζωντανό γειτονικό του κύτταρο. Έτσι, πρέπει να επισκεφτούμε ένα ένα όλα τα γειτονικά του κύτταρα. Θέλουμε, λοιπόν, έναν ακόμα διπλό βρόχο, που για κάθε κύτταρο **i,j** θα ξεκινάει από τα κύτταρα **i-1,j-1** και θα φτάσει έως και τα κύτταρα **i+1,j+1** (έως τα **i+2,j+2**, αν μιλάμε σε Python). Προσοχή όμως! Αυτός ο διπλός βρόχος περνάει και από το κύτταρο **i,j**, κάτι που δεν θέλουμε. Άρα με ένα **if** πρέπει να το εξαιρέσουμε. Όπως και με ένα **if** πρέπει να μην προσπαθήσουμε να επισκεφτούμε κύτταρα που δεν υπάρχουν. Για το κύτταρο στη θέση **0,0**, για παράδειγμα, δεν έχει κάποιο νόημα να κοιτάξουμε αν είναι ζωντανό το κύτταρο **-1,-1**, άσε που θα παίρναμε και ένα καθόλου τιμητικό μήνυμα λάθους από την Python.

```
while True:
    for i in range(M):
        for j in range(N):
            counter=0
            for ii in range(i-1,i+2):
                for jj in range(j-1,j+2):
                    if i!=ii or j!=jj:
                        if ii>=0 and ii<M and jj>=0 and jj<N:
                            if kyttara[ii][jj]=='X':
                                counter=counter+1
            if kyttara[i][j]=='X' and counter<2:
                nea_genia[i][j]=' '
            elif kyttara[i][j]=='X' and (counter==2 or counter==3):
                nea_genia[i][j]='X'
            elif kyttara[i][j]=='X' and counter>3:
                nea_genia[i][j]=' '
            elif kyttara[i][j]==' ' and counter==3:
                nea_genia[i][j]='X'
            else:
                nea_genia[i][j]=' '
    for i in range(M):
        for j in range(N):
            kyttara[i][j]=nea_genia[i][j]
    emfasiseOrganismo()
```

Σχήμα 13.9: Υπολογισμός επόμενης γενιάς.

Ο κώδικας που συζητάμε φαίνεται στο Σχήμα 13.9. Οι δείκτες **i,j** χρησιμοποιούνται για την επίσκεψη κάθε κυττάρου, οι δείκτες **ii,jj** για την επίσκεψη

κάθε γειτονικού, το:

if i!=ii or j!=jj

εξαιρεί την επίσκεψη του τρέχοντος κυττάρου, ενώ το:

if ii>=0 and ii<M and jj>=0 and jj<N

επιτρέπει την επίσκεψη μόνο σε γειτονικά κύτταρα που πραγματικά υπάρχουν. Ο μετρητής **counter** αυξάνεται κατά 1 μόνο όταν ο κώδικας έχει περάσει από όλες αυτές τις συνθήκες και έχει διαπιστώσει ότι το κύτταρο αυτό είναι ζωντανό:

if kyttara[ii][jj]=='X': counter=counter+1

Στη συνέχεια, και αφού βρήκαμε τον αριθμό των γειτόνων ενός κυττάρου, πρέπει να εφαρμόσουμε τους τέσσερις κανόνες σε αυτό. Αυτό φαίνεται παρακάτω, στο Σχήμα 13.8, στη δομή με το ένα **if**, τα τρία **elif** και το **else**. Εκεί χωρίζονται οι περιπτώσεις ανάλογα με τον **counter** για τα ζωντανά κύτταρα, προβλέπεται να γεννηθεί ένα κύτταρο με ακριβώς τρία ζωντανά γειτονικά αλλά και να σημειωθούν ως νεκρά όλα τα υπόλοιπα.

Στο τέλος, αντιγράφει τη νέα γενιά από τον πίνακα **nea_genia** στον πίνακα **kyttara**, έτσι ώστε να μπορούμε να περάσουμε στο επόμενο βήμα. Εμφανίζουμε στην οθόνη τον νέο οργανισμό και ολοκληρώσαμε το βήμα αυτό.

Το παιχνίδι είναι φτιαγμένο έτσι ώστε τα βήματα να μην σταματούν ποτέ. Ανάμεσα σε κάθε βήμα ζητείται από τον χρήστη να πατήσει το πλήκτρο **enter**, ώστε να μπορεί να παρακολουθεί την εξέλιξη. Ο κώδικας ολοκληρωμένος φαίνεται στο Σχήμα 13.10. Μπορείτε να τον δοκιμάσετε. Μπορείτε επίσης στο διαδίκτυο να βρείτε αρχικούς οργανισμούς που δίνουν ενδιαφέρουσες εξελίξεις. Έχει πλάκα.

Το Παιχνίδι της Ζωής είναι ενδεικτικό μιας ομάδας εφαρμογών στις οποίες αναζητούμε πληροφορίες από μια γειτονιά κελιών ενός πίνακα. Έτσι, θα μπορούσαμε να γενικεύσουμε και να αναζητήσουμε πληροφορίες, είτε για μια παραλλαγή του παραπάνω παιχνιδιού, είτε για κάποια άλλη εφαρμογή (και υπάρχουν πολλές) από μία ομάδα κελιών μεγέθους **3x3** ή και ακόμα μεγαλύτερη. Επίσης, το πρόβλημα μπορεί να γενικευτεί σε περισσότερες διαστάσεις, όπου οι γείτονες δεν είναι μόνο δεξιά, αριστερά, πάνω και κάτω, αλλά και μπροστά και πίσω (3 διαστάσεις) ή υπάρχουν σχέσεις στον τετραδιάστατο ή ν-διάστατο χώρο που δεν μπορούν να περιγραφούν στον τρισδιάστατο χώρο που είμαστε συνηθισμένοι να αντιλαμβανόμαστε.

```
def emfasiseOrganismo():
    for i in range(30):
        print()
    print('-------------')
    for i in range(M):
        print('|',end='')
        for j in range(N):
            print(kyttara[i][j],end='')
        print('|')
    print('-------------')
    print()
    print()
    input('Με Enter συνεχίζουμε στο επόμενο βήμα...')

M=18
N=11
kyttara=[[' ' for i in range(N)] for j in range(M)]
nea_genia=[[' ' for i in range(N)] for j in range(M)]
C=int(input('Με πόσα κύτταρα θέλεις να ξεκινήσεις; '))
for i in range(C):
    print('Κύτταρο ',i+1,':')
    x=int(input('Οριζόντια θέση:'))
    x=x-1
    y=int(input('Κατακόρυφη θέση:'))
    y=y-1
    kyttara[x][y]='X'
emfasiseOrganismo()
while True:
    for i in range(M):
        for j in range(N):
            counter=0
            for ii in range(i-1,i+2):
                for jj in range(j-1,j+2):
                    if i!=ii or j!=jj:
                        if ii>=0 and ii<M and jj>=0 and jj<N:
                            if kyttara[ii][jj]=='X':
                                counter=counter+1
            if kyttara[i][j]=='X' and counter<2:
                nea_genia[i][j]=' '
            elif kyttara[i][j]=='X' and (counter==2 or counter==3):
                nea_genia[i][j]='X'
            elif kyttara[i][j]=='X' and counter>3:
                nea_genia[i][j]=' '
            elif kyttara[i][j]==' ' and counter==3:
                nea_genia[i][j]='X'
            else:
                nea_genia[i][j]=' '
    for i in range(M):
        for j in range(N):
            kyttara[i][j]=nea_genia[i][j]
    emfasiseOrganismo()
```

Σχήμα 13.10: Ολοκληρωμένο το πρόγραμμα του Παιχνιδιού της Ζωής.

Ασκήσεις που μπορείτε να κάνετε μόνοι σας

- Δοκιμάστε να φτιάξετε την Κρεμάλα έτσι ώστε να διαβάζει τις λέξεις από ένα αρχείο.
- Δοκιμάστε να τροποποιήσετε την Κρεμάλα έτσι ώστε να χρησιμοποιεί τόνους στις λέξεις.
- Δοκιμάστε να τροποποιήσετε την Κρεμάλα έτσι ώστε να χρησιμοποιεί και κεφαλαία γράμματα.
- Ένα ακόμα παιχνίδι το οποίο θα μπορέσετε εύκολα να φτιάξετε συνδυάζοντας ιδέες από την Κρεμάλα και το Παιχνίδι της Ζωής είναι ο Ναρκαλιευτής. Δοκιμάστε το. Χρειάζεστε έναν πίνακα με τις βόμβες ή τον αριθμό των σημείων που γειτονεύουν με βόμβες και έναν πίνακα που κρατάει την πληροφορία ποια τετράγωνα έχουν ήδη ανοιχτεί. Χρησιμοποιήστε μία φωλιασμένη δομή από βρόχους **for** και **if** ανάλογη με αυτή του παιχνιδιού της ζωής για να υπολογίσετε τον πίνακα με τις γειτονικές βόμβες.
- Δοκιμάστε να φτιάξετε μία Τρίλιζα (το παιχνίδι που συμπληρώνουμε **O** και **X** σε ένα σχήμα της μορφής **#**) με πρώτο παίκτη τον χρήστη και δεύτερο παίκτη τον υπολογιστή. Είναι δυνατόν να υλοποιήσετε την πολιτική που θα ακολουθεί ο υπολογιστής ώστε να μην χάνει ποτέ. Αν τα καταφέρετε, θα έχετε φτιάξει ένα πολύ βαρετό παιχνίδι.

Κεφάλαιο 14: Συμβουλές προς έναν νέο προγραμματιστή

Φτάσαμε σιγά σιγά στο τέλος του βιβλίου. Αντί για κάποιον επίλογο σκέφτηκα να συλλέξω κάποια πράγματα που θα ήθελα να πω σε κάποιον ο οποίος αρχίζει σιγά σιγά να νιώθει προγραμματιστής. Κάποια από αυτά προκύπτουν από όσα ειπώθηκαν στα προηγούμενα κεφάλαια. Κάποια όχι, και ψάχνουν μία θέση σε αυτό το βιβλίο, μερικές γραμμές να χωρέσουν, αφού είναι θέματα τα οποία, ναι μεν έχει νόημα να γνωρίζει ένας προγραμματιστής, αποτελούν όμως ολόκληρο κλάδο στην Πληροφορική και διδάσκονται σε σχετικά ακαδημαϊκά μαθήματα.

Έτσι, θεώρησα, χρήσιμο, καλό σαν επίλογο και ενδιαφέρον να σημειώσω σε μία λίστα μερικές απλές και εύπεπτες, ελπίζω, συμβουλές, γραμμένες με απλό τρόπο και χωρίς να επιχειρούν να εμβαθύνουν. Ελπίζω να τις βρείτε χρήσιμες.

- **Καταγράφουμε τις απαιτήσεις μας (system requirement specification)**: Πριν ξεκινήσουμε να σχεδιάζουμε κώδικα, πόσω μάλλον πριν ξεκινήσουμε να γράφουμε κώδικα πρέπει να είμαστε απόλυτα σίγουροι ότι έχουμε αποφασίσει τι ακριβώς θέλουμε να φτιάξουμε. Πρέπει να συνομιλήσουμε διεξοδικά με ανθρώπους που γνωρίζουν το πρόβλημα πολύ καλά, πιο καλά ίσως και από εμάς. Αν, για παράδειγμα, θέλουμε να φτιάξουμε ένα πρόγραμμα που να καταγράφει την πίεση ενός ασθενή και να παρουσιάζει κάποια αποτελέσματα πάνω στις καταγραφές σε έναν γιατρό, μήπως θα ήταν καλύτερα να μιλήσουμε πρώτα με έναν γιατρό; Προσπαθούμε, λοιπόν, να καταλάβουμε, να συγκεντρώσουμε τις ανάγκες μας, και αφού βεβαιωθούμε ότι το κάναμε εξαντλητικά και οργανωμένα, όταν δηλαδή είμαστε σίγουροι ότι ξέρουμε το πρόβλημα που πάμε να λύσουμε, στο

βάθος φυσικά που μας αφορά και μας ενδιαφέρει, τότε μπορούμε να αρχίσουμε να σκεφτόμαστε το επόμενο βήμα.

- **Επιλέγουμε την κατάλληλη γλώσσα προγραμματισμού, περιβάλλον εργασίας, πλαίσιο ανάπτυξης (integrated development environment)**: Αν και στο βιβλίο αυτό επιλέξαμε την Python σαν εργαλείο για να μυηθούμε στον κόσμο του προγραμματισμού, δεν σημαίνει ότι θα τη χρησιμοποιήσουμε σε κάθε περίπτωση και για οποιοδήποτε πρόβλημα. Αν, ας πούμε, θέλουμε να αναπτύξουμε έναν πολύ γρήγορο κώδικα που θα αποτελεί μέρος του λειτουργικού συστήματος, πρέπει να ξεχάσουμε την Python και να αρχίσουμε να κοιτάμε προς τη C. Αν θέλουμε να φτιάξουμε κάποια δικτυακή εφαρμογή, υπάρχουν πολλά έτοιμα πλαίσια που θα μας λύσουν τα χέρια. Σε κάθε περίπτωση, μελετάμε καλά τις επιλογές μας πριν πάρουμε τις αποφάσεις μας, και αφού τις πάρουμε, εξετάζουμε και τα υπάρχοντα περιβάλλοντα ανάπτυξης. Εμείς είδαμε το IDLE στην αρχή του βιβλίου αυτού, αρκετά σύντομα. Ίσως κάποιο άλλο να μας είναι περισσότερο χρήσιμο και να μας κάνει την ανάπτυξη πιο ευχάριστη.

- **Σχεδιάζουμε τον κώδικά μας (code design)**: Η σχεδίαση του κώδικα είναι το ίδιο σημαντική όσο η καταγραφή των απαιτήσεων ή τα περισσότερα από τα βήματα που θα ακολουθήσουν. Λάθος επιλογές στη σχεδίαση ή συνειδητοποίηση σε δεύτερο χρόνο λανθασμένων επιλογών που θα μας αναγκάσουν να αλλάξουμε σημεία του κώδικα τα οποία έχουμε γράψει ή που θα μεγαλώσουν χωρίς λόγο τον κώδικα και θα τον κάνουν πιο δύσχρηστο, θα οδηγήσουν σε αύξηση του χρόνου που χρειάζεται η ανάπτυξη αλλά και του κόστους του λογισμικού, αν βέβαια δεν προγραμματίζουμε για τη δόξα αλλά για τον πραγματικό κόσμο και θέλουμε να ζήσουμε από αυτό. Ένα παράδειγμα κακού σχεδιασμού είναι το να μην παρατηρήσουμε από την αρχή ότι ένας κώδικας μπορεί να γραφεί σαν συνάρτηση και να το καταλάβουμε αρκετά πιο αργά. Θα πρέπει να γράψουμε τη συνάρτηση και να γυρίσουμε πίσω σε όλα τα σημεία που θα μπορούσε να είχε χρησιμοποιηθεί αλλά δεν έγινε. Αν κάποιος σκέφτηκε ότι το τελευταίο μπορεί και να αποφευχθεί, αφού έτσι και αλλιώς ο κώδικας λειτουργεί σωστά, ας δει την επόμενη συμβουλή.

- **Αναπτύσσουμε καλογραμμένο κώδικα (code development)**: Ο χρόνος τον οποίο θα δαπανήσουμε για την ανάπτυξη του κώδικά μας αποτελεί μία επένδυση για τη συνέχεια. Σε μία όχι και πολύ μεγάλη εφαρμογή, ο όγκος του κώδικα που αναπτύσσεται μετριέται σε χιλιάδες γραμμές κώδικα. Αν σαν παράμετρο βάλουμε και τον αριθμό των προγραμματιστών που εργά-

ζονται πάνω στον ίδιο κώδικα, τα πράγματα δυσκολεύουν περισσότερο. Κάνουμε ό,τι μπορούμε για να κρατήσουμε τον κώδικά μας όσο το δυνατόν πιο ευανάγνωστο. Το να γυρίσουμε πίσω και να αντικαταστήσουμε τον κώδικα με τη συνάρτηση που αναφέραμε προηγουμένως είναι το λιγότερο. Δομούμε τον κώδικα έτσι ώστε όλα να βρίσκονται στο σωστό σημείο. Δίνουμε ευανάγνωστα ονόματα στις μεταβλητές και στις συναρτήσεις. Προτιμούμε μια μεταβλητή να την ονομάσουμε acceleration αντί για a. Στοιχίζουμε τον κώδικα. Η Python μάς απαγορεύει να μην στοιχίσουμε τον κώδικα, και πολύ καλά κάνει. Σε άλλες γλώσσες όμως η ευθύνη είναι δική μας. Ο κώδικάς μας πρέπει να διαβάζεται και από άλλους. Για να μην πω ότι πρέπει να διαβάζεται και από εμάς τους ίδιους έναν χρόνο μετά (ή τρεις μέρες μετά).

- **Επαναχρησιμοποιούμε κώδικα (code reusability)**: Δεν είναι απαραίτητο να ανακαλύπτουμε κάθε φορά τον τροχό, ούτε να παιδευόμαστε χωρίς λόγο. Ο κώδικας που θέλουμε να φτιάξουμε μπορεί ήδη να έχει φτιαχτεί, σε ένα σημαντικό τουλάχιστον μέρος του, είτε από εμάς παλαιότερα είτε από άλλους και να βρίσκεται κάπου αναρτημένος στο διαδίκτυο. Αν ο κώδικας είναι δικός μας, ακόμα καλύτερα. Συμφωνήσαμε προηγουμένως ότι θα γράφουμε καλό κώδικα. Δεν θα είναι δύσκολο, λοιπόν, να τον επαναχρησιμοποιήσουμε. Αν δεν είναι δικός μας, πρέπει να τον ελέγξουμε προσεκτικά πριν τον ενσωματώσουμε στον δικό μας. Αν μάλιστα είμαστε πιο τυχεροί, αυτό που ψάχνουμε μπορεί να είναι διαθέσιμο μέσα από κάποια ευρέως χρησιμοποιούμενη βιβλιοθήκη. Στην περίπτωση αυτήν, νιώθουμε περισσότερο ασφαλείς, η χρήση του είναι ευκολότερη και τυποποιημένη. Καλό είναι τον κώδικα που γράφουμε και πιστεύουμε ότι θα επαναχρησιμοποιήσουμε να τον οργανώνουμε σε βιβλιοθήκες, όταν αυτό είναι δυνατόν ή έχει νόημα.

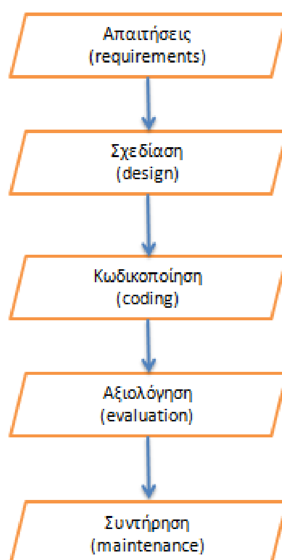
- **Τεκμηριώνουμε τον κώδικα που γράφουμε (code documentation)**: Με άλλα λόγια, του βάζουμε σχόλια. Όσο πιο αναλυτικά, τόσο ευκολότερα θα βρούμε κάποιο σφάλμα μέσα σε αυτόν, τόσο ευκολότερα θα τον καταλάβουμε και θα τον τροποποιήσουμε αν χρειαστεί να τον επαναχρησιμοποιήσουμε στο μέλλον και τόσο πιο εύκολα θα τον κατανοήσει ένας τρίτος, στενός ή από απόσταση συνεργάτης που θα χρειαστεί να το κάνει. Μην υποτιμάτε την ανάγκη της τεκμηρίωσης του κώδικα. Ο χρόνος που δαπανούμε σε αυτό είναι επένδυση για τη συνέχεια.

- **Αποσφαλμάτωνουμε τον κώδικα με επιμονή (code debugging)**: Η αποσφαλμάτωση είναι από τα πιο κουραστικά μέρη της ανάπτυξης. Το να

βρεις και να διορθώσεις όλα τα λάθη του κώδικα απαιτεί συστηματική και προσεκτική εργασία. Ο κώδικάς μας πιθανόν να δοθεί για χρήση σε ένα ευρύ κοινό, μη σχετικό με τη δική μας τεχνογνωσία. Πρέπει να εξαλειφθεί κάθε πιθανότητα οι χρήστες να έρθουν αντιμέτωποι με μία αναπάντεχη κατάσταση. Το κόστος και ο χρόνος της αποσφαλμάτωσης είναι ένα πολύ σημαντικό μέρος της συνολικής ανάπτυξης.

- **Αξιολογούμε και επιστρέφουμε για να διορθώσουμε τις αστοχίες μας (code evaluation)**: Το τελευταίο πράγμα που πρέπει να κάνουμε είναι να αξιολογήσουμε τον κώδικά μας. Φυσικά, αν μέσα από αυτήν τη διαδικασία προκύψουν κάποιες αστοχίες μας πρέπει να επιστρέψουμε και να τις διορθώσουμε. Στη διαδικασία της αξιολόγησης πρέπει να πάρει μέρος και ο ειδικός, αυτός δηλαδή που μας βοήθησε στην πρώτη φάση να συλλέξουμε τις απαιτήσεις μας.

Κάπου εδώ τελειώνουν οι γρήγορες συμβουλές. Μπορείτε να ανατρέξετε σε άλλα βιβλία ή ακαδημαϊκά μαθήματα, ώστε να δείτε όλα τα παραπάνω, όχι σαν συμβουλές, αλλά σαν μεθοδολογία και με τρόπο συστηματικό και επιστημονικό.



Σχήμα 14.1: Βασικές φάσεις ανάπτυξης λογισμικού.

Ας παραθέσουμε για πληρότητα ένα διάγραμμα (Σχήμα 14.1) που δείχνει τις βασικές φάσεις ανάπτυξης του λογισμικού, μια που οι παραπάνω συμβου-

λές το έχουν λάβει υπόψη τους. Στο διάγραμμα αυτό οι φάσεις δεν επικαλύπτονται και η μία ακολουθεί την άλλη. Φυσικά, αν από μία φάση (για παράδειγμα την αξιολόγηση) προκύψει ότι πρέπει να επιστρέψουμε σε κάποια άλλη φάση και να διορθώσουμε τις επιλογές ή τις παραλείψεις μας, αυτό μπορεί να γίνει. Η φυσιολογική ροή των φάσεων είναι: συγκέντρωση απαιτήσεων, σχεδίαση λογισμικού, υλοποίηση λογισμικού, αξιολόγηση λογισμικού και, τέλος, συντήρηση λογισμικού.

Σας ευχαριστώ που διαβάσατε μέρος του ή ολόκληρο αυτό το βιβλίο. Ελπίζω να σας φάνηκε χρήσιμο. Εύχομαι σε όλους καλό ταξίδι στον μαγευτικό κόσμο της Πληροφορικής που έχετε μπροστά σας.

Βιβλιογραφία

Παρακάτω μπορείτε να βρείτε μία σειρά από βιβλία, έντυπα και ηλεκτρονικά, ελληνικά, ξενόγλωσσα ή μεταφρασμένα. Κάποια από αυτά βρίσκονται ελεύθερα στο διαδίκτυο. Προτείνονται ως βιβλία τα οποία μπορείτε να χρησιμοποιήσετε σαν συμπλήρωμα ή συνέχεια της μελέτης σας. Πρόκειται για μία σειρά από αξιόλογα βιβλία τα οποία σχετίζονται με την Python, τη διδασκαλία του προγραμματισμού ή και τα δύο. Η λίστα έχει στοιχειοθετηθεί με αλφαβητική σειρά των συγγραφέων.

- Jennifer Campel, Paul Gries, Jason Montojo, Greg Wilson (2019). **Practical Programming, An Introduction to Computer Science Using Python**. Publisher: The Pragmatic Bookself.
- Allen B. Downey (2012). **Think Python**. Publisher: O'Reilly Media.
- John Guttag (2015). **Υπολογισμοί και Προγραμματισμός Με Την Python**. Μετάφραση: Παναγιώτης Καναβός, Επιμέλεια: Γεώργιος Μανής, Εκδόσεις Κλειδάριθμος.
- Brian Heinold (2012). **Introduction to Programming Using Python**. Publisher: Mount St. Mary's University, Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.
- Ellis Horowitz (1993). **Βασικές Αρχές Γλωσσών Προγραμματισμού**. 2η έκδοση, Εκδόσεις Κλειδάριθμος.
- Cody Jackson (2011). **Learning to Program Using Python**. Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο.
- Αχιλλέας Καμέας (2000). **Τεχνικές Προγραμματισμού**. Τόμος Β. ΠΛΗ-10, Ελληνικό Ανοικτό Πανεπιστήμιο.
- Brian Kernighan, Dennis Ritchie (2011). **Η Γλώσσα Προγραμματισμού C**. Μετάφραση: Θωμάς Μωραΐτης, Επιμέλεια Μετάφρασης: 2η Έκδοση, Εκδόσεις Κλειδάριθμος.

- Eric Roberts. (2004). **Η Τέχνη και Επιστήμη της C**. Μετάφραση: Γιώργος Στεφανίδης, Παναγιώτης Σταυρόπουλος, Αλέξανδρος Χατζηγεωργίου, Εκδόσεις Κλειδάριθμός.
- C. H. Swaroop (2015). **A Byte of Python**. Ηλεκτρονικό βιβλίο, ελεύθερα διαθέσιμο, Μετάφραση: ubuntu-gr.org Team.

Στο διαδίκτυο μπορείτε να βρείτε πολύ υλικό που σχετίζεται με την Python. Αναφέρονται ενδεικτικά οι παρακάτω διευθύνσεις ιστοσελίδων:

- **Welcome to Python.org**, `https://www.python.org`
- **Ελληνική Κοινότητα Προγραμματιστών Python**, `http://python.org.gr`
- **Python - Βικιπαίδεια**, `https://el.wikipedia.org/wiki/Python`
- **Python (programming language)** - Wikipedia,
  `https://en.wikipedia.org/wiki/Python_(programming_language)`
- **Learn Python - Free Interactive Tutorial**, `http://www.learnpython.org`

Ευρετήριο όρων

Από Ελληνικά στα Αγγλικά:

ακολουθία	sequence
αλγόριθμος	algorithm
αλφαριθμητικό	string
αναδρομή	recursion
αναδρομικές ακολουθίες	recursive sequences
αναζήτηση	search
ανάπτυξη κώδικα	code development
αντικείμενο	object
αντικειμενοστραφής	object oriented
αξιολόγηση κώδικα	code evaluation
αποσφαλμάτωση κώδικα	code debugging
αρχείο	file
αυτοσυσχέτιση	autocorrelation
βήμα	step
βρόχος	loop
γλώσσα μηχανής	machine language
γλώσσα παράλληλου προγραμματισμού	parallel programming language
γλώσσα προγραμματισμού	programming languages
γλώσσα προγραμματισμού υψηλού επιπέδου	high level programming language
γρήγορη ταξινόμηση	quick sort

δέντρο	tree
διάγραμμα ροής προγράμματος	flow chart
διαδικασία	procedure
διαδικασιακός προγραμματισμός	procedural programming
διερμηνεύσιμη	interpreted
διερμηνευτής	interpreter
δομή δεδομένων	data structure
δομημένος προγραμματισμός	structured programming
δυαδική αναζήτηση	binary search
εγγραφή	record
επαναχρησιμοποίηση κώδικα	code reusability
επιστήμη της Πληροφορικής	computer science
εργαλείο αποσφαλμάτωσης	debugger
ετεροσυσχέτηση	cross-correlation
κλάση	class
κληρονομικότητα	inheritance
κόσκινο του Ερατοσθένη	sieve of Eratosthenes
κωδικοποίηση	coding
λεξικά	dictionaries
λίστα	list
λογικό διάγραμμα	flowchart
λογικό λάθος	logic errors
λογισμικό ανοικτού κώδικα	open source software
μέθοδος	methods
μεταβλητή	variable
μεταβλητή επανάληψης	iteration variable
μεταγλωττιστής	compiler
μη προσημασμένος	unsinged
μνήμη μόνο ανάγνωσης	Read Only Memory (ROM)
μνήμη τυχαίας προσπέλασης	Read Access Memory (RAM)
μονοδιάστατος πίνακας	one dimensional array
ολοκληρωμένο περιβάλλον ανάπτυξης	integrated development environment
ορισμός απαιτήσεων συστήματος	system requirement specification
παράλληλες αρχιτεκτονικές	parallel architectures
πέρασμα παραμέτρων με αναφορά	call by reference
πέρασμα παραμέτρων με τιμή	call by value

πίνακας	array
πλειάδα	tuple
πολυδιάστατος πίνακας	multi-dimensional array
πολυπλοκότητα	complexity
προσήμασμένος	singed
σειριακή αναζήτηση	serial search
σταθερά	constant
συμβολοσειρά	string
συμπλήρωμα ως προς 2	complement of 2
συνάρτηση	function
συνέλιξη	convolution
σύνολο	set
συντακτικό λάθος	syntax errors
συσχέτιση	correlation
σφάλμα	bug
σχεδίαση κώδικα	code design
ταξινόμηση	sort
ταξινόμηση με επιλογή	selection sort
ταξινόμηση με συγχώνευση	merge sort
ταξινόμηση με φυσαλίδα	bubble sort
ταξινόμηση τη χρήση κάδων	bucket sort
τεκμηρίωση κώδικα	code documentation
τεμαχισμός	slicing
τεχνολογία λογισμικού	software engineering
φωλιασμένος βρόχος	nested loops
χρήστης	user

Από Αγγλικά στα Ελληνικά:

algorithm	αλγόριθμος
array	πίνακας
autocorrelation	αυτοσυσχέτιση
binary search	δυαδική αναζήτηση
bubble sort	ταξινόμηση με φυσαλίδα
bucket sort	ταξινόμηση τη χρήση κάδων
bug	σφάλμα

call by reference	πέρασμα παραμέτρων με αναφορά
call by value	πέρασμα παραμέτρων με τιμή
class	κλάση
code debugging	αποσφαλμάτωση κώδικα
code design	σχεδίαση κώδικα
code development	ανάπτυξη κώδικα
code documentation	τεκμηρίωση κώδικα
code evaluation	αξιολόγηση κώδικα
code reusability	επαναχρησιμοποίηση κώδικα
coding	κωδικοποίηση
compiler	μεταγλωττιστής
complement of 2	συμπλήρωμα ως προς 2
complexity	πολυπλοκότητα
computer science	επιστήμη της Πληροφορικής
constant	σταθερά
convolution	συνέλιξη
correlation	συσχέτιση
cross-correlation	ετεροσυσχέτηση
data structure	δομή δεδομένων
debugger	εργαλείο αποσφαλμάτωσης
dictionaries	λεξικά
file	αρχείο
flow chart	διάγραμμα ροής προγράμματος
flowchart	λογικό διάγραμμα
function	συνάρτηση
high level programming language	γλώσσα προγραμματισμού υψηλού επιπέδου
inheritance	κληρονομικότητα
integrated development environment	ολοκληρωμένο περιβάλλον ανάπτυξης
interpreted	διερμηνεύσιμη
interpreter	διερμηνευτής
iteration variable	μεταβλητή επανάληψης
logic errors	λογικό λάθος
loop	βρόχος
list	λίστα

machine language	γλώσσα μηχανής
merge sort	ταξινόμηση με συγχώνευση
methods	μέθοδος
multi-dimensional array	πολυδιάστατος πίνακας
nested loops	φωλιασμένος βρόχος
object	αντικείμενο
object oriented	αντικειμενοστραφής
one dimensional array	μονοδιάστατος πίνακας
open source software	λογισμικό ανοικτού κώδικα
parallel architectures	παράλληλες αρχιτεκτονικές
parallel programming language	γλώσσα παράλληλου προγραμματισμού
procedural programming	διαδικασιακός προγραμματισμός
procedure	διαδικασία
programming languages	γλώσσα προγραμματισμού
quick sort	γρήγορη ταξινόμηση
Read Access Memory (RAM)	μνήμη τυχαίας προσπέλασης
Read Only Memory (ROM)	μνήμη μόνο ανάγνωσης
record	εγγραφή
recursion	αναδρομή
recursive sequences	αναδρομικές ακολουθίες
search	αναζήτηση
selection sort	ταξινόμηση με επιλογή
sequence	ακολουθία
serial search	σειριακή αναζήτηση
set	σύνολο
sieve of Eratosthenes	κόσκινο του Ερατοσθένη
singed	προσήμασμένος
slicing	τεμαχισμός
software engineering	τεχνολογία λογισμικού
sort	ταξινόμηση
step	βήμα
string	αλφαριθμητικό
string	συμβολοσειρά
structured programming	δομημένος προγραμματισμός
syntax errors	συντακτικό λάθος
system requirement specification	ορισμός απαιτήσεων συστήματος

tree	δέντρο
tuple	πλειάδα
unsinged	μη προσημασμένος
user	χρήστης
variable	μεταβλητή

Σχήματα, Πίνακες και Ταινίες

Κεφάλαιο: 1 11

Ταινία 1.1: Οπτικό καλοσώρισμα. 17

Κεφάλαιο: 2 19

Σχήμα 2.1: Πρόσθεση δύο αριθμών. 23

Ταινία 2.1: Εγκατάσταση του IDLE. 25

Σχήμα 2.2: Ο διερμηνευτής της Python. 26

Σχήμα 2.3: Παράδειγμα χρήσης του διερμηνευτή. 26

Σχήμα 2.4: Ο εκδότης κειμένου του IDLE. 27

Σχήμα 2.5: Το πρόγραμμα της άθροισης στον εκδότη κειμένου. . . . 27

Σχήμα 2.6: Η εκτέλεση της άθροισης στον διερμηνευτή. 28

Σχήμα 2.7: Ένα παράδειγμα συντακτικού λάθους. 28

Ταινία 2.2: Το περιβάλλον της Python. 28

Κεφάλαιο: 3 31

Σχήμα 3.1: Τα βασικότερα σύμβολα που χρησιμοποιούνται στα διαγράμματα ροής. 32

Σχήμα 3.2: Η είσοδος (α) και η έξοδος (β) του προγράμματος της πρόσθεσης. 34

Σχήμα 3.3: Ολοκληρωμένο το διάγραμμα ροής της πρόσθεσης. . . . 35

Σχήμα 3.4: Διάγραμμα ροής για την επίλυση του τριωνύμου. 39

Ταινία 3.1: Επίλυση του τριωνύμου. 39

Σχήμα 3.5 Διάγραμμα ροής για την εύρεση του μέγιστου τριών αριθμών. 40

Σχήμα 3.6: Διάγραμμα ροής για την εύρεση του παραγοντικού 42

Σχήμα 3.7: Έλεγχος ορθής εισόδου για το διάγραμμα του παραγοντικού. 42

Σχήμα 3.8: Διαγράμματα ροής για άσκηση 45

Κεφάλαιο: 4 **47**

Σχήμα 4.1: Παράδειγμα αποθήκευσης μεταβλητών (γλώσσα C). . . . 48
Σχήμα 4.2: Ακέραιες μεταβλητές. 50
Σχήμα 4.3: Χώρος που καταλαμβάνουν στη μνήμη ακέραιοι αριθμοί διαφορετικού μεγέθους. 51
Σχήμα 4.4: Παράδειγμα συμβολοσειρών. 51
Σχήμα 4.5: Παράδειγμα λογικών και μιγαδικών μεταβλητών. 52
Σχήμα 4.6: Παραδείγματα αριθμητικών εκφράσεων. 54
Σχήμα 4.7: Παραδείγματα εκφράσεων με αλφαριθμητικά. 55
Σχήμα 4.8: Παραδείγματα εκφράσεων με μιγαδικούς αριθμούς. . . . 55
Σχήμα 4.9: Παραδείγματα χρήσης της input. 58
Σχήμα 4.10: Παράδειγμα χρήσης της print. 59

Κεφάλαιο: 5 **61**

Σχήμα 5.1: Παράδειγμα εγγραφών στην Pascal. 62
Σχήμα 5.2: Παραδείγματα με πλειάδες. 63
Σχήμα 5.3: Παραδείγματα με αλφαριθμητικά. 65
Σχήμα 5.4: Βασικές λειτουργίες σε μία λίστα. 66
Σχήμα 5.5: Μέθοδοι σε λίστες. 67
Σχήμα 5.6: Μέθοδοι σε λίστες (συνέχεια). 69
Σχήμα 5.7: Παραδείγματα με σύνολα. 70
Σχήμα 5.8: Πράξεις στα σύνολα. 71
Σχήμα 5.9: Προσθαφαίρεση στοιχείων σε σύνολα. 72
Σχήμα 5.10: Παραδείγματα τεμαχισμού ακολουθιών. 74
Σχήμα 5.11: Βήμα στον τεμαχισμό. 75
Σχήμα 5.12: Αναφορά σε στοιχείο της ακολουθίας μετρώντας από το τέλος. 75
Σχήμα 5.13: Αντικατάσταση και εισαγωγή σε ακολουθία. 76
Σχήμα 5.14: Παραδείγματα με λεξικά. 77
Σχήμα 5.15: Λεξικά μέσα σε λεξικά. 78
Πίνακας 5.1: Συνοπτικός πίνακας λειτουργιών σε αλφαριθμητικά. . . 79
Πίνακας 5.2: Συνοπτικός πίνακας λειτουργιών σε λίστες. 80
Πίνακας 5.3: Συνοπτικός πίνακας λειτουργιών σε σύνολα. 81
Πίνακας 5.4: Συνοπτικός πίνακας λειτουργιών σε λεξικά. 81

Κεφάλαιο: 6 **83**

Σχήμα 6.1: Ακολουθιακές εντολές σε ένα ρομπότ. 84
Σχήμα 6.2: Παράδειγμα ακολουθίας εντολών. 84
Σχήμα 6.3: Παράδειγμα στοίχισης εντολών στην if. 86
Σχήμα 6.4: Πρόσημο αριθμού. 86
Σχήμα 6.5: Αριθμός σε κλειστό διάστημα. 87
Σχήμα 6.6: Αριθμός σε κλειστό διάστημα, έκδοση με else. 88
Σχήμα 6.7: Πρόσημο αριθμού, έκδοση με if-elif-else. 89
Σχήμα 6.8: Πρόσημο αριθμού, έκδοση με φωλιασμένα if-else. 89
Σχήμα 6.9: Ο κώδικας που υλοποιεί το τριώνυμο. 90
Σχήμα 6.10: Παράδειγμα με τη δομή switch. 93
Σχήμα 6.11: Ισοδύναμο σε Python με τη δομή πολλαπλής επιλογής. . 93
Σχήμα 6.12: Παραδείγματα με τη δομή for. 94
Σχήμα 6.13: Σχηματική παράσταση για το εσωτερικό γινόμενο δύο διανυσμάτων. 95
Σχήμα 6.14: Κώδικας για το εσωτερικό γινόμενο δύο διανυσμάτων. . 96
Ταινία 6.1: Λίστα περιττών και άρτιων. 96
Σχήμα 6.15: Διαχωρισμός μιας λίστας ακέραιων σε λίστες άρτιων και περιττών. 97
Ταινία 6.2: Προπαίδεια. 97
Σχήμα 6.16: Η προπαίδεια. 97
Σχήμα 6.17: Παραδείγματα με τη while. 98
Σχήμα 6.18: Εύρεση ρίζας πολυωνύμου με τη μέθοδο της διχοτόμησης. 99
Σχήμα 6.19: Διάσπαση ενός αριθμού στα ψηφία του. 101
Ταινία 6.3: Διάσπαση αριθμού στα ψηφία του. 101
Σχήμα 6.20: Αμυντικός προγραμματισμός με τη χρήση της while. . . 101
Σχήμα 6.21: Αμυντικός προγραμματισμός με τη χρήση της do while. 102

Κεφάλαιο: 7 **104**

Σχήμα 7.1: Υλοποίηση της univ και της sum10. 106
Σχήμα 7.2: Λίστα με τους ακέραιους αριθμούς από το 1 μέχρι το 100 οι οποίοι διαιρούνται με το 2 και το 3 αλλά όχι με το 5. 107
Σχήμα 7.3: Παράδειγμα με τέλεια τετράγωνα. 108
Σχήμα 7.4: Παράδειγμα περάσματος παραμέτρου. 110
Σχήμα 7.5: Απόλυτη τιμή. 111
Σχήμα 7.6: Παραγοντικό. 112
Σχήμα 7.7: Ύψωση σε δύναμη. 113
Σχήμα 7.8: Λίστα θετικών και αρνητικών αριθμών. 114

Σχήμα 7.9: Λίστα από λίστες. 115
Ταινία 7.1: Εναλλαγή των τιμών δύο μεταβλητών. 115
Σχήμα 7.10: Εναλλαγή των τιμών δύο μεταβλητών. 116
Σχήμα 7.11: Φωνήεντα σε λίστα από συμβολοσειρές. 116
Ταινία 7.2: Φωνήεντα σε λίστα από συμβολοσειρές. 116
Σχήμα 7.12: Απόσταση από την αρχή των αξόνων. 117
Ταινία 7.3: Απόσταση από την αρχή των αξόνων. 117
Σχήμα 7.13: Πέρασμα παραμέτρου με τιμή. 118
Σχήμα 7.14: Διαδικασία εναλλαγής τιμών σε Pascal. 120
Σχήμα 7.15: Χρήση ονομάτων για τις παραμέτρους. 121
Σχήμα 7.16: Προκαθορισμένη αρχική τιμή και μεταβλητός αριθμός παραμέτρων. 123
Σχήμα 7.17: Προκαθορισμένη αρχική τιμή και θέση παραμέτρων. . . 123

Κεφάλαιο: 8 **125**

Σχήμα 8.1: Πρώτοι αριθμοί - έκδοση 1. 126
Σχήμα 8.2: Πρώτοι αριθμοί - έκδοση 2. 127
Σχήμα 8.3: Πρώτοι αριθμοί - έκδοση 3. 128
Σχήμα 8.4: Πρώτοι αριθμοί - έκδοση 4. 128
Σχήμα 8.5: Πρώτοι αριθμοί - έκδοση 5. 129
Σχήμα 8.6: Πρώτοι αριθμοί - έκδοση 6. 129
Σχήμα 8.7: Πρώτοι αριθμοί - έκδοση 7. 130
Σχήμα 8.8: Πρώτοι αριθμοί - έκδοση 8. 130
Σχήμα 8.9: Πρώτοι αριθμοί - έκδοση 9. 131
Σχήμα 8.10: Πρώτοι αριθμοί - έκδοση 10. 131
Πίνακας 8.1: Αριθμός φορών που χρησιμοποιήθηκε το υπόλοιπο της διαίρεσης σε κάθε έκδοση του αλγορίθμου. 132
Σχήμα 8.11: Το ”κόσκινο του Ερατοσθένη”. 133
Σχήμα 8.12: Μετατροπή δεκαδικού σε δυαδικό. 135
Σχήμα 8.13: Μετατροπή δυαδικού σε δεκαδικό. 136
Σχήμα 8.14: Υπολογισμός συμπληρώματος ως προς 2. 137

Κεφάλαιο: 9 **139**

Σχήμα 9.1: Συνάρτηση που υπολογίζει το παραγοντικό ενός αριθμού. 143
Σχήμα 9.2: Ολοκληρωμένη συνάρτηση για το παραγοντικό. 143
Ταινία 9.1: Ύψωση αριθμού σε δύναμη. 144
Σχήμα 9.3: Ύψωση αριθμού σε δύναμη. 144
Σχήμα 9.4: Αριθμοί Fibonacci με χρήση λίστας. 145

Σχήμα 9.5: Αριθμοί Fibonacci με τη χρήση τριών μεταβλητών. 146
Σχήμα 9.6: Αριθμοί Fibonacci με τη χρήση δύο μεταβλητών. 146
Σχήμα 9.7: Αναδρομικός υπολογισμός της ακολουθίας Fibonacci. . . 147
Ταινία 9.2: Εκτέλεση μιας αναδρομικής συνάρτησης βήμα προς βήμα. 148

Κεφάλαιο: 10 **151**
Σχήμα 10.1: Σειριακή αναζήτηση. 153
Ταινία 10.1: Παράδειγμα εκτέλεσης του αλγορίθμου σειριακής αναζήτησης. 153
Σχήμα 10.2: Σειριακή αναζήτηση σε ταξινομημένο πίνακα. 154
Ταινία 10.2: Παράδειγμα εκτέλεσης του αλγορίθμου της σειριακής αναζήτησης σε ταξινομημένο πίνακα. 155
Ταινία 10.3: Παράδειγμα εκτέλεσης του αλγορίθμου δυαδικής αναζήτησης. 156
Σχήμα 10.3: Αλγόριθμος δυαδικής αναζήτησης. 156
Σχήμα 10.4: Ταξινόμηση φυσαλίδας. 158
Ταινία 10.4: Παράδειγμα εκτέλεσης του αλγορίθμου ταξινόμησης φυσαλίδας. 159
Σχήμα 10.5: Βελτιωμένος αλγόριθμος ταξινόμησης φυσαλίδας. . . . 159
Ταινία 10.5: Παράδειγμα εκτέλεσης του βελτιωμένου αλγορίθμου φυσαλίδας. 160
Σχήμα 10.6: Αλγόριθμος ταξινόμησης με επιλογή. 161
Σχήμα 10.7: Αλγόριθμος ταξινόμησης με συγχώνευση. 162
Ταινία 10.6: Παράδειγμα εκτέλεσης του αλγορίθμου ταξινόμησης με συγχώνευση. 163
Σχήμα 10.8: Αλγόριθμος ταξινόμησης με χρήση κάδων. 163
Ταινία 10.7: Παράδειγμα εκτέλεσης αλγορίθμου ταξινόμησης με τη χρήση κάδων. 164
Σχήμα 10.9: Αλγόριθμος γρήγορης ταξινόμησης. 165
Ταινία 10.8: Παράδειγμα εκτέλεσης αλγορίθμου γρήγορης ταξινόμησης. 165

Κεφάλαιο: 11 **167**
Σχήμα 11.1: Πράξεις με μονοδιάστατους πίνακες. 170
Σχήμα 11.2: Έλεγχος αν όλα τα στοιχεία ενός πίνακα είναι ακέραιοι. 170
Σχήμα 11.3: Πρόσθεση δισδιάστατων πινάκων. 171
Σχήμα 11.4: Πολλαπλασιασμός πινάκων. 173
Σχήμα 11.5: Αναστροφή πίνακα. 174

Σχήμα 11.6: Αναστροφή ενός πίνακα επί τόπου. 174
Σχήμα 11.7: Άλλες συναρτήσεις με πίνακες. 175
Ταινία 11.1: Έλεγχος αν ένας πίνακας είναι μοναδιαίος. 176
Ταινία 11.2: Έλεγχος αν ένας πίνακας είναι συμμετρικός. 176
Σχήμα 11.8: Υπολογισμός συσχέτισης. 177
Σχήμα 11.9: Υπολογισμός συσχέτισης (συνέχεια). 178
Σχήμα 11.10: Συνάρτηση υπολογισμού εσωτερικού γινομένου. 178
Σχήμα 11.11: Η συνάρτηση της συσχέτισης. 179
Σχήμα 11.12: Η συνάρτηση της αυτοσυσχέτισης. 180
Σχήμα 11.13: Παράδειγμα υπολογισμού της συνέλιξης. 181
Σχήμα 11.14: Η συνάρτηση της συνέλιξης. 182

Κεφάλαιο: 12 **184**
Σχήμα 12.1: Παράδειγμα με εγγραφές σε ένα αρχείο. 187
Σχήμα 12.2: Το αρχείο που φτιάξαμε σε έναν εκδότη κειμένου. . . . 188
Σχήμα 12.3: Παράδειγμα αναγνώσεων από ένα αρχείο. 189
Σχήμα 12.4: Παράδειγμα διαπέρασης λίστας με τη for. 190
Σχήμα 12.5: Παράδειγμα δομής επανάληψης με while για διάβασμα από αρχείο. 191
Σχήμα 12.6: Παράδειγμα δομής επανάληψης με **for** για διάβασμα από αρχείο. 192
Σχήμα 12.7: Εγγραφή και ανάγνωση σε δυαδικά αρχεία. 193

Κεφάλαιο: 13 **195**
Σχήμα 13.1: Επιλογή της λέξης και αρχικοποιήσεις. 196
Σχήμα 13.2: Συνάρτηση εμφάνισης λίστας με τα γράμματα ή τις παύλες στην οθόνη. 197
Σχήμα 13.3: Αμυντικός προγραμματισμός για την εισαγωγή ενός μικρού ελληνικού γράμματος. 198
Σχήμα 13.4: Μετά την επιτυχή επιλογή γράμματος. 199
Σχήμα 13.5: Ολοκληρωμένο το πρόγραμμα της Κρεμάλας. 201
Σχήμα 13.6: Ορισμός των πινάκων που απαιτούνται. 202
Σχήμα 13.7: Περιγραφή του αρχικού οργανισμού. 202
Σχήμα 13.8: Εμφάνιση του οργανισμού στην οθόνη. 203
Σχήμα 13.9: Υπολογισμός επόμενης γενιάς. 204
Σχήμα 13.10: Ολοκληρωμένο το πρόγραμμα του Παιχνιδιού της Ζωής. 206

Κεφάλαιο: 14 **208**
Σχήμα 14.1: Βασικές φάσεις ανάπτυξης λογισμικού. 211