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.

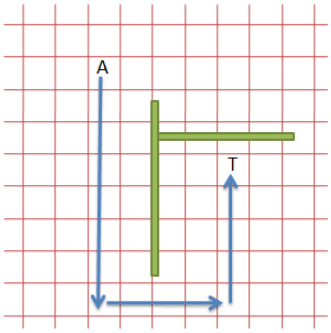
Αφού το έκανε αυτό θα του λέγαμε:

Πήγαινε 4 τετράγωνα δεξιά.

και τέλος

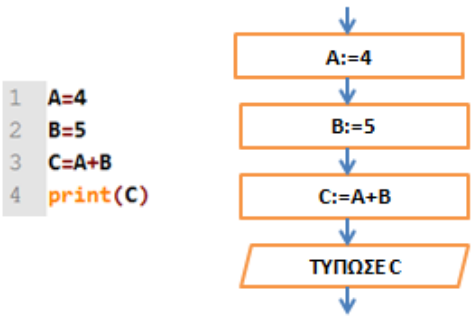
Πήγαινε 4 τετράγωνα επάνω.

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.

```
if condition:
    statement1
    statement2
statementN+1
statementN+2
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**.

```
x=int(input('Δώσε μου έναν αριθμό: '))
if (x>0):
    print ('Ο αριθμός είναι θετικός')
if (x<0):
    print ('Ο αριθμός είναι αρνητικός')
if (x==0):
    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.

```c
switch(menu)
{
    case 1: printf("selection 1");
            break;
    case 2: printf("selection 2");
            break;
    case 3: printf("selection 3");
            break;
    default: printf("not a valid selection");
}
```

Σχήμα 6.10: Παράδειγμα με τη δομή switch.

```python
if menu==1:
  print('selection 1')
elif menu==2:
  print('selection 2')
elif menu==3:
  print('selection 3')
else:
  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.

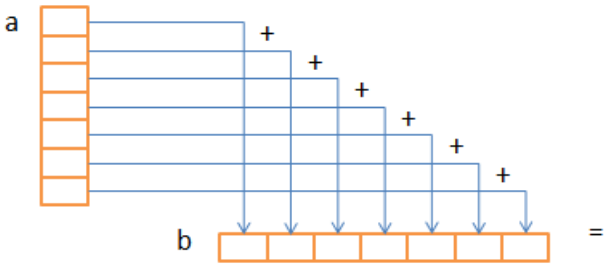
Θα παρατηρήσατε βέβαια ότι ο μετρητής (**i** στην περίπτωσή μας) αρχικοποιήθηκε στο 1, που είναι η πρώτη από τις τιμές στην παρένθεση του **range**, και έφτασε έως την τιμή 11, που είναι η δεύτερη από τις τιμές στην παρένθεση του **range**, χωρίς όμως να πάρει την τιμή αυτή. Στο δεύτερο βρόχο, η τρίτη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

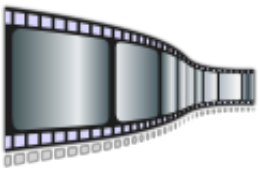
```
1  a=[1,3,5,7,9]
2  b=[2,4,6,8,0]
3  c=0
4  for i in range(len(a)):
5      c+=a[i]*b[i]
6  print (c)
7
8  C=[]
9  for i,j in zip(a,b):
10     C.append(i*j)
11 c=sum(C)
12 print (c)
```

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.

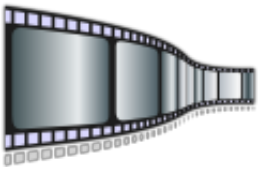
`http://repfiles.kallipos.gr/file/9283`

Ο κώδικας που το κάνει αυτό φαίνεται στο Σχήμα 6.15.

```
odd=[]
even=[]
for i in L:
    if i%2==0:
        even.append(i)
    else:
        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.

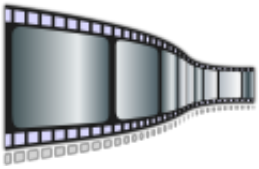
Έπειτα, αρχίζουν οι επαναλήψεις. Όσο το διάστημα **(a,b)** είναι μεγαλύτερο του επιτρεπόμενου σφάλματος (έλεγχος από το **while** στη γραμμή 10), υπολογίζουμε το μέσο του διαστήματος (γραμμή 11) και τις τιμές του πολυωνύμου στην αρχή και στη μέση του διαστήματος (γραμμές 12-13). Εάν το γινόμενο των δύο 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.