8

Δομές Δεδομένων II

8. Δομές Δεδομένων II

Εισαγωγή

Μια **δομή δεδομένων** μπορεί να οριστεί ως ένα σχήμα οργάνωσης σχετικών μεταξύ τους στοιχείων δεδομένων. Η επιλογή της κατάλληλης δομής δεδομένων παίζει σημαντικό ρόλο στην ανάπτυξη του αλγορίθμου, για την επίλυση ενός προβλήματος.

Στο κεφάλαιο αυτό θα αναπτυχθούν οι *ενσωματωμένες* δομές της Python, όπως τα αλφαριθμητικά, οι λίστες, οι πλειάδες και τα λεξικά. Σχετικά με τις λίστες και τα αλφαριθμητικά, που είχαν καλυφθεί και στην ύλη της Β' Λυκείου, θα γίνει μια πιο εκτενής παρουσίαση των βασικών λειτουργιών τους, μέσα από διάφορα παραδείγματα. Επίσης, με χρήση των ενσωματωμένων δομών της Python θα παρουσιαστούν υλοποιήσεις σύνθετων δομών δεδομένων, όπως είναι η στοίβα και η ουρά. Τέλος θα παρουσιαστούν θεωρητικά, οι δομές των γράφων και των δέντρων.

Διδακτικοί στόχοι

Μετά τη μελέτη του κεφαλαίου, θα μπορούμε να:

- αναλύουμε τρόπους αναπαράστασης στη μνήμη βασικών στατικών και δυναμικών δομών
- αναφέρουμε τα ιδιαίτερα χαρακτηριστικά των δομών: πίνακας, λίστα, στοίβα και ουρά
- κατονομάζουμε τη χρησιμότητα των δομών: δέντρο και γράφος
- επιλέγουμε και χρησιμοποιούμε κατάλληλες δομές στα προγράμματα που υλοποιούμε.

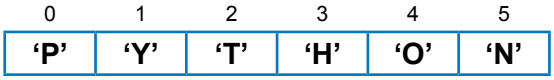
Λέξεις κλειδιά

Δομές δεδομένων, λίστα, στοίβα, ουρά, λεξικό, γράφος.

Διδακτικές Ενότητες

8.1 Συμβολοσειρές (strings)

Τα **αλφαριθμητικά** ή **συμβολοσειρές** στην Python είναι ακολουθίες από χαρακτήρες που έχουν σταθερό μέγεθος και *μη μεταβαλλόμενα* περιεχόμενα. Δηλαδή, δεν μπορούμε να προσθέσουμε ή να αφαιρέσουμε χαρακτήρες, ούτε να τροποποιήσουμε τα περιεχόμενα του αλφαριθμητικού. Γι' αυτό λέμε ότι η δομή αυτή ανήκει στις *μη μεταβαλλόμενες* (immutable) δομές της Python. Η αρίθμηση των χαρακτήρων σε ένα αλφαριθμητικό ξεκινάει από το 0. Για παράδειγμα: αν word = “PYTHON”, η αναπαράσταση του αλφαριθμητικού μοιάζει με το παρακάτω σχήμα:



Για παράδειγμα: το στοιχείο word[3] αναφέρεται στο 4ο γράμμα (‘H’), ενώ το στοιχείο word[0], στο 1ο γράμμα (‘P’). Ο τύπος των αλφαριθμητικών στην Python ονομάζεται **str**, από τα αρχικά της λέξης string. Παρακάτω δίνονται μερικά παραδείγματα εντολών με αλφαριθμητικά στον διερμηνευτή της Python:

```
>>> word = “PYTHON”
>>> len( word )
6
>>> print word[5]+word[0]
NP
>>> str(28) == ‘28’
True
>>> print int( ‘496’ ) + 4
500
```

- η συνάρτηση len επιστρέφει το μήκος, δηλαδή το πλήθος των χαρακτήρων του αλφαριθμητικού
- ο τελεστής + όταν εφαρμόζεται σε αντικείμενα τύπου string, έχει σαν αποτέλεσμα τη συνένωσή τους σε μια συμβολοσειρά
- η συνάρτηση str μετατρέπει μια τιμή σε συμβολοσειρά
- με τη συνάρτηση int μπορούμε να μετατρέψουμε ένα αλφαριθμητικό στον ακέραιο αριθμό που αναπαριστά.

Έλεγχος ύπαρξης

Ένας σημαντικός τελεστής που θα συναντήσουμε και αργότερα στις λίστες, είναι ο *υπαρξιακός τελεστής* **in**, ο οποίος ελέγχει, αν ένα αντικείμενο ανήκει σε ένα σύνολο αντικειμένων. Δεδομένου ότι οι συμβολοσειρές μπορούν να θεωρηθούν ως σύνολα χαρακτήρων, μπορούμε να τον χρησιμοποιήσουμε όπως φαίνεται παρακάτω.

```
>>> "Py" in "Python"
True
>>> "a" in "Python"
False
>>> "a" not in "Python"
True
```

```
>>> 'antonis' > 'antonia'
True
>>> '1000' < '2'
True
>>> 'babylon5' > 'babylon4'
True
```

Επίσης, οι γνωστοί *συγκριτικοί τελεστές* (<, <=, >, >=, ==, !=) ισχύουν και στις συμβολοσειρές, η λειτουργία των οποίων βασίζεται στη λεξικογραφική διάταξη των χαρακτήρων.

Παράδειγμα 1. Σάρωση των χαρακτήρων μιας συμβολοσειράς

Μια συμβολοσειρά είναι μια ακολουθία (sequence) από αντικείμενα, τα οποία, στην προκειμένη περίπτωση, είναι χαρακτήρες. Αν θέλουμε να σαρώσουμε τα αντικείμενα αυτά ένα-ένα, μπορούμε να το κάνουμε με μια εντολή επανάληψης for. Το παρακάτω τμήμα κώδικα δημιουργεί μια νέα συμβολοσειρά η οποία είναι όμοια με την αρχική, αλλά χωρίς τα κενά ανάμεσα στις λέξεις:

```
def trimSpaces( sentence ):
    result = “”
    for char in sentence :
        if char != “ ” :
            result += char
    return result

>>> phrase = “Houston we have a problem”
>>> trimSpaces( phrase )
‘Houstonwehaveaproblem’
```

Παράδειγμα 2. Καταμέτρηση φωνηέντων μιας φράσης Η παρακάτω συνάρτηση υπολογίζει πόσα φωνήεντα έχει η λέξη word.

```
def count_vowels( word ):
    vowels = “AEIOUaeiou”
    count = 0
    for letter in word :
        if letter in vowels:
            count += 1
    return count
```

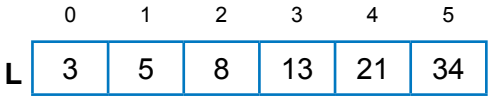
Σημείωση: Παρατηρήστε το διαφορετικό τρόπο με τον οποίο χρησιμοποιείται ο τελεστής **in**, στη μία περίπτωση για τον καθορισμό του εύρους της επανάληψης και στην άλλη για τον έλεγχο ύπαρξης του γράμματος letter στη λέξη vowels.

Παραπάνω κάνουμε χρήση του ιδιώματος for…in για να επεξεργαστούμε τους χαρακτήρες της συμβολοσειράς, έναν κάθε φορά.

8.2 Λίστες

Η **λίστα** είναι μια διατεταγμένη ακολουθία αντικειμένων, όχι απαραίτητα του ίδιου τύπου και αποτελεί τη **βασική δομή δεδομένων της Python**. Η λίστα, σε αντίθεση με τη συμβολοσειρά, είναι μια *δυναμική δομή* στην οποία μπορούμε να προσθέτουμε ή να αφαιρούμε στοιχεία (mutable). Κάθε αντικείμενο της λίστας χαρακτηρίζεται από ένα μοναδικό αύξοντα αριθμό, ο οποίος ορίζει τη θέση του στη λίστα, ενώ η προσπέλαση στα στοιχεία της λίστας γίνεται όπως στις συμβολοσειρές, με το όνομα της λίστας και τον αύξοντα αριθμό του αντικείμενου μέσα σε αγκύλες.

Η εντολή L = [3, 5, 8, 13, 21, 34] δημιουργεί τη μεταβλητή L που αναφέρεται στη λίστα [3, 5, 8, 13, 21, 34], όπως φαίνεται στην εικόνα:



Η αρίθμηση των στοιχείων, όπως στις συμβολοσειρές, έτσι και στις λίστες, ξεκινάει από το 0. Άρα το 1ο στοιχείο της λίστας είναι το L[0], το οποίο είναι ίσο με το 3, το 2ο το L[1] και τελευταίο το L[5].

```
>>> L = [ 3, 5, 8, 13, 21, 34 ]
>>> print L[ 0 ]
3
>>> print L[ 5 ]
34
```

Μπορεί κανείς ανά πάσα στιγμή να *προσθέσει*, να *αφαιρέσει* ή να *τροποποιήσει* οποιοδήποτε στοιχείο της λίστας.

```
>>> daysofweek = ["Δευτέρα",
"Τρίτη","Τετάρτη","Πέμπτη","Παρασκευή"]
>>> print daysofweek[ 0 ] + daysofweek[ 4 ]
ΔευτέραΠαρασκευή
```

```
>>> daysofweek = daysofweek + ["Σάββατο"]
>>> daysofweek[ 0 ] = daysofweek[ 1 ] = "Κυριακή"
>>> print daysofweek
["Κυριακή","Κυριακή","Τετάρτη", "Πέμπτη", "Παρασκευή", “Σάββατο”]
```

Έτσι, αν θέλουμε να προσθέσουμε ένα στοιχείο στο τέλος μιας λίστας, γράφουμε:

Λίστα = Λίστα + [στοιχείο]

ενώ στην αρχή της λίστας

Λίστα = [στοιχείο] + Λίστα

Στην πραγματικότητα όμως οι παραπάνω εντολές δεν προσθέτουν το στοιχείο στην ήδη υπάρχουσα λίστα αλλά δημιουργούν μια νέα λίστα κάθε φορά. Η λειτουργία αυτή έχει σημαντικό υπολογιστικό κόστος. Για αυτό αν θέλουμε να προσθέσουμε ένα στοιχείο στο τέλος της λίστας προτιμούμε τον τελεστή **+=** , όπως φαίνεται παρακάτω:

Λίστα += [στοιχείο]

Οι λίστες στην Python:

- Δεν έχουν σταθερό μέγεθος, δηλαδή μπορούν να αυξάνονται και να μειώνονται κατά την εκτέλεση του προγράμματος.
- Η αρίθμηση των δεικτών ξεκινάει από το 0, όπως ακριβώς στις συμβολοσειρές.
- Είναι *δυναμικές δομές*, και χαρακτηρίζονται από μεγάλη ευελιξία. Έτσι για παράδειγμα, μπορούμε να έχουμε σε μια λίστα ακόμα και στοιχεία διαφορετικού τύπου.

```
>>> mix = [6, 3.14159 , True, “Guido Van Rossum”]
>>> len(mix)
4
```

Στις λίστες μπορούμε να χρησιμοποιήσουμε τον υπαρξιακό τελεστή **in**, τη συνάρτηση **len** ή και τον τελεστή συνένωσης '+', ακριβώς όπως στις συμβολοσειρές.

```
>>> fruits = ['apple', 'orange']
>>> len(fruits)
2
>>> print fruits[0]
apple
>>> 'apple' in fruits
True
>>> powers = [2, 4, 8, 16]
>>> fib = [3, 5, 8, 13, 21]
>>> fib + powers
[3, 5, 8, 13, 21, 2, 4, 8, 16]
>>> powers + fruits
[2, 4, 8, 16, 'apple', 'orange']
>>> fib = fib + [ fib[3] + fib[4] ]
>>> print fib
[3, 5, 8, 13, 21, 34]
```

Επειδή οι λίστες και οι συμβολοσειρές ανήκουν και οι δύο σε μια πιο γενική κατηγορία δομών, τις *ακολουθιακές δομές* (sequences) της Python, ισχύουν οι ίδιοι τελεστές:

item in List: επιστρέφει True, αν το στοιχείο item υπάρχει μέσα στη λίστα List, αλλιώς επιστρέφει False.

item not in List: επιστρέφει True, αν το στοιχείο item δεν υπάρχει μέσα στη λίστα List, αλλιώς, αν υπάρχει επιστρέφει False.

Εκτός από τους τελεστές που αναφέραμε παραπάνω, υπάρχουν και συναρτήσεις που παίρνουν ως όρισμα μια λίστα. Οι συναρτήσεις των λιστών που θα χρησιμοποιήσουμε σε αυτό το κεφάλαιο είναι η len και η list.

len (List): Επιστρέφει το πλήθος των στοιχείων (ή μέγεθος) της λίστας.

list (String): Επιστρέφει μια λίστα με στοιχεία τους χαρακτήρες της συμβολοσειράς string. Η συνάρτηση αυτή μπορεί να μετατρέψει και άλλα είδη δομών σε λίστα, όπως είναι οι πλειάδες και τα λεξικά.

Οι λίστες, όπως και οι συμβολοσειρές, διαθέτουν μεγάλη ποικιλία μεθόδων, η χρήση των οποίων μπορεί να επεκτείνει, σε μεγάλο βαθμό, τη λειτουργικότητά τους. Στο κεφάλαιο αυτό θα χρησιμοποιήσουμε μόνο τις παρακάτω μεθόδους των λιστών, όπου L το όνομα της λίστας:

L.**append**(object): προσθήκη του στοιχείου object στο τέλος της λίστας L.

L.**insert**(index, object): προσθήκη του στοιχείου object, στη θέση index της λίστας L, μετακινώντας όλα τα στοιχεία από τη θέση index και μετά, κατά μία θέση.

L.**pop**([index]): Αφαίρεση από τη λίστα του στοιχείου που βρίσκεται στη θέση index. Αν δεν δοθεί θέση, τότε θα αφαιρεθεί το τελευταίο στοιχείο της λίστας.

Παρακάτω δίνονται μερικά παραδείγματα κλήσεων των μεθόδων που θα χρησιμοποιήσουμε και δίπλα, μέσα σε σχόλια, η νέα μορφή της λίστας.

```
>>> fib = [5, 8, 13, 21, 34]
>>> fib.pop(1)                  # fib = [5, 13, 21, 34 ]
>>> fib.append(55)              # fib = [5, 13, 21, 34, 55]
>>> fib.pop( )                  # fib = [5, 13, 21, 34]
>>> fib.insert( 2, 89 )         # fib = [5, 13, 89, 21, 34]
```

Αν και η Python ορίζει μια μεγάλη ποικιλία μεθόδων για την επεξεργασία και διαχείριση λιστών, στο βιβλίο αυτό θα χρησιμοποιηθούν μόνο οι παραπάνω, για την επίλυση προβλημάτων.

Διάσχιση Λίστας

Μπορούμε να επεξεργαστούμε τα στοιχεία μιας λίστας, ένα κάθε φορά, κάνοντας χρήση του παρακάτω ιδιώματος της δομής επανάληψης for:

```
for item in List :
    <Εντολές Επεξεργασίας του αντικειμένου item>
```

Παράδειγμα 1. Δημιουργία και εμφάνιση στοιχείων λίστας

Οι παρακάτω εντολές αρχικά κατασκευάζουν μια λίστα η οποία περιέχει όλους τους αριθμούς από το 1 έως και το 6 και στη συνέχεια εμφανίζουν κάθε αριθμό σε διαφορετική γραμμή.

```
L = [1, 2, 3, 4, 5, 6]
for number in L :
   print number
```

Παράδειγμα 2. Μέσος όρων των στοιχείων μιας λίστας

Για να υπολογίσουμε το μέσο όρο των στοιχείων μιας λίστας, πρώτα χρειάζεται να υπολογίσουμε το άθροισμα των στοιχείων, χρησιμοποιώντας μια μεταβλητή στην οποία προσθέτουμε, κάθε φορά, το επόμενο στοιχείο της λίστας:

```
sum = 0.0                    # το sum είναι πραγματικός (float)
for number in L :
    sum = sum + number
average = sum / len( L )     # δεν θα γίνει ακέραια διαίρεση
print average
```

Παράδειγμα 3. Μέγιστη τιμή σε μια λίστα

```
maximum = L[0]
for number in L :
    if number > maximum :
       maximum = number
print maximum
```

Παράδειγμα 4. Ρέστα

Καλούμαστε να σχεδιάσουμε το λογισμικό μιας ταμειακής μηχανής, το οποίο θα διαβάζει το ποσό που έδωσε ο πελάτης και το κόστος των αγορών του και θα εμφανίζει το ελάχιστο πλήθος κερμάτων ή χαρτονομισμάτων που θα δοθούν για ρέστα. Θεωρήστε ότι όλες οι τιμές είναι σε ακέραια πολλαπλάσια του ευρώ:

```
values = [100, 50, 20, 10, 5, 2, 1]
cost = input( “Δώσε το κόστος των αγορών” )
payment = input( “Δώσε το ποσό της πληρωμής” )
change = payment – cost
counter = 0
for value in values :
   counter = counter + ( change / value )
   change = change % value
print counter
```

Εφαρμογή. Διαχωρισμός λίστας

Η λειτουργία του διαχωρισμού μιας λίστας σε δύο μέρη με βάση κάποια κριτήρια, αποτελεί μια τυπική επεξεργασία των λιστών.

```
positives = [ ]
negatives = [ ]
for number in numbers :      # για κάθε αριθμό της λίστας
    if number > 0 :              # αν είναι θετικός
       positives.append( number )
       # πρόσθεσέ τον στη λίστα με τους θετικούς
    else :
       negatives.append( number )
       # αλλιώς στη λίστα με τους αρνητικούς
print positives
print negatives
```

Το παραπάνω πρόγραμμα διαχωρίζει τους αριθμούς μιας λίστας σε αρνητικούς και θετικούς. Υποθέτουμε ότι όλοι οι αριθμοί είναι διάφοροι του μηδενός. Να σημειωθεί ότι η αρχική λίστα παραμένει ανέπαφη.

Εφαρμογή. Συγχώνευση διατεταγμένων λιστών

Ένας από τους πιο γνωστούς και χρήσιμους αλγορίθμους της Πληροφορικής είναι ο αλγόριθμος της συγχώνευσης των στοιχείων δυο ταξινομημένων λιστών σε μία νέα, επίσης ταξινομημένη, λίστα. Ο αλγόριθμος αξιοποιεί το γεγονός ότι οι αρχικές λίστες είναι ήδη ταξινομημένες, ώστε να μη χρειαστεί να ταξινομήσει από την αρχή την τελική λίστα, κάτι το οποίο έχει σημαντικό υπολογιστικό κόστος για πολύ μεγάλες λίστες.

```
def merge( A, B ) :
    L = [ ]
    # όσο οι δυο λίστες έχουν στοιχεία
    while A != [ ] and B != [ ] :
        if A[0] < B[0] : # Αν το 1ο στοιχείο της A είναι το μικρότερο
            L.append( A.pop(0) )        # το μεταφέρουμε στο τέλος της L
        else :
            L.append( B.pop(0) )
            # αλλιώς μεταφέρουμε το πρώτο στοιχείο της B στην L
    return L + A + B
    # στο τέλος προσθέτουμε τα στοιχεία που έχουν μείνει
```

Αφού οι δύο λίστες είναι ταξινομημένες σε αύξουσα σειρά το ελάχιστο στοιχείο και των δύο λιστών είναι το μικρότερο από τα A[0], B[0]. Στη συνέχεια μεταφέρουμε αυτό το στοιχείο στην τελική λίστα.

Κάποια στιγμή, μία από τις δυο λίστες θα αδειάσει, οπότε η επανάληψη θα σταματήσει. Η άλλη λίστα όμως θα έχει κάποια στοιχεία τα οποία πρέπει να προστεθούν στο τέλος της τρίτης λίστας. Έτσι θα έπρεπε να γράψουμε:

```
if A == [ ] :
   L = L + B
else :
   L = L + A
```

} L = L + B + A

Οπότε αν η A είναι κενή το αποτελέσματα είναι L + B + [ ], ενώ αν η B είναι κενή το αποτέλεσμα είναι L + [ ] + A.

Η συνάρτηση range (έχει αναλυθεί και στην 4.1.3)

Η συνάρτηση range επιστρέφει, αν δώσουμε την αρχική, την τελική τιμή και το βήμα, μια λίστα από αριθμούς, όπως φαίνεται παρακάτω:

```
>>> range( 4 )
[0, 1, 2, 3]
>>> range( 0, 4 )
[0, 1, 2, 3]
>>> range( 0, 4, 1 )
[0, 1, 2, 3]
>>> range( 1, 1, 100 )
[ ]
>>> range( 1, 5, –1 )
[ ]
>>> range( 0 )
[ ]
>>> range( 10, 30, 5 )
[10, 15, 20, 25]
>>> range( 30, 10, –5 )
[30, 25, 20, 15]
>>> range( 1, 2, 100 )
[1]
```

Η συνάρτηση **range**(A, M, B) επιστρέφει μια λίστα αριθμών ξεκινώντας με τον αριθμό A μέχρι το M με βήμα B. Το M δεν συμπεριλαμβάνεται στη λίστα.

Έτσι τα παρακάτω τμήματα κώδικα εκτελούν την ίδια λειτουργία:

```
>>> L = [ 6, 28, 496, 8128 ]
>>> for item in L :
        print item ,
6 28 496 8128
>>> L = [ 6, 28, 496, 8128 ]
>>> for index in [ 0, 1, 2, 3 ] :
        print L[index] ,
6 28 496 8128
```

```
>>> L = [ 6, 28, 496, 8128 ]
>>> for index in range(0,4) :
        print L[index] ,
6 28 496 8128
```

Η range μας διευκολύνει επίσης στην εκτέλεση ενός τμήματος εντολών για έναν προκαθορισμένο αριθμό επαναλήψεων, όπως φαίνεται παρακάτω:

```
>>> sum = 0
>>> for i in range(101) :
        sum = sum + i
>>> print sum
5050
```

```
>>> for index in range(2,11,2) :
        print index ,
2 4 6 8 10
```

Ο τελεστής διαμέρισης :

Ένας πολύ χρήσιμος τελεστής είναι ο **τελεστής διαμέρισης** (slice operator) που συμβολίζεται με την άνω κάτω τελεία **:**, και ο οποίος έχει αναφερθεί στις παραγράφους 5.2, 5.3 του βιβλίου της Β' τάξης. Ο τελεστής διαμέρισης μπορεί να μας επιστρέψει ένα τμήμα μιας συμβολοσειράς ή μιας λίστας. Η έκφραση word[a **:** b] μας επιστρέφει το τμήμα της συμβολοσειράς ή της λίστας από το στοιχείο word[a] μέχρι και το στοιχείο word[b-1]. Μπορούμε να παραλείψουμε την αρχή ή το τέλος όπως φαίνεται στα παραδείγματα παρακάτω αν θέλουμε ένα τμήμα από την αρχή ή το τέλος της λίστας/συμβολοσειράς αντίστοιχα.

Δίνονται τα παρακάτω παραδείγματα:

```
>>> word = “zanneio gymnasio”
>>> word[ :7 ]
‘zanneio’
>>> word[ 8: ]
‘zanneio’
>>> word[ : ]
‘zanneio gymnasio’
```

```
>>> word[ 3:11 ]
‘neio gym’
>>> word[ 0:7 ]
‘zanneio’
>>> word[ 8:len(word) ]
‘gymnasio’
>>> word[ 0:len(word) ]
‘zanneio gymnasio’
```

Ο τελεστής **:** είναι πολύ σημαντικός στην επεξεργασία λιστών στην Python, γιατί μπορούμε να τον χρησιμοποιήσουμε για να πάρουμε ένα αντίγραφο μιας λίστας όπως παρακάτω:

```
>>> fibonacci = [5,8,13,21,34]
>>> fib = fibonacci[ : ]          # δημιουργεί αντίγραφο
>>> a = fib    # δείχνουν στην ίδια λίστα
>>> a.pop()  # οι αλλαγές επηρεάζουν και την a και τη fib
34
>>> fib.pop()
21
>>> a[0]=a[1]=55
>>> print a
[55, 55, 13]
>>> print fib
[55, 55, 13]
>>> print fibonacci
```

```
# Η αρχική λίστα fibonacci δεν έχει αλλάξει
[5, 8, 13, 21, 34]
```

Προσοχή! Δεν μπορεί να γίνει το ίδιο με τις συμβολοσειρές. Δηλαδή η εντολή word[:] δεν δημιουργεί αντίγραφο της συμβολοσειράς word. Αυτό μπορείτε να το διαπιστώσετε με χρήση της συνάρτησης id που δείχνει τη διεύθυνση στη μνήμη του κάθε αντικειμένου.

Ο τελεστής διαμέρισης μπορεί να φανεί πολύ χρήσιμος σε κάποιες περιπτώσεις όπως για παράδειγμα στην δραστηριότητα 8 του κεφαλαίου 5 του βιβλίου της Β' τάξης όπου ζητείται η υλοποίηση του αλγορίθμου κρυπτογράφησης του Καίσαρα. Αν κάθε γράμμα αντιστοιχίζεται με το γράμμα που βρίσκεται 3 θέσεις μπροστά τότε το νέο αλφάβητο μπορεί να προκύψει με τις παρακάτω εντολές:

```
>>> alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> cipherAlphabet = alphabet[ 3 : ] + alphabet[ : 3]
>>> print cipherAlphabet
DEFGHIJKLMNOPQRSTUVWXYZABC
```

8.3 Στοίβα

Εισαγωγική Δραστηριότητα

Όλοι χρησιμοποιούμε διάφορα προγράμματα για πλοήγηση στον παγκόσμιο ιστό, τα οποία είναι γνωστά ως φυλλομετρητές (browsers) και πολλές φορές, κατά την πλοήγηση, θέλουμε να επιστρέψουμε στην αμέσως προηγούμενη ιστοσελίδα στην οποία βρισκόμασταν. Γι' αυτό υπάρχει η λειτουργία **Πίσω** (Back) σε όλους τους φυλλομετρητές, η οποία μας μεταφέρει στην προηγούμενη ιστοσελίδα. Για να επιτευχθεί αυτή η λειτουργία, το περιβάλλον διατηρεί ένα ιστορικό των ιστοσελίδων τις οποίες επισκεφθήκαμε. Η δομή δεδομένων που χρησιμοποιείται για την αποθήκευση του ιστορικού, πρέπει να είναι τέτοια, ώστε οι ιστοσελίδες να ανακτώνται με την αντίστροφη σειρά επίσκεψης. Δηλαδή στην πρώτη θέση θέλουμε να είναι η ιστοσελίδα που επισκεφτήκαμε πιο πρόσφατα και στην τελευταία η αρχική ιστοσελίδα από την οποία ξεκινήσαμε.

Ας υποθέσουμε για παράδειγμα ότι, ξεκινώντας από την ιστοσελίδα http://www.python.org μεταβαίνουμε στην ιστοσελίδα http://www.pythontutor.com και στη συνέχεια στην ιστοσελίδα http://www.iep.edu.gr. Το ιστορικό των επισκέψεων μέχρι στιγμής περιέχει τις τρεις αυτές ιστοσελίδες όπως φαίνεται παρακάτω:

http://www.iep.edu.gr
http://www.pythontutor.com
http://www.python.org

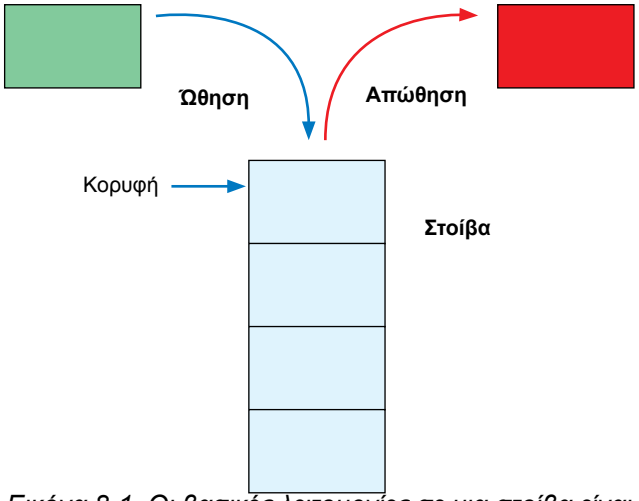
Η πρώτη ιστοσελίδα στην οποία έχουμε πρόσβαση είναι η τελευταία που επισκεφτήκαμε και βρίσκεται πάνω – πάνω στο ιστορικό. Αν μεταβούμε στην προηγούμενη ιστοσελίδα, τότε το ιστορικό θα πάρει τη μορφή:

http://www.pythontutor.com
http://www.python.org

Αν στην συνέχεια μεταβούμε στην ιστοσελίδα http://www.minedu.gov.gr/, τότε αυτή θα προστεθεί στην κορυφή του ιστορικού:

http://www.minedu.gov.gr/
http://www.pythontutor.com
http://www.python.org

Από τα παραπάνω μπορούμε να συμπεράνουμε ότι η δομή που χρησιμοποιείται από το ιστορικό, είναι μια λίστα στην οποία οι εισαγωγές και οι διαγραφές στοιχείων γίνονται από το πάνω άκρο μόνο. Στη λίστα αυτή το στοιχείο που προστέθηκε τελευταίο είναι και το πρώτο που θα εξαχθεί, έχουμε δηλαδή μια λειτουργία τύπου LIFO (**L**ast **I**n **F**irst **O**ut), δηλαδή ο τελευταίος που εισέρχεται στη λίστα, είναι και ο πρώτος που θα εξαχθεί. Η συγκεκριμένη δομή δεδομένων ονομάζεται **Στοίβα** και οι λειτουργίες εισαγωγής και εξαγωγής είναι γνωστές στη βιβλιογραφία ως **ώθηση** και **απώθηση**.



Εικόνα 8-1. Οι βασικές λειτουργίες σε μια στοίβα είναι η ώθηση και η απώθηση

Ένα άλλο παράδειγμα που χρησιμοποιείται συχνά για να περιγράψει μια στοίβα, είναι αυτό του σωρού με πιάτα που περιμένουν να πλυθούν. Κάθε νέο πιάτο που έρχεται για πλύσιμο τοποθετείται πάνω στον σωρό, με αποτέλεσμα να είναι το πρώτο που θα πλυθεί.

Η στοίβα αποτελεί μια από τις σημαντικότερες δομές δεδομένων της επιστήμης της Πληροφορικής και χρησιμοποιείται σε πολλά πεδία της, όπως είναι η θεωρία αλγορίθμων, η ανάπτυξη μεταγλωττιστών, η τεχνητή νοημοσύνη κ.ά.

Όταν η στοίβα είναι άδεια, είναι προφανές ότι δεν μπορεί να γίνει απώθηση. Άρα, όταν απωθούμε ένα στοιχείο από τη στοίβα, θα πρέπει προηγουμένως να έχουμε εξασφαλίσει ότι η στοίβα δεν είναι κενή. Για αυτό το λόγο, εκτός από την ώθηση και την απώθηση, πρέπει να υλοποιήσουμε και τον έλεγχο, αν η στοίβα είναι κενή. Άρα οι βασικές λειτουργίες που πρέπει να υποστηρίζει η υλοποίηση μιας στοίβας είναι:

- Δημιουργία μιας κενής στοίβας.
- Έλεγχος, αν η στοίβα είναι κενή.
- Ώθηση ενός στοιχείου στη στοίβα.
- Απώθηση ενός στοιχείου από τη στοίβα.

Η δομή της στοίβας μπορεί να υλοποιηθεί στην Python με μια λίστα στην οποία οι εισαγωγές και οι εξαγωγές στοιχείων γίνονται μόνο από το ένα άκρο. Παρακάτω δίνονται δύο υλοποιήσεις της στοίβας. Στην πρώτη περίπτωση, οι εισαγωγές/διαγραφές στοιχείων γίνονται στο τέλος της λίστας, ενώ στη δεύτερη, στην αρχή της.

Υλοποίηση Στοίβας σε Python με δύο τρόπους

```
def push(stack, item) :
    stack.append( item )
def pop(stack) :
    return stack.pop( )
def isEmpty(stack) :
    return len(stack) == 0
def createStack( ) :
    return [ ]
```

```
def push(stack, item) :
    stack.insert(0, item)
def pop(stack) :
    return stack.pop( 0 )
def isEmpty(stack) :
    return len(stack) == 0
def createStack( ) :
    return [ ]
```

Οι δύο υλοποιήσεις που δίνονται παραπάνω, διαφέρουν μόνο ως προς το σημείο της λίστας στο οποίο γίνονται οι εισαγωγές/διαγραφές των στοιχείων.

Εφαρμογή Στοίβας. Αντιστροφή αριθμών

Το παρακάτω πρόγραμμα δέχεται από το χρήστη αριθμούς μέχρι να δοθεί το 0 και τους εμφανίζει σε αντίστροφη σειρά από αυτή με την οποία δόθηκαν. Θέλουμε κάθε φορά να εμφανίσουμε πρώτο τον αριθμό που δόθηκε τελευταίος. Χρειαζόμαστε μια δομή δεδομένων που να υποστηρίζει τη λειτουργία LIFO, όπως η στοίβα.

```
stack = createStack()              # Δημιουργία της στοίβας stack
number = int( raw_input( ) )       # Διάβασε τον πρώτο αριθμό
while number != 0 :                # Όσο δεν δίνεται 0
    push(stack, number)            # Σπρώξε τον αριθμό στη στοίβα
    number = int( raw_input( ) )   # Διάβασε τον επόμενο αριθμό
while not isEmpty( stack ) :       # Μέχρι να αδειάσει η στοίβα
    number = pop(stack)            # βγάλε τον επόμενο αριθμό
    print number                   # και εκτύπωσέ τον
```

Παρατηρήστε ότι στο παραπάνω πρόγραμμα δεν φαίνεται ποια υλοποίηση χρησιμοποιείται. Αν τροποποιήσουμε την υλοποίηση της ώθησης, δε θα χρειαστεί να αλλάξουμε τίποτα στο πρόγραμμα. Αυτό είναι γνωστό ως δ*ιαχωρισμός διεπαφής (διασύνδεσης) – υλοποίησης (interface / implementation)*, αφού οι εφαρμογές που χρησιμοποιούν δομές δεδομένων όπως η στοίβα, είναι απολύτως ανεξάρτητες από την υλοποίηση.

Ουσιαστικά δε μας ενδιαφέρει πώς έχει υλοποιηθεί η στοίβα, αφού εμείς θέλουμε μόνο να χρησιμοποιήσουμε τις συναρτήσεις που έχουμε ορίσει παραπάνω.

Το σύνολο των επικεφαλίδων των συναρτήσεων το οποίο είναι διαθέσιμο στον προγραμματιστή που χρησιμοποιεί τη στοίβα, το ονομάζουμε διεπαφή ή διασύνδεση (interface) της δομής αυτής. Η διεπαφή μιας δομής ορίζει τι μπορεί να κάνει η δομή και όχι τον τρόπο με τον οποίο το κάνει. Το τελευταίο είναι ο ρόλος της υλοποίησης.

Διεπαφή της δομής δεδομένων Στοίβα

```
def push(stack, item)
def pop(stack)
def isEmpty(stack)
def createStack( )
```

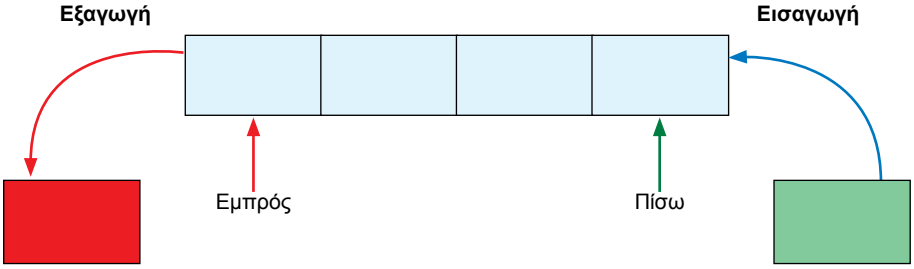
8.4 Ουρά

Μια δομή δεδομένων που χρησιμοποιείται για την μοντελοποίηση και προσομοίωση πραγματικών φαινομένων, είναι η δομή της ουράς. Τα φαινόμενα που μοντελοποιούνται αναφέρονται στην εξυπηρέτηση ανθρώπων, αντικειμένων ή προγραμμάτων. Τέτοια παραδείγματα ουρών είναι:

- Οι ουρές στις τράπεζες και τα σούπερ-μάρκετ.
- Η ουρά των προγραμμάτων που περιμένουν να εξυπηρετηθούν από τον επεξεργαστή του υπολογιστή σας.
- Η ουρά των αιτήσεων προς το διακομιστή ιστού (web server) ενός δικτυακού τόπου.

Τα φαινόμενα αυτά μελετώνται από διάφορους κλάδους των Μαθηματικών και της Πληροφορικής, όπως είναι η Θεωρία Ουρών (Queueing theory) και η Επιχειρησιακή 'Ερευνα (Operations Research).

Σε αντίθεση με τη στοίβα, που η λειτουργία της χαρακτηρίζεται ως LIFO (**L**ast **I**n **F**irst **O**ut), η λειτουργία της ουράς είναι γνωστή στη βιβλιογραφία ως FIFO (First In First Out), αφού το κάθε στοιχείο της ουράς εξυπηρετείται με τη σειρά που έφτασε στην ουρά.



Εικόνα 8-2

Δύο είναι οι βασικές λειτουργίες μιας ουράς:

- Εισαγωγή στοιχείου, η οποία γίνεται στο πίσω μέρος της ουράς.
- Εξαγωγή στοιχείου, η οποία γίνεται από το εμπρός μέρος της ουράς.

Υλοποίηση Ουράς σε Python

```
def enqueue(queue, item) :
    queue = queue.append( item )
def dequeue(queue) :
    return queue.pop( 0 )
def isEmpty(queue) :
    return len(queue) == 0
def createQueue( ) :
    return [ ]
```