ΠΛΗΡΟΦΟΡΙΚΗ Ι (Python)
Ενότητα 5

Λίστες (Lists)

Λίστα (list) στην Python ονομάζεται μια *δυναμική δομή δεδομένων* (στην πραγματικότητα, ένα *αντικείμενο*) που περιέχει πολλαπλά δεδομένα. Ουσιαστικά, η λίστα είναι ένας τρόπος αποθήκευσης ενός συνόλου δεδομένων υπό ένα κοινό όνομα (το όνομα της λίστας). Οι λίστες έχουν πολλά κοινά χαρακτηριστικά με τους *πίνακες* άλλων γλωσσών προγραμματισμού. Η βασικότερή τους διαφορά από τους πίνακες των περισσοτέρων γλωσσών προγραμματισμού είναι ότι αποτελούν, όπως προαναφέρθηκε, *δυναμικές* δομές δεδομένων, δηλαδή τα περιεχόμενά τους μπορούν να αυξάνονται ή να μειώνονται κατά την εκτέλεση του προγράμματος (περισσότερες λεπτομέρειες θα αναφερθούν στη συνέχεια).

- ➔ Κάθε δεδομένο μιας λίστας ονομάζεται *στοιχείο* (element) της λίστας.
- ➔ Μια λίστα μπορεί να περιέχει στοιχεία διαφορετικών τύπων.
- ➔ Κάθε στοιχείο μιας λίστας έχει έναν *δείκτη* (index) που προσδιορίζει τη θέση του στη λίστα. Η αρίθμηση των δεικτών αυτών ξεκινάει από το 0.

Δημιουργία λίστας

i) *Αναλυτικά*

```
λίστα = [στοιχείο_1, στοιχείο_2, ... ]
```

ή

```
λίστα = []
```

για τη δημιουργία κενής λίστας.

Π.χ.,

```
>>> list1 = [5, 10, 15, 20, 25]
>>> print(list1)
[5, 10, 15, 20, 25]
>>> list2 = [10, 'Hello', 2.5, 100]
>>> print(list2)
[10, 'Hello', 2.5, 100]
```

ii) *Με χρήση του τελεστή επανάληψης* *

```
λίστα = [στοιχείο_1, στοιχείο_2, ... ] * πλήθος_επαναλήψεων
```

Π.χ.,

```
>>> list3 = [5, 2.5] * 5
>>> print(list3)
[5, 2.5, 5, 2.5, 5, 2.5, 5, 2.5, 5, 2.5]
>>> list3 = ['Hello', '!', 2015] * 2
>>> print(list3)
['Hello', '!', 2015, 'Hello', '!', 2015]
>>> list4 = [0] * 10
>>> print(list4)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

iii) *Με χρήση της συνάρτησης* `list()`

Η συνάρτηση `range`, όπως αναφέρθηκε στην Ενότητα 2, επιστρέφει ένα αντικείμενο τιμών βρόχου, το οποίο είναι ένα αντικείμενο που περιέχει μία αλληλουχία τιμών, η οποία όμως δεν αποτελεί λίστα, με την έννοια της λίστας της Python. Μπορεί όμως να μετατραπεί σε κανονική λίστα, με τη χρήση της ενσωματωμένης συνάρτησης `list` της Python, δίνοντας έτσι άλλον έναν τρόπο δημιουργίας λιστών:

```
λίστα = list(range(...))
```

ή

```
λίστα = list()
```

για τη δημιουργία κενής λίστας.

Ουσιαστικά η συνάρτηση `list` δέχεται σε κάποια συγκεκριμένη δομή *τις τιμές* που θα περιέχει η λίστα, και τις αποθηκεύει σε δομή δεδομένων λίστας. Μπορεί να εφαρμοστεί σε οποιαδήποτε αλληλουχία τιμών και όχι μόνο σε τιμές βρόχου *(περισσότερα για αυτό σε επόμενη ενότητα)*.

Π.χ.,

```
>>> list5 = list(range(0,11,2))
>>> print(list5)
[0, 2, 4, 6, 8, 10]
>>> list6 = list()
>>> print(list6)
[]
```

Προσπέλαση στοιχείων

i) *Με χρήση της εντολής* `for`

Π.χ.,

```
>>> list7 = [5, 10, 15, 20, 25, 30]
>>> for x in list7:
        print(x)
5
```

```
10
15
20
25
30
```

ii) *Με χρήση δεικτών*

Όπως προαναφέρθηκε, κάθε στοιχείο μιας λίστας έχει έναν δείκτη, ανάλογα με τη θέση του στη λίστα. Η αρίθμηση των δεικτών ξεκινάει από το 0 και φτάνει έως το πλήθος των στοιχείων μείον 1. Η αναφορά σε συγκεκριμένο στοιχείο μιας λίστας μπορεί να γίνει ως εξής:

```
λίστα[δείκτης_στοιχείου]
```

Π.χ.,

```
>>> list8 = [2, -4, 6, -8, 10]
>>> print(list8[0], list8[1], list8[2], list8[3], list8[4])
2 -4 6 -8 10
>>> for index in range(5):
        print(list8[index])

2
-4
6
-8
10
>>> index = 0
>>> while index < 5:
        print(list8[index])
        index += 1

2
-4
6
-8
10
```

Μπορούν να χρησιμοποιηθούν και αρνητικές τιμές στους δείκτες μιας λίστας. Ως θέση -1 θεωρείται η τελευταία θέση της λίστας, -2 η προτελευταία, κοκ. Π.χ.,

```
>>> print(list8[-1], list8[-2], list8[-3])
10 -8 6
```

Σε περίπτωση που χρησιμοποιηθεί κάποια μη έγκυρη τιμή δείκτη, προκαλείται σφάλμα (ένας τύπος σφάλματος που λέγεται *εξαίρεση* και θα αναλυθεί σε επόμενη ενότητα). Π.χ. η παρακάτω εντολή προκαλεί σφάλμα γιατί δεν υπάρχει στοιχείο στη λίστα `list8` με δείκτη 5 (το τελευταίο (5ο) στοιχείο της λίστας έχει δείκτη 4):

```
>>> print(list8[5])
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    print(list8[5])
IndexError: list index out of range
```

Πολύ χρήσιμη για την αποφυγή τέτοιου είδους σφαλμάτων, είναι η ενσωματωμένη συνάρτηση **`len`** της Python, η οποία επιστρέφει το μέγεθος μιας λίστας, δηλαδή το πλήθος των στοιχείων της:

```
>>> items = len(list8)
>>> print(items)
5
```

Έτσι, ο δείκτης του τελευταίου στοιχείου μιας λίστας είναι το: `len(όνομα_λίστας)-1`. Άρα, η παρακάτω εντολή που χρησιμοποιήθηκε προηγουμένως ως παράδειγμα:

```
>>> for index in range(5):
        print(list8[index])
```

θα μπορούσε να γραφεί έτσι:

```
>>> for index in range(len(list8)):
        print(list8[index])
```

Σημαντική Παρατήρηση: Όταν μας ενδιαφέρουν **μόνο οι τιμές** των στοιχείων μιας λίστας προτιμάται ο προηγούμενος τρόπος ***i)***. Η προσπέλαση με δείκτες προτιμάται όταν μας ενδιαφέρει **και η θέση** του στοιχείου της λίστας.

iii) *Με χρήση του τελεστή* `:` *για απομόνωση τμημάτων λίστας (τεμαχισμός λίστας)*

Προσπέλαση σε συγκεκριμένα τμήματα μιας λίστας μπορεί να γίνει με τη χρήση του τελεστή `:` ως εξής:

`λίστα[`*αρχή*`:`*τέλος*`]` → τα στοιχεία της λίστας από το στοιχείο με δείκτη *αρχή* έως το στοιχείο με δείκτη *τέλος*-1.

`λίστα[:`*τέλος*`]` → τα στοιχεία της λίστας από το 1ο έως το στοιχείο με δείκτη *τέλος*-1.

`λίστα[`*αρχή*`:]` → τα στοιχεία της λίστας από το στοιχείο με δείκτη *αρχή* έως το τελευταίο στοιχείο.

`λίστα[`*αρχή*`:`*τέλος*`:`*βήμα*`]` → τα μη-συνεχόμενα στοιχεία της λίστας από το στοιχείο με δείκτη *αρχή* έως το στοιχείο με δείκτη *τέλος*-1 τα οποία “απέχουν” μεταξύ τους *βήμα*-1 θέσεις (δηλαδή ο δείκτης των στοιχείων που περιλαμβάνονται αυξάνεται κατά *βήμα*)

Π.χ.:

```
>>> list9 = [10, 20, 30, 40, 50, 60, 70, 80]
>>> print(list9[2:5])
[30, 40, 50]
>>> print(list9[:3])
[10, 20, 30]
>>> print(list9[4:])
[50, 60, 70, 80]
>>> print(list9[1:7:2])
[20, 40, 60]
```

```
>>> print(list9[-3:])
[60, 70, 80]
>>> print(list9[:-3])
[10, 20, 30, 40, 50]
```

Σε περίπτωση που χρησιμοποιηθεί κάποια μη έγκυρη τιμή δείκτη, *δεν* προκαλείται κάποιο σφάλμα. Αν ο δείκτης *τέλος* προσδιορίζει θέση μετά το τέλος της λίστας, χρησιμοποιείται στη θέση του το μήκος της λίστας. Αν ο δείκτης *αρχή* προσδιορίζει θέση πριν την αρχή της λίστας, χρησιμοποιείται στη θέση του το 0. Αν ο δείκτης *αρχή* είναι μεγαλύτερος του δείκτη *τέλος* (και οι τιμές τους είναι θετικές), επιστρέφεται η κενή λίστα.

Παράδειγμα – Μέσος όρος τιμών λίστας:

```
numbers = [1.5, 2, 1.2, 8.4, -2, -3.5]
total = 0
for n in numbers:
   total += n
avg = total/len(numbers)
print(avg)
```

Τεμαχισμός λίστας με αρνητικό βήμα

Στον τεμαχισμό μπορούμε να χρησιμοποιήσουμε και αρνητικό βήμα, οπότε διατρέχουμε τη λίστα από τα **δεξιά προς τα αριστερά.** Π.χ. για τη λίστα

```
>>> L = [0, 1, 2, 3, 4, 5, 6]
```

- *λίστα*[*αρχή*:*τέλος*:-1] → τα στοιχεία της λίστας από το στοιχείο με δείκτη *αρχή* έως το στοιχείο με δείκτη *τέλος*+1.

```
>>> print(L[5:2:-1])
[5, 4, 3]
```

- *λίστα*[:*τέλος*:-1] → τα στοιχεία της λίστας από το τελευταίο στοιχείο έως το στοιχείο με δείκτη *τέλος*+1.

```
>>> print(L[:2:-1])
[6, 5, 4, 3]
```

- *λίστα*[*αρχή*::-1] → τα στοιχεία της λίστας από το στοιχείο με δείκτη *αρχή* έως και το πρώτο στοιχείο.

```
>>> print(L[3::-1])
[3, 2, 1, 0]
```

Μια χρήσιμη εφαρμογή του τεμαχισμού με αρνητικό βήμα είναι η αντιστροφή της σειράς των στοιχείων της λίστας (η L δεν αλλάζει φυσικά, απλώς παίρνουμε ένα “αντίστροφο” αντίγραφο της).

```
>>> print(L[::-1])
[6, 5, 4, 3, 2, 1, 0]
```

Εκχώρηση νέων τιμών σε στοιχεία

Οι λίστες είναι μεταβλητοί (mutable) τύποι, δηλαδή οι τιμές των στοιχείων τους μπορούν να αλλάξουν. Αυτό γίνεται με εκχώρηση τιμής σε συγκεκριμένο στοιχείο:

```
λίστα[δείκτης] = τιμή
```

Π.χ.,

```
>>> list10 = [1, 2, 3, 4, 5]
>>> list10[1] = 20
>>> list10[3] = 40
>>> print(list10)
[1, 20, 3, 40, 5]
```

Σε αυτό το παράδειγμα, η ακόλουθη εντολή:

```
>>> list10[5] = 100
```

προκαλεί σφάλμα, αφού δεν υπάρχει στοιχείο με δείκτη 5 στη λίστα. Άρα, η εκχώρηση τιμής σε στοιχεία της λίστας μπορεί να γίνει μόνο για τα υπάρχοντα στοιχεία της, και δεν αποτελεί τρόπο δημιουργίας νέων στοιχείων.

Αρχικοποίηση λίστας

Για να αρχικοποιηθούν τα στοιχεία μιας λίστας σε συγκεκριμένες τιμές, θα πρέπει πρώτα να δημιουργηθούν, με μία από τις διαδικασίες δημιουργίας λίστας. Π.χ.,

```
# Δημιουργία μιας λίστας με 5 στοιχεία.
list11 = [0] * 5
# Εκχώρηση της τιμής 'θέση * 2' σε όλα τα στοιχεία της λίστας.
    # Με while:                       # Με for:
index = 0
while index < len(list11):          for index in range(len(list11)):
    list11[index] = 2*index             list11[index] = 2*index
    index += 1
```

Εύρεση στοιχείων λίστας με τον τελεστή `in`

Η έκφραση: *στοιχείο* `in` *λίστα* επιστρέφει `True` εάν το *στοιχείο* υπάρχει στη *λίστα*, και `False` σε διαφορετική περίπτωση. Π.χ.,

```
codes = ['U135', 'X979', 'A123', 'S888']
item = input('Δώσε κωδικό προϊόντος: ')
if item in codes:
    print('Το', item, 'βρέθηκε στη λίστα.')
else:
    print('Το', item , 'δε βρέθηκε στη λίστα.')
```

Αντίστοιχα, η έκφραση: *στοιχείο* `not in` *λίστα* ελέγχει εάν το *στοιχείο* δεν ανήκει στη *λίστα*.

Μέθοδοι για λίστες

Η Python περιέχει κάποιες μεθόδους που χρησιμοποιούνται για τη διαχείριση των λιστών. Μέθοδοι ονομάζονται κάποιοι ειδικοί τύποι συναρτήσεων που εφαρμόζονται με συγκεκριμένο τρόπο, ως εξής:

```
λίστα.μέθοδος(...)
```

Οι πιο σημαντικές μέθοδοι για λίστες είναι οι εξής:

- `append(`*στοιχείο*`)` → Προσθέτει το *στοιχείο* στο τέλος της λίστας.
- `index(`*στοιχείο*`)` → Επιστρέφει τον δείκτη του πρώτου στοιχείου του οποίου η τιμή ισούται με το *στοιχείο*. Αν το στοιχείο δεν υπάρχει στη λίστα, προκαλείται συγκεκριμένο σφάλμα *(περισσότερα για αυτό σε επόμενη ενότητα που αναλύει τον Χειρισμό Εξαιρέσεων)*.
- `insert(`*δείκτης*`, `*στοιχείο*`)` → Εισάγει το *στοιχείο* στη θέση που καθορίζει ο *δείκτης*. Όταν ένα στοιχείο εισάγεται στη λίστα, η λίστα επεκτείνεται σε μέγεθος για να συμπεριλάβει το νέο στοιχείο. Το στοιχείο που βρισκόταν προηγουμένως στη συγκεκριμένη θέση, και όλα τα στοιχεία μετά από αυτό, μετατοπίζονται κατά μία θέση προς το τέλος της λίστας.
  - Αν καθοριστεί δείκτης πέρα από το τέλος της λίστας, το στοιχείο θα προστεθεί στο τέλος της λίστας.
  - Αν ο δείκτης είναι αρνητικός και αναφέρεται σε μία μη-έγκυρη θέση, το στοιχείο θα προστεθεί στην αρχή της λίστας.
  - Αν ο δείκτης είναι αρνητικός και αναφέρεται σε έγκυρη θέση (σύμφωνα με την “αρνητική αρίθμηση” των δεικτών, όπου το -1 αναφέρεται στην τελευταία θέση, το -2 στην προτελευταία θέση κοκ), τότε το στοιχείο θα προστεθεί στην προηγούμενη θέση από αυτή στην οποία αναφέρεται ο αρνητικός δείκτης. Αυτό συμβαίνει γιατί πρώτα δημιουργείται μια νέα θέση στη λίστα για να προστεθεί το νέο στοιχείο (μετατοπίζοντας τα στοιχεία δεξιά της καθοριζόμενης θέσης κατά μία θέση δεξιά) και στη συνέχεια προστίθεται το στοιχείο. Το ίδιο ακριβώς συμβαίνει και στην περίπτωση καθορισμού θέσης με θετικό δείκτη, αλλά στην περίπτωση αρνητικού δείκτη τελικά το νεοεισερχόμενο στοιχείο “καταλήγει” μία θέση αριστερά της θέσης που καθορίστηκε με τον αρνητικό δείκτη.
- `sort()` → Ταξινομεί τα στοιχεία της λίστας σε αύξουσα σειρά (από τη μικρότερη προς τη μεγαλύτερη τιμή ή αλφαβητικά αν πρόκειται για συμβολοσειρές).
- `sort(reverse=True)` → Ταξινομεί τα στοιχεία της λίστας σε φθίνουσα σειρά.
- `remove(`*στοιχείο*`)` → Αφαιρεί από τη λίστα το *στοιχείο* στην πρώτη του εμφάνιση. Αν το στοιχείο δεν υπάρχει στη λίστα, προκαλείται συγκεκριμένο σφάλμα.
- `reverse()` → Αντιστρέφει τη σειρά των στοιχείων στη λίστα.

Π.χ.

```
>>> list12 = [10, 20, 30, 40, 50]
>>> list12.append(60)
>>> print(list12)
[10, 20, 30, 40, 50, 60]
>>> list12.index(30)
2
>>> list12.index(35)
```

```
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    list12.index(35)
ValueError: 35 is not in list
>>> list12.insert(2,100)
>>> print(list12)
[10, 20, 100, 30, 40, 50, 60]
>>> list12.insert(10,200)
>>> print(list12)
[10, 20, 100, 30, 40, 50, 60, 200]
>>> list12.insert(-3,300)
>>> print(list12)
[10, 20, 100, 30, 40, 300, 50, 60, 200]
>>> list12.insert(-15,400)
>>> print(list12)
[400, 10, 20, 100, 30, 40, 300, 50, 60, 200]
>>> list12.sort()
>>> print(list12)
[10, 20, 30, 40, 50, 60, 100, 200, 300, 400]
>>> list12.remove(60)
>>> print(list12)
[10, 20, 30, 40, 50, 100, 200, 300, 400]
>>> list12.remove(70)
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    list12.remove(70)
ValueError: list.remove(x): x not in list
>>> list12.reverse()
>>> print(list12)
[400, 300, 200, 100, 50, 40, 30, 20, 10]
```

Ενσωματωμένες συναρτήσεις για λίστες και σχετικές εντολές

Υπάρχουν τρεις πολύ χρήσιμες ενσωματωμένες συναρτήσεις που μπορούν να εφαρμοστούν σε αλληλουχίες τιμών, άρα και σε λίστες:

- `min(`*`λίστα`*`)` → Επιστρέφει το ελάχιστο της λίστας
- `max(`*`λίστα`*`)` → Επιστρέφει το μέγιστο της λίστας
- `sum(`*`λίστα`*`)` → Επιστρέφει το άθροισμα των στοιχείων της λίστας

Η εντολή `del` διαγράφει ένα στοιχείο από συγκεκριμένη θέση μιας λίστας, ως εξής:

```
del λίστα(δείκτης)
```

Π.χ., για τη λίστα `list12` του τελευταίου παραδείγματος: [400, 300, 200, 100, 50, 40, 30, 20, 10]

```
>>> del list12[1]
>>> print(list12)
[400, 200, 100, 50, 40, 30, 20, 10]
```

Αντίγραφα λιστών

Όταν μία λίστα εκχωρείται σε μια άλλη λίστα χρησιμοποιώντας τον τελεστή εκχώρησης, τότε δε δημιουργείται αντίγραφο της αρχικής λίστας, απλά η λίστα πλέον έχει δύο ονόματα. Αυτό φαίνεται

από το ακόλουθο παράδειγμα:

```
>>> list1 = [10, 20, 30, 40]
>>> list2 = list1
>>> print(list2)
[10, 20, 30, 40]
>>> list1[1] = 99
>>> print(list1)
[10, 99, 30, 40]
>>> print(list2)
[10, 99, 30, 40]
```

δηλαδή, μεταβάλλοντας κάποια τιμή στη `list1`, μεταβάλλεται και η `list2`. Άρα, ουσιαστικά με αυτόν τον τρόπο δεν δημιουργείται αντίγραφο της λίστας. Αντίγραφο μιας λίστας (π.χ. της `list1`) μπορεί να δημιουργηθεί με τους εξής τρόπους:

i) Με χρήση της μεθόδου `append` μέσα σε βρόχο `for`:

```
list2 = []
for item in list1:
    list2.append(item)
```

ii) Με χρήση του τελεστή `+` σε κενή λίστα:

```
list2 = []
list2 += list1
```

ή με μία εντολή:

```
list2 = [] + list1
```

iii) Με χρήση του τελεστή `:` ως εξής:

```
list2 = list1[:]
```

iv) Με χρήση της συνάρτησης `list`:

```
list2 = list(list1)
```

Σημείωση: Ο τελεστής `+` πραγματοποιεί συνένωση λιστών (περίπτωση (ii) προηγουμένως). Η πρακτική διαφορά του από τη μέθοδο `append`, εκτός του ότι ο τελεστής συνένωσης `+` εφαρμόζεται μεταξύ λιστών ενώ η `append` δέχεται στοιχείο λίστας, είναι ότι με τον τελεστή συνένωσης `+` μπορούν να προστεθούν πολλαπλά στοιχεία στο τέλος μιας λίστας, ενώ με τη συνάρτηση `append` προστίθεται ένα μόνο στοιχείο, ακόμα και αν αυτό αποτελεί ξεχωριστή λίστα. Άρα, π.χ.:

```
>>> list3 = [1, 2, 3]
>>> list4 = list3 + [4, 5]
>>> list4
[1, 2, 3, 4, 5]
>>> list4.append([6, 7, 8])
>>> list4
[1, 2, 3, 4, 5, [6, 7, 8]]
```

δηλαδή η λίστα `[6, 7, 8]` που δέχεται η `append` εκλαμβάνεται ως ένα στοιχείο και προστίθεται

ως το 6ο στοιχείο της λίστας `list4`. Η λίστα αυτή είναι πλέον δισδιάστατη, αφού περιέχει στοιχείο-λίστα – οι δισδιάστατες λίστες θα αναλυθούν παρακάτω).

Επίσης, μία άλλη σημαντική διαφορά του + από τη συνάρτηση `append` είναι ότι το αποτέλεσμα μιας έκφρασης συνένωσης πρέπει να εκχωρηθεί σε κάποια λίστα (όπως στο παραπάνω παράδειγμα στη `list4`), ενώ η κλήση της `append` λειτουργεί στην ίδια τη λίστα με την οποία καλείται. Ουσιαστικά, ως προς αυτή τη λειτουργικότητα, η `append` είναι παραπλήσια του τελεστή `+=`, ο οποίος συνενώνει μια λίστα με κάποια άλλη και εκχωρεί το αποτέλεσμα στην ίδια τη λίστα. Μάλιστα, υπάρχει διαφορά μεταξύ τού:

```
λίστα_Α = λίστα_Α + λίστα_Β
```

και του:

```
λίστα_Α += λίστα_Β
```

Η `λίστα_Α` και στις δύο περιπτώσεις θα περιέχει τα ίδια στοιχεία, όμως στην πρώτη περίπτωση δημιουργείται καινούρια λίστα για την αποθήκευση της συνένωσης (με το όνομα `λίστα_Α` και πάλι), ενώ στη δεύτερη περίπτωση η συνένωση αποθηκεύεται στην ήδη υπάρχουσα `λίστα_Α`. Άρα, συνήθως η δεύτερη εναλλακτική είναι προτιμότερη.

Κάποια συχνά σφάλματα:

i)

```
>>> list5 = [10, 20, 30]
>>> list5 = list5 + 40
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    list5 = list5 + 40
TypeError: can only concatenate list (not "int") to list
>>> list5 = list5 + [40]
>>> list5
[10, 20, 30, 40]
```

Δηλαδή, για να προστεθεί ένα μόνο στοιχείο στο τέλος μιας λίστας με τον τελεστή `+`, θα πρέπει να προστεθεί με τη μορφή λίστας (π.χ., `[40]`), παρόλο που πρόκειται για ένα μόνο στοιχείο.

ii)

```
>>> list6 = [0, 1, 2]
>>> list7 = list6.append(3)  # προβληματική κλήση
>>> print(list7)
None
>>> print(list6)
[0, 1, 2, 3]
```

Δηλαδή, η κλήση της `append` δεν έχει νόημα να εκχωρηθεί σε κάποια νέα λίστα, αφού δεν επιστρέφει κάτι αλλά απλά προσθέτει κάποιο στοιχείο στη λίστα με την οποία καλείται (εδώ τη `list6`).

Λίστες ως ορίσματα συναρτήσεων (Συναρτήσεις που δέχονται λίστες)

Μια συνάρτηση μπορεί να έχει (και) λίστες ως παραμέτρους εισόδου, όπως ακριβώς ισχύει και με τις μεταβλητές. Π.χ, η παρακάτω συνάρτηση δέχεται μία λίστα και επιστρέφει το γινόμενο των στοιχείων της:

```
def list_prod(values):
    prod = 1
    for n in values:
        prod *= n
    return prod
```

Χρήση της συνάρτησης:

```
>>> numbers = [1, 34, 20, -23, 2.6, 3.2 -2.5]
>>> print('Το γινόμενο των στοιχείων είναι', list_prod(numbers))
Το γινόμενο των στοιχείων είναι -28464.800000000007
```

Σημείωση: Όπως αναφέρθηκε στην προηγούμενη ενότητα, όταν καλείται μια συνάρτηση που έχει παραμέτρους εισόδου, τα ορίσματα που μεταβιβάζονται στη συνάρτηση αποθηκεύονται σε νέες, τοπικές μεταβλητές της συνάρτησης. Επομένως, οποιαδήποτε τροποποίηση γίνει στις παραμέτρους εισόδου της συνάρτησης μέσα στη συνάρτηση, παραμένει σε αυτήν και δεν “μεταφέρεται” στις μεταβλητές που χρησιμοποιήθηκαν ως ορίσματα (ο τρόπος αυτός μεταβίβασης ορισμάτων ονομάζεται “μεταβίβαση με τιμή” (pass-by-value). Στην περίπτωση της μεταβίβασης κάποιας λίστας όμως, φαινομενικά συμβαίνει το αντίθετο, δηλαδή οποιαδήποτε τροποποίηση κάποιου στοιχείου της λίστας μέσα στη συνάρτηση “μεταφέρεται” και στη λίστα που χρησιμοποιήθηκε ως όρισμα κατά την κλήση της συνάρτησης. Αυτό μοιάζει με τον τρόπο μεταβίβασης ορισμάτων άλλων γλωσσών προγραμματισμού γνωστό ως “μεταβίβαση με αναφορά” (pass-by-reference), όπου αντί για την τιμή του ορίσματος, μεταβιβάζεται η αναφορά (reference) στη θέση μνήμης που είναι αποθηκευμένη η τιμή του ορίσματος. Στην Python δεν συμβαίνει ακριβώς αυτό, αφού ακόμα και τα αντικείμενα (π.χ., οι λίστες) μεταβιβάζονται “με τιμή”. Όμως, ένα αντικείμενο στην Python είναι ουσιαστικά μια αναφορά στη θέση μνήμης που περιέχονται τα στοιχεία τού αντικειμένου. Δηλαδή, το όνομα μιας λίστας είναι μια αναφορά στη θέση μνήμης στην οποία είναι αποθηκευμένες οι τιμές της λίστας (χοντρικά). Επομένως, όταν μεταβιβάζεται μια λίστα σε μια συνάρτηση, γίνεται “μεταβίβαση με τιμή” της λίστας, η οποία όμως δεν είναι τίποτε άλλο από τη θέση μνήμης της αρχικής λίστας. Το όρισμα αυτό *αντιγράφεται σε μία τοπική λίστα*, που είναι και αυτή μια αναφορά στη θέση μνήμης της αρχικής λίστας. Δηλαδή αυτό που αντιγράφεται δεν είναι ολόκληρη η λίστα, αλλά η αναφορά στη θέση της (άρα η αρχική λίστα έχει πλέον δύο ονόματα, αυτό του ορίσματος και αυτό της τοπικής παραμέτρου εισόδου της συνάρτησης). Επομένως, οποιαδήποτε αλλαγή στην τοπική αυτή λίστα, πραγματοποιείται και στην αρχική λίστα.

Παράδειγμα 1:

```
def foo(foo_list):
    print(foo_list)
    foo_list.append(100)  # Η foo_list τροποποιείται
    print(foo_list)

def main():
    my_list = [1, 2, 3, 4, 5]
    print('my_list before foo: ', my_list)
    foo(my_list)
    print('my_list after foo: ', my_list)
```

```
main()
```

Έξοδος προγράμματος:

```
my_list before foo:  [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 100]
my_list after foo:  [1, 2, 3, 4, 5, 100]
```

Παράδειγμα 2:

```
def bar(bar_list):
    print(bar_list)
    bar_list.append(100)     # Η bar_list τροποποιείται
    bar_list = [10, 20, 30]  # Στη bar_list εκχωρείται νέα λίστα
    print(bar_list)

def main():
    my_list = [1, 2, 3, 4, 5]
    print('my_list before bar: ', my_list)
    bar(my_list)
    print('my_list after bar: ', my_list)

main()
```

Το παράδειγμα αυτό δείχνει ότι στην Python, ακόμα και για τα αντικείμενα, όπως είναι οι λίστες, δεν γίνεται "μεταβίβαση με αναφορά" αλλά γίνεται "μεταβίβαση με τιμή". **Αν** η Python λειτουργούσε με "μεταβίβαση με αναφορά", το πρόγραμμα θα εκτύπωνε:

```
my_list before bar:  [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[10, 20, 30]
my_list after bar:  [10, 20, 30]   <-- ΔΕΝ ΘΑ ΕΚΤΥΠΩΘΕΙ ΑΥΤΟ
```

γιατί η λίστα μέσα στη συνάρτηση, αφού τροποποιηθεί με την προσθήκη της τιμής 100 στο τέλος της, αλλάζει τιμές αφού εκχωρείται σε νέα λίστα, παίρνοντας πλέον τις τιμές `[10, 20, 30]`, και αφού η `bar_list` θα ήταν η αναφορά στη λίστα `my_list`, η τελευταία αλλαγή θα άλλαζε και τη `my_list`. Όμως, το πρόγραμμα θα εκτυπώσει το εξής:

```
my_list before bar:  [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[10, 20, 30]
my_list after bar:  [1, 2, 3, 4, 5, 100]
```

γιατί η `bar_list` είναι ένα καινούριο (δεύτερο) όνομα για την τιμή με όνομα `my_list`, οπότε η προσθήκη της τιμής 100 σε αυτό το όνομα (`bar_list`) είναι κοινή και για τη `my_list`, όμως η εντολή `bar_list = [10, 20, 30]` που εκχωρεί στο όνομα `bar_list` μια καινούρια λίστα, *δημιουργεί μία καινούρια αναφορά* για το όνομα `bar_list` σε μια νέα τιμή ([10, 20, 30]), ενώ το όνομα `my_list` συνεχίζει να αναφέρεται στην προηγούμενη τιμή του.

Άρα, συμπερασματικά, η Python λειτουργεί αποκλειστικά με "μεταβίβαση με τιμή" (pass-by-value), παρόλο που στην περίπτωση αντικειμένων (π.χ. λιστών) αυτό δεν είναι προφανές. Οποιεσδήποτε

τροποποιήσεις συγκεκριμένων στοιχείων μιας λίστας μέσα στη συνάρτηση "μεταφέρονται" στην αρχική λίστα, ενώ τροποποιήσεις σε επίπεδο λίστας (πχ εκχώρηση της λίστας σε νέα λίστα) παραμένουν αποκλειστικά μέσα στη συνάρτηση.

Ουσιαστικά, δεν υπάρχει κάποια πραγματική διαφορά μεταξύ των περιπτώσεων μεταβίβασης λίστας και μεταβίβασης μεταβλητών, αφού στην Python ακόμα και οι μεταβλητές είναι αντικείμενα. Όπως προαναφέρθηκε, και οι μεταβλητές, και οι λίστες, μεταβιβάζονται με τιμή. Μια εκχώρηση της μορφής: `μεταβλητή_εισόδου = νέα_τιμή` *μέσα σε μια συνάρτηση είναι αντίστοιχη της:* `λίστα_εισόδου = [νέες τιμές]`*, η οποία παρουσιάστηκε στο 2ο παράδειγμα παραπάνω. Και στις δύο περιπτώσεις, οι οποιεσδήποτε τροποποιήσεις παραμένουν αποκλειστικά μέσα στη συνάρτηση. Η μόνη περίπτωση να μεταφερθούν κάποιες τροποποιήσεις εκτός συνάρτησης, είναι να γίνουν αλλαγές σε επίπεδο στοιχείων της λίστας, δηλαδή τροποποιήσεις στοιχείων, προσθήκη νέων στοιχείων ή διαγραφή στοιχείων.*

Συναρτήσεις που επιστρέφουν λίστες

Συναρτήσεις που δημιουργούν λίστες, μπορούν να επιστρέφουν αναφορές στις λίστες αυτές. Π.χ.,:

```
def get_values():
    values = [ ]   # Δημιουργία μιας κενής λίστας
    more = 'N'
    while more == 'N':
        # Είσοδος τιμών από τον χρήστη και προσθήκη τους στη λίστα
        num = int(input('Δώσε έναν αριθμό: '))
        values.append(num)
        more = input('Θέλεις να δώσεις άλλον αριθμό; (N/O) ')
    return values   # Επιστροφή της λίστας
```

Χρήση της συνάρτησης:

```
numbers = get_values()  # Είσοδος μιας λίστας που περιέχει τιμές
print('Οι αριθμοί στη λίστα είναι:')
print(numbers)   # Εμφάνιση των τιμών της λίστας
```

Προγραμματιστικές Εφαρμογές:

(Για την εκπαιδευτικά καλύτερη ανάλυση και υλοποίηση των αλγορίθμων που ακολουθούν, δεν θα γίνει χρήση συγκεκριμένων ενσωματωμένων συναρτήσεων όπως min, sum κτλ., καθώς και μεθόδων για διαχείριση λιστών που παρέχει η Python, π.χ. sort κτλ.)

- ***Ταξινόμηση λίστας με τον αλγόριθμο της επιλογής (selection sort):***

1. Ξεκίνα από την πρώτη θέση.
2. Βρες το ελάχιστο στοιχείο από τη θέση αυτή και μετά.
3. Ενάλλαξε (swap) τα στοιχεία της θέσης που ξεκίνησες και της θέσης του ελάχιστου.
4. Πήγαινε στην επόμενη θέση και επανάλαβε το "Βήμα 2", μέχρι να φτάσεις στην προτελευταία θέση.

Υλοποίηση με συνάρτηση:

```
def selection(v):
    for i in range(len(v)-1):
        min_index = i  # έστω ότι το στοιχείο αυτό είναι το ελάχιστο
        # αναζήτηση του ελάχιστου:
        for j in range(i+1,len(v)):
            if v[j] < v[min_index]:
                min_index = j  # η θέση του ελάχιστου στοιχείου
        v[i], v[min_index] = v[min_index], v[i]
```

Η παραπάνω συνάρτηση αλλάζει απευθείας τη λίστα και επομένως μετά την κλήση της τα στοιχεία της αρχικής λίστας θα έχουν αντικατασταθεί από τα ταξινομημένα (βλ. παραπάνω για τον μηχανισμό κλήσης των συναρτήσεων).

Αν θέλαμε να κρατήσουμε την αρχική λίστα θα έπρεπε να δουλέψουμε με ένα αντίγραφο της και να επιστρέψουμε το αντίγραφο αυτό. Η συνάρτηση `selection` θα έπρεπε να γραφτεί ως εξής:

```
def selection(v):
    v = v[:]  # τώρα η v είναι αντίγραφο της αρχικής λίστας
    for i in range(len(v)-1):
        min_index = i  # έστω ότι το στοιχείο αυτό είναι το ελάχιστο
        # αναζήτηση του ελάχιστου:
        for j in range(i+1,len(v)):
            if v[j] < v[min_index]:
                min_index = j  # η θέση του ελάχιστου στοιχείου
        v[i], v[min_index] = v[min_index], v[i]
    return v
```

- ***Ταξινόμηση λίστας με τον αλγόριθμο εισαγωγής (insertion sort):***

1. Ξεκίνα από το δεύτερο στοιχείο.
2. Σύγκρινε το στοιχείο με ένα-ένα τα στοιχεία που βρίσκονται αριστερά του **μέχρι να βρεις** (άρα while-loop) κάποιο μικρότερό του, και όσο βρίσκεις κάποιο μεγαλύτερό του, ενάλλασσέ τα.
3. Πήγαινε στο επόμενο στοιχείο και επανάλαβε το “Βήμα 2”.

Υλοποίηση με συνάρτηση:

```
def insertion(v):
    for i in range(1,len(v)):
        while i>0 and v[i-1]>v[i]:      # Δε δημιουργείται πρόβλημα με την
            v[i], v[i-1] = v[i-1], v[i] # τροποποίηση της τιμής του i μέσα
            i -= 1                      # στο for! Στην επόμενη επανάληψη θα
    return v                            # πάρει την επόμενη τιμή της range.
```

Και εδώ, η συνάρτηση ταξινομεί άμεσα τη λίστα εισόδου. Αν θέλαμε να τη διατηρήσουμε ως είχε, θα πρέπει να δουλέψουμε με αντίγραφό της όπως και προηγουμένως.

- ***Γραμμική αναζήτηση (linear search):***

→ Ξεκινώντας από το πρώτο στοιχείο, σύγκρινε ένα-ένα τα στοιχεία του διανύσματος με το προς αναζήτηση στοιχείο (**key**), μέχρι να το βρεις.

Υλοποίηση με συνάρτηση:

```
def linear_search(v, key):
    loc = 0     # η τρέχουσα θέση αναζήτησης
    pos = False # η θέση του στοιχείου αναζήτησης
                # θα επιστρέψει False αν δε βρεθεί
    hit = False  # flag για το αν έχει βρεθεί το προς αναζήτηση στοιχείο
    while loc < len(v) and not hit:
        if v[loc]==key:
            pos = loc
            hit = True
        else:
            loc += 1
    return pos
```

- ***Δυαδική αναζήτηση (binary search):***

→ *Προϋπόθεση: Η λίστα πρέπει να είναι ταξινομημένη!*

1. Σύγκρινε το αναζητούμενο στοιχείο με το μεσαίο στοιχείο της λίστας
2. Όσο αυτά διαφέρουν και υπάρχουν ακόμα στοιχεία στη λίστα:

- αν είναι μικρότερο του στοιχείου, επανάλαβε το “Βήμα 1” για το αριστερό τμήμα της λίστας.
- αν είναι μεγαλύτερο του στοιχείου, επανάλαβε το “Βήμα 1” για το δεξί τμήμα της λίστας.

Υλοποίηση με συνάρτηση:

```
def binary_search(v, key):
    left = 0          # το αριστερό άκρο της τρέχουσας λίστας
    right = len(v)-1  # το δεξί άκρο της τρέχουσας λίστας
    pos = False
    hit = False
    while left <= right and not hit:
        mid = (left+right)//2
        if v[mid]==key:
            pos = mid
            hit = True
        elif key < v[mid]:
            right = mid-1
        else:
            left = mid+1
    return pos
```

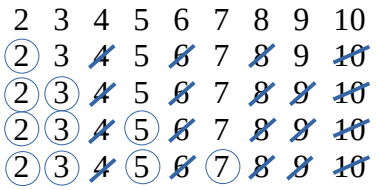
- ***Το κόσκινο του Ερατοσθένη (3ος αιώνας π.Χ.):***

Αλγόριθμος για τη εύρεση των πρώτων αριθμών από το 2 έως το Ν.

1. Γράψε όλους τους αριθμούς από το 2 έως το Ν.

2. Κύκλωσε τον πρώτο διαθέσιμο αριθμό (θα είναι πρώτος).
3. Διέγραψε τα πολλαπλάσιά του.
4. Πήγαινε στο “Βήμα 2”.

π.χ.: για N=10 έχουμε:



Υλοποίηση με συνάρτηση:

```
import math
def eratosthenis(N):
    if N < 2:
        return []
    P = list(range(N+1))  # Η λίστα των αριθμών
    P[0] = False # Όσοι διαγράφονται θα γίνονται False. Η πρώτη θέση με δείκτη
                 # 0 δε μας ενδιαφέρει. Δημιουργήθηκε για να συμβαδίζουν οι
                 # υπόλοιποι δείκτες με τις τιμές των αριθμών (για ευκολία).
    P[1] = False # Το 1 ΔΕΝ είναι πρώτος.
    for i in range(2, math.ceil(math.sqrt(N))):
        if P[i]:                        # Αν το i δεν έχει διαγραφεί
            for j in range(i*i,N+1,i): # πήγαινε σε κάθε πολλαπλάσιο του i
                P[j] = False           # και διέγραψέ το

    # Αποθήκευση των πρώτων που βρέθηκαν, στη λίστα primes:
    primes = []
    for p in P:
        if p:
            primes.append(p)

    return primes
```

Δισδιάστατες Λίστες

Οι δισδιάστατες λίστες είναι λίστες που τα στοιχεία τους είναι άλλες λίστες. Π.χ.,

```
numbers = [ [2, 4, 6, 8, 10], [1, 3, 5, 7, 9], [11, 12, 13, 14, 15] ]
```

Το 1ο στοιχείο της λίστας `numbers` είναι η λίστα: `[2, 4, 6, 8, 10]`
Το 2ο στοιχείο της λίστας `numbers` είναι η λίστα: `[1, 3, 5, 7, 9]`
Το 3ο στοιχείο της λίστας `numbers` είναι η λίστα: `[11, 12, 13, 14, 15]`

Επομένως, η λίστα `numbers` θα μπορούσε να αναπαρασταθεί με τη μορφή ενός δισδιάστατου πίνακα, ως εξής:

```
2    4    6    8    10
1    3    5    7    9
11   12   13   14   15
```

άρα αποτελεί μια δισδιάστατη λίστα. Για την πρόσβαση σε συγκεκριμένο στοιχείο μιας δισδιάστατης λίστας απαιτούνται δύο δείκτες, ένας για τη γραμμή και ένας για τη στήλη του στοιχείου:

```
λίστα[γραμμή][στήλη]
```

Π.χ., για την παραπάνω λίστα `numbers`:

```
>>> print(numbers[2][4])
15
>>> print(numbers[0][2])
6
```

Δημιουργία και αρχικοποίηση δισδιάστατης λίστας:

Μια δισδιάστατη λίστα μπορεί να δημιουργηθεί και αρχικοποιηθεί με διάφορους τρόπους. Για την Πληροφορική Ι, χρησιμοποιούμε τη μέθοδο append, ως εξής:

```
λίστα = []
for i in range(πλήθος_γραμμών):
    λίστα.append( [τιμή] * πλήθος_στηλών )
```

Άλλους τρόπους για αρχικοποιήσεις δισδιάστατων λιστών θα δούμε στην Πληροφορική ΙΙ.

Αφού έχει δημιουργηθεί μια λίστα, μπορεί μετά κανείς να διατρέξει τα στοιχεία της με ένθετο `for` και να τους εκχωρήσει νέες τιμές:

```
for i in range(πλήθος_γραμμών):
    for j in range(πλήθος_στηλών):
        λίστα[i][j] = νέα_τιμή
```

Με τον τρόπο αυτό, μπορεί να αρχικοποιηθεί το κάθε στοιχείο της λίστας σε διαφορετική τιμή.

- Σε μια δισδιάστατη λίστα, με τη χρήση *ενός* μόνο δείκτη γίνεται αναφορά σε ολόκληρη την αντίστοιχη γραμμή της λίστας:

  ```
  λίστα[δείκτης_γραμμής]
  ```

  Π.χ.:

  ```
  >>> numbers[1]
  [1, 3, 5, 7, 9]
  >>> numbers[2]
  [11, 12, 13, 14, 15]
  ```

- Σε μια δισδιάστατη λίστα, με τη χρήση του τελεστή `in` γίνεται αναφορά ανά γραμμή της

λίστας:

```
for γραμμή in λίστα
```

Π.χ.:

```
>>> for row in numbers:
      print(row)
[2, 4, 6, 8, 10]
[1, 3, 5, 7, 9]
[11, 12, 13, 14, 15]
```

- Το πλήθος *των γραμμών* μιας δισδιάστατης λίστας δίνεται από την εντολή: `len(λίστα)`

- Το πλήθος *των στηλών* μιας δισδιάστατης λίστας δίνεται από την εντολή: `len(λίστα[0])`

δηλαδή, το πλήθος των στηλών δίνεται από το πλήθος των στοιχείων της πρώτης γραμμής (το `λίστα[0]` είναι η 1η γραμμή της λίστας).

Για να ισχύει αυτό για μια λίστα, θα πρέπει η λίστα να έχει τον ίδιο αριθμό στοιχείων σε όλες τις γραμμές της. Αυτό γενικά, δεν είναι απαραίτητο να ισχύει, δηλαδή μπορεί να υπάρχει π.χ. η ακόλουθη λίστα:

```
numbers2 = [ [2, 4, 6, 8], [1, 3], [11, 12, 13] ]
```

η οποία στην 1η γραμμή έχει 4 στοιχεία, στη 2η γραμμή 2 στοιχεία και στην 3η γραμμή 3 στοιχεία.

Επίσης, όπως και στις κανονικές λίστες, και μία δισδιάστατη λίστα μπορεί να περιέχει στοιχεία διαφόρων τύπων:

```
results = [['Student1', 'Student2', 'Student3'], [1, 2, 1], [8, 7.5, 3.5]]
```

Σύμφωνα με τα παραπάνω, για να προστεθεί π.χ. ένα στοιχείο `'Student4'` στην παραπάνω λίστα με χρήση της μεθόδου `append`:

```
results[0].append('Student4')
```

Αντίγραφα δισδιάστατων λιστών

Όταν εφαρμόζουμε τις μεθόδους αντιγραφής που είδαμε για “μονοδιάστες” λίστες σε λίστες 2 διαστάσεων δημιουργείτα ένα *ρηχό αντίγραφο (shallow copy)*. Δηλαδή για τις εσωτερικές λίστες αντιγράφεται η διεύθυνση μνήμης τους και όχι τα στοιχεία τους.

```
A = [[1, 2, 3], [4, 5, 6]]
B = A[:]
```

Αν αλλάξουμε κάποιο στοιχείο της B, αλλάζει και το αντίστοιχο στοιχείο της `A`:

```
B[0][1] = 99
print(A[0][1])  # θα τυπώσει επίσης 99
```

Αυτό συμβαίνει γιατί οι εσωτερικές λίστες `B[0]` και `B[1]` αναφέρονται στα ίδια αντικείμενα `A[0]` και `A[1]` αντίστοιχα. Αντιγράφτηκε η διεύθυνση της λίστας `A[0]` και όχι τα στοιχεία της με συνέπεια να αναφερόμαστε στο ίδιο αντικείμενο στη μνήμη.

Αν θέλουμε να αντιγράψουμε τις λίστες και ως προς τα στοιχεία τους (deep copy), θα πρέπει να εφαρμόσουμε την αντιγραφή απευθείας στις εσωτερικές λίστες, σύμφωνα με αυτά που είδαμε για τις μονοδιάστατες λίστες. Π.χ. για την αντιγραφή της `list1` στη`list2`

i) Με χρήση της μεθόδου `append` μέσα σε βρόχο `for` που διατρέχει τη `list1`:

```
list2 = []
for inner_list in list1:
    list2.append(inner_list[:]) # ή list2.append(list(inner_list))
```

ii) Με απευθείας εκχώρηση αντιγράφων των εσωτερικών λιστών:

```
list2 = [[]] * len(list1)
for i in range(len(list1)):
     list2[i] = list1[i][:]  # ή list2[i] = list(list1[i])
```

iii) Με χρήση του τελεστή + ή += ως εξής:

```
list2 = [[]] * len(list1)
for i in range(len(list1)):
    list2[i] = list2[i] + list1[i]  # list2[i] += list1[i]
```

Άλλους τρόπους αντιγραφής δισδιάστατων λιστών θα δούμε στην Πληροφορική ΙΙ.

Παραδείγματα

1. Άθροισμα και μέσος όρων όλων των στοιχείων μιας δισδιάστατης λίστας:

```
numbers = [ [2, 4, 6, 8, 10], [1, 3, 5, 7, 9], [11, 12, 13, 14, 15] ]
total = 0
for r in range(len(numbers)):
    for c in range(len(numbers[r])):
        total += numbers[r][c]
print('Το άθροισμα των στοιχείων είναι', total)
average = total / (len(numbers)*len(numbers[0]))
print('Ο μέσος όρος των στοιχείων είναι', average)
```

ή:

```
numbers = [ [2, 4, 6, 8, 10], [1, 3, 5, 7, 9], [11, 12, 13, 14, 15] ]
total = 0
for r in range(len(numbers)):
    for v in numbers[r]:
        total += v
print('Το άθροισμα των στοιχείων είναι', total)
average = total / (len(numbers)*len(numbers[0]))
print('Ο μέσος όρος των στοιχείων είναι', average)
```

ή:

```
numbers = [ [2, 4, 6, 8, 10], [1, 3, 5, 7, 9], [11, 12, 13, 14, 15] ]
total = 0
for row in numbers:
    total += sum(row)
print('Το άθροισμα των στοιχείων είναι', total)
average = total / (len(numbers)*len(numbers[0]))
print('Ο μέσος όρος των στοιχείων είναι', average)
```

Σημείωση: Ο υπολογισμός του συνολικού πλήθους των στοιχείων της λίστας με την έκφραση `len(numbers)*len(numbers[0])` είναι σωστός μόνο στην περίπτωση που η λίστα έχει τον ίδιο αριθμό στοιχείων σε κάθε γραμμή. Για πιο γενικές περιπτώσεις, θα πρέπει να χρησιμοποιηθεί ένας επιπλέον μετρητής, π.χ., για την τρίτη εκδοχή του προγράμματος:

```
numbers = [ [2, 4, 6, 8, 10], [1, 3, 5, 7, 9], [11, 12, 13, 14, 15] ]
total = 0
nelements = 0
for row in numbers:
    total += sum(row)
    nelements += len(row)
print('Το άθροισμα των στοιχείων είναι', total)
average = total/nelements
print('Ο μέσος όρος των στοιχείων είναι', average)
```

2. Άθροισμα και μέσος όρων των στοιχείων κάθε γραμμής μιας δισδιάστατης λίστας:

```
numbers = [ [2, 4, 6, 8, 10], [1, 3, 5, 7, 9], [11, 12, 13, 14, 15] ]
for r in range(len(numbers)):
    total = 0
    for c in range(len(numbers[r])):
        total += numbers[r][c]
    print('Το άθροισμα των στοιχείων της γραμμής', r, είναι', total)
    average = total / len(numbers[0])
    print('Ο μέσος όρος των στοιχείων της γραμμής', r, είναι', average)
```

ή:

```
numbers = [ [2, 4, 6, 8, 10], [1, 3, 5, 7, 9], [11, 12, 13, 14, 15] ]
for row in numbers:
    total = sum(row)
    print('Το άθροισμα των στοιχείων της γραμμής', row, είναι', total)
    average = total / len(numbers[0])
    print('Ο μέσος όρος των στοιχείων της γραμμής', row, είναι', average)
```

Πλειάδες (Tuples)

Οι πλειάδες αποτελούν αμετάβλητο τύπο αλληλουχίας. Έχουν σχεδόν ίδια χαρακτηριστικά με αυτά των λιστών, με βασική διαφορά ότι οι τιμές τους δε μεταβάλλονται (σε αντίθεση με αυτές των λιστών).

Άλλη βασική διαφορά: Δημιουργούνται με παρενθέσεις και όχι με αγκύλες:

```
πλειάδα = (τιμή_1, τιμή_2, τιμή_3, ... )
```

Π.χ.,

```
>>> tuple1 = (3, -2, 1.5, 0)
>>> print(tuple1)
(3, -2, 1.5, 0)
```

Για τη δημιουργία πλειάδας ενός στοιχείου, πρέπει να περιλαμβάνεται και το κόμμα στην εντολή, διαφορετικά δημιουργείται απλή μεταβλητή:

```
>>> tuple2 = (5,)
>>> print(tuple2)
(5,)
>>> not_a_tuple = (10)
>>> print(not_a_tuple)
10
```

Κατά τα άλλα, ισχύουν οι περισσότερες λειτουργίες που ισχύουν και στις λίστες:

- Διαχείριση με δείκτες (μόνο για ανάκτηση τιμών των στοιχείων)
- Μέθοδοι όπως η `index`
- Ενσωματωμένες συναρτήσεις όπως οι `len` , `min` και `max`
- Εκφράσεις τεμαχισμού
- Ο τελεστής `in`
- Οι τελεστές `+` και `*`

Λόγοι ύπαρξης των Πλειάδων:

- Υπολογιστική απόδοση: η επεξεργασία με πλειάδες είναι ταχύτερη από την επεξεργασία με λίστες.
- Ασφάλεια: αποθήκευση δεδομένων χωρίς τον κίνδυνο τροποποίησης (κατά λάθος ή με άλλον τρόπο) από τον κώδικα του προγράμματος.
- Υπάρχουν ορισμένες πράξεις στην Python οι οποίες απαιτούν τη χρήση πλειάδων.

Μετατροπή Πλειάδας σε Λίστα και το αντίθετο:

```
>>> my_tuple = (1, 2, 3)
>>> my_list = list(my_tuple)
>>> print(my_list)
```

```
[1, 2, 3]
>>> my_new_tuple = tuple(my_list)
>>> print(my_new_tuple)
(1, 2, 3)
```