ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ Η/Υ
σε Python

Διάλεξη 4: Λίστες

Μάρκος Ζάμπογλου, mzampoglou@aegean.gr

Λίστα (List)

Μια ακολουθία (**sequence**) ή συλλογή (**collection**) διατεταγμένων αντικειμένων

- Με αριθμημένες θέσεις
- Δυναμικό μέγεθος (μεγαλώνει και μικραίνει)
- Μπορεί να περιέχει αντικείμενα κάθε τύπου

```
empty_list1 = []
empty_list2 = list()
list_of_numbers = [123, 432, 432, 324, 523]
mixed_list = [123, "apples", [2.4, 3-2j], None]
```

Πράξεις λιστών

- Πρόσθεση (+) σημαίνει συνένωση:

```
print([3,4,2]+[1,"hello"])
```

```
[3, 4, 2, 1, 'hello']
```

- Πολλαπλασιασμός (*) σημαίνει πολλές συνενώσεις

```
print(["Καλή","μέρα"]*4)
```

```
['Καλή', 'μέρα', 'Καλή', 'μέρα', 'Καλή', 'μέρα', 'Καλή', 'μέρα']
```

Indexing

- Για να απευθυνθούμε στις τιμές μιας λίστας, είτε για να τις χρησιμοποιήσουμε είτε για να τις τροποποιήσουμε, χρησιμοποιούμε τη θέση τους στη λίστα
  - Η ενέργεια αυτή λέγεται *indexing*

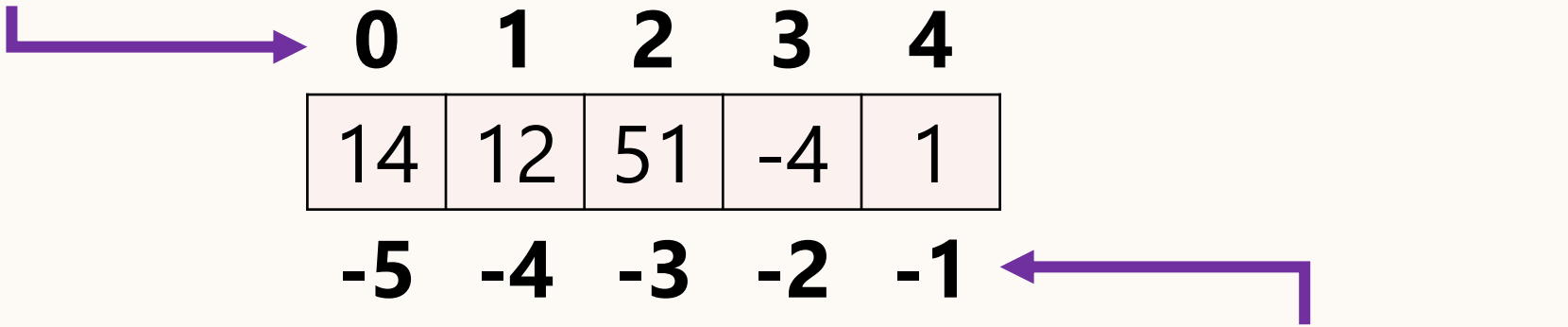
```
list_of_numbers = [123, 117, 432, 324, 523 , 218]
print(list_of_numbers[1])
117
print(list_of_numbers[-2])
523
```

Η αρίθμηση ξεκινάει από τη θέση 0

Η αρνητική αρίθμηση ξεκινάει από το τέλος

Zero-indexing και Backward indexing

Forward indexing:
Ξεκινάει από το 0



Backward indexing:
Ξεκινάει από το -1

Slicing 1/2

- Στην Python μπορούμε επίσης να κόψουμε ένα κομμάτι (*φέτα*) μιας λίστας που αποτελείται από παραπάνω από μία θέσεις
  - Η ενέργεια αυτή λέγεται *slicing*
  - Τα περιεχόμενα του slice ορίζονται από δύο ή τρεις αριθμούς, που χωρίζονται από μία ή δύο άνω και κάτω τελείες

```
list_of_numbers = [123, 117, 432, 324, 523, 218]

print(list_of_numbers[1:4])
[117, 432, 324]
print(list_of_numbers[0:-1:2])
[123, 432, 523]
print(list_of_numbers[5:2:-2])
[218, 324]
```

Η τελευταία θέση δε συμπεριλαμβάνεται στο slice

Η τρίτη τιμή ορίζει το βήμα. Αν δεν υπάρχει δεύτερο `:`, το βήμα θα είναι `1`

Αν το βήμα είναι αρνητικό, η θέση αρχής πρέπει να είναι μεγαλύτερη από τη θέση του τέλους

Slicing 2/2

```
list_of_numbers = [123, 117, 432, 324, 523, 218]

print(list_of_numbers[:3])
[123, 117, 432]
print(list_of_numbers[::2])
[123, 432, 523]
print(list_of_numbers[-15:100])
[123, 117, 432, 324, 523, 218]
print(list_of_numbers[100])
IndexError: list index out of range
```

Κενό σημαίνει «από την αρχή» ή «από το τέλος» αναλόγως αν το βήμα είναι θετικό ή αρνητικό

Με δύο διαδοχικές άνω και κάτω τελείες σηματοδοτούμε κενά στις πρώτες δύο θέσεις (άρα μόνο βήμα)

Οι δείκτες ενός slice μπορούν να βγαίνουν έξω από τα όρια της λίστας

Ο δείκτης μιας μεμονωμένης θέσης όμως, όχι

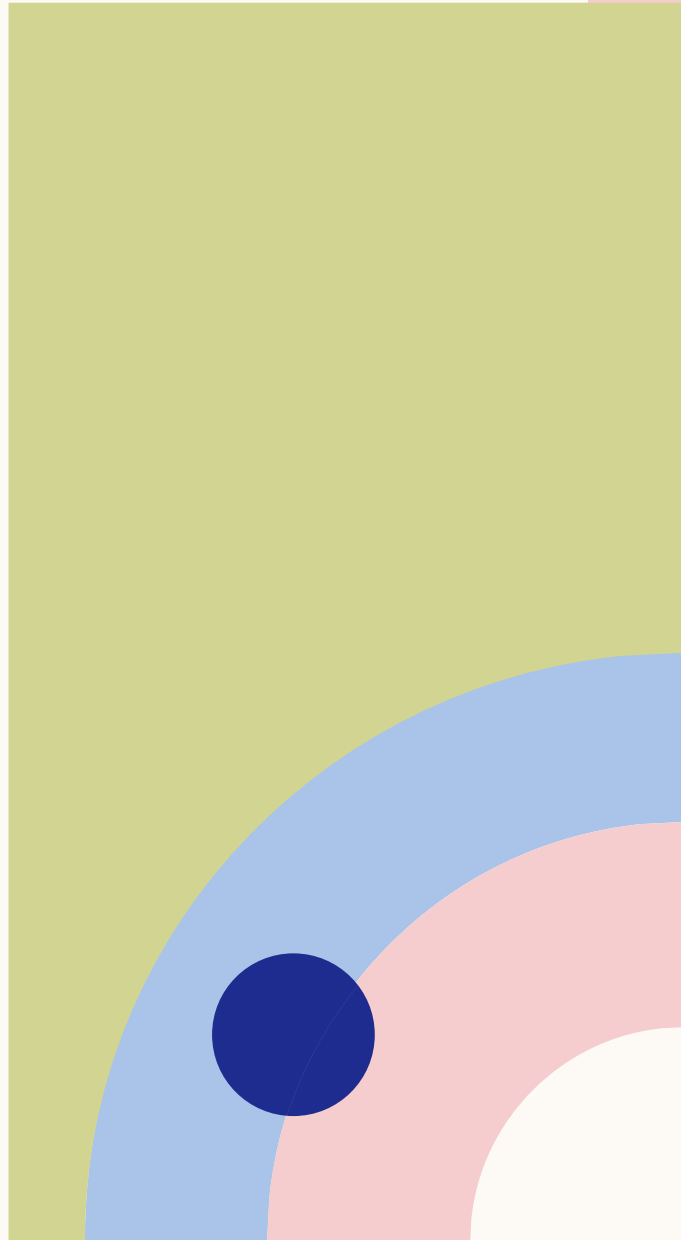
«Δυναμικό μέγεθος»

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
a[1:3]=[11, 12, 13, 14, 15, 16]
print(a)
```

```
[1, 11, 12, 13, 14, 15, 16, 4, 5, 6, 7, 8, 9]
```



Σάρωση στοιχείων λίστας

- Απευθείας:

```
a=[1,4,6,7,15]
for value in a:
  print(value)
```

ΣΗΜΑΝΤΙΚΟ!

Οι λίστες ανήκουν σε μια κατηγορία αντικειμένων που λέγονται **iterables** (*από το iteration: επανάληψη*) που σημαίνει ότι μπορούμε να τρέξουμε δομές **for** πάνω στα στοιχεία τους.

- Με δείκτη:

```
a=[1,4,6,7,15]
for i in range(len(a)):
  print(a[i])
```

Χρησιμοποιούμε τη συνάρτηση **len()** η οποία δέχεται ένα σύνθετο αντικείμενο και επιστρέφει το «μήκος» του (δηλαδή το πλήθος των στοιχείων του):

```
print(len(a))
5
```

Υπενθύμιση: Οι δύο χρήσεις της `in`

Ως λογική πράξη:

```
print(2 in [3, 5, 2, 1])
print("a" in ["b", "c", "d"])
```

```
True
False
```

Ως τμήμα μιας `for`:

```
for i in [3,4,6]:
  print(i)
```

```
3
4
6
```

Η εντολή del

Διαγραφή μεμονωμένου στοιχείου:

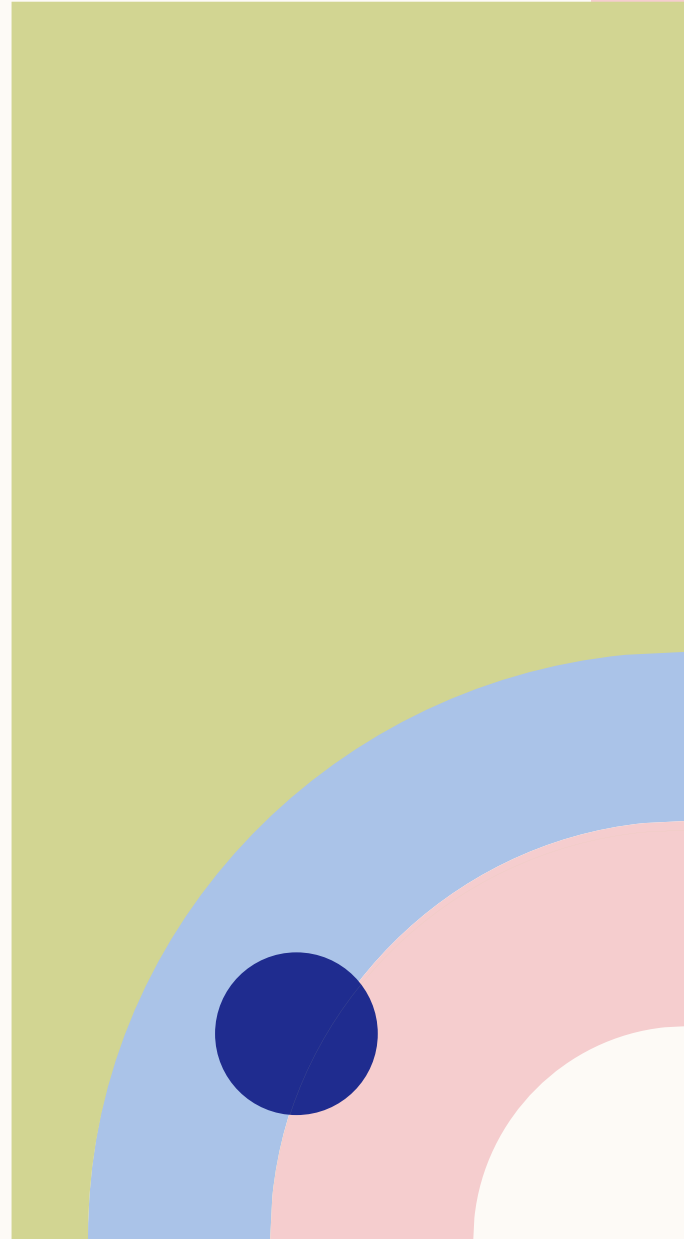
```
a = [2, 5, 6, 7, 3]
del(a[3])
print(a)
[2, 5, 6, 3]
```

Διαγραφή ολόκληρης της λίστας:

```
del(a)
print(a)
NameError: name 'a' is not defined
```

Παράδειγμα 4.1

Δίνεται μια λίστα **a** που περιέχει αριθμούς. Να γραφεί κώδικας σε Python που θα αντιστρέφει τα πρόσημα όσων τιμών χρειάζεται ώστε η λίστα να μην περιέχει αρνητικούς αριθμούς.



Παράδειγμα 4.1

Δίνεται μια λίστα **a** που περιέχει αριθμούς. Να γραφεί κώδικας σε Python που θα αντιστρέφει τα πρόσημα όσων τιμών χρειάζεται ώστε η λίστα να μην περιέχει αρνητικούς αριθμούς.

```
a = [2,5,-8,2,-7,8,-2]
for i in range(len(a)):
  if a[i]<0:
    a[i]=-a[i]
print(a)
```

```
a = [2,5,-8,2,-7,8,-2]
for i in range(len(a)):
  a[i]=abs(a[i])

print(a)
```

Έννοιες αυτής της διάλεξης (1/5)

- **Λίστα (List)**: δυναμικό αντικείμενο που περιέχει αριθμημένα στοιχεία κάθε τύπου
  - **Indexing & Slicing**: Forward & backward, zero-indexed
  - Η εντολή **`in`**: Είτε λογική πράξη (αναζήτηση) είτε επανάληψη/iteration σε μια **`for`**
  - **Iterable**: μπορούμε να πάρουμε τα στοιχεία του διαδοχικά, π.χ. με μία **`for`**

Υπενθύμιση: Συναρτήσεις

- **Συνάρτηση** στην Python είναι ένα τμήμα κώδικα που έχει όνομα, δέχεται εισόδους («παραμέτρους»), και επιστρέφει τιμές.
  - Έχει πάντα παρενθέσεις `()` στο τέλος του ονόματος
  - **`abs(a)`**
    - Δέχεται έναν αριθμό, υπολογίζει την απόλυτη τιμή του, και την επιστρέφει
  - **`max(a, b, c, d, e,…)`** ή **`max(a)`**
    - Δέχεται N αντικείμενα και υπολογίζει και επιστρέφει το μεγαλύτερο, ή ένα σύνθετο αντικείμενο και επιστρέφει το μεγαλύτερο στοιχείο του
  - **`input(a)`**
    - Δέχεται ένα string, το εμφανίζει στην οθόνη, διαβάζει ένα string από το χρήστη και το επιστρέφει
  - **`print(a, b, c, d, e,…)`**
    - Δέχεται N αντικείμενα, τα εμφανίζει στην οθόνη χωρισμένα με κενά, και επιστρέφει **`None`**
  - **`float(a)`**
    - Δέχεται ένα αντικείμενο, το μετατρέπει σε τύπο **`float`** και το επιστρέφει

Συναρτήσεις για λίστες

Πολλές συναρτήσεις της Python δέχονται λίστες ως είσοδο:

- **`print(X)`**: εμφανίζει τα στοιχεία της λίστας **`X`**, επιστρέφει **`None`**
- **`len(X)`**: επιστρέφει το πλήθος των στοιχείων της λίστας **`X`**
- **`sum(X)`**: επιστρέφει το άθροισμα των στοιχείων της λίστας **`X`**
- **`min(X)`**, **`max(X)`**: επιστρέφει το μεγαλύτερο ή το μικρότερο στοιχείο της λίστας **`X`**
- **`sorted(X)`**: επιστρέφει μια λίστα που περιέχει τα στοιχεία της λίστας **`X`** ταξινομημένα με αύξουσα σειρά
  - **Αριθμούς, strings (αλφαβητικά), λίστες**
    - Αρκεί να ανήκουν όλα στον ίδιο τύπο
  - **`sorted(X, reverse = True)`**: επιστρέφει μια λίστα που περιέχει τα στοιχεία της λίστας **`X`** ταξινομημένα με *φθίνουσα* σειρά

Άλλες συναρτήσεις της Python **δεν** δέχονται λίστες ως είσοδο:

- Π.χ. οι **`abs()`**, **`pow()`**, **`round()`** προκαλούν TypeError όταν δεχθούν λίστα ως είσοδο

Πρέπει να τα μάθουμε απέξω όλα αυτά;

- **Όχι.** Όταν προγραμματίζουμε είναι φυσικό να ζητάμε βοήθεια από το documentation της γλώσσας, τις σημειώσεις μας, το Web, κάποιο LLM
- Όμως είναι εύλογο ότι όταν κάποιο άτομο περάσει 30-40 ώρες προγραμματίζοντας (τόσο βγαίνουν τα εργαστήρια αυτού του εξαμήνου) κάποια πράγματα θα τα έχει μάθει από συνήθεια
  - Στις εξετάσεις, θα έχετε πρόσβαση στο documentation των συναρτήσεων που θα χρειαστείτε
    - Π.χ. δε χρειάζεται να μάθετε απέξω ότι το **`abs([-2,5])`** δεν επιστρέφει **`[2,5]`** αλλά **`TypeError`**
  - Θα πρέπει όμως να ξέρετε πώς λειτουργεί η Python
    - Π.χ. ότι η **`for`** συντάσσεται με **`in`** και ένα *iterable* (range, λίστα ή κάτι άλλο)
    - Ή πώς δουλεύει το indexing/slicing

Σημείωση για το όνομα `sum`

- Σε προηγούμενα παραδείγματα, χρησιμοποιήσαμε το όνομα **`sum`** ως <u>όνομα μεταβλητής</u> όπου συγκεντρώναμε το άθροισμα
  - Αυτή είναι μια βολική πρακτική, αλλά είναι *προβληματική*
- Αν δημουργήσουμε στη μνήμη μια μεταβλητή **`sum`**, τότε από εκεί και πέρα όταν αναφερόμαστε σε **`sum`** εννοούμε τη μεταβλητή μας
  - Χάνουμε την πρόσβαση στη συνάρτηση **`sum()`**
  - Είναι σα να φτιάξαμε μια μεταβλητή που την είπαμε **`print`**
    - Η Python το επιτρέπει, αλλά μάλλον δε θα μας βγει σε καλό

Συναρτήσεις και Μέθοδοι

- *Ξανά*: **Συνάρτηση** στην Python είναι ένα τμήμα κώδικα που έχει όνομα, δέχεται εισόδους («παραμέτρους»), και επιστρέφει τιμές.
- **Μέθοδος** στην Python είναι μια συνάρτηση η οποία είναι ενσωματωμένη σε ένα αντικείμενο

Αντικειμενοστραφής προγραμματισμός

- Ως τώρα συμβολίζαμε τις μεταβλητές ως "κουτάκια" στη μνήμη
  - Αυτή είναι μια επαρκής αναπαράσταση για κλασικές γλώσσες όπως η C, αλλά *δεν είναι αλήθεια* στις αντικειμενοστραφείς γλώσσες όπως η Python
- Στην Python, ακόμα και η πιο απλή integer είναι στην πραγματικότητα ένα σύνθετο **αντικείμενο**
  - Ένα αντικείμενο είναι ένα στιγμιότυπο μιας κλάσης
    - Μια **κλάση** είναι το «καλούπι» από το οποίο φτιάχνω αντικείμενα
    - Π.χ. υπάρχει η κλάση **`float`** και όταν εγώ γράφω **`a=5.2`** δημιουργώ ένα αντικείμενο **`a`** που είναι στιγμιότυπό της **`float`**

Μια **κλάση** αποτελεί ένα συνδυασμό *κώδικα* και *δεδομένων*

Μέθοδοι αντικειμένων

- Όταν γράφω **`a=5.5`** δημιουργώ ένα στιγμιότυπο της κλάσης **`float`** με όνομα **`a`**, που αποτελεί συνδυασμό *κώδικα* και *δεδομένων*.
- Τα **δεδομένα** είναι προφανή: είναι μόνο ο αριθμός **`5.5`**
- Ο **κώδικας** αποτελείται από συναρτήσεις ενσωματωμένες στην **`a`** (δηλαδή μεθόδους της **`float`**). Π.χ.:
  - **`a.is_integer()`**: Χωρίς παραμέτρους. Επιστρέφει **`True`** αν το **`a`** είναι ακέραιο (π.χ. **`5.0`**) και **`False`** αλλιώς
  - **`a.as_integer_ratio()`**: Χωρίς παραμέτρους. Επιστρέφει ένα **`tuple`** που εκφράζει τον **`a`** ως πηλίκο δύο ακεραίων (π.χ. αν το **`a`** είναι **`5.5`** θα επιστρέψει **`11`** και **`2`**)
- Προσέξτε τη σύνταξη: **`όνομα_αντικειμένου.όνομα_μεθόδου()`**
- Προσέξτε τη λειτουργία: η μέθοδος εκτελείται **πάνω στα δεδομένα του αντικειμένου**

Σχεδόν όλες οι αντικειμενοστρα-φείς γλώσσες χρησιμοποιούν την τελεία.

Μέθοδοι λιστών

- **`list.index(x)`**: δέχεται μια τιμή και επιστρέφει (**`int`**) τη θέση της τιμής **`x`** μέσα στη λίστα
  - Αν η λίστα δεν περιέχει τον αριθμό, προκαλείται **`ValueError`**
  - Η λύση είναι, πριν αναζητήσουμε **πού** είναι ο αριθμός μέσα σε μια λίστα, να ελέγξουμε **αν** υπάρχει ο αριθμός μέσα στη λίστα:

```
if x in a:
    print(a.index(x))
```

- **`list.count(x)`**: δέχεται μια τιμή **`x`** και επιστρέφει (**`int`**) το πλήθος των εμφανίσεων της **`x`** μέσα στη λίστα

Η x μπορεί να είναι οποιοσδήποτε τύπος αντικειμένου!

Μέθοδοι in-place («επιτόπου»)

- Οι μέθοδοι λιστών που είδαμε, παράγουν νέα αντικείμενα με βάση τα περιεχόμενα της λίστας
- Υπάρχουν άλλες μέθοδοί, που ο κώδικάς τους τροποποιεί απευθείας τα στοιχεία του αντικειμένου
  - Λέμε ότι λειτουργούν **in-place**
  - Οι *περισσότερες* in-place μέθοδοι επιστρέφουν **`None`**

```
a = [5, 23.5, 18, float("-inf"), 1917]
b = a.sort()
print(b)
print(a)
None
[-inf, 5, 18, 23.5, 1917]
```

Η **`List.sort()`** επέστρεψε **`None`** στο **`b`**

Η ίδια η **`a`** όμως μεταβλήθηκε

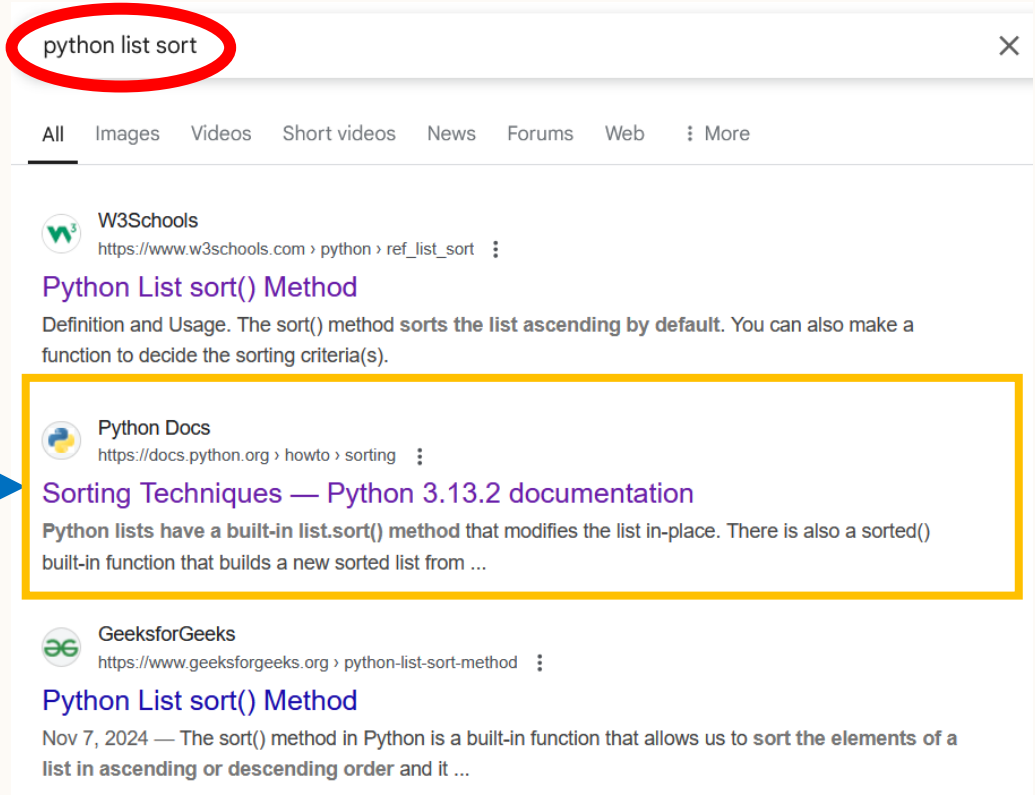
Μέθοδοι λιστών (in-place)

- **`list.sort()`**: Ταξινομεί τη λίστα με αύξουσα σειρά
  - **Αριθμούς, strings (αλφαβητικά), λίστες**
  - **`List.sort(reverse=True)`**: Ταξινομεί τη λίστα με φθίνουσα σειρά
    - (το "**`reverse`**" είναι μια *προαιρετική παράμετρος* της **`sort()`**)
- **`list.reverse()`**: Αντιστρέφει τη σειρά των στοιχείων
  - Απλή συνωνυμία με την παραπάνω παράμετρο της **`sort()`**!
- **`list.append()`**: Προσθέτει το στοιχείο **`x`** στο τέλος της λίστας
- **`list.insert(i,x)`**: Τοποθετεί το στοιχείο **`x`** στη θέση **`i`** και μετακινεί όλα τα επόμενα στοιχεία προς τα δεξιά
- **`list.remove(x)`**: Διαγράφει το αριστερότερο στοιχείο με τιμή **`x`** από τη λίστα
  - Αν η λίστα δεν περιέχει καμία τιμή **`x`**, προκαλείται **`ValueError`**

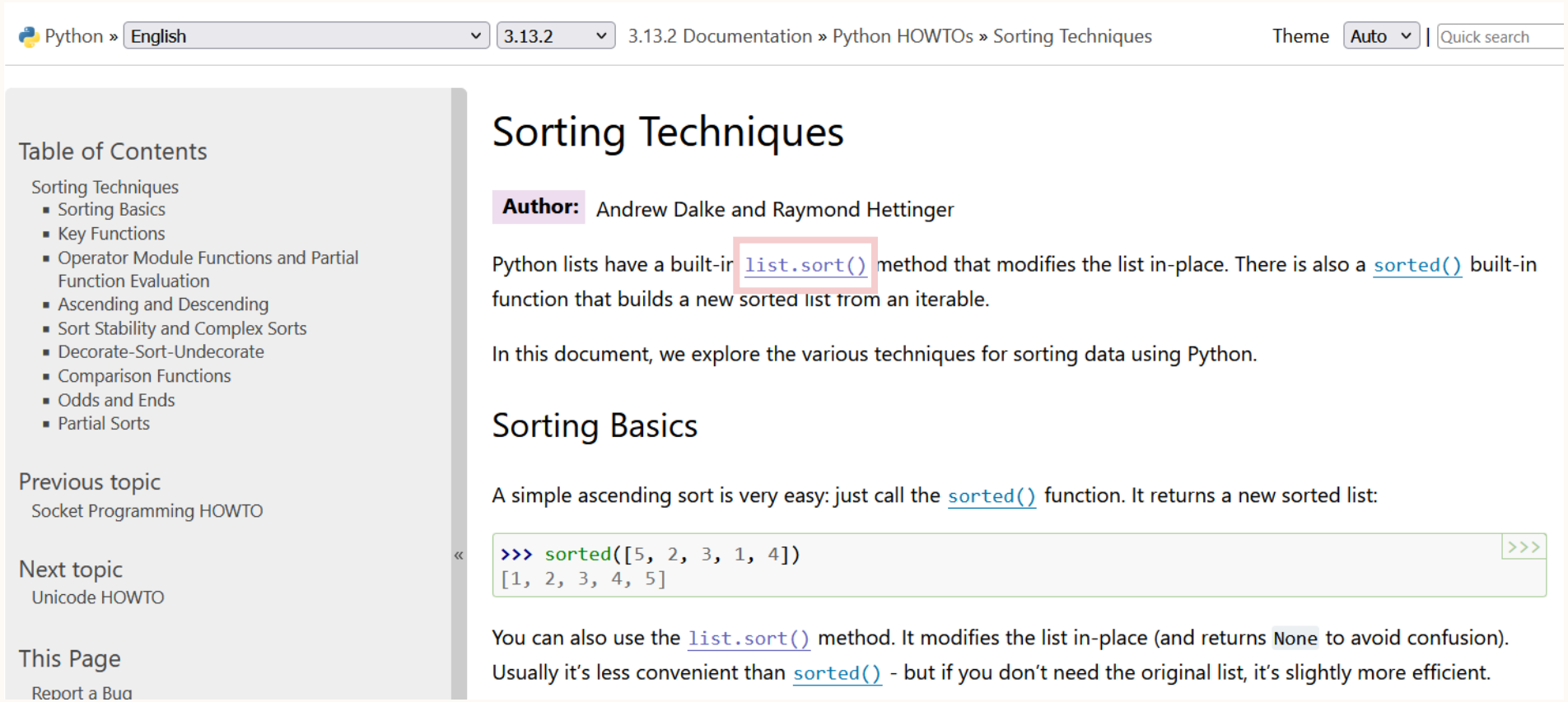
Παρένθεση: η χρήση του Documentation

- Ας πούμε ότι θέλω να ψάξω πώς δουλεύει η `list.sort()`

Καλύτερα να ψάξω στο official documentation, αν και πολλές φορές οι εκπαιδευτικές σελίδες όπως το W3Schools είναι πολύ βοηθητικές



Documentation



Python » English 3.13.2 3.13.2 Documentation » Python HOWTOs » Sorting Techniques Theme Auto | Quick search

Table of Contents

Sorting Techniques
- Sorting Basics
- Key Functions
- Operator Module Functions and Partial Function Evaluation
- Ascending and Descending
- Sort Stability and Complex Sorts
- Decorate-Sort-Undecorate
- Comparison Functions
- Odds and Ends
- Partial Sorts

Previous topic
Socket Programming HOWTO

Next topic
Unicode HOWTO

This Page
Report a Bug

Sorting Techniques

Author: Andrew Dalke and Raymond Hettinger

Python lists have a built-in `list.sort()` method that modifies the list in-place. There is also a `sorted()` built-in function that builds a new sorted list from an iterable.

In this document, we explore the various techniques for sorting data using Python.

Sorting Basics

A simple ascending sort is very easy: just call the `sorted()` function. It returns a new sorted list:

```
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]
```

You can also use the `list.sort()` method. It modifies the list in-place (and returns `None` to avoid confusion). Usually it's less convenient than `sorted()` - but if you don't need the original list, it's slightly more efficient.

Documentation

sort(*, *key=None, reverse=False*)

This method sorts the list in place, using only `<` comparisons between items. Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

`sort()` accepts two arguments that can only be passed by keyword (keyword-only arguments):

key specifies a function of one argument that is used to extract a comparison key from each list element (for example, `key=str.lower`). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of `None` means that list items are sorted directly without calculating a separate key value.

The `functools.cmp_to_key()` utility is available to convert a 2.x style *cmp* function to a *key* function.

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Παράδειγμα 4.2

Να γραφεί πρόγραμμα που θα ξεκινάει με μια λίστα αριθμών (π.χ. `[3, 5, 14, 3, 6, 7, 23]`) και θα δημιουργεί μια νέα λίστα η οποία θα περιλαμβάνει μόνο τα άρτια (ζυγά) στοιχεία της.

```
a = [3, 5, 14, 3, 6, 7, 23]
```

Παράδειγμα 4.2

Να γραφεί πρόγραμμα που θα ξεκινάει με μια λίστα αριθμών (π.χ. `[3, 5, 14, 3, 6, 7, 23]`) και θα δημιουργεί μια νέα λίστα η οποία θα περιλαμβάνει μόνο τα άρτια στοιχεία της.

```
a = [3, 5, 14, 3, 6, 7, 23]
b = []

for i in range(len(a)):
  if a[i] % 2 == 0:
    b.append(a[i])

print(b)
```

Παράδειγμα 4.3

Να γραφεί πρόγραμμα που θα δέχεται 10 αριθμούς από το χρήστη. Στη συνέχεια, θα εμφανίζει μόνο όσους από αυτούς τους αριθμούς ήταν πάνω από το μέσο όρο.

Παράδειγμα 4.3

Να γραφεί πρόγραμμα που θα δέχεται 10 αριθμούς από το χρήστη. Στη συνέχεια, θα εμφανίζει μόνο όσους από αυτούς τους αριθμούς ήταν πάνω από το μέσο όρο.

```
a = []
for i in range(10):
  a.append(float(input("Give a number:")))
mo = sum(a)/10

for num in a:
  if num>mo:
    print(num)
```

Παράδειγμα 4.4

Να γραφεί πρόγραμμα που θα δέχεται 10 αριθμούς από το χρήστη. Στη συνέχεια, θα εμφανίζει *μια λίστα* που θα περιέχει μόνο όσους από αυτούς τους αριθμούς ήταν πάνω από το μέσο όρο.

Παράδειγμα 4.4

Να γραφεί πρόγραμμα που θα δέχεται 10 αριθμούς από το χρήστη. Στη συνέχεια, θα εμφανίζει *μια λίστα* που θα περιέχει μόνο όσους από αυτούς τους αριθμούς ήταν πάνω από το μέσο όρο.

```
a = []
for i in range(10):
  a.append(float(input("Give a number:")))
mo = sum(a)/10

b=[]
for num in a:
  if num>mo:
    b.append(num)
print(b)
```

Έννοιες αυτής της διάλεξης (2/5)

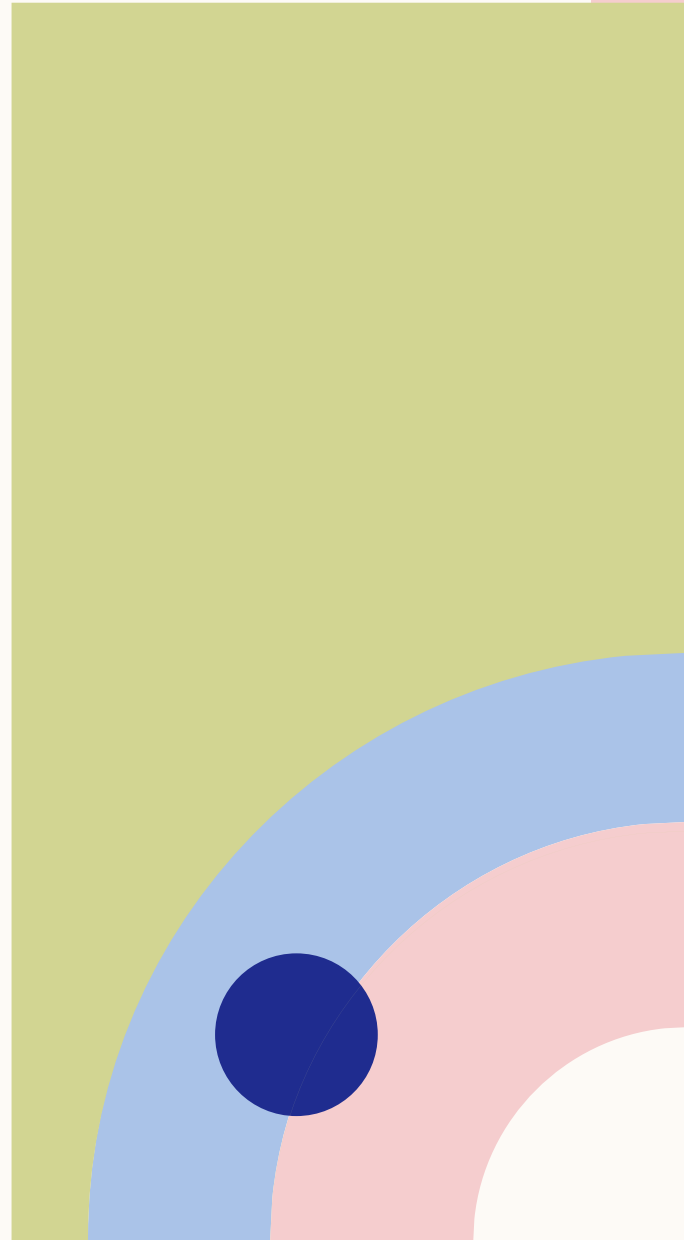
- **Λίστα (List)**: δυναμικό αντικείμενο που περιέχει αριθμημένα στοιχεία κάθε τύπου
  - **Indexing & Slicing**: Forward & backward, zero-indexed
  - Η εντολή **`in`**: Είτε λογική πράξη (αναζήτηση) είτε επανάληψη/iteration σε μια **`for`**
  - **Iterable**: μπορούμε να πάρουμε τα στοιχεία του διαδοχικά, π.χ. με μία **`for`**
- **Συνάρτηση (Function)**: κώδικας με όνομα, εισόδους (*παραμέτρους*), εξόδους
- **Μέθοδος**: συνάρτηση ενσωματωμένη σε ένα αντικείμενο
- **Αντικείμενο**: στιγμιότυπο μιας κλάσης/τύπου
  - Μέθοδος που λειτουργεί **in-place**: τροποποιεί τις τιμές του αντικειμένου χωρίς απαραίτητα να έχει εξόδους
    - Π.χ. **`list.sort()`**, **`list.append()`**

Λίστα με λίστες

- Είπαμε ότι μια λίστα μπορεί να περιέχει κάθε τύπο αντικειμένου
  - Άρα μπορεί να περιέχει και λίστες
  - Π.χ. `b=[2, 4, [24,25,26]]`
- Είπαμε επίσης ότι με το indexing μπορούμε να «πάρουμε» ένα στοιχείο μιας λίστας
  - Άρα το `b[2]` είναι η λίστα `[24,25,26]`
  - …άρα το `b[2][0]` είναι ο integer `24`
- Αφού μπορώ να έχω λίστες μέσα σε λίστες (μέσα σε λίστες, μέσα σε λίστες…), μπορώ να απευθύνομαι σε αυτές με διαδοχικές αγκύλες που θα κάνουν indexing

Παράδειγμα 4.5

Να γραφεί κώδικας που θα υπολογίζει όλη την προπαίδεια από το 1 μέχρι το 9, και θα αποθηκεύει τις τιμές σε μια λίστα από εννιά εσωτερικές λίστες. Η πρώτη θα περιέχει την προπαίδεια του 1, η δεύτερη του 2 κλπ.



Παράδειγμα 4.5

Να γραφεί κώδικας που θα υπολογίζει όλη την προπαίδεια από το 1 μέχρι το 9, και θα αποθηκεύει τις τιμές σε μια λίστα από εννιά εσωτερικές λίστες. Η πρώτη θα περιέχει την προπαίδεια του 1, η δεύτερη του 2 κλπ.

```
prop = []
for i in range(1,10):
  prop.append([])
  for j in range(1,10):
    prop[i-1].append(i*j)

print(prop)

for p in prop:
  print(p)
```

List Comprehension

«Σύνθεση λίστας»

```
newlist = [expression for item in iterable if condition]
```

Π.χ.:

```
original_list = [1, 15, 5, -17, 32, 6]
a = [x+1 for x in original_list if x>10]
print(a)
```

```
[16, 33]
```

Τι θα επιστρέψει;

```
[x**2 for x in [2, 3, -1, 6, 7] if x%2==0]
```

```
[a[0] for a in [[2,4], [4,6,8], ["α"], [True,2,1]] if len(a)>2]
```

```
L=[4, 6, 12, 4, 5, 16]
[a-1 for a in L[::2] if a<10]
```

Το condition είναι προαιρετικό:

```
[y**2 for y in range(10)]
```

```
[abs(a) for a in [-3, 4.2, "Hello", -0.3]]
```

Παράδειγμα 4.6

Να γραφεί κώδικας που θα ξεκινάει με δυο λίστες αριθμών, και **με μία μόνο εντολή** θα εμφανίζει μια νέα λίστα που θα περιέχει όλα τα στοιχεία και των δυο λιστών που διαιρούνται ακριβώς με το 5, *διαιρεμένα με το 5*.

```
a = [11,25,133]
b = [15, 67, 51, 165]
```

Παράδειγμα 4.6

Να γραφεί κώδικας που θα ξεκινάει με δυο λίστες αριθμών, και **με μία μόνο εντολή** θα εμφανίζει μια νέα λίστα που θα περιέχει όλα τα στοιχεία και των δυο λιστών που διαιρούνται ακριβώς με το 5, *διαιρεμένα με το 5*.

```
a = [11,25,133]
b = [15, 67, 51, 165]
print([n/5 for n in a if n%5==0] + [n/5 for n in b if n%5==0])

# εναλλακτικά
print([n/5 for n in a+b if n%5==0])
```

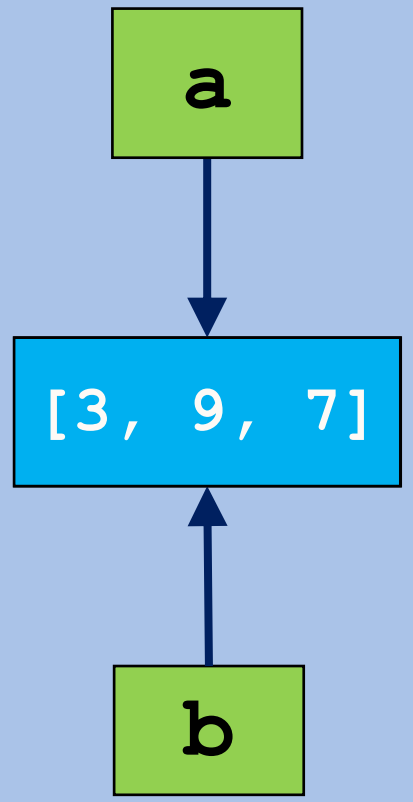
Έννοιες αυτής της διάλεξης (3/5)

- **Λίστα (List)**: δυναμικό αντικείμενο που περιέχει αριθμημένα στοιχεία κάθε τύπου
  - **Indexing & Slicing**: Forward & backward, zero-indexed
  - Η εντολή **`in`**: Είτε λογική πράξη (αναζήτηση) είτε επανάληψη/iteration σε μια **`for`**
  - **Iterable**: μπορούμε να πάρουμε τα στοιχεία του διαδοχικά, π.χ. με μία **`for`**
- **Συνάρτηση (Function)**: κώδικας με όνομα, εισόδους (*παραμέτρους*), εξόδους
- **Μέθοδος**: συνάρτηση ενσωματωμένη σε ένα αντικείμενο
- **Αντικείμενο**: στιγμιότυπο μιας κλάσης/τύπου
  - Μέθοδος που λειτουργεί **in-place**: τροποποιεί τις τιμές του αντικειμένου χωρίς απαραίτητα να έχει εξόδους
    - Π.χ. **`list.sort()`**, **`list.append()`**
- **Λίστα με λίστες:** αναφερόμαστε στα στοιχεία της με **`A[i][j][k]`**…
- **Σύνθεση** (**comprehension**): Ειδική σύνταξη που παράγει **λίστες** από **iterables**

Ανάθεση με αναφορά (Assignment by Reference)

VS

Ανάθεση με τιμή (Assignment by Value)



Τι θα εμφανίσει;

```
my_list = [2, 4, 5, 6]
new_list = my_list
print(new_list)
[2, 4, 5, 6]
new_list[3] = 11
print(new_list)
[2, 4, 5, 11]
print(my_list)
[2, 4, 5, 11]
```

What???

Γιατί άλλαξε η **my_list** όταν τροποποιήσαμε τη **new_list**;

Ανάθεση By Value

- Η παραδοσιακή απεικόνιση μιας μεταβλητής είναι ένα κουτάκι με όνομα και τιμή
- Όταν αναθέτω μια τιμή σε μια μεταβλητή, αλλάζω το περιεχόμενο του κουτιού
- Όταν αναθέτω μια μεταβλητή σε μια άλλη, αντιγράφω στο κουτί της δεύτερης την τιμή της πρώτης


```
a=10
```

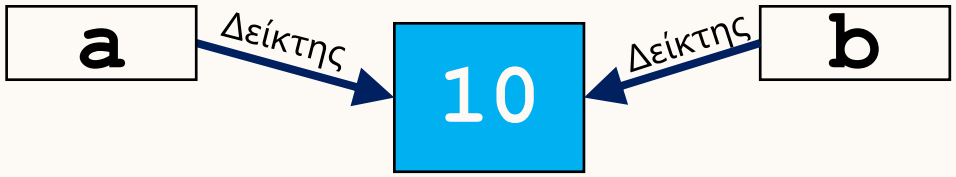
```
b=a
```

Αυτός ο τρόπος ανάθεσης, που λέγεται *ανάθεση με βάση την τιμή* (**Assignment By Value**) **ΔΕΝ** είναι ο τρόπος που λειτουργεί η Python!

Ανάθεση By Reference

- Στην Python, το όνομα ενός αντικειμένου είναι ένας **δείκτης** («αναφορά») που δείχνει σε μια θέση μνήμης.
- Όταν αναθέτουμε μια τιμή σε ένα αντικείμενο, το όνομα δείχνει στη θέση μνήμης που περιέχει την τιμή
  - Το αποτέλεσμα μοιάζει με το «κουτάκι μνήμης»
- Όταν όμως αναθέτω μια μεταβλητή σε μια άλλη, το δεύτερο όνομα **δείχνει στην ίδια θέση μνήμης**

```
a=10
b=a
```



Ώπα μια στιγμή!

- Αφού ξέρουμε ότι στους ακέραιους (και στις float, και στις bool) η Python συμπεριφέρεται «κανονικά».

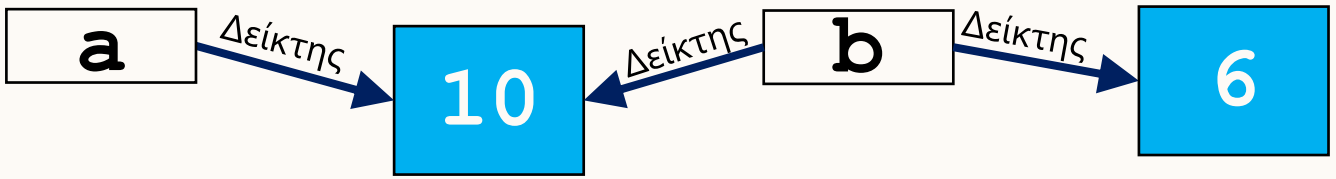
```
a=10
b=a
b=6
print(a)
10
```

Το a δεν άλλαξε μαζί με το b.

Τι συμβαίνει;

```
a=10
b=a
b=6
```

Όταν **αναθέτω μια νέα τιμή** στη b, η παλιά ανάθεση **αντικαθίσταται εξ' ολοκλήρου**.
Έτσι, η τιμή στην οποία δείχνει η **a δεν επηρεάζεται**.



Ανάθεση ολόκληρης λίστας

- Στην πραγματικότητα, αν αναθέσω μια ολόκληρη λίστα στη b, θα συμβεί το ίδιο.

```
a=[3, 6, 7]
b=a
b=[4, 9]
print(a)
```

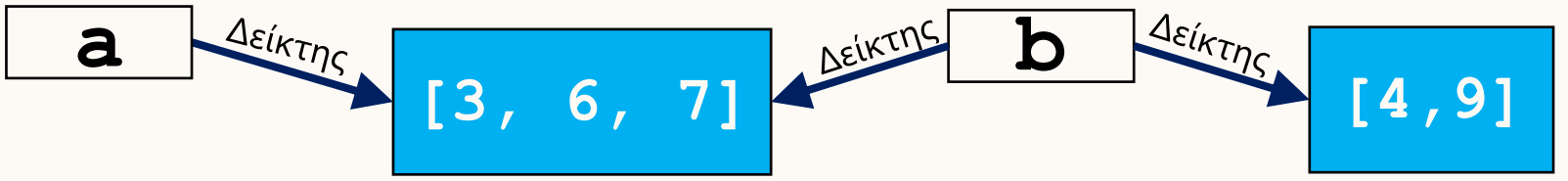
```
[3, 6, 7]
```

Και πάλι το a δεν άλλαξε μαζί με το b.

Τι συμβαίνει αυτή τη φορά;

```
a=[3, 6, 7]
b=a
b=[4, 9]
```

Επειδή **αναθέτω** μια **ολόκληρη** λίστα στη **b**, η παλιά ανάθεση **αντικαθίσταται και πάλι εξ' ολοκλήρου**.
Έτσι, η τιμή στην οποία δείχνει η **a** πάλι **δεν επηρεάζεται**.



Το πρόβλημα: τροποποίηση αντικειμένου

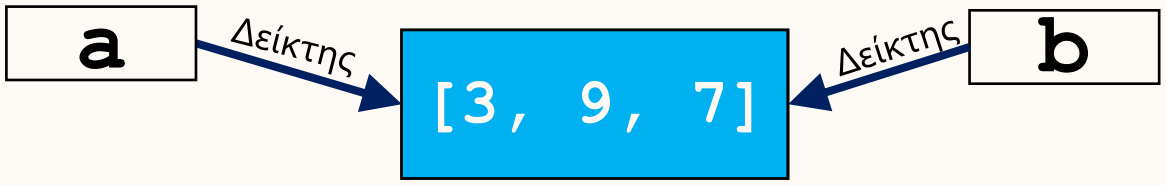
- Πότε λοιπόν εμφανίζεται το φαινόμενο που είδαμε αρχικά;

```
a=[3, 6, 7]
b=a
b[1] = 9
print(a)
[3, 9, 7]
```

Τώρα το a άλλαξε μαζί με το b.
Ας δούμε τι συνέβη.

```
a=[3, 6, 7]
b=a
b[1] = 9
```

Επειδή **τροποποιώ** τη λίστα στην οποία δείχνει η **b**, αντί να αναθέσω μια ολόκληρη νέα λίστα, **η ανάθεση ΔΙΑΤΗΡΕΙΤΑΙ**.
Έτσι, και τα δυο ονόματα **συνεχίζουν να δείχνουν στην ίδια, τροποποιημένη λίστα**.

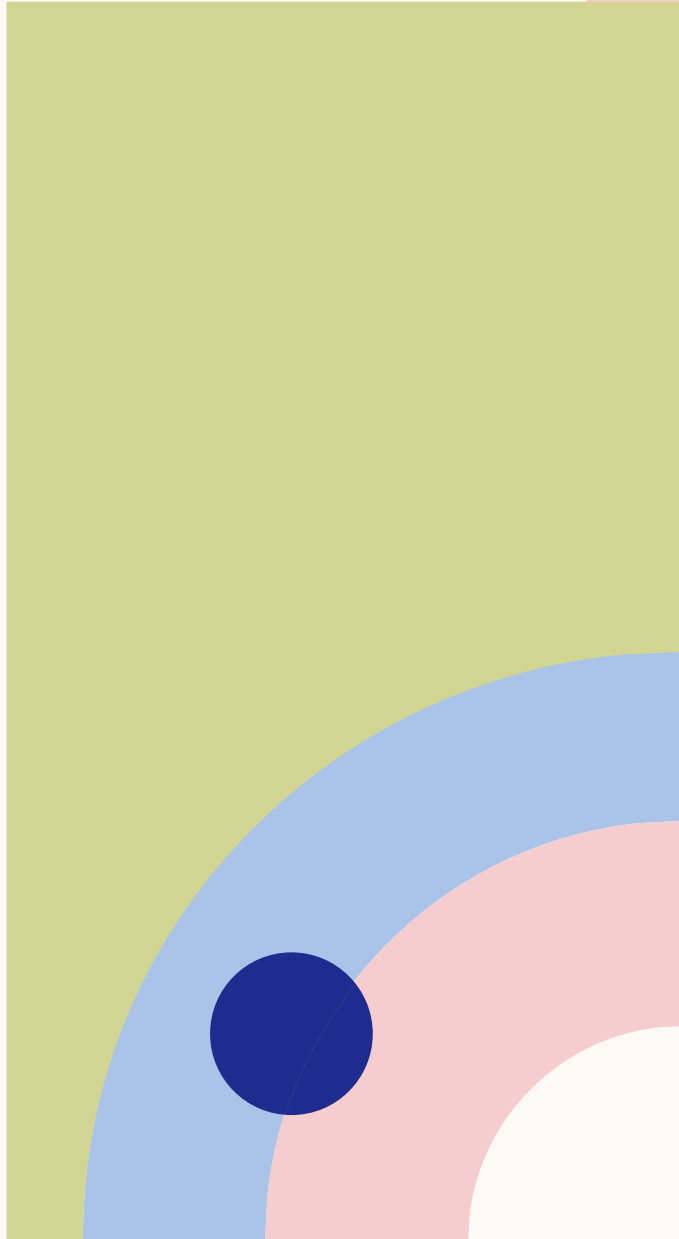


Άρα;

- Άρα, το φαινόμενο εμφανίζεται μόνο όταν τροποποιούμε ένα τμήμα ενός αντικειμένου και όχι όταν του αναθέτουμε εξ' ολοκλήρου νέα τιμή.
  - Στους int, float, complex, bool, αυτό δε συμβαίνει ποτέ γιατί *δεν έχουν τμήματα*.
- Στις λίστες μπορεί να εμφανιστεί όταν αναθέτουμε τιμή σε μία θέση τους ή σε ένα slice τους.
- **Προσοχή**: δεν είναι *πρόβλημα*, είναι μηχανισμός λειτουργίας της Python
  - Μπορεί να είναι πολύ χρήσιμος
  - Μπορεί να *γίνει* πρόβλημα, αν δεν προσέχουμε…

Mutables και Immutables

- Υπάρχουν αντικείμενα στην Python που μας επιτρέπουν να τροποποιούμε μόνο ένα τμήμα τους. Τα λέμε **Mutables**
  - Είδαμε τις **λίστες** (lists)
  - Επίσης **λεξικά** (dictionaries), **σύνολα** (sets), αλλά και άλλες σημαντικές κλάσεις
  - Μόνο τα mutables έχουν συναρτήσεις in-place
- Τα αντικείμενα που δεν επιτρέπουν τροποποιήσεις λέγονται **Immutables**
  - **`float`**, **`int`**, **`complex`**, **`bool`**
  - Επίσης το **`string`** που θα δούμε σε επόμενο κεφάλαιο, και **πλειάδα** (**`tuple`**) που θα δούμε εδώ στη συνέχεια



Mutables και Immutables

Ο κανόνας είναι απλός:

- **Αν τροποποιήσω ένα τμήμα** ενός **mutable** αντικειμένου, τροποποιείται αυτόματα η τιμή που θα επιστρέψουν **όλες** οι μεταβλητές που δείχνουν σε αυτό
- **Αν αναθέσω εξ' ολοκλήρου μια νέα τιμή** σε ένα αντικείμενο (mutable ή immutable), τότε δημιουργείται μια νέα αναφορά που δεν επηρεάζει **καμία** άλλη μεταβλητή

Έννοιες αυτής της διάλεξης (τέλος!)

- **Λίστα (List)**: δυναμικό αντικείμενο που περιέχει αριθμημένα στοιχεία κάθε τύπου
  - **Indexing & Slicing**: Forward & backward, zero-indexed
  - Η εντολή **`in`**: Είτε λογική πράξη (αναζήτηση) είτε επανάληψη/iteration σε μια **`for`**
  - **Iterable**: μπορούμε να πάρουμε τα στοιχεία του διαδοχικά, π.χ. με μία **`for`**
  - **Mutable**: μπορούμε να τροποποιούμε τα στοιχεία του χωρίς να αλλάζουμε τις αναφορές σε αυτό
- **Συνάρτηση (Function)**: κώδικας με όνομα, εισόδους (*παραμέτρους*), εξόδους
- **Μέθοδος**: συνάρτηση ενσωματωμένη σε ένα αντικείμενο
- **Αντικείμενο**: στιγμιότυπο μιας κλάσης/τύπου
  - Μέθοδος που λειτουργεί **in-place**: τροποποιεί τις τιμές του αντικειμένου χωρίς απαραίτητα να έχει εξόδους
    - Π.χ. **`List.sort()`**, **`List.append()`**
- **Λίστα με λίστες:** αναφερόμαστε στα στοιχεία της με **`A[i][j][k]`**…
- **Σύνθεση** (**comprehension**): Ειδική σύνταξη που παράγει **λίστες** από **iterables**
- Στην Python όλες οι αναθέσεις είναι **by reference.** Αν τροποποιήσουμε μέρος ενός immutable, τροποποιούνται όλες οι μεταβλητές που δείχνουν σε αυτό.