Προγραμματισμός με τη γλώσσα python

Alexandros Kanterakis (mailto:kantale@ics.forth.gr) kantale@ics.forth.gr"

Διάλεξη 4η, 5 Νοεμβρίου 2019

Οι μέθοδοι `split` και `join`

Αν θέλουμε να "σπάσουμε" ένα string σε μία λίστα από πολλά string τότε μπορούμε να χρησιμοποιήσουμε τη `split`

In [14]:
```
"a+b+c".split('+')
```

Out[14]:
```
['a', 'b', 'c']
```

In [15]:
```
"hello world".split(' ')
```

Out[15]:
```
['hello', 'world']
```

In [16]:
```
"I like to move it move it".split('move')
```

Out[16]:
```
['I like to ', ' it ', ' it']
```

In [17]:
```
a = '''
ἄνδρα μοι ἔννεπε, μοῦσα, πολύτροπον, ὃς μάλα πολλὰ
πλάγχθη, ἐπεὶ Τροίης ἱερὸν πτολίεθρον ἔπερσεν·
πολλῶν δ' ἀνθρώπων ἴδεν ἄστεα καὶ νόον ἔγνω,
πολλὰ δ' ὅ γ' ἐν πόντῳ πάθεν ἄλγεα ὃν κατὰ θυμόν,
ἀρνύμενος ἥν τε ψυχὴν καὶ νόστον ἑταίρων.
ἀλλ' οὐδ' ὣς ἑτάρους ἐρρύσατο, ἱέμενός περ·
αὐτῶν γὰρ σφετέρῃσιν ἀτασθαλίῃσιν ὄλοντο,
νήπιοι, οἳ κατὰ βοῦς Ὑπερίονος Ἠελίοιο
ἤσθιον· αὐτὰρ ὁ τοῖσιν ἀφείλετο νόστιμον ἦμαρ.
'''
a.split('\n')
```

Out[17]:
```
['',
 'ἄνδρα μοι ἔννεπε, μοῦσα, πολύτροπον, ὃς μάλα πολλὰ',
 'πλάγχθη, ἐπεὶ Τροίης ἱερὸν πτολίεθρον ἔπερσεν·',
 'πολλῶν δ' ἀνθρώπων ἴδεν ἄστεα καὶ νόον ἔγνω,',
 'πολλὰ δ' ὅ γ' ἐν πόντῳ πάθεν ἄλγεα ὃν κατὰ θυμόν,',
 'ἀρνύμενος ἥν τε ψυχὴν καὶ νόστον ἑταίρων.',
 'ἀλλ' οὐδ' ὣς ἑτάρους ἐρρύσατο, ἱέμενός περ·',
 'αὐτῶν γὰρ σφετέρῃσιν ἀτασθαλίῃσιν ὄλοντο,',
 'νήπιοι, οἳ κατὰ βοῦς Ὑπερίονος Ἠελίοιο',
 'ἤσθιον· αὐτὰρ ὁ τοῖσιν ἀφείλετο νόστιμον ἦμαρ.',
 '']
```

Αν η `split` δεν έχει κάποιο όρισμα, τότε αφαιρεί όλα τα κενά (και τα tabs και τα enters) μεταξύ των λέξεων σε ένα string:

In [19]:
```
"hello                world".split()
```

Out[19]:
```
['hello', 'world']
```

Η μέθοδος `join` κάνει το αντίθετο. Παίρνει μία λίστα από strings και τα εννώνει σε ένα string:

```
In [20]: '+'.join(['a','b','c'])
Out[20]: 'a+b+c'

In [21]: ' '.join(['hello', 'world'])
Out[21]: 'hello world'

In [22]: print ('\n'.join(['line 1', 'line 2']))
line 1
line 2
```

Η μέθοδος `strip`

Η μέθοδος `strip` αφαιρεί όλα τα κενά και τα "enter" από την αρχή και από το τέλος ενος string

```
In [23]: " a".strip()
Out[23]: 'a'

In [24]: "a ".strip()
Out[24]: 'a'

In [25]: " a ".strip()
Out[25]: 'a'

In [26]: "                    a                                    ".strip()
Out[26]: 'a'

In [27]: '''

                    a

         '''.strip()
Out[27]: 'a'
```

Η μέθοδος `enumerate`

Η μέθοδος `enumerate` παίρνει μία λίστα και δημιουργεί μία άλλη λίστα η οποία περιέχει και τις θέσεις (indexes) και τα στοιχεία της πρώτης λίστας:

```
In [28]: a = ["python", "mitsos", "Crete"]
         print (list(enumerate(a)))
[(0, 'python'), (1, 'mitsos'), (2, 'Crete')]
```

Dictionaries

Μέχρι στιγμής έχουμε μάθει τους παρακάτω τύπους μεταβλητών:

In [51]:
```
a=0 # ακέραιοι
a=True # λογικοί
a="324234" # αλφαριθμιτικά
a=5.6 # δεκαδικοί
a=[2,4,4] # listes
```

Τα dictionaries είναι ένα νέος τύπος μεταβλητής. Τα dictionaries έχουν δεδομένα με τη μορφή κλειδί --> τιμή. Κάθε κλειδί (key) είναι μοναδικό. Για παράδειγμα:

In [52]:
```
a = {"mitsos": 50, "anna": 40}
```

In [53]:
```
print(a)
```

```
{'mitsos': 50, 'anna': 40}
```

In [54]:
```
a['mitsos']
```

Out[54]: 50

In [55]:
```
a['anna']
```

Out[55]: 40

Η `keys` επιστρέφει μία λίστα με όλα τα κλειδιά του dictionary

In [57]:
```
a.keys()
```

Out[57]: dict_keys(['mitsos', 'anna'])

Η `values` επιστρέφει μία λίστα με όλες τις τιμές του dictionary

In [58]:
```
a.values()
```

Out[58]: dict_values([50, 40])

Μπορούμε να προσθέσουμε ένα νέο ζευγάρι κλειδί,τιμή με τον εξής τρόπο:

In [59]:
```
a["kitsos"] = 100
```

In [60]:
```
print (a)
```

```
{'mitsos': 50, 'anna': 40, 'kitsos': 100}
```

Το κλειδί μπορεί να είναι αριθμός, string και boolean και tuple. Ενώ το value μπορεί να είναι οτιδίποτε.

In [62]:
```
a[123] = 0.1
a[3.14] = "hello"
a[False] = [1,2,3]
a[(4,7)] = 4
```

In [63]:
```
print (a)
```

```
{'mitsos': 50, 'anna': 40, 'kitsos': 100, 123: 0.1, 3.14: 'hello', False: [1,
2, 3], (4, 7): 4}
```

In [64]:
```
# Προσοχή! False == 0 !
a[0]
```

Out[64]:
```
[1, 2, 3]
```

Το κλειδί ΔΕΝ μπορεί να είναι λίστα:

In [65]:
```
a[[1,2,3]] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-65-6cebb9942dfe> in <module>()
----> 1 a[[1,2,3]] = 0

TypeError: unhashable type: 'list'
```

Στη python μπορούμε να έχουμε dictionaries μέσα σε lists και lists μέσα σε dictionaries χωρίς κανένα περιορισμό

In [66]:
```
d = {"a": {2:"a"}, 3: ["hello", False, []], 3.1: True}
print (d)
```

```
{'a': {2: 'a'}, 3: ['hello', False, []], 3.1: True}
```

Μπορούμε να συνθέσουμε listes και dictionaries από άλλες listes και dictionaries:

In [67]:
```
[d, d, d["a"]]
```

Out[67]:
```
[{'a': {2: 'a'}, 3: ['hello', False, []], 3.1: True},
 {'a': {2: 'a'}, 3: ['hello', False, []], 3.1: True},
 {2: 'a'}]
```

In [68]:
```
{"a": d, "b": d[3]}
```

Out[68]:
```
{'a': {'a': {2: 'a'}, 3: ['hello', False, []], 3.1: True},
 'b': ['hello', False, []]}
```

Υπάρχει και το άδειο dictionary

In [69]:
```
a = {}
```

Η `len` επιστρέφει το πλήθος των εγγραφών που έχει ένα dictionary:

In [70]:
```
person = {"name": "alex", "age": 50, "occupation": "master"}
```

In [71]:
```
len(person)
```

Out[71]:
```
3
```

In [72]:
```
len({})
```

Out[72]:
```
0
```

Μπορούμε να ελέγξουμε αν ένα κλειδί υπάρχει σε ένα dictionary

In [73]:
```
"name" in person
```

Out[73]:
```
True
```

In [74]:
```
"alex" in person
```

Out[74]:
```
False
```

Μπορούμε να ελέγξουμε αν μία τιμή υπάρχει σε dictionary:

In [75]:
```
"alex" in person.values()
```

Out[75]:
```
True
```

Μπορούμε να κάνουμε επανάληψη σε όλα τα στοιχεία ενός dictionary:

In [76]:
```
for i in person:
    print (i)
```

```
name
age
occupation
```

In [77]:
```
for i in person:
    print ("key: {}  Value: {}".format(i, person[i]))
```

```
key: name  Value: alex
key: age  Value: 50
key: occupation  Value: master
```

Προσοχή! Μία ερώτηση είναι "με ποια σειρά επιτρέφονται τα ζευγάρια κλειδί-τιμή σε ένα dictionary?". Η απάντηση είναι τυχαία! Η python δεν κάνειε κάποια ιδιαίτερη προσπάθεια να διατηρήσει την σειρά των κλειδιών-τιμών. Ο λόγος που γίνεται αυτό είναι ταχύτητα και μνήμη. Το dictionary είναι μια δομή που βελτιστοποιεί τη ταχύτητα αποθήκευσης και ανάκλησης των εγγραφών. Έχει εξαιρετική επίδοση για εγγραφές πολλών εκατομυρίων (δοκιμάστε το!). Αν υλοποιούσε και ταξινόμηση των εγγραφών τότε θα έχανε σε ταχύτητα.

Παρόλα αυτά υπάρχει η δομή OrderedDict (https://docs.python.org/3/library/collections.html#collections.OrderedDict) η οποία υλοποιεί αυτή τη δυνατότητα. Η OrderedDict πρέπει να τη κάνει κάποιος `import` για να τη χρησιμοποιείσει. ΘΑ μάθουμε σε επόμενο μάθημα πως γίνεται αυτό.

ΠΡΟΣΟΧΗ 2!. Την ώρα που έγραφα τα παραπάνω διάβασα αυτό (https://stackoverflow.com/questions/39980323/dictionaries-are-ordered-in-python-3-6). Στη python 3.6 και μετά τα dictionaries είναι ταξινομημένα! Δεδομένου ότι αυτό ισχύει μόνο για κάποιες εκδόσεις και κάποιες υλοποιήσεις της python, θα σας πρότεινα (προς το παρόν) να είστε πολύ προσεκτικοί και ο κώδικας σας να είναι τέτοιος που να υποθέτει ότι τα dictionaries δεν είναι ταξινομημένα. το γράφει άλλωστε και στο documentation (https://docs.python.org/3.6/whatsnew/3.6.html#other-language-changes):

> The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon (this may change in the future,

Φαίνεται ότι στη παρούσα έκδοση όντως η σειρά των εγγραφών διατηρείται (η σειρά που μπήκαν στο dictionary):

In [78]:
```
{"z":0, "k":0, "e":0, "f":0, "r": 0, "w":0, "l":0}.keys()
```

Out[78]:
```
dict_keys(['z', 'k', 'e', 'f', 'r', 'w', 'l'])
```

Όταν όμως έτρεξα το ίδιο σε μια παλιότερη python τότε η σειρά ήταν αλφαριθμητικά ταξινομημένη...

```
>>> {"z":0, "k":0, "e":0, "f":0, "r": 0, "w":0, "l":0}.keys()
['e', 'f', 'k', 'l', 'r', 'w', 'z']
```

Προσπέλαση στοιχείων σε dictionary

Μπορούμε να χρησιμοποιήσουμε πάνω από μια φορά το `[][]` ώστε να προσπελάσουμε κάποιο στοιχείο:

In [80]:
```
person = {"name": "alex", "age": 50, "occupation": "master", "exper": ["pytho
n", "karate"]}
```

In [81]:
```
print (person)
```

```
{'name': 'alex', 'age': 50, 'occupation': 'master', 'exper': ['python', 'karat
e']}
```

In [82]:
```
print (person['exper'][0])
```

```
python
```

In [83]:
```
print (person['exper'][1])
```

```
karate
```

In [84]:
```
print (person['exper'])
```

```
['python', 'karate']
```

In [85]:
```
a = ["a", "b", {"name": "mitsos", "surnmae": "sdfsdfsdf"}]
```

In [86]:
```
a[0]
```

Out[86]: `'a'`

In [87]:
```
a[1]
```

Out[87]: `'b'`

In [88]:
```
a[2]
```

Out[88]: `{'name': 'mitsos', 'surnmae': 'sdfsdfsdf'}`

In [89]:
```
a[2]['name']
```

Out[89]: `'mitsos'`

Iteration σε ένα dictionary

Έστω ένα list και ένα dictionary:

In [90]:
```
a = [1,2,3]
b = {"a":1, "b":2, "c":3}
```

Μπορούμε να κάνουμε iterate (επανάληψη) σε ένα list ως εξής:

In [92]:
```
for x in a:
    print (x)
```

```
1
2
3
```

Το ίδιο μπορούμε να κάνουμε και σε ένα dictionary:

In [93]:
```
for x in b:
    print (x, b[x])
```

```
a 1
b 2
c 3
```

Μπορούμε όμως να πάρουμε τα ζευγάρια κλειδιά-τιμές του dictionary ως μία λίστα χρησιμοποιώντας την `items()`

In [94]:
```
list(b.items())
```

Out[94]:
```
[('a', 1), ('b', 2), ('c', 3)]
```

Άρα όπως έχουμε δει και από πριν μπορούμε να κάνουμε iterate και να αναθέσουμε σε δύο μεταβλητές το κλειδί-τιμή κάθε μέλους του dictionary:

In [95]:
```
for x,y in b.items():
    print (x,y)
```

```
a 1
b 2
c 3
```

Dictionary Comprehension

Σε προηγούμενη διάλεξη είχαμε πει τα list comprehensions

In [96]:
```
# List comprehension
[x for x in [1,2,3,4] if x>2]
```

Out[96]:
```
[3, 4]
```

Είχαμε πει ότι το παραπάνω είναι ισοδύναμο με:

In [97]:
```
a = []
for x in [1,2,3,4]:
    if x>2:
        a.append(x)
print (a)
```

```
[3, 4]
```

Το ίδιο μπορούμε να κάνουμε και με τα dictionaries:

In [98]:
```
{ x:x*10 for x in range(1,10)}
```

Out[98]:
```
{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60, 7: 70, 8: 80, 9: 90}
```

Αυτό είναι ισοδύναμο με:

In [99]:
```
a={}
for x in range(1,10):
    a[x] = x*10
print (a)
```

```
{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60, 7: 70, 8: 80, 9: 90}
```

Ένα άλλο παράδειγμα:

In [100]:
```
{ x:'hello {}'.format(x*10) for x in range(1,10)}
```

Out[100]:
```
{1: 'hello 10',
 2: 'hello 20',
 3: 'hello 30',
 4: 'hello 40',
 5: 'hello 50',
 6: 'hello 60',
 7: 'hello 70',
 8: 'hello 80',
 9: 'hello 90'}
```

Σύνολα

Η `set` είναι μία δομή δεδομένων που μοντελοποιεί ένα σύνολο. Κάθε στοιχείο σε ένα set μπορεί να υπάρχει **μόνο μία φορά**:

In [101]:
```
set([1,2,3])
```

Out[101]:
```
{1, 2, 3}
```

In [102]:
```
set([1,2,3,2])
```

Out[102]:
```
{1, 2, 3}
```

In [103]:
```
a = set(['a','b', 'a'])
a
```

Out[103]:
```
{'a', 'b'}
```

In [104]:
```
'b' in a
```

Out[104]:
```
True
```

In [105]:
```
set("Hello World!")
```

Out[105]:
```
{' ', '!', 'H', 'W', 'd', 'e', 'l', 'o', 'r'}
```

Η πράξη `&` μεταξύ δύο set μας επιστρέφει την τομή των συνόλων:

In [106]:
```
a = set([1,2,3,4])
b = set([3,4,5,6])
a & b
```

Out[106]: {3, 4}

Η πράξη | μεταξύ δύο set μας επιστρέφει την ένωση των συνόλων:

In [107]:
```
a | b
```

Out[107]: {1, 2, 3, 4, 5, 6}

Η πράξη - μεταξύ δύο σετ α και β μας επιστρέφει τα στοιχεία της α που δεν υπάρχουν στην β :

In [109]:
```
a - b
```

Out[109]: {1, 2}

In [110]:
```
b - a
```

Out[110]: {5, 6}

In [111]:
```
(a - b) & (b-a)
```

Out[111]: set()

Τα sets είναι ένας επιπλέον τύπος δεδομένων:

In [112]:
```
type(set([1,2,3]))
```

Out[112]: set

In [113]:
```
a = set([1,2,3])
type(a) is set
```

Out[113]: True

set comprehension

Όπως ακριβώς με τις λίστες και τα dictionaries, μπορούμε να έχουμε comprehensions και με τα sets:

In [114]:
```
{x%4 for x in range(10)}
```

Out[114]: {0, 1, 2, 3}

Παραλειπόμενα

Όπως πάντα ξεκινάμε με κάποια παραλειπόμενα από τις προηγούμενες διαλέξεις

Το εκθετικό

Το εκθετικό στη python είναι ** και όχι ^ :

In [413]:
```
2**2
```

Out[413]: 4

In [414]:
```
2^2
```

Out[414]: 0

Το ^ ονομάζεται bitwise XOR (https://en.wikipedia.org/wiki/Bitwise_operation#XOR)

shadowing

Στη python (σε αντίθεση με άλλες γλώσσες) μπορείτε να χρησιμοποιήσετε ως ονόματα μεταβλητών, ονόματα συνερτήσεων της python. **Προσοχή** αποφύγετε να το κάνετε αυτό γιατί πολύ απλά, "χάνετε" τις συναρτήσεις αυτές!

In [415]:
```
list(range(10))
```

Out[415]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [416]:
```
list = 5
```

In [417]:
```
list(range(10)) # Πετάει μήνημα λάθους. Η list πλέον έχει αντικατασταθεί από το 5!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-417-e73507177191> in <module>()
----> 1 list(range(10)) # Πετάει μήνημα λάθους. Η list πλέον έχει αντικαταστα
θεί από το 5!

TypeError: 'int' object is not callable
```

Συναρτήσεις με προκαθορισμένα ορίσματα

Θυμόμαστε λίγο τις συναρτήσεις:

In [394]:
```
def f():
    return 4
```

In [395]:
```
f()
```

Out[395]: 4

In [396]:
```
def f(a,b):
    return a+b
```

In [397]:
```
f(1,2)
```

Out[397]: 3

Σε μία συνάρτηση μπορούμε να ορίσουμε μια προκαθορισμένη τιμή για κάποια παράμετρο:

In [398]:
```
def f(a, b=2):
    return a+b
```

Αν δώσουμε τιμή σε αυτή τη παράμετρο τότε η προκαθοριμένη τιμή αγνοείται:

In [400]:
```
f(5,3)
```

Out[400]: 8

Αν όμως δεν δώσουμε τότε χρησιμοποιεί τη προκαθορισμένη τιμή:

In [401]:
```
f(5) # Επιστρέφει 5+2
```

Out[401]: 7

Συναρτήσεις που επιστρέφουν παραπάνω από 1 τιμή

Στη python μία συνάρτηση μπορεί να επιστρέψει παραπάνω από μία τιμή:

In [402]:
```
def f():
    return 1,2
```

In [403]:
```
a,b = f()
print (a)
print (b)
```

```
1
2
```

Αν αποθηκεύσουμε σε μία μόνο μεταβλητή το αποτέλεσμα μίας συνάρτησης η οποία επιστρέφει παραπάνω από μία τιμές τότε αυτό που επιστρέφει είναι ένα tuple (θα το πούμε αμέσως μετά)

In [404]:
```
a = f()
```

In [405]:
```
print (a)
```

```
(1, 2)
```

In [409]:
```
type(a)
```

Out[409]: list

Tuples

Τα tuples είναι δομές δεδομένων που μοιάζουν με τη λίστα. Η διαφορά τους είναι ότι στα tuples δεν μπορούμε να αλλάξουμε μία τιμή. Αντί για αγκύλες (`[1,2,3]`) στα tuples χρησημοποιούμε παρενθέσεις (`(1,2,3)`).

In [410]:
```
a = (1,2,3)
type(a)
```

Out[410]: tuple

In [411]:
```
a[2] = 7 #Πετάει μήνυμα λάθους. Δεν μπορούμε να το αλλάξουμε
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-411-42aa7089a9ac> in <module>()
----> 1 a[2] = 7 #Πετάει μήνυμα λάθους. Δεν μπορούμε να το αλλάξουμε

TypeError: 'tuple' object does not support item assignment
```

In [412]:
```
b = [1,2,3]
b[2] = 7 # Αυτό είναι οκ, δεν πετάει μήνυμα λάθους
```

Παρόλο που στα tuples δεν μπορώ να προσθέσω ή να αφαιρέσω ένα στοιχείο μπορώ να βάλω στοιχεία στις λίστες ή στα dictionaries που περιέχουν:

In [432]:
```
a = (1,[4,5],10)
a[1].append(6)
print (a)
```

```
(1, [4, 5, 6], 10)
```

Sorting

Με την εντολή `sorted` μπορούμε να ταξινομήσουμε μία λίστα:

In [79]:
```
a = [3,4,5,3,2,1]
```

In [80]:
```
sorted(a)
```

Out[80]:
```
[1, 2, 3, 3, 4, 5]
```

Προσοχή! η `sorted` ΔΕΝ αλλάζει τη λίστα. Αποθηκεύει το αποτέλεσμα σε μία άλλη μεταβλητή:

In [81]:
```
a
```

Out[81]:
```
[3, 4, 5, 3, 2, 1]
```

In [82]:
```
b = sorted(a)
```

In [83]:
```
b
```

Out[83]:
```
[1, 2, 3, 3, 4, 5]
```

Μπορούμε να ταξινομήσουμε μόνο λίστες που έχουν τον ίδιο τύπο δεδομένων:

In [419]:
```
sorted(["b", "a", "c"])
```

Out[419]:
```
['a', 'b', 'c']
```

In [88]:
```
sorted(["b", "a", 100, "c"])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-88-b245b7eeb5df> in <module>()
----> 1 sorted(["b", "a", 100, "c"])

TypeError: '<' not supported between instances of 'int' and 'str'
```

Μπορούμε να ταξινομήσουμε από το μεγαλύτερο προς το μικρότερο:

In [89]:
```
sorted([3,4,5,2,3,4,5,2,1], reverse=True)
```

Out[89]:
```
[5, 5, 4, 4, 3, 3, 2, 2, 1]
```

Όπως και με τη `min` και τη `max` , αν αυτό που ταξινομούμε είναι λίστα από λίστες (ή tuple), τότε ελέγχει πρώτα το πρώτο στοιχείο της υπολίστας. Αν είναι ίσο, τότε ελέγχει το δεύτερο κτλ:

In [92]:
```
a = [
    ["mitsos", 50],
    ['gianni', 40],
    ['gianni', 30]
]
sorted(a)
```

Out[92]:
```
[['gianni', 30], ['gianni', 40], ['mitsos', 50]]
```

Στο παραπάνω παράδειγμα το `['gianni', 30]` είναι μικρότερο από το `['gianni', 40]` :

In [420]:
```
['gianni', 30] < ['gianni', 40]
```

Out[420]:
```
True
```

Πολλές φορές θέλουμε να ταξινομήσουμε μία λίστα που περιέχει υπολίστες αλλά θέλουμε η ταξινόμηση να γίνει όχι με βάση το πρώτο στοιχείο αλλά με βάση μία δική μας συνάρτηση. Π.χ. Έστω η λίστα:

In [421]:
```
a = [["gianni", 30, 20000], ["mitsos", 50, 4000], ["anna", 60, 100000]]
```

Ας υποθέσουμε ότι θέλουμε να ταξινομήσουμε τα στοιχεία της λίστας με βάση το τρίτο στοιχείο τους (20000, 4000, 100000). Προσέχτε ότι αν τρέξουμε τη `sorted` τότε δεν θα μας επιστρέψει αυτό που θέλουμε:

In [422]:
```
sorted(a)
```

Out[422]:
```
[['anna', 60, 100000], ['gianni', 30, 20000], ['mitsos', 50, 4000]]
```

Εμείς θέλουμε το στοιχείο που έχει το 4000 να βγει πρώτο μετά το στοιχείο που έχει το 20000 να βγει δεύτερο και το στοιχείο που έχει το 100000 να βγει τελευταίο.

Σε αυτή τη περίπτωση μπορούμε να φτιάξουμε μία συνάρτηση η οποία όταν παίρνει ως όρισμα κάποιο στοιχείο μιας λίστας να μας επιστρέφει την τιμή μέσω της οποίας θα γίνει η ταξινόμηση:

In [423]:
```
def sort_according_to_this(x):
    return x[2]
```

Αν τώρα βάλω σε αυτή τη συνάρτηση ένα στοιχείο της λίστας θα μου επιστρέψει το τρίτο στοιχείο του, το οποίο είναι και αυτό που θέλω να βασιστεί η ταξινόμηση:

```
In [424]: sort_according_to_this(a[0])
Out[424]: 20000

In [425]: sort_according_to_this(a[1])
Out[425]: 4000

In [426]: sort_according_to_this(a[2])
Out[426]: 100000
```

Τώρα μπορώ να περάσω ως όρισμα τη συνάρτηση `sort_according_to_this` στη `sorted` και να ταξινομήσει τη λίστα `a` με βάση το τρίτο στοιχείο του κάθε στοιχείου της:

```
In [427]: sorted(a, key=sort_according_to_this)
Out[427]: [['mitsos', 50, 4000], ['gianni', 30, 20000], ['anna', 60, 100000]]
```

Προσέχτε ότι τα τρίτα στοιχεία της λίστας είναι ταξινομημένα από το μικρότερο στο μεγαλύτερο. Το ίδιο μπορεί να γίνει με lambda function:

```
In [428]: sorted(a, key=lambda x:x[2])
Out[428]: [['mitsos', 50, 4000], ['gianni', 30, 20000], ['anna', 60, 100000]]
```

Ένα άλλο παράδειγμα. Έστω η λίστα:

```
In [113]: a = ["heraklion", "patras", "thessaloniki", "athens"]

In [114]: sorted(a)
Out[114]: ['athens', 'heraklion', 'patras', 'thessaloniki']
```

Η παρακάτω εντολή ταξινομεί τη συνάρτηση με βάσει το μήκος των strings:

```
In [115]: sorted(a, key=lambda x : len(x))
Out[115]: ['patras', 'athens', 'heraklion', 'thessaloniki']
```

Μπορεί να γραφεί ακόμα πιο σύντομα:

```
In [430]: sorted(a, key=len)
Out[430]: [['gianni', 30, 20000], ['mitsos', 50, 4000], ['anna', 60, 100000]]
```