Προγραμματισμός με τη γλώσσα python

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

Διάλεξη 3η, Παρασκευή 1 Νοεμβρίου 2017

List Comprehensions

Τα List Comprehensions είναι ένας μηχανισμός για να δημιουργήσουμε μία λίστα από κάποια άλλη λίστα. Η επίσημη περιγραφή βρίσκεται εδώ (https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions). Η γενικότερη μορφή είναι:

```
a = [ΕΚΦΡΑΣΗ for ΜΕΤΑΒΛΗΤΗ in LIST]
```

Το οποίο είναι ισοδύναμο με:

```
a = []
for ΜΕΤΑΒΛΗΤΗ in LIST:
    a.append(ΕΚΦΡΑΣΗ)
```

Παράδειγμα:

In [102]:
```
a = [1,2,3]
```

In [103]:
```
b = [x+1 for x in a]
print (b)
```

```
[2, 3, 4]
```

Το παραπάνω είναι ισοδύναμο με:

In [105]:
```
a = [1,2,3]
b = []
for x in a:
    b.append(x+1)
print (b)
```

```
[2, 3, 4]
```

Κάποια άλλα παραδείγματα:

In [107]:
```
a = ["a", "b", "c"]
["hello: " + x for x in a]
```

Out[107]:
```
['hello: a', 'hello: b', 'hello: c']
```

In [108]:
```
a = ["a", "b", "c"]
b  = [x * 3 for x in a]
print (b)
```

```
['aaa', 'bbb', 'ccc']
```

In [109]:
```
a = [1,2,3,4,5,6]
[x/2 for x in a]
```

Out[109]:
```
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
```

Επίσης μπορούμε να χρησημοποιήσουμε την `if` στα list comprehensions. Η γενικότερη μορφή είναι:

```
a = [ΕΚΦΡΑΣΗ for ΜΕΤΑΒΛΗΤΗ in LIST if ΛΟΓΙΚΗ_ΕΚΦΡΑΣΗ]
```

Το οποίο είναι ισοδύναμε με:

```
a = []
for ΜΕΤΑΒΛΗΤΗ in LIST:
    if ΛΟΓΙΚΗ_ΕΚΦΡΑΣΗ:
        a.append(ΕΚΦΡΑΣΗ)
```

Παραδείγματα:

In [110]:
```
a = [1,2,3,4,5,6]
[x/2 for x in a if x>4]
```

Out[110]:
```
[2.5, 3.0]
```

Αυτό είναι ισοδύναμο με:

In [111]:
```
a = [1,2,3,4,5,6]
b = []
for x in a:
    if x>4:
        b.append(x/2)
print (b)
```

```
[2.5, 3.0]
```

Άλλο ένα παράδειγμα. Έστω η λίστα:

In [112]:
```
a = [1,2,3,4,5,4,3,5,6,7,8,7,6,5,5,4]
```

Ποιές είναι όλες οι θέσεις που έχουν τι τιμή 4;

Πρώτος τρόπος:

In [113]:
```
[i for i, x in enumerate(a) if x==4]
```

Out[113]:
```
[3, 5, 15]
```

Δεύτερος τρόπος:

In [114]:
```
a = [1,2,3,4,5,4,3,5,6,7,8,7,6,5,5,4]
[x for x in range(len(a)) if a[x] == 4]
```

Out[114]:
```
[3, 5, 15]
```

In [115]:
```
list(range(len(a)))
```

Out[115]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Επίσης ένα list comprehension μπορεί να έχει παραπάνω από 1 `for` :

In [116]:
```
a = [1,2,3]
b = ["a", "b", "c"]
["{}{}".format(x,y) for x in a for y in b]
```

Out[116]:
```
['1a', '1b', '1c', '2a', '2b', '2c', '3a', '3b', '3c']
```

Αυτό είναι ισοδύναμο με:

In [117]:
```
c = []
for x in a:
    for y in b:
        c.append("{}{}".format(x,y))
print (c)
```

```
['1a', '1b', '1c', '2a', '2b', '2c', '3a', '3b', '3c']
```

Χρησιμοποιώντας list comprehension μπορούμε να παράξουμε ένα string που περιέχει όλη τη προπαίδεια:

In [118]:
```
print ('\n'.join(["{} X {} = {}".format(x,y,x*y) for x in range(1,11) for y in ra
```

```
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
2 X 10 = 20
3 X 1 = 3
3 X 2 = 6
3 X 3 = 9
3 X 4 = 12
3 X 5 = 15
3 X 6 = 18
3 X 7 = 21
3 X 8 = 24
3 X 9 = 27
3 X 10 = 30
4 X 1 = 4
4 X 2 = 8
4 X 3 = 12
4 X 4 = 16
4 X 5 = 20
4 X 6 = 24
4 X 7 = 28
4 X 8 = 32
4 X 9 = 36
4 X 10 = 40
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
6 X 1 = 6
6 X 2 = 12
6 X 3 = 18
6 X 4 = 24
6 X 5 = 30
6 X 6 = 36
6 X 7 = 42
6 X 8 = 48
6 X 9 = 54
6 X 10 = 60
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
```

In [119]:
```
print ('\n'.join(["{} X {} = {}".format(x,y,x*y) for x in range(1,11) for y in ra
```

```
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
2 X 10 = 20
3 X 1 = 3
3 X 2 = 6
3 X 3 = 9
3 X 4 = 12
3 X 5 = 15
3 X 6 = 18
3 X 7 = 21
3 X 8 = 24
3 X 9 = 27
3 X 10 = 30
4 X 1 = 4
4 X 2 = 8
4 X 3 = 12
4 X 4 = 16
4 X 5 = 20
4 X 6 = 24
4 X 7 = 28
4 X 8 = 32
4 X 9 = 36
4 X 10 = 40
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
6 X 1 = 6
6 X 2 = 12
6 X 3 = 18
6 X 4 = 24
6 X 5 = 30
6 X 6 = 36
6 X 7 = 42
6 X 8 = 48
6 X 9 = 54
6 X 10 = 60
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
```

Η συνάρτηση range

Η συνάρτηση `range` δημιουργεί κάτι* που αναπαραστάει μία αριθμητική ακολουθία.

* Αυτό το "κάτι" ονομάζεται generator και θα το μάθουμε στην επόμενη διάλεξη

In [32]:
```
range(1,10) # Αυτό είναι ένα generator
```

Out[32]:
```
range(1, 10)
```

Το σημαντικό είναι ότι αν του εφαρμόσουμε τη `list` τότε μας επιστρέφει μία λίστα:

In [33]:
```
list(range(10)) # Από το 0 έως το 10 (χωρίς το 10)
```

Out[33]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [34]:
```
list(range(5,10)) # Από το 5 έως το 10 (χωρίς το 10)
```

Out[34]:
```
[5, 6, 7, 8, 9]
```

In [35]:
```
list(range(1,11,2)) # Από το 1 έως το 11 (χωρίς το 11) με βήμα 2
```

Out[35]:
```
[1, 3, 5, 7, 9]
```

Η αριθμητική πρόοδος μπορεί να είναι και "ανάποδη"

In [36]:
```
list(range(11,1,-2))
```

Out[36]:
```
[11, 9, 7, 5, 3]
```

In [37]:
```
list(range(10,1,-1))
```

Out[37]:
```
[10, 9, 8, 7, 6, 5, 4, 3, 2]
```

Η `list(range(...))` επιστρέφει μία λίστα που μπορούμε να κάνουμε πράξεις κανονικά όπως έχουμε μάθει:

In [38]:
```
a = list(range(100, 120, 5)) + ["mitsos"]
print (a)
```

```
[100, 105, 110, 115, 'mitsos']
```

Γιατί στη python όποτε βλέπουμε το `"XYZ"[a:b]` , `[1,2,3][a:b]` , `range(a,b)` αυτό σημαίνει από το `a` μέχρι το `b` **ΧΩΡΙΣ ΤΟ `b`** ; Αυτή η ιστορία ξεκινάει από πολύυυυυ παλιά (https://en.wikipedia.org/wiki/Zero-based_numbering). Το βασικό είναι ότι όταν λέμε σε έναν υπολογιστή "θέλω Ν" στοιχεία, τότε με βάση μιας παλιάς σύμβασης, το πρώτο στοιχείο έχει δείκτη (index) 0, το δεύτερο 1, κτλ. Οπότε όταν λέμε `range(10)` ο υπολογιστής φτιάχνει τη λίστα από το 0 μέχρι και το 9. Όταν λέμε range(5,7) τότε στην ουσία ζητάμε από τον υπολογιστή μία λίστα με 2 στοιχεία (7-5). Το πρώτο σύμφωνα με την ίδια σύμβαση θα είναι "από εκεί που ξεκινάει η αρίθμηση" δηλαδή το 5. Αφού η λίστα πρέπει να έχει 2 στοιχεία τότε το δεύτερο θα είναι το επόμενο δηλαδή το 6. Αυτή η αρίθμηση μας "βολεύει" και για κάποιους μαθηματικούς υπολογισμούς (https://www.johndcook.com/blog/2008/06/26/why-computer-scientists-count-from-zero/). Κάποια επιπλέον παραδείγματα:

In [39]:
```
list(range(3,10,2))
```

Out[39]:
```
[3, 5, 7, 9]
```

In [40]:
```
list(range(3,11,2))
```

Out[40]:
```
[3, 5, 7, 9]
```

In [41]:
```
list(range(3,12,2))
```

Out[41]:
```
[3, 5, 7, 9, 11]
```

In [42]:
```
list(range(10)) # list(range(0,10))
```

Out[42]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [43]:
```
list(range(5,7)) # list(range(a,b)) # b-a
```

Out[43]:
```
[5, 6]
```

Η εντολή `for`

Με τον εντολή `for` μπορούμε να επαναλάβουμε εντολές για κάθε στοιχεία μιας λίστας

In [44]:
```
for x in [1,4,6]: print (x)
```

```
1
4
6
```

In [45]:
```
for x in [1,4,6]: print ("The number is:", x)
```

```
The number is: 1
The number is: 4
The number is: 6
```

In [46]:
```
for x in range(1,10): print ("The number is:", x)
```

```
The number is: 1
The number is: 2
The number is: 3
The number is: 4
The number is: 5
The number is: 6
The number is: 7
The number is: 8
The number is: 9
```

In [47]:
```
for i in range(1,10,3): print ("Hello:", i)
```

```
Hello: 1
Hello: 4
Hello: 7
```

Αν οι εντολές που θέλουμε να επαναλάβουμε είναι πάνω από 1 τότε είναι **ΥΠΟΧΡΕΩΤΙΚΟ** να τις βάλουμε στην επόμενη γραμμή και πιο μέσα. Αυτό ονομάζεται υποχρεωτικό indentation (https://en.wikipedia.org/wiki/Indentation_style) ή αλλιώς κανόνας off-side (https://en.wikipedia.org/wiki/Off-side_rule)!

In [48]:
```
for i in range(1,10,3):
    print ("command A:", i)
    print ("command B:", i)
```

```
command A: 1
command B: 1
command A: 4
command B: 4
command A: 7
command B: 7
```

Αν υπάρχει `for` μέσα στη `for` τότε πρέπει οι επόμενες γραμμές να μπούν ακόμα πιο μέσα:

In [50]:
```
for i in range(1,5):
    for j in range(1,5):
        print (i,j)
```

```
1 1
1 2
1 3
1 4
2 1
2 2
2 3
2 4
3 1
3 2
3 3
3 4
4 1
4 2
4 3
4 4
```

Παράδειγμα: Ο πίνακας πολλαπλασιασμού:

In [51]:
```
for i in range(1,11):
    for j in range(1,11):
        print ("{} X {} = {}".format(i,j,i*j))
    print ("=" * 10)
```

```
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
==========
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
```

Επίσης μπορούμε να κάνουμε επανάληψη χρησιμοποιώντας string αντί για λίστα:

In [52]:
```
for letter in "python":
    print (letter)
```

```
p
y
t
h
o
n
```

Αν μία λίστα έχει υπο-λίστες με παραπάνω από 1 στοιχεία τότε μπορούμε να χρησιμοποιήσουμε παραπάνω απο 1 μεταβλητές στη `for` :

In [53]:
```
a = [[2, "Crete"], [3, "Cyprus"], [4, "Majiorca"]]
for x, y in a:
    print ("Number: {} Island: {}".format(x,y))
```

```
Number: 2 Island: Crete
Number: 3 Island: Cyprus
Number: 4 Island: Majiorca
```

Φυσικά το ίδιο μπορεί να γίνει αν έχει υπο-λίστες με 3 στοιχεία κτλ..

In [54]:
```
a = [[1,2,3], ["a", "b", "c"]]
for x,y,z in a:
    print ("{} {} {}".format(x,y,z))
```

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

Η `if` μπορεί να είναι μέσα σε μία `for` :

In [55]:
```
for i in range(1,10):
    if i>5:
        print (i)
```

```
6
7
8
9
```

In [56]:
```
for i in range(1,10):
    if i>=5:
        print (i)
```

```
5
6
7
8
9
```

if ... elif ... else

Το είπαμε και στη προηγούμενη διάλεξη, αλλά.. ας τα ξαναπούμε!

Στην `if` μπορούμε να δηλώσουμε παραπάνω από μία συνθήκες ή να να χρησιμοποιήσουμε την `else` για να δηλώσουμε τι να γίνει αν όλες οι συνθήκες στις `if` και `elif` είναι `False`

In [49]:
```
age = 23
if age<18:
    status = 'ανήλικος'
else:
    status = 'ενήλικος'

print (status)
```

```
ενήλικος
```

Προσέχτε ότι το παραπάνω δεν ελέγχει τη περίπτωση λάθους:

In [51]:
```
age = -4

if age<18:
    status = 'ανήλικος'
else:
    status = 'ενήλικος'

print (status)
```

```
ανήλικος
```

Μπορούμε να βάλουμε πολλές `elif` ώστε να ελέγχουμε για πολλά ενδεχόμενα:

In [55]:
```
age = 50
if age <0:
    status = "λάθος. Αρνητική τιμή"
elif age < 18:
    status = "ανήλικος"
elif age < 120:
    status = "ενήλικος"
else:
    status = "λάθος. Υπερβολικά μεγάλη τιμή"
print (status)
```

```
ενήλικος
```

Δοκιμάστε το παραπάνω για διάφορες τιμές του `age` .

Επίσης, δεν είναι απαραίτητο να χρησιμοποιήσουμε την `else` :

In [58]:
```
age = 150
if age <0:
    status = "λάθος. Αρνητική τιμή"
elif age < 18:
    status = "ανήλικος"
elif age < 120:
    status = "ενήλικος"
elif age >= 120:
    status = "λάθος. Υπερβολικά μεγάλη τιμή"
print (status)
```

```
λάθος. Υπερβολικά μεγάλη τιμή
```

types

Η συνάρτηση `type` μας επιστρέφει τον τύπο μιας μεταβλητής:

In [262]:
```
type(3)
```

Out[262]: int

In [263]:
```
type('hello')
```

Out[263]: str

In [264]:
```
type(5.2)
```

Out[264]: float

In [265]:
```
type(True)
```

Out[265]: bool

In [266]:
```
type([1,2,3])
```

Out[266]: list

In [267]:
```
type({"a":3})
```

Out[267]: dict

Πως μπορώ να ελέγξω τι τύπος είναι μια μεταβλητή;

In [269]:
```
a='hello'
if type(a) is str:
    print ("This is string")
```

```
This is string
```

Χρησιμοποιώντας το `type(variable).__name__` μπορώ να πάρω τον τύπο μιας μεταβλητής ως string:

In [271]:
```
a = False
type(a).__name__
```

Out[271]: 'bool'

Μετατροπή από έναν τύπο σε έναν άλλο

Η python μας δίνει τις εξής συναρτήσεις για να μετατρέψουμε από έναν τύπο σε έναν άλλο:

In [298]:
```
str(3)
```

Out[298]: '3'

In [299]:
```
str(True)
```

Out[299]: 'True'

In [300]:
```
str([1,2,3])
```

Out[300]: '[1, 2, 3]'

In [301]:
```
str({1:2})
```

Out[301]: '{1: 2}'

In [302]:
```
int('3')
```

Out[302]: 3

In [303]:
```
int(4.5)
```

Out[303]: 4

In [305]:
```
int(False)
```

Out[305]: 0

In [306]:
```
int(True)
```

Out[306]: 1

In [322]:
```
int('                    9            ')
```

Out[322]: 9

In [324]:
```
int('7 hello')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-324-d90c12f1c0d9> in <module>()
----> 1 int('7 hello')

ValueError: invalid literal for int() with base 10: '7 hello'
```

In [307]:
```
float('4.5')
```

Out[307]: 4.5

In [308]:
```
float(True)
```

Out[308]: 1.0

In [315]:
```
bool('False')
```

Out[315]: True

In [316]:
```
bool('')
```

Out[316]: False

In [317]:
```
bool([1,2])
```

Out[317]: True

In [318]:
```
bool({})
```

Out[318]: False

In [310]:
```
list('hello')
```

Out[310]: ['h', 'e', 'l', 'l', 'o']

In [312]:
```
list({'a':1, 'b': 2})
```

Out[312]: ['a', 'b']

Προσοχή! Δεν μπορούν να μετατραπούν όλοι οι τύποι σε όλους. Π.χ:

In [320]:
```
int([1,2])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-320-b0932f661ffe> in <module>()
----> 1 int([1,2])

TypeError: int() argument must be a string, a bytes-like object or a number, no
t 'list'
```

Πράξεις πάνω σε λίστες

Σε λίστες μπορούμε να κάνουμε τις παρακάτω πράξεις:

In [287]:
```
sum([2,3,4]) # Το άθροισμα όλων των στοιχείων της λίστας:
```

Out[287]: 9

Προσοχή Η `sum` πρέπει να έχει μόνο `int` ή `float`

In [290]:
```
sum(['a', 'b'])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-290-0ca1e4efb8fe> in <module>()
----> 1 sum(['a', 'b'])

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [288]:
```
min([3,5,4]) # Το μικρότερο στοιχείο
```

Out[288]: 3

In [289]:
```
max(['heraklion', 'patras', 'thessaloniki']) # Το μεγαλύτερο στοιχείο
```

Out[289]: 'thessaloniki'

In [71]:
```
max(['heraklion', 'patras', 't'])
```

Out[71]: 't'

Μία ιδιότητα που έχουν οι `max` και η `min` είναι ότι αν η λίστα περιέχει υπολίστες τότε ψάχνουν το μικρότερο στοιχείο στο πρώτο στοιχείο της υπολίστας. Αν υπάρχουν δύο ή περισσότερες υπολίστες με το ίδιο μικρότερο πρώτο στοιχείο τότε ψάχνουν στο δεύτερο κτλ.:

In [144]:
```
min([[5, "b"], [7, "t"], [6, 'r'], [5, 'a']])
```

Out[144]: [5, 'a']

Και αυτό γιατί:

In [145]:
```
[[5, 'a']] < [[5, 'b']]
```

Out[145]: True

Η `all` επιστρέφει `True` αν **όλα** τα στοιχεία μιας λίστας είναι `True`

In [78]:
```
all([True, True, True])
```

Out[78]: True

In [79]:
```
all([True, False, True])
```

Out[79]: False

In [80]:
```
all([3,4,5,4,5])
```

Out[80]: True

In [81]:
```
all([3,4,5,'',4,5])
```

Out[81]: False

Η `any` μας επιστρέφει `True` αν έστω και ένα στοιχείο της λίστας είναι `True` :

In [82]:
```
any([False, False, False])
```

Out[82]: False

In [83]:
```
any([False, False, False, "mitsos"])
```

Out[83]: True

Προσοχή Η άδεια λίστα έχει `all` `True` και `any` `False` :

In [297]:
```
all([])
```

Out[297]: True

In [296]:
```
any([])
```

Out[296]: False

Functions (συναρτήσεις)

Οι [συναρτήσεις (https://en.wikipedia.org/wiki/Subroutine)](https://en.wikipedia.org/wiki/Subroutine) είναι ένα τεράστιο κομμάτι της θεωρίας υπολογιστών. Μέσω των συναρτήσεων μπορούμε να "σπάσουμε" τον κώδικά μας σε μικρα λειτουργικά κομμάτια και να τον κάνουμε πιο οργανωμένο και πιο επαναχρησιμοποιήσουμε. Π.χ. μία συνάρτηση που υπολογίζει τον μέσο όρο μπορούμε να τη χρησιμοποιήσουμε σε πολλά σημεία του κώδικά μας. Στην python ορίζουμε μία συνάρτηση μέσω της `def` :

In [17]:
```
def f():
    print ('hello')

f()
```

hello

Μία συνάρτηση μπορεί να "επιστρέφει" μία τιμή σε αυτόν που την κάλεσε:

In [23]:
```
def f():
    print ('hello')
    return 4
a=f()
print (a)
```

```
hello
4
```

Μία συνάρτηση μπορεί να παίρνει καμία, μία ή παραπάνω παραμέτρους:

In [24]:
```
def f(t):
    return t+1
a=f(3)
print (a)
```

```
4
```

In [25]:
```
def f(q,w,e,r,t):
    return q+w-e/r*t
a=f(2,3,4,5,6)
print (a)
```

```
0.1999999999999993
```

Επίσης μία συνάρτηση μπορεί να έχει παραμέτρους με προκαθορισμένες τιμές. Αν η συνάρτηση κληθεί χωρίς να δοθεί μία τιμή σε αυτές τις παραμέτρους τότε χρησιμοποιείται η προκαθορισμένη τιμή:

In [2]:
```
def f(a,b=4):
    return a+b
```

In [3]:
```
f(2,3)
```

Out[3]:
```
5
```

In [4]:
```
f(2)
```

Out[4]:
```
6
```

Μία συνάρτηση μπορεί να έχει πολλές παραμέτρους με προκαθοριμένες τιμές:

In [5]:
```
def f(a,b=2,c=4):
    return a+b+c
```

Πρέπει όμως όλες αυτές οι παράμετροι να είναι δηλωμένες μετά από τις παραμέτρους χωρίς προκαθορισμένες τιμές:

In [6]:
```
def f(a,b=2,c):
    return 42
```

```
  File "<ipython-input-6-7943a91cece0>", line 1
    def f(a,b=2,c):
            ^
SyntaxError: non-default argument follows default argument
```

Μία συνάρτηση μπορεί να πάρει μεταβλητό πλήθος από παραμέτρους. Αυτό γίνεται αν στο τέλος των παραμέτρων δηλώσουμε μία επιπλέον παράμετρο με αστεράκι. Η επιπλέον παράμετρος που έχει δηλωθεί με αυτόν τον τρόπο θα περιέχει ένα tuple με όλες τις επιπλέον τιμές με τις οποίες έχει κληθεί η συνάρτηση:

In [9]:
```
def f(a,b,*c):
    print ('Η παράμετρος a:', a)
    print ('Η παράμετρος b:', b)
    print ('Η παράμετρος c:', c)
```

In [10]:
```
f(1,2,3,4,5,6,7,8)
```

```
Η παράμετρος a: 1
Η παράμετρος b: 2
Η παράμετρος c: (3, 4, 5, 6, 7, 8)
```

Προσέξτε ότι το c περιέχει ένα tuple με τις επιπλέον τιμές που περάσαμε στη συνάρτηση. Αυτό το tuple μπορεί να είναι και άδειο:

In [11]:
```
f(1,2)
```

```
Η παράμετρος a: 1
Η παράμετρος b: 2
Η παράμετρος c: ()
```

Μπορούμε να "περάσουμε" σε μία συνάρτηση τις τιμές της, δηλωμένες σε ένα tuple ή σε μία λίστα, χρησιμοποιώντας το αστεράκι όταν τη καλούμε:

In [12]:
```
def f(a,b,c,d):
    print ('Η παράμετρος a:', a)
    print ('Η παράμετρος b:', b)
    print ('Η παράμετρος c:', c)
    print ('Η παράμετρος d:', d)
```

In [13]:
```
a = [5,8,3,2]
f(*a)
```

```
Η παράμετρος a: 5
Η παράμετρος b: 8
Η παράμετρος c: 3
Η παράμετρος d: 2
```

Μποούμε ακόμα και να αναμείξουμε τις τιμές που παιρνάμε κανονικά και αυτές που περνάμε με το αστεράκι:

In [15]:
```
a = [8,7]
f(2, 4, *a)
```

```
Η παράμετρος a: 2
Η παράμετρος b: 4
Η παράμετρος c: 8
Η παράμετρος d: 7
```

Προσοχή! όταν μεταβάλουμε ένα όρισμα της συνάρτησης, τότε αν αυτό το όρισμα είναι string, int, float ή bool (αυτοί οι τύποι λέγονται primitive (https://en.wikipedia.org/wiki/Primitive_data_type)) τότε αυτή η μεταβολή δεν φαίνεται από εκεί που καλέσαμε τη συνάρτηση:

In [22]:
```
def f(a):
    a = a + 1 # Αλλάζουμε την a

a=4
f(a)
print (a) # Η a δεν άλλαξε!
```

```
4
```

Το παραπάνω δεν ισχύει αν η μεταβλητή είναι λίστα dictionary ή set:

In [26]:
```
def f(a_list):
    a_list.append(1) # Προσθέτουμε τη τιμή 1 στη λίστα a

a=[3,4]
print ('a BEFORE calling f is: {}'.format(a))
f(a)
print ('a AFTER calling f is: {}'.format(a))
```

```
a BEFORE calling f is: [3, 4]
a AFTER calling f is: [3, 4, 1]
```

In [28]:
```
def f(a_dict):
    a_dict['a'] = 4

a = {'b': 5}
print ('a BEFORE calling f is: {}'.format(a))
f(a)
print ('a AFTER calling f is: {}'.format(a))
```

```
a BEFORE calling f is: {'b': 5}
a AFTER calling f is: {'b': 5, 'a': 4}
```

Οι συναρτήσεις δεν μπορούν να "δουν" τα primitive data types (int, string, bool) που έχουν οριστεί έξω από αυτές:

In [137]:
```
a=4
def f():
    a=5
f()
print (a)
```

```
4
```

Μπορούν όμως να "δουν" τα dictionaries, lists, ... που έχουν οριστεί έξω από αυτές:

In [138]:
```
a = []
def f():
    a.append(3)
f()
print (a)
```

```
[3]
```

Για να μπορέσει να "δει" μία συνάρτηση ένα primitive data type που έχει οριστεί έξω από αυτή, μπορούμε να χρησιμοποιήσουμε τη `global` :

Προσοχή Υπάρχει μια εξαιρετικά ενοχλητική ομάδα προγραμματιστών που θεωρεί ότι [ποτέ δεν πρέπει να χρησιμοποιούμε τη global (https://stackoverflow.com/questions/19158339/why-are-global-variables-evil)](). Αγνοήστε τους.

In [1]:
```
a=4
def f():
    global a
    a=5
f()
print (a)
```

```
5
```

Προσοχή! οτιδήποτε υπάρχει "κάτω" από τη `return` αγνοείται:

In [29]:
```
def f():
    print ("hello")
    return 5
    print ("dsfgsdfg")

f()
```

```
hello
```

Out[29]:
```
5
```

Μία συνάρτηση που δεν επιστρέφει τίποτα επιστρέφει μία τιμή που είναι `None` .

In [145]:
```
def f():
    print ("hello")

print (f())
```

```
hello
None
```

Η `None` είναι ένας νέος τύπος δεδομένου:

In [146]:
```
type(None)
```

Out[146]:
```
NoneType
```

Η `None` έχει μία ιδιαιτερότητα: είναι και τιμή αλλά και τύπος δεδομένων:

In [31]:
```
a = None
a is None
```

Out[31]:
```
True
```

Προσέχτε τη διαφορά του παραπάνω με αυτό:

In [32]:
```
a = 3
a is int
```

Out[32]:
```
False
```

Μία συνάρτηση μπορεί να περιέχει μία άλλη συνάρτηση:

In [150]:
```
def f(r):
    def g():
        return r + 5
    return g() + 3

f(1)
```

Out[150]: 9

Οι συναρτήσεις είναι και αυτές μεταβλητές τύπου `function` :

In [35]:
```
type(f)
```

Out[35]: function

Μπορούμε να ελέγξουμε αν μία μεταβλητή είναι συνάρτηση με τη `callable` :

In [36]:
```
callable(f)
```

Out[36]: True