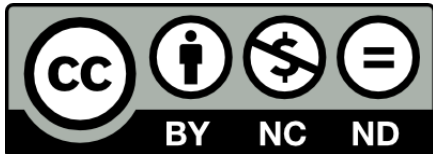


Εισαγωγή στην **Python**

7





Copyright

Το παρόν εκπαιδευτικό υλικό προσφέρεται ελεύθερα υπό τους όρους της άδειας Creative Commons:

- Αναφορά Δημιουργού - Μη Εμπορική Χρήση - Όχι Παράγωγα Έργα 3.0.

Για να δείτε ένα αντίγραφο της άδειας αυτής επισκεφτείτε τον ιστότοπο

<https://creativecommons.org/licenses/by-nc-nd/3.0/gr/>

Στ. Δημητριάδης, 2015

def

Συναρτήσεις Functions



Περιεχόμενα

- Συνάρτηση: Τι είναι και πώς δηλώνεται
- Σύνταξη συναρτήσεων
- Πώς εκτελεί η Python μια συνάρτηση
- Εμβέλεια μεταβλητών
- Πέρασμα Ορισμάτων
- Συναρτήσεις “Γεννήτορες” (Generators)
- Συναρτησιακός προγραμματισμός

Συνάρτηση: Τι είναι και πώς δηλώνεται



def Τι είναι μια συνάρτηση (function)

- Η συνάρτηση **ομαδοποιεί ένα σύνολο εντολών** ώστε να είναι διαθέσιμες να κληθούν –με ένα κοινό όνομα αναφοράς- οποτεδήποτε και οπουδήποτε χρειαστεί στο πρόγραμμα
- (α) Όταν **ορίζεται** μια συνάρτηση καθορίζεται μια ομάδα **παραμέτρων** που δηλώνουν το πέρασμα δεδομένων από το πρόγραμμα που καλεί προς τη συνάρτηση
- (β) Όταν **καλείται** μια συνάρτηση δίνουμε ταυτόχρονα και μια σειρά από «**ορίσματα**» (δεδομένα), τα οποία αντιστοιχούνται στις παραμέτρους της συνάρτησης
- (γ) Όταν **εκτελείται** μια συνάρτηση χρησιμοποιεί τα ορίσματα ώστε να υπολογίσει μία ή περισσότερες **τιμές-αποτέλεσμα**. Οι τιμές αυτές **επιστρέφουν** στο πρόγραμμα που κάλεσε τη συνάρτηση ως αποτέλεσμα εκτέλεσης των εντολών της συνάρτησης.
- Πλεονεκτήματα χρήσης Συναρτήσεων:
- **Επαναχρησιμοποίηση** κώδικα: ελαχιστοποίηση πλεονασμού στον κώδικα
- **Τμηματοποίηση**: διαχωρισμός κώδικα σε σαφή τμήματα με καλά προσδιορισμένες λειτουργίες

def Δήλωση συνάρτησης

```
def name(arg1, arg2, ... argN) :  
    statements  
    return value
```

ΜΗΝ ΞΕΧΝΑΤΕ

- **def**: δεσμευμένη λέξη για τη δήλωση συνάρτησης (από το define – ορίζω)
- **name**: το όνομα της συνάρτησης, ένας αναγνωριστής (identifier)
- **(arg1, arg2, ... argN)**: η λίστα **παραμέτρων** (parameters) (ή **ορισμάτων** (arguments) όταν καλείται) της συνάρτησης (*προαιρετικό*)
- **statements**: το block εντολών που εκτελεί η συνάρτηση όταν καλείται
- **return**: δεσμευμένη λέξη, δηλώνει το τέλος της συνάρτησης και την επιστροφή της τιμής value (*προαιρετικό*)
- **value**: τιμή (μεταβλητή ή έκφραση) που υπολογίστηκε και επιστρέφεται στον κώδικα κλήσης (*προαιρετικό*)

def Παραδείγματα

- 1

```
>>> def ginomeno(x, y):  
        return x*y
```

```
>>> ginomeno(2, 4)  
8
```

- Μπορείτε να δηλώσετε μια απλή συνάρτηση και στο κέλυφος
 - Μετά τη return x*y πατήστε 2 φορές Enter

```
def ginomeno(x, y):  
    return x * y
```

```
a = ginomeno(2, 4)  
print(a)
```

- Γράψτε το ίδιο παράδειγμα στον συντάκτη και τρέξτε το πρόγραμμα
- Ισοδύναμα μπορείτε να γράψετε και:
print(ginomeno(2,4))



def Παραδείγματα

- 2

```
def ginomeno(x, y):  
    return x * y
```

```
a = ginomeno(2, 4)  
print(a)
```

Παράμετροι (parameter): Τα ονόματα των μεταβλητών που γράφουμε στην παρένθεση στον ορισμό της συνάρτησης

Ορίσματα (arguments): Οι τιμές που περνάμε στη συνάρτηση κατά την κλήση της (γράφονται επίσης μέσα στην παρένθεση)

- Κατά την κλήση μιας συνάρτησης τα ορίσματα (τιμές) που περνάτε μέσα στην παρένθεση αντιστοιχούν ένα προς ένα στις παραμέτρους που δηλώσατε στον ορισμό της συνάρτησης (εκτός κι αν ορίσουμε διαφορετικά)
- Στο παραπάνω παράδειγμα:
 - $2 \rightarrow x$
 - $4 \rightarrow y$

def Πολυμορφισμός στις συναρτήσεις

```
def ginomeno(x, y):  
    return x * y  
  
print(ginomeno('Ni', 4))
```

```
>>>  
NiNiNiNi  
>>>
```

- Τι θα τυπώσει η print; Γιατί;
- Τι συμπεράσματα βγάζετε για τον τρόπο λειτουργίας των συναρτήσεων;
- Οι συναρτήσεις λειτουργούν *πολυμορφικά*, δηλ. ανάλογα με τον τύπο των ορισμάτων που δέχονται και εφόσον –φυσικά- οι τελεστές των εντολών έχουν νόημα για τα ορίσματα
- *Πολυμορφισμός*: το νόημα μιας λειτουργίας (πράξης, τελεστή, κλπ.) ισχύει για πολλές μορφές τύπων δεδομένων. Έτσι το τι ακριβώς θα προκύψει ως αποτέλεσμα εξαρτάται από τον τύπο των αντικειμένων στα οποία εφαρμόζονται οι τελεστές

def Πέρασμα ορισμάτων

- 1/2

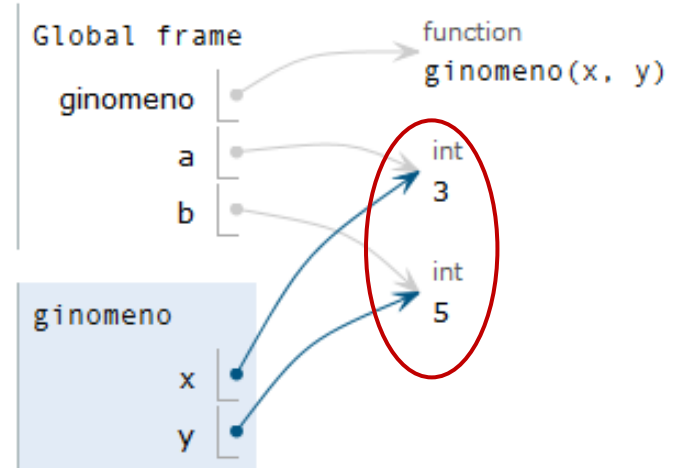
Python 3.3

```
→ 1 def ginomeno(x,y):  
→ 2     return x*y  
3  
4 a = 3  
5 b = 5  
6  
7 c=ginomeno(a, b)  
8 print(c)
```

[Edit code](#)

Frames

Objects



- Όταν καλείται η συνάρτηση τα αντικείμενα-τιμές των a , b στο κύριο πρόγραμμα «αντιστοιχούνται» στις παραμέτρους x & y της συνάρτησης
 - Δηλ. οι x , y αναφέρονται («δείχνουν») στα αντικείμενα-τιμές 3 & 5
- Γενικά: όταν καλείται μια συνάρτηση το πέρασμα ορισμάτων γίνεται με «**αναφορά αντικειμένου**» (object reference). Δηλ. γίνεται μια σύνδεση (binding) των αναγνωριστών στη συνάρτηση με τα αντικείμενα στο κύριο πρόγραμμα
 - Το θέμα αυτό είναι συνθετότερο και αναπτύσσεται στη συνέχεια

def Πέρασμα ορισμάτων

- 2/2



- Μετά την εκτέλεση της συνάρτησης:
- (α) απελευθερώνεται η μνήμη που είχε δεσμευτεί για τις τοπικές μεταβλητές της συνάρτησης
- (β) η συνάρτηση έχει επιστρέψει το αντικείμενο-τιμή που υπολόγισε (μεταβλητή c)

Σύνταξη συναρτήσεων



def (1) ΧΩΡΙΣ ΠΑΡΑΜΕΤΡΟΥΣ / ΧΩΡΙΣ RETURN

- Η συνάρτηση μπορεί να μην έχει παραμέτρους
- Επίσης μπορεί να μην υπάρχει η τελευταία δήλωση return

```
def Hello():  
    print('Hello World!')
```

- Όταν καλείται εκτελείται απλά η print()
- Παρόλο που δεν υπάρχει return θεωρείται (σιωπηρά) πως υπάρχει και επιστρέφει την τιμή **None**

```
def Hello():  
    print('Hello World!')  
    return None
```

def (2) ΧΩΡΙΣ ΠΑΡΑΜΕΤΡΟΥΣ / ΜΕ RETURN

- Η συνάρτηση μπορεί να μην έχει παραμέτρους αλλά επιστρέφει μία τιμή (ή περισσότερες) μέσω της return

```
def ask_name():  
    name = input('Ποιό είναι το όνομά σου: ')  
    return name
```

- Παραδείγματα κλήσης:

```
print('Όνομα: ', ask_name())
```

```
myname = ask_name()  
print(myname)
```

def (3) ΜΕ ΠΑΡΑΜΕΤΡΟΥΣ / ΧΩΡΙΣ RETURN

- Η συνάρτηση έχει παραμέτρους εισόδου αλλά δεν έχει return

```
def diafora(x, y):  
    diaf = x-y  
    print('Διαφορά = ', diaf)
```

```
diafora(4, 2)
```

- Η συνάρτηση ολοκληρώνει μια ενέργεια χωρίς να επιστρέφει κάτι στον κώδικα που την καλεί

def (4) ΜΕ ΠΑΡΑΜΕΤΡΟΥΣ / ΜΕ RETURN

- Η συνάρτηση έχει παραμέτρους εισόδου και επιστρέφει τιμές μέσω της return

```
def dynamh(x,y):  
    return x**y
```

```
dyn = dynamh(2,4)  
print(dyn)
```

def (5) ΜΕ ΠΑΡΑΜΕΤΡΟΥΣ ΠΡΟΚΑΘΟΡΙΣΜΕΝΗΣ ΤΙΜΗΣ

- Η συνάρτηση έχει παραμέτρους εισόδου στις οποίες δίνονται προκαθορισμένες τιμές

```
def dynamh(x, y=2) :  
    return x**y
```

```
dyn = dynamh(2)  
print(dyn)
```

```
dyn = dynamh(2, 3)  
print(dyn)
```

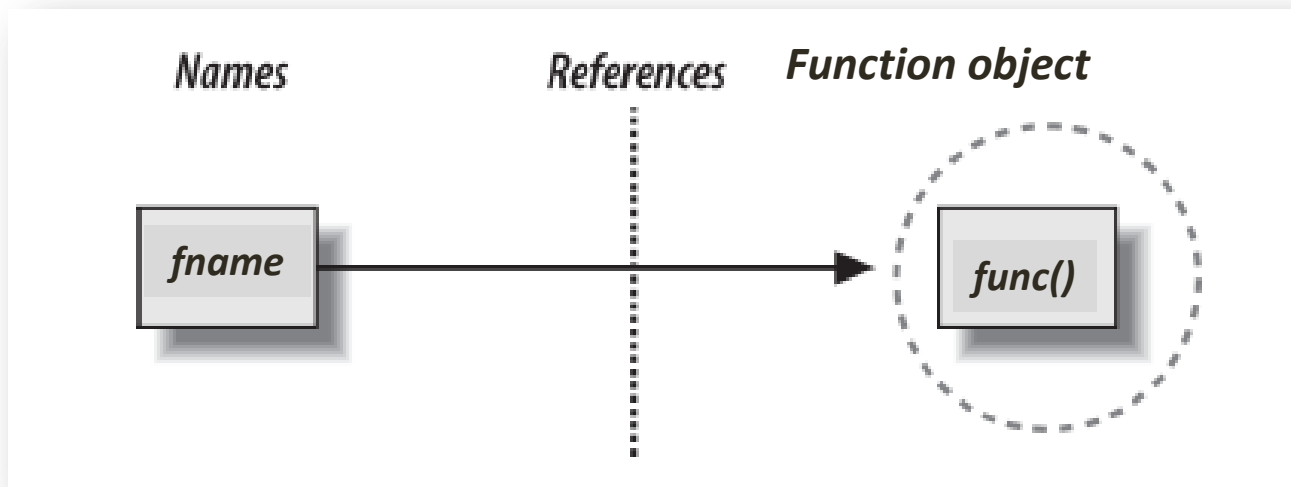
- Κατά την κλήση μπορεί να μην δοθεί όρισμα στην παράμετρο προκαθορισμένης τιμής
- Αν όμως δοθεί τότε υπερισχύει (override) της προκαθορισμένης τιμής
- Τί θα εμφανίσουν οι print σε κάθε περίπτωση;

Πώς **εκτελεί** η Python μια συνάρτηση



Πώς εκτελεί η Python μια συνάρτηση; - 1

- Η δήλωση **def** είναι μια πραγματική **εκτελέσιμη** εντολή – όταν εκτελείται δημιουργεί ένα **νέο αντικείμενο της συνάρτησης** και το συνδέει με ένα όνομα (όπως ισχύει και για τα ονόματα μεταβλητών)



Πηγή: Lutz, M. (2013). *Learning Python, 5th ed.*, O'Reilly: Cambridge

- Άρα σε μεγάλο βαθμό η δήλωση `def` **μοιάζει με την εντολή ανάθεσης** (assignment, =) απλά **αναθέτει/συνδέει ένα όνομα με το αντικείμενο συνάρτησης** κατά την εκτέλεση

Πώς εκτελεί η Python μια συνάρτηση; - 2

- Για να καταλάβουμε τι συμβαίνει όταν δημιουργείται μια συνάρτηση ας κάνουμε τα παρακάτω στο κέλυφος:

```
>>> def f1(x,y):  
    return x+y  
  
>>>  
>>> type(f1)  
<class 'function'>  
>>>  
>>> dir(f1)  
['__annotations__', '__call__', '__code__', '__defaults__', '__delattr__', '__doc__', '__eq__', '__format__', '__getattr__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

- Ορίζουμε τη συνάρτηση
- Δηλ. δημιουργούμε ένα αντικείμενο τύπου 'function'
- Το αντικείμενο f1 κληρονομεί όλες τις ιδιότητες και μεθόδους μιας συνάρτησης

```
>>> f1.__name__  
'f1'  
>>>  
>>> f1.__class__  
<class 'function'>
```

- Εμφανίζουμε ορισμένες ιδιότητες της f1
- `__name__` : το όνομα της f1
- `__class__` : ο τύπος της f1

Συνέπειες: (α) Μια συνάρτηση μπορεί να ορίζεται οπουδήποτε

ΠΑΡΑΔΕΙΓΜΑ

- **Η εντολή def μπορεί να εμφανίζεται οπουδήποτε** μπορεί να εμφανιστεί και μια άλλη εντολή
- (δηλ. όχι μόνον στην αρχή του προγράμματος)

```
if test:
    def func(): # Η συνάρτηση ορίζεται με μορφή A
    ...
else:
    def func(): # Η συνάρτηση ορίζεται με μορφή B
    ...
    ...

# εκτελείται διαφορετική μορφή της συνάρτησης ανάλογα πώς ορίστηκε
func()
```

```
pet = input('Τι ζώακι έχεις; Σκύλο(σ)/Γάτα(γ) ')
if pet=='σ':
    def pet_name():
        return input('Πώς λένε το σκύλο σου; ')
elif pet=='γ':
    def pet_name():
        return input('Πώς λένε τη γάτα σου; ')

print(pet_name())
```

Συνέπειες: (β) Μια συνάρτηση μπορεί να συνδέεται με διάφορους αναγνωριστές

- Το αντικείμενο-συνάρτηση μπορεί να συνδέεται με διάφορους αναγνωριστές (δηλ. να αλλάζει όνομα) κατά την εκτέλεση

ΠΑΡΑΔΕΙΓΜΑ

```
def dynamh(x, y=2) :  
    return x**y  
  
power = dynamh  
print(power(2))
```

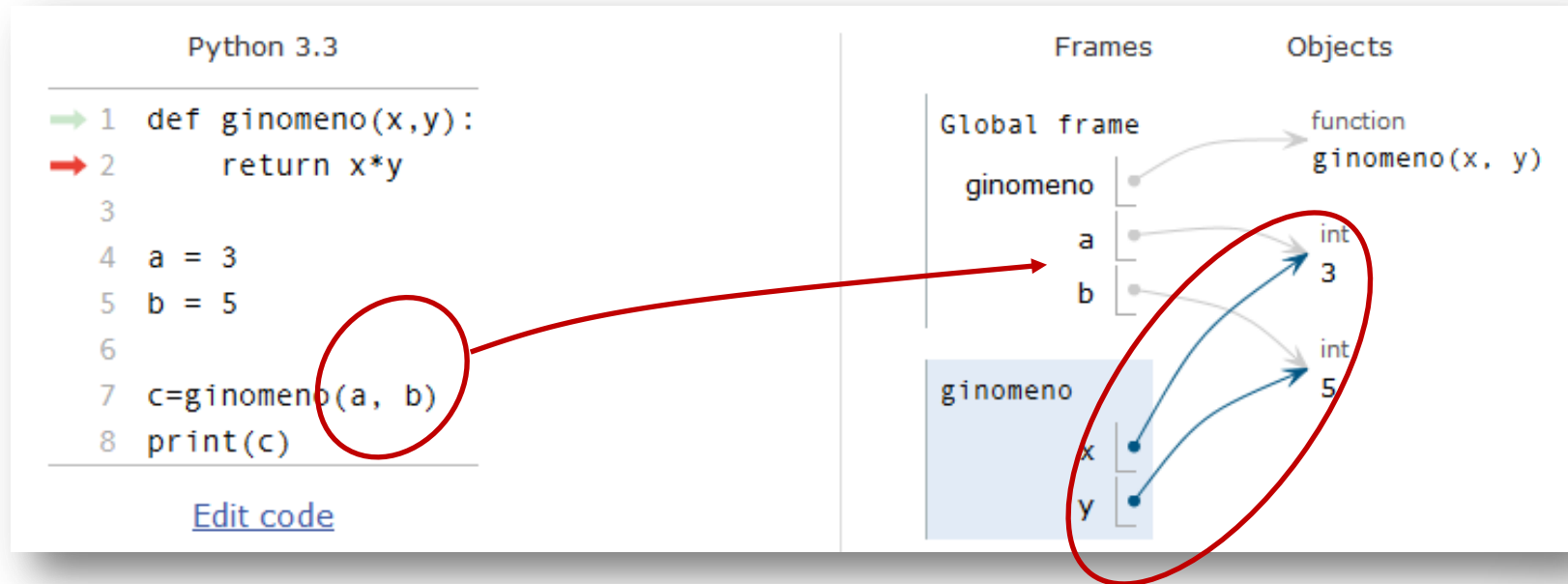
Μετά την εκτέλεση της εντολής
 power = dynamh
η συνάρτηση μπορεί να κληθεί
και με το όνομα 'power'

Πέρασμα ορισμάτων



Πέρασμα ορισμάτων σε συνάρτηση --1

- Το πέρασμα ορισμάτων γίνεται αρχικά με **«αναφορά αντικειμένου»** (object reference)
- Δηλ. τα ονόματα παραμέτρων μέσα στη συνάρτηση **αναφέρονται** στα αντικείμενα-τιμές που στέλνονται από το κύριο πρόγραμμα ως ορίσματα



- *Παράδειγμα:* οι παράμετροι x , y της συνάρτησης «αναφέρονται» στα αντικείμενα 3 και 5 που είναι συνδεδεμένα με τα ονόματα a , b που περνούν ως ορίσματα

- Όμως στην συνέχεια γίνεται κάτι διαφορετικό ανάλογα αν τα ορίσματα είναι αντικείμενα **μεταλλάξιμα ή όχι**
- **(α) Μη-μεταλλάξιμα** αντικείμενα:
 - (αμετάλλακτα, immutable, πχ. ακέραιοι, αλφαριθμητικά)
 - Αν μέσα στη συνάρτηση πάρουν τιμή με εντολή ανάθεσης τότε **δημιουργείται αντίγραφο τοπικά στη συνάρτηση**
- **(β) Μεταλλάξιμα** αντικείμενα:
 - (mutable, πχ. λίστες)
 - Περνούν πάντοτε στη συνάρτηση χωρίς δημιουργία αντιγράφου, ακόμα κι αν αλλάξει τιμή τους με εντολή ανάθεσης
 - Δηλ. κάθε μεταβολή τους τοπικά στη συνάρτηση «καθρεφτίζεται» αμέσως και στην αντίστοιχη δομή στο κύριο πρόγραμμα

Πέρασμα ορισμάτων -- Παράδειγμα

- Πριν να εκτελέσετε τον κώδικα κάντε μια πρόβλεψη τι θα εμφανίσει η print
- Στη συνέχεια τρέξτε τον κώδικα και δείτε το αποτέλεσμα
- Ήταν σωστή η πρόβλεψή σας, ναι ή όχι και γιατί;
- Δείτε τις εξηγήσεις στις επόμενες διαφάνειες και σκεφθείτε αν αυτή είναι και η δική σας κατανόηση

```
def myfunc(a, b):  
    a = 4  
    b[0] = 100  
  
X = 1  
L = [10, 20]  
  
myfunc(X, L)  
  
print(X, L)
```

Πέρασμα ορισμάτων: Παράδειγμα

- 1α

```
def myfunc(a, b):  
    a = 4  
    b[0] = 100
```

`X = 1`

`L = [10, 20]`

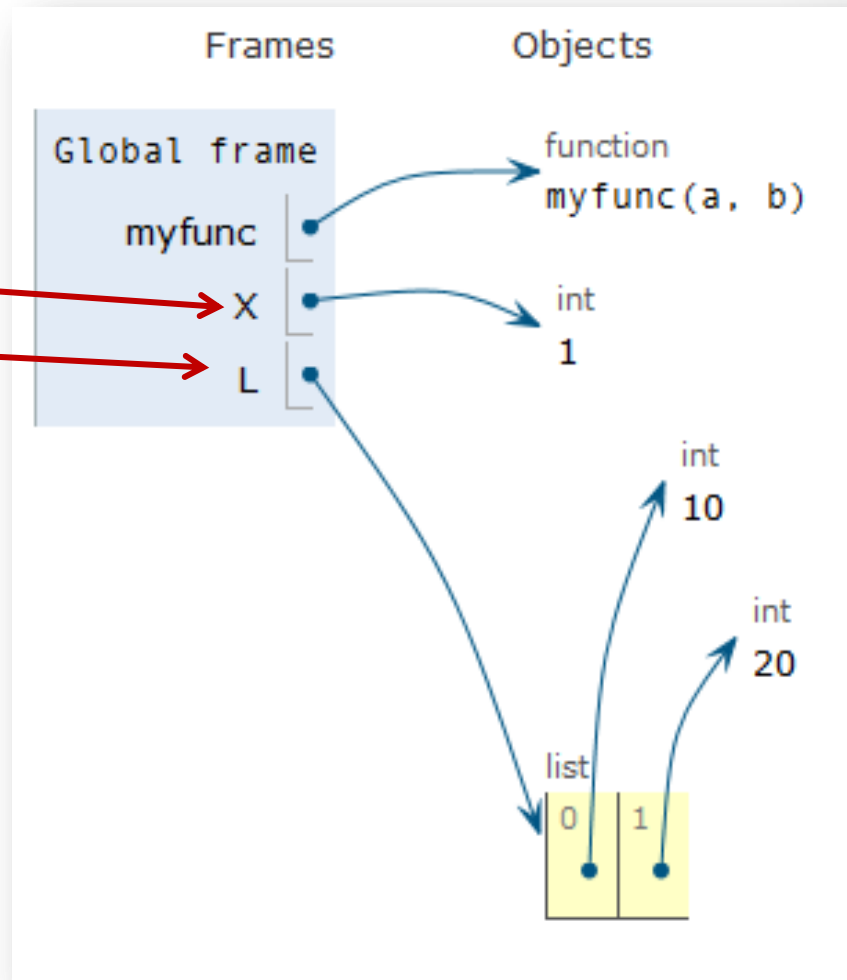
`myfunc(X, L)`

`print(X, L)`

ΜΗ-ΜΕΤΑΛΛΑΞΙΜΟ

ΜΕΤΑΛΛΑΞΙΜΟ

- (1) **Πριν** την κλήση



Πέρασμα ορισμάτων: Παράδειγμα

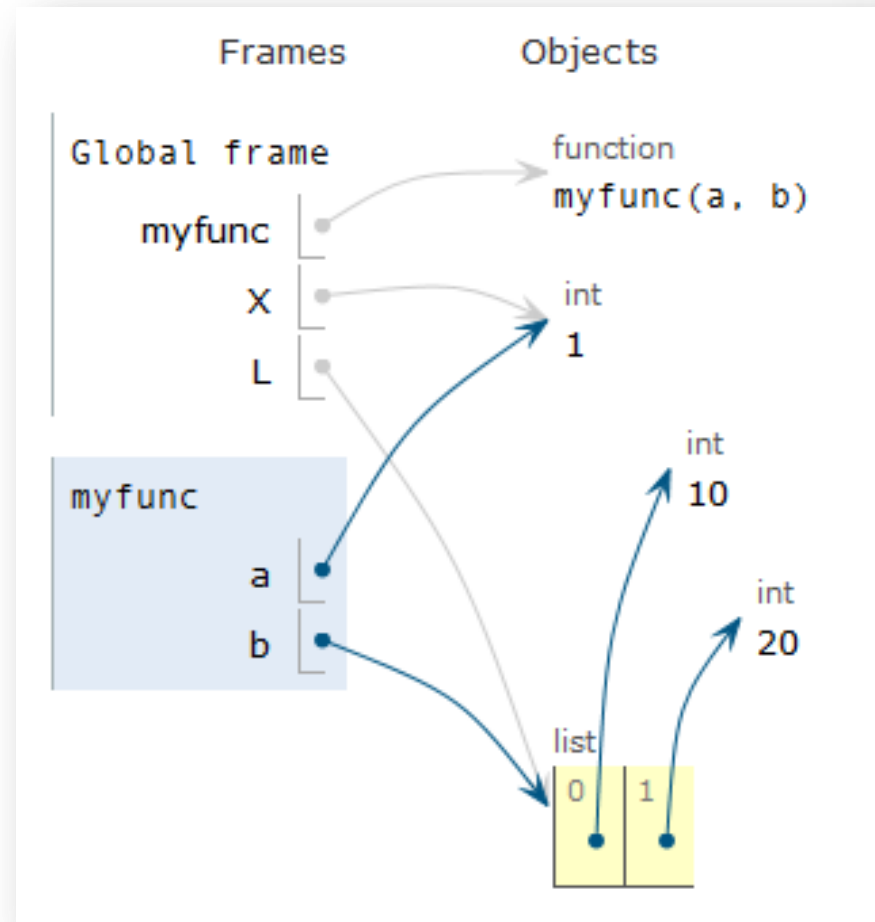
- 1β

```
def myfunc(a, b):  
    a = 4  
    b[0] = 100
```

```
X = 1  
L = [10, 20]
```

```
myfunc(X, L)
```

```
print(X, L)
```

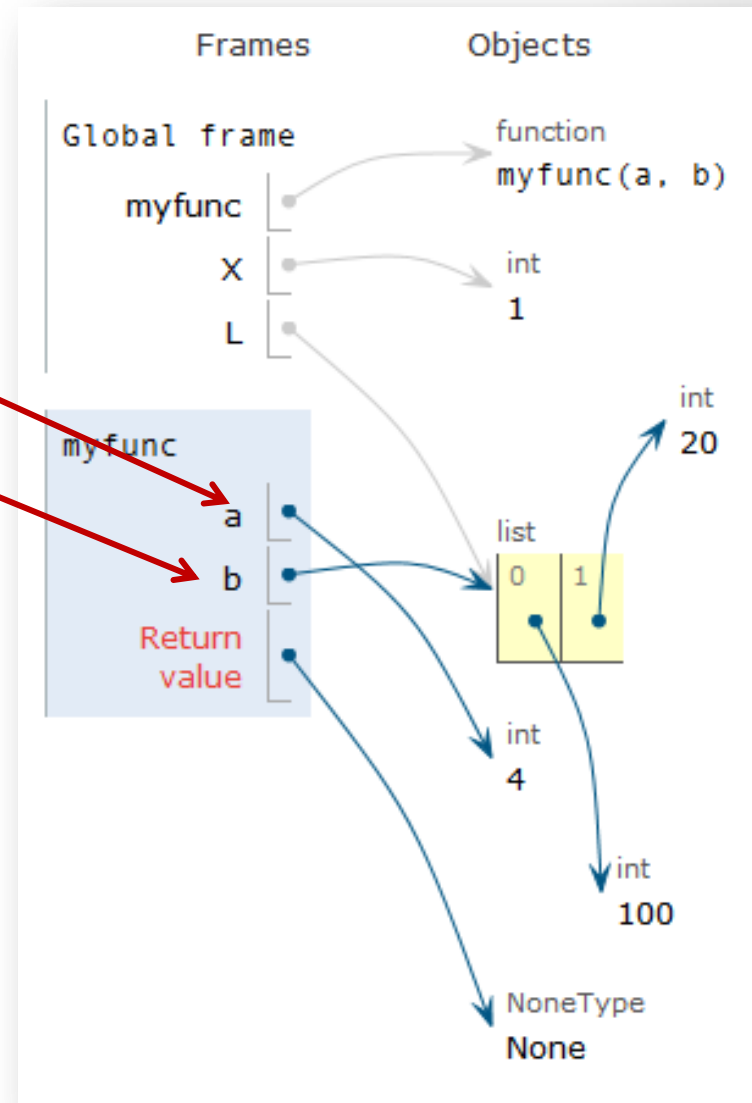


- (2) **Κατά** την κλήση:
Πέρασμα ορισμάτων

Πέρασμα ορισμάτων: Παράδειγμα

- 1γ

```
def myfunc(a, b):  
    a = 4  
    b[0] = 100  
  
X = 1  
L = [10, 20]  
  
myfunc(X, L)  
  
print(X, L)
```



- (3) **Εκτέλεση** της συνάρτησης
- Για την ακέραια `a` (immutable) έχει δημιουργηθεί νέο αντικείμενο '4'
- Για τη λίστα `b` (mutable) **ΔΕΝ** δημιουργήθηκε νέο αντικείμενο: άρα **ΚΑΘΕ** αλλαγή στη λίστα `b` αντανακλά και στη λίστα `L` καθολικής εμβέλειας



Πέρασμα ορισμάτων: Παράδειγμα

- 1δ

```
def myfunc(a, b):  
    a = 4  
    b[0] = 100
```

```
X = 1  
L = [10, 20]  
  
myfunc(X, L)  
  
print(X, L)
```

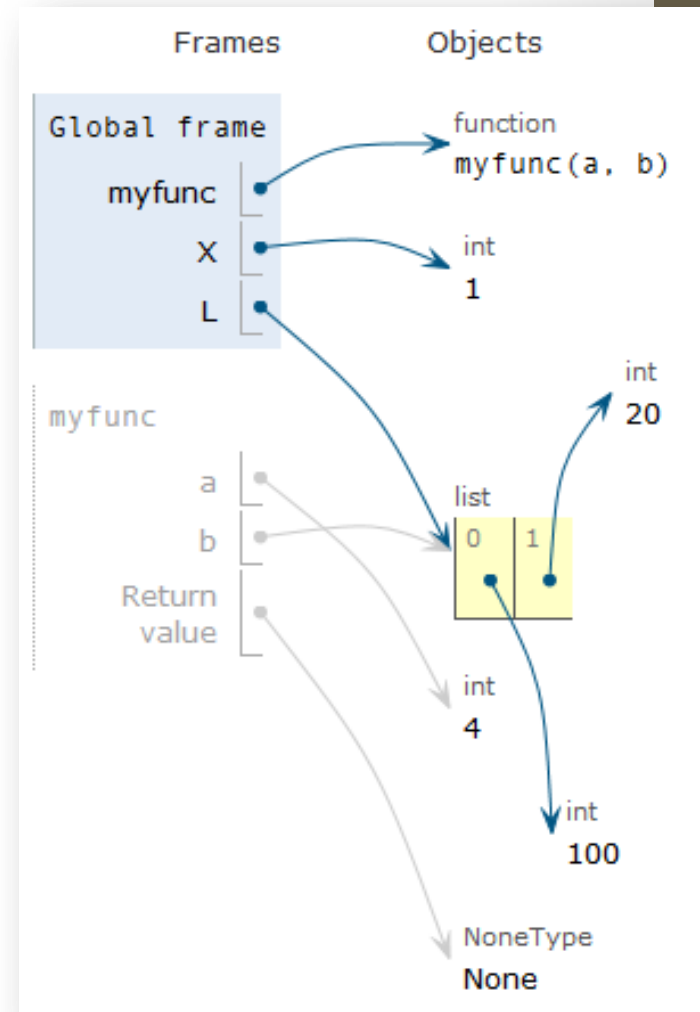
- (5) Η έξοδος του προγράμματος

Program output:

```
1 [100, 20]
```

- X: δεν έχει αλλάξει τιμή
- L: είναι διαφορετική

- (4) Μετά την κλήση



Γενικός κανόνας για το πέρασμα **ορισμάτων**


- Τα ορίσματα περνούν σε μια συνάρτηση με «**αναφορά αντικειμένου**»
- Όμως αν μέσα στη συνάρτηση αλλάξουν τιμή με εντολή ανάθεσης, τότε:
 - (α) αν είναι **αμετάλλακτα** → δημιουργείται **νέο** αντικείμενο **τοπικά** στη συνάρτηση
 - (β) αν είναι **μεταλλάξιμα** → **δεν** δημιουργείται νέο αντικείμενο στη συνάρτηση, αλλά κάθε αλλαγή στη συνάρτηση επηρεάζει και το αντίστοιχο δεδομένο στο κύριο πρόγραμμα

Αν θέλω το **μεταλλάξιμο** δεδομένο που περνώ ως όρισμα να είναι **τοπικό**;

- **Λύση (Α)**: περάστε ένα αντίγραφο του μεταλλάξιμου δεδομένου ως όρισμα
- Πχ. Περνώντας μια λίστα L αντί για την κλήση `myfunc(X, L)` γράψτε **`myfunc(X, L[:])`**
- **Λύση (Β)**: δημιουργήστε ένα αντίγραφο του μεταλλάξιμου δεδομένου μέσα στη συνάρτηση ώστε να μην επηρεάζεται το αρχικό αντικείμενο
- Πχ. αντί

```
def myfunc(a, b):  
    a = 4  
    b[0] = 100
```

γράψτε:



```
def myfunc(a, b):  
    b = b[:]  
    a = 4  
    b[0] = 100
```

Εμβέλεια μεταβλητών

Variable Scope

Εμβέλεια μεταβλητών: **Τοπική** (local)

- **Τοπική** μεταβλητή (**local**): Κάθε μεταβλητή (όνομα) που **δημιουργείται*** μέσα σε μια συνάρτηση είναι **τοπική** (local) στη συνάρτηση αυτή
 - Λέμε: «Έχει τοπική εμβέλεια (local scope)» ή «Ανήκει στον τοπικό χώρο ονομάτων (local namespace) της συνάρτησης»
 - * Μια μεταβλητή **δημιουργείται με μια εντολή ανάθεσης** πχ. `spam = 0`, `L = []` κλπ.

ΠΑΡΑΔΕΙΓΜΑ

```
def intersect(seq1, seq2):  
    res = []  
    for x in seq1:  
        if x in seq2:  
            res.append(x)  
    print(res)
```

```
s1 = 'SPAM'  
s2 = 'SCAM'  
intersect(s1, s2)
```

- Οι `res` και `x` είναι **τοπικές** (local) στη συνάρτηση `Intersect`

Εμβέλεια μεταβλητών: **Καθολική** (global)

- **Καθολική** μεταβλητή (**Global**): Μια μεταβλητή που δημιουργείται σε επίπεδο κύριου προγράμματος (δηλ. έξω από όλες τις συναρτήσεις) είναι **καθολική (global)** στο πρόγραμμα
 - Λέμε: «Έχει καθολική εμβέλεια» ή «Ανήκει στον καθολικό χώρο ονομάτων (global namespace) του κύριου προγράμματος»

ΠΑΡΑΔΕΙΓΜΑ

```
def intersect(seq1, seq2):  
    res = []  
    for x in seq1:  
        if x in seq2:  
            res.append(x)  
    print(res)  
  
s1 = 'SPAM'  
s2 = 'SCAM'  
intersect(s1, s2)
```

- Οι s1 και s2 είναι **καθολικές (global)** στο πρόγραμμα

Εμβέλεια μεταβλητών : η δήλωση **nonlocal**

- **Μη τοπική (nonlocal)**: Μια μεταβλητή που δηλώνεται **nonlocal** μέσα σε μια φωλιασμένη συνάρτηση ανήκει στον χώρο ονομάτων της **αμέσως ανώτερης, δηλ. περικλείουσας, συνάρτησης**
- Οι **first & last** είναι τοπικές στη φωλιασμένη `nested` γιατί δημιουργούνται εκεί
- ΑΛΛΑ: αν δηλωθούν **nonlocal** είναι **μη-τοπικές** δηλ. ανήκουν στην περικλείουσα συνάρτηση `intersect`. Έτσι θα ταυτίζονται με τις `first, last` της `intersect`

```
def intersect(seq1, seq2):  
  
    def nested():  
        #nonlocal first, last  
        first = res[0]  
        last = res[len(res)-1]  
        return  
  
    first = last = '_'  
    res = []  
    for x in seq1:  
        if x in seq2:  
            res.append(x)  
    print(res)  
    nested()  
    print(first, last)  
  
s1 = 'SPAM'  
s2 = 'SCAM'  
intersect(s1, s2)
```

Εμβέλεια μεταβλητών : Πόσα επίπεδα;

- Ο κανόνας **LEGB**
- **LEGB** = Local / Enclosing / Global / Built-in
- Η Python αναζητά και αναγνωρίζει ονόματα σε **4 επίπεδα εμβέλειας**:
 - **Local** (Τοπική)
 - Κάθε απλή συνάρτηση
 - **Enclosing** (Περικλείουσα)
 - Κάθε συνάρτηση που περικλείει φωλιασμένες συναρτήσεις
 - **Global** (Καθολική)
 - Κύριο πρόγραμμα
 - **Built-in** (Ενσωματωμένη)
 - Τα standard ονόματα της γλώσσας / Άλλα πακέτα / Βιβλιοθήκες, κλπ.

Εμβέλεια μεταβλητών Γενικός κανόνας **LEGB**

Built-in (Python)

Names preassigned in the built-in names module: `open`, `range`, `SyntaxError`....

Global (module)

Names assigned at the top-level of a module file, or declared `global` in a `def` within the file.

Enclosing function locals

Names in the local scope of any and all enclosing functions (`def` or `lambda`), from inner to outer.

Local (function)

Names assigned in any way within a function (`def` or `lambda`), and not declared `global` in that function.

Πηγή: Lutz, M. (2013). *Learning Python*, 5th ed., O'Reilly: Cambridge

- Γενικός κανόνας: κάθε μεταβλητή είναι **local** στο επίπεδο της συνάρτησης όπου **δημιουργείται**, εκτός εάν εμφανίζεται σε δηλώσεις **global** ή **nonlocal**

Αναζήτηση ονομάτων στα επίπεδα LEGB -1

- Τι συμβαίνει αν μια μεταβλητή απλά **χρησιμοποιείται** σε μια συνάρτηση αλλά δεν δημιουργείται σ' αυτή;
- (α) **Αν απλά χρησιμοποιείται** χωρίς να εμφανίζεται σε εντολή ανάθεσης τότε ο διερμηνέας αναζητά τις μεταβλητές αυτές σε υψηλότερο επίπεδο στην ιεραρχία LEGB

```
def check(x,y):  
    if low < x+y < upper:  
        return True  
    else:  
        return False  
  
low = 1  
upper = 10  
print(check(5, 7))
```

- Στο παράδειγμα οι **low** & **upper** είναι **καθολικές** μεταβλητές που χρησιμοποιούνται σωστά μέσα στην check
- Ο διερμηνέας αναζητά τις low & upper σε ανώτερο επίπεδο (global) και εφόσον τις βρίσκει τις χρησιμοποιεί μέσα στη συνάρτηση check()

Αναζήτηση ονομάτων στα επίπεδα LEGB -2

- (β) **Αν εμφανίζεται σε εντολή ανάθεσης** τότε ο διερμηνέας κάνει κάτι διαφορετικό ανάλογα αν το όνομα συνδέεται με αντικείμενο **μεταλλάξιμου** ή **αμετάλλακτου** τύπου
- Στο παράδειγμα θα υπάρξει πρόβλημα με τη μεταβλητή `suma` (αμετάλλακτος ακέραιος) εφόσον δεν δηλωθεί `global`
- Όμως για την `res` δεν υπάρχει πρόβλημα (μεταλλάξιμη λίστα). Ο διερμηνέας την αναγνωρίζει ως μεταβλητή υψηλότερου (`global`) επιπέδου

```
def intersect (seq1, seq2):  
    #global suma  
    for x in seq1:  
        if x in seq2:  
            suma += 1  
            res.append(x)  
  
res=[]  
suma=0  
s1 = 'SPAM'  
s2 = 'SCAM'  
intersect(s1, s2)  
print(res, suma)
```

Εμβέλεια

Παραδείγματα - 1

```
X = 99

def func(Y):
    Z = X + Y
    return Z

print(func(1))
```

- Στα παραδείγματα σκεφθείτε:
- (α) ποιες μεταβλητές είναι τοπικές και ποιες καθολικές
- (β) τι θα τυπώσει κάθε φορά η print και γιατί
 - X: Global
 - Z, Y: Locals
 - Τυπώνει 100

```
X = 88

def func():
    X = 99

func()
print(X)
```

- Global
- Local

- Τυπώνει 88 (global)

Εμβέλεια

Παραδείγματα - 2

```
X = 88
```

```
def func():  
    global X  
    X = 99
```

```
func()  
print(X)
```

• Η X δηλώνεται και παραμένει **global** και μέσα στη συνάρτηση

- X: Global
- Τυπώνει 99

```
y, z = 1, 2
```

```
def globvar():  
    global x  
    x = y + z
```

- Όλες οι μεταβλητές είναι **Global**
- Δεν χρειάζεται να δηλωθούν οι y & z
- Αναγνωρίζονται ως global λόγω του κανόνα LEGB

```
X = 99
```

```
def func1():  
    global X  
    X = 88
```

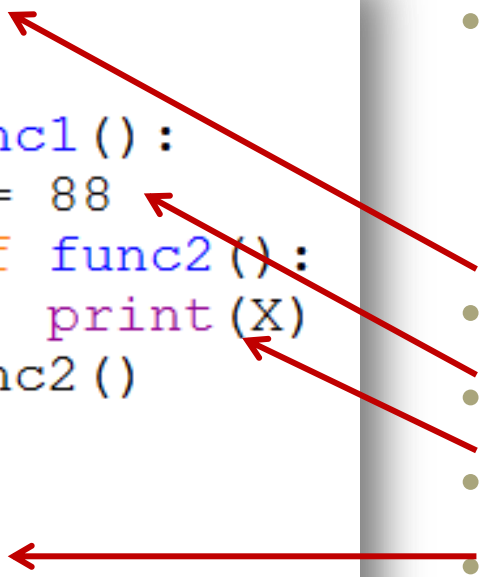
```
def func2():  
    global X  
    X = 77
```

- Η X δηλώνεται global σε **περισσότερες** συναρτήσεις
- Το ποια τιμή θα έχει κάποια στιγμή η X εξαρτάται από το πώς θα κληθούν οι συναρτήσεις

```
X = 99

def func1():
    X = 88
    def func2():
        print(X)
    func2()

func1()
```



- Όταν μια μεταβλητή αμετάλλακτου τύπου παίρνει τιμή μέσα σε συνάρτηση **αυτόματα γίνεται τοπική** στη συνάρτηση
- Global
- Local στη func1()
- Non-local για τη func2()
- Τυπώνει 88

```
def checkPass(password):  
    has_upper = False  
    has_lower = False  
    has_num = False  
    state = 'Απορρίπτεται'  
  
    def checkUpper(x):  
        nonlocal has_upper  
        if x>='A' and x<='Z':  
            has_upper = True  
  
    def checkLower(x):  
        nonlocal has_lower  
        if x>='a' and x<='z':  
            has_lower = True  
  
    def checkNum(x):  
        nonlocal has_num  
        if x>="0" and x<="9":  
            has_num = True
```

- Η `has_upper` είναι local στον χώρο της συνάρτησης `checkPass`
- If 'A' <= x <= 'Z'
- Στον χώρο της φωλιασμένης `checkUpper` δηλώνεται `nonlocal`
- Άρα αναγνωρίζεται ως η **ίδια μεταβλητή** στον χώρο της περικλείουσας `checkPass`
- Παρόμοια για τις μεταβλητές `has_lower` & `has_num`

Γενικός κανόνας για την αναζήτηση ονομάτων στην ιεραρχία **LEGB**

- Κάθε μεταβλητή είναι **local** στο επίπεδο της συνάρτησης όπου **δημιουργείται**, εκτός εάν εμφανίζεται σε δηλώσεις **global** ή **nonlocal**
- Αν μια μεταβλητή **δεν δημιουργείται μέσα στη συνάρτηση** η Python την αναζητά σε ανώτερα επίπεδα LEGB και εφόσον την εντοπίσει ο κώδικας λειτουργεί σωστά
- Όμως αν μέσα στη συνάρτηση μια μεταβλητή **αλλάξει τιμή με εντολή ανάθεσης**, τότε:
 - (α) αν είναι **αμετάλλακτος τύπος** → θεωρείται **τοπική** και δημιουργείται **νέο** αντικείμενο τοπικά στη συνάρτηση
 - (β) αν είναι **μεταλλάξιμος τύπος** → **δεν θεωρείται τοπική** αλλά κάθε αλλαγή στη συνάρτηση επηρεάζει και το αντίστοιχο δεδομένο στο κύριο πρόγραμμα

Συναρτήσεις ‘Γεννήτορες’

Generators



Generator: Τι είναι;

- Στην Python μια συνάρτηση **generator** είναι μια συνάρτηση η οποία εκτελείται **σταδιακά** (σε διαδοχικές φάσεις) **επιστρέφοντας μια τιμή σε κάθε επανάληψη** εκτέλεσής της
- Με τον τρόπο αυτό: η συνάρτηση δεν δεσμεύει μεγάλα τμήματα μνήμης και επιστρέφει μια σειρά τιμών
- Δηλ. μπορεί να χρησιμοποιηθεί ως **απαριθμήσιμη δομή (iterator)** σε μια εντολή for
- Μια συνάρτηση generator χρησιμοποιεί **αντί για return** την εντολή **yield** : Όταν εκτελείται η **yield** η συνάρτηση **επιστρέφει τιμή** στον κύριο κώδικα αλλά **παραμένει στη μνήμη** ώστε να εκτελεστεί και πάλι στην επόμενη επανάληψη

Generator:

- Η `numbers()` δεσμεύει ένα πολύ μεγάλο τμήμα μνήμης για να δημιουργήσει και επιστρέψει τη λίστα `numlist`
- Αντίθετα μια συνάρτηση generator εκτελείται σταδιακά και επιστρέφει την τιμή της `num` κάθε φορά που εκτελείται η **`yield num`**

Παράδειγμα

```
# Δημιουργία Λίστας:  
# Μεγάλη δέσμευση μνήμης  
  
def numbers(n):  
    num, numlist = 0, []  
    while num < n:  
        numlist.append(num)  
        num += 1  
    return numlist  
  
sumall = sum(numbers(1000000))  
print(sumall)
```

```
# Μια συνάρτηση γεννήτορας  
# επιστρέφει τιμές διαδοχικά  
# και δεν δεσμεύει τη μνήμη
```

```
def numbers(n):  
    num = 0  
    while num < n:  
        yield num  
        num += 1  
  
sumall = sum(numbers(1000000))  
print(sumall)
```

Generator:

Ο ρόλος της **yield**

- Για να δημιουργήσετε ένα generator πρέπει σε κάποιο σημείο της συνάρτησης να υπάρχει η εντολή **yield** (συνοδευόμενη από μία ή περισσότερες τιμές)
- Όταν εκτελείται η **yield** ο γεννήτορας επιστρέφει την ή τις τιμή/-ές στο κύριο πρόγραμμα ενώ η συνάρτηση παραμένει στη μνήμη ώστε να εκτελεστεί την επόμενη φορά (επανάληψη, iteration)

```
# Μια συνάρτηση γεννήτορας  
# επιστρέφει τιμές διαδοχικά  
# και δεν δεσμεύει τη μνήμη  
  
def numbers(n):  
    num = 0  
    while num < n:  
        yield num  
        num += 1  
  
sumall = sum(numbers(1000000))  
print(sumall)
```

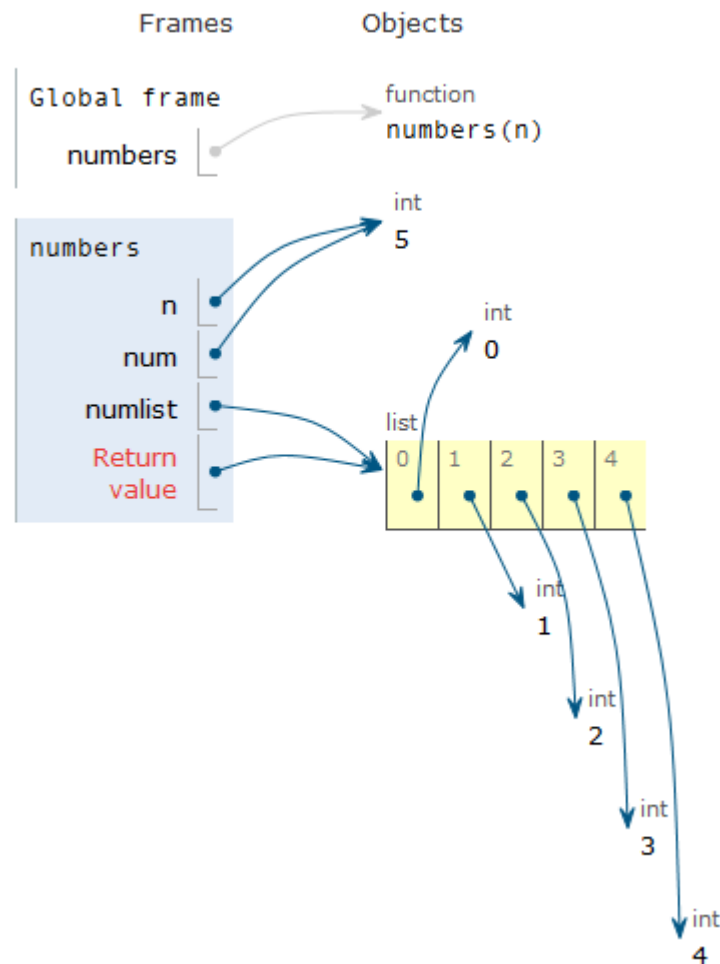
Generator:

Σύγκριση εκτέλεσης

1/2

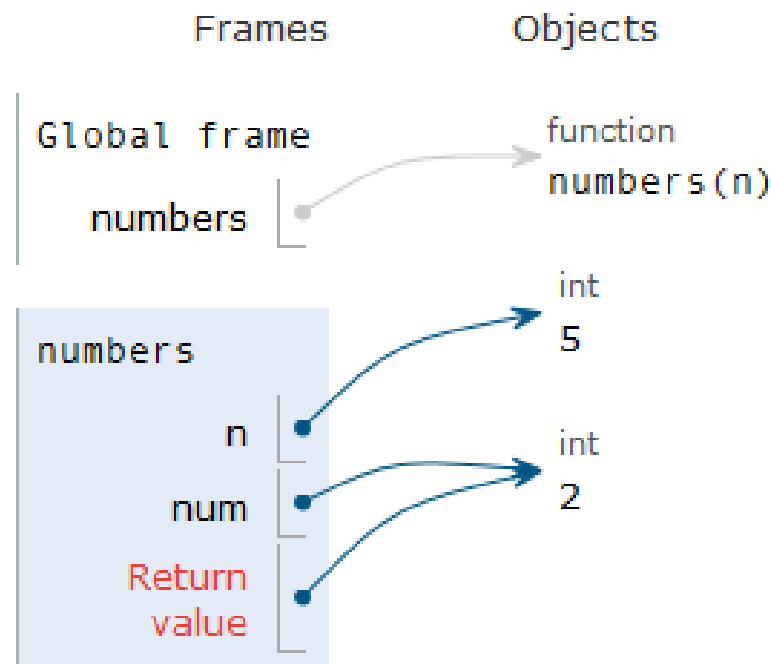
```
# Δημιουργία Λίστας:  
# Μεγάλη δέσμευση μνήμης  
  
def numbers(n):  
    num, numlist = 0, []  
    while num < n:  
        numlist.append(num)  
        num += 1  
    return numlist  
  
sumall = sum(numbers(1000000))  
print(sumall)
```

- Η κλασική συνάρτηση (με return) δημιουργεί τη λίστα στη μνήμη



```
# Μια συνάρτηση γεννήτορας  
# επιστρέφει τιμές διαδοχικά  
# και δεν δεσμεύει τη μνήμη  
  
def numbers(n):  
    num = 0  
    while num < n:  
        yield num  
        num += 1  
  
sumall = sum(numbers(1000000))  
print(sumall)
```

- Η συνάρτηση γεννήτορας (με `yield`) επιστρέφει διαδοχικές τιμές σε κάθε επανάληψη εκτέλεσης



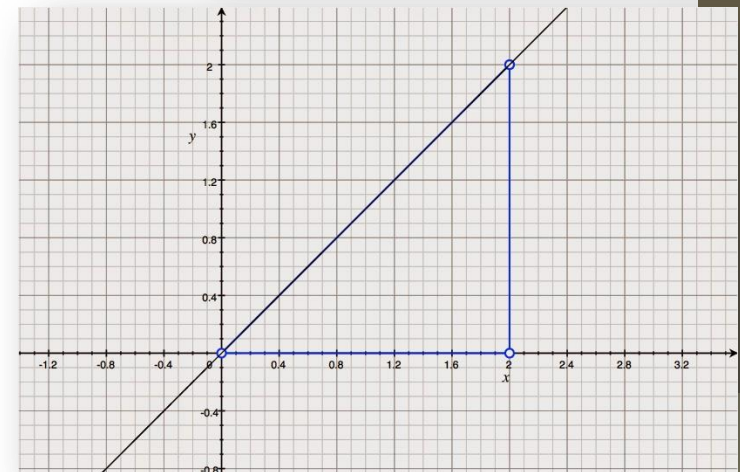
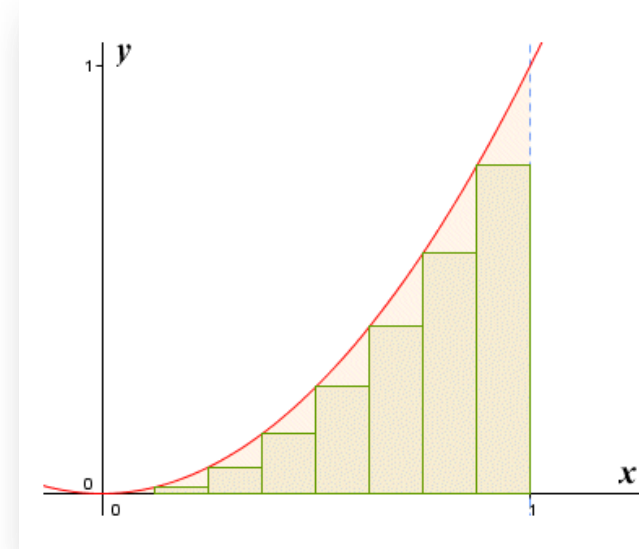
Generator: Παράδειγμα drange()

- Η συνάρτηση `drange()` είναι ένας γεννήτορας που μπορεί να χρησιμοποιηθεί σε βρόχο επανάληψης `for` με δεκαδικό βήμα

```
def drange(start, stop, step):  
    r = start  
    while r < stop:  
        yield r  
        r += step  
  
for k in drange(0, 1, 0.1):  
    print(round(k, 2))
```

Generator: Άσκηση «Υπολογισμός Ολοκληρώματος»

- Γράψτε πρόγραμμα που να υπολογίζει την τιμή του ορισμένου ολοκληρώματος της συνάρτησης $f(x) = x$ στο διάστημα $[0,1]$
- Υπόδειξη 1: το ορισμένο ολοκλήρωμα μιας συνάρτησης υπολογίζεται ως το εμβαδόν που περικλείεται από τη συνάρτηση και τον άξονα x για το συγκεκριμένο διάστημα $[x_A, x_T]$ στο οποίο γίνεται ο υπολογισμός
- Υπόδειξη 2: Χρησιμοποιήστε τον γεννήτορα `drange()` για να γράψετε το βρόχο στον οποίο γίνεται ο υπολογισμός του εμβαδού σταδιακά (με βάση το βήμα που θα δηλώσετε)
- Υπόδειξη 3: Γράψτε τη συνάρτηση $f(x)$ ως κλασική συνάρτηση Python στον κώδικά σας



Συναρτησιακός προγραμματισμός

Παράδειγμα



Συναρτησιακός προγραμματισμός

- Στην Python οι συναρτήσεις χαρακτηρίζονται ως **«πρώτης τάξης αντικείμενα»** (first class objects) κάτι που στην πράξη σημαίνει πως ότι μπορούν να χρησιμοποιηθούν όπου και όπως χρησιμοποιούνται και οι υπόλοιποι τύποι δεδομένων
- Αυτό ανοίγει ισχυρές δυνατότητες **συναρτησιακού προγραμματισμού** (functional programming), ενός τρόπου προγραμματισμού που βασίζεται σε υπολογισμούς μέσω συναρτήσεων
- Στη συνέχεια δίνεται ένα απλό παράδειγμα για το πώς οι συναρτήσεις μπορούν να περάσουν ως ορίσματα σε άλλη συνάρτηση

Πέρασμα συναρτήσεων ως ορισμάτων συνάρτησης

```
def f2(x):  
    return x**2  
  
def f3(x):  
    return x**3  
  
def f4(x):  
    return x**4  
  
def fcall(fpass, x):  
    return fpass(x)  
  
# 1)  
x=10  
y = fcall(f2, x)  
print(y)  
  
# 2)  
  
Ldef = [f2, f3, f4]  
for i, ideo in enumerate(Ldef):  
    print(fcall(ideo, i))
```

- Οι συναρτήσεις f2, f3, f4 υπολογίζουν αντίστοιχα δυνάμεις
- Η fcall δέχεται ως όρισμα μια συνάρτηση fpass και μια αριθμητική τιμή x
- Στο πρώτο παράδειγμα η fcall καλείται με όρισμα την f2 ώστε να υπολογίσει το τετράγωνο του x
- Στο δεύτερο παράδειγμα οι συναρτήσεις εμφανίζονται ως στοιχεία της λίστας Ldef
- Στη συνέχεια ο βρόχος περνά σταδιακά όλες τις συναρτήσεις-στοιχεία της Ldef ως ορίσματα στην fcall και υπολογίζονται οι αντίστοιχες δυνάμεις του μετρητή i

Περιεχόμενα

- Συνάρτηση: Τι είναι και πώς δηλώνεται
- Σύνταξη συναρτήσεων
- Πώς εκτελεί η Python μια συνάρτηση
- Εμβέλεια μεταβλητών
- Πέρασμα Ορισμάτων
- Συναρτήσεις “Γεννήτορες” (Generators)
- Συναρτησιακός προγραμματισμός