Ελληνικό Μεσογειακό Πανεπιστήμιο
Σχολή Μηχανικών / Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

# HMY01K06
# Επιστημονικός Προγραμματισμός με Python

## Διάλεξη Έβδομη
## Συναρτήσεις , Δομοστοιχεία, Πακέτα

*Φθινόπωρο 2025*

# Συναρτήσεις (functions)

- Μια *συνάρτηση* είναι ένα *μπλοκ οργανωμένου επαναχρησιμοποιήσιμου κώδικα* που χρησιμοποιείται για να φέρει σε πέρας *μια συγκεκριμένη εργασία*

- Οι συναρτήσεις μας βοηθάνε να *οργανώνουμε* και να *διαχειριζόμαστε* καλύτερα τα *μεγάλα προγράμματα* καθώς ο κώδικας σπάει σε *μικρότερα αυτοτελή κομμάτια* (αρθρώματα / modules)

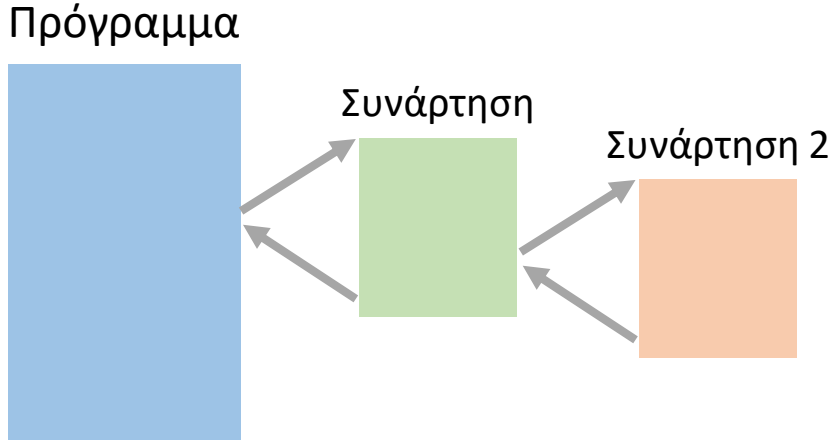
- Κάθε συνάρτηση αποτελεί *μικρογραφία ενός προγράμματος*, καθώς μπορεί να έχει τα δικά της *δεδομένα εισόδου* και να παράγει τα δικά της *αποτελέσματα*

- Κάθε συνάρτηση έχει το δικό της *όνομα* και *τρέχει* μόνο *όταν την καλέσουμε*

- Οι συναρτήσεις ορίζονται *μια φορά*, στην αρχή *(συνήθως)* ενός προγράμματος και καλούνται μέσα από αυτό *όσες φορές χρειαστεί*

# Συναρτήσεις (functions)

- Μια συνάρτηση
  - καλείται μέσα από ένα πρόγραμμα
  - μπορεί να *καλεί* άλλες συναρτήσεις *ή να καλείται* μέσα από άλλες συναρτήσεις *(συνάρτηση μέσα σε συνάρτηση)*
  - μπορεί ακόμα *να καλεί τον ίδιο της τον εαυτό (αναδρομική συνάρτηση)*

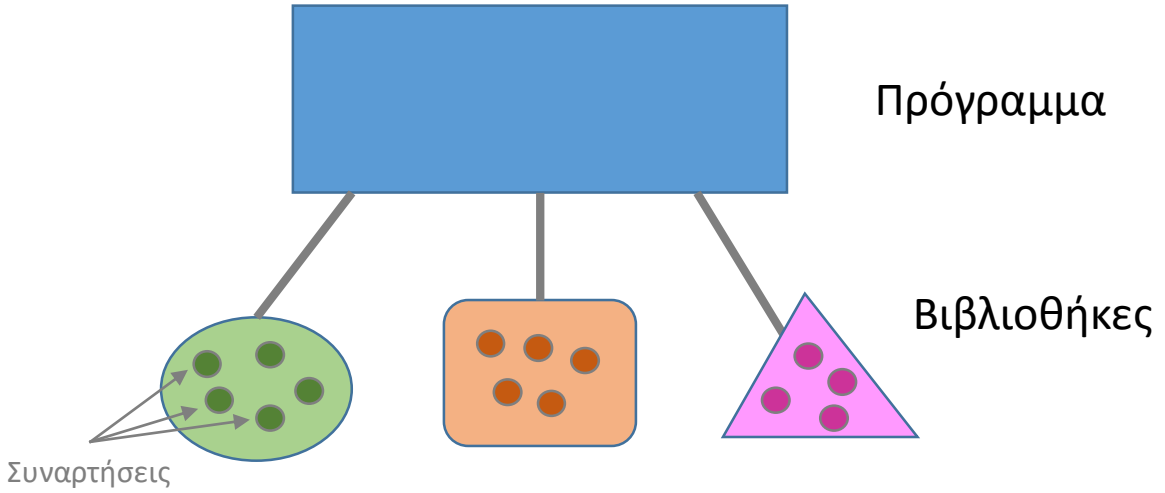



- Η Python *-όπως άλλωστε και οι περισσότερες γλώσσες προγραμματισμού-* παρέχει πολλές *ενσωματωμένες συναρτήσεις (built in functions) όπως πχ.* `print()`,`len()` *κλπ.* αλλά μπορούμε επίσης να δημιουργήσουμε τις δικές μας συναρτήσεις

# Βιβλιοθήκες συναρτήσεων

Συλλογές συναρτήσεων με *συναφή λειτουργικότητα*

- Συναρτήσεις με *παρεμφερή λειτουργικότητα (πχ. τριγωνομετρικές, γραφικές)* συνήθως κωδικοποιούνται μαζικά και αποθηκεύονται σε ξεχωριστές βιβλιοθήκες *(πακέτα συναρτήσεων - modules)* τα οποία εισάγονται σε ένα πρόγραμμα με τις κατάλληλες εντολές




- Στην Python οι βιβλιοθήκες συναρτήσεων ονομάζονται *πακέτα (packages)* και εισάγονται μέσα στο πρόγραμμα με την εντολή `import`

# Ορισμός συναρτήσεων στην Python

- Οι συναρτήσεις είναι *μπλοκ κώδικα* που αρχίζουν με την λέξη κλειδί `def` ακολουθούμενη από το *όνομα της συνάρτησης* και παρενθέσεις

```
def my_function():
    .....
    .....
    .....
```

Το μπλοκ του κώδικα μιας συνάρτησης *γράφεται σε εσοχή ("πιο μέσα")*, όπως ακριβώς γίνεται και με τα `if`, `for` κλπ.

- Μέσα στις παρενθέσεις τοποθετούνται *μηδέν ή περισσότερα* ονόματα χωρισμένα με κόμμα που ονομάζονται *ορίσματα* (arguments) ή αλλιώς *παράμετροι εισόδου* της συνάρτησης
- Κάθε συνάρτηση *καλείται* μέσα από ένα *μπλοκ κώδικα (πρόγραμμα, άλλη συνάρτηση…)* γράφοντας το *όνομά* της και αντιστοιχώντας τα *ορίσματα εισόδου* της με *τιμές* ή *προσδιοριστές* του μπλοκ που την καλεί
- Όταν τερματίσει μια συνάρτηση η εκτέλεση συνεχίζεται από *την αμέσως επόμενη εντολή* του μπλοκ του κώδικα από το οποίο είχε κληθεί

# Τερματισμός συναρτήσεων

Μια συνάρτηση τερματίζει και ο έλεγχος επιστρέφει στον κώδικα που την κάλεσε στις εξής δυο περιπτώσεις:

- όταν η εκτέλεσή της φτάσει στην τελευταία εντολή του μπλοκ της
- όταν συναντήσει την εντολή `return` κάπου ενδιάμεσα ή στο τέλος του μπλοκ εντολών της

Η εντολή `return <έκφραση>` χρησιμοποιείται για έξοδο από την συνάρτηση, επιστρέφοντας προαιρετικά ***ένα και μοναδικό*** *αντικείμενο (αριθμό, λίστα, πλειάδα, συμβολοσειρά, λεξικό κλπ.)* στον κώδικα που την κάλεσε, *μέσω του ονόματός της* το οποίο συνήθως βρίσκεται στο δεξί μέλος μια εκχώρησης πχ. `a = len('Hello')`

Ένα σκέτο `return` *(χωρίς κάποια έκφραση)* επιστρέφει την τιμή `None`

Αν η συνάρτηση τερματίσει χωρίς `return` *(δηλαδή όταν φτάσει στην τελευταία εντολή του μπλοκ)* τότε επίσης επιστρέφει τη τιμή `None`

# Γιατί άραγε να υπάρχουν συναρτήσεις που δεν επιστρέφουν τίποτα;

Για δυο τουλάχιστον λόγους:

- Μια συνάρτηση μπορεί να χρησιμοποιείται πχ. για να *εκτυπώσει* κάτι στην οθόνη, ή να *γράψει κάτι σε ένα αρχείο*

- Μια συνάρτηση μπορεί να έχει εσωτερικά δυνατότητα πρόσβασης *(και άρα επεξεργασίας)* σε συγκεκριμένα *αντικείμενα* του μπλοκ του κώδικα που την έχει καλέσει *(θα δούμε αργότερα πως)*

  Στην περίπτωση αυτή η επεξεργασία αυτών των αντικειμένων *γίνεται εσωτερικά στη συνάρτηση*, οπότε δεν είναι απαραίτητο να επιστραφεί κάτι όταν αυτή ολοκληρωθεί

# Συναρτήσεις και docstrings

Η *πρώτη εντολή* μιας συνάρτησης μπορεί προαιρετικά να είναι *μια συμβολοσειρά σε τριπλά εισαγωγικά ('''...''')* που ονομάζεται *συμβολοσειρά τεκμηρίωσης* ή *docstring (documentation string)*

```
def my_function():
      '''Μια απλή συνάρτηση χωρίς κάτι το ιδιαίτερο'''
      ...
      ...
```

Η συμβολοσειρά αυτή *(που συνήθως εκτείνεται και σε περισσότερες από μια γραμμές)* περιέχει ένα σύντομο κείμενο περιγραφής της συνάρτησης το οποίο εμφανίζεται όταν ενας χρήστης δώσει την εντολή `help(my_function)` *(συνήθως αυτό γίνεται μέσα από το διαδραστικό περιβάλλον)*

```
>>>help(my_function)
Μια απλή συνάρτηση χωρίς κάτι το ιδιαίτερο
```

# Δημιουργία και κλήση συνάρτησης

```
def my_function():
    ''' Μια απλή συνάρτηση χωρίς κάτι το ιδιαίτερο'''
    print("Γεια σας, μέσα από μια συνάρτηση")
```

*Δεν ξεχνάμε την άνω-κάτω τελεία*

*docstring (προαιρετικά)*

*Ο κώδικας της συνάρτησης γράφεται 1 tab πιο μέσα όπως και τα **if**, **for** κλπ*

Για να *καλέσουμε* αυτή τη συνάρτηση κάπου μέσα από τον κώδικά μας, απλά αναγράφουμε το όνομά της ακολουθούμενο από ένα ζευγάρι παρενθέσεων

```
my_function()
```

*Παράδειγμα:*

```
def my_function():
    print(" Γεια σας, μέσα από μια συνάρτηση")
    ...

...
my_function()
```

***Κώδικας συνάρτησης γραμμένος πριν το πρόγραμμα***

***Κλήση της συνάρτησης μέσα στο πρόγραμμα***

***Κώδικας προγράμματος***

Θα εμφανίσει στην κονσόλα: `Γεια σας, μέσα από μια συνάρτηση`

# Επιστροφή αποτελέσματος συνάρτησης

- Για να κάνουμε μια συνάρτηση να επιστρέψει μια *ένα αντικείμενο (απλή τιμή, λίστα, κ.λπ.)* στο πρόγραμμα από το οποίο καλείται, χρησιμοποιούμε την εντολή `return` ακολουθούμενη από είτε μια *τιμή,* είτε μια *έκφραση (η οποία τελικά επαληθεύεται και αυτή σε μια τιμή)*
- Κατά κανόνα το `return` τοποθετείται ως τελευταία εντολή του μπλοκ της συνάρτησης αλλά μπορεί να τοποθετηθεί και ενδιάμεσα, συνήθως μέσα σε ενα `if`

```
def epi_pente(x):
  return 5 * x
```

```
z = epi_pente(3)
print(z)

print(epi_pente(8))

print(f'{epi_pente(3*4/2)}')

print(17/epi_pente(2))

print(epi_pente("Z"))
```

```
15

40

30

1.7

ZZZZZ
```

# Επιστροφή αποτελέσματος συνάρτησης

- Μια συνάρτηση μπορεί να περιέχει *κανένα*, *ένα* ή *περισσότερα* `return`
  *Αν δεν υπάρχει κανένα* `return`, *τότε η συνάρτηση τερματίζει όταν η εκτέλεση φτάσει στην τελευταία γραμμή του μπλοκ της*

- Σε κάθε περίπτωση με το `return` η συνάρτηση *σταματάει να εκτελείται* ακόμα και αν ακολουθούν κι άλλες εντολές μετά από αυτό

- Ακόμα και αν μια συνάρτηση περιέχει ένα ή περισσότερα `return` μπορεί να τερματίσει *χωρίς να εκτελεστεί κανένα από αυτά (πως;)*

Παράδειγμα:

```
def f(x):
  if x > 4:
    return 3 * x
  elif x < 0:
    return 5 * x
. . .
. . .
print(f(5), f(-2), f(2) )
```

---

Θα εμφανίσει

```
15 -10 None
```

# Πέρασμα δεδομένων σε συνάρτηση - ορίσματα θέσης

- Ο πιο συνηθισμένος τρόπος να *περάσουμε δεδομένα* μέσα σε μια συνάρτηση είναι χρησιμοποιώντας *ορίσματα*
- Τα ορίσματα (arguments) *-που λέγονται αλλιώς και παράμετροι της συνάρτησης-* είναι *ονόματα μεταβλητών* που τοποθετούνται *μέσα στην παρένθεση* της συνάρτησης χωρισμένα με κόμμα

```
def my_function(arg1, arg2, arg3):
```

- Όταν *καλούμε* μια συνάρτηση μέσα από ένα πρόγραμμα, βάζουμε στην παρένθεση *ισάριθμες* μεταβλητές, τιμές ή και γενικότερα εκφράσεις χωρισμένες με κόμμα, σε *πλήρη αντιστοιχία* με τα *ορίσματα* της συνάρτησης

```
result = my_function(var1, var2, var3):
```

# Πέρασμα δεδομένων σε συνάρτηση - ορίσματα θέσης

- Το *πλήθος των τιμών* πρέπει να είναι *ίσο* με το *πλήθος των ορισμάτων* *(κατά κανόνα)*
- Η *πρώτη τιμή* αντιστοιχεί στο *πρώτο όρισμα*, η δεύτερη τιμή στο δεύτερο, … κ.ο.κ.
- Εφόσον η συνάρτησή μας επιστρέφει κάποιο *αποτέλεσμα* (μέσω `return`) τότε συνήθως τοποθετούμε την συνάρτηση στο *δεξί μέλος μιας εκχώρησης* είτε *αυτόνομα*, είτε ως *μέλος* μιας γενικότερης *έκφρασης*

```
def piliko(diereteos, dieretis):
    return diereteos / dieretis
...
x = 10
y = 4
print(piliko(x,y))        #Θα εκτυπώσει 2.5
print(piliko(20,y))       #Θα εκτυπώσει  5.0

z = 3 * piliko(x,y)
print(z)                  #Θα εκτυπώσει 7.5
```

Επειδή η αντιστοίχηση των ορισμάτων με τιμές εξαρτάται απόλυτα από την θέση τους, ονομάζονται και *ορίσματα θέσης* (*positional arguments* ή σκέτο *arguments*)

Όπως θα δούμε παρακάτω υπάρχει και άλλος τρόπος αντιστοίχησης ορισμάτων με τιμές που δεν εξαρτάται από την ακριβή τους θέση (*ορίσματα κλειδιά* – *keyword arguments*)

# Τα ορίσματα μιας συνάρτησης μπορεί να μην έχουν πάντα τον ίδιο τύπο

Μπορούμε να περάσουμε ένα όρισμα *οποιουδήποτε τύπου δεδομένων* σε μια συνάρτηση (συμβολοσειρά, αριθμό, λίστα, λεξικό κ.λπ. *(ακόμα και άλλη συνάρτηση*) και θα αντιμετωπιστεί ως *ο ίδιος τύπος* δεδομένων μέσα στη συνάρτηση

Στο επόμενο παράδειγμα βλέπουμε πως μπορούμε να καλέσουμε την ίδια συνάρτηση με όρισμα είτε μια *λίστα* είτε μια *συμβολοσειρά*, είτε με μια αριθμητική ακολουθία, με αντίστοιχο αποτέλεσμα στην κάθε περίπτωση

```
def print_a_sequence(any_sequence):
  for x in any_sequence:
    print(x, end=' ')
```

```
# Κλήση της συνάρτησης με όρισμα μια λίστα
  fruits = ["apple", "banana", "cherry"]
  print_a_sequence(fruits)

Θα εκτυπώσει apple banana cherry
```

```
# Κλήση με όρισμα μια συμβολοσειρά
  print_a_sequence("This")
  Θα εκτυπώσει T h i s
```

```
# Κλήση με όρισμα μια ακολουθία τιμών
  print_a_sequence(range(6,14,2))
    Θα εκτυπώσει 6 8 10 12
```

```
# Όμως: κλήση με όρισμα έναν ακέραιο
  print_a_sequence(48)
  TypeError: 'int' object is not iterable
```

# Το πέρασμα των ορισμάτων γίνεται με αναφορά

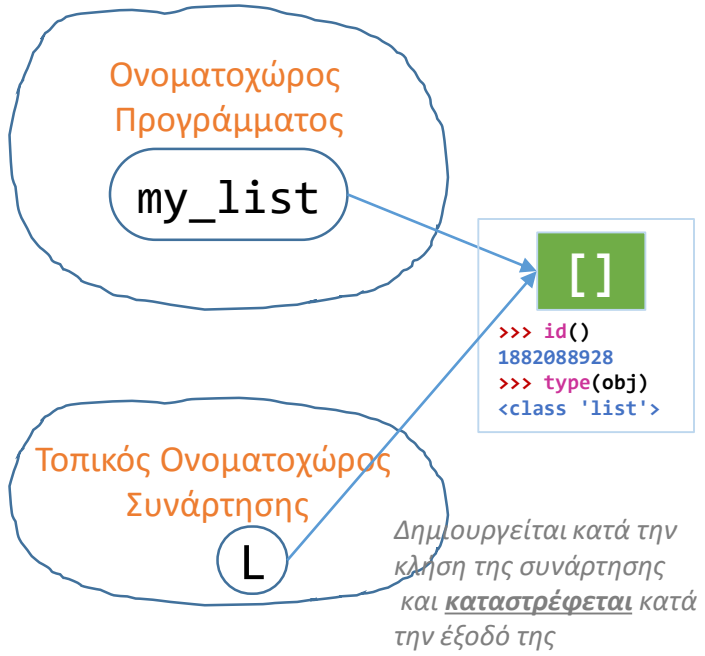
Το πέρασμα όλων των ορισμάτων σε μια συνάρτηση της Python γίνεται με αναφορά (*by reference*)

Αυτό σημαίνει ότι αν μέσα στην συνάρτηση *αλλάξει το αντικείμενο*(*) που έχει περάσει μέσα από ένα όρισμα, η αλλαγή αυτή *αντανακλάται και πίσω στον κώδικα* ο οποίος κάλεσε την συνάρτηση

```
# Ορισμός της συνάρτησης
def changeme(L):
    '''Αλλάζει τη λίστα που περνάμε στη συνάρτηση'''
    L.extend([1,2,3,4]);
    print ( "Η L μέσα στη συνάρτηση έχει γίνει: " , L)

# Κλήση της συνάρτησης
mylist = [10,20,30];
print("Η mylist ΠΡΙΝ κληθεί η συνάρτηση είναι: " , mylist)
changeme( mylist );
print("Η mylist ΑΦΟΥ κληθεί  η συνάρτηση είναι: " , mylist)
```

```
Η mylist ΠΡΙΝ κληθεί η συνάρτηση είναι:  [10, 20, 30]
Η L μέσα στη συνάρτηση έχει γίνει:       [10, 20, 30, 1, 2, 3, 4]
Η mylist ΑΦΟΥ κληθεί  η συνάρτηση είναι: [10, 20, 30, 1, 2, 3, 4]
```




(*) Φυσικά αυτό έχει νόημα μόνο για *μεταβλητά* αντικείμενα

# Είδη ορισμάτων των συναρτήσεων της Python

*Αυτά που έχουμε δει μέχρι τώρα*

1. *Ορίσματα θέσης* (*positional arguments*) – υπάρχουν δυο ειδών:
   - *Τυπικά ή απαιτούμενα ή* ορίσματα θέσης (*required positional arguments*)
   - Ορίσματα θέσης *με προκαθορισμένες τιμές* (*default positional arguments*)
2. *Ορίσματα-κλειδιά* (*keyword arguments*)
3. *Ομαδοποιημένα ορίσματα θέσης* μεταβλητού *(απροσδιόριστου)* πλήθους (`*args`)
4. *Ομαδοποιημένα ορίσματα-κλειδιά* μεταβλητού *(απροσδιόριστου)* πλήθους (`**kwargs`)

# Ορίσματα θέσης με προκαθορισμένες (default) τιμές

- Κατά την δημιουργία μια συνάρτησης μπορούμε προαιρετικά σε κάποια -ή ακόμα και σε όλα- τα *ορίσματά της* να δώσουμε *προκαθορισμένες (default) αρχικές τιμές* *γράφοντας* <*όρισμα*> = <*προκαθορισμένη τιμή*>

```
def my_function(x, y=0):
```

- Ορίσματα που έχουν προκαθορισμένες τιμές μπορούν να *παραλειφθούν* κατά την κλήση μιας συνάρτησης. Στην περίπτωση αυτή, τα ορίσματα αυτά θα πάρουν *τις προκαθορισμένες τιμές τους*
- Αν όμως *παραλείψουμε* ένα όρισμα *που δεν έχει προκαθορισμένη τιμή*, τότε μας επιστρέφεται *μήνυμα λάθους*

Απλό όρισμα *χωρίς προκαθορισμένη (default) τιμή*

```
def Say_Hello(Name):
  print("Γειά σου " + Name + "!")
. . .
Say_Hello("Κώστα")
Say_Hello("Μαρία")
Say_Hello()

Γειά σου Κώστα!
Γειά σου Μαρία!
TypeError: Say_Hello()
missing 1 required positional argument: 'Name'
```

Απλό όρισμα με *προκαθορισμένη (default) τιμή*

```
def Say_Hello(Name = "άγνωστε"):
  print("Γειά σου " + Name + "!")
. . .
Say_Hello("Κώστα")
Say_Hello("Μαρία")
Say_Hello()


Γειά σου Κώστα!
Γειά σου Μαρία!
Γειά σου άγνωστε!
```

# Ορίσματα-κλειδιά

- Όταν ***καλούμε*** μια συνάρτηση, μπορούμε *εναλλακτικά* να περνάμε κάποιες από τις παραμέτρους της με τα λεγόμενα *ορίσματα-κλειδιά* (keywords arguments) δηλαδή ορίσματα που έχουν τη μορφή

  <όρισμα> = <τιμή>

- Για να γίνει αυτό, θα πρέπει κατά την κλήση της συνάρτησης *να γνωρίζουμε επακριβώς τα ονόματα αυτών των ορισμάτων* όπως δηλώθηκαν κατά τον ορισμό της συνάρτησης
- Όταν υπάρχουν ορίσματα-κλειδιά, η *σειρά* με την οποία αναγράφονται ***παύει να είναι σημαντική***
- Κατά την κλήση μιας συνάρτησης τα *ορίσματα-κλειδιά* γράφονται πάντα ***μετά*** *τα ορίσματα θέσης (αν υπάρχουν τέτοια)*, για τα οποία εξακολουθεί να ισχύει η αντιστοίχηση *ένα-προς-ενα*

```
def my_function(child1, child2, child3):
  print("Το νεότερο παιδί είναι ο ", child3)
. . .
my_function("Πέτρος", child3 = "Νίκος", child2 = "Κώστας")
```

*Ορίσματα θέσης (μπαίνουν πρώτα)*

*Ορίσματα-κλειδιά (ακολουθούν τα ορίσματα θέσης)*

Θα εμφανίσει στην κονσόλα: `Το νεότερο παιδί είναι ο Νίκος`

# Απροσδιόριστος αριθμός ορισμάτων σε συνάρτηση – Παράδειγμα 1

Κάποιες φορές χρειάζεται να ορίσουμε μια συνάρτηση χωρίς να ξέρουμε από πριν *το πλήθος των ορισμάτων της* *(ακούγεται περίεργο, αλλά ορισμένες φορές είναι ιδιαίτερα χρήσιμο)*

Έστω π.χ. ότι θέλουμε να δημιουργήσουμε μια συνάρτηση `add()` που όταν καλείται να μας επιστρέφει *το άθροισμα* των αριθμών που της δίνουμε σαν ορίσματα δηλαδή πχ:

`add(1,2)` να επιστρέφει `3`

`add(2,6,1)` να επιστρέφει `9`

`add(1,2,1,3)` να επιστρέφει `7`

Αυτό γίνεται ως εξής:

```
def add(*num):
    sum = 0
    for n in num:
        sum = sum + n
    print(sum)
```

Το αστεράκι (*) μπροστά από ένα όρισμα σημαίνει ότι το συγκεκριμένο όρισμα είναι μια *πλειάδα ή λίστα ή άλλος ακολουθιακός τύπος* που περιέχει *απροσδιόριστο αριθμό ορισμάτων θέσης* των οποίων το περιεχόμενο ορίζεται *κάθε φορά που καλείται αυτή η συνάρτηση*

# Απροσδιόριστος αριθμός ορισμάτων σε συνάρτηση – Παράδειγμα 2

Αν στον ορισμό της συνάρτησης υπάρχουν και απαιτούμενα *(τυπικά) ορίσματα θέσης*, τότε αυτά *αναγράφονται πρώτα*, και ακολουθούν τα ομαδοποιημένα ορίσματα θέσης με τη μορφή **<όνομα παραμέτρου ομαδοποιημένων ορισμάτων >*

```
def family(father, mother, *kids):
  print("Πατέρας: ", father, " - Μητέρα: ", mother )
  print("Παιδιά: ",end= "")
  for x in kids:
            print(x, end= " ")
...

family("Πέτρος", "Ελένη", "Κώστας", "Μαρία", "Νίκος")
Πατέρας: Πέτρος - Μητέρα: Ελένη
Παιδιά: Κώστας Μαρία Νίκος

family("Πέτρος ", "Ελένη")
Πατέρας: Πέτρος - Μητέρα: Ελένη
Παιδιά:

family(mother = "Ελένη", father = "Πέτρος",  "Κώστας", "Μαρία")
Πατέρας: Πέτρος - Μητέρα: Ελένη
Παιδιά: Κώστας Μαρία
```

Ομαδοποιημένα ορίσματα θέσης που αντιστοιχούν στο ***kids**

Πέρασμα τυπικών παραμέτρων με ορίσματα-κλειδιά

# Απροσδιόριστος αριθμός ορισμάτων σε συνάρτηση – Γενίκευση 1

Στην Python μπορούμε να περνάμε αυθαίρετο αριθμό ορισμάτων σε μια συνάρτηση χρησιμοποιώντας τα δυο ειδικά σύμβολα `*args` (*ομαδοποιημένα ορίσματα θέσης* / *grouped arguments* ) και `**kwargs` (*ομαδοποιημένα ορίσματα-κλειδιά* / *grouped keyword arguments* )

- Κάθε συνάρτηση περιλαμβάνει ***το πολύ*** ένα όρισμα τύπου `*args` και ένα τύπου `**kwargs`
- Η δήλωση `*args` προηγείται πάντα της δήλωσης `**kwargs`

Κατά τον *ορισμό* μιας συνάρτησης:

- το `*args` θεωρείται ως μια λίστα/πλειάδα με *απροσδιόριστο πλήθος* ορισμάτων ***θέσης***
- το `**kwargs` θεωρείται ως ένα λεξικό με *απροσδιόριστο πλήθος* ορισμάτων-***κλειδιών***

Κατά την *κλήση* μιας συνάρτησης:

- το `args` περιέχει σε μορφή λίστας/πλειάδας ενα *συγκεκριμένο πλήθος* ορισμάτων ***θέσης***
- το `kwargs` περιέχει σε μορφή λεξικού ενα *συγκεκριμένο πλήθος* ορισμάτων -***κλειδιών***

Για τα ονόματα των ομαδοποιημένων ορισμάτων έχει επικρατήσει να χρησιμοποιούμε το `*args` και το `**kwargs`. Μπορούμε όμως αν θέλουμε χρησιμοποιήσουμε δικά μας ονόματα πχ `*kids` *(βλ. προηγούμενο παράδειγμα)* ή `**parms` αρκεί να φροντίσουμε *να ξεκινάνε πάντα με ένα ή αντίστοιχα δυο αστεράκια*

# Απροσδιόριστος αριθμός ορισμάτων σε συνάρτηση – Γενίκευση 2

Σε μια συνάρτηση, τα *`*args`* (ομαδοποιημένα ορίσματα θέσης) και *`**kwargs`* (ομαδοποιημένα ορίσματα-κλειδιά) ακολουθούν τα απλά ορίσματα θέσης (εφόσον υπάρχουν τέτοια)

Έτσι ο γενικότερος ορισμός μιας συνάρτησης είναι

```
def function_name( [arg1], [arg2], . . . [argk], [*args], [**kwargs] ):
```

Όπου

- `[arg1], [arg2], . . . [argk]` τα *τυπικά ορίσματα θέσης* (παράμετροι) της συνάρτησης
- `[*args]` *μια πλειάδα ή λίστα* που περιέχει ομαδοποιημένα *ορίσματα θέσης*
- `[**kwargs]` ένα *λεξικό* που περιέχει ομαδοποιημένα *ζεύγη από ορίσματα-κλειδιά*

και όπως είπαμε και νωρίτερα:

- Σε κάθε συνάρτηση δεν μπορεί να ορίζονται περισσότερα από ένα *`*args`* και ένα *`**kwargs`*
- Η δήλωση *`*args`* προηγείται πάντα της δήλωσης *`**kwargs`*

# Συνοπτικό παράδειγμα χρήσης των *args και **kwargs

```
>>> def test(p, q, *ar, **kw):
        print(p, q)
        print(ar)
        print(kw)
```

*Ορισμός μιας συνάρτησης που περιέχει ταυτόχρονα: απλά ορίσματα θέσης* `p,q`, *ομαδοποιημένα ορίσματα θέσης* `*ar` *και ομαδοποιημένα ορίσματα-κλειδιά* `**kw`

---

```
>>> test(1, 2, 3, 4, 5, 6, 7, a=2, b=3, c=5)
1, 2
(3, 4, 5, 6, 7)
{'a': 2, 'c': 5, 'b': 3}
```

*Κλήση της συνάρτησης μόνο με απλά ορίσματα θέσης και ορίσματα-κλειδια*

---

```
>>> tup=(1, 2, 3, 4)
>>> dic={'a': 10, 'b':20}
>>> test(8, 9, *tup, **dic)  #όμοιο με το test(8, 9, 1, 2, 3, 4, a=10, b=20)
8, 9
(1, 2, 3, 4)
{'a': 10, 'b': 20}
```

*Κλήση της συνάρτησης με απλά ορίσματα θέσης, μια πλειάδα και ένα λεξικό*

# Η εντολή pass

- Κάποιες φορές, για λόγους *πληρότητας της δομής ενός προγράμματος* επιθυμούμε να ορίσουμε μια συνάρτηση *αρχικά μόνο κατ' όνομα*, και να επανέλθουμε *αργότερα* για να *γράψουμε τον κώδικά της*
- Επειδή το μπλοκ του κώδικα μιας συνάρτησης *δεν μπορεί να είναι κενό*, το αντικαθιστούμε προσωρινά την εντολή `pass`

```
def my_function():
    pass
```

- Με τον τρόπο αυτό μπορούμε να *συνεχίσουμε να γράφουμε τον κώδικα* στο πρόγραμμά μας, ακόμα και αν *περιέχει κλήσεις* σε αυτή τη συνάρτηση
- Σε *περίπτωση κλήσης* από τον κώδικα του προγράμματος, μια συνάρτηση που έχει οριστεί με `pass`, *επιστρέφ*ει πάντα `None`
- Όταν είμαστε έτοιμοι, *αντικαθιστούμε* το `pass` με τον *κανονικό κώδικα* της συνάρτησης

# Αναδρομικές Συναρτήσεις

*Αναδρομικές* ονομάζονται οι συναρτήσεις που *καλούν τον εαυτό τους* μέσα από τον κώδικά τους. Χρησιμοποιούνται μεταξύ άλλων και για να υλοποιήσουν αναδρομικούς τύπους

πχ υπολογισμός του ν παραγοντικό (n factorial ή απλά n!)

$$n! = \underbrace{1 * 2 * 3 * \ldots * (n-1)}_{(n-1)!} * n = \underbrace{(n-1)!}_{\text{Αναδρομικός τύπος}} * n$$

Εξ ορισμού:
**1! = 1**

Υλοποίηση αναδρομικής συνάρτησης `factorial(n)`

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return factorial(n-1) * n
```

Κάθε αναδρομική συνάρτηση έχει στον κώδικά της μια *"συνθήκη εξόδου από την αναδρομή"*, δηλαδή ένα σημείο από το οποίο η συνάρτηση επιστρέφει κάποια τιμή *χωρίς να χρειαστεί να καλέσει ξανά τον εαυτό της.*

Αν δεν υπήρχε αυτή η συνθήκη εξόδου, τότε η συνάρτηση *θα καλούσε τον εαυτό της επ' άπειρον*

# Αναλυτικός υπολογισμός του 3! με αναδρομή

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return factorial(n-1) * n
```

```
x = factorial(3)
```

- Η `factorial(3)` καλεί την `factorial(2)`
- Η `factorial(2)` καλεί την `factorial(1)`
- Η `factorial(1)` επιστρέφει στην `factorial(2)` που την κάλεσε, την τιμή `1`
- Η `factorial(2)` επιστρέφει στην `factorial(3)` που την κάλεσε, την τιμή `1*2 (=2)`
- Η `factorial(3)` επιστρέφει στο πρόγραμμα που την κάλεσε, την τιμή `1*2*3 (=6)`
- Το x παίρνει την τιμή `6`

# Εναλλακτικός τρόπος υπολογισμού του ν παραγοντικό

Ο υπολογισμός του ν-παραγοντικό μπορεί φυσικά να γίνει και χωρίς τη χρήση αναδρομής, με τον "παραδοσιακό" τρόπο:

```
def factorial_simple(n):
    npar = 1
    for j in range(2,n+1)
        npar *= j
    return npar
```

# Εμβέλεια και διάρκεια ζωής μιας μεταβλητής

## *Scope and lifetime of a variable*

- *Εμβέλεια ή περιοχή ισχύος* μιας μεταβλητής είναι το *κομμάτι του προγράμματος* μέσα στο οποίο η μεταβλητή αυτή *αναγνωρίζεται*

  Οι *παράμετροι κλήσης* και οι *μεταβλητές* που ορίζονται μέσα σε μια συνάρτηση *δεν είναι ορατές* έξω από αυτήν. Λέμε ότι έχουν *τοπική εμβέλεια* (*local scope*) και τις ονομάζουμε *τοπικές μεταβλητές* (*local variables*)

- *Διάρκεια ζωής* (*lifetime*) μιας μεταβλητής είναι η περίοδος κατά την οποία η μεταβλητή *υπάρχει στη μνήμη* (δηλ. το *όνομά* της *συνδέεται* κάποιο *αντικείμενο*)

  Οι *τοπικές μεταβλητές* μιας συνάρτησης διαρκούν *όσο η συνάρτηση εκτελείται*, και *καταστρέφονται* μόλις η συνάρτηση επιστρέψει. Επομένως αν *καλέσουμε ξανά* μια συνάρτηση, αυτή *δεν θα θυμάται τις τιμές* που είχαν οι τοπικές μεταβλητές της από την προηγούμενη κλήση τους

*Ο ορισμός είναι γενικός για όλες τις γλώσσες προγραμματισμού.*
*Θυμόμαστε ότι στην Python το αντίστοιχο της μεταβλητής είναι ο προσδιοριστής*

# Εμβέλεια και διάρκεια ζωής μιας μεταβλητής

Μεταβλητές *(στην Python προσδιοριστές)* που ορίζονται σε *διαφορετικές περιοχές ισχύος* του προγράμματος *μπορούν να έχουν το ίδιο όνομα*

Αυτό *καλό θα ήταν να αποφεύγεται* επειδή κάνει τον κώδικά μας πιο "*δυσανάγνωστο*", όμως είναι *απολύτως φυσιολογικό* στην Python *(και σε άλλες γλώσσες)*

Τοπική μεταβλητή συνάρτησης my_funct

Μεταβλητή προγράμματος με ίδιο όνομα

```
def my_func():
    x = 10
    print("Value of x in function:", x)

x = 20
my_func()
print("Value of x in main program:", x)
```

---

```
Value of x in function: 10
Value of x in main program: 20
```

# Τοπικές μεταβλητές Προσοχή!

Όταν σε μια συνάρτηση *ορίσουμε μια μεταβλητή* δίνοντάς της κάποια τιμή (π.χ. x=3), τότε η μεταβλητή αυτή θεωρείται *τοπική μεταβλητή της συνάρτησης* (local variable) ακόμα και αν έχει *ίδιο όνομα* με άλλη μεταβλητή έξω από τη συνάρτηση *(το είδαμε στο προηγούμενο παράδειγμα)*

Αν όμως μέσα στην συνάρτηση προσπαθήσουμε να χρησιμοποιήσουμε μια μεταβλητή *χωρίς να την έχουμε ορίσει προηγουμένως* τότε η Python ανατρέχει τον *ονοματοχώρο του προγράμματος που περιέχει τη συνάρτηση* ψάχνοντας να δει αν υπάρχει μεταβλητή με αυτό το όνομα, και τότε:

- Αν μεν *υπάρχει*, τότε θεωρεί ότι *αναφερόμαστε σε αυτήν* οπότε την χρησιμοποιεί κανονικά
- Αν *δεν υπάρχει*, τότε μας επιστρέφεται *μήνυμα λάθους*

```
x = 5
def f1():
    x = 42
    print("Μέσα στην f1: x=", x)
 def f2():
    print("Μέσα στην f2: x=", x)
f1()
f2()
Μέσα στην f1: x = 42
Μέσα στην f2: x = 5
```

```
y = 5
def f1():
    x = 42
    print("Μέσα στην f1: x=", x)
 def f2():
    print("Μέσα στην f2: x=", x)
f1()
f2()
Μέσα στην f1: x = 42
print("Μέσα στην f2: x=", x)
NameError: name 'x' is not defined
```

# Καθολικές ή οικουμενικές μεταβλητές (global variables)

- Ορίζονται *μέσα σε μια συνάρτηση* βάζοντας μπροστά της τη λέξη-κλειδί `global` *(πχ.* `global x`*)*
- Παρά το ότι ορίζονται μέσα στη συνάρτηση ***ΔΕΝ*** τοποθετούνται στον ονοματοχώρο της συνάρτησης, αλλά σε εκείνον του *προγράμματος που καλεί την συνάρτηση* που σημαίνει ότι *εξακολουθούν να υφίστανται και μετά την επιστροφή της συνάρτησης*
- Αν στον ονοματοχώρο του προγράμματος υπάρχει *άλλη μεταβλητή με το ίδιο όνομα* τότε αυτή *αντικαθίσταται* από την καινούρια που ορίσαμε, *υπό την προϋπόθεση* ότι η global μεταβλητή *έχει αρχικοποιηθεί (ο ορισμός μιας μεταβλητής ως global ενεργοποιείται μόνο εφόσον δοθεί κάποια **τιμή** σε αυτήν)*

```
x = "Hello"
def set_x():
    global x
    x = 42
def display_x():
    print("x=", x)
...
set_x()
display_x()
```

```
x=42
```

```
x = "Hello"
def set_x():
    global x
    #x = 42
def display_x():
    print("x=", x)
...
set_x()
display_x()
```

```
x= Hello
```

Η **x** ορίζεται μεν σαν global, αλλά ουδέποτε αρχικοποιείται μέσα στην συνάρτηση

Προσοχή όμως: αν *ορίσουμε* μια καθολική μεταβλητή με το ίδιο όνομα αλλά ***δεν την χρησιμοποιήσουμε***, τότε η ομώνυμη μεταβλητή μέσα στο πρόγραμμα *δεν επηρεάζεται*.

# Ενσωματωμένες (built in) συναρτήσεις

Σύνολο συναρτήσεων *οι οποίες έχουν ενσωματωθεί (built-in) στον πυρήνα της Python*
ώστε να είναι *ανά πάσα στιγμή διαθέσιμες* για χρήση

**A**
abs()
aiter()
all()
any()
anext()
ascii()

**B**
bin()
bool()
breakpoint()
bytearray()
bytes()

**C**
callable()
chr()
classmethod()
compile()
complex()

**D**
delattr()
dict()
dir()
divmod()

**E**
enumerate()
eval()
exec()

**F**
filter()
float()
format()
frozenset()

**G**
getattr()
globals()

**H**
hasattr()
hash()
help()
hex()

**I**
id()
input()
int()
isinstance()
issubclass()
iter()

**L**
len()
list()
locals()

**M**
map()
max()
memoryview()
min()

**N**
next()

**O**
object()
oct()
open()
ord()

**P**
pow()
print()
property()

**R**
range()
repr()
reversed()
round()

**S**
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

**T**
tuple()
type()

**V**
vars()

**Z**
zip()

_
__import__()

Στην έκδοση 3.12 της Python υπάρχουν 71 ενσωματωμένες συναρτήσεις

Ο αριθμός αυτός συνήθως αλλάζει από έκδοση σε έκδοση

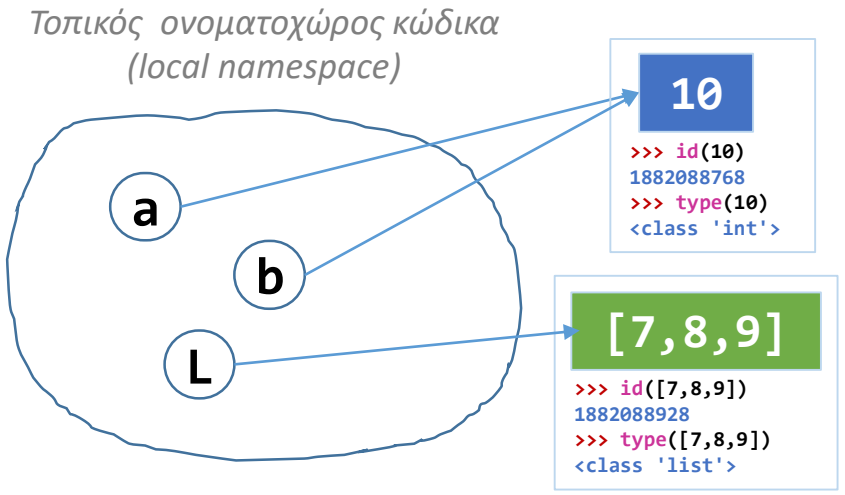
# Ονοματοχώροι (Namespaces)

Συλλογές από *συμβολικά ονόματα* που το καθένα *προσδιορίζει (αναφέρεται σε)* ένα συγκεκριμένο *αντικείμενο*

Παράδειγμα: αν σε ένα κομμάτι κώδικα υπάρχουν οι εντολές `a = 10`, `b = 10` και `L = [7,8,9]`

τότε τα ονόματα `a`, `b` και `L` ανήκουν στον ονοματοχώρο του συγκεκριμένου κώδικα και απεικονίζονται ως εξής:

- τα μεν `a` και `b` στον (ίδιο) ακέραιο αριθμό `10`
- το δε `L` στην λίστα `[7,8,9]`




Μπορούμε να σκεφτούμε έναν ονοματοχώρο σαν *λεξικό* όπου τα *κλειδιά* είναι τα *ονόματα* των αντικειμένων και οι *τιμές* τα ίδια τα *αντικείμενα*

```
{'a':10, 'b':10, L:[7,8,9]}
```

*Στην πραγματικότητα, έτσι ακριβώς υλοποιούνται οι ονοματοχώροι εσωτερικά στην Python (σαν λεξικά)*

# Ονοματοχώροι (Namespaces)

*Σε ένα πρόγραμμα Python υπάρχουν τεσσάρων ειδών ονοματοχώροι*

***Built in / Global / Enclosing / Local***

- Κάθε ονοματοχώρος έχει διαφορετική *"διάρκεια ζωής (lifespan)"*
- Καθώς η Python εκτελεί ένα πρόγραμμα, *δημιουργεί νέους ονοματοχώρους* όποτε απαιτείται και τους *διαγράφει* όταν δεν χρειάζονται πλέον
- Τυπικά, σε ένα πρόγραμμα μπορούν να συνυπάρχουν ταυτόχρονα *πολλοί διαφορετικοί ονοματοχώροι* όμως είναι εντελώς *απομονωμένοι* μεταξύ τους
- *Διαφορετικοί ονοματοχώροι* μπορεί να περιέχουν *ίδια ονόματα* τα οποία όμως εν γένει απεικονίζονται σε *διαφορετικά αντικείμενα*

# Αρχικός ονοματοχώρος της Python (built-in namespace)

Δημιουργείται *αυτόματα κάθε φορά που ξεκινάμε την Python* και περιέχει *όλα τα ενσωματωμένα ονόματα* (built-in names) που αντιστοιχούν σε *αντικείμενα* που είναι *διαθέσιμα* σε μας *εξ' ορισμού*, επομένως δεν χρειάζεται να τα προσδιορίσουμε εμείς οι ίδιοι.

*Ανάμεσά τους περιλαμβάνονται*

- *Ονόματα κυριολεκτημάτων (literals)*

  `True, False, None...`

- *Ονόματα ενσωματωμένων συναρτήσεων*

  `print, len, list, set, range...`

- *Ονόματα εξαιρέσεων (exceptions)*

  `SyntaxError, IndexError, EOFError...`

*Μπορούμε να δούμε το περιεχόμενό του καλώντας την συνάρτηση* `dir(_builtins_)` *η οποία επιστρέφει μια λίστα με όλα τα παραπάνω ονόματα*

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'BaseException','BlockingIOError', 'BrokenPipeError', 'BufferError',
 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
 'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning',
 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
 'Warning', 'ZeroDivisionError', '_', '__build_class__', '__debug__',
 '__doc__', '__import__', '__loader__', '__name__', '__package__',
 '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray',
 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate',
 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',
 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

# Ονοματοχώρος προγράμματος (global namespace)

- Περιέχει όλα τα *ονόματα* που ορίζονται στο επίπεδο του *κύριου προγράμματος*

- Δημιουργείται *όταν ξεκινά* η εκτέλεση ενός προγράμματος και παραμένει σε ισχύ *μέχρι αυτό να τερματιστεί*

- Κάθε *δομοστοιχείο*(*) *(module)* που φορτώνεται μέσα από ένα πρόγραμμα με την εντολή `import`(*) έχει τον δικό του ξεχωριστό ονοματοχώρο Global

  *Αυτό πρακτικά σημαίνει ότι κατά την εκτέλεση ενός προγράμματος μπορεί να υπάρχουν ενεργοί παράλληλα περισσότεροι από ενας ονοματοχώροι global*

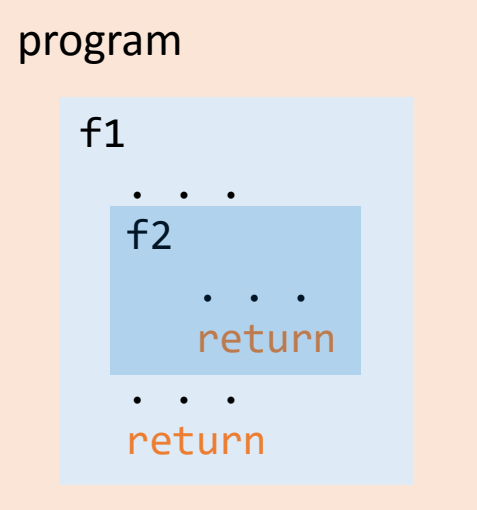
(*) *θα μιλήσουμε γι' αυτά αναλυτικά παρακάτω*

# Τοπικοί ονοματοχώροι συναρτήσεων (*local και enclosing)*

- *Local:* Τοπικοί ονοματοχώροι *συναρτήσεων* που καλούνται μέσα από ένα πρόγραμμα
- Είναι *ενεργοί* για όσο διάστημα *εκτελείται* μια συνάρτηση, και *καταστρέφονται* κατά την *έξοδό* της
- Αν ένα πρόγραμμα καλεί μια συνάρτηση $f_1$ και αυτή με την σειρά της καλεί μια άλλη συνάρτηση $f_2$ τότε ο *local* ονοματοχώρος της $f_1$ είναι ταυτόχρονα *enclosing* για την $f_2$ η οποία -φυσικά- έχει και αυτή τον δικό της *local* ονοματοχώρο

Σε αυτή την περίπτωση κάθε όνομα της $f_2$ ανήκει είτε

- στον τοπικό της ονοματοχώρο
- στον τοπικό ονοματοχώρο της $f_1$ που είναι ταυτόχρονα *enclosing* για την $f_2$
- στον ονοματοχώρο *global* του προγράμματος
- στον ονοματοχώρο *built-in* της Python



# Δομή Ονοματοχώρων της Python

# Μεταβλητές και μέθοδοι με απλές / διπλές κάτω παύλες

Στην Python οι προσδιοριστές (ονόματα) που αρχίζουν ή/και τελειώνουν με απλή κάτω παύλα ή διπλή κάτω παύλα (*double underscore "dunder"* ) έχουν ειδική σημασία

| | | |
|---|---|---|
| Απλή κάτω παύλα στην αρχή | `_foo` | Υποδεικνύει ότι η μεταβλητή αυτή προορίζεται για εσωτερική χρήση (τοπική μεταβλητή). Γενικά δεν επιβάλλεται από την Python, απλά προτείνεται ως καλή πρακτική για τους προγραμματιστές. |
| Απλή κάτω παύλα στο τέλος | `foo_` | Χρησιμοποιείται κυρίως για να ορίσουμε μεταβλητές που έχουν παρόμοιο όνομα με λέξεις-κλειδιά της Python *π.χ.* `len_, print_ ...` |
| Διπλή κάτω παύλα στην αρχή | `__foo` | Έχει ειδική σημασία όταν χρησιμοποιείται σε περιβάλλον ορισμού κλάσεων (name mangling) . Η χρήση της *επιβάλλεται* από την Python. |
| Διπλή κάτω παύλα στην αρχή και το τέλος | `__foo__` | Χρησιμοποιείται σε ονόματα ειδικών εσωτερικών μεταβλητών και μεθόδων της Python. *π.χ.* `__name__, __doc__, __init__...` |
| Ειδική περίπτωση: Σκέτη κάτω παύλα | `_` | Μερικές φορές χρησιμοποιείται ως όνομα για *προσωρινές* ή *ασήμαντες* μεταβλητές *("ό,τι νάναι")* *π.χ.* `for _ in range(10)` **Επίσης**: Περιέχει το αποτέλεσμα της πιο πρόσφατης έκφρασης στο διαδραστικό περιβάλλον της Python. |

# Δομοστοιχεία και Πακέτα (Modules and Packages)

- *Δομοστοιχείο (module)* : Κάθε *αρχείο με την κατάληξη .py* που περιέχει εντολές της γλώσσας Python. - Κάθε πρόγραμμα Python που αποθηκεύουμε σε αρχείο, αποτελεί δομοστοιχείο της Python

- *Πακέτο (package):* Ενας *φάκελος* που περιέχει αρχεία με την κατάληξη .py (δομοστοιχεία) και ένα επιπλέον *ειδικό αρχείο* με όνομα "`__init__.py`"
  - Το αρχείο "`__init__.py`" χρησιμοποιείται για να σηματοδοτήσει ότι ο φάκελος είναι ένα *πακέτο* και *όχι ενας απλός φάκελος* με αρχεία της Python
  - Το *όνομα του φακέλου* αποτελεί και το *όνομα του πακέτου*
  - Το αρχείο "`__init__.py`" ΄μπορεί να είναι *κενό*, ή να περιέχει εντολές αρχικοποίησης του πακέτου που εκτελούνται όταν φορτώνεται
  - Τα περιεχόμενα δομοστοιχεία ενός πακέτου είναι συνήθως *αυτόνομες συναρτήσεις* οι οποίες καλούνται από τα προγράμματα που αποκτούν πρόσβαση στο πακέτο μέσω της εντολής *import* που θα δούμε αμέσως μετά

# Η εντολή import

Για να *εισάγουμε* και να *χρησιμοποιήσουμε* στο πρόγραμμά μας κώδικα που περιέχεται σε άλλο *δομοστοιχείο* ή *πακέτο* της Python χρησιμοποιούμε την εντολή `import`

Πχ με την εντολή `import math` εισάγουμε στο πρόγραμμά μας το πακέτο `math` που περιέχει χρήσιμες μαθηματικές συναρτήσεις, προσθέτοντας το όνομά του στον καθολικό ονοματοχώρο του προγράμματός μας. Αν θέλουμε να δούμε τι περιέχει ένα πακέτο *(αφού το εισάγουμε)* το κάνουμε δίνοντας `dir(όνομα_πακέτου)`

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'tau', 'trunc']
```

# Η εντολή import

Για να καλέσουμε μια συνάρτηση από ένα πακέτο που έχουμε εισάγει, χρησιμοποιούμε τη σύνταξη <όνομα πακέτου> . <όνομα συνάρτησης>

Πχ. `a = math.cos(2.35)` ή `print(math.pi)`

Αν θέλουμε να χρησιμοποιήσουμε μόνο συγκεκριμένες συναρτήσεις από ένα πακέτο, μπορούμε να εισάγουμε μόνο αυτές

```
>>> from math import cos
>>> from math import pi
```

Στην περίπτωση αυτή μπορούμε να τις καλούμε γράφοντας σκέτο το όνομα τους

Πχ. `a = cos(2.35)` ή `print(pi)`

Αν θέλουμε μπορούμε να εισάγουμε μια συνάρτηση *αλλάζοντάς της το όνομα με ένα δικό μας*, χρησιμοποιώντας το `as`

```
>>> from math import cos as synimitono
>>> print(synimitono(2.35))
```

# Η εντολή import

Για να εισάγουμε *όλες τις συναρτήσεις* ενός πακέτου, χρησιμοποιούμε την σύνταξη:

```
from <όνομα πακέτου> import *
```

Στην περίπτωση αυτή όλα τα ονόματα των συναρτήσεων του πακέτου προστίθενται στον καθολικό ονοματοχώρο και μπορούν να χρησιμοποιηθούν απ' ευθείας

```
>>> from math import *
>>> print(cos(2.35), sin(pi), log2(0.4))
```

Η πρακτική αυτή *δεν συνιστάται* όταν χρειάζεται να εισάγουμε *περισσότερα από ένα πακέτα,* καθώς υπάρχει περίπτωση κάποιες συναρτήσεις να έχουν *κοινό όνομα* στα δυο πακέτα.

Στην περίπτωση αυτή, το (κοινό) όνομα *αναφέρεται* στην συνάρτηση του πακέτου που *εισήχθη τελευταίο*

# Παράδειγμα δημιουργίας δικού μας πακέτου συναρτήσεων

1. Στο IDLE ή άλλο Python editor δημιουργούμε ένα αρχείο με τίτλο *calculations.py* και μέσα του ορίζουμε τις συναρτήσεις add και sub. Αν δοκιμάσουμε να το τρέξουμε, παρατηρούμε ότι δεν συμβαίνει τίποτα *(γιατί;)*
2. Αποθηκεύουμε το αρχείο στο path που περιέχει *όλες τις βιβλιοθήκες της Python* (συνήθως πρόκειται για τον φάκελο `Lib` μέσα στον φάκελο της Python) , ή εναλλακτικά *προσθέτουμε το path του αρχείου* στη λίστα με τα paths της Python `os.path`

calculations.py

```
def add(x,y):
    return (x+y)
def sub(x,y):
    return (x-y)
```

## Διαφορετικοί τρόποι εισαγωγής και χρήσης του πακέτου που δημιουργήσαμε

```
>>> import calculations
>>> print (calculations.add(3,2))
5
```

```
>>> from calculations import add
>>> print (add(3,4))
7
```

```
>>> from calculations import *
>>> print (sub(3,2))
1
```

## Τι ακριβώς κάνει η συνθήκη `if __name__ == "__main__":` ;

Πολλές φορές όταν γράφουμε ένα πακέτο με συναρτήσεις, θα θέλαμε να δοκιμάσουμε την λειτουργία τους πριν τις τοποθετήσουμε σε μια βιβλιοθήκη της Python. Για να γίνει αυτό, εργαζόμαστε ως εξής:

Το *__name__* είναι μια ειδική μεταβλητή του περιβάλλοντος της Python η οποία περιέχει το όνομα του προγράμματος (script) που εκτελείται κάθε στιγμή στην Python. Η μεταβλητή αυτή παίρνει διαφορετική τιμή ανάλογα με τον τρόπο που ένα script φορτώνεται για εκτέλεση

- Αν το script είναι το *κυρίως πρόγραμμα*, τότε το `__name__` παίρνει την ειδική τιμή `"__main__"`.
- Αν το script ανήκει σε ένα δομοστοιχείο ή πακέτο που έχει εισαχθεί στο κυρίως πρόγραμμα με την εντολή `import`, τότε το `__name__` παίρνει το όνομα του αρχείου που περιέχει αυτό το στοιχείο

Η διαφοροποίηση αυτή μας δίνει την δυνατότητα να τοποθετούμε ένα μπλοκ κώδικα μέσα σε ένα script πχ που περιέχει ένα πακέτο συναρτήσεων ο οποίος *να εκτελείται μόνο αν αυτό το script τρέχει απευθείας σαν αυτόνομο πρόγραμμα*, αλλά όχι όταν το ίδιο script εισάγεται με την εντολή `import` σε ένα άλλο πρόγραμμα

Πρακτικά αυτό γίνεται γράφοντας στο τέλος του πακέτου την συνθήκη `if __name__ == "__main__"` και κάτω από αυτήν ένα μπλοκ του κώδικα με εντολές για δοκιμή τις συναρτήσεων

Αν το πακέτο τρέξει φυσιολογικά στο περιβάλλον της Python, ο κώδικας αυτός θα εκτελεστεί, οπότε θα μπορέσουμε να δούμε στην πράξη την λειτουργία του πακέτου. Αν όμως το ίδιο πακέτο εισαχθεί σε άλλο πρόγραμμα με την εντολή `import`, τότε το μπλοκ αυτό του κώδικα δεν θα εκτελεστεί

# Παράδειγμα χρήσης της συνθήκης `if __name__ == "__main__":`

*Συνέχεια του προηγούμενου παραδείγματος*

Στο τέλος του πακέτου έχουμε τοποθετήσει τη συνθήκη `if __name__ == "__main__":` και ακριβώς αποκάτω ένα μπλοκ κώδικα για δοκιμή των συναρτήσεων

- Αν τρέξουμε το πακέτο αυτόνομα, ο κώδικας αυτός θα εκτελεστεί κανονικά και θα εκτυπώσει

```
Testing add 7
Testing sub 4
```

- Αν το πακέτο εισαχθεί σε άλλο πρόγραμμα με την εντολή `import` τότε ο κώδικας μέσα στο `if` δεν πρόκειται να εκτελεστεί

calculations.py

```
def add(x,y):
    return (x+y)
def sub(x,y):
    return (x-y)

if __name__ == "__main__":
    print("Testing add ", add(3+4))
    print("Testing sub ", sub(5-1))
```

# Τέλος Διάλεξης

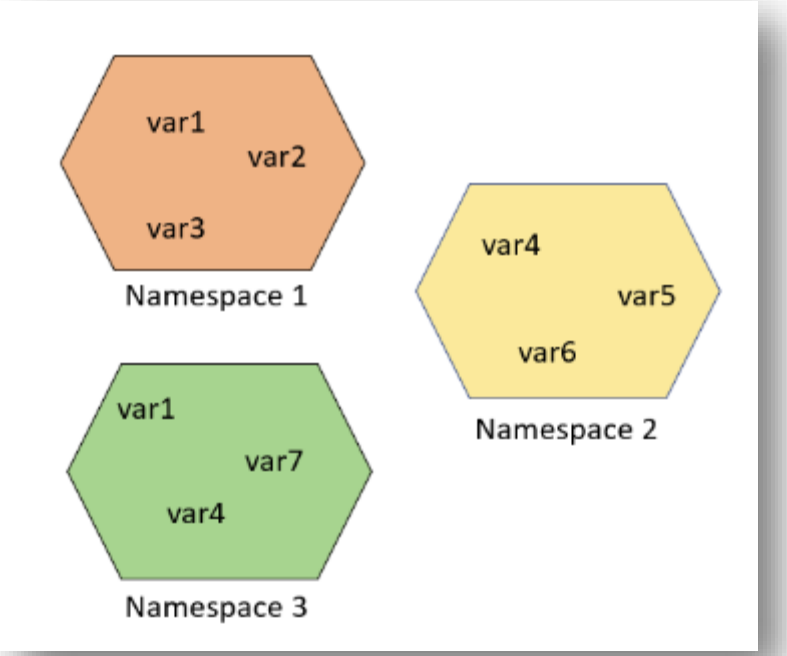
# Ερωτήσεις;

*Τμήματα αυτής της διάλεξης περιέχουν πληροφορίες από πηγές που είναι ελεύθερες στο διαδίκτυο όπως η Βικιπαιδεια και ανοιχτές σημειώσεις παρεμφερών διαλέξεων*

# Παράρτημα

## Περισσότερα για την περιοχή ισχύος μεταβλητής

## Συναρτήσεις εξέτασης Ονοματοχώρων



# Περιοχή ισχύος (scope) μεταβλητής

Η ύπαρξη μεταβλητών με *ίδιο όνομα* που ανήκουν σε *διαφορετικούς ονοματοχώρους* μπορεί να δημιουργήσει *σύγχυση*

Τυπικά, κάποια *ονόματα μεταβλητών* του τοπικού ονοματοχώρου μιας συνάρτησης που εκτελείται μπορεί να είναι *ίδια* με αντίστοιχα του προγράμματος που την κάλεσε, ή μιας άλλης *(προς το παρόν αδρανούς)* συνάρτησης

*Σε κάθε περίπτωση ισχύει ο γενικός κανόνας*: αν το *όνομα* μιας μεταβλητής *χρησιμοποιείται* σε μια συνάρτηση *χωρίς να έχει οριστεί* σε αυτήν, τότε η Python *την θεωρεί εξ ορισμού global* και την ψάχνει στον ονοματοχώρο που περιέχει την συνάρτηση (enclosing namespace)

# Σημαντική διαφοροποίηση της Python

## από άλλες γλώσσες προγραμματισμού

Μεταβλητές που δηλώνονται μέσα σε μια συνάρτηση, χωρίς κάποιο χαρακτηρισμό

- Στην Python:

  Θεωρούνται εξ' ορισμού *local*, *εκτός αν δηλωθούν διαφορετικά*

- Στις περισσότερες άλλες γλώσσες προγραμματισμού (πχ C++, java):

  Θεωρούνται εξ' ορισμού *global*, *εκτός αν δηλωθούν διαφορετικά*

Ο κύριος λόγος πίσω από αυτήν την προσέγγιση είναι ότι οι μεταβλητές global είναι γενικά κακή πρακτική και πρέπει να αποφεύγονται *(γιατί;)*

Στις περισσότερες περιπτώσεις αντί να χρησιμοποιήσουμε μια μεταβλητή *global*, είναι προτιμότερο να χρησιμοποιήσουμε μια *παράμετρο* για να περάσουμε την τιμή της σε μια συνάρτηση ή ένα `return` αν θέλουμε να την επιστρέψουμε

Η Python υλοποιεί αυτή την ανορθόδοξη προσέγγιση προσπαθώντας να ενθαρρύνει αυτή την καλή προγραμματιστική πρακτική

# Συναρτήσεις εξέτασης Ονοματοχώρων

## dir() / locals() / globals()

`dir()`: Αν κληθεί χωρίς παράμετρο, τότε επιστρέφει μια λίστα με τα ονόματα όλων των προσδιοριστών που είναι διαθέσιμοι στον τοπικό ονοματοχώρο

Παράδειγμα: Αμέσως μετά το άνοιγμα του κελύφους (shell) της Python (πχ καλούμε την **dir()**

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__']
```

Στη συνέχεια ορίζουμε μια *μεταβλητή* και μια *συνάρτηση* και καλούμε ξανά την **dir()**

```
>>> a=4
>>> def foo():
  return
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'a', 'foo' ]
```

Δείτε τα ονόματα των ειδικών εσωτερικών μεταβλητών και μεθόδων της Python με διπλή κάτω παύλα

Δείτε τα δυο νέα ονόματα που προστέθηκαν στον ονοματοχώρο

# Συναρτήσεις εξέτασης Ονοματοχώρων

`dir()`: Αν κληθεί με παράμετρο το όνομα ενός δομοστοιχείου (module) ή μιας βιβλιοθήκης, τότε επιστρέφει τα ονόματα όλων των προσδιοριστών που περιέχονται σε αυτό το στοιχείο *(δηλαδή όλα τα ονόματα στον τοπικό ονοματοχώρο του)*

Παράδειγμα:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'tau', 'trunc']
>>>
```

# Συναρτήσεις εξέτασης Ονοματοχώρων

`locals()` / `globals()`: Επιστρέφουν αντίστοιχα ένα λεξικό με περιεχόμενο τα ονόματα του τοπικού / καθολικού ονοματοχώρου με τις αντίστοιχες τιμές τους. Τα περιεχόμενα των λεξικών αυτών δεν πρέπει να τροποποιούνται

Σε συνέχεια του προηγούμενου παραδείγματος

Οι συναρτήσεις `locals()` και `globals()` δεν δέχονται ορίσματα στις παρενθέσεις τους

```
>>> locals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'a': 4,
'foo': <function foo at 0x03151348>, 'math': <module 'math' (built-in)>}
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'a': 4,
'foo': <function foo at 0x03151348>, 'math': <module 'math' (built-in)>}
```

Στο παράδειγμα αυτό, επειδή βρισκόμαστε στο επίπεδο του κελύφους, ο *τοπικός* και ο *καθολικός* ονοματοχώρος *ταυτίζονται*, οπότε και οι δυο συναρτήσεις επιστρέφουν *τα ίδια αποτελέσματα*

# Παράδειγμα χρήσης locals() / globals() (1/2)

```
def foo(p2):
    p1=11
    print(p1, p2)
    print(locals())

a=4
b=8
print(globals())
foo(55)
print(globals())
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, '__file__': 'C:/Users/PYTHON
Programs/locals vs globals.py', 'argv': ['C:Users/PYTHON
Programs/locals vs globals.py'], 'foo': <function foo at
0x036C07C8>, 'a': 4, 'b': 8}
```

```
11 55     <-- αποτέλεσμα του print(p1, p2)
{'p2': 55, 'p1': 11} <-- αποτέλεσμα του print(locals())
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, '__file__': 'C:/Users/PYTHON
Programs/locals vs globals.py', 'argv': ['C:/Users/PYTHON
Programs/locals vs globals.py'], 'foo': <function foo at
0x036C07C8>, 'a': 4, 'b': 8}
```

# Παράδειγμα χρήσης locals() / globals() (2/2)

```
def foo(p2):
    global p1
    p1=11
    print(p1, p2)
    print(locals())


a=4
print(globals())
b=8
foo(55)
print(globals())
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, '__file__': 'C:/Users/PYTHON
Programs/locals vs globals.py', 'argv': ['C:Users/PYTHON
Programs/locals vs globals.py'], 'foo': <function foo at
0x036C07C8>, 'a': 4}
```

```
11 55     <-- αποτέλεσμα του print (p1, p2)
{'p2': 55} <-- αποτέλεσμα του print (locals()) Το p1 δεν εμφανίζεται. (γιατί;)
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, '__file__': 'C:/Users/PYTHON
Programs/locals vs globals.py', 'argv': ['C:/Users/PYTHON
Programs/locals vs globals.py'], 'foo': <function foo at
0x036C07C8>, 'a': 4, 'b': 8, 'p1': 11} Το p1 εμφανίζεται εδώ (γιατί;)
```