Programarea calculatoarelor si limbaje de programare II

Transmiterea Argumentelor si Notiuni Avansate despre Functii

Universitatea Politehnica din Bucureşti

Sumar

- ***Argumente***
- Notiuni avansate

Transmiterea argumentelor

- Argumentele functiilor se transmit prin asignare, deci referinta – pointer.
- Reasignarile argumentelor in functie NU afecteaza apelantul functiei.
- Argumentele imuabile sunt practic transmise prin valoare (ca in limbajul C)
- Argumentele modificabile sunt practic transmise prin pointeri (C).

Referinte partajate in argumente

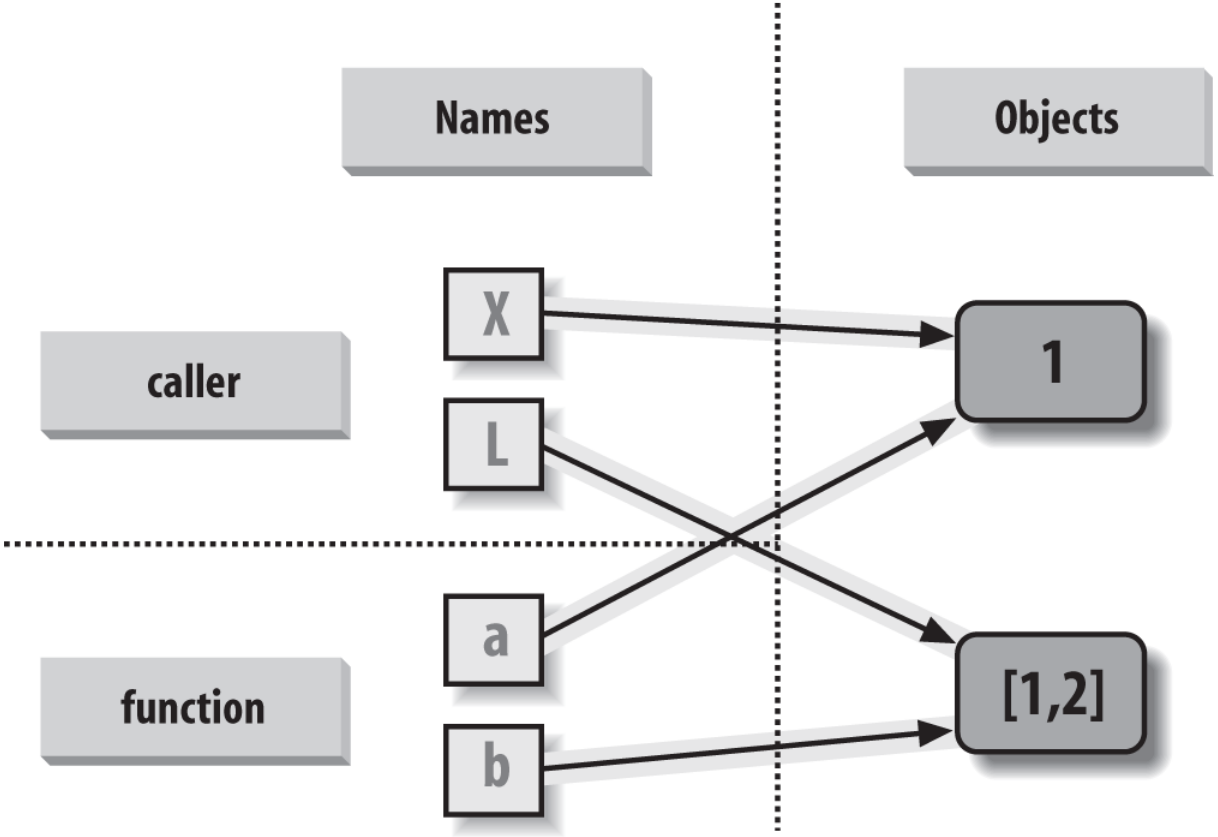
- Reasignarile din functie nu modifica apelantul functiei:

```
>>> def f(a): # a va referi argumentul actual
        a = 99 # a este modificat local
>>> b = 88
>>> f(b) # Aici, a si b refera 88
>>> print(b) # b nu a fost modificat de functie
88
```

- Modificarile in-place ale argumentelor modificabile afecteaza apelantul functiei:

```
>>> def changer(a, b): # Argumentele sunt
    asignate ca referinte la obiecte
        a = 2 # Modificare locala
        b[0] = 'spam' # Modificare obiect
    partajat in-place
>>> X = 1 # Obiect imuabil, int
>>> L = [1, 2] # Obiect modificabil, list
>>> changer(X, L) # Apel functie
>>> X, L # X e nemodificat, L e diferit dupa apel
(1, ['spam', 2])
```

Referinte...



- Asignarile argumentelor actuale permit accesul prin referinte la obiectele transmise de apelantul functiei

Evitarea modificarii argumentelor

- Transmiterea prin referinta economiseste memorie si mareste viteza de executie.
- Copiind obiectele modificabile, fie la apel, fie in functie, se evita afectarea spatiului de nume al apelantului functiei:

```
>>> # Copiere la apel:
>>> L = [1, 2]
>>> changer(X, L[:]) # Copiere prin slicing
```

```
>>> # Copiere in interiorul functiei:
>>> def changer(a, b):
        b = b[:] # Copiere lista prin slicing
        a = 2
        b[0] = 'spam' # Modificare a copiei
```

- Conversia la un obiect imuabil, e.g. *tuple*, forteaza o eroare:

```
>>> L = [1, 2]
>>> changer(X, tuple(L)) # Conversie la tuple
```

Rezultate multiple

- Instructiunea *return* poate returna un *tuple* de obiecte rezultat (chiar si fara paranteze dar cu virgula!) ce pot fi, dupa apel, atribuite prin asignare de *tuple*, astfel simulandu-se rezultate multiple:

```
>>> def multiple(x, y):
        x = 2 # Modificare locala
        y = [3, 4]
        return x, y # tuple rezultat

>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L) # Asignare de tuple
>>> X, L # Variabilele apar modificate
(2, [3, 4])
```

- Observatie: in Python v2.x argumentele pot fi despachetate implicit:

def f((a, (b, c))): ... # *In v2.x, apel cu:* **f((1, (2, 3)))**

def f(T): (a, (b, c)) = T # *In v3.x, despachetare explicita!*

Tipuri de argumente

- Pozitionale, de la stanga la dreapta.
- Cu cuvinte cheie, *nume=valoare,* numele permite argumente in orice ordine.
- Cu valori implicite, *nume=valoare,* cand lipsesc.
- *Varargs* (variable arguments), cu ***** , pentru despachetarea colectiilor in argumente individuale la apel, si pentru colectarea unui numar variabil de argumente in *def*.
- In v3.x, numai cu cuvinte cheie, nu pozitional.

Sintaxa argumentelor

Sintaxa	Loc	Explicatie
func(value)	Apel	Arg. pozitional
func(name=value)	Apel	Arg. cuvant cheie, prin nume
func(*iterable)	Apel	Arg. individuale din iterabil
func(**dict)	Apel	Arg. cheie=valoare din dictionar
def func(name)	def	Arg. normal, pozitional sau prin nume
def func(name=value)	*def*	Arg. implicit, daca lipsa la apel
def func(*name)	*def*	Arg. colectate pozitional in *tuple-ul* name
def func(**name)	*def*	Arg. cu cuvinte cheie colectate in *dict-ul* name
def func(*other, name)	*def*	Arg. numai cu cuvinte cheie in v3.x
def func(*, name=value)	*def*	Arg. numai cu cuvinte cheie in v3.x

Sintaxa...

- La apel:
  - argumentele actuale simple sunt determinate pozitional
  - *name=value* denota determinare prin numele argumentelor (keywords)
  - **iterable* sau ***dict* denota argumente de tip secventa/iterabil sau dictionar care sunt despachetate ca argumente individuale
- In antetul functiei (in *def*):
  - argumentele simple sunt determinate pozitional sau prin nume
  - *name=value* specifica valori implicite, in lipsa
  - **name* colecteaza argumente suplimentare pozitional in tuplul *name*
  - ***name* colecteaza argumente suplimentare cu cuvinte cheie in dictionarul *name*
  - In Python v3.x, argumentele normale sau cu valori implicite care urmeaza dupa **name* sau doar *, trebuie sa fie numai cu cuvinte cheie

Sintaxa...

- O functie cu argumente cu valori implicite permite apelul cu mai putine argumente
- Un apel cu mai multe argumente este posibil, cu * in *def*
- Ordinea argumentelor la apel: pozitionale, cu cuvinte cheie, **iterable*, ***dict*
- Ordinea argumentelor in *def*: normale, cu valori implicite, **name* (sau * in v3.x), numai cu cuvinte cheie *name* sau *name=value* in v3.x, ***name*.
- Python rezolva, la apel, argumentele in 5 pasi:
    1. Asignarea argumentelor poz. care nu sunt cu cuvinte cheie
    2. Asignarea argumentelor cu cuvinte cheie
    3. Asignarea argumentelor simple suplimentare la tuplul *name
    4. Asignarea argumentelor suplimentare cu cuvinte cheie la dict-ul **name
    5. Asignarea valorilor implicite, in lipsa.

    In final, argumentele cu valori multiple produc o eroare.

Argumente, exemple

- Argumente pozitionale:

```
>>> def f(a, b, c): print(a, b, c)
>>> f(1, 2, 3)
1 2 3
```

- Argumente cu cuvinte cheie:

```
>>> f(c=3, b=2, a=1) # Determinare prin nume
1 2 3
>>> # Ordinea nu conteaza
```

```
>>> f(1, c=3, b=2) # Combinatie, intai pozitional
    a = 1, apoi prin nume
1 2 3
```

- Claritate cu cuvinte cheie:

  func(name='Bob', age=40, job='dev')

- Argumente cu valori implicite, in lipsa:

```
>>> def f(a, b=2, c=3): print(a, b, c) # a
    obligatoriu, b si c optionale
>>> f(1) # Valori implicite
1 2 3
>>> f(a=1) #Tot implicit
1 2 3
```

Argumente...

```
>>> f(1, 4) # a si b precizate
1 4 3
>>> f(1, 4, 5) # Toate precizate
1 4 5
```

```
>>> f(1, c=6) # b este sarit, ramane implicit
1 2 6
```

- Argumente cu cuvinte cheie si valori implicite:

```
>>> def func(spam, eggs, toast=0, ham=0): #
    Primele doua argumente obligatorii
        print((spam, eggs, toast, ham))

>>> func(1, 2)
(1, 2, 0, 0)
>>> func(1, ham=1, eggs=0)
(1, 0, 0, 1)
```

```
>>> func(spam=1, eggs=0)
(1, 0, 0, 0)
>>> func(toast=1, eggs=2, spam=3)
(3, 2, 1, 0)
>>> func(1, 2, 3, 4)
(1, 2, 3, 4)
```

Argumente...

- Oricate argumente in *def*:

>>> **def f(*args): print(args)** *# args e tuple ce colecteaza toate arg. actuale, pozitional*	>>> **f()**
	()
	>>> **f(1)**
	(1,)
	>>> **f(1, 2, 3, 4)**
	(1, 2, 3, 4)

>>> **def f(**args): print(args)** *# args e dict ce colecteaza arg. actuale cu cuvinte cheie, nume=valoare*	>>> **f()**
	{}
	>>> **f(a=1, b=2)**
	{'a': 1, 'b': 2}

>>> **def f(a, *pargs, **kargs): print(a, pargs, kargs)** *# Combinatie, pozitionale si cu cuvinte cheie, la urma*	>>> **f(1, 2, 3, x=1, y=2)** 1 (2, 3) {'y': 2, 'x': 1}

Argumente...

- Despachetarea argumentelor la apel:

```
>>> def func(a, b, c, d): print(a, b, c, d)
```

```
>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args) # Ca apelul de func(1, 2, 3, 4)
1 2 3 4
```

```
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
```

```
>>> func(**args) # Ca apelul de func(a=1, b=2, c=3, d=4)
1 2 3 4
```

```
>>> func(*(1, 2), **{'d': 4, 'c': 3}) # Ca apelul de func(1, 2, d=4, c=3)
1 2 3 4
>>> func(1, *(2, 3), **{'d': 4}) # Ca apelul de func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, c=3, *(2,), **{'d': 4}) # Ca apelul de func(1, 2, c=3, d=4)
1 2 3 4
```

```
>>> func(1, *(2, 3), d=4) # Ca apelul de func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, *(2,), c=3, **{'d':4}) # Ca apelul de func(1, 2, c=3, d=4)
1 2 3 4
```

Argumente...

- Apeluri generice:

```
if sometest:
    action, args = func1, (1,) # Apel de func1 cu un argument
else:
    action, args = func2, (1, 2, 3) # Apel de func2 cu 3 argumente
...etc...
action(*args) # Apel generic
```

```
>>> ...func3 este definita sau importata aici...
>>> args = (2,3) # Constructie de argumente pe parcurs
>>> args += (4,)
>>> args
(2, 3, 4)
>>> func3(*args) # Apel generic
```

- Trasarea executiei cu apeluri generice:

```
def tracer(func, *pargs, **kargs): # Orice argumente sunt acceptate
    print('calling:', func.__name__) # Numele functiei, str, atribut predefinit: __name__
    return func(*pargs, **kargs) # Apelul functiei de trasat
```

```
def func(a, b, c, d):
    return a + b + c + d
print(tracer(func, 1, 2, c=3, d=4)) # Output:
calling: func
10
```

Argumente...

- Functia predefinita *apply()* din Python v2.x:

func(*pargs, **kargs) *# v3.x: func(*sequence, **dict)*

apply(func, pargs, kargs) *# v2.x: apply(func, sequence, dict)*

- Argumente **numai** cu cuvinte cheie in Python v3.x:

```
>>> def kwonly(a, *b, c): # c e arg. de transmis cu cuvinte cheie; b tuplu, oricate arg. poz.
        print(a, b, c)
>>> kwonly(1, 2, c=3)
1 (2,) 3
>>> kwonly(a=1, c=3)
1 () 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
```

```
>>> def kwonly(a, *, b, c): # Doar a pozitional, b, c cu cuvinte cheie!
        print(a, b, c)
>>> kwonly(1, c=3, b=2)
1 2 3
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes 1 positional argument but 3 were given
>>> kwonly(1)
TypeError: kwonly() missing 2 required keyword-only arguments: 'b' and 'c'
```

Argumente...

- Argumente **numai** cu cuvinte cheie in Python v3.x si valori implicite:

```
>>> def kwonly(a, *, b='spam', c='ham'): # Doar 3 argumente, b, c cu cuvinte cheie si valori implicite, in lipsa
        print(a, b, c)
>>> kwonly(1)
1 spam ham
>>> kwonly(1, c=3)
1 spam 3
>>> kwonly(a=1)
1 spam ham
>>> kwonly(c=3, b=2, a=1)
1 2 3
```

```
>>> kwonly(1, 2)
TypeError: kwonly() takes 1 positional argument but 2 were given
```

Argumente...

```
>>> # Cu argumente cu cuvinte cheie, obligatorii, b:
>>> def kwonly(a, *, b, c='spam'):
        print(a, b, c)
>>> kwonly(1, b='eggs')
1 eggs spam
>>> kwonly(1, c='eggs')
TypeError: kwonly() missing 1 required keyword-only argument: 'b'
>>> kwonly(1, 2)
TypeError: kwonly() takes 1 positional argument but 2 were given
```

```
>>> # c e obligatoriu, cu cuvinte cheie:
>>> def kwonly(a, *, b=1, c, d=2):
        print(a, b, c, d)
>>> kwonly(3, c=4)
3 1 4 2
>>> kwonly(3, c=4, b=5)
3 5 4 2
>>> kwonly(3)
TypeError: kwonly() missing 1 required keyword-only argument: 'c'
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes 1 positional argument but 3 were given
```

- ***** inseamna numar fix de argumente, dupa cu cuvinte cheie!

Ordonarea Argumentelor

- In *def*, ***args* NU poate fi urmat de argumente neaparat cu cuvinte cheie:

```
>>> def kwonly(a, **pargs, b, c):
SyntaxError: invalid syntax
```

```
>>> def kwonly(a, **, b, c): # Nici ** nu e bine
SyntaxError: invalid syntax
```

- In *def*, *argumentele neaparat cu cuvinte cheie se plaseaza intre *args si **args*:

```
>>> def f(a, *b, **d, c=6): print(a, b, c, d)
SyntaxError: invalid syntax
>>> def f(a, *b, c=6, **d): print(a, b, c, d)
>>> f(1, 2, 3, x=4, y=5) # c este implicit, 6
1 (2, 3) 6 {'y': 5, 'x': 4}
>>> f(1, 2, 3, x=4, y=5, c=7) # c este precizat
1 (2, 3) 7 {'y': 5, 'x': 4}
```

```
>>> f(1, 2, 3, c=7, x=4, y=5)
1 (2, 3) 7 {'y': 5, 'x': 4}
>>> def f(a, c=6, *b, **d): print(a, b, c, d) #
    c NU este cu cuvant cheie, doar al 2-lea
    arg. cu valoare implicita 6
>>> f(1, 2, 3, x=4)
1 (3,) 2 {'x': 4}
```

Ordonarea...

- La apel, argumentele cu cuvinte cheie trebuie puse inainte de ***args*
- Iar fata de **args*, fie inainte, fie dupa sau chiar in ***args*:

```
>>> def f(a, *b, c=6, **d): print(a, b, c, d) # c intre * si **
>>> f(1, *(2, 3), **dict(x=4, y=5)) # Despachetare in arg. individuale + impachetare in tuple si dict
1 (2, 3) 6 {'y': 5, 'x': 4}
>>> f(1, *(2, 3), **dict(x=4, y=5), c=7) # c trebuie sa fie inainte de **args!
SyntaxError: invalid syntax
>>> f(1, *(2, 3), c=7, **dict(x=4, y=5)) # c este precizat
1 (2, 3) 7 {'y': 5, 'x': 4}
>>> f(1, c=7, *(2, 3), **dict(x=4, y=5)) # c este fie inainte (aici) fie dupa *
1 (2, 3) 7 {'y': 5, 'x': 4}
>>> f(1, *(2, 3), **dict(x=4, y=5, c=7)) # c este in **
1 (2, 3) 7 {'y': 5, 'x': 4}
```

Rolul argumentelor neaparat cu cuvinte cheie

- Usureaza definirea functiilor cu numar variabil de argumente pozitionale si optiuni de configurare (e.g. *print()*)

```
>>> def process(*args, notify=False): ...
>>> process(X, Y, Z) # Apel cu 3 argumente,
    notify e False, implicit
>>> process(X, Y, notify=True) # Apel cu 2
    argumente si notify=True, precizat
```

Functia *min()*

- Desi predefinita in Python, cateva variante de implementare:

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res

def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first
```

```
def min3(*args):
    tmp = list(args)
    tmp.sort()
    return tmp[0]

print(min1(3, 4, 1, 2))
print(min2("bb", "aa"))
print(min3([2,2], [1,1], [3,3]))
```

- **Output:**

```
1
aa
[1, 1]
```

minmax()

- Generalizare, calculeaza fie min, fie max:

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y # Pentru min
def grtrthan(x, y): return x > y # Pentru max
```

```
print(minmax(lessthan, 4, 2, 1, 5, 6, 3))
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

- **Output:**

1

6

- Functiile pot fi transmise ca argumente, fiind obiecte in Python

Functii pentru multimi

- Intersectie si reuniune (de secvente):

```
def intersect(*args):
    res = []
    for x in args[0]: # Prima secventa
        if x in res: continue # Fara duplicate!
        for other in args[1:]: # Celelalte secvente
            if x not in other: break # Lipseste!
        else: # Pe for, adauga element
            res.append(x)
    return res
```

```
def union(*args):
    res = []
    for seq in args: # Parcurgere argumente
        for x in seq: # Parcurgere secventa
            if not x in res:
                res.append(x) # Adaugare element
    return res
```

Functii...

```
>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"
>>> intersect(s1, s2), union(s1, s2) # Doua argumente
(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])
>>> intersect([1, 2, 3], (1, 4)) # Date mixte, list si tuple
[1]
>>> intersect(s1, s2, s3) # Trei argumente
['S', 'A', 'M']
>>> union(s1, s2, s3)
['S', 'P', 'A', 'M', 'C', 'L']
```

- Testare cu preschimbarea ordinii secventelor si trasare:

```
>>> def tester(func, *items, trace=True):
        for i in range(len(items)):
            items = items[1:] + items[:1]
            if trace: print(items)
            print(sorted(func(*items)))

>>> tester(intersect, 'a', 'abcdefg', 'abdst', 'albmcnd')
('abcdefg', 'abdst', 'albmcnd', 'a')
['a']
('abdst', 'albmcnd', 'a', 'abcdefg')
['a']
('albmcnd', 'a', 'abcdefg', 'abdst')
['a']
('a', 'abcdefg', 'abdst', 'albmcnd')
['a']
```

Functii...

```
>>> tester(union, 'a', 'abcdefg', 'abdst', 'albmcnd', trace=False)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'l', 'm', 'n', 's', 't']
```

```
>>> tester(intersect, 'ba', 'abcdefg', 'abdst', 'albmcnd', trace=False)
['a', 'b']
['a', 'b']
['a', 'b']
['a', 'b']
```

```
>>> intersect([1, 2, 1, 3], (1, 1, 4)) # Fara duplicate
[1]
>>> union([1, 2, 1, 3], (1, 1, 4))
[1, 2, 3, 4]
```

```
>>> tester(intersect, 'ababa', 'abcdefga', 'aaaab', trace=False)
['a', 'b']
['a', 'b']
['a', 'b']
```

Emularea functiei *print()*

- **In v2.x**: *from __future__ import print_function* **SAU**:

```
import sys
def print3(*args, **kargs):
    sep = kargs.get('sep', ' ') # Valori implicite
    end = kargs.get('end', '\n')
    file = kargs.get('file', sys.stdout)
    flush =  kargs.get('flush', False)
    output = ''
    first = True # Pt. primul args
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
    if flush:
        file.flush()

print3(1, 2, 3)
print3(1, 2, 3, sep='') # fara separator
print3(1, 2, 3, sep='...')
print3(1, [2], (3,), sep='...') # Diverse obiecte
print3(4, 5, 6, sep='', end='') # Fara linie noua
print3(7, 8, 9)
print3() # Linie noua
import sys
print3(1, 2, 3, sep='??', end='.\n',
    file=sys.stderr) # Redirectare la stderr
```

Emularea...

- **Output:**

1 2 3

123

1...2...3

1...[2]...(3,)

4567 8 9

1??2??3.

- **In v3.x**, cu argumente neaparat cu cuvinte cheie:

```
def print3(*args, sep=' ', end='\n',
    file=sys.stdout, flush=False):
    output = ''
    first = True

    for arg in args:
        output += ('' if first else sep) +
str(arg)
        first = False
    file.write(output + end)
    if flush:
        file.flush()
```

```
>>> print3(99, name='bob') # Capteaza
    argumente straine!
TypeError: print3() got an unexpected keyword
    argument 'name'
```

Emularea...

- **In v2.x sau v3.x**, detecteaza optiuni straine:

```
def print3(*args, **kargs):
    sep = kargs.pop('sep', ' ') # kargs e dict!
    end = kargs.pop('end', '\n')
    file = kargs.pop('file', sys.stdout)
    flush = kargs.pop('flush', False)
    if kargs: raise TypeError('extra keywords: %s' % kargs) # Detecteaza cuvinte cheie straine
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
    if flush:
        file.flush()
```

```
>>> print3(99, name='bob')
TypeError: extra keywords: {'name': 'bob'}
```

Importanta cuvintelor cheie

- In programarea cu *tkinter*, standardul Python de GUI:

```
from tkinter import *
widget = Button(text="Press me", command=someFunction)
```

- In apelarea functiilor predefinite:

```
sorted(iterable, *, key=None, reverse=False)
```

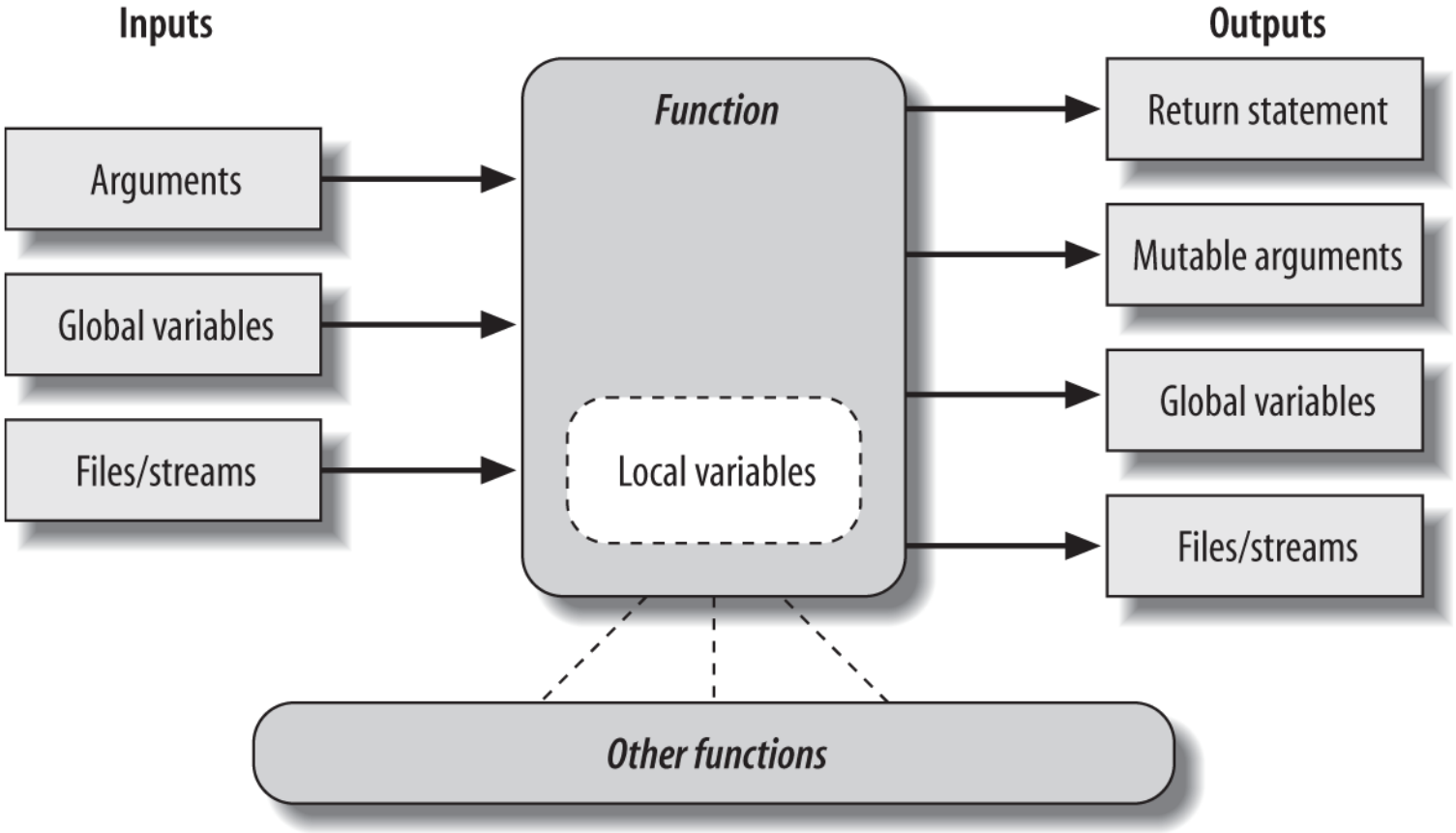
Sumar

- Argumente
- ***Notiuni avansate***

Concepte in proiectarea functiilor

- Interfete bine definite:
  - se folosesc argumente pentru input si *return* pentru outputul functiilor
  - variabilele globale trebuie evitate
  - argumentele modificabile nu trebuie sa fie modificate in-place, decat intentionat
  - modificarea directa a variabilelor din alte module, de evitat
- Coeziune:
  - fiecare functie trebuie sa aiba un singur scop, bine definit
  - codul unei functii trebuie sa fie cat mai scurt – cel mult doua pagini

Concepte...



- Arhitectura functiilor: in v3.x functiile pot modifica variabile si cu *nonlocal*

Recursivitate

- Functiile recursive:
  - se apeleaza pe ele insele, in mod direct (e.g. A->A) sau indirect (e.g. A->B->...->A)
  - simplifica problema de rezolvat
  - codul are macar un punct fix – unde rezultatul este returnat fara alt apel recursiv.
- Permit parcurgerea datelor cu structura arbitrara (e.g. liste, arbori)

```
>>> def mysum(L): # Insumare, recursiva
        if not L:
            return 0 # Punct fix
        return L[0] + mysum(L[1:]) # Apel recursiv, simplifica problema
```

```
>>> mysum([1, 2, 3, 4, 5])
15
```

Recursivitate...

- Versiune cu trasare:

```
>>> def mysum(L):                    [1, 2, 3, 4, 5]
        print(L) # Trasarea executiei [2, 3, 4, 5]
        if not L:                     [3, 4, 5]
            return 0                  [4, 5]
        return L[0] + mysum(L[1:])    [5]
                                      []
>>> mysum([1, 2, 3, 4, 5])           15
```

- Versiuni alternative, care suporta argumente variate:

```
def mysum(L): # Cu expresie ternara
    return 0 if not L else L[0] + mysum(L[1:])

def mysum(L): # Si cu slicing
    return L[0] if len(L) == 1 else L[0] +
           mysum(L[1:])
```

```
def mysum(L): # Si cu asignare extinsa
    first, *rest = L
    return first if not rest else first +
           mysum(rest)
```

Recursivitate...

```
>>> mysum([1]) # mysum([]) nu merge!
1
>>> mysum([1, 2, 3, 4, 5])
15
```

```
>>> mysum(('s', 'p', 'a', 'm')) # Argumente diverse, str
'spam'
>>> mysum(['spam', 'ham', 'eggs'])
'spamhameggs'
```

- Recursivitate indirecta:

```
>>> def mysum(L):
        if not L: return 0 # Punct fix!
        return nonempty(L) # nonempty apeleaza mysum
>>> def nonempty(L):
        return L[0] + mysum(L[1:]) # Recursivitate indirecta
```

```
>>> mysum([1.1, 2.2, 3.3, 4.4])
11.0
```

Recursivitate...

- Cu numar variabil de argumente (pozitionale):

```
>>> def mysum(*pargs):
        zero = None
        def asum( *pargs ): # asum e recursiva!
            nonlocal zero
            if not pargs: return zero
            print( pargs )
            a, *b = pargs
            zero = type( a )() # Element neutru la adunare
            return a + asum( *b )
        return asum( *pargs )
>>> mysum()
>>> mysum( 1, 2, 3 ) # int
(1, 2, 3)
(2, 3)
(3,)
6
>>> mysum( (1,), (2,), (3, 4 ) ) # tuple
((1,), (2,), (3, 4))
((2,), (3, 4))
((3, 4),)
(1, 2, 3, 4)
>>> mysum( 's', 'p', 'a', 'm' ) # str
('s', 'p', 'a', 'm')
('p', 'a', 'm')
('a', 'm')
('m',)
'spam'
```

Iteratie vs. recursivitate

- Parcurgere cu *while*:

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> while L:
        sum += L[0]
        L = L[1:]
>>> sum
15
```

- Iteratie automata cu *for*:

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> for x in L: sum += x
>>> sum
15
```

- De regula, recursivitatea este mai lenta – din cauza apelurilor de functii

Structuri arbitrare

- Subliste, **cu** recursivitate:

```
def sumtree(L):
    tot = 0
    for x in L: # Iteratie pe L
        if not isinstance(x, list):
            tot += x #  Pct. fix, adunare
        else:
            tot += sumtree(x) # Recurenta
    return tot
L = [1, [2, [3, 4], 5], 6, [7, 8]] # list, incluse
print(sumtree(L)) # Afiseaza 36
# Cazuri speciale:
print(sumtree([1, [2, [3, [4, [5]]]]])) # Afiseaza 15
print(sumtree([[[[[1], 2], 3], 4], 5])) # Tot 15
```

- Subliste, **fara** recursivitate, parcurgere pe latime:

```
def sumtree(L): # Coada FIFO
    tot = 0 # Acumulator
    items = list(L) # Copie
    while items:
        front = items.pop(0) # Elem. prim
        if not isinstance(front, list):
            tot += front # Adunare
        else:
            items.extend(front) # La sfarsit, daca lista
    return tot
```

Structuri...

- Subliste, **fara** recursivitate, parcurgere in adancime:

```
def sumtree(L): # Stiva LIFO
    tot = 0 # Acumulator
    items = list(L) # Copie
    while items:
        front = items.pop(0) # Elem. prim
        if not isinstance(front, list):
            tot += front # Adunare
        else:
            items[:0] = front # La inceput, daca lista
    return tot
```

- Controlul stivei de apel recursiv, in Python:

```
>>> import sys
>>> sys.getrecursionlimit() # Implicit 1000 de apeluri
1000
>>> sys.setrecursionlimit(10000) # Marire la 10000
>>> help(sys.setrecursionlimit) # Ajutor despre limitarea numarului de apeluri recursive
```

Apeluri indirecte de functii

- Functiile sunt obiecte in Python:

```
>>> def echo(message): # Functie numita echo
        print(message)
>>> echo('Direct call') # Apel direct
Direct call
>>> x = echo # x devine referinta catre echo()
>>> x('Indirect call!') # Apel indirect cu x()
Indirect call!
```

- Functiile pot fi argumente ale altor apeluri:

```
>>> def indirect(func, arg):
        func(arg) # Apel de argumentul func, cu paranteze ()
>>> indirect(echo, 'Argument call!') # echo este argument pentru functia indirect
Argument call!
```

- Functiile pot fi plasate in alte obiecte:

```
>>> schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
>>> for (func, arg) in schedule:
        func(arg) # Apel functie
Spam!
Ham!
```

Apeluri...

- Fabricile de functii returneaza functii spre apelare ulterioara:

```
>>> def make(label): # Produce pe echo, dar nu o apeleaza
        def echo(message):
            print(label + ':' + message)
        return echo
>>> F = make('Spam') # Argument retinut din domeniul lui make
>>> F('Ham!') # Apelul functiei returnate
Spam:Ham!
>>> F('Eggs!')
Spam:Eggs!
```

Examinarea functiilor

- Functiile sunt obiecte cu operatii definite, e.g. apelul:

```
>>> def func(a):
        b = 'spam'
        return b * a
>>> func(8) # Apel clasic de functie
'spamspamspamspamspamspamspamspam'
```

- Functiile au atribute generice, de sistem:

```
>>> func.__name__ # Nume functie
'func'
>>> dir(func)
['__annotations__', '__call__', '__class__',
    '__closure__', '__code__',
...
'__repr__', '__setattr__', '__sizeof__',
    '__str__', '__subclasshook__']
>>> func.__code__ # Obiect de tip cod
<code object func at 0x0000027F33FB91E0, file
    "<pyshell#79>", line 1>
>>> dir( func.__code__ )
['__class__', '__delattr__', '__dir__', '__doc__',
    '__eq__', '__format__', '__ge__',
...
 'co_argcount', 'co_cellvars', 'co_code',
    'co_name', 'co_names', 'co_nlocals',
    'co_stacksize', 'co_varnames']
>>> func.__code__.co_varnames
('a', 'b')
>>> func.__code__.co_argcount
1
```

Atribute de functii

- Utilizatorul poate atasa atribute functiilor:

```
>>> func
<function func at 0x0000027F33FB94C8>
>>> func.count = 0 # Atributul count
>>> func.count += 1
>>> func.count
1
>>> func.handles = 'Button-Press' # Atributul
    handles
>>> func.handles
'Button-Press'
>>> dir(func)
['__annotations__', '__call__', '__class__',
...
'__subclasshook__', 'count', 'handles']
```

- Numele atributelor de system sunt delimitate de __
- Atributele servesc la memorarea informatiilor de stare per functie.

Adnotarea functiilor in v3.x

- Adnotarile se aplica argumentelor sau/si rezultatului functiei, in antet:

```
>>> def func(a: 'spam', b: (1, 10), c: float) -> int:
        return a + b + c

>>> func(1, 2, 3)
6
```

- Adnotarile sunt colectate in *dict*-ul ***__annotations__***:

```
>>> func.__annotations__ # Pentru rezultat, cheia 'return'
{'a': 'spam', 'b': (1, 10), 'c': <class 'float'>, 'return': <class 'int'>}
>>> for arg in func.__annotations__: # Adnotarile sunt usor de accesat
        print(arg, '=>', func.__annotations__[arg])

a => spam
b => (1, 10)
c => <class 'float'>
return => <class 'int'>
```

- Adnotarile se plaseaza inaitea valorilor implicite ale argumentelor *(=valoare)*
- Adnotarile **nu** sunt permise in expresii *lambda*

lambda, functii anonime

- Sintaxa:

  lambda *argument1*, *argument2*,... *argumentN* **:** *expresie*

- Argumentele pot lipsi; returneaza o functie, fara nume
- *lambda* este o expresie, NU o instructiune (ca *def*)
- *lambda* are corpul format dintr-o **singura** expresie – asemanatoare cu expresia dupa *return* dintr-un *def*

```
>>> def func(x, y, z): return x + y + z
>>> func(2, 3, 4)
9
```

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

- *lambda* accepta argumente cu valori implicite:

```
>>> x = (lambda a="fee", b="fie",
        c="foe": a + b + c)
>>> x("wee")
'weefiefoe
```

lambda...

- *lambda* adauga un spatiu de nume suplimentar (ca *def*)
- *lambda* foloseste regula conturului, LEGB (ca *def*):

```
>>> def knights():
        title = 'Sir'
        action = (lambda x: title + ' ' + x) # title, din knights()
        return action # Returneaza un obiect de tip functie
>>> act = knights()
>>> msg = act('robin') # x este 'robin'
>>> msg # Output
'Sir robin'
>>> act # act este o functie!
<function knights.<locals>.<lambda> at 0x00000000029CA488>
```

Utilizari ale *lambda*

- Desi optionala, lambda se foloseste la crearea tabelelor de salt:

```
L = [lambda x: x ** 2, # Functii inline
     lambda x: x ** 3,
     lambda x: x ** 4] # O lista cu 3 functii

for f in L:
    print(f(2)) # Afiseaza 4, 8, 16
print(L[0](3)) # Afiseaza 9
```

```
def f1(x): return x ** 2 # Cu def
def f2(x): return x ** 3
def f3(x): return x ** 4
L = [f1, f2, f3] # Referire la numele functiilor

for f in L:
    print(f(2)) # Afiseaza 4, 8, 16
print(L[0](3)) # Afiseaza 9
```

- Sau in crearea tabelelor cu *dict*:

```
>>> key = 'got'
>>> {'already': (lambda: 2 + 2),
     'got': (lambda: 2 * 4),
     'one': (lambda: 2 ** 6)}[key]() #
```

Indexare, apoi apel

8

- **Codul inline este corect localizat!**

Utilizari...

- Deoarece instructiunea *if* nu este permisa in *lambda*:

```
if a: b # Instructiunea if
else: c
b if a else c # Expresii echivalente
((a and b) or c)
>>> lower = (lambda x, y: x if x < y else y)
>>> lower('bb', 'aa')
'aa'
>>> lower('aa', 'bb')
'aa'
```

- Iteratiile in *lambda* se pot scrie cu functia *map()* sau colectii iterative:

```
>>> import sys
>>> showall = lambda x:
        list(map(sys.stdout.write, x))
>>> t = showall(['spam\n', 'toast\n'])
spam
toast
>>> showall = lambda x:
        [sys.stdout.write(line) for line in x]
>>> t = showall(('bright\n', 'side\n', 'of\n',
        'life\n')) # t este lista cu nr. car. scrise
bright
side
of
life
```

Utilizari...

- *lambda* suporta si incluziuni:

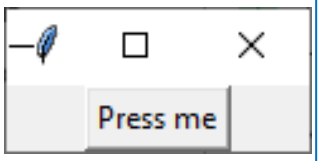
```
>>> action = (lambda x: (lambda y: x + y))
>>> act = action(99)
>>> act(3)
102
```

```
>>> ((lambda x: (lambda y: x + y))(99))(4)
103
```

- *lambda* si GUI:

```
>>> from tkinter import Button, mainloop
>>> x = Button(
        text='Press me',
        command=(lambda:print('Spam\n')))
>>> x.pack()
```

```
>>> mainloop() # La fiecare click...
Spam
Spam
```



- **tkinter GUI API**

Programare functionala

- Cu functia ***map()***, se aplica functii asupra iterabilelor:

```
>>> counters = [1, 2, 3, 4]
>>> updated = []
>>> for x in counters:
        updated.append(x + 10)
>>> updated
[11, 12, 13, 14]
```

- **Cu map:**

```
>>> def inc(x): return x + 10
>>> list(map(inc, counters)) # Se executa inc pt. fiecare element din counters
[11, 12, 13, 14]
>>> list(map((lambda x: x + 10), counters)) # Cu lambda in loc de inc
[11, 12, 13, 14]
```

```
>>> pow(3, 4) # 3**4
81
>>> list(map(pow, [1, 2, 3], [2, 3, 4])) # 1**2, 2**3, 3**4
[1, 8, 81]
```

Cu N secvente, functia aplicata de map trebuie sa aiba N argumente!

Programare...

```
>>> list(map(inc, [1, 2, 3, 4]))
[11, 12, 13, 14]
```

```
>>> [inc(x) for x in [1, 2, 3, 4]] # Colectie echiv.
[11, 12, 13, 14]
```

- Cu functia ***filter()***, se selecteaza din iterabile elementele pentru care o functie de test este *True*:

```
>>> list(range(-5, 5))
[−5, −4, −3, −2, −1, 0, 1, 2, 3, 4]
>>> list(filter((lambda x: x > 0), range(-5, 5)))
[1, 2, 3, 4]
>>> # Cod echivalent:
>>> res = []
>>> for x in range(-5, 5): # Cod echivalent
        if x > 0:
            res.append(x)
```

```
>>> res
[1, 2, 3, 4]
>>> # Colectie iterativa echivalenta:
>>> [x for x in range(-5, 5) if x > 0]
[1, 2, 3, 4]
```

Programare...

- Cu functia ***functools.reduce()*** se combina iterabilele intr-un singur rezultat:

```
>>> from functools import reduce # In v3.x
>>> # Suma:
>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> # Produs:
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

```
>>> # Echivalent, suma,  cu instructiuni:
>>> L = [1,2,3,4]
>>> res = L[0]
>>> for x in L[1:]:
        res +=  x
>>> res
10
```

```
>>> def myreduce(function, sequence): # Cod echiv.
        tally = sequence[0]
        for nxt in sequence[1:]:
            tally = function(tally, nxt)
        return tally
```

```
>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5])
15
>>> myreduce((lambda x, y: x * y), [1, 2, 3, 4, 5])
120
```