Programarea calculatoarelor si limbaje de programare II

Functii

Universitatea Politehnica din Bucureşti

Sumar

- ***Notiuni de baza***
- Domeniul de valabilitate al variabilelor

Definitii

- Functia este folosita la gruparea unui set de instructiuni ce se vor executa intr-un program – prin apelarea numelui functiei.
- Reprezinta alternativa programarii (anevoioase) prin editare cu copy si paste.
- Functiile permit reutilizarea codului si inlesnesc proiectarea si administrarea programelor ample.

Expresii

Instructiune/expresie	Exemple
Apel de functie	myfunc('spam', 'eggs', meat=ham, *rest)
def	def printer(message): print('Hello ' + message)
return	def adder(a, b=1, *c): return a + b + c[0]
global	x = 'old' def changer(): global x; x = 'new'

nonlocal (3.X)	def outer():
	  x = 'old'
	  def changer():
	  nonlocal x; x = 'new'

yield	def squares(x): for i in range(x): yield i ** 2
lambda	funcs = [lambda x: x**2, lambda x: x**3]

Rolul functiilor

- Contribuie la maximizarea gradului de reutilizare a codului si la minimizarea redundantelor.
- Reprezinta un instrument de factorizare a codului, reducandu-se efortul de intretinere in timp a codului.
- Ajuta la impartirea dpdv procedural a codului, permitand abordari bottom/up sau top/down in proiectarea programelor complexe.

Cum se scriu functii noi in Python

- Instructiunea **def** este cod executabil – adica o functie nu exista pana cand instructiunea **def** este executata (e.g. intr-un modul)
- **def** creeaza un obiect de tip functie si il asigneaza numelui functiei (variabila in Python)
- **lambda** creeaza tot un obiect de tip functie dar il si returneaza, putand fi folosita in expresii (e.g. pt. functii *in-line*, unde **def** nu este permis)
- Instructiunea **return** este folosita la returnarea rezultatului functiei catre apelant (implicit **None**)
- Instructiunea **yield** returneaza obiectul rezultat dar si memoreaza starea curenta – permitand generarea rezultatelor, e.g. printr-o iteratie

Cum...

- **global** declara variabile din module ce pot fi asignate in functii (nu doar referite)
- **nonlocal** declara variabile din functii inconjuratoare spre a fi modificate in functia inclusa
- Argumentele functiilor se transmit prin atribuire (referinte catre obiecte); daca argumentele sunt modificabile *in place* atunci pot servi la transmiterea rezultatelor
- Argumentele sunt transmise pozitional, de la stanga spre dreapta; exista si argumente transmise prin nume, e.g. *name=value*, si argumente de tip **pargs* si ***kargs*, cu notatie cu asterix (pt. oricate argumente)

Instructiunea ***def***

- **def** creeaza un obiect de tip functie si il asigneaza unui *nume* (de functie):
- Sintaxa:

```
def nume( arg1, arg2,…, argN ):          #antet, incheiat cu :
    instructiuni                         #corp functie
```

- Corpul functiei poate contine instructiunea **return**:

```
def nume( arg1, arg2,…, argN ):
    …
    return rezultat
```

- Instructiunea **yield** se foloseste la generarea unei serii de valori, in timp.

def este o instructiune executabila

```
if test:
    def func():  # Definitie alternativa a functiei
        ...
else:
    def func():  # Alta definitie
        ...
...
func()          # Apelul functiei func() asa cum a rezultat in urma testului precedent.

altnume = func      #atribuire (referinta)
altnume()           #apelul aceleiasi functii

def func(): …           #crearea obiectului de tip functie
func()                  #apelul functiei
func.atribut = valoare  #atasarea unui atribut arbitrar functiei
```

Exemple

- Definirea functiei (crearea obiectului de tip functie):

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


>>>
```

- Apelul functiei:

```
>>> times(2, 4)
8
>>> x = times(3.14, 4 )  #rezultat memorat in x
>>> x                    #afisare x
12.56
>>> times('Hi', 4)       #rezultat diferit, repetitie de secventa (str)
'HiHiHiHi'
>>>
```

Polimorfism in Python

- Polimorfism: semnificatia unei operatii depinde de obiectele asupra carora este efectuata
- In Python, se recomanda codificarea independenta de tipurile de date, ceea ce conteaza sunt interfetele (operatiile) suportate de obiecte.
- Python detecteaza automat nepotrivirea de interfata.

Intersectie de secvente

```
>>> def intersect(seq1, seq2):
        res = []                    # Rezultat initial vid
        for x in seq1:              # Scanare seq1
            if x in seq2:           # x, element comun ?
                res.append(x)       # Adaugare in lista rezultat, res
        return res
>>> #Apel functie intersect() cu argumente de tip str:
>>> s1 = "SPAM"
>>> s2 = "SCAM"
>>> intersect( s1, s2 )
['S', 'A', 'M']
>>> #Expresie iterativa, echivalenta:
>>> [x for x in s1 if x in s2]
['S', 'A', 'M']
```

Polimorfism

- Functia intersect() este polimorfica:

```
>>> x = intersect([1, 2, 3], (1, 4))  #Argumente mixte, list si tuple
>>> x                                 #Afisare rezultat
[1]
>>>
```

- Primul argument trebuie sa suporte o parcurgere cu *for*, iar cel de-al doilea – operatorul *in*.
- Argumente nesuportate sunt detectate automat, cu exceptii.

Variabile locale

- Sunt variabile al caror nume este recunoscut doar in codul asociat unei functii
- Exista doar atunci cand functia este executata (apelata)
- In mod implicit, orice variabila asignata intr-o functie este locala, de exemplu:
  - *res*, este atribuita
  - *seq1*, *seq2*, findca argumentele se transmit prin atribuire
  - *x*, fiindca este folosit pentru parcurgere de catre instructiunea *for* (este asignat).
- Desi *return* returneaza rezultatul (obiectul *res*), numele *res* este sters.

Sumar

- Notiuni de baza
- ***Domeniul de valabilitate al variabilelor***

Valabilitatea variabilelor in Python

- Variabilele sunt create, modificate sau accesate intr-un *namespace* – context/domeniu in care a avut loc asignarea/atribuirea initiala a variabilei
- Functiile adauga un namespace/domeniu suplimentar:
  - Variabilele asignate intr-un *def* sunt vizibile doar in acel *def* si nu sunt vizibile din afara functiei
  - Variabilele asignate in *def* sunt diferite de variabilele din afara *def*-ului, chiar daca au acelasi nume.
- Asignarea/atribuirea variabilelor se poate face in:
  - intr-un *def*, variabilele fiind locale respectivei functii
  - intr-un *def* exterior, fiind *nonlocal* pentru functiile incluse
  - in afara oricarui *def*, fiind *global* pentru intreg modulul

Valabilitatea...

- Exemplu:

```
>>> X = 99              #Variabila globala, la nivel de modul
>>> def func():
        X = 88          #Variabila diferita, locala functiei
```

- Functiile evita coliziunea de nume de variabile, fiind unitati de program de sine statatoare.

Proprietati ale *Namespace*

- Modulul reprezinta un spatiu de nume global; variabilele globale sunt si atribute ale obiectului de tip modul, in urma importarii modulului.
- Spatiul global este de fapt asociat cu un singur fisier de cod Python.
- Asignarile dintr-o functie sunt locale, cu exceptia variabilelor declarate *global* (exista la nivel de modul) sau *nonlocal* (exista la nivel de functie ce cuprinde alta functie care poate sa o modifice)
- Variabilele neasignate sunt fie locale functiilor inconjuratoare, globale din modul, sau predefinite in modulul built-ins – oferit de Python
- Fiecare apel de functie creeaza un nou spatiu de nume local, chiar si in cazul apelurilor recursive (directe).

Proprietati...

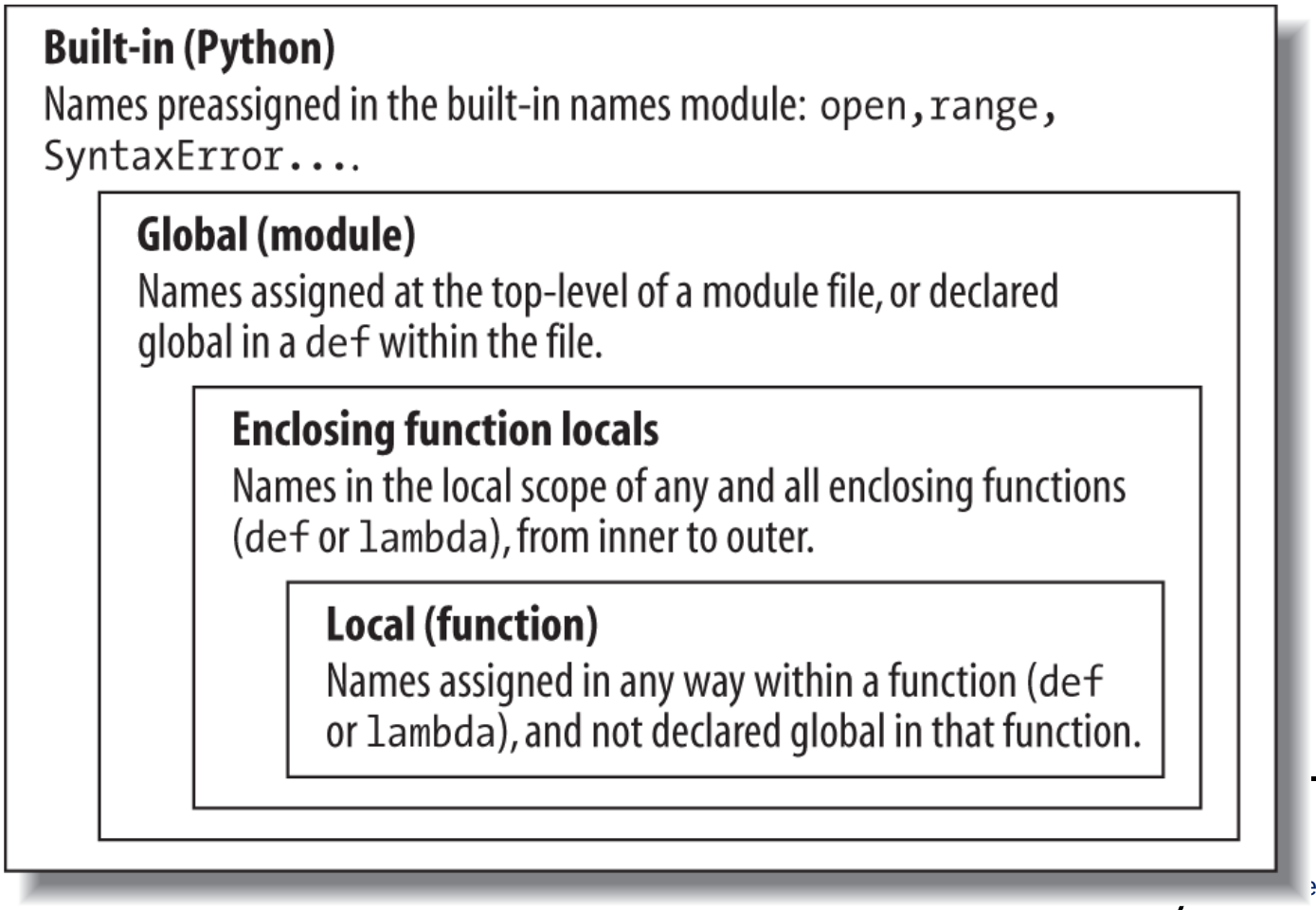
- Codul interactiv (ex. *Idle*) este si el scris intr-un modul (__*main*__), variabilele fiind globale sesiunii interactive
- Asignarile din functii produc variabile locale: cu atribuiri **=,** numele modulelor importate cu *import*, numele functiilor din *def*, argumentele functiilor apelate, etc.
- Modificarile *in-place* nu reprezinta variabile locale! De exemplu, pentru o lista L globala, instructiunea dintr-o functie L.append(X) **nu** transforma L intr-o variabila locala, doar obiectul referit de L este schimbat (L = X face L local!)

Regula LEGB (conturului)

- Variabilele asignate intr-o instructiune *def* (sau *lambda*) sunt implicit locale
- Variabilele referite sunt cautate in patru spatii de nume: local, functii inconjuratoare (daca prezente), global si *built-in* (predefinite de Python)
- Variabilele declarate cu instructiuni *global* sau *nonlocal* sunt asociate modulului respectiv unei functii inconjuratoare.
- Regula LEGB (conturului):
  - Variabilele (fara calificativi) dintr-o functie sunt gasite in patru spatii de nume: local (**L**), functii inconjuratoare (enclosing, **E**), global (**G**), built-in (**B**), in aceasta ordine; daca nu sunt gasite, se produce o eroare.
  - Variabilele asignate intr-o functie sunt create sau modificate local, cu exceptia declararii lor ca global sau nonlocal
  - Variabilele asignate in afara oricarei functii sunt de nivel global, cel al modulului

Regula…

- Regula conturului:



Atentie, variabilele cu calificativi, ex. **object.spam**, sunt atribute de obiecte si **nu** urmeaza regula LEGB, cautarea facandu-se per obiecte si nu spatii de nume!

Alte spatii de nume in Python

- Variabilele din expresii iterative (ex. *[**x** for x in I]*) de tip: generator (), lista [], set {} si dictionar {k:v} – in Python v3.x, sunt locale expresiei
- Variabilele care refera exceptii in blocul *except* sunt locale blocului, e.g. *except E as* ***x*** – in Python v3.x
- Spatiul de nume local unei clase, introdusa de instructiunea *class.* Numele dintr-o clasa sunt locale si reprezinta atribute ale obiectului de tip clasa. Deci clasele nu participa in regula LEGB, numele dintr-o clasa fiind accesate ca atribute de obiect.

Exemple, spatii de nume

```
                    # Nivel global
X = 99              # X si func sunt asignate in modul, deci globale

def func(Y):        # Y si Z sunt asignate in functie, deci locale
    # Spatiu local
    Z = X + Y       # X este o referinta globala
    return Z

func(1)             # apel de func() in modul: rezultat=100
```

- Variabilele locale sunt nume temporare care exista doar cand functia este apelata/executata. Obiectele referite sunt eliberate din memorie de catre *garbage-collector* daca nu mai sunt referite in continuare.
- Distinctia global/local usureaza proiectarea functiilor, usureaza depanarea programelor, functiile fiind unitati software de sine statatoare.

Spatiul de nume ***built-in***

- Este de fapt un modul numit *builtins* ce contine nume predefinite in Python
- Numele predefinite pot fi inspectate dupa importare:

```
>>> import builtins
>>> dir( builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError',
…
'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

- Primele sunt exceptii, celelalte functii predefinite

```
>>> zip                    #acces normal
<class 'zip'>
>>> builtins.zip           #permite redefinirea functiilor predefinite de Python
<class 'zip'>
>>> zip is builtins.zip    #acelasi obiect
True
```

Redefinirea numelor predefinite

- Este permisa de regula LEGB, care gaseste prima asociere a unei variabile, numele globale sau predefinite putand fi inlocuite:

```
def hider():
    open = 'spam'       # Variabila locala, mascheaza functia predefinita open()
    ...
    open('data.txt')    # Eroare: nu se mai deschid fisiere cu open (str) in aceasta functie!
>>> open = 99           # Inlocuire la nivel global, in modulul __main__
```

- Sunt foarte multe nume de functii/exceptii predefinite:

```
>>> len(dir(builtins)), len([x for x in dir(builtins) if not x.startswith('__')])
(154, 146)
```

- Este de obicei o eroare (bug); se poate evita folosind ***PyChecker***
- Redefinirile se pot inlatura cu: **del** *name* din spatiul curent.
- Functiile mascheaza variabilele globale, iar modificarea acestora se poate face numai cu **global/nonlocal** in **def**.
- In Python v2.x se poate **__builtin__.True = False** !

Instructiunea ***global***

- Seamana cu o declaratie, dar precizeaza doar spatiul de nume
- Sintaxa: **global n1, n2,…** cuvantul cheie *global* urmat de o lista de variabile separate cu virgula
- Reguli:
  - Variabilele din *global* sunt asignate in modul.
  - *global* se foloseste numai daca dorim asignarea variabilelor globale din interiorul unui *def*
  - Variabilele globale pot fi referite din functii fara declaratia *global*
- Exemplu:

```
X = 88              # X, variabila globala
def func():
        global X
        X = 99      # X, variabila globala din afara def

func()              # Apel func()
print(X)            # Se afiseaza 99
```

global…

- Alt exemplu, in care variabila x este asignata in functie si creata in spatiul de nume al modulului:

```
y, z = 1, 2          # Variabile globale in modul
def all_global():
    global x         # x din modul, poate fi asignat
    x = y + z        # y si z nu trebuie declarate – regula conturului, LEGB
```

- Se recomanda evitarea folosirii variabilelor globale pentru memorarea informatiilor de stare:

```
X = 99               # Se observa ca valoarea variabilei X depinde de ordinea apelurilor
def func1():         # Comunicarea intre functii se face mai bine prin interfete bazate pe argumente
    global X         # de intrare si rezultatele returnate de fiecare functie
    X = 88


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

global…

- Se recomanda evitarea modificarii variabilelor din alte module/fisiere:

```
# first.py
X = 99              # Existenta fisierului second.py este necunoscuta aici…
# second.py
import first
print(first.X)      # Referirea unui nume din alt fisier este OK
first.X = 88        # Modificarea implicita este periculoasa – dificil de intretinut astfel de cod
# Comunicarea intre module/fisiere se face mai bine cu functii cu argumente si rezultate returnate:
# first.py
X = 99
def setX(new):      # Functia accesor face modificarile externe explicite
    global X        # Accesul este efectuat intr-un singur loc
    X = new
# second.py
import first
first.setX(88)      # Apel de functie in loc de modificare directa!
```

Alte moduri de accesare a variabilelor globale

- Prin importarea modulului, directa sau via *sys.modules['nume']*:

```
# thismod.py
var = 99
def local():
    var = 0 # Variabila locala

def glob1():
    global var # Declaratie globala
    var += 1 # Schimbare globala! (normala)

def glob2():
    var = 0 # Variabila locala
    import thismod # Import pe sine insusi
    thismod.var += 1 # Schimbare globala!

def glob3():
    var = 0 # Modificare locala
    import sys # Import sys (acces la tabelele interpretorului de Python)
    glob = sys.modules['thismod'] # Acces la obiectul modul
    glob.var += 1 # Schimbare globala!

def test():
    print(var)
    local(); glob1(); glob2(); glob3()
    print(var)
>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102
```

Functii incluse

- Regula conturului se extinde:
  - Variabilele referite sunt cautate intai local, **apoi in toate functiile inconjuratoare**, nivelul global si builtins.
  - Variabilele asignate sunt locale, cu exceptia declararii globale – la nivel de modul si cu exceptia declararii *nonlocal* (v3.x) – intr-o functie inconjuratoare.
- Exemplu:

```
X = 99                  # Variabila globala, nefolosita
def f1():
    X = 88              # Variabila locala in functia f1() (inconjuratoare)
    def f2():
        print(X)        # Referinta catre X din f1()
    f2()

f1()                    # Se afiseaza 88
```

Functii…

- Maparea functioneaza chiar si dupa ce functia inconjuratoare si-a incheiat executia:

```
def f1():
        X = 88
        def f2():
                print(X)    # X este cel din f1()
        return f2           # Returneaza referinta catre f2() dar fara sa o apeleze!

action = f1()               # Rezultatul este functia f2()
action()                    # Apel (de f2): se afiseaza 88
```

“Fabrici” de functii

- Reprezinta sabloane de proiectare (*design pattern*)
- Functia inclusa **retine** valori – de stare – din spatiile de nume inconjuratoare, chiar daca acestea nu mai sunt in memorie.
- Se folosesc in proiectarea mediilor GUI (*Graphical User Interface*) in care inputul utilizatorilor nu poate fi anticipat.

```
>>> def maker(N):
        def action(X): # action() este produs de maker()
            return X ** N # action() retine N din spatiul de nume inconjurator
        return action

>>> f = maker(2) # N este 2
>>> f
<function maker.<locals>.action at 0x0000027257956A68>
>>> f(3) # X este 3, N ramane 2: 3 ** 2
9
>>> f(4) # 4 ** 2
16
>>> # g() este o functie noua:
>>> g = maker(3) # g retine 3, f retine 2
>>> g(4)
64
>>> f(4)
16
>>>
```

Tehnici pentru retentia informatiilor de stare intre apeluri

- Cu ajutorul variabilelor globale
- Cu atribute ale instantelor de clase
- Cu referinte in spatiile de nume inconjuratoare
- Cu argumente implicite
- Cu atribute de functii
- Chiar si cu argumente implicite modificabile

Tehnica argumentelor implicite

- Functioneaza in toate versiunile de Python:

```
def f1():
    x = 88
    def f2(y=x): # x din f1() este retinut cu argumentul implicit
        print(y)
    f2() # Apel fara argument – deci cu valoarea implicita

f1() # Afiseaza 88
```

- Acelasi efect se obtine fara functii incluse, mai simplu:

```
>>> def f1():
        x = 88 # x este transmis ca argument, fara functii incluse (nested)
        f2(x) # Referinta catre o functie viitoare este OK
>>> def f2(x):
        print(x) # Mai bine fara incluziune!
>>> f1() # Afiseaza 88
```

Utilizarea expresiei ***lambda***

- *lambda* introduce un spatiu de nume suplimentar, ca *def*:

```
def func():
        x = 4
        action = (lambda n: x ** n) # x este retinut din func()
        return action
x = func()
print( x(2) ) # Afiseaza 16, 4 ** 2
```

- Versiunea cu argumente implicite:

```
>>> def func():
        x = 4
        action = (lambda n, y=x: y ** n) # x este transmis ca argument implicit
        return action

>>> x=func()
>>> x(2) # Apel cu al doilea argument lipsa, rezultat 16
```

Iteratiile necesita argumente implicite

- *def* sau *lambda* aflate intr-un *for* si care retin o variabila modificata de *for*, vor folosi aceeasi valoare, cea din ultima iteratie a *for*-ului.

```
>>> def makeActions():
        acts = []       # Lista de functii, initial vida
        for i in range(5): # Se incearca memorarea fiecarui i
            acts.append(lambda x: i ** x) # Dar se retine doar ultimul i, 4
        return acts

>>> acts = makeActions()
>>> acts[0]
<function makeActions.<locals>.<lambda> at 0x00000272579981F8>
>>> acts[0](2) # Toate sunt 4 ** 2, 4=valoarea ultimului i
16
>>> acts[1](2) # Ar trebui sa fie 1 ** 2 (1)
16
>>> acts[2](2) # Ar trebui sa fie 2 ** 2 (4)
16
>>> acts[4](2) # Numai aici ar trebui sa fie 4 ** 2 (16)
16
```

Iteratiile...

- Cu argumente implicite, care sunt evaluate cand functia inclusa este creata, se reuseste retinerea valorilor curente ale variabilei (nu cele finale):

```
>>> def makeActions():
        acts = []
        for i in range(5):
            acts.append(lambda x, j=i: j ** x) # Se retine i-ul curent
        return acts


>>> acts = makeActions() # Lista de functii
>>> acts[0](2) # 0 ** 2, apel  cu al doilea argument lipsa
0
>>> acts[1](2) # 1 ** 2, apel  cu al doilea argument lipsa
1
>>> acts[2](2) # 2 ** 2, apel  cu al doilea argument lipsa
4
>>> acts[4](2) # 4 ** 2, apel  cu al doilea argument lipsa
16
```

Incluziunea pe oricate nivele

- Functiile pot fi incluse pe oricate nivele, care sunt examinate de la interior catre exterior, pentru identificarea referintelor

```
>>> def f1():
        x = 99
        def f2():
            def f3():
                print(x) # x este gasit in spatiul de nume al functiei f1()
            f3()
        f2()

>>> f1()
99
```

- Astfel de cod, pe multe nivele, este nerecomandat!

Instructiunea ***nonlocal*** in Python v3.x

- *nonlocal* permite atat citirea cat si scrierea variabilelor din functiile inconjuratoare
- Este limitat la functiile inconjuratoare **nu** si la nivel global
- Variabilele declarate *nonlocal* trebuie sa existe deja intr-o functie inconjuratoare
- Simplifica implementarea informatiilor de stare modificabile, **fara** clase cu atribute, mostenire, comportament variabil.
- Comparatie cu *global*:
  - *global* cauta incepand cu modulul, permite asignarea numai in modul, desi cautarea continua in *builtins*
  - *nonlocal* este limitat la functiile inconjuratoare, numele trebuie sa existe, variabilele pot fi modificate; cautarea nu se extinde la global sau builtins.
- Sintaxa:

  nonlocal n1, n2,...

nonlocal...

- Exemplu **fara** *nonlocal*:

```
>>> def tester(start):
        state = start # Referinta fara nonlocal este normala
        def nested(label):
            print(label, state) # Starea din spatiul de nume inconjurator este retinuta
        return nested

>>> F = tester(0)
>>> F('spam')
spam 0
>>> F('ham')
ham 0
```

```
>>> def tester(start):
        state = start
        def nested(label):
            print(label, state)
            state += 1 # Modificare imposibila fara nonlocal
        return nested

>>> F = tester(0)
>>> F('spam') # Eroare raportata la apelul lui nested()
UnboundLocalError: local variable 'state' referenced before assignment
```

nonlocal...

- Exemplu **cu** *nonlocal*:

```
>>> def tester(start):
        state = start # state este diferit per apel
        def nested(label):
            nonlocal state # state este retinut din functia tester()
            print(label, state)
            state += 1 # Modificare permisa de nonlocal
        return nested


>>> F = tester(0)
>>> F('spam')
spam 0
>>> F('ham')
ham 1
>>> F('eggs')
eggs 2
```

```
>>> G = tester(42) # Un nested() nou care incepe de la 42
>>> G('spam')
spam 42
>>> G('eggs')
eggs 43
>>> F('bacon') # F() continua de la 3
bacon 3
```

- **Fiecare apel de tester() are un *state* diferit**

Limitari ale ***nonlocal***

- Variabilele declarate cu *nonlocal* trebuie sa existe deja (intr-un def inconjurator) si NU sunt cautate in modul sau builtins:

```
>>> def tester(start): # Lipseste state
        def nested(label):
            nonlocal state # Trebuie sa existe intr-un def inconjurator!
            state = 0
            print(label, state)
        return nested
SyntaxError: no binding for nonlocal 'state' found
>>> def tester(start): # state este asignat ulterior
        def nested(label):
            global state # Nu trebuie sa existe deja in modul!
            state = 0
            print(label, state)
        return nested

>>> F = tester(0)
>>> F('abc')
abc 0
>>> state # variabila globala creata de nested()
0
>>> spam = 99
>>> def tester(start): # spam nu este vazut
        def nested(label):
            nonlocal spam # Global spam nu este vazut!
            state = 0
            print('Current=', spam)
            spam += 1
        return nested
SyntaxError: no binding for nonlocal 'spam' found
```

Stare cu ***nonlocal,*** numai in v3.x

- Variabilele declarate cu *nonlocal* permit crearea unor informatii de stare modificabile:

```
>>> def tester(start):
        state = start # Un state per apel
        def nested(label):
            nonlocal state # state din tester()
            print(label, state)
            state += 1 # Modificare permisa de nonlocal
        return nested
```

```
>>> F = tester(0)
>>> F('spam') # state vizibil in nested()
spam 0
>>> F.state # state NU este atribut de functie
AttributeError: 'function' object has no attribute 'state'
>>>
```

Stare cu ***global***

- *global* permite o singura copie, partajata, a informatiei de stare, plasata in modul:

```
>>> def tester(start):
        global state # state este in modul
        state = start # modificare/creare in modul
        def nested(label):
            global state # declaratie in toate functiile
            print(label, state)
            state += 1
        return nested
>>> F = tester(0)
>>> F('spam') # state este incrementat per apel
spam 0
>>> F('eggs')
eggs 1
>>> G = tester(42) # state este resetat in modul!
>>> G('toast')
toast 42
>>> G('bacon')
bacon 43
>>> F('ham') # Si F() resimte schimbarea lui state, partajat la nivel global, in modul!
ham 44
>>>
```

Stare cu ***class***

- Clasele isi folosesc atributele pentru memorarea informatiei de stare:

```
>>> class tester: # Cu class, metode
        def __init__(self, start): # Constructor
            self.state = start # salvare state
        def nested(self, label):
            print(label, self.state) # Acces state
            self.state += 1 # Atribut, modificabil
>>> F = tester(0) # Noua instanta, apel de __init__
>>> F.nested('spam') # Argumentul self este implicit
spam 0
>>> F.nested('ham')
ham 1
>>> G = tester(42) # Alta instanta, state diferit
>>> G.nested('toast')
toast 42
>>> G.nested('bacon')
bacon 43
>>> F.nested('eggs') # In F, state este nemodificat
eggs 2
>>> F.state # state este accesibil din afara clasei
3
>>> class tester: # Varianta cu __call__()
        def __init__(self, start):
            self.state = start
        def __call__(self, label): # Intercepteaza
apelul de instanta
            print(label, self.state) # nested este
nenecesar
            self.state += 1
>>> H = tester(99)
>>> H('juice') # Apel de __call__()
juice 99
>>> H('pancakes')
pancakes 100
```

Stare cu atribute de functii

- Atributele de functii sunt atasate direct functiilor; portabil in v3.x si v2.x; acces din afara functiei (ca la clase):

```
>>> def tester(start):
        def nested(label):
            print(label, nested.state) # nested este nume vizibil, din tester
            nested.state += 1 # Modificare de atribut, nu de variabila!
        nested.state = start # Atribut atasat functiei nested(), mai sus creata
        return nested
>>> F = tester(0)
>>> F('spam') # F este nested() cu state atasat
spam 0
>>> F('ham')
ham 1
>>> F.state # Acces din afara functiei la state
2
>>> G = tester(42) # G are state propriu
>>> G('eggs')
eggs 42
>>> F('ham')
ham 2
>>> F.state # state e accesibil si per apel
3
>>> G.state
43
>>> F is G # Obiecte diferite!
False
>>>
```

Stare cu obiecte modificabile

- Obiecte modificabile din spatiul de nume inconjurator pot fi schimbate in-place, fara *nonlocal*:

```
>>> def tester(start):
        def nested(label):
            print(label, state[0]) # Referinta la obiect din tester(), state (mai jos creat)
            state[0] += 1 # Modificare in-place, nu a variabilei state!
        state = [start] # Lista cu un obiect, start
        return nested
```

```
>>> F = tester(0)
>>> F('spam')
spam 0
>>> F('ham')
ham 1
>>>  # Etc...
```

Modificarea functiei predefinite *open()*

```
# In fisierul makeopen.py
import builtins


def makeopen(id):
    original = builtins.open
    def custom(*pargs, **kargs):
        print('Custom open call %r:' % id , pargs, kargs)
        return original(*pargs, **kargs)
    builtins.open = custom
```

>>> **F = open('script2.py')** *# Apel de open din builtins*

>>> **F.read()**

'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'

>>> **from makeopen import makeopen** *# Import makeopen()*

>>> **makeopen('spam')** *# open din builtins este schimbat*

>>> **F = open('script2.py')** *# Apel modificat*

Custom open call 'spam': ('script2.py',) {}

>>> **F.read()**

'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'

>>> **makeopen('eggs')** *# Inca o modificare!*

>>> **F = open('script2.py')** *# Starea fiecareia este memorata*

Custom open call 'eggs': ('script2.py',) {}

Custom open call 'spam': ('script2.py',) {}

>>> **F.read()**

'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'

Modificarea...

```
# Cu clase!
>>> import builtins
>>> class makeopen: # self() este apel de __call__
        def __init__(self, id):
            self.id = id
            self.original = builtins.open
            builtins.open = self
        def __call__(self, *pargs, **kargs):
            print('Custom open call %r:' %
self.id, pargs, kargs)
            return self.original(*pargs, **kargs)


>>> makeopen( 'spam' ) # Modificare open() din
builtins
<__main__.makeopen object at
0x0000021CFA03A9C8>
>>> F = open('script2.py') # Noul open
Custom open call 'spam': ('script2.py',) {}
>>> F.read() # Citire intregul fisier
'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'
>>>
```