

## SEDINTA 3 – FUNCTII ȘI CLASE (OOP)

---

### Lucrul cu liste si dictionare avansat

---

O lista permite operatii precum repetarea si concatenarea. Astfel in Fig.1 putem vedea ca putem crea o lista prin repetarea unei alte liste sau prin concatenarea unor liste.

```
>>> lista = ["element"]*5
>>> print lista
['element', 'element', 'element', 'element', 'element']
>>> lista = ["elemnt"]+["element"]+["element"]
```

Fig.1

Concatenarea si repetitia nu este valabila si pentru dictionare. Astfel in Fig. 2 putem vedea ca incercarea de a repeta un dictionar va ridica o eroare. Similar, daca incercam sa definim doua dictionare cum ar fi dic1 si dic2 si sa le concatenam vedem ca aceasta practica returneaza o eroare. Prin urmare, un dictionar nu poate fi concatenat sau repetat.

```
>>> dictionar = {1:"ceva"}
>>> dictionar*4
```

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    dictionar*4
TypeError: unsupported operand type(s) for *: 'dict' and 'int'
>>>
>>> dic1={1:"ceva"}
>>> dic2={2:"altceva"}
>>> dic1+dic2
```

```
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    dic1+dic2
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
>>>
```

Fig.2

In figurile Fig. 3 si Fig.4 putem vedea doua metode de manipulare a sirurilor de caractere. Fig. 3 presupune ca prelucrarea sirurilor de caractere sa se faca in mod direct printr-o buclare for. Astfel putem crea o lista in care fiecare element este un caracter al sirului de caractere.

```

>>> l=[]
>>> sir = "Sir de caractere"
>>> for e in sir:
>>>     l+=e

>>> l
['S', 'i', 'r', ' ', 'd', 'e', ' ', 'c', 'a', 'r', 'a', 'c',
', 't', 'e', 'r', 'e']
>>>

```

Fig.3

In Fig. 4 putem vedea o alta metoda de manipulare a sirurilor de caractere ce presupune utilizarea metodei de manipulare split. Aceasta metoda imparte default sirul de caractere in cuvinte (dupa simbolul space). Unirea unor elemente ale unei liste in sir de caractere se poate face cu ajutorul unui for.

```

>>> sir="sir de caractere"
>>> lista=sir.split()
>>> noul_sir=""
>>> for elem in lista:
>>>     noul_sir+=elem+" "

>>> print noul_sir
sir de caractere
>>>

```

Fig.4

O alta posibilitate de a cicla prin dictionar sau lista este cu ajutorul while. In Fig. 5 se poate vedea ca se va cicla lista ca o conditie pana cand scrie este egal cu unul din elementele listei.

```

>>> scrie = ""
>>> lista = ["a", "b", "c"]
>>> while scrie not in lista:
>>>     scrie = raw_input("Trebuie sa scrii a sau b sau c:\t")

Trebuie sa scrii a sau b sau c: d
Trebuie sa scrii a sau b sau c: qwr
Trebuie sa scrii a sau b sau c: a
>>> print scrie
a

```

Fig.5

Aceasta practica se poate utiliza si la un dictionar; deci putem folosi un dictionar ca o conditie. While va cicla pana cand variabila cu numele <<scrie>> va fi gasita ca una din cheile dictionarului.

```
>>> scrie = ""
>>> dic={"1":"a"}
>>> while scrie not in dic:
        scrie = raw_input("Trebuie sa scrii 1\t")

Trebuie sa scrii 1      1
>>> print scrie
1
```

Fig.6

## Funcția în Python

Toate programele pe care le-am făcut în ședințele anterioare sunt formate dintr-un șir continuu de sintaxe. Dacă un program este format din câteva linii nu există o complexitate ridicată. Dar dacă am dori să facem ceva ce ar conține o complexitate avansată de mii de linii cu siguranță avem nevoie de funcții. Funcțiile ne permit repetarea unui cod dat de câte ori avem nevoie.

Definirea unei funcții se poate face astfel:

```
def instructiuni():
    #bloc de expresii
```

Cuvântul `def` este obligatoriu și indică definirea unei funcții; `instructiuni` este un cuvânt ce denumește noua funcție; acesta poate fi schimbat. Se poate înlocui cu aproape orice cuvânt și reprezintă numele funcției. Nu uita să pui parantezele și două puncte, acestea fiind vitale pentru definirea unei funcții. În imediată apropiere a definirii numelui funcției trebuie să definim și blocul funcției, adică să-i oferim instrucțiunile de care avem nevoie. Blocul de expresii al funcției este separat prin indentare de restul codului.

Linia de definire a unei funcții și blocul sau formează definiția funcției (eng. *function definition*), și are rolul de a defini funcția fără ca să o ruleze.

Cu siguranță am întâlnit funcții predefinite până acum, cum ar fi `len()` sau `type()`.

Există cuvinte ce nu pot fi nume de funcție deoarece sunt rezervate deja la alte funcții cum ar fi `print` sau `del` sau `len`. Prin urmare, nu pot fi reutilizate. Fig. 5 exemplifică două din cazuri:

```
>>> def del():
SyntaxError: invalid syntax
>>> def print():
SyntaxError: invalid syntax
```

Fig.7

Apelarea se face cu ajutorul numelui functiei urmat de paranteze:

`instructiuni()`

Ne propunem să realizăm meniul unui joc interactiv de X și 0 care să aibă un adversar electronic.

Python permite crearea de funcții particulare care primesc sau nu parametrii, realizează ceva operații cu acești parametrii și returnează sau nu un răspuns. Mai jos regăsim un program ce creează și utilizează o funcție, program care o discutăm ulterior.

```
# Functie meniu
# Demonstreaza utilizarea functiei
# Ion Studentul - 1/26/13

def instructiuni():
    """ Afiseaza instructiunile jocului X și 0. """
    print \
    """
    Bine ati venit La incercarea intelectuala binecunoscuta sub numele de X și 0.
    Aceasta batalie se va da intre om și masina reprezentata de un procesor de
    silicon.

    Poti efectua o mutare prin introducerea unui numar intre 1 - 9, numar ce va fi
    corespondent pozitiei ilustrate in tabel:

    1 | 2 | 3
    ---
    4 | 5 | 6
    ---
    7 | 8 | 9

    Imbarbateaza-te omule, caci ai nevoie!!! bataia incepe!\n
    """

# main
print "Iata instructiunile jocului X și 0:"
instructiuni()
print "Aplelarea se poate face de cate ori e nevoie:"
instructiuni()

raw_input("\n\nApasa <enter> pt a iesi.")
```

Prima linie din blocul funcției este o descriere, numită și *docstring* (sau *documentation string*):

```
""" Afiseaza instructiunile jocului X si 0 """
```

Un docstring este un șir de caractere definit cu trei ghilimele ce are rolul de a informa ce rol are funcția în program sau ce returnează. Aceasta trebuie să fie prima linie din bloc, dar se poate întinde peste mai multe linii. Funcțiile pot funcționa foarte bine fără docstring, dar documentarea poate ajuta colegii sau clientul ce cumpără produsul. Docstring-ul poate apărea ca popout în IDLE sau chiar poate fi apelat, utilizând `instructiuni.func_doc`

```
>>> instructiuni.func_doc
' Afiseaza instructiunile jocului X si 0.'
```

Fig.8

Pentru a face vizibilă apariția acestui popout va trebui să scrieți numele funcției urmat de o paranteză rotundă, apoi să apăsați succesiunea de taste CTRL+\ (backslash). O altă modalitate ar fi să înlocuiți succesiunea de taste cu apelarea din meniul Edit a opțiunii “Show call tip” așa cum se poate observa și în figurile alăturate.

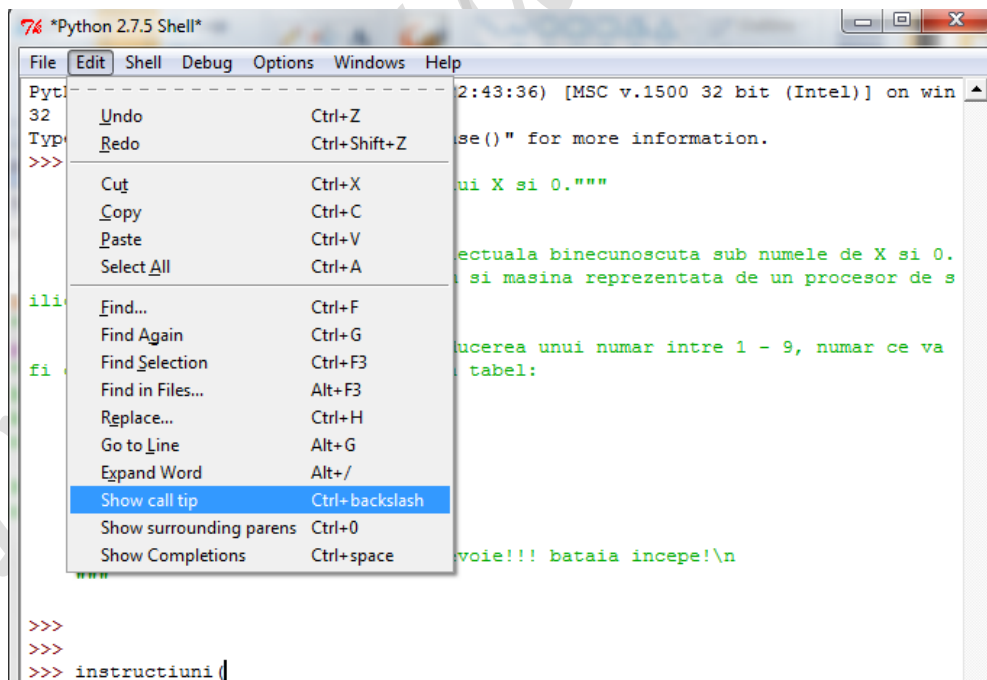
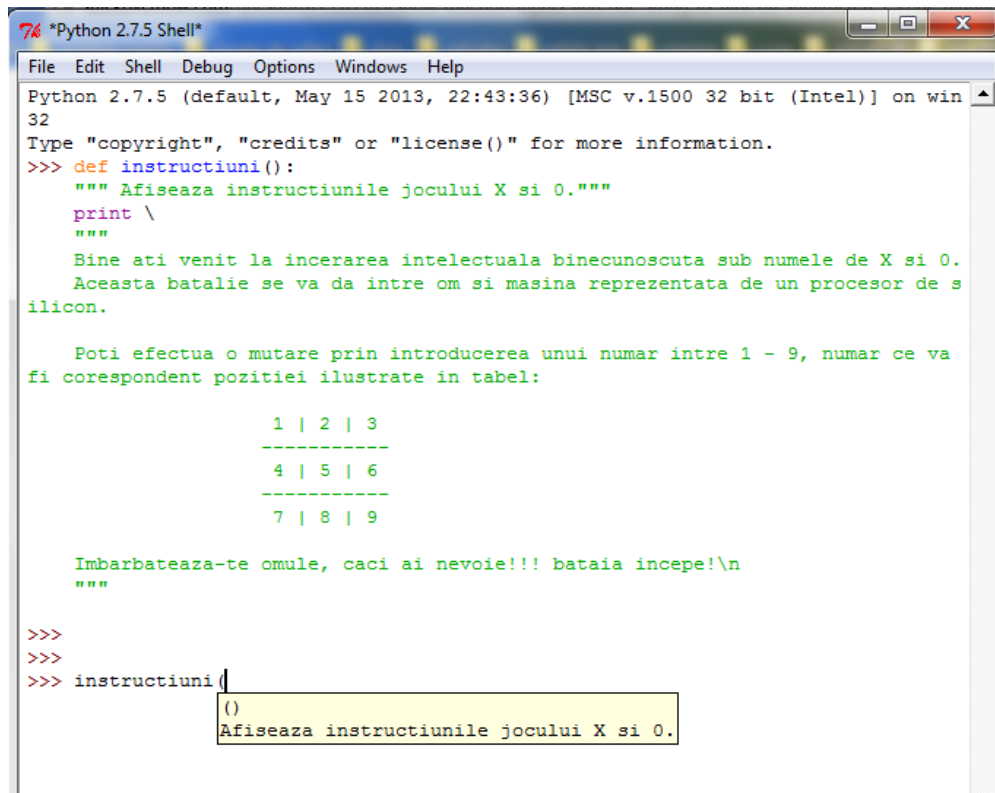


Fig.9



```
Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> def instructiuni():
    """ Afiseaza instructiunile jocului X si 0."""
    print \
    """
    Bine ati venit la incercarea intelectuala binecunoscuta sub numele de X si 0.
    Aceasta batalie se va da intre om si masina reprezentata de un procesor de s
    ilicon.

    Poti efectua o mutare prin introducerea unui numar intre 1 - 9, numar ce va
    fi corespondent pozitiei ilustrate in tabel:

        1 | 2 | 3
        -----
        4 | 5 | 6
        -----
        7 | 8 | 9

    Imbarbateaza-te omule, caci ai nevoie!!! bataia incepe!\n
    """
>>>
>>>
>>> instructiuni()
()
Afiseaza instructiunile jocului X si 0.
```

Fig.10

Apelarea se poate realiza similar cu definirea unei funcții, și anume prin adăugarea numelui funcției urmat de o pereche de paranteze rotunde.

`instructiuni()`

Aceasta modalitate va executa corpul funcției, dar va sari peste docstring fără sa-l afișeze. Corpul funcției noastre este format dintr-o sintaxă de print ce se întinde peste mai multe linii. De fiecare data când noi apelăm funcția interpretorul va rula blocul funcției.

Datorita faptului ca o funcție care nu e definită va genera eroare la apelare, este firesc ca funcțiile să se definească la începutul programului imediat sub secțiunea în care declaram variabilele. Aceasta ordine ar trebui respectata deoarece modificarea programului de către un programator ce nu are cunoștințe despre cod devine greoaie în alte cazuri.

Prin scrierea și apelarea de funcții încercăm să aplicăm și conceptul de abstractizare. Abstractizarea ne lasă să vedem imaginea de ansamblu fără a ne face griji despre detalii. Abstractizarea este un lucru comun în viață de zi cu zi. Spre exemplu, într-un supermarket raioanele sunt numerotate pt. a fi mai rapid de a fi identificat de lucrătorii centrului comercial. E mai facil să spui rândul 19 în loc de rândul cu chei, clești și

ciocane. Un alt exemplu: un mecanic folosește abstractizarea pt. a cere o cheie ucenicului, spre exemplu o cheie numărul 10 pentru un șurub ce are 10 mm partea hexagonală și 6 mm baza fileului. Si lista poate continua. Si mecanicul și ucenicul lui știu ce înseamnă acest lucru astfel conversația își atinge scopul doar cerând o “cheie de 10”. Așa putem crea diferite funcții ce au diferite nume și docstring explicativ. Prin această metoda abstractizăm cum ajungem la această metoda și vedem doar imaginea de ansamblu.

Funcția pe care am prezentat-o anterior este foarte simplă și atinge doar anumite funcționalități pe care le poate face o funcție.

Un alt lucru pe care o funcție poate să-l facă este să aibă parametri de intrare. Parametrii captează valoarea trimisă către funcție și mai poartă numele de argument. O funcție poate avea mulți parametri.

```
>>>
>>> def adunam(param1,param2):
    """In functie de acesti parametrii vom face altceva.
    Spre exemplificare vom crea o functie care aduna doua numere"""
    print param1+param2

>>> adunam(1,2)
3
>>> x=adunam(2,3)
5
```

Fig.11

Astfel parametrii param1 și param2 pot fi utilizați ca orice variabilă, afișând sau modificând valoarea acestora.

Apelarea funcției cu mai mulți parametri sau mai puțini parametri returnează eroare. În corpul erorii putem vedea că adunam() primește exact 2 parametri, condiție neîndeplinită în apelarea funcției adunam() din Fig.12.

```
>>> def adunam(param1,param2):
    """In functie de acesti parametrii vom face altceva.
    Spre exemplificare vom crea o functie care aduna doua numere"""
    print param1+param2

>>> adunam(1)

Traceback (most recent call last):
  File "<pyshell#240>", line 1, in <module>
    adunam(1)
TypeError: adunam() takes exactly 2 arguments (1 given)
>>> adunam(1,2,3)

Traceback (most recent call last):
  File "<pyshell#241>", line 1, in <module>
    adunam(1,2,3)
TypeError: adunam() takes exactly 2 arguments (3 given)
```

Fig. 12

Prin urmare, trebuie sa acordam atentie acestor functii care primesc parametrii de intrare si sa dam exact numarul de parametrii solicitat. In cele ce urmeaza ne dorim sa modificam functia adunam() pentru a permite returnarea adunarii celor doi parametrii.

In Fig. 13 putem vedea ca functia adunam nu returneaza nimic. Numerele pe care noi le-am vazut la apelarea functiei se datoreaza sintaxei print din blocul de sintaxe al functiei adunam.

```
>>>
>>> def adunam(param1,param2):
    """In functie de acesti parametrii vom face altceva.
    Spre exemplificare vom crea o functie care aduna doua numere"""
    print param1+param2

>>> adunam(1,2)
3
>>> x=adunam(2,3)
5
>>> x
>>> print x
None
>>>
```

Fig.13



```

>>> def adunam(param1,param2):
    """In functie de acesti parametrii vom face altceva.
    Spre exemplificare vom crea o functie care aduna doua numere"""
    print param1+param2
    #pentru a returna ceva trebuie sa utilizam cuvantul cheie return
    return param1+param2

>>> x=adunam(2,3)
5
>>> x
5
>>> x+2
7

```

Fig.14

Asa cum se poate vedea si in Fig. 14, cu ajutorul cuvantului return putem returna calculul matematic dintre cele doua numere introduse ca parametrii, calcul ce va fi stocat de variabila x. Aceasta va putea fi utilizata ca orice alta variabila;deci putem sa realizam x+2, rezultat ce ne da 7.

În următorul exemplu vom lua în considerare posibilitatea ca funcția să aibă parametri, adică valori de intrare și rezultate returnate, adică valori de ieșire.

```

# Functie input și output
# Demonstreaza parametrii și valorile returnate
# Ion Studentul - 1/26/13

Nrlist={1:True,2:False,3:False,4:False,5:True,6:False,7:False,8:False,9:False}
raspDaNu=None

def Afiseaza(mesaj):
    """ Afiseaza un mesaj dat."""
    print "Acesta este mesajul: "+str(mesaj).upper()+"!"

def Intreaba_da_nu():
    """ Intreaba da sau nu."""
    raspuns = None
    while raspuns not in ("d", "n"):
        print "Raspunde prin \"d\" sau \"n\"."
        raspuns = raw_input().lower()
    if (raspuns=="d"):
        raspuns="da"
    elif (raspuns=="n"):
        raspuns="nu"
    else:
        pass
    print "Raspunsul ales de tine este "+raspuns

def Returneaza(numar):
    """ Verifica daca exista in este marcata sau nu."""
    if(numar in Nrlist):

```

```

    if(Nrlist[numar]==False):
        return "marcat ca fals"
    else:
        return "marcat ca adevarat"
else:
    return "nemarcat"

#rulare
Afiseaza("Salut Python.\n")

nr = Returneaza(2)
print "Iata ce ne returneaza Returneaza(2):", nr, "\n"

raspuns = Intreaba_da_nu()

raw_input("\n\nApasa <enter> pt a iesi.")

```

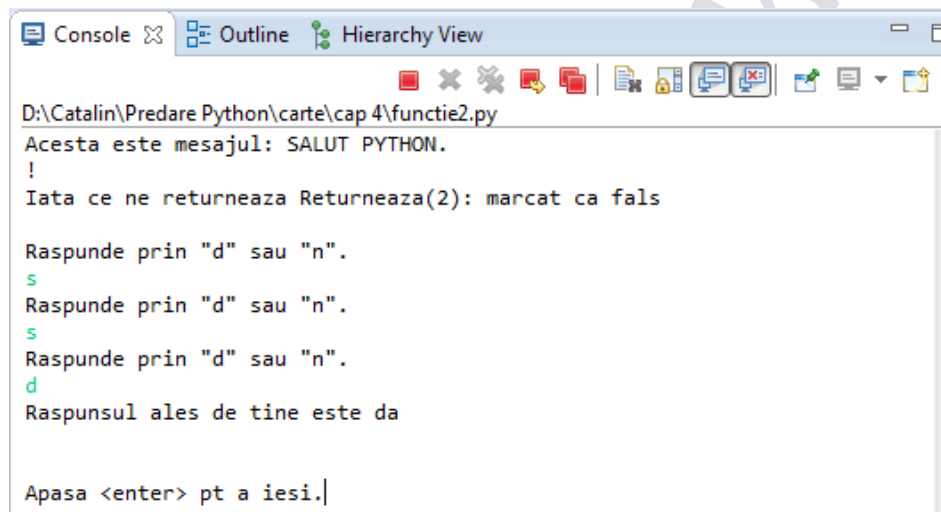


Fig.15

Prima funcție definită în programul de mai sus este funcția Afiseaza(mesaj) ce are rolul de a primi o valoare prin parametrul sau și de a fișă acest mesaj. Parametrii sunt variabilele esențiale și se regăsesc în interiorul parantezelor unei definiții de funcții

```
def Afiseaza(mesaj):
```

Parametrii captează valoarea trimisă către funcție. Astfel variabila locală mesaj poate fi utilizată ca orice variabilă, afișând sau modificând valoarea acesteia.

Atenție am numit aceasta variabilă ca fiind locală (internă) deoarece orice variabilă ce este asignată printr-un parametru nu este disponibilă în afara funcției, ci doar în interiorul acesteia.

```

Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> def Afiseaza(mesaj):
    """ Afiseaza un mesaj dat."""
    print "Acesta este mesajul: "+str(mesaj).upper()+"!"

>>> Afiseaza("ceva")
Acesta este mesajul: CEVA!
>>> print mesaj

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print mesaj
NameError: name 'mesaj' is not defined

```

Fig.16

In fig. 16 incercam sa demonstram ca un parametru nu este disponibil si in exteriorul unei functii. Prin apelarea funcției Afiseaza() aceasta va afisa mesajul dat ca parametru de intrare. Totusi variabila mesaj nu exista dupa terminarea rularii functiei.

Chiar dacă funcția Afiseaza() are un singur parametru , o funcție poate avea multipli parametri separați prin virgula.

Atenție însă, dacă o funcție are patru parametri atunci de fiecare data când este apelata trebuie să aibă patru valori/variabile despărțite cu virgula. Iată un exemplu:

```

>>> def Afiseaza(mesaj,importanta,expeditor, destinatar):
    "Afiseaza un mesaj"
    print "email..."
    print "Catre:",destinatar
    print "De la:",expeditor
    print "Importanta:",importanta
    print "mesaj:",mesaj

>>> Afiseaza("Ma duc la magazin... Iti cumpar ceva si pentru tine?","in asteptare",
"vacina", "vecinu")
email...
Catre: vecinu'
De la: vacina
Importanta: in asteptare
mesaj: Ma duc la magazin... Iti cumpar ceva si pentru tine?
>>> Afiseaza("Nu, multumesc!")

Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    Afiseaza("Nu, multumesc!")
TypeError: Afiseaza() takes exactly 4 arguments (1 given)

```

Fig.15

Se poate vedea în Fig.15 ca în apelarea unei funcții trebuie obligatoriu să adăugăm numărul de parametri definiți de funcție. Altfel ne va returna eroarea `TypeError`.

Următoarea funcție și anume `Intreaba_da_nu()` este o funcție care aparent nu ne transmite nimic nou. Dar totuși să ne uităm cu atenție la această funcție:

```
def Intreaba_da_nu():
    """ Intreaba da sau nu. """
    raspuns = None
    while raspuns not in ("d", "n"):
        print "Raspunde prin \"d\" sau \"n\"."
        raspuns = raw_input().lower()
    if (raspuns=="d"):
        raspuns="da"
    elif (raspuns=="n"):
        raspuns="nu"
    else:
        pass
    print "Raspunsul ales de tine este "+raspuns
```

Funcția își propune să formeze o buclă din care va ieși doar dacă răspunsul dat de utilizator este printre variante. Apoi va formata răspunsul într-o formă mai ușor de citit de utilizator. Acum avem o variabilă ce este definită în funcție. Dorim să vedem dacă după o apelare a funcției putem să utilizăm variabila. Prin urmare, așa cum se poate urmări în Fig.16, variabila locală `raspuns` nu poate fi apelată în exteriorul unei funcții.

```
>>> Intreaba_da_nu()
Raspunde prin "d" sau "n".
d
Raspunsul ales de tine este da
>>>
>>>

>>> print raspuns

Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    print raspuns
NameError: name 'raspuns' is not defined
```

Fig.16

Dacă nu putem folosi variabilele locale atunci cum putem să utilizăm informațiile din interior. Răspunsul este prin returnarea valorilor dorite apelând comanda `return`.

Să ne uităm cu atenție la funcția `Returneaza()`:

```
def Returneaza(numar):
    """ Verifica daca exista in este marcata sau nu. """
    if(numar in Nrlist):
        if(Nrlist[numar]==False):
            return "marcat ca fals"
        else:
            return "marcat ca adevarat"
    else:
        return "nemarcat"
```

```
>>>
```

```
Acesta este mesajul: SALUT PYTHON.
```

```
!
```

```
Iata ce ne returneaza Returneaza(2): marcat ca fals
```

```
Raspunde prin "d" sau "n".
```

```
rasuns
```

```
Raspunde prin "d" sau "n".
```

```
Raspunde prin "d" sau "n".
```

```
n
```

```
Raspunsul ales de tine este nu
```

```
None <= este var raspuns
```

```
Apasa <enter> pt a iesi.
```

```
>>>
```

Fig.17

Funcția returnează are ca parametru variabila locala numar. Aceasta variabila este căutată ca cheie în dicționarul Nrlist. Acest aspect este importând deoarece aici demonstrem ca o funcție poate apela orice variabila definita în exteriorul funcției. Deci dacă variabila <<numar>> stocheaza o valoare care este cheie în dicționarul Nrlist o să verificam valoarea din spatele cheii.

Cuvântul cheie return are scopul de a returna valoarea dată.

Astfel la apelarea funcției Returneaza(2) putem vedea în Fig.17 ca aceasta returnează "marcat ca fals"

Imposibilitatea unei variabile locale să fie accesata din exterior se numește încapsulare și are rolul de tine codul independent. Prin urmare dacă avem o funcție care face sute

de calcule matematice pe baza a 3 parametrii și are nevoie de zeci de variabile locale, dar returnează un singur număr ca răspuns putem izola acele variabile locale, codul rulând mult mai rapid deoarece după terminarea funcției acestea sunt eliminate. Deci dacă avem o sută de funcții cu o sută de variabile locale fiecare programul nu va încarcă decât câte 100 de variabile pe rând neafectând rapiditatea programului.

Încapsularea este un principiu al abstractizării, abstractizare care te salvează de a înțelege toate detaliile, privind în ansambluri ușor de schematizat.

Un alt lucru minunat la funcții este reutilizarea lor în alte programe. Spre exemplu, este un lucru comun să întrebi da sau nu. Putem să reutilizăm această funcție, operație numită reutilizarea software (eng. "software reuse"), economisind timp nu doar în proiectul actual ci și în proiectele viitoare. Acest aspect are ca scop creșterea productivității companiei, aspect dorit de orice manager. Un alt atu al reutilizării software este îmbunătățirea calității produsului deoarece aceste funcții sunt deja testate, fiind bug-free (sau se considera bug-free).

O modalitate de a reutiliza software-ul este copierea funcțiilor în noul program, Dar este și o cale mai eficientă, și anume, de a le importa prin crearea de module. Aceasta a doua metoda va fi învățată într-un capitol viitor.

În următoarea secțiune vom vorbi despre utilizarea parametrilor cu valori standard și attribute cheie. Mai jos putem observa un program care propune aceste funcționalități.

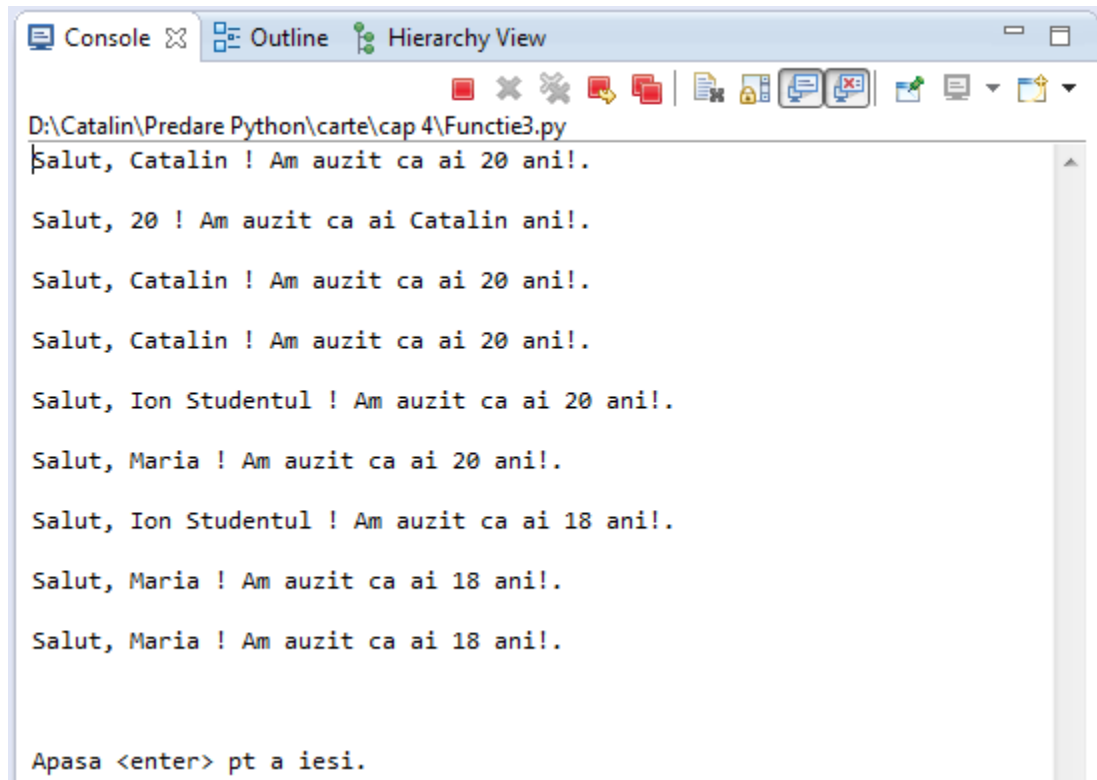
```
# Funcție parametrii default și keyword args
# Demonstrează parametrii standard și attribute cheie
# Ion Studentul - 1/26/13

# parametrii pozitionali
def ziNastere1(name, age):
    print "Salut,", name, "!", "Am auzit ca ai", age, "ani!.\n"

# parametrii cu valori standards
def ziNastere2(name = "Ion Studentul", age = 20):
    print "Salut,", name, "!", "Am auzit ca ai", age, "ani!.\n"
ziNastere1("Catalin", 20)
ziNastere1(20, "Catalin")
ziNastere1(name = "Catalin", age = 20)
ziNastere1(age = 20, name = "Catalin")

ziNastere2()
ziNastere2(name = "Maria")
ziNastere2(age = 18)
ziNastere2(name = "Maria", age = 18)
ziNastere2("Maria", 18)

raw_input("\n\nApasa <enter> pt a iesi.")
```



```

D:\Catalin\Predare Python\carte\cap 4\Functie3.py
Salut, Catalin ! Am auzit ca ai 20 ani!.

Salut, 20 ! Am auzit ca ai Catalin ani!.

Salut, Catalin ! Am auzit ca ai 20 ani!.

Salut, Catalin ! Am auzit ca ai 20 ani!.

Salut, Ion Studentul ! Am auzit ca ai 20 ani!.

Salut, Maria ! Am auzit ca ai 20 ani!.

Salut, Ion Studentul ! Am auzit ca ai 18 ani!.

Salut, Maria ! Am auzit ca ai 18 ani!.

Salut, Maria ! Am auzit ca ai 18 ani!.

Apasa <enter> pt a iesi.

```

Fig.18

Funcția **ziNastere1()** este o funcție ce nu are nimic diferit fata de funcțiile studiate pana în prezent. Astfel prin apelarea acestei funcții vedem ca ordinea parametrilor contează și trebuie să apelam tot mereu în ordinea în care au fost definiți acești parametri. În caz contrar, observam ca în loc de nume regăsim vârsta și în loc de vârsta regăsim numele. Aceasta utilizare folosește parametri poziționali; prin urmare poziția parametrilor în cadrul apelării este importanta.

```

ziNastere1("Catalin", 20)
ziNastere1(20, "Catalin")

```

```

Salut, Catalin ! Am auzit ca ai 20 ani!.

Salut, 20 ! Am auzit ca ai Catalin ani!.

```

Fig.19

În cazul în care nu reținem ordinea parametrilor pe care i-am utilizat putem să folosim parametri ce au cuvinte cheie. Prin aceștia stabilim valorile parametrilor fără a tine cont de ordine.

```
ziNastere1(name = "Catalin", age = 20)
ziNastere1(age = 20, name = "Catalin")
```

```
Salut, Catalin ! Am auzit ca ai 20 ani!.
```

```
Salut, Catalin ! Am auzit ca ai 20 ani!.
```

Fig.19

Nu poți apela o funcție prin combinarea parametrilor poziționali cu parametrii ce conțin cuvinte cheie.

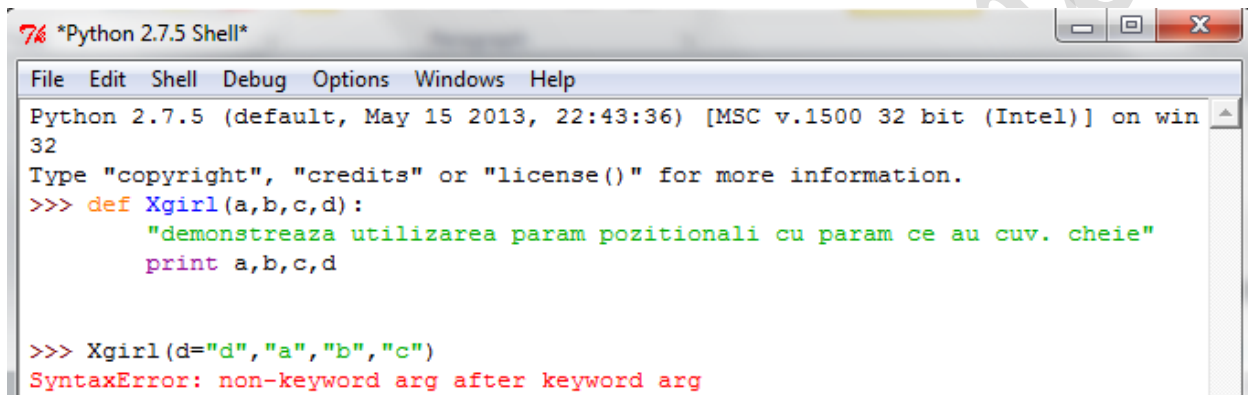


Fig.20

Funcția **ziNastere2()** este o funcție ce stabilește în definirea funcției parametrii standard (default). Prin acești parametrii default putem să ometem la apelarea funcției anumiți parametrii sau chiar pe toți, valorile acestor parametrii omiși vor fi valorile default definite în funcție. Așa cum e normal putem folosi fie parametrii poziționali, fie parametrii ce conțin cuvinte cheie la apelarea funcției.

Prin urmare dacă apelăm funcția **ziNastere2()** fără a avea parametrii name va lua valoarea standard (Ion Studentul) și vârsta standard(20).

La apelarea **ziNastere2()** utilizând un parametru ce conține cuvântul cheie name="Maria"; parametru name va lua valoarea data în apelare (Maria), iar parametru age va lua valoarea standard. Similar și pentru apelarea funcției ziNastere2(age = 18). Vedem ca apelarea ziNastere2(name = "Maria", age = 18) și ziNastere2("Maria", 18) au același rezultat. Si asta datorita faptului ca la ambele apelări valorile standarde au fost rescrise, fie prin parametrii poziționali, fie prin parametrii ce au cuvinte cheie.

```
ziNastere2()
ziNastere2(name = "Maria")
ziNastere2(age = 18)
```



```
ziNastere2(name = "Maria", age = 18)
ziNastere2("Maria", 18)
```

```
Salut, Ion Studentul ! Am auzit ca ai 20 ani!.
```

```
Salut, Maria ! Am auzit ca ai 20 ani!.
```

```
Salut, Ion Studentul ! Am auzit ca ai 18 ani!.
```

```
Salut, Maria ! Am auzit ca ai 18 ani!.
```

```
Salut, Maria ! Am auzit ca ai 18 ani!.
```

Fig.20

Prin magia încapsulării putem să ascundem parametri locali definiți în interiorul funcției de cei globali definiți în exteriorul funcției. Încapsularea se prezintă la funcții ca imposibilitatea de a accesa o variabilă locală din exteriorul funcției.

O soluție la această problemă este utilizarea cuvântului cheie `return` și a parametrilor de intrare. Dar ce se întâmplă dacă trebuie să utilizăm în interiorul funcției 20 variabile. Ar fi destul de ineficient să cream funcții cu 20 de parametri. Am putea utiliza o listă de intrare în care să punem cei 20 de parametri; astfel am putea defini o listă ce are în interior cei 20 de parametri. Dar această practică este greoaie și ar genera multe linii de cod pentru a o rezolva. Avem nevoie de o soluție mai rapidă, soluție prezentată mai jos.

Se definește *namespace* (sau *scope*) anumite zone diferite ale programului care sunt separate una de alta. Aceste zone spre exemplu sunt două funcții distincte, și acesta este motivul pt. care variabilele locale nu pot fi modificate direct (datorită zonelor namespace diferite).

Un namespace este o mapare de la nume de variabile la obiecte. Cele mai multe namespace sunt în prezent puse în aplicare ca dicționare Python. Exemple de namespace sunt : set de nume built-in ( care conțin funcții , cum ar fi `print()` , și nume de excepție built-in ca lista ) , variabile la nivel global sau variabilele locale într-o invocare a unei funcții .

Un obiect este un termen studiat la clase, deci explicat în secțiunea clase a acestui capitol.

Prin urmare, fiecare funcție are namespace-ul propriu, dar și programul în sine are un namespace. O funcție poate vedea toate variabilele din namespace-ul programului (denumite variabile globale), dar și toate variabilele definite în namespace-ul propriu (denumite variabile locale). De asemenea poate modifica doar variabilele definite în namespace-ul propriu.

Vom demonstra accesibilitatea variabilelor din namespace-uri diferite în programul de mai jos. Am adăugat două programe mai jos ce sunt similare, diferența fiind comentarea unei linii. Va rog să facem un exercițiu simplu, va rog să urmăriți cele doua programe și sa-mi spuneți care e diferența și ce ar genera aceasta schimbare?

Programul 1:

```
# Functie ce demonstreaza namespace-urile
# Demonstreaza mespace-urile
# Ion Studentul - 1/26/13

#variabilele globale
nume="Ion Studentul"
fructe=3

print "-----program namespace-----"
# definitii
def AreMere(num):
    """Foloseste paramentrul num și încr variabila globala
    De asemenea se modifica și o variabila locala"""
    print "Functia AreMere:"
    print "Salut, ", num, "!", "Ai ",fructe," mere?."
    print num+" al doilea\n"

def ArePere():
    print "Functia ArePere:"
    """Foloseste nume variabila globala"""
    print "Salut, ", nume, "!\n"

def FaraEroare():
    """Nu încearca să modifice o variabila globala- genereaza eroare"""
    print "Functia FaraEroare:"
    print fructe

def Eroare():
    """Încearca să modifice o variabila globala- genereaza eroare"""
    print "Functia Eroare:"
    print fructe
    fructe+=fructe

#main
AreMere(nume)
ArePere()
FaraEroare()
#Linia cu apelarea functiei Eroare() este comentata
#Eroare()

raw_input("\n\nApasa <enter> pt a iesi.")
```

Programul 2:

```

# Functie ce demonstreaza namespace-urile
# Demonstreaza mespace-urile
# Ion Studentul - 1/26/13

#variabilele globale
nume="Ion Studentul"
fructe=3

print "-----program namespace-----"
# definitii
def AreMere(num):
    """Foloseste paramentru num și încr variabila globala
    De asemenea se modifica și o variabila locala"""
    print "Functia AreMere:"
    print "Salut, ", num, "!", "Ai ", fructe, " mere?."
    print num+" al doilea\n"

def ArePere():
    print "Functia ArePere:"
    """Foloseste nume variabila globala"""
    print "Salut, ", nume, "!\n"

def FaraEroare():
    """Nu incarca să modifice o variabila globala- genereaza eroare"""
    print "Functia FaraEroare:"
    print fructe

def Eroare():
    """Incarca să modifice o variabila globala- genereaza eroare"""
    print "Functia Eroare:"
    print fructe
    fructe+=fructe

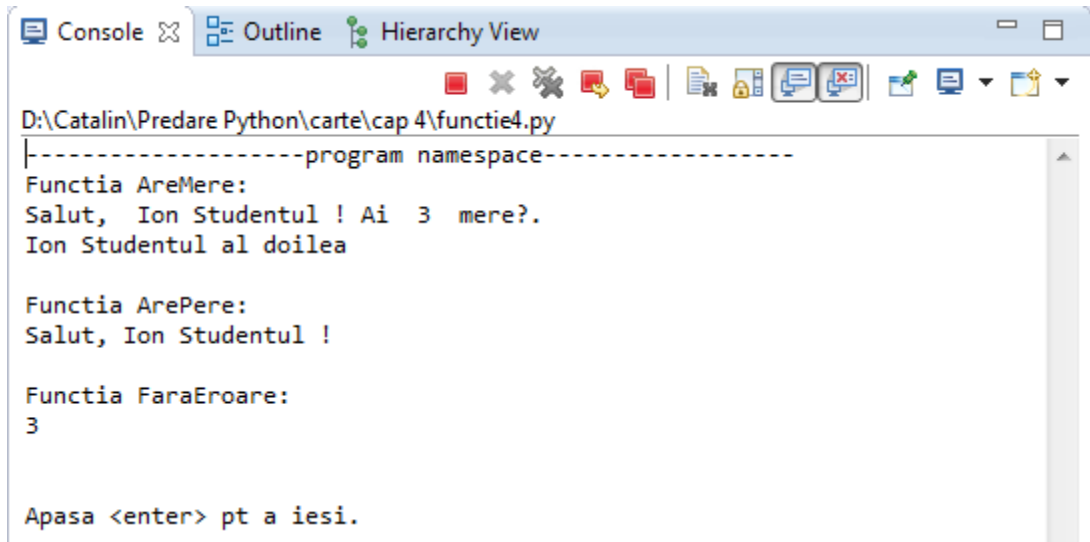
#main
AreMere(nume)
ArePere()
FaraEroare()
#Linia cu apelarea functiei Eroare() nu este comentata
Eroare()

raw_input("\n\nApasa <enter> pt a iesi.")

```

Așa cum era de așteptat, diferența este în comentarea liniei ce rulează funcția Eroare().

Vom regăsi mai jos și cele două rulări ale programului, și anume Fig. 21 afișează rularea programului ce are linia Eroare() comentată, iar Fig. 22 are linia Eroare() necomentată.



```

D:\Catalin\Predare Python\carte\cap 4\functie4.py
-----program namespace-----
Functia AreMere:
Salut, Ion Studentul ! Ai 3 mere?.
Ion Studentul al doilea

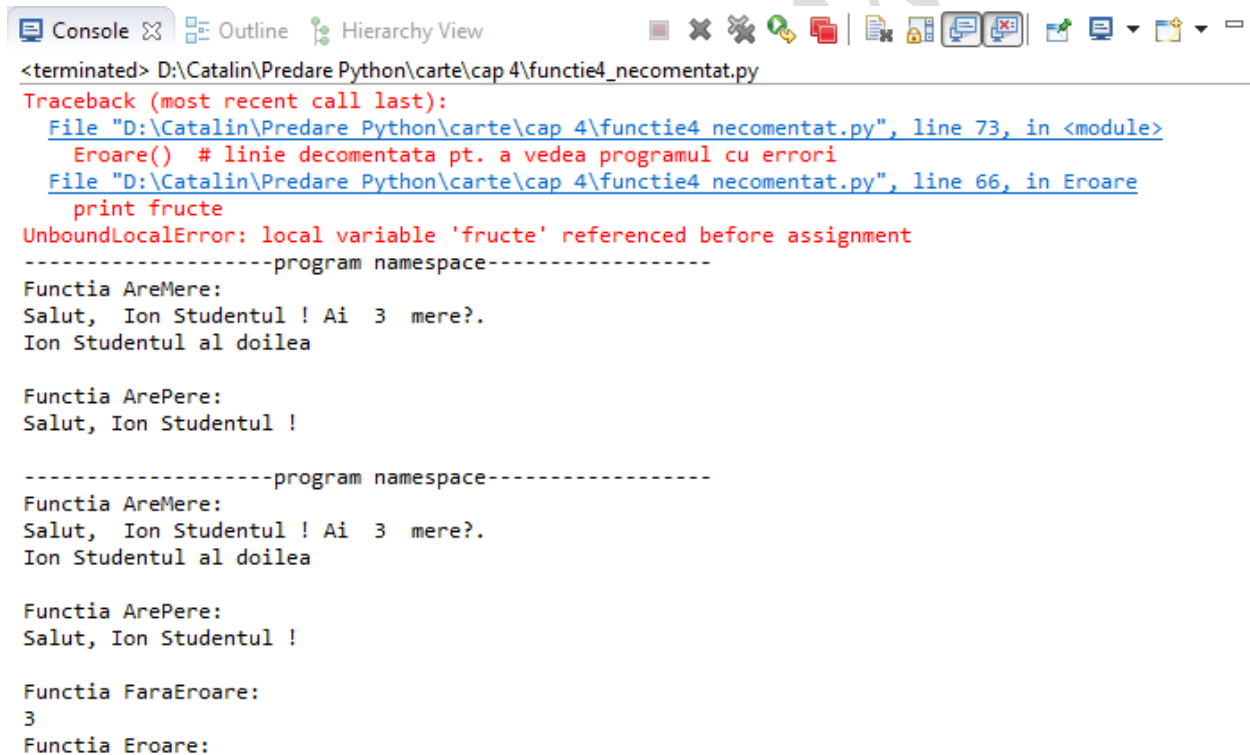
Functia ArePere:
Salut, Ion Studentul !

Functia FaraEroare:
3

Apasa <enter> pt a iesi.

```

Fig.21



```

<terminated> D:\Catalin\Predare Python\carte\cap 4\functie4_necomentat.py
Traceback (most recent call last):
  File "D:\Catalin\Predare Python\carte\cap 4\functie4_necomentat.py", line 73, in <module>
    Eroare() # linie decommentata pt. a vedea programul cu errori
  File "D:\Catalin\Predare Python\carte\cap 4\functie4_necomentat.py", line 66, in Eroare
    print fructe
UnboundLocalError: local variable 'fructe' referenced before assignment
-----program namespace-----
Functia AreMere:
Salut, Ion Studentul ! Ai 3 mere?.
Ion Studentul al doilea

Functia ArePere:
Salut, Ion Studentul !

-----program namespace-----
Functia AreMere:
Salut, Ion Studentul ! Ai 3 mere?.
Ion Studentul al doilea

Functia ArePere:
Salut, Ion Studentul !

Functia FaraEroare:
3
Functia Eroare:

```

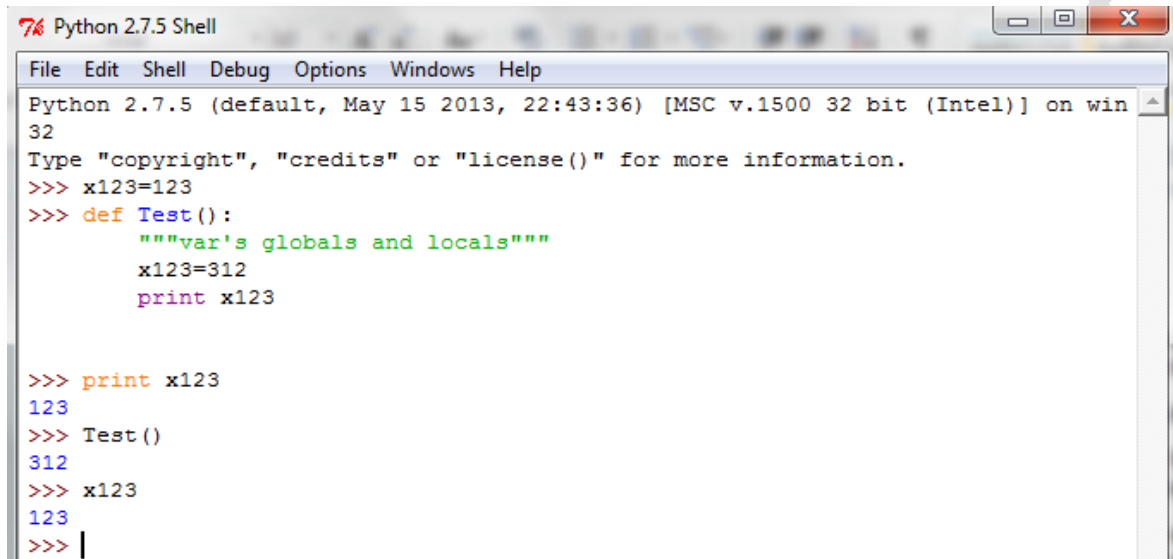
Fig.22

Se poate vedea ca funcția AreMere() poate apela num ca variabila primita prin parametrul num, dar și variabila globala fructe și o poate citi.

Putem observa ca funcția ArePere() poate apela variabila globala nume.

Putem vedea ca funcția FaraEroare() poate apela variabila fructe și să o afișeze, în schimb când funcția Eroare() dorește să modifice o variabila globala va da eroare.

Întrebarea adresată este care e soluția să putem modifica variabile globale fără a folosi return și parametrii? Ce se întâmplă dacă vom denumi cu același nume o variabila globala și una locala?



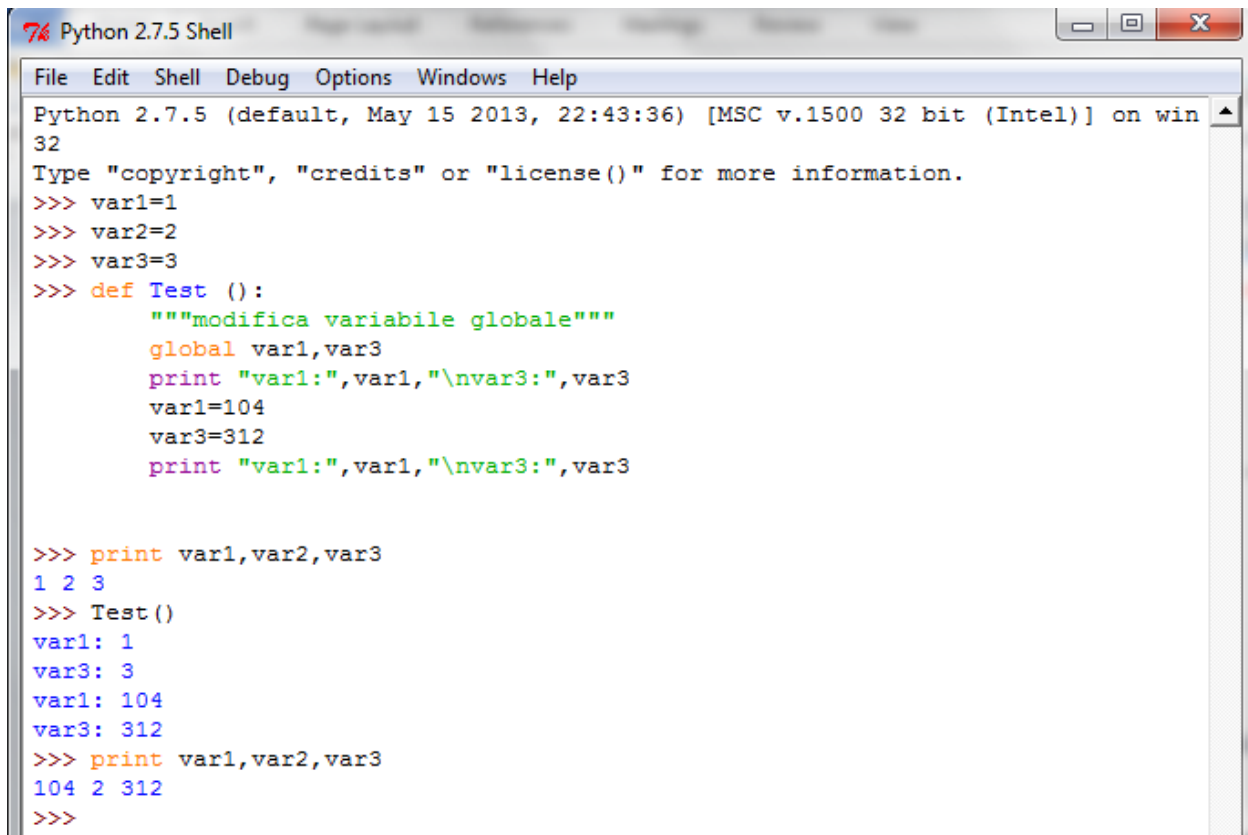
```
Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> x123=123
>>> def Test():
    """var's globals and locals"""
    x123=312
    print x123

>>> print x123
123
>>> Test()
312
>>> x123
123
>>> |
```

Fig.23

In Fig. 24 putem vedea crearea unei variabile locale și una globala cu același nume, dar ele sunt diferite deoarece sunt în namespace-uri diferite. Prin urmare nu putem modifica o variabilă globală creând o variabila locala cu același nume. Se evită crearea de variabile globale și locale care au același nume datorita confuziei pe care o poate crea.

Pentru a rezolva cerința propusă (de a modifica o variabila globală din interiorul unei funcții) vom folosi cuvântul cheie global. Acesta acțiune are efectul scontat. Iată o demonstrație mai jos:



```

Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> var1=1
>>> var2=2
>>> var3=3
>>> def Test ():
    """modifica variabile globale"""
    global var1,var3
    print "var1:",var1,"\nvar3:",var3
    var1=104
    var3=312
    print "var1:",var1,"\nvar3:",var3

>>> print var1,var2,var3
1 2 3
>>> Test()
var1: 1
var3: 3
var1: 104
var3: 312
>>> print var1,var2,var3
104 2 312
>>>

```

Fig.25

Se poate vedea în Fig.25 ca datorita cuvântului cheie “global” putem modifica variabilele var1 și var3. Dacă încercăm să modificăm var2 în cadrul funcției Test() vom genera o eroare deoarece var2 nu se regăsește în linia “global var1,var3” din cadrul funcției. Deci doar variabilele declarate ca fiind globale în interiorul unei funcții pot fi modificate și citite. Dar ce se întâmplă dacă declaram în interiorul unei funcții că o variabila este globala chiar dacă aceasta variabila nu exista? Iata un exemplu mai jos:

```

>>> sss=7
>>> #am definit variabila sss
>>> def test():
    """variabila globala"""
    global ss #am omis un s
    pass

>>> print ss

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    print ss
NameError: name 'ss' is not defined
>>> test()
>>> print ss

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print ss
NameError: name 'ss' is not defined
>>> def test():
    """variabila globala"""
    global ss #am omis un s
    ss=123

>>> print ss

Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    print ss
NameError: name 'ss' is not defined
>>> test()
>>> print ss
123

```

Fig.26

În Fig.26 se poate vedea că am definit variabila `sss` dar în cadrul funcției am omis un `s`, prin urmare variabila este declarată ca `ss`. Chiar dacă această variabilă nu există, funcția nu generează o eroare, ba chiar este inițiată o variabilă nouă numită `ss` ce va deveni globală, iar valoarea ei poate fi utilizată și în namespace-ul global.

În sfârșit am ajuns și la clase, la Object Oriented Programming (OOP).

În cele ce urmează vom discuta despre clasa, obiecte.

Dar de ce am avea nevoie de clase? Pai, va propun un mic exercițiu în care vom utiliza cunoștințele acumulate până în prezent în Python. Dacă avem nevoie de înregistrări ce am putea folosi? Exact, dicționare. Dacă răspunsul ar fi liste este un corect de asemenea. Putem utiliza liste înlănțuite (nested list) sau dicționare care să susțină o listă de atribute în interior.

Totuși aceste soluții au minusurile lor. Vom enumera câteva din ele în cele ce urmează.

În primul rând ar trebui să fim conștienți că dacă dorim ca fiecare intrare să aibă caracteristici diferite una de cealaltă ar trebui să le tratăm diferit. Deci pentru fiecare mod ne trebuie un tratament diferențiat, fapt ce determină un cod mare și ineficient, greu de administrat. Un alt minus este că intrările nu sunt structurate sau încapsulate. Acest fapt determină pentru un număr mare de intrări o încetinire a programului. Va mai amintiți de la funcții că încapsularea ascunde detaliile de care nu avem nevoie pentru ca programul să devină eficient. Deci dacă baza de date va avea 10.000 de intrări programul va fi mai lent decât dacă vom avea 10 intrări.

Aici Python devine atractiv deoarece OOP în Python are caracteristicile necesare pentru ca programul să devină eficient:

- structurare - fiecare din intrări are propriile caracteristici
- încapsulare - izolează informația pentru a putea rula eficient
- scalabilitate - poate face față cu ușurință la schimbările codului- exemplu: adăugarea unui tip special inexistent până în prezent

Înainte de a introduce clasele, trebuie mai întâi să vorbim despre scope-ul Python în ceea ce privește clasele. Putem spune că un set de atribute ale unui obiect formează, de asemenea, un namespace. Deci, încapsularea în cazul claselor ajută ca programul să devină scalabil, nu contează mărimea lui.

Namespace-urile sunt create în momente diferite și durata de viață este diferită. Namespace-ul ce conține numele încapsulate cum ar fi print, len, list etc. este creat când interpretorul pornește programul și nu este șters vreodată de-a lungul programului, acest namespace se găsește în modulul `__builtin__`. Global namespace este creat când interpretorul începe să citească scriptul pentru rulare; acest namespace aparține modulului `__main__`. Chiar dacă nu vom studia modulele în acest capitol, aceste lucruri evidențiază de ce se regăsesc namespace-uri diferite.

Un obiect este o instanță a unei clase. Asta înseamnă că instanțiem clasa creând un obiect, ce poate avea caracteristici proprii. Chiar și o clasă este un obiect deoarece astfel se poate oferi metode de importare a altor clase sau de redenumire.



Fiecare obiect poate avea un set proprietati distincte numite atribute si acestea reprezinta orice numele obiectului urmat de un punct, apoi numele atributului- de exemplu , în expresia obiectulMeu.atributulNostru, atributulNostru este un atribut al obiectului obiectulMeu.

Clasele sunt ușor de utilizat în Python; de fapt vom începe chiar prin a defini o clasa.

Cea mai simpla definiție a unei clase arata astfel:

```
class NumeClasa(object):
    <sintaxa-1>
    .
    .
    .
    < sintaxa-N>
```

Unde NumeClasa poate fi ce nume dorim, iar object este un cuvânt cheie și trebuie folosit ca atare. Putem crea și clase care nu au cuvântul cheie <<object>> fără să dea eroare dar au fost introduse opțiuni aditionale din Python 2.2 pana în prezent pe baza acestui cuvânt. Astfel în Python 2.2.3 a fost introdus tipul object și acesta este motivul real pentru care cuvântul <<object>> este obligatoriu. Așa cum se poate vedea și în Fig.27, interpretorul nu ridică eroare dacă nu utilizăm cuvântul cheie object la definirea unei clase.

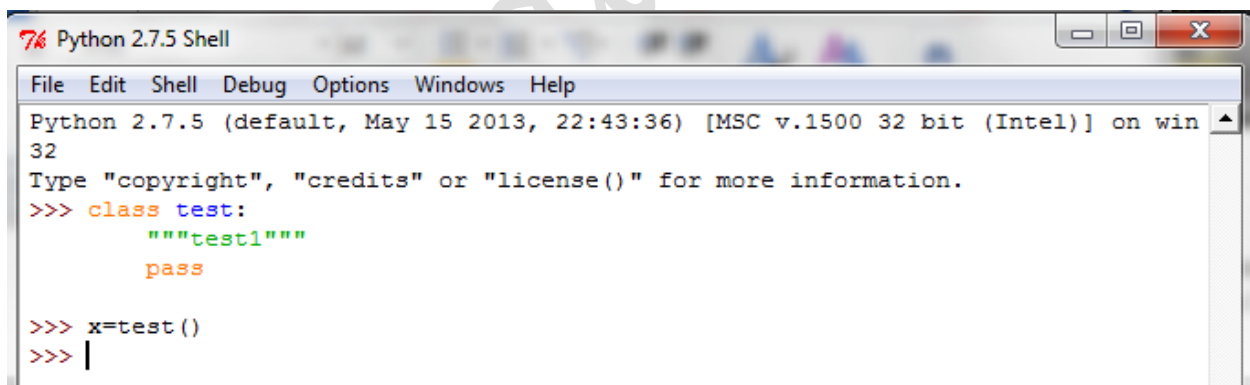


Fig.27

O metoda este o „functie” definita in interiorul unei clase. Aceasta are rolul de a rula un bloc de expresii la apelarea metodei. O metoda este un atribut al unui obiect. Prin urmare, trebuie ca obiectul sa fie creat inainte de utilizarea metodei. Metoda se regaseste la toate obiectele. In Fig. 28 se poate vedea cum cream o metoda care afiseaza sirul de caractere „ceva” la apelare.

```

>>>
>>> class Nume(object):
>>>     """Prima mea clasa"""
>>>     def Metoda(self):
>>>         print "ceva"
>>>
>>> obj1=Nume()
>>> obj1.Metoda()
ceva
>>> |

```

Fig. 28

Sa cream prima noastră clasa în programul de mai jos:

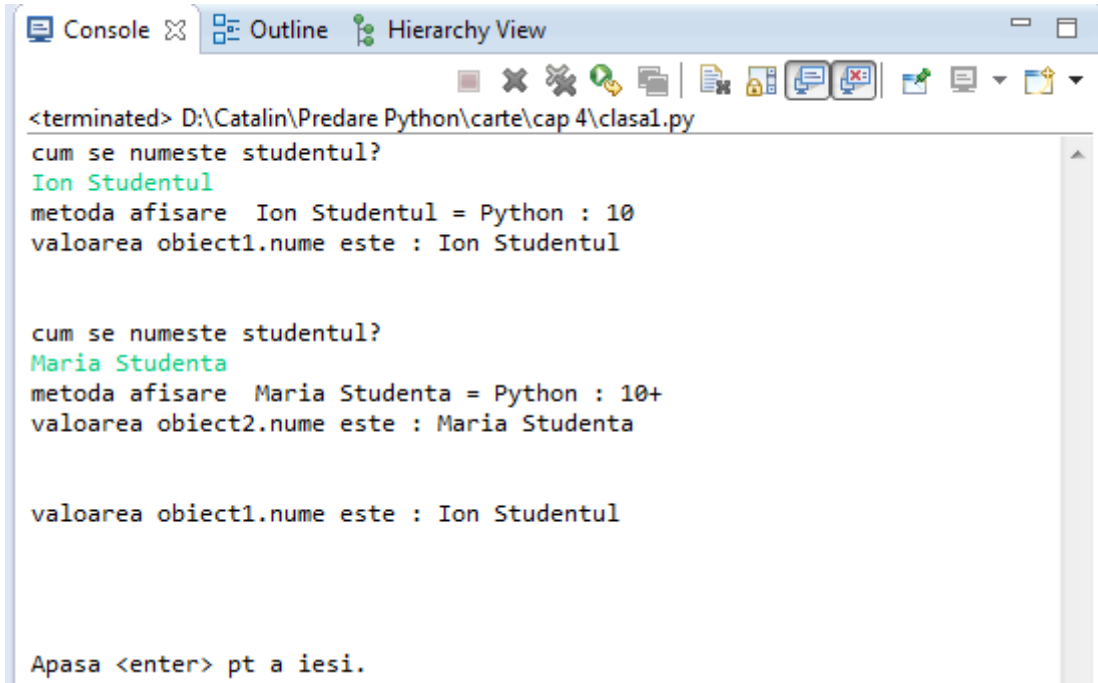
```

1 # Program clasa1
2 # Demonstreaza utilizarea clasei
3 # Ion Studentul - 1/26/13
4
5 class NumeClasa(object):
6     """clasa mea"""
7     def metodaNume(self):
8         """nume student"""
9         self.num = raw_input("cum se numeste studentul?\n")
10    def metodaNotaPython(self,materie,nota):
11        """setare materie/nota"""
12        self.materie=materie
13        self.nota=nota
14    def metodaAfisare(self):
15        """afiseaza toate caracteristicile"""
16        print "metoda afisare\t", self.num,"=",self.materie,":", self.nota
17
18    obiect1=NumeClasa()
19    obiect2=NumeClasa()
20
21    obiect1.metodaNume()
22    obiect1.metodaNotaPython("Python", "10")
23    obiect1.metodaAfisare()
24    print "valoarea obiect1.num este :",obiect1.num,"\n\n"
25
26    obiect2.metodaNume()
27    obiect2.metodaNotaPython("Python", "10+")
28    obiect2.metodaAfisare()
29    print "valoarea obiect2.num este :",obiect2.num,"\n\n"
30
31    print "valoarea obiect1.num este :",obiect1.num,"\n\n"
32
33    raw_input("\n\nApasa <enter> pt a iesi.")

```

Fig.29

Iată și o posibilă rulare a programului:



```
<terminated> D:\Catalin\Predare Python\carte\cap 4\clasa1.py
cum se numeste studentul?
Ion Studentul
metoda afisare Ion Studentul = Python : 10
valoarea obiect1.numa este : Ion Studentul

cum se numeste studentul?
Maria Studenta
metoda afisare Maria Studenta = Python : 10+
valoarea obiect2.numa este : Maria Studenta

valoarea obiect1.numa este : Ion Studentul

Apasa <enter> pt a iesi.
```

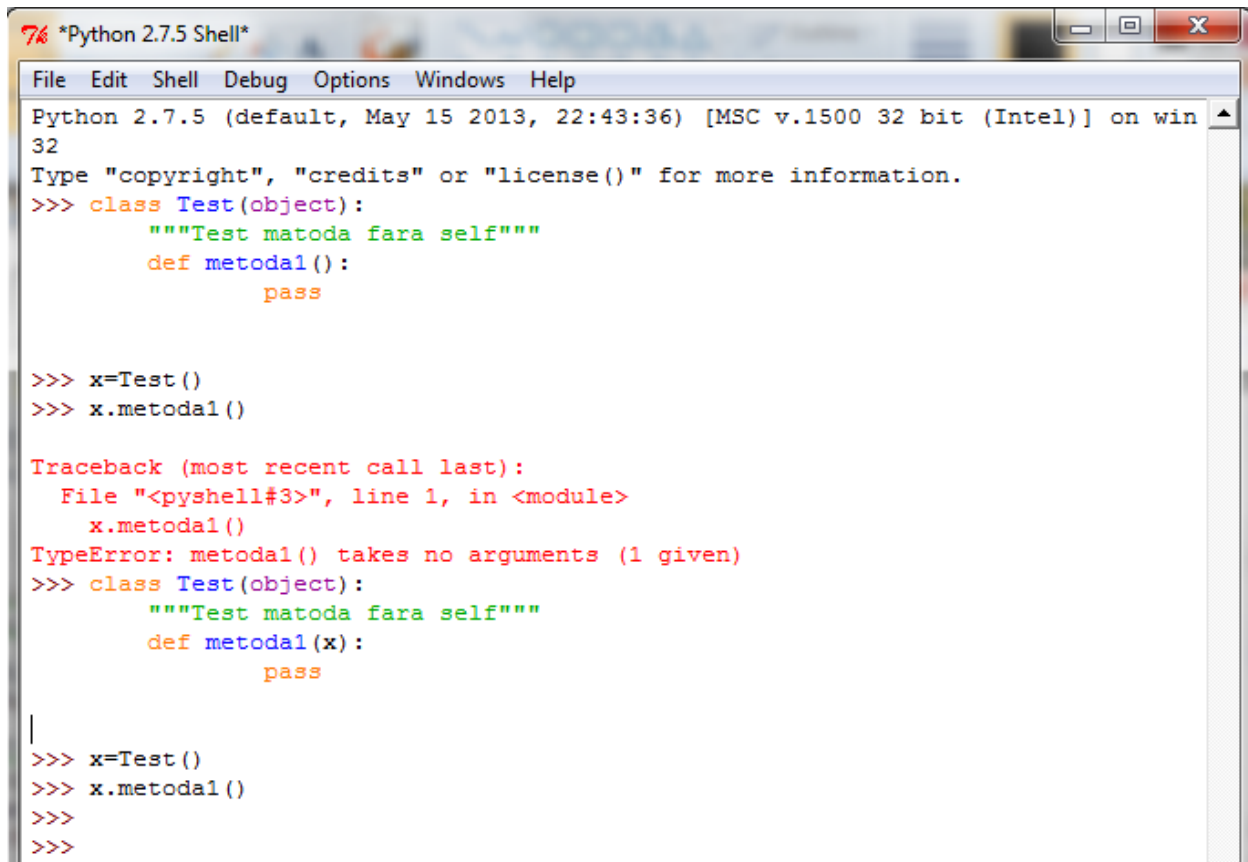
Fig.30

La rândul 5 al programului începem definirea clasei NumeClasa; puteți alege propriul nume, așa cum spuneam mai devreme, în locul numelui NumeClasa.

La linia 6 regăsim un comentariu util pentru ca administrarea programului în caz de eroare să fie mai ușoară.

La rândul 7 s-ar părea ca cream o funcție. De fapt, aceasta funcție se numește metoda. și are rolul de a modulariza acțiunile pe care dorim să le facem în cadrul unei clase. Astfel acțiunile ce generează atribute sau operațiuni cu acestea se pot efectua sau nu, creând flexibilitate și scalabilitate.

Prima metoda pe care o realizăm este metodaNume. Orice metoda trebuie să aibă cuvântul cheie `self` ca parametru pentru ca de fiecare dată când noi apelăm o metoda clasa ce a generat obiectul trimite ca prim parametru un obiect cu toate metodele și variabilele declarate în cadrul funcției. Acest cuvânt cheie `self` este o bună practică-chiar universal valabilă, dar putem folosi ce cuvânt dorim, cu condiția ca în cadrul acelei metode să utilizăm același cuvânt cheie în loc de `self`. Considerăm improprie această practică deoarece toți programatorii Python folosesc cuvântul `self`. Prin urmare, vă recomand să folosim cuvântul `self` ca prim parametru în crearea unei metode. În fig. 30 se poate vedea că am creat o clasă numită `Test`. Apoi cream un obiect numit `x`. Dacă încercăm să apelăm metoda `metoda1` prin obiectul `x` vedem că va genera o eroare datorită lipsei unei variabile. Dacă redefinim clasa `Test` și adăugăm la metoda orice variabilă aceasta va putea funcționa.



```

Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> class Test(object):
    """Test metoda fara self"""
    def metodal():
        pass

>>> x=Test()
>>> x.metodal()

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    x.metodal()
TypeError: metodal() takes no arguments (1 given)
>>> class Test(object):
    """Test metoda fara self"""
    def metodal(x):
        pass

>>> x=Test()
>>> x.metodal()
>>>
>>>

```

Fig.31

Fiecare metoda ar trebui să fie însoțită de o descriere, fapt ce se regăsește în rândul al 8-lea din Fig. 29.

La rândul al 9-lea se poate vedea primul atribut pe care îl cream. Orice atribut creat trebuie să conțină ca prim atribut "self" pentru ca atributul să poată fi apelat din cadrul clasei. Va amintiți ca la inițializarea unei funcții cream un namespace nou ce încapsulează variabilele locale? E bine acest lucru se întâmplă și în cadrul unei clase. Totuși diferă metoda de transmitere a variabilelor locale. Nu ar fi o soluție dacă am folosi cuvântul cheie global? Nu, deoarece nu am mai avea încapsularea de care avem nevoie, totul devenind global. Deci soluția este să cream la obiectul principal al clasei self atribute ce pot fi accesate direct, nu doar în cadrul metodei. Acest lucru îl voi demonstra într-o secțiune imediat următoare. Valoarea self.name este obținută de la tastatura, așa cum se poate vedea și în Fig.30. unde a fost tastat, spre exemplu "Ion Studentul".

Linia a zecea creează o noua metoda. Exact ca și o funcție, o metoda poate avea mai mulți parametri ce pot fi utilizați doar la nivelul acelei metodei. Pentru a putea folosi valorile parametrilor metodei și în afara acelei metode, trebuie ca noi să inițiem noi atribute ale obiectului.

Astfel atributul `self.materie` va lua valoarea `materie`, iar `self.nota` va lua valoarea `nota`. Prin aceasta metoda putem să lucram cu valorile date nu numai la nivelul metodei, ci și după apelarea acesteia.

La rândul 14 regăsim crearea unei noi metode numita `metodaAfisare`. Acesta nu primește decât parametrul standard, adică `self`. La linia 15 se poate observa un comentariu al metodei.

Pe rândul 16 din Fig.18 se poate vedea ca apelam attributele inițializate mai devreme și le afișăm. Aici as dori să adaug ca putem realiza orice operație ne dorim cu aceste attribute.

La linia 18 și 19 cream câte un obiect distinct, numit `obiect1`, și respectiv `obiect2`.

La linia 21 apelam metoda `metodaNume` aplicata la `obiect1`. Aceasta metoda va inițializa atributul `self.nume` cu datele introduse de la tastatura din timpul rulării.

În figura 19 se poate vedea ca s-a testat spre exemplificare "Ion Studentul".

La rândul 22 putem vedea apelarea metodei `metodaNotaPython` ce primește doi parametri în fața de `self`. Așa cum am discutat mai devreme metoda `metodaNotaPython` inițializează două attribute: `obiect1.nota` va avea valoarea "10", iar `obiect1.materie` va avea valoarea "Python".

La linia 23 se apelează metoda `metodaAfisare` ce are ca urmare afișarea celor trei attribute create anterior. Astfel demonstrăm ca attributele create pot exista și în afara unei metode, fiind apelabilă la nivel de clasă. De altfel putem să vizualizăm valoarea unui atribut și în mod direct, nu numai printr-o metoda apelând `obiect.atribut`; cum, de altfel, se poate vedea și la linia 24.

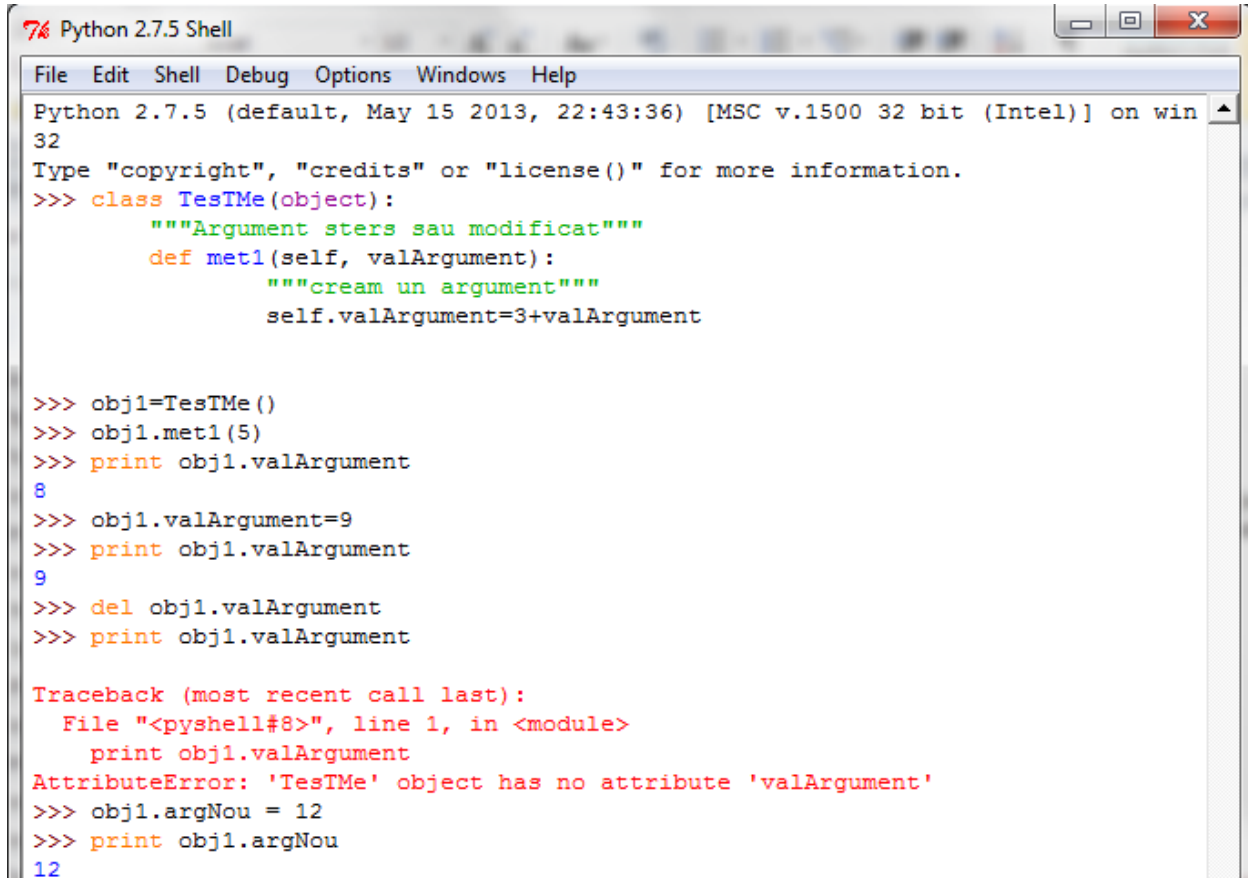
La liniile 26,27,28,29 se repeta aceeași procedura ca și la liniile 21,22,23,24, cu observația ca attributele sunt create pentru obiectul `obiect2`. Valorile parametrilor sunt diferite, creând-se attribute cu valori diferite de `obiect1`. Astfel, `obiect2.nume` va avea valoarea "Maria Studenta" deoarece acest text a fost introdus de la tastatura (Fig.19). Atributul `obiect2.nota` va avea valoarea "10+" în loc de "10" cum era la `obiect1`.

La rândul 31 este un duplicat al rândului 24 pentru a se putea vedea ca valoarea atributului `obiect1.nume` nu s-a schimbat.

Se poate vedea caracteristicile de bază a clasei: încapsularea (fiecare obiect are namespace-ul sau propriu incorporat într-un namespace al clasei), scalabilitate (pot crea oricând o nouă metoda care să deservească noilor scopuri), structurare (e ușor de observat ce atribut al cărui obiect apelăm).

Înainte de a trece la o nouă secțiune trebuie să subliniez că fiecare clasă are propria încapsulare (namespace), fiecare metoda are propriul namespace și fiecare obiect are propriul namespace.

Un atribut poate fi șters sau valoarea lui poate fi modificata. Comportamentul unui atribut este similar unei variabile. Ba mai mult, crearea unui atribut se poate face și din exteriorul clasei, exact ca declararea unei variabile. Acest comportament are rolul de a crește flexibilitatea.



```
Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> class TestMe(object):
    """Argument sters sau modificat"""
    def met1(self, valArgument):
        """cream un argument"""
        self.valArgument=3+valArgument

>>> obj1=TestMe()
>>> obj1.met1(5)
>>> print obj1.valArgument
8
>>> obj1.valArgument=9
>>> print obj1.valArgument
9
>>> del obj1.valArgument
>>> print obj1.valArgument

Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print obj1.valArgument
AttributeError: 'TestMe' object has no attribute 'valArgument'
>>> obj1.argNou = 12
>>> print obj1.argNou
12
```

Fig.32

Se poate vedea în Fig. 32 ca cream o clasa numita TestMe ce are o metoda numita met1. După crearea obiectului, dacă folosim metoda vedem o prelucrare a parametrului dat (se va aduna cu 3 numărul dat ca parametru).

Putem vedea ca prelucrarea atributului se poate face similar ca la o variabila.

Putem face mai mult de atât; putem să ștergem chiar un obiect apelând comanda “del”. Sintaxa utilizată este: del obiect . Aceasta acțiune se poate observa și în Fig.33

```

>>> obj1.argNou = 12
>>> print obj1.argNou
12
>>> del obj1
>>> print obj1.argNou

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <modu
    print obj1.argNou
NameError: name 'obj1' is not defined
>>> |

```

Fig.33

Fiecare obiect are un set de atribute ce pot fi unice de alte obicete. Pot crea atribute si in exteriorul clasei asa cum putem vedea si in Fig.32 si in Fig.33. Acesta creaza posibilitatea ca obiectele sa detina asa cum am afirmat mai sus un set unic fata de alte obiecte, deci oferind scalabilitate.

In următoare secțiune trebuie să vedem cum putem crea o clasa care are parametri de intrare.

```

# Program clasa2
# Demonstreaza utilizarea constructor
# Ion Studentul - 1/26/13

class NumeClasa(object):
    """clasa mea"""
    def __init__(self,nume):
        """nume student"""
        self.nume= nume

    def metodaAfisare(self):
        """afiseaza toate caracteristicile"""
        print "metoda afisare\t", self.nume,"=",self.materie,":", self.nota

obiect1=NumeClasa("Ion Studentul")

obiect1.metodaAfisare()
print "valoarea obiect1.nume este :",obiect1.nume,"\n\n"

raw_input("\n\nApasa <enter> pt a iesi.")

```



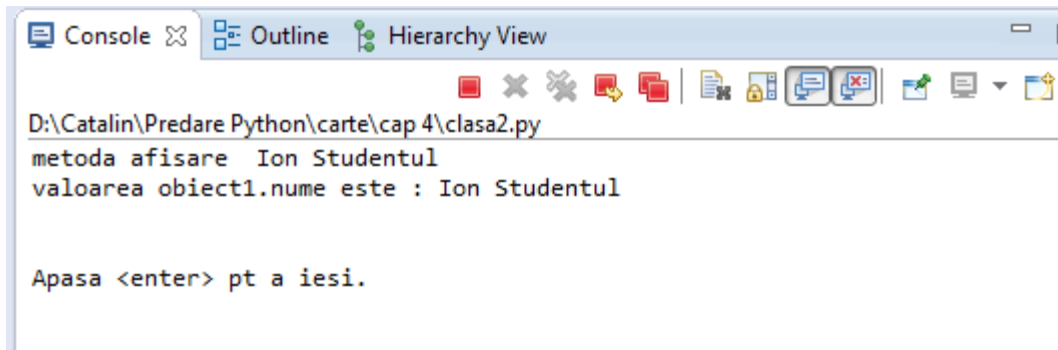


Fig. 34

Se poate observa în Fig.34 ca inițializarea obiectului obiect1 trebuie ca clasa să primească și un atribut. Metoda `__init__` este o metoda speciala care este rulata de fiecare data când un obiect este creat. Prin urmare, dacă aceasta metoda are nevoie de parametri, la initializarea unui obiect clasa ii transmite metodei. În acest caz metoda init inițializează un atribut cu valoarea parametrului. Pentru a ne asigura de funcționarea acestei metode am creat și o doua metoda numita metodaAfisare ce are ca scop afișarea atributului.

O alta metoda speciala este `__str__` prin care putem să printa un obiect.

În mod uzual nu putem să printam un obiect, așa cum indica și fig.34

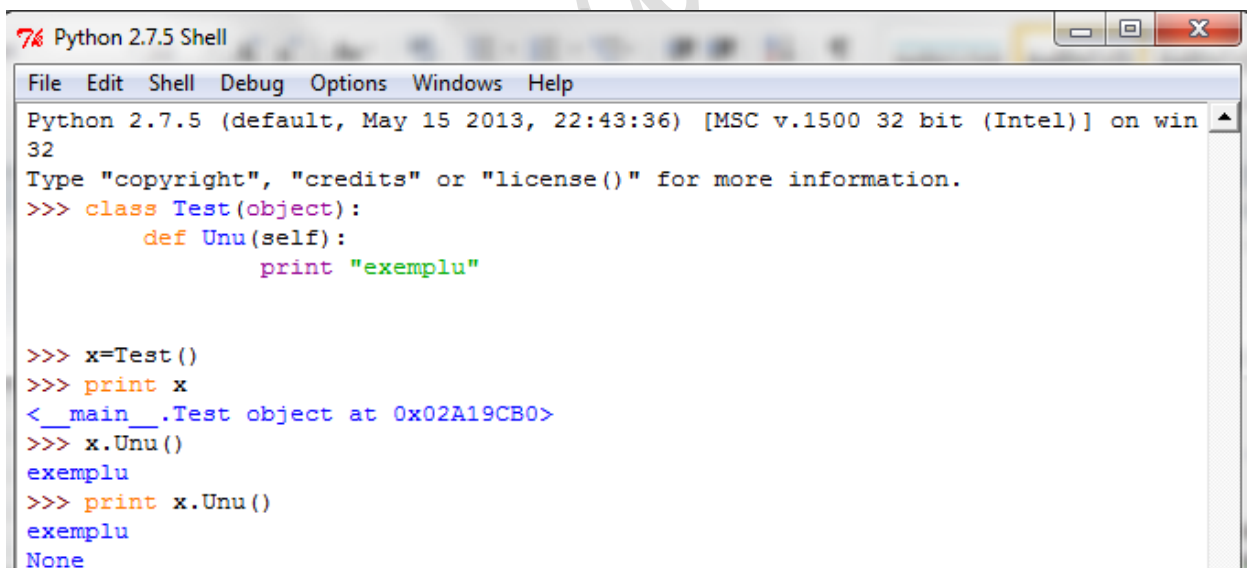


Fig.35

În fig.35 putem observa crearea unei clase numita Test. Aceasta are o singura metoda care afișează "exemplu". Dacă cream un obiect și apoi printam acel obiect vedem un mesaj care indică o adresă de memorie. Dacă afișăm apelarea unei metode ne



returnează None deoarece metoda nu returnează nimic. În programul de mai jos putem vedea cum se poate efectua o metodă care ar permite afișarea unui obiect.

```
# Program clasa4
# Demonstreaza utilizarea metodei speciale __str__
# Ion Studentul - 1/26/13

class NumeClasa(object):
    """clasa mea"""
    def __init__(self, nume):
        """nume student"""
        self.nume= nume

    def __str__(self):
        """afiseaza toate caracteristicile"""
        return "metoda afisare:\t"+str(self.nume)

obiect1=NumeClasa("Ion Studentul")

print obiect1

raw_input("\n\nApasa <enter> pt a iesi.")
```

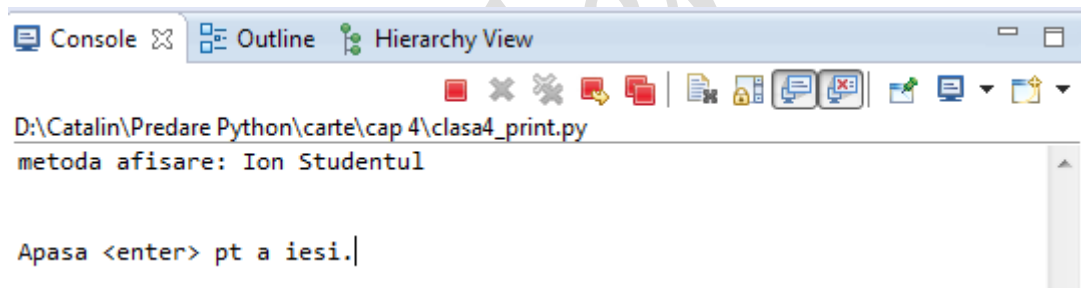


Fig.36

În programul de mai sus folosim metoda `__str__`. Aceasta permite prin return să afișăm obiecte. Se poate vedea că creăm o clasă care are două metode speciale. Una este `__init__` și are rolul de a inițializa un obiect și creează un atribut la acel obiect (`nume`).

Dacă printăm obiectul atunci metoda `__str__` va fi apelată și vom vedea rularea codului din interior. Astfel în fig.36 putem vedea că la printarea obiectului `obiect1` vedem rularea efectivă a metodei `__str__`.