Capitolul 7. OOP – Avansat

7.1 Variabila statica

7.1.1 Variabila statica

Variabila statica (variabila de clasa) este creata in afara metodei si se defineste imediat dupa docstring. Aceasta ne permite sa cream atribute care sa nu fie afectate de metode fiind alocate tuturor instantelor.

```
class Test:
    """Creaza clasa test"""
    culoare = "verde"

obiect = Test()
obiect.culoare                  # accesare variabila
"verde"
Test.culoare                    # accesare variabila
"verde"
```

Variabila “culoare” este variabila de clasa. La creare, instanta “obiect” va primi automat atributul culoare, cu valoarea pe care o are.

Putem accesa atributul dat de variabila de clasa atat prin numele oricarei instante (**NumeInstanta.NumeVariabila)**, cat si prin numele clasei (**NumeClasa. NumeVariabila**). Daca dorim sa utilizam aceasta variabila in interiorul clasei putem utiliza **Self.NumeVariabila** sau **NumeClasa. NumeVariabila**. In exteriorul clasei variabila statica poate fi accesata cu **NumeInstanta.NumeVariabila** sau **NumeClasa. NumeVariabila**.

7.1.2 Atributele clasei, metodei, instantei

In exemplul urmator avem o variabila de clasa “culoare” si o variabila “viteza” in metoda constructor.

```
class Test:
        """Creaza clasa test"""
        culoare = "verde"

        def __init__( self):
                self.viteza = 100

obiect = Test()
obiect.culoare                  # atribut al clasei
"verde"
obiect.culoare = "galben"       # atribut propriu instantei
obiect.culoare
"galben"
obiect.viteza                   # atribut propriu instantei default  la creare
100
obiect.viteza = 150             # atribut propriu instantei – personalizare
Test.culoare

Test.viteza
Traceback (most recent call last):
File "<input>", line 1, in <module>
AttributeError: type object 'Test' has no attribute 'viteza'
```

Culoare este atributul clasei si va fi alocat fiecarei instante, cu valoarea acestuia (verde). Totusi, fiecare instanta poate sa aloce o variabila proprie cu nume identic cu al variabilei de clasa (culoare). In aceasta situatie atributul propriu instantei va avea prioritate si va da starea instantei (galben).

Viteza este atributul preluat la crearea instantei si va prelua valoarea prestabilita. (100). Si pentru acest caz putem atribui pentru obiectul nostru o valoare diferita pentru viteza (**obiect.viteza = 150**).

Dupa cum se observa putem apela atat atributul propriu, cat si atributul clasei (culoare si viteza) prin **NumeInstanta.NumeVariabila**.

Atributul clasei (culoare) poate fi apelat si prin **NumeClasa. NumeVariabila**, ceea ce nu se poate intampla si cu atributul specific instantei (viteza), caz in care va fi returnata o eroare. In toate cazurile, variabila clasei isi va pastra valoarea proprie, chiar daca instantelor le atribuim alte valori (Ex: culoarea verde).

Asa cum ati observat, valorile unor variabile ale instantelor pot fi modificate si din exterior, “ocolind” metodele. De aici rezulta necesitatea unor masuri de validare a datelor, pentru a ne asigura ca nu sunt modificate din exterior ci doar prin intermediul metodelor.

Instructiunile pentru validare vor fi incluse in metodele clasei.

In toate situatiile, valoarea proprie a atributului unei instante precede valoarea data de clase aceluiasi atribut.

7.1.3 Variabila globala

Dictionarul ***globals()*** contine toate variabile disponibile in namespace-ul global.

Putem verifica existenta unei variabile in **globals()** astfel:

```
a = 100                    # Creare VG
b = 'caine'

globals()                  # lista cu VG la un moment dat
```

```
if 'b' in globals():
    print ( 'OK' )              # verifica existenta VG

globals() [ 'b' ] = 'pisica'    # atribuire sau modificare valoare VG
```

Putem defini o variabila globala in interiorul clasei, atat pentru date de intrare cat si pentru date de iesire.

7.2 Recursivitatea intr-o clasa

7.2.1 Functia recursiva

Recursiv este ceva ce se poate repeta la infinit.

Functie recursiva constituie o medoda utilizata in programare si este o functie care se poate apela pe ea insasi de una sau mai multe ori.

In exemplul urmator avem functia factorial, care se apleleaza pe ea insasi. Pentru a observa acest lucru am introdus un print cu ajutorul caruia vizionam fiecare rezultat intermediar. Practic, ca sa calculeze n! are nevoie de (n-1)!, care la randul lui are nevoie de (n-2)! si asa mai departe. Primul rezultat va fi evident pentru 2!, care ne permite sa calculam 3!, pana ajungem la rezultatul final n!. Printarea rezultatelor intermediare se va face in ordinea in care se calculeaza, de la 2! la n!.

```
def factorial(n):
    if n == 1:
        return 1
    else:
        res = n * factorial ( n - 1 )
        print ( "rez intermed pentru ", n, " * factorial(", n - 1, "): ", res )
        return res
```

factorial (5)

rezultat intermediar pentru 2 * factorial(1): 2

rezultat intermediar pentru 3 * factorial(2): 6

rezultat intermediar pentru 4 * factorial(3): 24

rezultat intermediar pentru 5 * factorial(4): 120

120

7.2.2 Clasa recursiva

Clasa recursiva constituie o medoda utilizata in programare, in care o clasa se poate apela pe ea insasi de una sau mai multe ori.

```
from random import randint
# importam functia randint care returneaza un numar intreg aleator dint-un interval

class Banca:
    """Institutie financiara"""
    NrConturi = 0       # numara cate conturi saunt create la un moment dat

# incrementeaza numarul de conturi disponibile la fiecare instantiere

def __init__(self, sold=0):
    self.sold = sold
    Banca.NrConturi += 1

# cream o metoda care poate crea multiple instante apeland clasa

def creez_cont(self, Cate):
for i in range(Cate):
    x = randint(0, 10000000000)                    # numar cont aleator
```

```
globals()['Cont' + '%010d' % x] = Banca()        # creare instanta
print ('Cont' + '%010d' % x)                     # listare instanta
```

7.2.3 Apelarea unei clase dintr-o alta clasa

In urmatorul exemplu, o instanta a clasei B aplica metoda inmulteste, proprie acestei clase cu un parametru obtinut prin apelarea de catre o instanta a clasei A a metodei aduna, proprie acestei a doua clase.

```
class A:            # clasa A cu metoda aduna

    def aduna(self, a, b):
        return a + b

class B:            # clasa B cu metoda inmulteste

    def inmulteste(self, x, y):
        return x * y
bb = B()                        # instanta in clasa B

aa = A()                        # instanta in clasa A

print(bb.inmulteste(aa.aduna(3, 7), 20))
200
```

7.3 Incapsularea

7.3.1 Incapsularea

Incapsularea reprezinta unul din principiile de baza ale OOP.

Cu ajutorul incapsularii asiguram introducerea corecta a datelor (atribute) specifice fiecarei instante. Datele trebuie accesate doar prin intermediul metodelor. Reprezentarea interna a obiectelor nu poate si nu trebuie sa fie vazuta din exterior.

In Python, in mod normal atributele unei instante isi pot modifica valoarea oriunde si oricate masuri de precautie ar fi luate pot fi ocolite. Asta pentru ca in Python, spre deosebire de alte limbaje de programare, nu exista informatii ascunse, toate fiind publice. Exista totusi instrumente cu ajutorul carora putem regleementa acest lucru si pastra integritaatea obiectelor, prin intermediul incapsularii.

7.3.2 Metoda speciala

Metodele speciale sunt de forma **__metoda__** si se apeleaza prin modalitati diferite. Ati avut cateva exemple in acest sens:

- metoda **__init__** se apeleaza automat la crearea unei instante;
- metoda **__str__** se apeleaza prin **print(NumeInstanta)**;
- metoda **__del__** se apeleaza prin **del NumeInstanta**;
- metoda **__doc__** se apeleaza prin **NumeClasa. __doc__**;
- unele metode se pot apela ca si functiile primind ca argument numele instantei;
- etc.

Toate aceste metode au in comun incapsularea. Practic utilizatorul nu trebuie sa stim cum face, trebuie sa stim doar ce face respectiva metoda.

7.3.3 Metoda privata

Metodele private sunt de forma **__MetodaPrivata** (numele metodei este precedat de dublu underscore). O metoda privata nu poate fi accesata de catre o instanta in mod direct (obiect.__MetodaPrivata sau obiect.MetodaPrivata). Ca s-o accesa utilizam sintaxa: **obiect._Clasa__MetodaPrivata** (obiect, punct, numele clasei precedat de underscore si numele metodei private).

7.3.4 Variabila privata

Variabila privata (atribut privat) este reprezentat sub forma **__VariabilaPrivata**.

Aceasta nu poate fi apelata si modificata direct de catre instanta. Apelarea se poate face prin intermediul unei metode sau prin sintaxa: **obiect._Clasa__VariabilaPrivata**.

Exercitiul 612 ilustreaza cum functioneaza metoda privata si atributul privat.

7.4 Mostenirea

7.4.1 Clasa parinte/clasa copil. Superclasa/subclasa

Mostenirea este o alta proprietate importanta a claselor in OOP. Atributele si metodele unei clase (***parent***), numita si ***superclasa*** sunt mostenite de o clasa (***child)***, numita si ***subclasa***. Consecinta este ca metodele superclasei sunt accesibile si subclasei fara nicio restrictie. Obiectele (instantele) superclasei nu se mostenesc. Metodele subclasei nu sunt accesibile superclasei.

Pot exista mai multe niveluri de mostenire.

O preocupare importanta este aceea de a evita dublarea unor portiuni de cod, care sa faca acelasi lucru atat in superclase cat si in subclase.

O clasa child, daca nu are o metoda constructor, va cauta ierarhic la clasa parent urmatoare pana va gasi una, daca aceasta exista. Putem initializa atribute atat in clasa parent cat si in clasa proprie.

Sintaxa este urmatoarea:

class ClasaNoua(ClasaMostenita):

Bloc de instructiuni

7.4.1 Ierarhia atributelor in clasele mostenite

Exista o ierarhie de cautare a atributelor (obiect.atribut) in cazul claselor mostenite, astfel:

- in propria instanta (atribute proprii instantei);
- in propria clasa (metodele proprii);
- in clasele mostenite de la cea mai apropiata pana la cea mai indepartata (metode mostenite).

7.4.1 Mostenirea multipla

In cazul mostenirii multiple, daca exista doua atribute cu acelasi nume va cauta mai intai in adancime, in ordinea claselor mostenite. Daca o clasa mosteneste doua clase, prima in ordinea normala va fi eliminata (va fi mostenit atributul celei de-a doua).

Sintaxa este urmatoarea:

class ClasaNoua(ClasaMostenita1, ClasaMostenita2[, ...]):

Bloc de instructiuni

Exercitiile 615-616 ilustreaza cum functioneaza mostenirea.

7.5 Polimorfism

7.5.1 Polimorfism

Polimorfismul este o alta proprietate a claselor in OOP si functiilor. Polimmorfism inseamna o metoda sau o functie cu denumire identica, dar cu functionalitati diferite.

Cel mai simplu model este al functiei len(), pe care o putem aplica la obiecte diferite in mod diferit (la siruri de caractere numara caracterele in timp ce la liste numara elementele listei).

La o clasa polimorfica fiecare metoda este diferita, dar similara din punct de vedere conceptual.

7.6 Recapitulare

7.6.1 Recapitulare