Capitolul 6. OOP – Notiuni introductive

6.1 Introducere in OOP

6.1.1 Introducere in OOP

Pana acum am discutat despre programarea procedurala. Am putut utiliza functii, variabile, diferite tiputi de date predefinite (siruri de caractere, liste, dictionare, tuplu, etc). Cu ajutorul acestora putem efectua programe simple. Daca dorim sa abordam programe mai complexe, cum ar fi un program care contine baze de date, am putea apela la intrumentele invatate: liste, dictionare sau combinatii ale acestora. Aceasta solutie are unele minusuri cum ar fi:

- Trebuie sa respectam o structura impusa de instrumentele folosite (liste sau dictionare), nu avem libertatea de a trata diferit anumite aspecte (Ex: daca avem o categorie vehicule, acestea pot fi cu motor sau nu, pot avea roti sau nu, pot avea caracteristici diferite, etc). Toate acestea ar putea face codul foarte stufos, ineficient, greu de administrat si de intretinut;
- Programul ar avea un numar foarte mare de intrari, ceea ce duce la o rulare mai lenta;
- Validarea datelor poate fi imposibila in anumite situatii;

Toate aceste neajunsuri pot fi rezolvate utilizand programarea orientata pe obiecte (OOP), care introduce un nou tip de date, tip care imbina in codul python date, functionalitati si atribute.

Principalele ***caracteristici ale OOP*** sunt urmatoarele:

Structurare:

- usor de utilizat;
- usor de inteles;
- usor de intretinut;
- usor de extins;

Scalabilitate:

- crearea de noi tipuri de date, inexistente, ceea ce rezolva problema atributelor diferite (in exemplul nostru pentru vehicule diferite);

Refolosirea codului:

- un nou tip de date creat, testat si functional poate fi refolosit fara probleme in alte programe (modularizare);

Incapsulare:

- informatia este ascunsa pentru a rula mai eficient. Clientul are acces la functionalitati, nu-l intereseaza cum au fost rezolvate tehnic. Aceasta face ca intrarile, la un moment dat sa fie mult mai putine decat in cazul programarii procedurale si astfel sa creasca viteza de procesare;

Programarea orientata pe obiecte are patru principii fundamentale, pe care le vom aborda in capitolul urmator. Acestea sunt: ***abstractizarea, incapsularea, mostenirea si polimorfismul***.

OOP – principala modalitate pentru design de software. Datele sunt organizate in ***obiecte***, functionalitatile sunt incluse in ***metode*** aplicabile obiectelor. Obiectele si metodele impreuna formeaza ***clasa***, "containerul" – noul tip de date. In exemplul nostru, clasa ar fi vehiculul. Obiectele ar putea fi autoturism, camion, bicicleta, etc. Metodele sunt cele care definesc anumite caracteristici comune sau diferite.

Clasa ajuta la abstractizarea caracteristicilor unui lucru din viata reala. In viata reala un ***obiect*** este ceva tangibil. In software obiectul este destinat sa faca ceva. Este o colectie de date si functionalitati. Un obiect definit intr-o clasa poarta numele de ***instanta***. O valoare atribuita obiectului poarta denumirea de ***atribut***. Fiecare instanta poate avea: atribute definite de clasa (de regula general valabile pentru toate instantele clasei) si atribute proprii (de regula specifice unei anumite instante). ***Metoda*** este o functie care poate fi aplicata obiectului.

Conceptul de OOA (analiza orientata pe obiecte) reprezinta procesul de identificare a obiectelor si a relatiilor dintre acestea. Determina ce trebuie sa faca nu cum sa faca, cerintele programului.

Conceptul de OOD (design orientat pe obiecte) reprezinta etapa in care transformam necesitatile in specificatii. Cum se face. Se definesc obiectele, functionalitatile, relatiile dintre ele.

Conceptul de OOP (programarea orientata pe obiecte) reprezinta etapa in care se implementeaza OOD. Practic se realizeaza programul.

In practica procesul este iterativ, aplicandu-se la fiecare modul in parte.

6.1.2 Clase

Definitie: Clasa reprezinta un nou tip de date, compus, definit de programator, care are unul sau mai multe atribute si caruia i se pot aplica diferite metode.

Spre deosebire de alte limbaje de programare in care avem primitive (ex: string, intreg, lista, etc), in Python toate acestea sunt obiecte, practic apartin unor clase. Cu type() determinam ce fel de obiect este, carei clase ii apartine, cu dir() ce metode i se pot aplica;

Sintaxa pentru crearea unei clase:

```
class NumeClasa:
    """doc_string"""
    bloc de instructiuni
```

Numele claselor respecta aceleasi reguli de denumire ca si variabilele. Este recomandat sa inceapa cu o litera mare. Nu putem folosi cuvinte cheie sau rezervate.

Docstringul va cuprinde detalii despre clasa respectiva si este similar cu docstringul functiilor. Se poate apela cu instructiunea:

NumeClasa.__doc__

Blocul de instructiuni poate contine una sau mai multe instructiuni (variabile ale clasei, metode, metode speciale) despre care vom discuta in continuare.

Exemplu:

```
class Biciclete:                          # Numele clasei
    """Creaza clasa biciclete"""          # Docstring
    pass                                  # Instructiuni (pass daca lipsesc)
```

Biciclete.__doc__

"Creaza clasa biciclete"

6.1.3 Obiecte

Definitie: Obiectul este o instanta a unei clase (instantiere – crearea de obiecte). Putem crea cate obiecte dorim. Fiecare obiect are atat atribute generale, specifice clasei din care face parte cat si atribute specifice obtinute prin aplicarea metodelor acceptate.

Sintaxa pentru crearea unui obiect:

```
NumeObiect = NumeClasa([parametrii])          # mai intai cream ob.
```

Putem sterge un obiect.

Sintaxa pentru stergerea unui obiect:

```
del NumeObiect
```

Exemplu:

```
class Biciclete:                          # Numele clasei
    """Creaza clasa biciclete"""          # Docstring
    pass                                  # Instructiuni (pass daca lipsesc)
```

```
Pegas = Biciclete()                    # cream o instanta, un obiect
Cross = Biciclete()                    # cream o instanta, un obiect

del Pegas
del Cross
```

6.1.4 Metode

Definitie: Metoda este o functie definita in interiorul unei clase, cu ajutorul careia putem da atribute (caracteristici specifice) obiectelor (instantelor).

Daca o clasa este "containerul" metoda constituie interfata, butonul cu ajutorul caruia executam diferite functii. Codul acesteia este ascuns utilizatorului. Aplicam metoda prin numele ei si eventualii parametrii.

Functionalitatea instantelor este data de aplicarea metodei. Toate instantele unei clase au acces la metodele acesteia, dar nu sunt obligate sa le aplice pe toate, ceea ce face ca obiecte diferite sa aiba fuctionalitati diferite.

O metoda va avea intotdeauna cel putin un atribut, denumit generic ***self***. Acest parametru nu trebuie specificat si nu are o valoare. Practic prin self se face legatura dintre instanta si atributele specifice metodei. Dupa self putem adauga unul sau mai multi parametri, in functie de necesitatile metodei. La aplicarea metodei vom adauga valori doar pentru ceilalti parametri, nu si pentru self.

Utilizarea constructiei **self.parametru = parametru** in cadrul unei metode asigura posibilitatea utilizarii variabilei **self.parametru** oriunde avem nevoie de ea in interiorul clasei. Putem s-o asemanam cu **global** in interiorul clasei in cazul functiilor (o metoda de output in afara metodei, in interiorul clasei). Practic, daca am folosi **global** vizibilitatea variabilei s-ar extinde si in afara clasei, caz in care n-ar mai fi posibila incapsularea.

Metoda poate avea propriul docstring, care va da detalii despre ce face aceasta. Vizualizarea acestuia se face cu instructiunea: **NumeClasa.NumeMetoda.__doc__**

Sintaxa pentru apelarea unei metode:

```
NumeObiect.Metoda([parametrii])        # aplicam metoda
```

Exemplu:

```
class Biciclete:                          # Numele clasei
    """Creaza clasa biciclete"""          # Docstring

    def nr_viteze(self, viteze):          # Metoda de adaugare atribut
        """Seteaza numar viteze"""        # Docstring metoda
        self.viteze = viteze


Pegas = Biciclete()                       # cream o instanta, un obiect

Cross = Biciclete()                       # cream o instanta, un obiect

Pegas. nr_viteze(3)                       # aplicam metoda oricarei instante

Cross. nr_viteze(27)                      # aplicam metoda oricarei instante


Biciclete.__doc__

"Creaza clasa biciclete"

Biciclete.nr_viteze.__doc__

"Seteaza numar viteze"
```

Metoda nr_viteze() poate fi aplicata tuturor instantelor, cu valori specifice fiecarei instante.

Se observa si rezultatul apelarii docstringului pentru clasa si pentru metoda.

Scalabilitatea este data de posibilitatea de a crea noi metode, care sa poata crea noi atribute.

6.1.5 Atribute

Definitie: Atributul reprezinta o valoare specifica unei functionalitati aplicata pentru o instanta data.

O instanta poate accesa atat variabilele definite in clasa cat si pe cele definite prin aplicarea propriilor metode.

Totalitatea atributelor pe care le are o instanta, la un moment dat, reprezinta ***starea*** instantei. Practic, instanta preia valorile atributelor primite. Obiectul este definit de starea si comportamentul acestuia.

O instanta va prelua cu prioritate atributul propriu fata de al clasei. De exemplu, daca stabilim ca atribut de clasa pentru biciclete numarul de roti egal cu 2 (putem sa-i spunem atribut default), putem avea biciclete care sa aiba un atribut specific cu un numar diferit de roti (ex: 4, doua roti principale si doua ajutatoare).

Daca in instanta avem o variabila cu un nume similar cu al unei variabile a clasei, variabila clasei va fi ascunsa temporar, cat timp exista cea a instantei. Aceste aspecte tin de namespace si scope si vor fi tratate in subcapitolul urmator.

Exemplu:

```
class Biciclete:                          # Numele clasei
    """Creaza clasa biciclete"""          # Docstring clasa

    def nr_viteze(self, viteze = 1):      # Metoda de adaugare atribut
        """Seteaza numar viteze"""        # Docstring metoda
        self.viteze = viteze

    def provenienta(self, p = 'Romania' ):
        """Seteaza tara provenienta"""
        self.p = p

Pegas = Biciclete()                       # cream o instanta, un obiect

Cross = Biciclete()                       # cream o instanta, un obiect
```

```
Pegas. nr_viteze(3)                # aplicam metoda oricarei instante
Cross. nr_viteze(27)               # aplicam metoda oricarei instante

Pegas.viteze                       # vizualizam atribut specific Pegas
Cross.viteze                       # vizualizam atribut specific Cross

Carpati = Biciclete()              # cream o instanta noua

Carpati.nr_viteze()                # aplicam metode noii instante
Carpati.provenienta()

Carpati.p                          # vizualizam atributele specifice instantei
'Romania'                          # se observa preluarea parametrilor default
Carpati.viteze
1

Cross.provenienta('Germania')      # aplicam metoda instantei vechi

Cross.p

'Germania'
```

Atributele au rolul de a ***structura*** codul. Fiecare obiect are propria stare, propriile atribute si propriile valori pentru acestea.

Daca facem un rezumat, putem spune ca un program reprezinta o multime de obiecte care utilizeaza impreuna resursele de care dispun. Obiectul reprezinta o mica parte din program, care contine cod si date. Putem astfel sa impartim problema de rezolvat in parti mai mici, care ne ajuta sa ignoram detaliile din restul programului. Rezolvand partile mici facem sa functioneze intregul.

6.2 Namespace si scope in OOP

6.2.1 Namespace

Definitie: Namespace reprezinta locul unde sunt vizibile functiile, clasele, metodele si obiectele. Pratic este o mapare nume – obiect.

Exista mai multe niveluri de namespace:

- ***Al functiilor predefinite***, creat la start si nu va fi oprit niciodata. Practic, functiile predefinite pot fi apelate de oriunde din program, oricand;
- ***Al modulelor***, creat la pornirea acestora. Practic, toate functiile si clasele definite intr-un modul pot fi apelate oriunde in program incepand cu momentul apelarii modulului (**import NumeModul** – detalii la capitolul dedicat modulelor);
- ***Local***, al functiilor, disponibil in timpul apelarii acestora. Practic, variabilele definite in interiorul functiilor sunt accesibile doar la momentul apelarii. Daca dorim sa avem date de iesire nu o putem face prin intermediul variabilelor locale ci prin intermediul return;
- ***Al clasei, obiectului sau metodei***. Fiecare dintre acestea au namespace proprii.

O functie care apartine de global namespace, este vizibila oriunde avem nevoie de ea. O metoda este vizibila doar in namespace-ul clasei in care a fost creata. Diferenta este facuta prin modul de accesare.

Prin delimitarea clara a namespace-ului se asigura practic ***incapsularea***.

6.2.2 Scope

Clasa este un obiect ca oricare altul si se comporta in acelasi mod ca orice variabila (string, integer, etc). Timpul de viata al acesteia (domeniul de vizibilitate) se numeste ***scope***. Timpul de viata al functionalitatilor clasei (metodei) este asemenator cu al variabilelor si este strans legat de namespace.

Scope poate fi:

- Domeniul cel mai apropiat, accesat primul (exemplu instanta unei clase);
- Domeniul functiilor incorporate (exemplu metoda, clasa);
- Domeniul functiilor globale de la nivelul modulelor;
- Domeniul cel mai indepartat, accesat ultimul (functii predefinite).

De aici avem urmatoarele implicatii pentru OOP:

- O instanta va accesa cu prioritate o variabila proprie (atribut propriu) si doar daca nu are o valoare proprie va accesa valoarea data acestui atribut de metoda, iar daca nici metoda nu are o valoare proprie va accesa valoarea data de clasa;
- O metoda va putea functiona doar in clasa in care a fost creata;
- Variabilele definite in interiorul metodei vor functiona doar la apelarea metodei (exceptie cele definite cu self.NumeVariabila, care vor fi vizibile in intreaga clasa);

6.3 Metode speciale

6.3.1 Metoda speciala __init__

Metoda speciala **__init__** este numita si ***metoda constructor***. Python nu are explicit o metoda constructor. Metoda **__init__** poate fi asimilata cu o metoda

constructor. Totusi, spre deosebire de alte limbaje de programare, instanta este creata deja cand se apeleaza (automat) aceasta metoda.

Metoda **__init__** este o metoda speciala, privata, care este rulata de fiecare data când un obiect este creat, initializand diferite atribute (variabile). Metoda poate avea parametri pozitionali sau parametri default in functie de situatie.

Metoda speciala **__init__** nu poate folosi **return**.

```
def __init__(self, nume, culoare):
                                    # metoda construtor, metoda privata
    self.nume = nume                # apelata automat la crearea unei instante
    self.culoare = culoare          #  initializeaza atribute la crearea instantei
```

In acest exemplu, la crearea oricarei instante va trebui sa specificam cei doi parametri, nume si culoare. Exista si posibilitatea definirii de parametri cu valori default. Modul de lucru este identic cu parametrii functiilor.

6.3.2 Metoda speciala __str__

Metoda **__str__** este o metoda speciala care este rulata de fiecare data printam un obiect. Nu se apeleaza direct. Pur si simplu cu print;

Metoda speciala **__str__** are obligatoriu un return.

```
def __str__(self):              # metoda speciala pentru printarea starii obiectului
    """Afiseaza starea obiectului"""
    return "Bicicleta: ({0}, culoare: {1}, viteza: {2}, prov: {3})".format
    (self.nume, self.culoare, self.viteza, self.p)
```

Prin aceasta metoda vom putea ca, prin instructiunea **print (NumeInstanta),** sa vizualizam starea obiectului.

6.3.3 Metoda speciala __del__

Metoda **__del__** este o metoda speciala pentru stergerea unui obiect;

```
del obiect          - apelare
```

```
def __del__(self):                    # metoda de stergere a instantei
    print('Stergerea a fost efectuata.')
```

6.3.4 Clase cu parametri default

Putem crea clase cu parametri default. Valorile parametrilor default vor fi setate in metoda **__init__**;

La crearea unei instante valorile parametrilor default vor fi preluate automat sau vor fi specificate valori proprii.

Exemplu:

```
class Biciclete:                              # Numele clasei
    """Creaza clasa biciclete"""              # Docstring clasa

    def __init__(self, nume, culoare ='neagra' ):
        """Metoda constructor"""
        self.nume = nume                      # culoare are valoare default
        self.culoare = culoare

    def __str__(self):
        """ Afiseaza starea obiectului """
        return "Bicicleta: ({0}, culoare: {1}, viteza: {2}, prov: {3})"\
        .format(self.nume, self.culoare,  self.viteza, self.p)
```

```
    def __del__(self):
        """Sterge obiecte"""
        print('Stergerea a fost efectuata.')
    def nr_viteze(self, viteze = 1):        # Metoda de adaugare atribut
        """Seteaza numar viteze"""          # Docstring metoda
        self.viteze = viteze

    def provenienta(self, p = 'Romania' ):
        """Seteaza tara provenienta"""
        self.p = p

Pegas = Biciclete('Pegas','Alb')        # cream o instanta, un obiect
Pegas. nr_viteze(3)                     # aplicam metoda oricarei instante
Pegas.provenienta()

Cross = Biciclete('Cross')              # preia culoarea default, neagra
Cross. nr_viteze(27)                    # aplicam metoda oricarei instante
Cross.provenienta('Germania')

Carpati = Biciclete()                   # cream o instanta noua
Carpati.nr_viteze()                     # aplicam metode noii instante
Carpati.provenienta()

print( Pegas )

del Pegas
```

6.4 Recapitulare

6.4.1 Recapitulare