1.2 Recapitulare OOP

În cadrul programării orientate pe obiect ne vom concentra pe definirea obiectelor necesare cu ajutorul claselor, pe care mai departe le vom crea (sau instanția) - instanțele create în cadrul unui program vor interacționa în vederea prelucrării de date. Această abordare diferă semnificativ de programarea procedurală, unde aveam un set de date de intrare ce erau trecute printr-o secvență de instrucțiuni în vederea obținerii unor date la ieșire.

În abordarea obiect orientaă, **clasa** are rolul cel mai important, întrucât reprezintă elementul de limbaj cu ajutorul căruia definim o schiță a obiectelor de care avem nevoie să fie prezente în cadrul aplicației noastr. În cadrul acesteia vom încadra atribute și comportamente care vor descrie modul în care ”arată” instanțele clasei noastre. Cu ajutorul **atributelor** vom reprezenta informațiile cunocute despre obiect, în timp ce **comportamentele**  vor reprezenta ce știe obiectul să facă.

De exemplu, dacă ne raportăm la o clasă Pisica: atributele pot fi culoare, nume, greutate – în timp ce comportamentele **spune\_miau(), bea\_lapte(), doarme().**

Atributele vor fi reprezentate de variabilele definite în interiorul clasei în timp ce comportamentele vor fi definte cu ajutorul metodelor. În momentul stabilirii acestora, este important de reținut că fiecare instanță în sine trebuie să fie cât mai puternic individualizată pe baza valorii atributelor și modului de execuție a metodelor.

Concret, dacă am avea toate instanțele clasei Pisica cu aceeași valoare pentru **culoare**, atunci aceasta nu ar mai reprezenta un atribut, ci mai degraba o zonă de memorie de clasă. În mod asemănător, la execuția comportamentelor menționate mai sus, este indicat ca fiecare instanță să se comporte diferit.

Cuvântul cheie folosit pentru definirea claselor este **class,**iar cel mai simplu mod pentru definirea unei clase este:

class Pisica:

    pass

Ulterior definirii clasei se poate merge mai departe cu instanțierea acesteia:

leia = Pisica()

sheba = Pisica()

Mai departe putem merge să adăugăm atribute acestei clase, împreună cu comportamente

class Pisica:

    def \_\_init\_\_(self, nume, varsta, greutate):

        self.nume = nume

        self.varsta = varsta

        self.greutate = greutate

    def spune\_miau(self):

        print("Meow, numele meu este {}".format(self.nume))

Dupa adăugarea acestor atribute, modul de instanțiere prezentat mai sus nu mai este valabil, întrucât în cadrul metodei *\_\_init\_\_* am specificat trei parametrii ce trebuie dați pentru un apel corect. Așadar, instanțierile vor fi corectate pentru a se ajunge la următorul format:

leia = Pisica("Leia", 4, 5.6)

sheba = Pisica("Sheba", 3, 4.7)

Ulterior, aceste instanțe vor putea apela comportamentul definit anterior.

leia.spune\_miau()

sheba.spune\_miau()

Rezultatul operațiilor de mai sus trebuie sa fie:

Meow, numele meu este Leia

Meow, numele meu este Sheba

1.3 Moștenire și suprascriere

Moștenirea este un procedeu specific programării orientate pe obiect. O clasă ce moștenește o altă clasă preia atributele și comportamentele de la aceasta. Scopul este de a avea cât mai puțin cod duplicat între clase ce au elemente comune. Principiul de bază este acela de particulariza un prototip deja definit de către o clasă anume.

Prin particularizare înțelegm că vom crea ceva nou, pornind de la ceva existent. Acest concept este mai ușor de înțeles dacă ne raportăm la un caz concret - dacă avem deja schița/desenul unei biciclete și ne dorim să schițăm o tricicletă - nu este necesar să plecăm de la 0. Ne putem raporta la ceea ce avem deja definit, urmând să adăugăm atribute specifice noului obiect.

În cadrul Python există trei modalități de particularizare a unui prototip existent:

1. Adăugarea de atribute noi
2. Adăugarea de comportamente noi
3. Modificarea comportamentelor existente/moștenite) - suprascrierea

Plecând de la exemplul prezentat în cadrul sub-capitolului 1.2, cu clasa **Pisica**, putem să ne gândim și la posibilitatea adăugării de clase pentru alte animale - de exemplu **Catel**.

Nu putem realiza o relație de moștenire între clasele **Pisica** și **Catel** - dar putem identifica atributele/comportamentele comune celor două clase și să le grupăm în cadrul unei clase separate - **Animal**. Mai departe, cele doua clase Pisica și Catel, vor moșteni clasa **Animal** și vor adăuga particularități.

Sintaxa specifică de moștenire este: **ClasaCareMosteneste(ClasaDinCareMosteneste).**

În cazul în care o clasă este definită și nu are paranteze puse dupa numele său sau are parantezele goale înseamnă că moștenește doar pe **object** - aceasta reprezintă rădăcina tuturor claselor în limbajul Python. Orice clasă moștenește direct sau indirect pe **object**.

class Animal:

    def \_\_init\_\_(self):

        pass

class Catel(Animal):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

class Pisica(Animal):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

Dacă adăguăm atributele și comportamentlee comune în cadrul clasei Animal, particularizând mai departe cele două sub-clase vom obține:

class Animal:

    def \_\_init\_\_(self, nume):

        self.nume = nume

    def doarme(self):

        print("Animalul {} doarme".format(self.nume))

class Catel(Animal):

    def \_\_init\_\_(self, nume, dresat):

        super().\_\_init\_\_(nume)

        self.dresat = dresat

    def latra(self):

        print("Catelul {} latra.".format(self.nume))

class Pisica(Animal):

    def \_\_init\_\_(self, nume):

        super().\_\_init\_\_(nume)

    def spune\_miau(self):

        print("Pisica {} miauna".format(self.nume))

    def toarce(self):

        print("Pisica {} toarce".format(self.nume))

Pe baza implementării curente, putem spune că toate animalele au comportamentul **doarme() și atributul self.nume.**Pe de altă parte, doar instanțele clasei Catel au atributul **self.dresat**și comportamentul **latra()** - în același timp, doar instanțele clasei Pisica au comportamentele de **spune\_miau()** și **toarce()**.

Suprascrierea reprezintă procesul de redefinire a comportamentului unei metode moștenite, păstrând însă identificatorul. Folosim acest lucru în momentul în care moștenim o metodă dintr-o clasă dar ne dorim să îi particularizăm comportamentul pentru clasa ce o moștenește.

Vechiul comportament poate să fie în continuare apelat cu ajutorul lui **super()**, dar acest lucru se face doar în cadrul clasei în care are loc suprascrierea. Dacă nu se face acest lucru, instanțele clasei care a realizat suprascrierea nu mai pot apela comportamentul original, așa cum a fost el inițial moștenit.

class Animal:

    def scoate\_sunet(self):

        print("Animalul scoate sunete generale")

class Pisica(Animal):

    def scoate\_sunet(self):

        print("Pisica scoate sunete specifice")

class Catel(Animal):

    def scoate\_sunet(self):

        super().scoate\_sunet()

        print("Pe langa sunetele generale, catelul scoate si sunete specifice")

În acest exemplu, în cadrul clasei Pisica suprascriem comportamentul de **scoate\_sunet(),**pierzând cu totul comportamentul original moștenit din cadrul clasei Animal. Pe de altă parte, în clasa Catel, înainte de a adăuga comportament specific clasei curente, apelăm comportamentul original, din clasa Animal.