Curs 5: Tipuri definite de utilizator

- Programare orientată obiect
- Principii de definire a tipurilor utilizator
- TAD - Tip abstract de date

Curs 4.

- Organizarea aplicației pe funcții, module și pachete
- Arhitectura stratificată
- Excepții

Review calculator modular

Câteva probleme:

- Starea calculatorului:
  - varianta cu variabilă globală:
    - avem mai multe variabile globale care pot fi cu ușurință accesate din exterior (posibil stricând starea calculatorului)
    - variabila globală face testarea mai dificilă
    - nu este o legătură clară între aceste variabile (starea calculatorului este împrăștiat în cod)
  - varianta fără variabile globale:
    - starea calculatorului este expus (nu există garanții ca metodele se apelează cu un obiect care reprezintă calculatorul)
    - trebuie sa transmitem starea, ca parametru pentru fiecare funcție legată de calculator
- Numere raționale
  - reprezentarea numerelor este expusa: ex: `rez= suma(total[0],total[1],a,b) ,` putem cu ușurința altera numărul rațional (ex. Facem `total[0] = 8` care posibil duce la încălcarea reguli cmmdc(a,b) ==1 pentru orice numărul rațional a/b
  - codul pentru adunare, înmulțire, etc de numere raționale este diferit de modul in care facem operații cu numere întregi. Ar fi de preferat sa putem scrie r = r1+r2 unde r,r1,r2 sunt numere raționale

Programare orientată obiect

Este metodă de proiectare și dezvoltare a programelor:

- Oferă o abstractizare puternică și flexibilă
- Programatorul poate exprima soluția în mod mai natural (se concentrează pe structura soluției nu pe structura calculatorului)
- Descompune programul într-un set de obiecte, obiectele sunt elementele de bază
- Obiectele interacționează pentru a rezolva problema, există relații între clase
- Clasele introduc tipuri noi de date, modelează elemente din spațiul problemei, fiecare obiect este o instanța a unui tip de data (clasă)

Clasă

Definește in mod abstract caracteristicile unui lucru.
Descrie două tipuri de atribute:

- câmpuri (proprietăți) – descriu caracteristicile
- metode (operații) – descriu comportamentul

Clasele se folosesc pentru crearea de noi tipuri de date (tipuri de date definite de utilizator)
Tip de date:

- domeniu
- operații

Clasele sunt folosite ca un șablon pentru crearea de obiecte (instanțe), clasa definește elementele ce definesc starea si comportamentul obiectelor.

Definiție de clasă în python

```
class MyClass:
    <statement 1>
    ….
    <statement n>
```

Este o instrucțiune executabilă, introduce un nou tip de date cu numele specificat.

Instrucțiunile din interiorul clasei sunt în general definiții de funcții, dar și alte instrucțiuni sunt permise

Clasa are un spațiu de nume propriu, definițiile de funcții din interiorul clasei (metode) introduc numele funcțiilor în acest spațiu de nume nou creat. Similar și pentru variabile

Obiect

Obiect (instanță) este o colecție de date și funcții care operează cu aceste date

Fiecare obiect are un tip, este de tipul clasei asociate: este instanța unei clase

Obiectul:

- înglobează o stare: valorile câmpurilor
- folosind metodele:
  - putem modifica starea
  - putem opera cu valorile ce descriu starea obiectelor

Fiecare obiect are propriul spațiu de nume care conține câmpurile și metodele.

Creare de obiecte. Creare de instanțe a unei clase (__init__)

Instanțierea unei clase rezulta in obiecte noi (instanțe). Pentru crearea de obiecte se folosește notație similară ca și la funcții.

```
x = MyClass()
```

Operația de instanțiere (“apelul” unei clase) creează un obiect nou, obiectul are tipul MyClass

O clasă poate defini metoda specială __init__ care este apelată în momentul instanțierii

```
class MyClass:
    def __init__(self):
        self.someData = []
```

__init__ :

- creează o instanță
- folosește “self” pentru a referi instanța (obiectul) curent (similar cu “this” din alte limbaje orientate obiect)

Putem avea metoda __init__ care are și alți parametrii în afară de self

Câmpuri

```
x = RationalNumber(1,3)
y = RationalNumber(2,3)
x.m = 7
x.n = 8
y.m = 44
y.n = 21
```

```
class RationalNumber:
    """
       Abstract data type for rational numbers
       Domain: {a/b where a and b are integer numbers b!=0}
    """

    def __init__(self, a, b):
        """
          Creates a new instance of RationalNumber
        """
        #create a field in the rational number
        #every instance (self) will have this field
        self.n = a
        self.m = b
```

self.n = a vs n=a

1 Creează un atribut pentru instanța curentă
2 Creează o variabilă locală funcției

Metode

Metodele sunt funcții definite in interiorul clasei care au acces la valorile câmpurilor unei instanțe.

În Python metodele au un prim argument: instanța curentă
Toate metodele primesc ca prim parametru obiectul curent (self)

```
def testCreate():
    """
      Test function for creating rational numbers
    """
    r1 = RationalNumber(1,3)  #create the rational number 1/3
    assert r1.getNominator()==1
    assert r1.getDenominator()==3
    r1 = RationalNumber(4,3)  #create the rational number 4/3
    assert r1.getNominator()==4
    assert r1.getDenominator()==3

class RationalNumber:
    """
      Abstract data type rational numbers
      Domain: {a/b  where a,b integer numbers, b!=0, greatest common divisor
a, b =1}
    """
    def __init__(self, a, b):
        """
          Initialize a rational number
          a,b integer numbers
        """
        self.__nr = [a, b]


    def getDenominator(self):
        """
           Getter method
           return the denominator of the rational number
        """
        return self.__nr[1]

    def getNominator(self):
        """"
          Getter method
          return the nominator of the method
        """
        return self.__nr[0]
```

Metode speciale. Supraîncărcarea operatorilor. (Operator overloading)

__str__ - conversie in tipul string (print representation)

```
def __str__(self):
        """
          provide a string representation for the rational number
          return a string
        """
        return str(self.__nr[0])+"/"+str(self.__nr[1])
```

__lt__ , **__le__**, **__gt__**, **__ge__**- comparații (<,<=,>,>=)

```
def testCompareOperator():
    """
    Test function for < >
    """
    r1 = RationalNumber(1, 3)
    r2 = RationalNumber(2, 3)
    assert r2>r1
    assert r1<r2
```

```
def __lt__(self, ot):
        """
          Compare 2 rational numbers (less than)
          self the current instance
          ot a rational number
          return True if self<ot,False otherwise
        """
        if self.getFloat()<ot.getFloat():
            return True
        return False
```

__eq__ - verify if equals

```
def testEqual():
    """
      test function for ==
    """
    r1 = RationalNumber(1, 3)
    assert r1==r1
    r2 = RationalNumber(1, 3)
    assert r1==r2
    r1 = RationalNumber(1, 3)
    r1 = r1.add(RationalNumber(2, 3))
    r2 = RationalNumber(1, 1)
    assert r1==r2
```

```
def __eq__(self, other):
        """
        Verify if 2 rational are equals
        other - a rational number
        return True if the instance is
equal with other
        """
        return self.__nr==other.__nr
```

Operator overloading

__add__(self, other) - pentru a folosi operatorul "**+**"

```
def testAddOperator():
    """
     Test function for the + operator
    """
    r1 = RationalNumber(1,3)
    r2 = RationalNumber(1,3)
    r3 = r1+r2
    assert r3 == RationalNumber(2,3)
```

```
def __add__(self,other):
        """
        Overload + operator
          other  - rational number
        return a rational number,
          the sum of self and other
        """
        return self.add(other)
```

Metoda **__mul__(self, other)** - pentru operatorul "*"

Metoda **__setItem__(self,index, value)** – dacă dorim ca obiectele noastre sa se comporte similar cu liste/dicționare, sa putem folosi "[]"

a = A()
a[index] = value

__getItem__(self, index) – sa putem foloi obiecul ca si o secvență

a = A()
for el in a:
  pass

__len__(self) - pentru len

__getslice__(self,low,high) - pentru operatorul de slicing

a = A()
b = a[1:4]

__call__(self, arg) - to make a class behave like a function**,** use the "()"

a = A()
a()

Vizibilitate și spații de nume în Python

Spațiu de nume (*namespace)* este o mapare intre nume și obiecte
Namespace este implementat în Python folosind dicționarul
Cheie: Nume
Valoare – Object

Clasa introduce un nou spațiu de nume
Metodele sunt într-un spațiu de nume separat, spațiu de nume corespunzător clasei.

```
class Student:
    def __init__(self,nume, prenume):
        self.__nume = nume
        self.__prenume = prenume

    def getNume(self):
        return self.__nume

    def getFullName(self):
        return "{} {}".format(self.__nume,self.__prenume)

print (Student.__dict__)
```

```
{'__module__': '__main__', '__doc__': None, 'getNume': <function
Student.getNume at 0x0089F8A0>, '__dict__': <attribute '__dict__' of
'Student' objects>, '__init__': <function Student.__init__ at 0x0089F810>,
'getFullName': <function Student.getFullName at 0x0089F858>, '__weakref__':
<attribute '__weakref__' of 'Student' objects>}
```

Fiecare instanța de clasa are propriu spațiu de nume (aici se țin atributele instanței)

```
st1 = Student("Ion","Vasilescu")
print (st1.__dict__)
```

```
{'_Student__nume': 'Ion', '_Student__prenume': 'Ionescu'}
```

Toate regulile (legare de nume, vizibilitate/scope, parametrii formali/actuali, etc.) legate de denumiri (funcții, variable) sunt același pentru atributele clasei (metode, câmpuri) ca si pentru orice alt nume in Python, doar trebuie luat în considerare ca avem un namespace dedicat clasei

Atribute de clasă vs atribute de instanțe

Variabile membre (câmpuri)

- atribute de instanțe – valorile sunt unice pentru fiecare instanță (obiect)
- atribute de clasă – valoarea este partajata de toate instanțele clasei (toate obiectele de același tip)

```
class RationalNumber:
    """
       Abstract data type for rational numbers
       Domain: {a/b where a and b are integer numbers b!=0}
    """
    #class field, will be shared by all the instances
    numberOfInstances =0

    def __init__(self, a, b):
        """
          Creates a new instance of RationalNumber
        """
        self.n = a
        self.m = b
        RationalNumber.numberOfInstances+=1   # accessing class fields

def testNumberInstances():
    assert RationalNumber.numberOfInstances == 0
    r1 = RationalNumber(1,3)
    #show the class field numberOfInstances
    assert  r1.numberOfInstances==1
    # set numberOfInstances from the class
    r1.numberOfInstances = 8
    assert r1.numberOfInstances==8  #access to the instance field
    assert RationalNumber.numberOfInstances==1  #access to the class field

testNumberInstances()
```

Metode statice

Funcții din clasă care nu operează cu o instanța.

```
class RationalNumber:
    #class field, will be shared by all the instances
    numberOfInstances =0

    def __init__(self,n,m):
        """
         Initialize the rational number
         n,m - integer numbers
        """
        self.n = n
        self.m = m
        RationalNumber.numberOfInstances+=1

    @staticmethod
    def getTotalNumberOfInstances():
        """
          Get the number of instances created in the app
        """
        return RationalNumber.numberOfInstances

    @classmethod
    def fromString(cls,s):
        """
          Create a Rational numbar obiect from its string reprezentation
          cls - class
          s - string representation 1/3
        """
        parts = s.split("/")
        return RationalNumber(int(parts[0]),int(parts[1]))

def testNumberOfInstances():
    """
     test function for getTotalNumberOfInstances
    """
    assert RationalNumber.getTotalNumberOfInstances()==0
    r1 = RationalNumber(2, 3)
    assert RationalNumber.getTotalNumberOfInstances()==1

testNumberOfInstances()
```

ClassName.attributeName – folosit pentru a accesa un atribut asociat clasei (câmp, metoda)

Decoratorul **@staticmethod** este folosit pentru a marca o funcție statică. Aceste funcții nu au ca prim argument (self) obiectul curent.

Decoratorul **@classmethod** similar cu @staticmethod dar se primeste un prim parametru clasa

Principii pentru crearea de noi tipuri de date

Încapsulare

Datele care reprezintă starea și metodele care manipulează datele sunt strâns legate, ele formează o unitate coezivă.
Starea si comportamentul ar trebui încapsulat în același unitate de program (clasa)

Ascunderea informațiilor

Reprezentarea interna a obiectelor (a stării) trebuie protejat față de restul aplicației. Ascunderea reprezentării protejează integritatea datelor și nu permite modificarea stării din exteriorul clasei, astfel se evită setarea, accidentala sau voita, unei stări inconsistente.

Clasa comunica cu exteriorul doar prin interfața publică (mulțimea tuturor metodelor vizibile in exterior) și ascunde orice detalii de implementare (modul în care am reprezentat datele, algoritmii folosiți, etc).

De ce:

Definirea unei interfețe clare și ascunderea detaliilor de implementare asigură ca alte module din aplicație sa nu pot face modificări care ar duce la stări inconsistente. Permite evoluția ulterioară (schimbare reprezentare, algoritmi etc) fără să afectăm restul aplicației

Limitați interfața (metodele vizibile în exterior) astfel încât să existe o libertate în modificarea implementării (modificare fără a afecta codul client)

Codul client trebuie să depindă doar de interfața clasei, nu de detalii de implementare. Dacă folosiți acest principiu, atunci se pot face modificări fără a afecta restul aplicației

Membri publici. Membrii privați – Ascunderea implementării in Python

Trebuie sa protejăm (ascundem) reprezentarea internă a clasei (implementarea)

In Python ascunderea implementării se bazează pe convenții de nume.
_name sau __name pentru un atribut semnalează faptul ca atributul este “privat”

Un nume care începe cu _ sau __ semnalează faptul ca atributul (câmp, metode) ar trebui tratat ca fiind un element care nu face parte din interfața publică. Face parte din reprezentarea internă a clasei, nu ar trebui accesat din exterior.

Recomandări

- Creați metode pentru a accesa câmpurile clasei (getter)
- folosiți convențiile de nume _,__ pentru a delimita interfața publică a clasei de detaliile de implementare
- Codul client ar trebui sa funcționeze (fără modificări) chiar daca schimbăm reprezentarea internă, atâta timp cât interfața publică rămâne neschimbată. Clasa este o abstractizare, o cutie neagra (black box)
- Specificațiile funcțiilor trebuie sa fie independente de reprezentare

Cum creăm clase

Folosim Dezvoltare dirijată de teste

Specificațiile (documentația) pentru clase includ:

- scurtă descriere
- domeniul – ce fel de obiecte se pot crea. În general descrie câmpurile clasei
- Constrângeri ce se aplică asupra datelor membre: Ex. Invariant – condiții care sunt adevărate pentru întreg ciclu de viața al obiectului

```
class RationalNumber:
    """
      Abstract data type rational numbers
      Domain:{a/b where a,b integer numbers, b!=0, greatest common divisor a, b =1}
      Invariant:b!=0, greatest common divisor a, b =1
    """
    def __init__(self, a, b):
```

Se creează funcții de test pentru:

- Crearea de instanțe
- Fiecare metodă din clasă

Câmpurile clasei (reprezentarea) se declară private (__nume). Se creează metode getter pentru a accesa câmpurile clasei

Tipuri abstracte de date (Abstract data types)

Tip abstract de date:

- operațiile sunt specificate independent de felul în care operația este implementată
- operațiile sunt specificate independent de modul de reprezentare a datelor

Un tip abstract de date este: Tip de date+ Abstractizarea datelor + Încapsulare

Review Calculator rațional – varianta orientat obiect

Putem schimba cu ușurința reprezentarea internă pentru clasa RationalNumber (folosim a,b în loc de lista [a,b])

Curs 5: Tipuri definite de utilizator

- Programare orientată obiect
- Principii de definire a tipurilor utilizator
- Tip abstract de date

Curs 6: Principii de proiectare

- Diagrame UML
- Șabloane GRASP