Fundamentele programării

Excepții recapitulare

Funcțiile pot arunca excepții pentru a semnala o eroare.

```
int("bla")
```

```
Traceback (most recent call last):
  File "C:\Curs5\ex.py", line 6, in <module>
    int("bla")
ValueError: invalid literal for int() with base 10: 'bla'
```

Funcțiile create de noi pot arunca și ele excepții.

```
def myBeautifulFunction():
    #.....
    raise ValueError("Motivul pentru care arunc exceptie")
    #.....

myBeautifulFunction()
```

```
Traceback (most recent call last):
  File "C:\Curs5\ex.py", line 13, in <module>
    myBeautifulFunction()
  File "C:\Curs5\ex.py", line 10, in myBeautifulFunction
    raise ValueError("Motivul pentru care arunc exceptie")

ValueError: Motivul pentru care arunc exceptie
```

Tratarea excepțiilor

Excepțiile pot fi tratate în blocul de instrucțiuni unde apar sau în orice bloc exterior care a apelat blocul în care a apărut excepția.

```
try:
    #cod in care posibil apar exceptii (arunca exceptii)
    myBeautifulFunction()
    #....
except ValueError as ex:
    #executa acest cod daca a aparut eroarea ValueError
    print(ex) #putem accesa obiectul care a aruncat eroare
except ZeroDivisionError:
    #executa acest cod daca a aparut eroarea ZeroDivisionError
    pass
finally:
    #executa tot timpul(si daca a aparut exceptie si daca nu)
    pass
```

Clasă

Definește caracteristicile unui lucru.
O clasă are două tipuri de atribute:

- câmpuri
- metode

Clasele se folosesc pentru crearea de noi tipuri de date

Definiție de clasă în python

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

Introduce un nou tip de date cu numele specificat.
Clasa are un namespace propriu, definițiile de funcții din interiorul clasei introduc numele funcțiilor în acest spațiu de nume nou creat. Similar și pentru variabile.

Obiect

Obiectele sunt o colecție de date și funcții care operează cu aceste date.

Fiecare obiect este un tip, este de tipul clasei asociate: este instanța clasei.

Obiectul:

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

Fiecare obiect are propriul namespace care conține câmpurile și metodele.

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

Instanțierea unei clase rezultă în obiecte noi (instanțe). Crearea obiectelor se face prin atribuirea clasei.

```
x = MyClass()
```

operația de instanțiere 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

metoda __init__ poate avea mai mulți parametri:

```
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
```

Câmpuri:

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

self.n = a vs n = a în contextul unei clase:

- self.n = a: n devine un atribut al instanței și poate fi folosit oriunde în instanța clasei.
- n = a: n este o variabilă locală în acea metodă și dispare când metoda se termină.

Metode

Metodele sunt funcții definite în interiorul clasei care au acces doar la câmpurile unei instanțe.

În Python metodele au acces la la valorile câmpurilor unei instanțe (adică cele cu self)

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

Sunt funcții predefinite în Python, încep și se termină cu __.

Aceste metode permit modificarea unor operatori atunci când le aplicăm unui obiect sau între obiecte.

1. __init__(self) : este apelată când creezi o instanță (obiect)
2. __str__(self) . este apelată când convertim un obiect într-un string. Ex: print(obj)

```
class MyClass:
    def __str__(self):
        return "Acesta este un obiect din MyClass"

obj = MyClass()
print(obj)  # Output: Acesta este un obiect din MyClass
```

3. __eq__(self, other) . este apelată când comparăm două obiecte cu operatorul ==

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def __eq__(self, other):
        return self.value == other.value

obj1 = MyClass(10)
obj2 = MyClass(10)
print(obj1 == obj2)  # Output: True, pentru că valoarea lor este aceeași
```

4. __lt__ (← le, gt, ge) (self, other) . este apelată când comparăm două obiecte cu operatorul: <, <=, >, >=

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def __lt__(self, other):
        return self.value < other.value

obj1 = MyClass(5)
obj2 = MyClass(10)
print(obj1 < obj2)  # Output: True, pentru că 5 < 10
```

5. `__add__(self, other)`: operatorul "+"

6. `__mul__(self, other)`: operatorul "*"

7. `__len__(self)`: operatorul "len"

8. `__setitem__(self, index, value)`: dacă dorim ca obiectul să se comporte ca listă / dicționare

```
class MyList:
    def __init__(self, initial_data=None):
        # Inițializăm lista internă cu datele primite sau o listă goală
        self.data = initial_data if initial_data is not None else []

    def __setitem__(self, index, value):
        # Verificăm dacă indexul este valid
        if index >= len(self.data):
            print(f"Extind lista pentru a seta valoarea la indexul {index}")
            # Extindem lista pentru a putea seta un index mai mare decât dimensiunea curentă
            self.data.extend([None] * (index - len(self.data) + 1))
        print(f"Setăm valoarea {value} la indexul {index}")
        self.data[index] = value
```

```
class MyList:
    def __init__(self):
        self.data = {}

    def __setitem__(self, key, value):
        self.data[key] = value
        print(f'Set item at index {key} to {value}')
```

9. `__getitem__(self, index)`: să putem folosi obiectul ca o secvență

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        if isinstance(index, slice):  # Verifică dacă este o operație de slicing
            print(f'Slicing from {index.start} to {index.stop}')
            return self.items[index]
        else:
            return self.items[index]

# Utilizare
my_list = MyList([1, 2, 3, 4, 5])
print(my_list[1:4])  # Apelează __getitem__ cu slicing
```

10. `__call__(self, arg)`: permite un obiect să fie apelat ca o funcție

```
class Greeter:
    def __init__(self, name):
        self.name = name

    def __call__(self):
        return f"Hello, {self.name}!"

greeter = Greeter("Alice")
print(greeter())  # Output: Hello, Alice!
```

Atribute de clasă vs atribute de instanță

- atribute de instanțe – valorile sunt unica pentru fiecare instanță
- atribute de clasă – valoarea este partajată de toate instanțele clasei

```
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

Sunt funcții din clasă care nu operează cu o instanță.

@staticmethod este folosit pentru a marca o funcție statică.

Aceste funcții nu au ca prim argument self.

@classmethod este folosit pentru a marca o funcție statică.

Aceste funcții au ca prim parametru clasa.

```
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()
```

Principii pentru crearea de noi tipuri de date

- Încapsulare:
  Datele care reprezintă starea și metodele care manipulează datele sunt strâns legate.
- Ascunderea informațiilor:
  Starea obiectelor trebuie protejată față de restul aplicației.
  Ascunderea reprezentării protejează integritatea datelor și nu permite modificarea stării din exteriorul clasei.

Membri publici. Membri privați

Ascunderea datelor se bazează pe ___, atributul ___ x este privat.

Cum creăm clasa

Folosim Dezvoltare dirijată de teste.

Specificațiile includ:

- scurtă descriere
- domeniul - ce fel de obiecte se pot crea
- restricții asupra datelor

```python
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 se declară private. Se creează metode getter pentru a accesa câmpurile.