User Defined Types

Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

October 28, 2015

Overview

User defined types - classes and objects

- **Object oriented programming** - a programming paradigm that uses objects to design applications.
- **Remember!** - **Types** classify values. A type denotes a **domain** (a set of values) and **operations** on those values.

Let's review modular calculator

Issues with the global variables version:

- You can easily break global vars!
- They make testing difficult
- The relation between them is difficult

Issues with the no-global variables version:

- The state of the calculator is exposed to the world
- The state has to be transmitted as parameter to every function

User defined types - classes

Class - a construct that is used as a template to create instances of itself - referred to as class instances, class objects, instance objects or simply **objects**. A class defines constituent members which enable these class instances to have *state* and behaviour.

Classes in Python

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

- The class definition is an executable statement.
- The statements inside a class definition are usually function definitions, but other statements are allowed
- When a class definition is entered, a new namespace is created, and used as the local scope - thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

User defined types - objects

Object - a collection of data and functions that operates on that data. Class instances are of the type of the associated class. Objects support two kinds of operations: attribute (data or method) references and instantiation.

User defined types - objects

Creating instances of a class (__init__) Class *instantiation* uses function notation.

```
x = MyClass()
```

- The instantiation operation (calling a class object)creates an empty object. x will be an instance of type MyClass

User defined types - objects

- A class may define a special method named __init__

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

__init__

- Create an instance
- Use self to refer to that instance

User defined types - objects


```
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
        """
        self.n = a
        self.m = b

r1 = RationalNumber(1,3)  #create the rational number 1/3
```

self.n = a vs n = a

1. Creates an attribute for the current instance
2. Creates a function local variable

Fields

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

Fields

```
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. Creates an attribute for the current instance
2. Creates a function local variable

Methods

- **Methods** - functions in a class that can access values from a specific instance.
- In Python the method will automatically receive a first argument: the current instance
- All the methods need to have an argument (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
```

Methods

```
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]
```

Special methods

__str__ - convert into a string type (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])
```

Special methods

Comparisons - $<, >, \leq, \geq$

```
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
```

Special methods

__eq__ - verify if equals

```python
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
```

```python
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

Overload **__add__(self, other)** - to be able to use " +" operator

```
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)
```

Operator overloading

- Overload **__mul__(self, other)** - to be able to use the "*" operator
- Overload **__setItem__(self,index, value)** - to make a class behave like an array/dictionary, use the "[]"
  $a = A()$
  $a[index] = value$
- **__getItem__(self, index)** - to make a class behave like an array
  $a = A()$
  for el in a:
  pass

Operator overloading

- **__len__(self)** - overload len
- **__getslice__(self,low,high)** - overload slicing operator
  $a = A()$
  $b = a[1:4]$
- **__call__(self, arg)** - to make a class behave like a function, use the "()"
  $a = A()$
  $a()$

Python scope and namespace

- A *namespace* is a mapping from names to objects.
- Namespaces are implemented as Python dictionaries
  - Key: name
  - Value - Object
- A class introduce a new name space
- Methods and fields of a class are in a separate namespace (the namespace of the class)
- All the rules (bound a name, scope/visibility, formal/actual parameters, etc.) related to the names (function, variable) are the same for class attributes (methods, fields). Just keep in mind that the class have it's own namespace

Data members (fields)


```python
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

    numberOfInstances =0
    def __init__(self,n,m):
        self.n = n
        self.m = m
        RationalNumber.numberOfInstances+=1
```

Data members (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()
```

Class Methods

```
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
```

Class Methods


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

testNumberOfInstances()
```

- *ClassName.attributeName* - used to access the class attributes (fields, methods)
- The *staticmethod* form is a function decorator. A static method does not receive an implicit first argument.

Encapsulation

- The data that represents the state of the object and the methods that manipulate that data are stored together as a cohesive unit. State and behaviour are kept together, they are encapsulated.

Information hiding

- The internal representation of an object need to be hidden from view outside of the object's definition
- Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state
- Divide the code into a public interface, and a private implementation of that interface

Information hiding

Why?

- Defining a specific interface and isolate the internals to keep other modules from doing anything incorrect to your data
- Limit the functions that are visible (part of the interface),so you are free to change the internal data without breaking the client code
- Write to the Interface, not the the Implementation
- If you are using only the public functions you can change large parts of your classes without affecting the rest of the program

Public and private members - Data hiding in Python

- We need to protect (hide) the internal representation (the implementation) Provide accessors (getter) to the data
- Encapsulations is particularly important when the class is used by others
- Nothing in Python makes it possible to enforce data hiding - it is all based upon convention. use the convention: _name or __name for fields, methods that are " private"
- A name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Guidelines

- Client code does not have to know about the implementation details of the methods or the internal data representation (abstraction, the class is a black box)
- Client code needs to work even if we change the implementation or data representation
- Function and class specification have to be independent of the data representation and the method's implementation (Data Abstraction)

Abstract data types

- Operations are specified independently of their implementation
- Operations are specified independently of the data representation
- Abstract data type is a Data type + Data Abstraction + Data Encapsulation

SimplestClassExample

Try to run the code in the provided example (same directory as Lecture Notes).

```
class FirstClass:
    pass

class SecondClass:
    def __init__(self):
        self.testOne = "Test One"
        self._testTwo = "Test Two"
        self.__testThree = "Test Three"
```