BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Algorithms and Programming

Lecture 5 – Classes

Camelia Chira

Course content

- Introduction in the software development process
- Procedural programming
- Modular programming
- **Abstract data types**
- Software development principles
- Testing and debugging
- Recursion
- Complexity of algorithms
- Search and sorting algorithms
- Backtracking
- Recap

Last time

- Software Design Principles
- Working with files in Python

Files

- Working with files in Python
  - Processing files using file type objects
  - Operations with file objects
    - Open
    - Read
    - Write
    - Close
  - Exceptions

File operations

- Open a file – create an instance of a file type object
  - Function `open(filename, openMode)`
    - `filename` is a string containing the name of the file (and its path)
    - `openMode`
      - “r” – open to read
      - “w” – open to write
      - “a” – open to add

```
def read_from_file():
        fin = open("../data.txt", "r")
         # read operations
        fin.close()
```

```
def read_from_file():
        try:
            fin = open("../data.txt", "r")
            for line in fin:
                print(line)
            fin.close()
        except IOError as ex:
            print("Reading File errors: ", ex)
```

File operations

- Write to a file
  - Function `write(info)` – writes the string info to a file
    - If `info` is a number or a list – conversion to string and then write to file
    - If `info` is an object (instance of a class) – the class must implement the `__str__` method

```
def appendToFile():
    fout = open("test.out", "a")
    fout.write("\n appended lines\n")
    x = 5
    fout.write(str(x))
    fout.close()
```

```
def writeToFile():
    fout = open("test.out", "w")
    fout.write("first file test...")
    x = 5
    fout.write(str(x))
    l1 = [2, 3, 4]
    fout.write(str(l1))
    l2 = ["aw", "ert", "45", "GGGGGGGGG"]
    fout.write(str(l2))
    fout.close()
```

File operations

- Read from a file
  - `readline()` – reads a line from a file and returns it as string
  - `read()` – reads all lines from a file and returns the content as a string
- Close a file
  - `close()` – closes the file, possibly making the memory space free

```
def readFromFileALine():
    fin = open("test.in", "r")
    line = fin.readline()
    print(line)
    fin.close()
```

```
def readFromFileAllLines():
    fin = open("test.in", "r")
    line = fin.readline()
    while (line != ""):
        print(line)
        line = fin.readline()
    fin.close()
```

```
def readEntireFile():
    fin = open("test.in", "r")
    line = fin.read()
    print(line)
    fin.close()
```

Working with files

- Exceptions
  - `IOError`
  - When one of the file operations (open, read, write, close) can not be executed
    - The file that has to be opened is not found
    - The file can not be opened (technical reasons)
    - The file can not be written to (no more disk space)

```
def runFiles():
    try:
        writeToFile()
        appendToFile()
        readFromFileALine()
        readFromFileAllLines()
        readEntireFile()
    except IOError as ex:
        print("some errors: ", ex)
```

Open function

fileObject = `open(filename, openMode)`

`openMode:`

r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

```
def processFile1():
    try:
        f = open("data.txt", "r")
        #some operations on the file
        f.close()
    except IOError as e1:
        print("something is wrong as IO..." + str(e1))
    except ValueError as e2:
        print("something is wrong as value..." + str(e2))
```

```
something is wrong as IO...[Errno 2] No such file
or directory: 'data.txt'
```

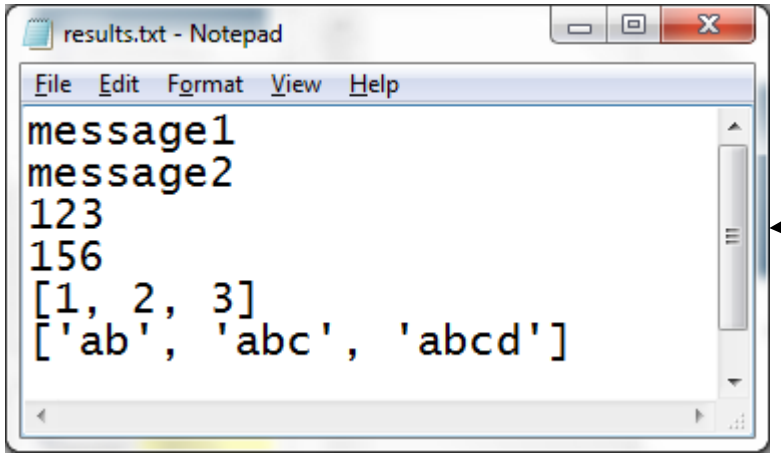
Example

- Reading text files

```
def processFile2():
    try:
        f = open("data.txt", "r")
        allLines = f.read() #read all the lines
        for char in allLines: #consider each char from lines
            print(char)

        lines = allLines.split("\n")
        for line in lines: #consider each line from the file
            print(line)

        for line in lines: #consider each line from the file
            words = line.split(" ")

        for w in words:
            print(w)

        lastLine = lines[2]
        words = lastLine.split(" ")
        s = 0
        for w in words:
            s = s + int(w)
        print("sum = ", s)

        f.close()
    except IOError as e1:
        print("something is wrong as IO..." + str(e1))
    except ValueError as e2:
        print("something is wrong as value..." + str(e2))
```

Example

- Writing to text files




```
def processFile3():
    try:
        f = open("results.txt", "w")

        f.write("message1" + "\n")
        f.write("message2\n")

        f.write(str(123) + "\n")
        f.write(str(156) + "\n")

        l1 = [1,2,3]
        f.write(str(l1) + "\n")

        l2 = ["ab", "abc", "abcd"]
        f.write(str(l2) + "\n")

        f.close()
    except IOError as e1:
        print("something is wrong as IO..." + str(e1))
    except ValueError as e2:
        print("something is wrong as value..." + str(e2))
```

Python object serialization

- Serialization: transform an object to a format that can be stored so that the original object can be recreated later

- Python
  - Serialization in binary files
    - Marshal
    - Pickle
  - Serialization in text files
    - JSON

Pickle

- The serialization process in Python
- pickle module
- Pickling
  - Convert an object to a binary format that can be stored
  - dump function
- Unpickling
  - The process that takes a binary array and converts it to an object
  - load function

```
import pickle as pk

def processFile4():
    data = [1, 2, 3]
    try:
        f = open("res.pk", "wb")
        pk.dump(data, f)
        f.close()
    except IOError as e1:
        print("something is wrong as IO..." + str(e1))
    except ValueError as e2:
        print("something is wrong as value..." + str(e2))

def processFile5():
    try:
        f = open("res.pk", "rb")
        data = pk.load(f)
        print(data)
        f.close()
    except IOError as e1:
        print("something is wrong as IO..." + str(e1))
    except ValueError as e2:
        print("something is wrong as value..." + str(e2))
```

Today

- Abstract data types (ADT) or user-defined data types

- Developing classes using Python

Abstract data types

- Object-oriented programming
  - Concepts
  - Working principles

- Classes
  - Concept
  - How to define and use in Python

Object-oriented programming (OOP)

- Develop programs using
  - Objects – basic unit, each an instance of a class
  - Classes – links and inheritance

- Objects
  - Ex. “abc”, 12, [1, 2, 3]
  - Each object has a type, internal data representation and a set of procedures that can be used to interact with the object
  - An object is an instance of a type

OOP Concepts

- Class
  - Defines in an abstract way the characteristics of a thing:
    - Characteristics (attributes, fields, properties)
    - Behaviour (methods, operations)
  - Implementation
  - Creating a class: define the class name and the class attributes
  - Using a class: creating new instances of the class and using operations on it
- Object
  - Instance of a class
  - Attributes (the internal representation defined by the class)
  - Interface with the object using methods or functions (define the behavior)
- Method
  - They form the interface of an object
  - Objects communicate via methods

OOP: Example

- `my_list = [1, 2, 3]`
  - list in Python
  - An object
  - Internal representation?
  - Methods?
    - `len(my_list)`
    - `my_list.append(4)`
    - `del(my_list[1])`
    - etc

OOP characteristics

- **Encapsulation**
  - Capturing data and keeping it safely and securely from outside interfaces
  - Hiding the implementation – control the access
- **Inheritance**
  - A class can be derived from a base class with all features of base class and some of its own
  - Increases code reusability (reuse and improve code from a class)
- **Polymorphism**
  - An object of a class can be used in the same way as if it were a different object belonging to a different class
  - Flexibility and loose coupling – code can be extended and easily maintained over time

Creating your own types with classes

- Abstract Data Type
  - Export a name (a data type)
  - Define a domain of values for the data
  - Define an interface (the operation possible with the new data type)
  - Restrict access to the components of the new type (access only through methods)
  - Hide the implementation of the new type
- Create the class vs. using an instance of the class
- Use the `class` keyword to define a new type

Creating your own types: example 1

- Abstract Data type: Rational Number
  - Name: `Rational`
  - Domain

    $$\{(a,b), a, b \in Z, b \neq 0, \gcd(a,b) = 1\}$$
  - Operations:
    - Initialization
    - Acces to components (nominator, denominator)
    - Copy
    - Comparison
    - Add/subtract/multiply/divide/etc
    - ...

Creating your own types: example 2

- User-defined type: Flower
  - Name: `Flower`
  - Domain

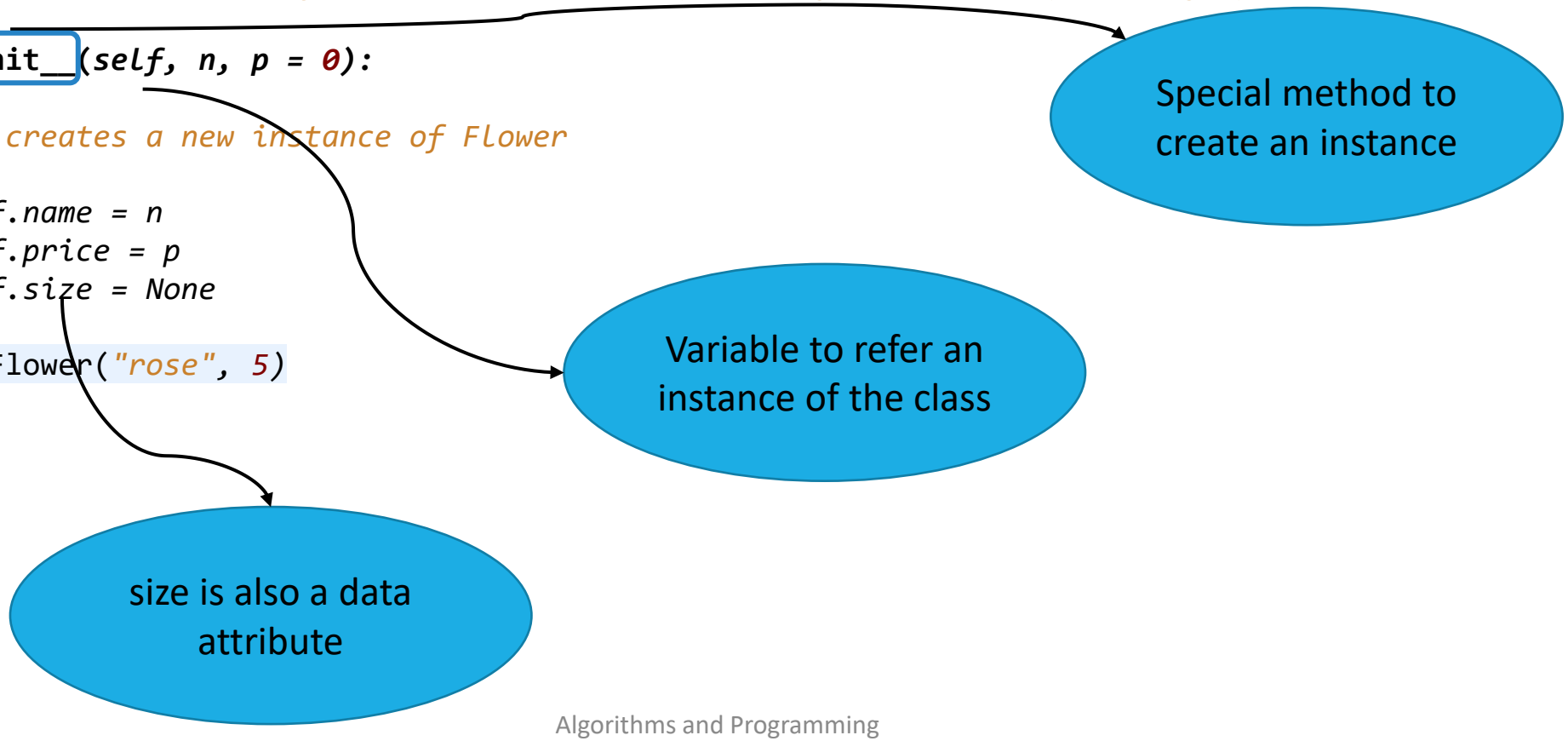
    $\{(name, price), name\ -string, price \in N\}$
  - Operations:
    - Initialization
    - Acces to attribute values
    - Copy
    - Comparison
    - ...

Abstract Data Type (ADT)

- Exporting a name (a data type)
- Define a domain of values for the data
- Define an interface (the operation possible with the new data type)
- Restrict access to the components of the ADT (access only through methods)
- Hide the implementation of ADT

How to define a class

```
class Flower:
    '''
    a flower is a structure of two elements: name (a string) and price (an integer)
    '''
    def __init__(self, n, p = 0):
        '''
            creates a new instance of Flower
        '''
        self.name = n
        self.price = p
        self.size = None

myFlower = Flower("rose", 5)
```



Abstract Data Type (ADT)

- Define the class
  - Specify:
    - Data attributes
    - Methods
  - The name of the new type is the class name: `class Flower:`
  - Use `self` to refer to instances while defining the class:
    - `self.name, self.size < 10`
    - `self` is a parameter of methods
  - Data and methods in the class are the same for all instances of class
- An instance is a specific object: `f1 = Flower("rose", 5)`
  - Attribute values can vary `f2 = Flower("tulip", 3)`

Creating a class in Python

- Class
  - Describes objects that follow the same specification and have the same characteristics
  - Attributes
    - The data describing the objects
  - Methods (procedural attributes)
    - The operations that can be performed on the data
- Class definition

```
class ClassName:
#statement1
#...
#statement n
```

```
class Rational:
    '''
    A rational number is composed by 2 numbers: numerator and denominator > 0
    denominator <> 0, gcd(numerator, denominator) == 1
    '''
    def __init__(self, num, denom):
        '''
            creates a new instance of Rational
        '''
        self.numerator = num
        self.denominator = denom
```

In Python, use a special method called **__init__** to initialize data attributes

Creating an instance of the class

- Object
  - An instance of a class
  - When a new object variable is created, the type has to be indicated (e.g. Rational)

```
class Rational:
    '''
    A rational number is composed by 2 numbers: numerator and denominator > 0
    denominator <> 0, gcd(numerator, denominator) == 1
    '''
    def __init__(self, num, denom):
        '''
            creates a new instance of Rational
        '''
        self.numerator = num
        self.denominator = denom

r = Rational(2, 3)
```

```
>>> r
<__main__.Rational object at 0x02E74F30>
>>> r.numerator
2
>>> r.denominator
3
>>>
```

Creating classes: remarks

- Defining a class creates a new namespace (used as local scope – the variables and functions defined by the class will belong to this namespace)
- Every object (instance of the class) has its own namespace / symbol table that contains the attributes and functions of the object
- To initialize an object, a class uses the `__init__` method which:
  - Is automatically called when a new object is created
  - Has at least one parameter (***self***) which refers to the object created
  - Can have other parameters (***num, denom***)
    - Can have default values

```
class Rational:

def __init__(self, num = 0, denom = 1):
        '''
            creates a new instance of Rational
        '''
        self.numerator = num
        self.denominator = denom
```

```
r1 = Rational(2,3) #r1 = 2/3
r2 = Rational(3)   #r2 = 3/1
r3 = Rational()    #r3 = 0/1
r4 = Rational(denom=5) #r4= 0/5
```

Adding methods to a class

- Methods
  - Functions defined inside a class that can access data (values, attributes) from the class
  - The first parameter of any method is the current instance (object) – `self`
  - Methods can be called by `objectName.methodName(parameterList)`

```
class Rational:
    '''
    A rational number is composed by 2 numbers: numerator and denominator > 0
    denominator <> 0, gcd(numerator, denominator) == 1
    '''
    def __init__(self, num = 0, denom = 1):
        # creates a new instance of Rational
        self.numerator = num
        self.denominator = denom

    def getNumerator(self):
        # getter method
        # return the numerator of the rational number
        return self.numerator

     def getDenominator(self):
        # getter method
        # return the denominator of the rational number
        return self.denominator
```

```
def test_create():
    r1 = Rational(2,3)
    assert r1.getNumerator() == 2
    assert r1.getDenominator() == 3

    r2 = Rational(5,4)
    assert r2.getNumerator() == 5
    assert r2.getDenominator() == 4
```

```
r1 = Rational(2,3)  #r1 = 2/3

print("r1 = ", r1.getNumerator(), "/", r1.getDenominator())
```

Adding methods to a class

```
from utils import gcd
class Rational:
    ...
    def __init__(self, num = 0, denom = 1):
        # ...

    def getNumerator(self):
        # ...

    def getDenominator(self):
        # ...

    def add(self, other):
        '''
            add two rational numbers (self + other)
            return a new rational number self = self + other
        '''
        a = self.numerator * other.denominator + self.denominator * other.numerator
        b = self.denominator * other.denominator
        d = gcd(a, b)
        self.numerator = a // d
        self.denominator = b // d
        return self
```

```
def test_add():
    r1 = Rational(2,3)
    r2 = Rational(5,4)
    r3 = r1.add(r2)
    assert r3.getNumerator() == 23 and r3.getDenominator() == 12
```

Special methods

```
r1 = Rational(2,3)
print(r1)
```

```
<__main__.Rational object at 0x029FEEB0>
```

Provide a __str__ (double underscores before/after) method in the class

```
def __str__(self):
    return str(self.numerator) + "/" + str(self.denominator)
```

```
r1 = Rational(2,3)
print(r1)
```

```
2/3
```

Special methods – Python

- String conversion- define: `__str__` , use: `str(...)`
- Comparisons- define `__eq__` , use `==, !=`

```
>>> r1 = Rational(2,3)
>>> str(r1)
'2/3'
>>> r2 = Rational(2,3)
>>> r1 == r2
True
>>> r3 = Rational(5,3)
>>> r1 == r3
False
>>> r1 != r3
True
>>>
```

```
class Rational:
    ...
    def __str__(self):
        '''
            provides a string representation of a rational number
            return a string
        '''
        return str(self.numerator) + "/" + str(self.denominator)

    def __eq__(self, other):
        '''
            compares 2 rational numbers: self and other
            return True, if self == other
                   False, otherwise
        '''
        if ((self.numerator == other.numerator) and (self.denominator == other.denominator)):
            return True
        else:
            return False
```

```
def test_str():
    r1 = Rational(2,3)
    assert r1.__str__() == "2/3"

def test_eq():
    r1 = Rational(2,3)
    r2 = Rational(2,3)
    assert r1 == r2
    r3 = Rational(5,3)
    assert r1 != r3
```

Special methods

`__str__(self)`	`print(self)` `str(...)`
`__eq__(self, other)`	`self == other`
`__add__(self, other)`	`self  + other`
`__sub__(self, other)`	`self - other`
`__lt__(self, other)`	`self < other`
`__len__(self)`	`len(self)`
`...`	

https://docs.python.org/3/reference/datamodel.html#basic-customization

Methods – follow the specifications

```
class Rational:
    '''
    A rational number is composed by 2 numbers: numerator and d
    denominator <> 0, gcd(numerator, denominator) == 1
    '''
    def __init__(self, num = 0, denom = 1):
        '''
            creates a new instance of Rational
        '''
        if (denom == 0):
            raise ValueError("0 denominator not allowed")
        if (num < 0) or (denom < 0):
            raise ValueError("numerator and denominator must be
        d = gcd(num, denom)
        self.numerator = num // d
        self.denominator = denom // d

    def getNumerator(self):
        ...

    def getDenominator(self):
        ...
```

```
def test_create():
    r1 = Rational(2,3)
    assert r1.getNumerator() == 2 and r1.getDenominator() == 3
    r2 = Rational(5,4)
    assert r2.getNumerator() == 5 and r2.getDenominator() == 4
    r3 = Rational(25, 15)
    assert r3.getNumerator() == 5 and r3.getDenominator() == 3

    try:
        r4 = Rational(2, 0)
        assert False
    except ValueError as er:
        print("something goes wrong...", er)
        assert True

    try:
        r5 = Rational(2, -3)
        assert False
    except ValueError as er:
        print("something goes wrong...", er)
        assert True

    try:
        r6 = Rational(-2, 3)
        assert False
    except ValueError as er:
        print("something goes wrong...", er)
        assert True

    try:
        r7 = Rational(-2, -3)
        assert False
    except ValueError as er:
        print("something goes wrong...", er)
        assert True
```

Getter and setter methods

```
class Flower:
    '''
    a flower is a structure of two elements: name (a string) and price (an integer)
    '''
    def __init__(self, n = "", p = 0):
        self.name = n
        self.price = p

    def getName(self):
        return self.name
    def getPrice(self):
        return self.price

    def setName(self, n = ""):
        self.name = n
    def setPrice(self, p):
        if (p < 0):
            raise ValueError("the price must be positive...")
        self.price = p

    def __str__(self):
        return self.name+ "-" + str(self.price)
```

Access data

```
class Flower:
    ...
    def getName(self):
        return self.name
    def getPrice(self):
        return self.price

    def setName(self, n = ""):
        self.name = n
    def setPrice(self, p):
        if (p < 0):
            raise ValueError("the price must be positive...")
        self.price = p

    def __str__(self):
        return self.name+ "-" + str(self.price)

myFlower = Flower("rose", 5)
myFlower.name
myFlower.getName()
```

- Class definition changes => errors!
- ***Recommended:*** use getters and setters to access data attributes

✓ *Good style*
✓ *Easy to maintain code*
✓ *Prevents bugs*

Default arguments

```
class Flower:
    def __init__(self, n = "", p = 0):
        self.name = n
        self.price = p

    def getName(self):
        return self.name
    def getPrice(self):
        return self.price

    def setName(self, n = ""):
        self.name = n
    def setPrice(self, p):
        self.price = p

    def __str__(self):
        return self.name+ "-" + str(self.price)
```

```
# default arguments for formal parameters are used
# if no actual argument is given
f1 = Flower("rose")
print(f1)

f2 = Flower(p=3)
print(f2)

f3 = Flower()
print(f3)

f3.setName("daisy")
print(f3)

f3.setName()
print(f3)

f3.setPrice()
```

```
rose-0
-3
-0
daisy-0
-0
Traceback (most recent call last):
  File "C:\Users\cami\Desktop\c.py", line 122, in <module>
    f3.setPrice()
TypeError: setPrice() missing 1 required positional argument: 'p'
>>>
```

ADT Recommendations

- Create getter and setter methods to access the data attributes
- Hide the implementation details
  - The class is an abstraction (a black box)
  - The interface of the class stays the same while internal changes can occur
  - Client code should work without any changes even when internal changes occur in the class
- Document each class
  - Short description
  - What objects can be created (based on the data attributes)
  - Restrictions that apply to data
- Create classes using

test-driven development

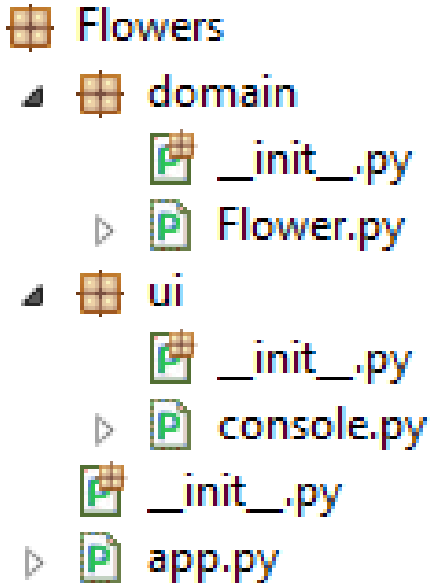
  - Create test functions for
    - Creating an instance of the class
    - Each method of the class

```
class Rational:
    """
    Abstract data type rational numbers
    A rational number is composed by 2 numbers: numerator and denominator
    Domain:{a/b where a,b integer numbers, b!=0, greatest common divisor =1}
    Invariant:b!=0, greatest common divisor a, b =1
    """
    def __init__(self, num = 0, denom = 1):

...
```

ADT Examples in detail

- ADT `Flower`
  - 1 representation
    - Coupling two pieces of information (name, price)

- Example: `05Flowers.zip`

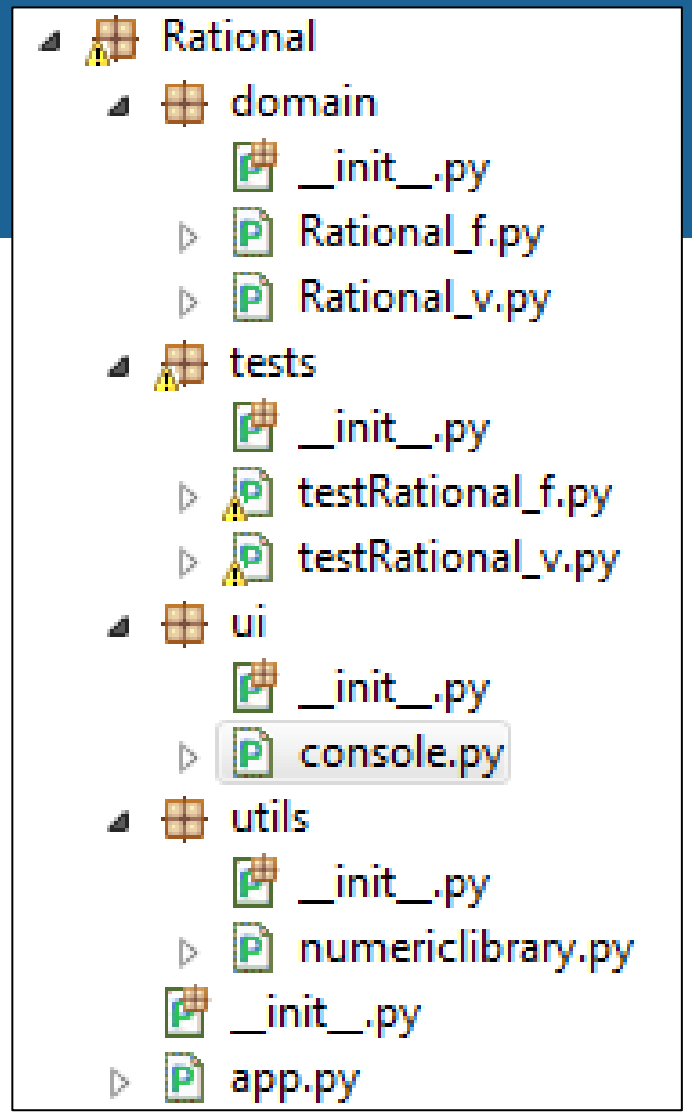


ADT Examples in detail

- ADT `Rational`
  - 2 representations
    - Coupling two pieces of information

    (numerator, denominator)
    - List with 2 elements: numerator and denominator

- Example: `05Rational.zip`



Recap today

- Classes

- Examples
  - Flower
  - Rational

Reading materials and useful links

1. The Python Programming Language - https://www.python.org/
2. The Python Standard Library - https://docs.python.org/3/library/index.html
3. The Python Tutorial - https://docs.python.org/3/tutorial/
4. M. Frentiu, H.F. Pop, Fundamentals of Programming, Cluj University Press, 2006.
5. MIT OpenCourseWare, Introduction to Computer Science and Programming in Python, https://ocw.mit.edu, 2016.
6. K. Beck, Test Driven Development: By Example. Addison-Wesley Longman, 2002. http://en.wikipedia.org/wiki/Test-driven_development
7. M. Fowler, Refactoring. Improving the Design of Existing Code, Addison-Wesley, 1999. http://refactoring.com/catalog/index.html