BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Algorithms and Programming

Lecture 3 – Modular Programming

Mihai Ioan Popescu

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

- A simple feature driven software development process
- Programming paradigms – procedural programming
- Functions
  - Definition
  - Call
  - The process of writing a function - TDD
- Variable scope

Today

- The process of writing a function - TDD

- Modular programming
  - Modules
  - Packages
  - `import` statement

- Exceptions

How to write functions

- **Test driven development (TDD)**
  - Implies creation of tests (that clarify the requirements) before writing the code of the function
- Steps to create a new function:
  1. Add a new test / several tests
  2. Execute tests and verify that at least one of them failed
  3. Write the body of the function
  4. Run all tests
  5. Refactor the code

How to write functions

- Problem: Determine the greatest common divisor of two numbers
- TDD steps to create a new function f:

  1. *Add a new test / several tests*
     - Define a test function test_f() containing the test cases using assertions
     - Concentrate on the specification of f
     - Define the function: name, parameters, pre-conditions, post-conditions, empty body

```python
def test_gcd(): #test function for gcd
    assert gcd(14,21) == 7
    assert gcd(24, 9) == 3
    assert gcd(3, 5) == 1
    assert gcd(0, 3) == 3
    assert gcd(5, 0) == 5
```

```python
'''
Descr: computes the gcd of 2 natural numbers
Data: a, b
Precondition: a, b - natural numbers
Results: res
Postcondition:res=(a,b)
'''
def gcd(a, b):
    pass
```

How to write functions

- Problem: Determine the greatest common divisor of two numbers
- TDD steps to create a new function:
  2. *Execute tests and verify that at least one of them failed*

```
# run all tests by invoking the test function
test_gcd()
```

```
Traceback (most recent call last):
  File "C:\Users\cami\Desktop\c.py", line 21, in <module>
    test_gcd()
  File "C:\Users\cami\Desktop\c.py", line 3, in test_gcd
    assert gcd(14,21) == 7
AssertionError
>>>
```

How to write functions

- Problem: Determine the greatest common divisor of two numbers
- TDD steps to create a new function:
  3. *Write the body of the function*
     - Implement the function according to the pre- and post- conditions so that all tests are successful

```python
'''
Descr: computes the gcd of 2 natural numbers
Data: a, b
Precondition: a, b - natural numbers
Results: res
Postcondition: res=(a,b)
'''
def gcd(a, b):
    if (a == 0):
        if (b == 0):
            return -1 # a == b == 0
        else:
            return b # a == 0, b != 0
    else:
        if (b == 0): # a != 0, b == 0
            return a
        else: # a != 0, b != 0
            while (a != b):
                if (a > b):
                    a = a - b
                else: b = b - a
            return a # a == b
```

How to write functions

- Problem: Determine the greatest common divisor of two numbers
- TDD steps to create a new function:

  4. *Run all tests*
     - The function respects the specifications

```
# run all tests by invoking the test function
test_gcd()
```

How to write functions

- Problem: Determine the greatest common divisor of two numbers
- TDD steps to create a new function:
  5. *Refactoring the code*
     - Restructuring the code of the function, modifying the internal structure without changing the external behavior
     - ***Refactoring nethods:***
       - Extraction method
       - Substitution of an algorithm
       - Replacing a temporary expression with a function

How to write functions

- Problem: Determine the greatest common divisor of two numbers
- TDD steps to create a new function:
  5. *Refactoring the code*
     - ***Refactoring nethods:***
       - Extraction method – part of the code is transferred to a new function

```python
def printHeader():
    print("Table header")

def printTable():
    printHeader()
    print("line 1...")
    print("line 2...")
```

```python
def printHeader():
    print("Table header")

def printLines():
    print("line 1...")
    print("line 2...")

def printTable():
    printHeader()
    printLines()
```

How to write functions

- Problem: Determine the greatest common divisor of two numbers
- TDD steps to create a new function:
  5. *Refactoring the code*
     - Restructuring the code of the function, modifying the internal structure without changing the external behavior
     - ***Refactoring nethods:***
       - Substitution of an algorithm – changing the body of a function (to be more clear, more efficient)

```python
def foundPerson(peopleList):
    for person in peopleList:
        if person == "Emily":
            return "Emily was found"
        if person == "John":
            return "John was found"
        if person == "Mary":
            return "Mary was found"
    return ""

myList = ["Don", "John", "Mary", "Anna"]
print(foundPerson(myList))
```

```python
def foundPerson(peopleList):
    candidates = ["Emily", "John", "Mary"]
    for person in peopleList:
        if candidates.count(person) > 0:
            return candidates[candidates.index(person)] + \
                " was found"
    return ""

myList = ["Don", "John", "Mary", "Anna"]
print(foundPerson(myList))
```

How to write functions

- Problem: Determine the greatest common divisor of two numbers
- TDD steps to create a new function:
  5. *Refactoring the code*
     - Restructuring the code of the function, modifying the internal structure without changing the external behavior
     - ***Refactoring methods:***
       - Replacing a temporary expression with a function
         - A temporary variable stores the result of an expression
         - Include the expression in a new function
         - Use the new function instead of the variable

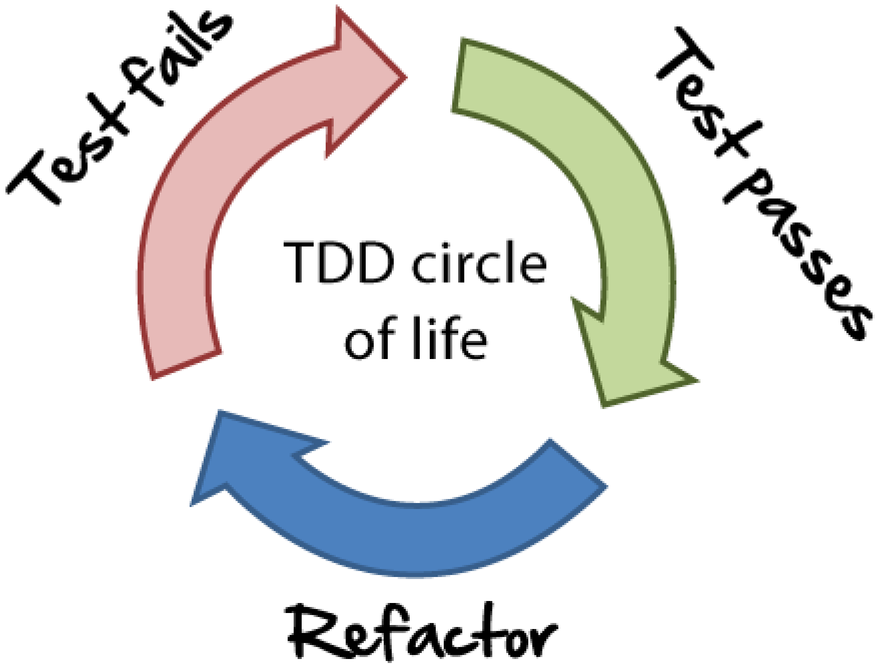
```python
def productValue(quantity, price):
    value = quantity * price
    if (value > 1000):
        return 0.95 * value
    else:
        return value
```

```python
def computeValue(q, p):
    return q * p

def productValue(quantity, price):
    if (computeValue(quantity, price) > 1000):
        return 0.95 * computeValue(quantity, price)
    else:
        return computeValue(quantity, price)
```

TDD

- Think first (what each part of the program has to do), write code after
- Analyse boundary behaviour, how to handle invalid parameters before writing any code



http://www.agilenutshell.com/test_driven_development

Modular programming

- Method to design and implement an algorithm by using modules
- Based on decomposing the problem in subproblems considering:
  - Separating concepts
  - Layered architectures
  - Maintenance and reuse of code
  - Cohesion of elements in a module
  - Link between modules
- Module
  - Structural unit (that can communicate with other units), changeable
  - Collections of functions and variables that implement a well-defined feature

Defining a module in Python

- Module
  - A file that contains Python statements and definitions
    - Variables – global names, visible at the level of the module
    - Function definitions – available in that module and in other modules that import the current module
    - Other statements - initialization
- A module has:
  - Name (**__name__**)
    - The file name is the module name with the suffix ” .py” appended
    - __name__ is set to __main__ if the module is executed on its own
    - __name__ is set to moduleName if the module is imported in another module
  - Docstring (**__doc__**)
    - Triple-quoted module doc string that defines the contents of the module file
    - Summary of the module, description about the module’s contents, purpose and usage.
  - A symbol table that contains all the names (variables and functions) introduced by the module – dir(moduleName)

```python
#...
def gcd(a, b):
    #...
def test_gcd():
    assert gcd(0, 2) == 2
    #...

if __name__ == "__main__":
    test_gcd()
```

Example: fibo.py

```python
'''
Created on 17 Oct 2019

@author: cami

This module provides 2 functions related to the Fibonacci sequence
'''

def fibTerm(n):
    '''
    Return the n-th term of the Fibonacci sequence
    Input: n - the index of the required term (first term, 0 has index 0)
    Output: The n-th term of the sequence
    '''
    a, b, i = 0, 1, 0
    while i < n:
        a, b = b, a + b
        i += 1
    return a
```

Example: fibo.py

```
def fibSequence(n):
    '''
    Return the first n terms of the Fibonacci sequence
    Input: n - the number of terms of the sequence
    Output: The sequence of terms
    '''
    result = []

    for i in range(0, n+1):
        result.append(fibTerm(i))
    return result

'‘’
When the module is executed directly, the variable __name__ is set to __main__
'''
if __name__ == "__main__":
    n = int(input("How many terms?"))
    print(fibSequence(n))
```

Importing a module in Python

- A Python module must be imported in order to use it

- The `import` statement:
  - Searches the global namespace for the module.
    If the module exists, it is already imported and nothing more needs to be done
  - Searches for the module
  - Variables and functions defined in the module are inserted into a new symbol table (a new namespace).
    The module name is added to the current symbol table.

- *Example: testFibo.py*

```python
'''
Created on 17 Oct 2018

@author: cami
'''

import fibo

print(fibo.fibTerm(7))
print(fibo.fibSequence(7))
```

Also try:

```python
help(fibo)
help(fibo.fibTerm)
```

Importing a module in Python

- A Python module can import:
  - Other modules
    - `import [path.]moduleName`
  - Elements of a module
    - `from moduleName import itemName`

- The `import` statement:
  - Introduces a module, looking for the name of the module in:
    - Current folder (where is the file containing the import)
    - List of folders specified by environment variable PYTHONPATH
    - List of folders specified by environment variable PYTHONHOME (an installation-dependent default path)

  - If the module exists:
    - If already imported, do nothing
    - Otherwise, execute the statements in the module

  - Otherwise, throw an exception: `ImportError`

Example: importing a module

Module: numlib.py

```python
'''
Descr: computes the gcd of 2 natural numbers
Data: a, b
Precondition: a, b - natural numbers, b > 0
Results: res
Postcondition:res=(a,b)
'''
def gcd(a, b):
    if (a == 0):
        if (b == 0):
            return -1    # a == b == 0
        else:
            return b     # a == 0, b != 0
    else:
        if (b == 0):     # a != 0, b == 0
            return a
        else:            # a != 0, b != 0
            while (a != b):
                if (a > b):
                    a = a - b
                else:
                    b = b - a
            return a     # a == b
```

File name: numlib.py
Module name: numlib

Module: test.py

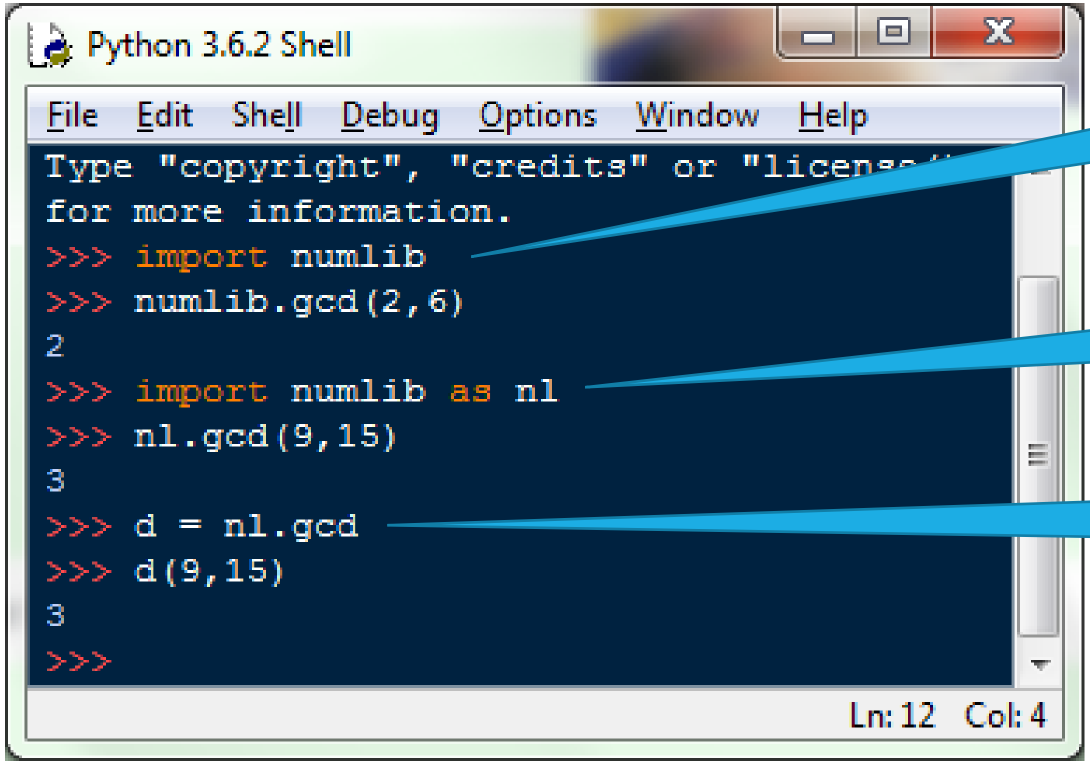
```python
import numlib

def run_gcd():
    a = int(input("Input the first number: "))
    b = int(input("Input the second number: "))
    print("Greatest Common Divisor of ", a, "and ", /
          b, " is ", numlib.gcd(a,b))

run_gcd()
```

Example: importing a module

- Import the module in the interactive Python shell and use it



Import the module and use it using the dot notation

Import the module under new name

Assign a local name to a function

Example: importing names from a module

Module: numlib.py

```python
'''
Descr: computes the gcd of 2 natural numbers
Data: a, b
Precondition: a, b - natural numbers, b > 0
Results: res
Postcondition:res=(a,b)
'''
def gcd(a, b):
...
```

Module: test.py

```python
from numlib import gcd

def run_gcd():
    a = int(input("Input the first number: "))
    b = int(input("Input the second number: "))
    print("Greatest Common Divisor of ", a, "and ", b, " is ", gcd(a,b))

run_gcd()
```

Possible, but not recommended:

```python
from numlib import *
```

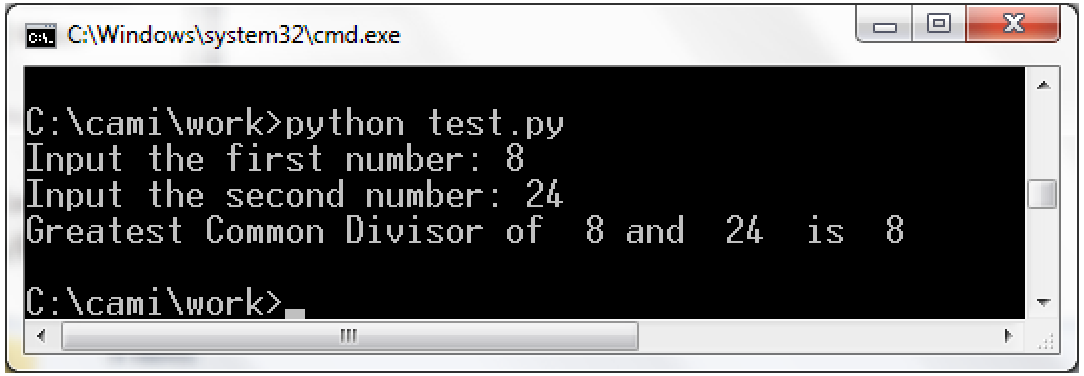
Executing modules as scripts

- `python test.py`
  - Execute a Python module
    - The module is executed (similar to being imported), *and also*
    - The system variable `__name__` is set to `__main__`

```python
from numlib import gcd

def run_gcd():
    a = int(input("Input the first number: "))
    b = int(input("Input the second number: "))
    print("Greatest Common Divisor of ", a, "and ", b,\
 " is ", gcd(a,b))

if __name__ == "__main__":
    run_gcd()
```



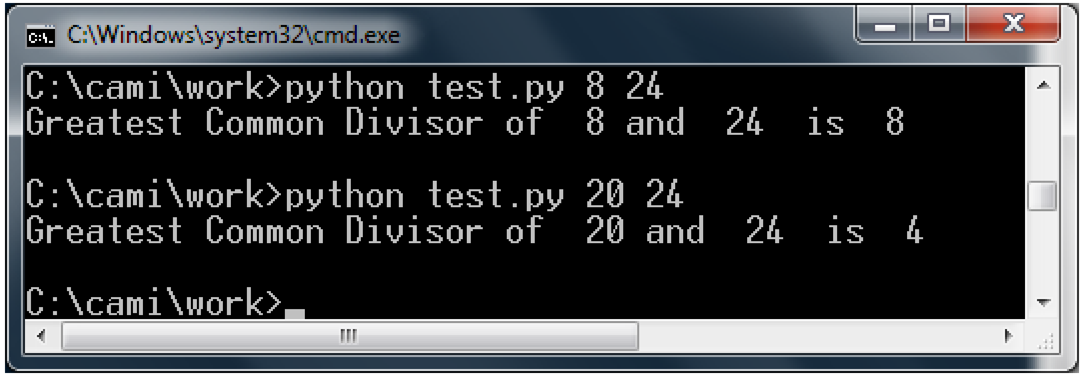
Executing modules as scripts

- `python test.py`
  - Execute a Python module
    - The module is executed (similar to being imported), *and also*
    - The system variable `__name__` is set to `__main__`

```
from numlib import gcd

def run_gcd(a, b):
    print("Greatest Common Divisor of ", a, \
"and ", b, " is ", gcd(a,b))

if __name__ == "__main__":
    import sys
    run_gcd(int(sys.argv[1]), int(sys.argv[2]))
```



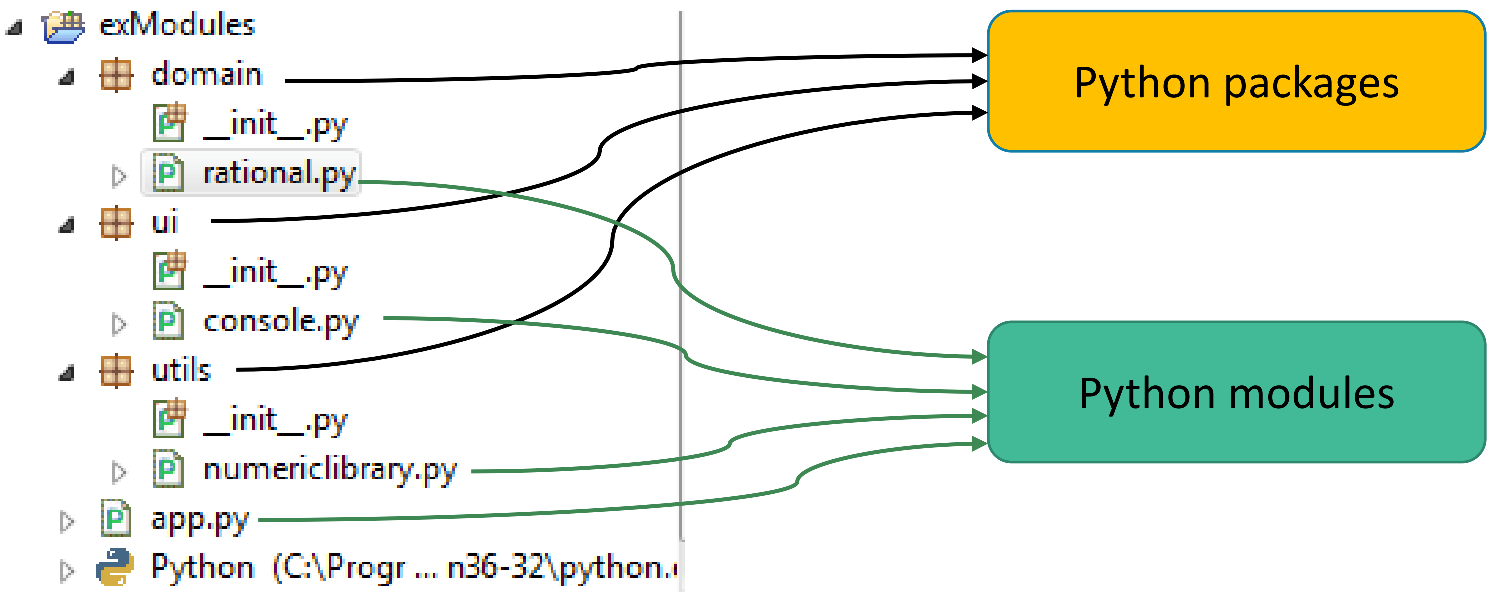
Modular programming

- Packages in Python
  - Using packages
    - A way to structure the code
  - If there are several modules (files)
    - Structure them in hierarchical folders
  - Python package = a folder that contains:
    - Python modules
    - The module `__init__.py` - used for initialization statements
  - Importing the modules from a package:
    - `import packageName.moduleName`
    - `from packageName.moduleName import itemName`

Modular programming

- Organizing an application using modules and packages
  - User interface
    - Functions dealing with user interaction
    - Read / write operations – only here should be
  - Domain
    - Functions dealing with the features of the application
  - Infrastructure
    - Useful functions that have a high potential to be reused
  - Coordinator
    - Functions to initialize and start the application
- Example - RationalNumbers contains the following packages and modules:
  - `app.py` - module for starting the application
  - `domain`
    - `rational.py` – module for computations on rational numbers
  - `utils`
    - `numericlibrary.py` – module for useful math computations
  - `ui`
    - `console.py` – module for the user interface

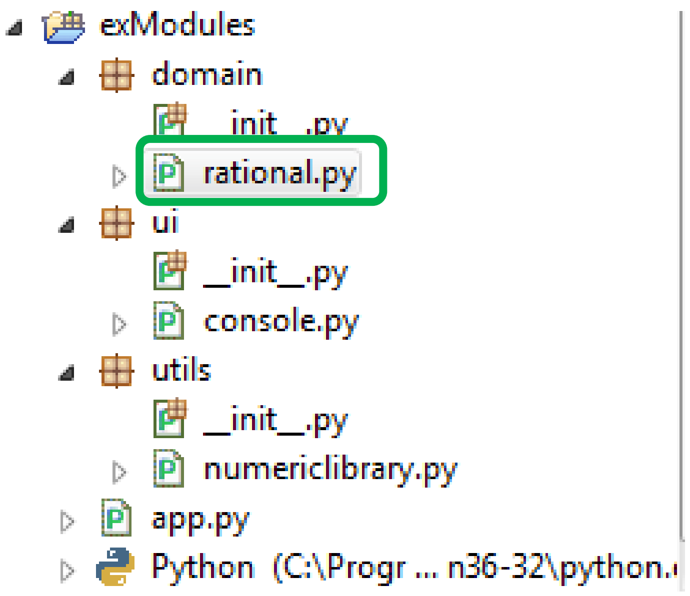
Example: RationalNumbers



app.py

```python
from utils.numericlibrary import gcd
print(gcd(2 , 6))
```

Example: RationalNumbers



rational.py

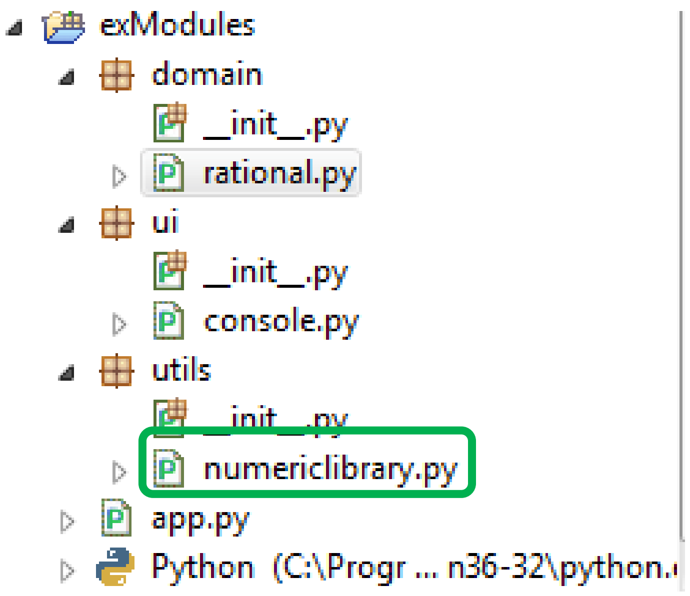
```
import utils.numericlibrary

def test_rsum():
    assert rsum([2, 3], [4, 5]) == [22, 15]
    assert rsum([1, 4], [1, 4]) == [1, 2]
    assert rsum([1, 2], [1, 2]) == [1, 1]

'''
Descr: computes the sum of two rational numbers
Data: r1, r2
Precondition: r1, r2 - rational numbers
Results: rs
Postcondition:rs - rational number, rs = r1 + r2
'''
def rsum(r1, r2):
    numerator = r1[0] * r2[1] + r1[1] * r2[0]
    denominator = r1[1] * r2[1]
    divisor = utils.numericlibrary.gcd(numerator, denominator)
    rs = [numerator / divisor, denominator / divisor]
    return rs

test_rsum()
```

Example: RationalNumbers



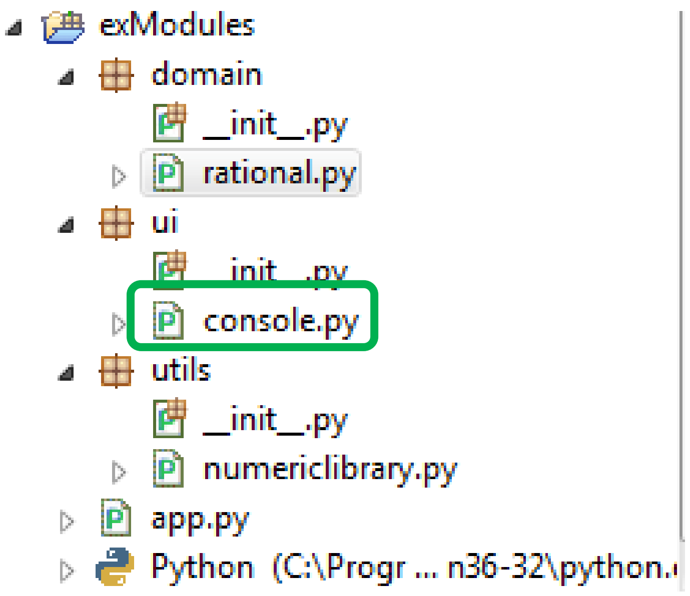
numericlibrary.py

```
def test_gcd():
    #test function for gcd
    assert gcd(14,21) == 7
    assert gcd(24, 9) == 3
    assert gcd(3, 5) == 1
    assert gcd(0, 3) == 3
    assert gcd(5, 0) == 5

'''
Descr: computes the gcd of 2 natural numbers
Data: a, b
Precondition: a, b - natural numbers, b > 0
Results: res
Postcondition:res=(a,b)
'''
def gcd(a, b):
    if (a == 0):
        #...
    else:
        #...

test_gcd()
```

Example: RationalNumbers



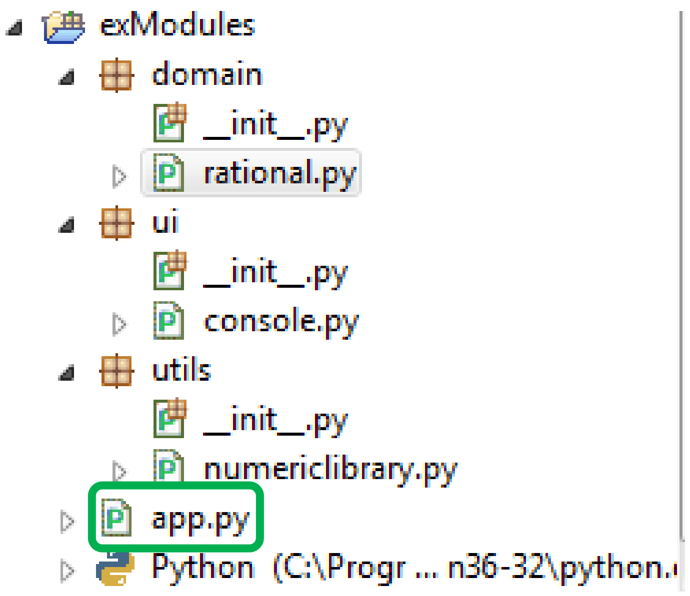
console.py

```python
import utils.numericlibrary
import domain.rational

def readRational():
    num = int(input("numerator = "))
    denom = int(input("denominator = "))
    while (denom == 0):
        print("denominator must be different to 0...give a new value")
        denom = int(input("denominator = "))
    num = num / utils.numericlibrary.gcd(denom, num)
    denom = denom / utils.numericlibrary.gcd(denom, num)
    return [num, denom]

def run():
    finish = False
    r_sum = [0, 1]
    while (not finish):
        r = readRational()
        if (r[0] == 0):
            finish = True
        else:
            r_sum = domain.rational.rsum(r_sum, r)
    print(r_sum)
```

Example: RationalNumbers



app.py

```python
import ui.console

ui.console.run()
```

When importing a module in Python

- Variables and functions defined by the moldule are inserted in a new symbol table
- The name of the module (__name__) is inserted in the current symbol table

```
#only import the name ui.console into the current symbol table
import ui.console

#invoke run by providing the doted notation ui.console of the package
ui.console.run()

#import the function name gcd into the local symbol table
from utils.numericlibrary import gcd

#invoke the gcd function from module utils.numericlibrary
print(gcd(2,6))

#import all the names (functions, variables) into the local symbol table
from domain.rational import *

#invoke the rsum function from module rational
print(rsum([2,6],[1,6]))
```

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