BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Fundamentals of Programming

Lecture 3 – Modular Programming

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

- A simple feature driven software development process
- Programming paradigms – procedural programming
- Functions
  - Definition
  - Call
  - How to write a function - TDD
- Variable scope

Today

- Modular programming

- Modules
- Packages
- `import` statement

- Exceptions

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
    - __main__ if the module is executed on its own
    - 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 22 Oct 2017

@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

'''
Module is executed directly when __name__ is __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 22 Oct 2017

@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

Python 3.6.2 Shell

File Edit Shell Debug Options Window Help

```
Type "copyright", "credits" or "licens
for more information.
>>> import numlib
>>> numlib.gcd(2,6)
2
>>> import numlib as nl
>>> nl.gcd(9,15)
3
>>> d = nl.gcd
>>> d(9,15)
3
>>>
```

Ln: 12 Col: 4

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 *
```

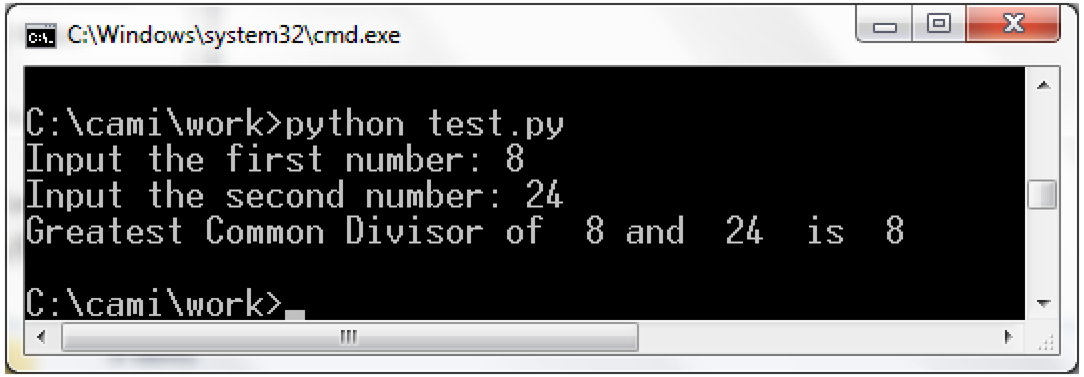
Executing modules as scripts

- `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()
```



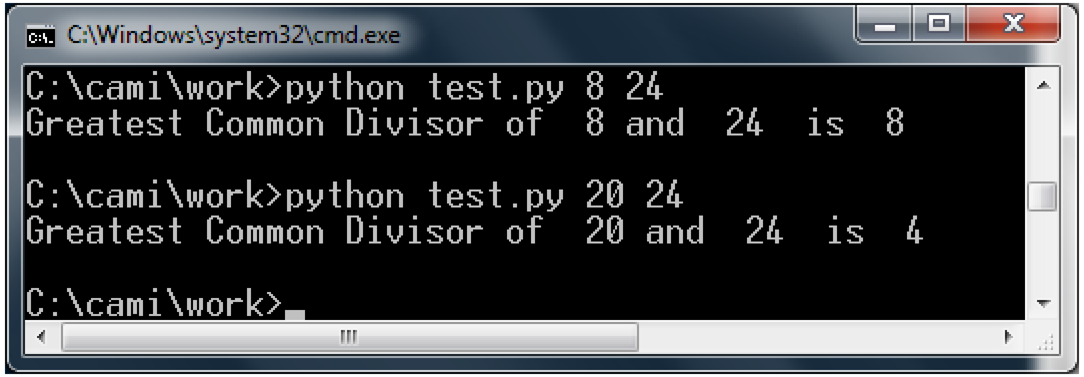
Executing modules as scripts

- `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__`

```python
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]))
```



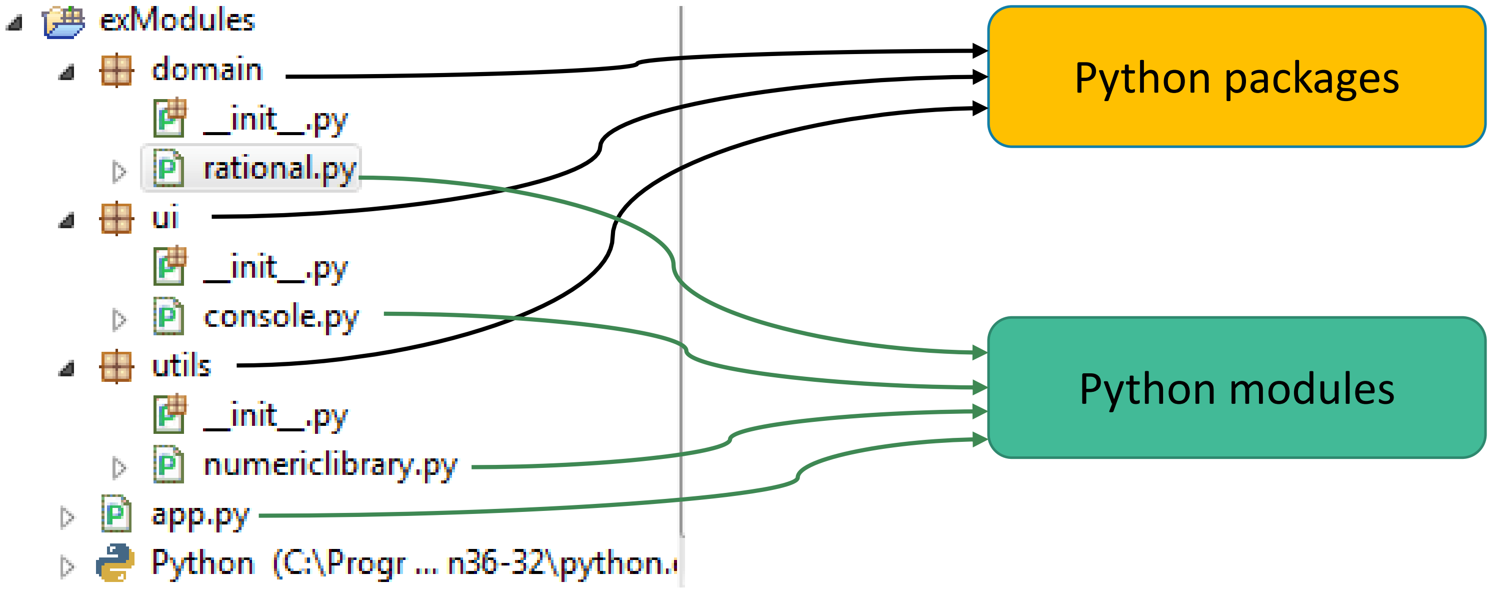
Modular programming

- 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

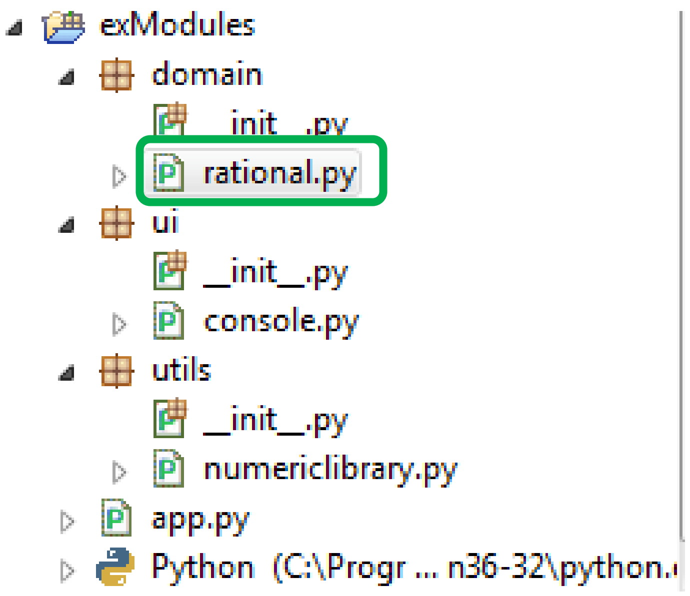
Example: RationalNumbers



app.py

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

Example: RationalNumbers



rational.py

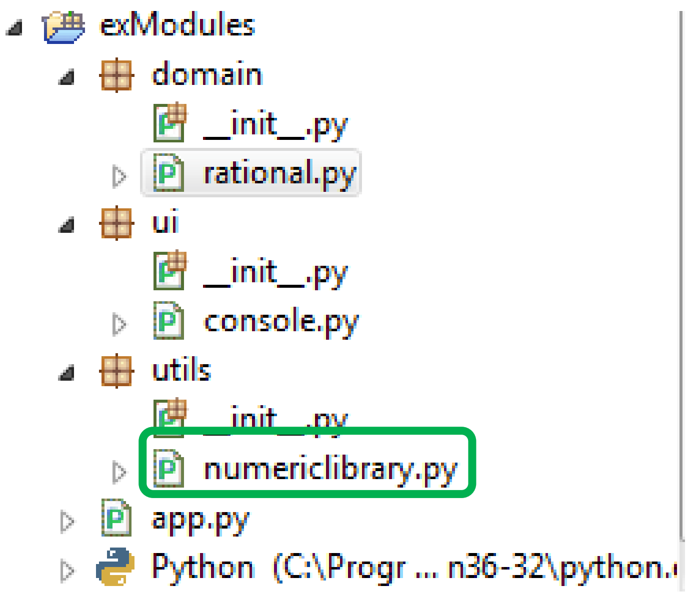
```
import utils.numericlibrary

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



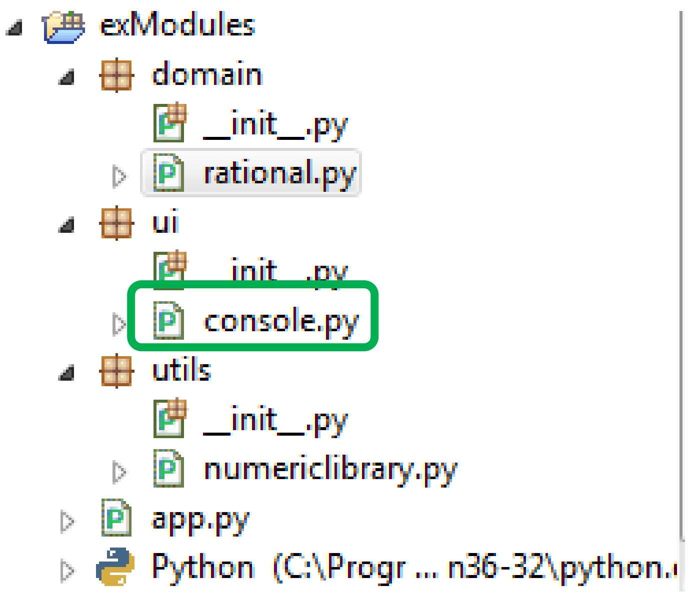
numericlibrary.py

```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

'''
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



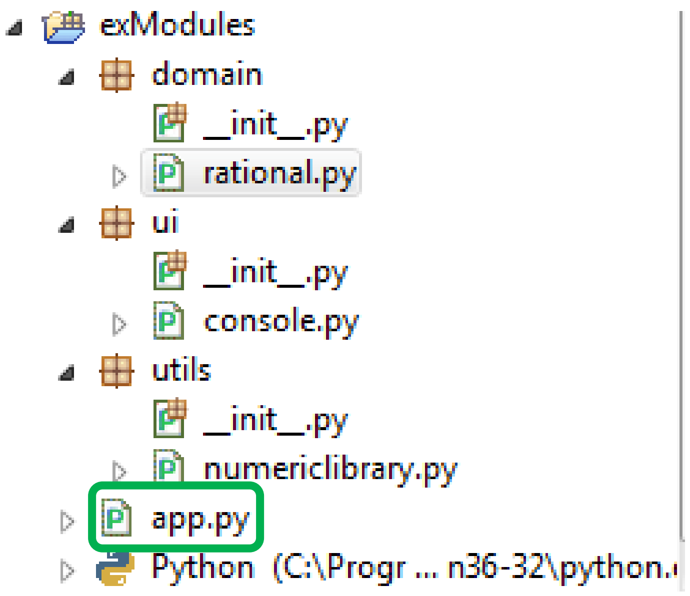
console.py

```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]))
```

Eclipse IDE

- Eclipse is an advanced IDE
  - Free, configurable and easy to use
- Provides lots of plugins to allow development in many languages, including Java, C/C++, Python...
- **Eclipse + PyDev:** setting up for Python development
  - By default, Eclipse can be used to develop Java software
  - To develop in Python: get the **PyDev plugin** (www.pydev.org)
  - PyDev links Eclipse to the installed Python interpreter and libraries, provides wizards for project creation, syntax highlighting and code completion
- Working with projects, navigating and editing source files and program resources, testing and debugging

Testing and debugging

- Separate the code in modules – test and debug them separately
- Document modules and functions
- Debugging the code
  - Identify why a program is not working as expected
  - Study the events that generate an error
  - Use print!
- Testing the code
  - No syntax errors
  - No semantic errors
  - Use assertions
  - Unit testing: validate each unit, test each function separately

Error messages

- SyntaxError

```
a = input("First number is“
b = input("First number is“)
```

- NameError

```
>>> a
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    a
NameError: name 'a' is not defined
>>> a=5
>>> a
5
```

- TypeError

```
a = input('First number is')
int(a)
b = a % 2
print (b)
```

- IndexError

```
>>> my_list = [1, 2, "a", 3]
>>> my_list[1]
2
>>> my_list[5]
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    my_list[5]
IndexError: list index out of range
```

- AttributeError
- IOError

Easy to fix

Exceptions

- Concept
  - Exceptions are raised when errors are detected during program execution
  - Exceptions can interrupt the normal execution of a block
- Mechanism
  - Exceptions are identified and thrown by the Python interpretor
  - Use the code to indicate the special situations

```python
d = int(input("Enter a number: ")) # d = 0
print(5 / d)
x = d * 10
```

```
Traceback (most recent call last):
  File "C:\cami\work\work-ubb\teaching
est.py", line 33, in <module>
    print(5 / d)
ZeroDivisionError: division by zero
```

Exceptions: mechanism

- Identify exceptions
  - raise statement
  - Python interpretor

```
def functionThatRaiseAnException(a, b):
    if (cond):
        raise ValueError("message")
    # code
```

- Catch and treat an exception
  - try..except(..finally) statement

```
try:
    # main code
except ExceptionType1 as e1:
    #if e1 appears, this code is executed
except ExceptionType2 as e2:
    #if e2 appears, this code is executed
else:
    #if there is no exception, this code is executed
```

```python
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

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

run_gcd()
```

```python
def gcd_v2(a, b):
    if (a == 0):
        if (b == 0):
            raise ValueError("one number must be != 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

def ru
    a
    b
    pr                                                  )

run_gc
```

```
Input the first number: 0
Input the second number: 0
Traceback (most recent call last):
  File "C:\cami\work\work-ubb\teaching\Fundament
est.py", line 31, in <module>
    run_gcd()
  File "C:\cami\work\work-ubb\teaching\Fundament
est.py", line 28, in run_gcd
    print("Greatest Common Divisor of ", a, "and
  File "C:\cami\work\work-ubb\teaching\Fundament
est.py", line 11, in gcd
    raise ValueError("one number must be != 0")
ValueError: one number must be != 0
```

Exceptions

- Mechanism
  - Catch exceptions – Python code can include handlers for exceptions
  - Statement `try...except`
  - The clause `finally` - statements that will always be executed (clean-up code)

```
try:
    #code that may raise exceptions
except ValueError:
    #code to handle the error
finally:
    #code that is executed in all the cases
```

```
try:
    d = 0
    print (5 / d)
    x = d * 10
except ZeroDivisionError:
    print("division by zero error...")
finally:
    print("all cases...")
```

```
def run_gcd():
    a = int(input("Input the first number: "))
    b = int(input("Input the second number: "))
    try:
        div = gcd_v2(a,b)
        print("gcd of ", a, " and ", b, " is ", div)
    except ValueError as ex:
        print("exceptional case: ", ex)
    finally:
        print("do you
```

```
Input the first number: 0
Input the second number: 0
exceptional case:  one number must be != 0
do you want to try again?
```

Exceptions: more examples

```
try:
    a = int(input("Enter the first number: "))
    b = int(input("Enter the second number: "))
    print("a+b = ", a+b)
    print("a/b = ", a/b)
    print("a**b = ", a**b)
except ValueError:
    print("The value you entered is not a number!")
except ZeroDivisionError:
    print("The second number can not be zero: division by zero!")
except:
    print("An exception occurred...")
```

Exceptions and testing

```python
def gcd_v2(a, b):
    if (a == 0):
        if (b == 0):
            raise ValueError("one number must be != 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
```

```python
def run_gcd():
    a = int(input("Input the first number: "))
    b = int(input("Input the second number: "))
    try:
        div = gcd_v2(a,b)
        print("gcd of ", a, " and ", b, " is ", div)
    except ValueError as ex:
        print("exceptional case: ", ex)
    finally:
        print("do you want to try again?")
```

```python
def test_gcd_v2():
    assert gcd_v2(0, 2) == 2
    assert gcd_v2(2, 0) == 2
    assert gcd_v2(3, 2) == 1
    assert gcd_v2(6, 2) == 2
    assert gcd_v2(4, 6) == 2
    assert gcd_v2(24, 9) == 3

    try:
        gcd_v2(0, 0)
        assert False
    except ValueError:
        assert True
```

Exceptions

- When should exceptions be used?
  - Identify special situations
    - Ex1: A function does not receive parameters according to its specification
    - Ex2: Operating on data from files that do not exist or can not be accessed
    - Ex3: Impossible operations (division by 0)
  - Force compliance with specifications (pre-conditions)

Recap today

- Modular programming
  - Concepts and principles
  - Python elements for modular programming
    - Modules
    - Packages
    - Import statements

- Exceptions
  - Concept
  - Mechanism
  - Examples

Next time

- User defined data types
- Classes

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

Bibliography

The content of this course has been prepared using the reading materials from previous slide, different sources from the Internet as well as lectures on Fundamentals of Programming held in previous years by:

- Prof. Dr. Laura Dioşan - www.cs.ubbcluj.ro/~lauras
- Conf. Dr. Istvan Czibula - www.cs.ubbcluj.ro/~istvanc
- Lect. Dr. Andreea Vescan - www.cs.ubbcluj.ro/~avescan
- Lect. Dr. Arthur Molnar - www.cs.ubbcluj.ro/~arthur