BABEŞ-BOLYAI UNIVERSITY

Faculty of Mathematics and Computer Science

Fundamentals of Programming

Lecture 2 – Procedural Programming

Camelia Chira

Last time

- Programming process
    - What is programming?
    - Basic elements of Python
        - Python programs
        - Data types
        - Variables and expressions
        - Statements: assignments, conditionals, loops

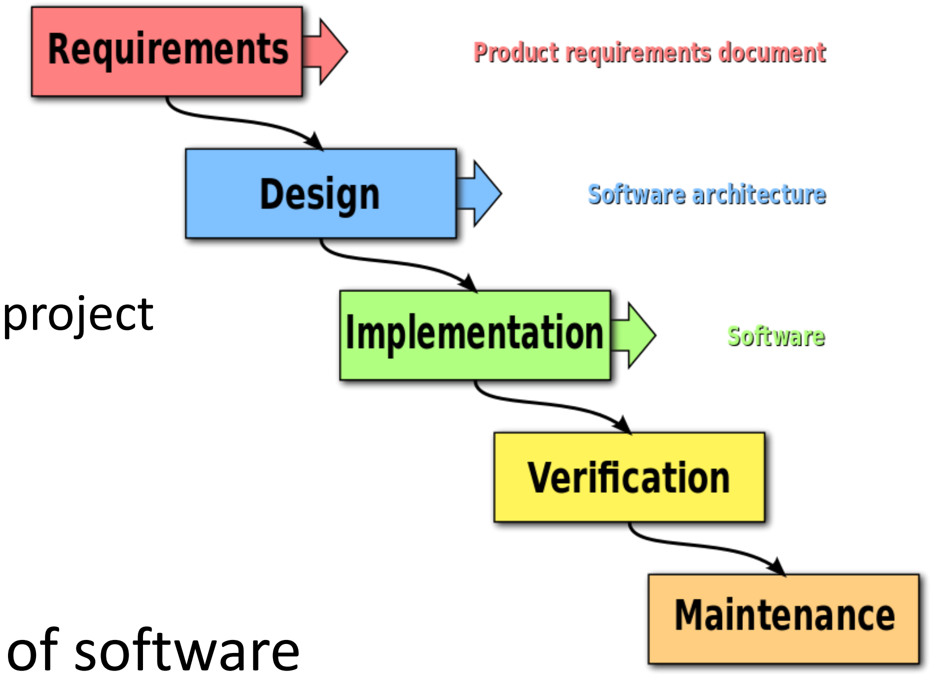
Today

- Software development process
  - Simple feature-driven development
- Programming paradigms
- Functions
  - Definition
  - Call
  - How to write a function

Software development process

- Roles in software engineering
  - Programmer / software developer
    - Writes and develops programs for users
  - Client
    - The one interested / affected by the results of a project
  - User
    - Runs the program on the computer
- Software development process:
  - Includes creation, launch and maintainence of software
  - Indicate the steps to be followed and their order

The waterfall model



Software development process

- Steps in solving a problem:
  - Problem definition
  - Requirements
  - Use case scenario
  - Identify the features and separate them on iterations
  - Identify the activities or tasks (for each feature) and describe them

Software development process

- Steps in solving a problem:
  - Problem definition
    - A short description of the problem
    - *A teacher (**client**) needs an application for students (**users**) learning how to find the smallest prime number greater than a given natural number n.*
  - Requirements
    - Define what is required from the client perspective (what the application needs to do)
    - Identify input and output data
    - ***Input**: n – a natural number*
    - ***Output**: the smallest prime number greater than n*
  - Use case scenario

Run	1	2	3	4
Input	5	0	11	-3
Output	7	2	13	Please enter a natural number:

Software development process

- Steps in solving a problem:
  - Identify the features and plan the iterations
    - **Feature**
      - Defined as a client function
      - Specified as (action, result, object)
        - Action – a function that the application needs to provide
        - Result – obtained as a result of executing the function
        - Object – an entity where the application implements the function
      - *F1: finding the smallest prime number greater than given n*
    - **Iteration**
      - Time period when a stable and runnable version of a product is created (with documentation)
      - Helps to plan the delivery of features
      - ***I1=F1***

Software development process

- Steps in solving a problem:
  - List of activities or tasks (for each feature) and their description
    - Recommendations:
      - Define one activity for each operation
      - Define an activity for user interface (UI) interaction
      - Define an activity for UI operations
      - Determine the dependencies between activities
    - *A1: verify if a given number is prime or not*
    - *A2: find the smallest prime number greater than a given natural number*
    - *A3: implement the initialization of a number, finding the smallest prime number greater then n and return the result*
    - *A4: implement the UI*

Software development process

- Steps in solving a problem:
  - List of activities (for each feature) and their **description**
    - Testing cases
      - Specify a set of input data and expected results to evaluate a part of a program
      - *A1: verify if a given number is prime or not*

Input	Output
2	True
6	False
3	True
-2	False
1	False

Software development process

- Steps in solving a problem:
  - List of activities (for each feature) and their **description**
    - Implementation
      - *A1: verify if a given number is prime or not*

```
# Description: verifies if the number n is prime
# Data: n
# Precondition: n - natural number
# Results: res
# Postcondition: res=FALSE, if n is not prime or res=TRUE, if n is prime
if (n < 2):
    print("no ", n, " is not prime (is composed)")
else:
    d = 2
    isPrime = True
    while (d * d <= n) and (isPrime == True):
        if (n % d == 0):
            isPrime = False
        else:
            d = d + 1
    if (isPrime == True):
        print("no ", n, " is prime")
    else:
        print("no ", n, " is not prime")
```

Simple feature-driven development

- Build a feature list from problem statement
- Plan iterations
- For each iteration
  - Model planned features
  - Implement and test the features
  - Obs:
    - At the beginning of each iteration: analyze each feature – determine the activities (tasks) required – schedule the tasks – implementat and test each independently.
    - *An iteration will result in a working program for the client (will interact with the user, perform some computation, show results)*

Programming paradigms

- Fundamental style of computer programming
- Imperative programming
  - Computations described through statements that modify the state of a program (control flow – sequence of statements exexuted by the computer)
  - Examples
    - **Procedural programming** – each program is formed by several procedures (subroutines or functions)
    - Object oriented programming
- Declarative programming
  - Expresses the logic of a computation (without specifying the control flow)
  - Examples
    - Functional programming (LISP)
    - Logic programming (Prolog, SQL)

Procedural programming – functions

- **Procedural programming** – each program is formed by several procedures (subroutines or functions)
- Function
  - A block of statements that can be reused
  - Are run in a program only when they are called
  - Function characteristics:
    - Has a **name**
    - Has a list of **parameters**
    - Can **return** a value
    - Has a body (a block of statements)
    - Has a specification (**docstring**) formed of:
      - A description
      - Type and description of parameters
      - Conditions imposed on input parameters (pre-conditions)
      - Type and description of return value
      - Conditions imposed on output values (post-conditions)
      - Exceptions that can occur during its execution

Functions in Python

- A function is defined using reserved keyword def
- Execution of function is produced only upon calling / invoking it

```python
def get_max(a, b):
    """
    Compute the maximum of 2 numbers a, b - numbers
    Return a number - the maximum of two integers.
    Raise TypeError if parameters are not integers.
    """
    if a>b:
        return a
    return b

get_max(2, 3)
```

Functions in Python

- Calling a function
  - Recap. **block** = part of a Python program (identified by indentation) executed as a unit
  - The body of a function is a block
  - A block is executed in a new execution frame which:
    - Contains administrative information (useful in the debugging phase)
    - Determines where and how the execution of the program will continue (after the execution of the current block is completed)
    - Defines 2 name spaces (local and global) that affect the execution of the block

Functions in Python

- Calling a function
  - New **scope/frame/environment** created when enter a function
  - Name space
    - A container of names
    - Link between name and object
    - Features similar to a dictionary
  - Binding
    - Adding a name to the name space
  - Rebinding
    - Changing the link between a name and an object
  - Unbinding
    - Removing a name from the name space

Functions in Python

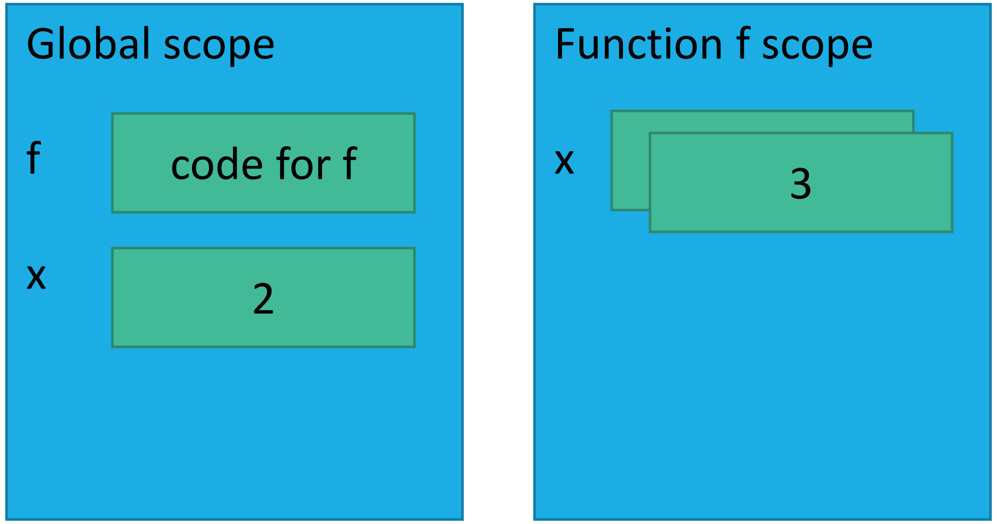
- Calling a function
  - Formal parameters
    - Identify input parameters
    - Each call to the function should respect the number and type of requested parameters
  - Actual parameters
    - Values given to the formal parameters when function is called
    - Stored in local symbol tables of the called function
    - Via reference

```python
def search(element, lis):
    for x in lis:
        if (x == element):
            return True
    return False

a = [2, 3, 4, 5, 6]
el = 3
if (search(el, a) == True):
    print("el was found...")
else:
    print("el was not found...")
```

Variable scope

- Scope – defines if a variable is visible inside a block
    - Scope of a variable defined in a block is that block
    - Variables defined on a certain indentation level are considered local to that block

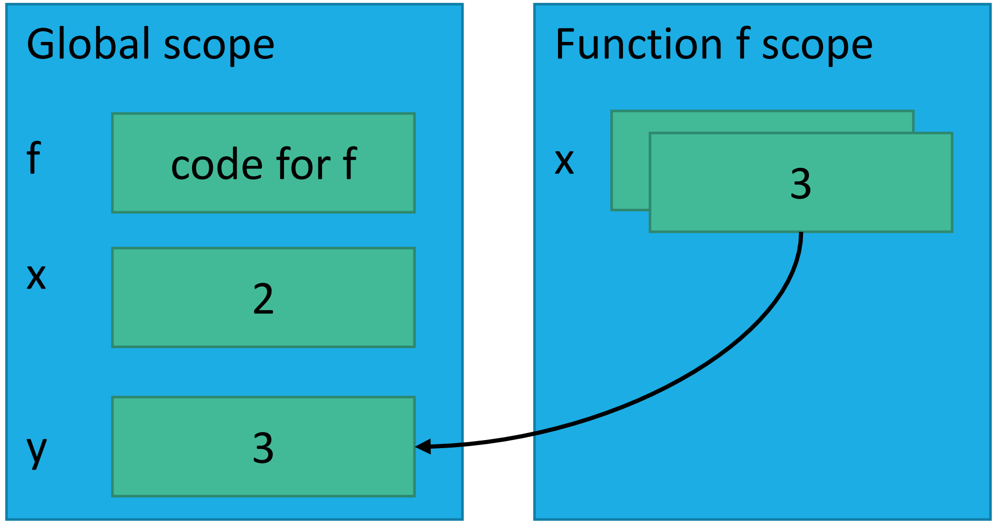
```python
def f(x):
    x = x + 1
    print("Inside f, x = ", x)

x = 2
f(x)
```



Variable scope

- Scope – defines if a variable is visible inside a block
  - Scope of a variable defined in a block is that block
  - Variables defined on a certain indentation level are considered local to that block

```python
def f(x):
    x = x + 1
    print("Inside f, x = ", x)
    return x

x = 2
y = f(x)
```



Variable scope

```python
def f():
    a = 1
    a += 1
    print("Inside f, a = ", a)

a = 5
print("a = ", a)
f()
print("a = ", a)
```

a is redefined local variable

```
a =  5
Inside f, a =  2
a =  5
>>>
```

```python
def f():
    print("Inside f, a = ", a)
    print("Inside f, a^2 = ", a ** 2)

a = 5
f()
print("a = ", a)
```

```
a =  5
Inside f, a =  5
Inside f, a^2 =  25
a =  5
>>>
```

```python
def f():
    a += 1
    print("Inside f, a = ", a)

a = 5
print("a = ", a)
f()
print("a = ", a)
```

Unbound Local Error

```
a =  5
Traceback (most recent call last):
  File "C:\Users\cami\Desktop\c.py", line 7, in <module>
    f(a)
  File "C:\Users\cami\Desktop\c.py", line 2, in f
    a += 1
UnboundLocalError: local variable 'a' referenced before assignment
```

Variable scope

- Types of variables
  - **Local** – a name (of variable) defined in a block
  - **Global** – a name defined in a module
  - Free – a name used in a block but defined somewhere else

```python
g1 = 1 # g1 - global variable (also local, a module being a block)

def fun1(a): # a is a formal parameter
    b = a + g1 # b - local variable, g1 - free variable
    if b > 0: # a, b, and g1 are visible in all blocks of this function
        c = b - g1 # b is visible here, also g1
        b = c # c is a local variable defined in this block
    return b # c is not visible here

def fun2():
    global g1
    d = g1 # g1 - global variable
    g1 = 2 # g1 must be declared global, before
    return d + g1 # any references to g1 in this function

print(fun1(1))
print(fun2())
```

Variable scope

- Where is a variable visible?
  - Rules to determine the scope of a name (variable or function)
    - A name is visible only inside the block where it is defined
    - The formal parameters of a function belong to the body of the function (are visible only inside the function)
    - Names defined outside of a function (at module level) belong to the module scope
    - When a name is used in a block, its visibility is determined using the nearest scope (that contains that name)

```python
a = 100
def f():
    a = 300
    print(a) # 300

f()
print(a) # 100
```

```python
a = 100
def f():
    global a
    a = 300
    print(a) # 300

f()
print(a) # 300
```

Variable scope

- Inspecting the local / global variables of a program
  - `locals()`
  - `globals()`

```python
a = 300
def f():
    a = 500
    print(a)
    print(locals())
    print(globals())

f()
print(a)
```

```
{'a': 500}
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__': {}, '__builtins__':
<module 'builtins' (built-in)>, '__file__':
'C:\\Users\\cami\\Desktop\\c.py', 'a': 300, 'f': <function f at
0x0065AC90>}
```

Variable scope

```python
def change_or_not_immutable(a):
    print ('Locals ', locals())
    print ('Before assignment: a = ', a, ' id = ', id(a))
    a = 0
    print ('After assignment: a =', a, ' id = ', id(a))

g1 = 1 #global immutable int
print ('Globals ', globals())
print ('Before call: g1 = ', g1, ' id = ', id(g1))
change_or_not_immutable(g1)
print ('After call: g1 = ', g1, ' id = ', id(g1))
```

```
Globals  {'__name__': '__main__', '__doc__': None, '__package__':
r__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': Non
ons__': {}, '__builtins__': <module 'builtins' (built-in)>, 'chang
able': <function change_or_not_immutable at 0x0055AC90>, 'g1': 1}
Before call: g1 =  1  id =  505571456
Locals  {'a': 1}
Before assignment: a =  1  id =  505571456
After assignment: a =  0  id =  505571440
After call: g1 =  1  id =  505571456
>>>
```

```python
def change_or_not_mutable(a):
    print ('Locals ', locals())
    print ('Before assignment: a = ', a,' id = ', id(a))
    a[1] = 5
    print ('After first assignment: a = ', a,' id = ', id(a))
    a = [0]
    print ('After second assignment: a = ', a,' id = ', id(a))

g2 = [0, 1] #global mutable list
print ('Globals ', globals())
print ('Before call: g2 = ', g2, ' id = ', id(g2))
change_or_not_mutable(g2)
print ('After call: g2 = ', g2, ' id = ', id(g2))
```

```
Globals  {'__name__': '__main__', '__doc__': None, '_
r__': <class '_frozen_importlib.BuiltinImporter'>, '_
ons__': {}, '__builtins__': <module 'builtins' (built
rs\\cami\\Desktop\\c.py', 'change_or_not_mutable': <f
le at 0x027FAC90>, 'g2': [0, 1]}
Before call: g2 =  [0, 1]  id =  48222336
Locals  {'a': [0, 1]}
Before assignment: a =  [0, 1]  id =  48222336
After first assignment: a =  [0, 5]  id =  48222336
After second assignment: a =  [0]  id =  41938416
After call: g2 =  [0, 5]  id =  48222336
>>>
```

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

```
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
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

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

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

```
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

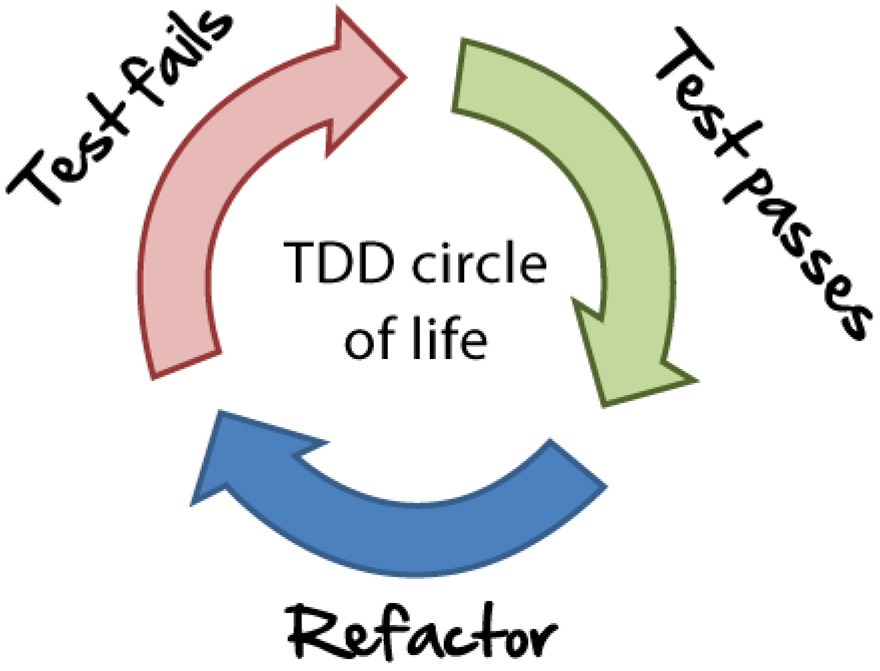
```python
def productValue(quantity, price):
    value = quantity * price
    if (value > 1000):
        return 0.95 * value
    else:
        return value
```

```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

Recap today

- Simple feature-driven development
- Procedural programming
- Functions
  - Definition
  - Call
  - How to write a function
- Variable scope

Next time

- Modular programming

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