BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Algorithms and Programming

Lecture 4 – Software design principles

Camelia Chira

Course content

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

Last time

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

- Exceptions
  - Concept
  - Mechanism
  - Examples

Today

- Exceptions
- Architecture of an application
- Principles of organizing the code
- Working with files

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

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

```
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
```

```
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()
```

```
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

```
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 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?")
```

```
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)

Organize the code of an application

- Layered architectures
  - Decomposing by features – 2 perspectives:
    - Functional perspective – identifying different features specified by the problem
    - Technical perspective – introducing technical features (such as user interaction, file management, databases, networks , etc)
  - Recommended solution:
    - Decompose a complex application on layers
    - Concentrate the code related to the domain of the problem in a single layer and isolate it
    - Ensure cohesive layers

Organize the code of an application

- How to organize the code
  - Create modules for
    - User interface
      - Functions dealing with user interaction
      - Contains read operations and display methods
      - The only module used to read and output data
    - Domain of the application
      - Functions dealing with the problem domain
    - Infrastructure
      - Useful functions that are highly to be reused (e.g. logging, network I/O)
    - Application coordinator
      - Initializes and starts the application

Organize the code of an application

- Good software design:
  - ✓ Code easy to understand
  - ✓ Easy to test
  - ✓ Easy to maintain
  - ✓ Easy to develop and modify (e.g. add features)
- Key design principles
  - Single Responsibility Principle
  - Separation of Concerns Principle
  - Reuse Principle
  - Cohesion and Coupling Principle

Organize the code of an application

- Single Responsibility Principle
  - Responsibility = a reason to change something
  - Responsibility
    - Of a function = to compute something
    - Of a module = responsibilities of all functions in the module
  - The principle of a single responsibility
    - A function / module should have one responsibility
  - Multiple responsibilities:
    - Difficulties in understanding and using
    - Impossibility of testing
    - Impossibility of reusing
    - Difficulties in maintenance and development

```
#Function with multiple responsibilities
#implement user interaction (read/print)
#implement a computation (filter)
def filterScore():
    global l
    left = input("Start score:")
    right = input("End score:")
    for el in l:
        if ((el > left) and (el < right)):
            print el
```

Organize the code of an application

- Separation of concerns
  - Process of separating a program based on features that overlap as little as possible

```
def filterScoreUI():
    global l
    inf = input("Start sc:")
    sup = input("End sc:")
    rez = filterScore(l, inf, sup)
    for e in rez:
        print(e)

def filterScore(l, left, right):
    """
    filter elements of list l
    st, end - integers limits
    return list of elements
    filtered by left and right limits
    """
    rez = []
    for el in l:
        if ((el > left) and (el < right)):
            rez.append(el)
    return rez
```

```
def testFilterScore():
    l = [5, 2, 6, 8]
    assert filterScore(l, 3, 4) == []
    assert filterScore(l, 1, 30) == l
    assert filterScore(l,3,7)==[5, 6]
```

Organize the code of an application

- Reuse Principle
  - Using modules improves the maintenance of an application
    - For instance:
      - it is easier to isolate and correct mistakes
      - modify existing functionalities
  - Using modules facilitates reuse of elements defined in the application
    - Ex. Numericlibrary module (isPrime, gcd,...) can be used in several applications
  - Managing the dependencies increases reuse
    - Dependencies between functions
      - A function invokes (calls) another function / other functions
    - Dependencies between modules
      - The functions from a module invoke functions from other modules

Organize the code of an application

- Cohesion and Coupling Principle
  - The definition of modules should consider:
    - The cohesion degree
      - – how dependent are some elements on other elements of the module
    - The coupling degree
      - – how dependent is a module on other modules

Organize the code of an application

- Cohesion and Coupling Principle
  - Cohesion
    - Measures the degree to which a module has a single, well-focused purpose
  - A module can have:
    - High cohesion – the elements of the module are highly dependent on each other
      - Ex. Rational module contains operations specific to rational numbers
    - Low cohesion – the elements relate more to other activities (and not to each other)
      - Ex. Rational module uses functions from numericlibrary module
  - A highly cohesive module performs only one task
    - Needs reduced interaction with other parts of the program
    - Advantages:
      - Such modules are easier to maintain and not frequently changed
      - More usable because they are modules designed for a well-focused purpose

Organize the code of an application

- Cohesion and Coupling Principle
  - Coupling
    - Measures the intensity of connections between modules (how much a module knows of another module)
  - Modules can have:
    - High coupling – modules that are highly dependent on each other
      - A high coupling between modules leads to difficulties in:
        - Understanding the code
        - Reuse the code
        - Isolate possible mistakes
    - Low coupling – independent modules
      - Low coupling facilitates development of applications that are easy to modify (as the interdependency between functions/modules is minimal)
  - A low coupling between modules is desired
    - ✓ Good design principle

Organize the code of an application

- Layered architecture
  - Organizing the application on layers should consider:
    - Low coupling between modules
      - Modules do not need to know details about other modules – futures changes are easier to make
    - High cohesion of each module
      - The elements of a module should be highly related

Organize the code of an application

- Layered architecture
  - Organizing the application on layers should follow an architectural design pattern that:
    - Allows design of systems
      - Flexible
      - Using components (as independent as possible)
  - Each layer communicates with the previous layer
  - Each layer has a well-defined interface that is used by the superior layer (implementation details are hidden)

Organize the code of an application

- Layered architecture
  - Generally, the architecture of a system can be designed using the following layers:
    - User Interface Layer
      - Functions, modules, classes for the user interface
      - *User Interface Layer / UI Layer / View Layer / Presentation Layer*
    - Domain Layer
      - All functions generated by the use cases (the logic of the application)
      - *Domain Layer / Business Logic Layer / Model Layer*
    - Infrastructure Layer
      - Functions with a general character, reusable
    - Coordinator

Example

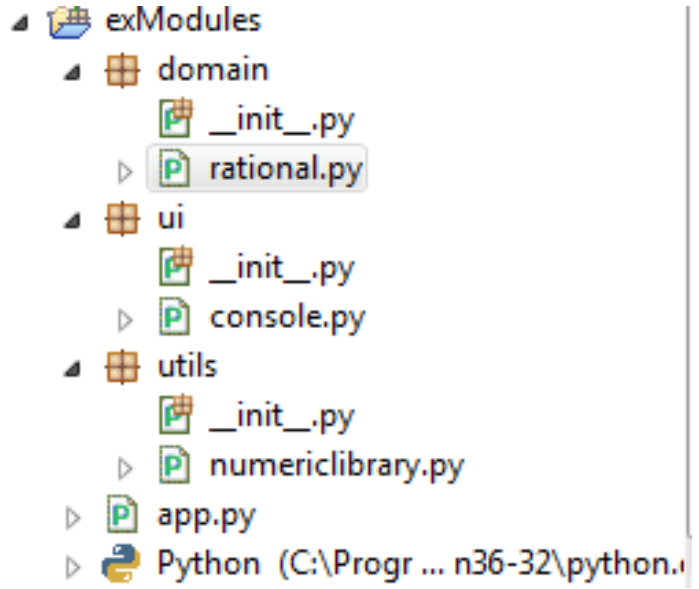
```
#UI
def filterScoreUI():
    global l
    inf = input("Start sc:")
    sup = input("End sc:")
    rez = filterScoreDomain(l, inf, sup)
    for e in rez:
        print(e)
```

```
#domain
l = [5, 2, 6, 8]
def filterScoreDomain(left, right):
    global l
    rez = filterScore(l, left, right)
    l = rez
    return rez
```

```
#utility function (infrastructure)
def filterScore(l, left, right):
    """
    filter elements of list l
    st, end - integers limits
    return list of elements
    filtered by left and right limits
    """
    rez = []
    for el in l:
        if ((el > left) and (el <right)):
            rez.append(el)
    return rez

def testFilterScore():
    l = [5, 2, 6, 8]
    assert filterScore(l, 3, 4) == []
    assert filterScore(l, 1, 30) == l
    assert filterScore(l,3,7)==[5, 6]

testFilterScore()
```

Other examples

- Rational
  - Example from lecture 3
  - Application to manage rational numbers
  - Layers: domain, ui, utils, app
- More examples
  - Next lecture



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'
```

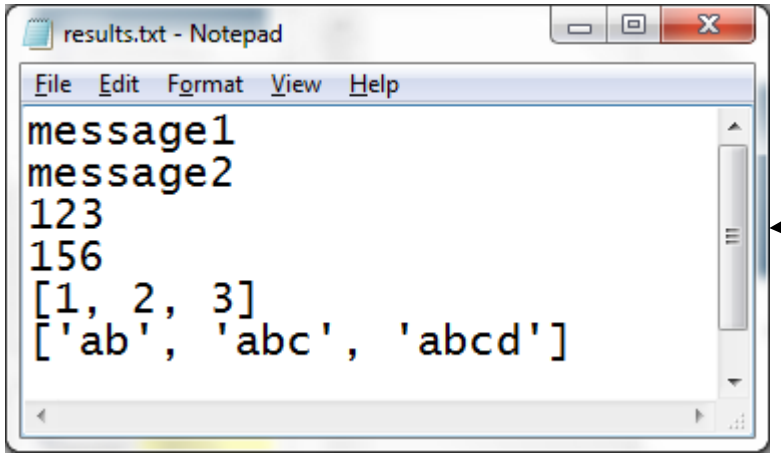
Example

- 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))
```

Recap today

- Layered Architectures (UI, Domain, Infrastructure)
- Design Principles
  - Responsibility Principle
  - Separation of Concerns Principle
  - Reuse Principle
  - Cohesion and Coupling Principle
- Working with files
  - File operations
  - Examples

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