Fundamentals of programming

Lecture 3. Modular programming

- Test-driven development
- Refactoring
- Modular programming

Test-driven development (TDD) software development process

How to write functions?

Apply test-driven development (TDD) !

TDD Steps:

- Add a test - Create automated test cases
- Run the test (initially will fail)
- Produces the minimum amount of code to pass that test
- Run the test (succeed)
- Refactor the code

TDD Step 1. Create automated test cases

When you work on a task (work-item) start by creating a test function

Work item: Compute the greatest common divider

```
def test_gcd():
    """
      test function for gdc
    """
    assert gcd(0, 2) == 2
    assert gcd(2, 0) == 2
    assert gcd(2, 3) == 1
    assert gcd(2, 4) == 2
    assert gcd(6, 4) == 2
    assert gcd(24, 9)  == 3
```

Concentrate on the specification of f.

```
def gcd(a, b):
    """
    Return the greatest common divisor of two positive integers.
    a,b integer numbers, a>=0; b>=0
    return an integer number, the  greatest common divisor of a and b
    """
    pass
```

TDD Step 2 - Run the test

```
#run the test - invoke the test function
test_gcd()
```

```
Traceback (most recent call last):
  File "C:/curs/lect3/tdd.py", line 20, in <module>  test_gcd()
  File "C:/curs/lect3/tdd.py", line 13, in test_gcd
    assert gcd(0, 2) == 2
AssertionError
```

- validates that the test function is working correctly and that the new test does not mistakenly pass without requiring any new code.
- it rules out the possibility that the new test will always pass, and therefore be worthless

TDD Step 3 – Write the code to pass the test

- concentrate on implementing the function according to pre/post-conditions and on passing all test cases
- do not concentrate on technical aspects (duplicated code, optimizations, etc).

```
def gcd(a, b):
    """
    Return the greatest common divisor of two positive integers.
    a,b integer numbers, a>=0; b>=0
    return an integer number, the  greatest common divisor of a and b
    """
    if a == 0:
        return b
    if b == 0:
        return a
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

TDD Step 4 - Invoke the test functions and see them succeed

```
>>> test_gcd()
>>>
```

If all test cases pass, the programmer can be confident that the code meets all the tested requirements.

TDD Step 5 – Refactor the code

- clean up the code using refactoring techniques.

Apply test-driven development (TDD) steps

TDD requires developers to create automated unit tests that clarify code requirements before writing the code itself.
When you create a new function (*f*), follow TDD steps:

- Add a test
    - **Define a test function (*test_f*()) which contains test cases written using *assert*ions.**
    - Concentrate on the **specification of *f***.
    - **Define *f*: name, parameters, specification, precondition, post condition, and an empty body.**
- Run all tests and see if the new one fails
    - Your program may have many functions, so **many test functions**.
    - At this stage, ensure the new **test_f() fails**, while **other test functions pass** (written previously).
- Write the body of f
    - Now the specification of f is well written and you concentrate on **implementing the function according to pre/post-conditions** and on **passing all test cases written for *f***.
    - **Do not concentrate on technical aspects (duplicated code, optimizations, etc).**
- Run all tests and see them succeed
    - Now, the developer is confident that the function meets the specification.
    - The final step of the cycle can be performed.
- Refactor code
    - Finally, you must clean up the code using refactorings techniques.

Refactoring

Code refactoring is "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior" [5].

Code smell is any symptom in the source code of a program that possibly indicates a deeper problem:

- **Duplicated code:** identical or very similar code exists in more than one location.
- **Long method:** a method, function, or procedure that has grown too large.

Refactoring: Rename method/variable

- rename a variable or a method name to something meaningful

```python
def verify(k):
    """
      Verify if a number is prime
      nr - integer number, nr>1
      return True if nr is prime
    """
    l = 2
    while l<k and k % l>0:
        l=l+1
    return l>=k
```

```python
def isPrime(nr):
    """
      Verify if a number is prime
      nr - integer number, nr>1
      return True if nr is prime
    """
    div = 2  #search for divider
    while div<nr and nr % div>0:
        div=div+1
    #if the first divider is the
    # number itself than nr is prime
    return div>=nr;
```

Refactoring: Extract Method

- You have a code fragment that can be grouped together.
- Turn the fragment into a method whose name explains the purpose of the method.

```
def startUI():
    list=[]
    print list
    #read user command
    menu = """
            Enter command:
               1-add element
               0-exit
           """
    print(menu)
    cmd=input("")
    while cmd!=0:
        if cmd==1:
            nr=input("Give element:")
            add(list, nr)
            print list
        #read user command
        menu = """
            Enter command:
               1-add element
               0-exit
           """
        print(menu)
        cmd=input("")

startUI()
```

```
def getUserCommand():
    """
     Print the application menu
     return the selected menu
    """
    menu = """
            Enter command:
               1-add element
               0-exit
           """
    print(menu)
    cmd=input("")
    return cmd

def startUI():
    list=[]
    print list
    cmd=getUserCommand()
    while cmd!=0:
        if cmd==1:
            nr=input("Give element:")
            add(list, nr)
            print list
        cmd=getUserCommand()

startUI()
```

Refactoring:Substitute Algorithm

- You want to replace an algorithm with one that is clearer.
- Replace the body of the method with the new algorithm.

```python
def isPrime(nr):
    """
      Verify if a number is prime
      nr - integer number, nr>1
      return True if nr is prime
    """
    div = 2  #search for divider
    while div<nr and nr % div>0:
        div=div+1
    #if the first divider is the
    # number itself than nr is prime
    return div>=nr;
```

```python
def isPrime(nr):
    """
      Verify if a number is prime
      nr - integer number, nr>1
      return True if nr is prime
    """
    for div in range(2,nr):
        if nr%div == 0:
            return False
    return True
```

Calculator – procedural version

Modular programming

Modular programming is a software design technique that increases the extent to which software is composed of separate, interchangeable "components"", called **modules** by breaking down program functions into modules, each of which accomplishes one function and contains everything necessary to accomplish this.

What is a module

Module is a collection of functions and variables that implements a well defined functionality

Python module definition

A module is a file containing Python statements and definitions (executable statements).
Module

- name: The file name is the module name with the suffix ".py" appended
- docstring: triple-quoted module doc string that defines the contents of the module file. Provide summary of the module and a description about the module's contents, purpose and usage.
- executable statements: function definitions, module variables,initialization code

Eclipse + PyDev IDE (Integrated Development Environment)

- **Project**
- **Package**
- **Module**
- **Run,Debug program**

Import modules

In order to use a module wee need to import it.
The import statement:

- Search the global namespace for the module. If the module exists, it had already been imported nothing more needs to be done.
- Search for the module If the module name can't be found anywhere, an **ImportError** exception is raised.
- If the module file was found, execute the statements from the module.

The executable statements (including function definitions) from a module are executed only the *first* time the module is imported somewhere.

`from` doted.package[module] `import` {module, function}}

```
from utils.numericlib import gcd

#invoke the gdc function from module utils.numericlib
print gdc(2,6)
```

```
from rational import *

#invoke the rational_add function from module rational
print rational_add(2,6,1,6)
```

```
import ui.console

#invoke the run method from the module ui.console
ui.console.run()
```

Module search path

The import statement will search for a file called modulename.py:

- the directory containing the input script
- in the list of directories specified by the environment variable **PHYTONPATH**
- in the list of directories specified by the environment variable **PYTHONHOME** an installation-dependent default path; on Unix, this is usually.:/usr/local/lib/python.

Initialize the module

The module can contain any executable statements. When the module is first imported the statements are executed. We can put some statements (other than function definition) that will do any necessary initialization of the module.

Variable scope in a module

On importing a module the variables and functions defined in the module will be inserted into a new symbol table (a new namespace). Only the module name will be added to the current symbol table

You can use the built-in **dir() dir(module_name)** function to examine the symbol tables

```
#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 gdc into the local symbol table
from utils.numericlib import gcd

#invoke the gdc function from module utils.numericlib
print gdc(2,6)
```

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

#invoke the rational_add function from module rational
print rational_add(2,6,1,6)
```

Packages

Packages are a way of structuring Python’s module namespace by using “dotted module names”
The `__init__.py` files are required to make Python treat the directories as containing packages (it can also execute initialization code for the package)

How to organize your source code

Create separate modules for:

- User interface - Functions related to the user interaction. Contains input, print operations. This is the only module where input/print operations are present
- Domain / Application – Contains functions related to the domain of the problem
- Infrastructure – Utility functions with high reuse potential
- Application coordinator – Initialize and start up the application

Calculator – modular version