Fundamentele programării

Curs 3. Programare modulară

- Refactorizare
- Module
- Organizarea aplicației pe module și pachete

Curs 2. Programare procedurală

- Funcții
- Cum se scriu funcții
- Funcții de test

Dezvoltare dirijată de teste (test-driven development - TDD)

Dezvoltarea drijată de teste presupune crearea de teste automate, chiar înainte de implementare, care clarifică cerințele

Pașii TDD pentru crearea unei funcții:

- Addaugă un test – crează teste automate
- Rulăm toate testele și verificăm ca noul test pică
- Scriem corpul funcției
- Rulăm toate testele și ne asigurăm că trec
- Refactorizăm codul

TDD Pas 1. Creare de teste automate

Când lucrăm la un task începem prin crearea unei funcții de test

Task: Calculeaza cel mai mare divizor comun

```
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
```

Ne concentrăm la specificarea funcției.

```
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 Pas 2 - Rulăm testele

```
#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
```

- Validăm că avem un test funcțional – se execută, eșuează.
- Astfel ne asigurăm că testul este executat și nu avem un test care trece fară a implementa ceva – testul ar fi inutil

TDD Pas 3 – Implementare

- implementare funcție conform specificațiilor (pre/post condișii), scopul este sa tracă testul
- soluție simplă, fără a ne concentra pe optimizări, evoluții ulterioare, cod duplicat, 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 Pas 4 – Executare funcții de test- toate cu succes

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

Dacă toate testele au trecut – codul este testat, e conform specificațiilor și nu s-au introdus erori (au trecut și testele scrise anterior)

TDD Pas 5 – Refactorizare cod

- restructurearea codului folosind refactorizări

Refactorizare

Restructurarea codului, alterând structura internă fără a modifica comportamentul observabil.
Scopul este de a face codul mai ușor de:

- înțeles
- întreținut
- extins

Semnale că este nevoie de refactorizare **(code smell) –** elemente ce pot indica probleme mai grave de proiectare**:**

- **Cod duplicat**
- **Metode lungi**
- **Liste lungi de paramtetrii**
- **Instrucțiuni condiționale care determină diferențe de comportament**

Refactorizare: Redenumire funcție/variabilă

- redenumim funcția/variabila alegând un nume sugestiv

```
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
```

```
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;
```

Refactorizare: Extragerea de metode

- parte dintr-o funcție se transformă într-o funcție separată
- o expresie se transformă într-o funcție

.

```
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()
```

Refactorizare: Substituire algoritm

```
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;
```

```
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 – versiune procedurală

Modular programming

Descompunerea programului în module (componente separate interschimbabile) având în vedere:

- separarea conceptelor
- coeziunea elementelor dintr-un modul
- cuplarea între module
- întreținerea și reutilizarea codului
- arhitectura stratificată

Modulul este o unitate structurală separată, interschimbabilă cu posibilitatea de a comunica cu alte module.
O colecție de funcții și variabile care implementează o funcționalitate bine definită

Modul în Python

Un modul în Python este un fișier ce conține instrucțiuni și definiții Python.
Modul

- **nume**: Numele fișierului este numele modulului plus extensia ".py"
  - variabila __name__
    - este __main__ dacă modulul este executat de sine stătător
    - este numele modulului
- **docstring**: Comentariu multiline de la începutul modulului. Oferă o descriere a modulului: ce conține, care este scopul, cum se foloseste, etc.
  - Variabila __doc__
- **instrucțiuni**: definiții de funcții, variabile globale per modul, cod de inișializare

Import de module

Modulul trebuie importat înainte de a putea folosi.
Instrucțiunea import:

- Caută în namespace-ul global, dacă deja există modulul înseamnă ca a fost deja importat și nu mai e nevoie de alte acțiuni
- Caută modulul și dacă nu găsește se aruncă o eroarea **ImportError**
- Dacă modulul s-a găsit, se execută instrucțiunile din modul.

Instrucțiunile din modul (înclusiv definițiile de funcții) se execută doar o singură dată (prima dată când modulul este importat în aplicație).

`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()
```

Calea unde se caută modulele (Module search path)

Instrucțiunea import caută fisierul modulname.py în:

- directorul curent (directorul de unde s-a lansat aplicația)
- în lista de directoare specificată în variabila de mediu **PHYTONPATH**
- în lista de directoare specificată în variabila de mediu **PYTHONHOME** (este calea de instalare Python; de exemplu pe Unix, în general este .:/usr/local/lib/python.

Inițializare modul

Modulul poate conține orice instrucțiuni. Când modulul este importat prima dată se execută toate instrucțiunile.Putem include instrucțiuni (altele decăt definițiile de funcții) care inițializează modulul.

Domeniu de vizibilitate în modul

La import:

- se crează un nou spațiu de nume
- variabilele și funcțiile sunt introduse în noul spațiu de nume
- doar numele modulului (__name__) este adăugat în spațiul de nume curent.

Putem folosi instrucțiunea built-in **dir() dir(module_name)** pentru a examina conținutul modulului

```
#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)
```

Pachete în Python

Modalitate prin care putem structura modulele.
Dacă avem mai multe module putem organiza într-o structură de directoare
Putem referi modulele prin notația pachet.modul

Fiecare director care conține pachete trebuie sa conțină un fișier `__init__.py`. Acesta poate conține și instrucțiuni (codul de inițializare pentru pachet)

Eclipse + PyDev IDE (Integrated Development Environment)

Eclipse: Mediu de dezvoltare pentru python (printre altele).
Pydev: Plugin eclipse pentru dezvoltare aplicații Python în Eclipse

Permite crearea, rularea, testarea, depanarea de aplicații python

Instalare:

- Java JRE 7 (varianta curenta de PyDev funcționează doar cu această versiune de Java
- Eclipse Luna (sau alta versiune de eclipse de la 3.5 în sus)
- Instalat pluginul de PyDev

Detalii pe: pydev.org
http://pydev.org/manual_101_install.html

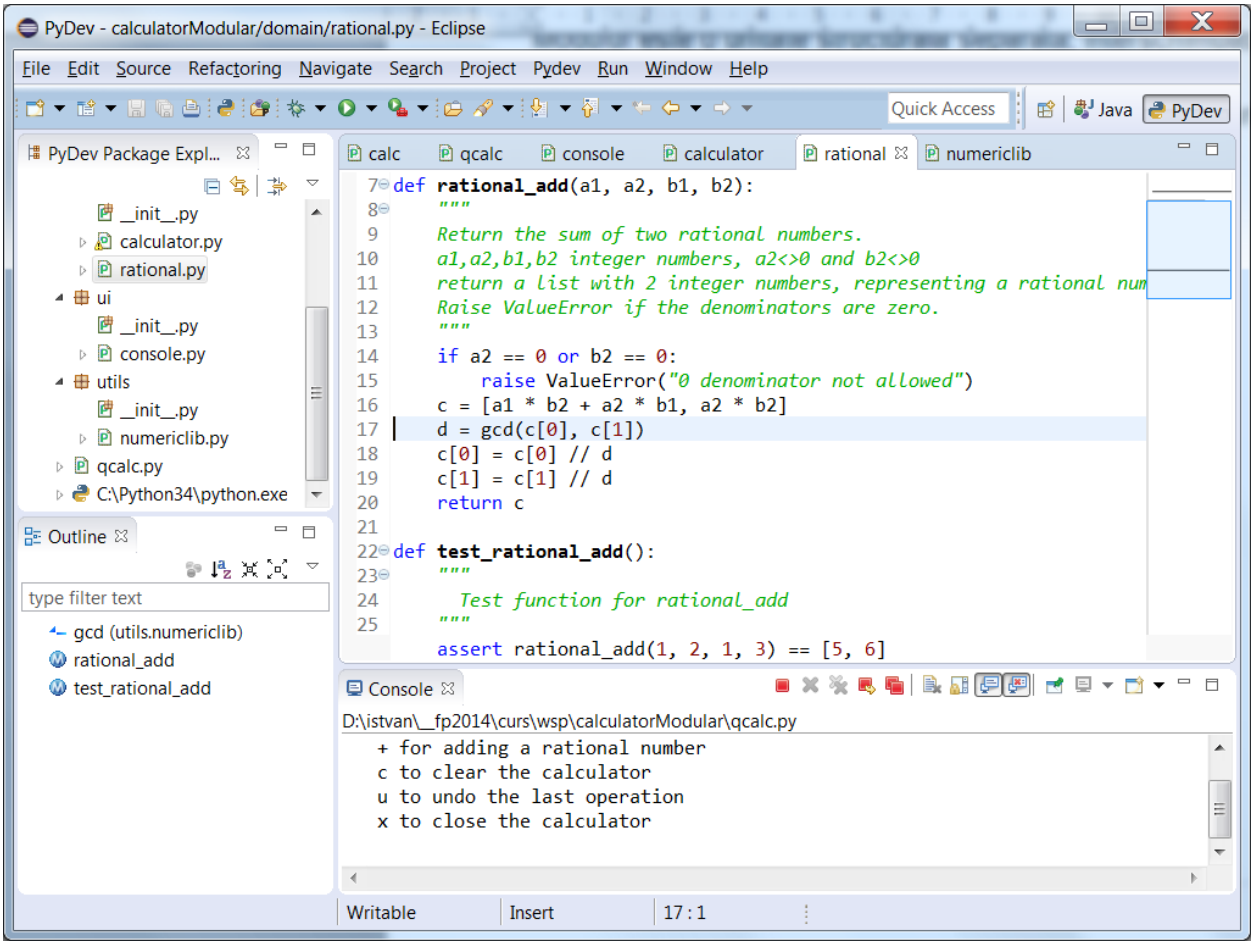
- **Proiect**
- **Editor Python**
- **ProjectExplorer: Pachete/Module**
- **Outline: Funcții**
- **Rulare/Depanare programe**

Cum organizăm aplicația pe module și pachete

Se crează module separate pentru:

- Interfață utilizator - Funcții legate de interacțiunea cu utilizatorul. Conține instrucțiuni de citire tipărire, este singurul modul care conţine tiparire-citire
- Domeniu (Domain / Application) – Conține funcții legate de domeniul problemei
- Infrastructură – Funcții utilitare cu mare potențial de refolosire (nu sunt strict legate de domeniul problemei)
- Coordonator aplicație – Inițializare/configurare și pornire aplicație

Calculator – versiune modulară



Curs 3. Programare modulară

- Refactorizare
- Module

Curs 4. Tipuri definite de utilizator

- Organizarea aplicației pe module și pachete
- Excepții
- Tipuri definite de utilizator

• Referințe

- *The Python language reference.* *http://docs.python.org/py3k/reference/index.html*
- *The Python standard library.* *http://docs.python.org/py3k/library/index.html*
- *The Python tutorial.* *http://docs.python.org/tutorial/index.html*
- Kent Beck. *Test Driven Development: By Example. Addison-Wesley Longman, 2002*. See also Test-driven development. http://en.wikipedia.org/wiki/Test-driven_development
- Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999. See also http://refactoring.com/catalog/index.html