Fundamentele programării

Curs 3. Programare modulară

- Refactorizare
- Module
- Organizarea aplicației pe module și pachete

Recapitulare:

- Scenarii de rulare
- Funcțiile sunt create o dată și folosite pentru tot parcursul existenței aplicației -> este important să fie ușor de înțeles/folosit
- Orice funcție are:
  - Documentație / Specificații
  - Teste automate
  - Înainte să implementăm ne gândim ce trebuie să facă
  - Ne gândim la cum se folosește (apelează) nu la cum e mai ușor de implementat
- Testarea este fundamentală pentru crearea de aplicații

Test-driven development, presupune crearea de teste automate chiar înainte de implementare, care verifică cerințele

Pașii TDD pentru crearea unei funcții

1. Adaugă teste automate
2. Rulăm testele și verificăm că pică (ne asigurăm că funcția nu există)
3. Scriem codul funcției
4. Rulăm toate testele și ne asigurăm că trec
5. Refactorizăm codul (îmbunătățim structura și organizarea)

TDD Pas 1. Crearea de teste automate

Când lucrăm la un task începem prin crearea unei funcții de test. Exemplu task: cmmdc

```
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 fără a implementa ceva - testul ar fi inutil

TDD Pas 3. Implementare

- implementare funcție conform specificațiilor (pre/post condiții), scopul e să treacă 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

TDD Pas 5. Refactorizare cod

- restructurarea codului (curățarea, optimizarea și eliminarea codului redundant sau ineficient)

Refactorizarea

Scopul este de a face codul mai ușor de:

- înțeles
- întreținut
- extins

Refactorizare: Redenumire funcție / variabilă cu nume sugestive

```
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: Extragere de metode

- o 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

```
.....
def run():
    """
      Implement the user interface
    """
    calc = reset_calc()
    finish = False
    while not finish:
        printCurrent(calc)
        printMenu()
        m = input().strip()
        if (m == 'x'):
            finish = True
        elif (m == '+'):
            addToCalc(calc)
        elif (m == 'c'):
            calc = reset_calc()
        elif (m == 'u'):
            undo(calc)
        else:
            print ("Invalid command")

    print ("By!!!")
.....
# run the test - invoke the test function
test_gcd()
test_rational_add()
test_calculator_add()
test_undo()

run()
```

Funcții pure

O funcție care nu are efecte secundare. Nu modifică parametrii de intrare, nu folosește variabile globale. Pentru aceleași intrări produce tot timpul aceleași rezultate.

Funcții ca și obiecte

Toate funcțiile în python sunt obiecte

- Se pot atribui la variabile
- pot fi folosite ca parametru la alte funcții
- Se pot adăuga în în containere (liste, dicționare, etc.)

```
def patrat(a):
    return a*a
```

```
def modul(a):
    if a<0: return -a
    return a
```

```
#assign a function to a variable
func = patrat
print(func(-2))
func = modul
print(func(-2))
```

```
#list of functions
lstFct = [patrat,modul]
for fct in lstFct:
    print(fct(-2))
```

```
def aplica(lista, fnc):
    """
       Aplica functia fnc pe fiecare element al listei
       lista - lista de elemente
       fnc - functie cu un singur parametru
       returneaza o lista noua
    """
    rez = []
    for el in lista:
        rez.append(fnc(el))
    return rez

print(aplica([1,2,-3,5,-2], patrat))
```

Programare modulară

Modulul este o colecție de funcții și variabile care implementează o funcționalitate bine definită.

Import de module

- from module import funtion
- import module

```
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()
```

Pachete în Python

O metodă prin care putem structura modulele sunt în directoare. Putem referi modulele prin notația pachet.modul

Cum se organizează aplicația pe module și pachete

Se creează module separate pentru:

- Interfață utilizator (conține instrucțiuni de citire tipărire)
- Domeniu (conține funcții legate de domeniul problemei)
- Infrastruct (funcții cu mare potențial de reîolosire, nu sunt strict legate de domeniul problemei)
- Coordonare aplicație (inițializare / configurare și pornire aplicație)