# Fundamentele programării

## Curs 2. Programare procedurală

- Funcții
- Cum se scriu funcții
- Funcții de test

## Curs 1. Procesul de dezvoltare software

- Ce este programarea
- Elemente de bază al limbajului Python
- Proces de dezvoltare bazat pe funcționalități

# Programare procedurală

**Paradigmă de programare**
stil fundamental de scriere a programelor, set de convenții ce dirijează modul în care gândim programele.

**Programare imperativă**
Calcule descrise prin instrucțiuni care modifică starea programului. Orientat pe acțiuni și efectele sale

**Programare procedurală**
Programul este format din mai multe proceduri (funcții, subrutine)

# Ce este o funcție

O **funcție** este un bloc de instrucțiuni de sine stătător care are:

- un ***nume***,
- poate avea o **listă de parametrii** (formali),
- poate **returna** o valoare
- are un **corp** format din instrucțiuni
- are o ***documentație*** (specificație) care include:
  - o scurtă descriere
  - *tipul* și descriere parametrilor
  - condiții impuse parametrilor de intrare (*precondiții*)
  - tipul și descrierea valorii returnate
  - condiții impuse rezultatului, condiții care sunt satisfăcute în urma executării (*post-condiții*).
  - Excepții ce pot să apară

```
def 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

def isPrime(a):
    """
      Verify if a number is prime
      a an integer value  (a>1)
      return True if the number is prime, False otherwise
    """
```

# Funcții

Toate funcțiile noastre trebuie să:

- folosească nume sugestive (pentru numele funcției, numele variabilelor)
- să oferă specificații
- să includă comentarii
- să fie testată

O funcție ca și în exemplu de mai joi, este corectă sintactic (funcționează în Python) dar la laborator/examen nu consideram astfel de funcții:

```
def f(k):
    l = 2
    while l<k and k % l>0:
        l=l+1
    return l>=k
```

Varianta acceptată este:

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

# Definiția unei funcții în Python

Folosind instrucțiunea **def** se pot definii funcții în python.
Interpretorul executa instrucțiunea def, acesta are ca rezultat introducerea numelui funcției (similar cu definirea de variabile)
Corpul funcției nu este executat, este doar asociat cu numele funcției

```
def 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
```

## Apel de funcții

Un ***bloc*** de instrucțiuni în Python este un set de instrucțiuni care este executat ca o unitate. Blocurile sunt delimitate folosind indentarea. Corpul unei funcții este un bloc de instrucțiuni și este executat în momentul în care funcția este apelată.

```
max(2,5)
```

La apelul unei funcții se creează un nou cadru de execuție, care :

- informații administrative (pentru depanare)
- determină unde și cum se continuă execuția programului (dupa ce execuția funcției se termină)
- definește două spații de nume: locals și globals care afectează execuția funcției.

# Spații de nume (namespace)

- este o mapare între nume (identificatori) și obiecte
- are funcționalități similare cu un dicționar (in general este implementat folosind tipul dicționar)
- sunt create automat de Python
- un spațiu de nume poate fi referit de mai multe cadre de execuție

Adăugarea unui nume în spațiu de nume: legare  ex: x = 2
Modificarea unei mapări din spațiu de nume: re-legare

În Python avem mai multe spațiile de nume, ele sunt create în momente diferite și au ciclu de viața diferit.

- General/implicit – creat la pornirea interpretorului, conține denumiri predefinite (built-in)
- global – creat la încărcarea unui modul, conține nume globale
  - globals() - putem inspecta spațiu de nume global
- local – creat la apelul unei funcții, conține nume locale funcției
  - locals() - putem inspecta spațiu de nume local

## Transmiterea parametrilor

**Parametru formal** este un identificator pentru date de intrare. Fiecare apel trebuie să ofere o valoare pentru parametru formal (pentru fiecare parametru obligatoriu)
**Parametru actual** valoare oferită pentru parametrul formal la apelul funcției.

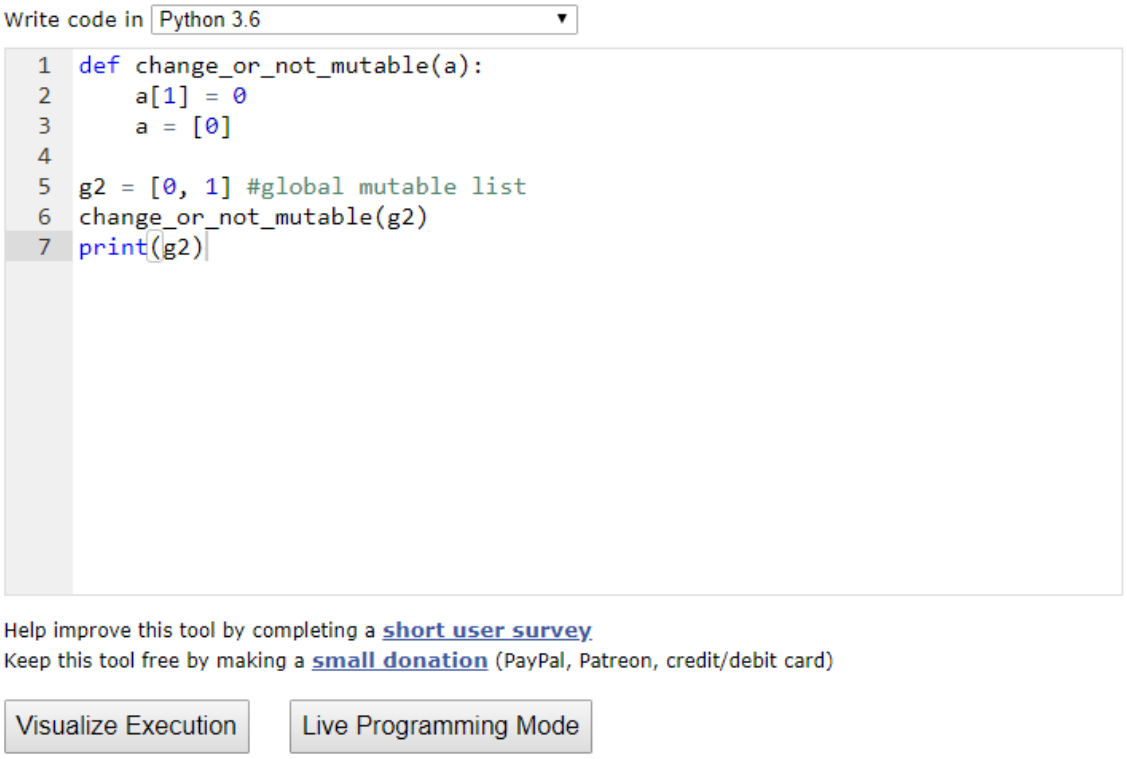
- Parametrii sunt transmiși prin referință. Parametru formal (identificatorul) este legat la valoarea (obiectul) parametrului actual.
- Parametrii sunt introduși în spațiu de nume local

```
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))
```

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

# Vizualizare memorie in timpul execuției

http://www.pythontutor.com/visualize.html




## Vizibilitatea variabilelor

Domeniul de vizibilitate (scope) – Definește vizibilitatea unui nume într-un bloc.

- Variabilele definite într-o funcție au domeniul de vizibilitate locală (funcția) – se poate accesa doar în interiorul funcției
- Variabilele definite într-un modul au vizibilitate globală (globală pe modul)
- Orice nume (variabile, funcții) poate fi folosit doar după ce a fost legat (prima atribuire)
- Parametrii formali au domeniu de vizibilitate funcția (aparțin spațiului de nume local)

```
global_var = 100

def f():
    local_var = 300
    print (local_var)
    print (global_var)
```

# Domeniu de vizibilitate

Reguli de accesare a variabilelor (sau orice nume) într-o funcție:

- când se folosește un nume de variabilă într-o funcție se caută în următoarele ordine în spațiile de nume:
  - spațiu local
  - spațiu local funcției exterioare (doar dacă avem funcție declarată în interiorul altei funcții)
  - spațiu global (nume definite în modul)
  - spațiul built-in
- operatorul = schimba/creează variabile în spațiu de nume local
- Putem folosi declarația `global` pentru a referi/importa o variabilă din spațiu de nume global în cel local
- `nonlocal` este folosit pentru a referi variabile din funcția exterioară (doar dacă avem funcții în funcții

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

f()
print (a)
```

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

f()
print (a)
```

globals() locals() - funcții built-in prin care putem inspecta spațiile de nume

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

f()
print (a)
```

# Cum scriem funcții – Cazuri de testare

Înainte să implementăm funcția scriem cazuri de testare pentru:

- a specifica funcția (ce face, pre/post condiții, excepții)
- ca o metodă de a analiza problema
- să ne punem în perspectiva celui care folosește funcția
- pentru a avea o modalitate sa testam după ce implementăm

Un caz de testare specifică datele de intrare și rezultatele care le așteptam de la funcție

Cazurile de testare:

- se pot face în format tabelar, tabel cu date/rezultate.
- executabile: funcții de test folosind `assert`
- biblioteci/module pentru testare automată

Instrucțiunea `assert` permite inserarea de aserțiuni (expresii care ar trebui sa fie adevărate) în scopul depanării/verificării aplicațiilor.

`assert` expresie

Folisim `assert` pentru a crea teste automate

# Funcții de test  - Calculator

1 Funcționalitate 1. Add a number to calculator.

2 Scenariu de rulare pentru adăugare număr

3 Activități (Workitems/Tasks)

| | |
|---|---|
| T1 | Calculează cel mai mare divizor comun |
| T2 | Sumă două numere raționale |
| T3 | Implementare calculator: init, add, and total |
| T4 | Implementare interfață utilizator |

## T1 Calculează cel mai mare divizor comun

Cazuri de testare Format tabelar

| Input: (params a,b) | Output: gdc(a,b) |
|---|---|
| 2 3 | 1 |
| 2 4 | 2 |
| 6 4 | 2 |
| 0 2 | 2 |
| 2 0 | 2 |
| 24 9 | 3 |

Funcție de test

```
def test_gcd():
    assert gcd(2, 3) == 1
    assert gcd(2, 4) == 2
    assert gcd(6, 4) == 2
    assert gcd(0, 2) == 2
    assert gcd(2, 0) == 2
    assert gcd(24, 9) == 3
```

Implementare gdc

```
def gcd(a, b):
    """
     Compute the greatest common divisor of two positive integers
     a, b integers a,b >=0
     Return the greatest common divisor of two positive integers.
    """
    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
```

# Cum se scriu funcții

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

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

Pașii TDD pentru crearea unei funcții:

- Adaugă un test
  - **Scrieți o funcție de test** (*test_f*()) care conține cazuri de testare sub forma de aserțiuni (instrucțiuni assert).
  - La acest pas ne concentram la specificațiile funcției ***f***.
  - Definim funcția *f*: nume, parametrii, precondiții, post-condiții, și corpul gol (instrucțiunea pass).
- Rulăm toate testele și verificăm ca noul test pică
  - Pe parcursul dezvoltării o sa avem mai multe funcții, astfel o să avem mai multe funcții de test .
  - La acest pas ne asigurăm ca toate testele anterioare merg, iar testul nou adăugat pică.
- Scriem corpul funcției
  - La acest pas avem deja specificațiile, ne concentrăm doar la implementarea funcției conform specificațiilor și ne asigurăm ca noile cazuri de test scrise pentru funcție trec (funcția de test)
  - **La acest pas nu ne concentrăm la aspecte tehnice** (cod duplicat, optimizări, etc).
- Rulăm toate testele și ne asigurăm că trec
  - Rulând testele ne asigurăm că nu am stricat nimic și noua funcție este implementată conform specificațiilor
- Refactorizare cod
  - La acest pas îmbunătățim codul, folosind reautorizări

## Curs 2. Programare procedurală

- Funcții
- Cum se scriu funcții
- Funcții de test

## Curs 3. Programare modulară

- Module
- Organizarea aplicației pe module și pachete

# 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