Fundamentele Programării

Limbajul de programare PYTHON

Ce este programarea

Hardware / software

Hardware - *computere*(desktop,laptop, etc) și alte dispozitive (mobile, atm, etc)
Software - *programe sau sisteme* ce rulează pe hardware
Limbaj de programare – Notații și reguli pentru scrierea de programe (sintaxă și semantică)
Python: Limbaj de programare de nivel înalt (high level programming language).
Interpretor Python: un program care permite rularea/interpretarea programelor scrise in limbajul Python.
Biblioteci Python: Funcții, module, tipuri de date disponibile în Python, scrise de alți programatori

Program 1 - Hello world
print (‘Hello world’)

Ce fac computerele

- Stochează date
    - Memoria internă
    - Memoria externă (hard, stick, CD, etc)
- Operează
    - procesor
- Comunică
    - Prin tastatură, mouse, ecran
    - Conexiuni de tip rețea

Informații și date

Date - o colecție de simboluri stocate într-un computer (Ex. 123 decimal sau şirul de caractere ‘abc’) sunt stochate folosind reprezentarea binara
Informații - interpretarea unor date (Ex. 123, ‘abc’)
Procesarea datelor și informațiilor

- Dispozitivele de intrare transformă informațiile în date (ex. 123 citit de la tastatură)
- Datele sunt stocate în memorie (ex. 1111011 pentru numărul 123)
- Dispozitivele de ieșire produc informații din date

Operații de bază ale procesoarelor

- În reprezentare binară
- Operații (and, or, not; add, etc)

Elemente de bază ale unui program Python

Program 2 - Adding two integers

```
# Reads two integers and prints the sum of them
a = input("Enter the first number: ")
b = input("Enter the second number: ")
c = int(a) + int(b)
print("The sum of ", a, " + ", b, " is ", c)
```

Elemente lexicale

Un program Python este alcătuit din mai multe linii de cod

Comentarii

- încep cu # și țin până la sfârșitul liniei
- încep cu ’’’ și țin mai multe rânduri, până la un nou ’’’

Identificatori:secvențe de caractere (litere, cifre, _) care încep cu o literă sau cu _

Literali: notații pentru valorile constante sau pentru tipuri definite de utilizator

Modelul de date

Toate datele într-un program Python – **obiecte**
Un obiect are :

- **o identitate** – adresa lui în memorie
- **un tip** – care determină operațiile posibile precum și valorile pe care le poate lua obiectul
- **o valoare**.

Odată creat, identitatea și tipul obiectului nu mai pot fi modificate.
Valoarea unor obiecte se poate modifica

- Obiecte **mutabile** - se poate modifica
- Obiecte **ne-mutabile** – nu se poate modifica

Tipuri de date standard

Tipul de date definește **domeniul** de valori posibile și **operațiile** permise asupra valorilor din domeniu.

Numerice – Numerele sunt inmutabile – odată create valoare nu se mai poate schimba (operațiile crează noi obiecte).

int (numere întregi):

- numerele întregi (pozitive și negative), dimensiune limitat doar de memoria disponibilă
- Operații: +, -, *, /, //, **, % comparare:==,!=,<, > operații pe biți: |, ^, &, <<, >>, ~
- Literali: 1, -3

bool (boolean):

- Valorile True și False.
- Operații: and, or, not
- Literali: False, True; 0, 1

float (numere reale):

- numerele reale (dublă precizie)
- Operations: +, -, *, / comparare:==,!=,<, >
- Literals: 3.14

Tipuri de date standard

Secvențe:

- Mulțimi finite și ordonate, indexate prin numere ne-negative.
- Dacă a este o secvență atunci:
    - len(a) returnează numărul de elemente;
    - a[0], a[1], …, a[len(a)-1] elementele lui a.
- Examples: [1, ‘a’]

Stringuri:

- este o secvență inmutabilă;
- caractere Unicode .
- Literali: ‘abc’, “abc”

Liste

- secvență mutabilă
- ex: [] sau [1, ‘a’, [1, 3]]

Liste

operații:

- creare `[7, 9]`
- accesare valori,lungime (**index**, **len**), modificare valori (**listele sunt mutabile**), verificare daca un element este in lista (`2` in `[1, 2, 'a']`)
- stergere inserare valori (**append**,**insert**,**pop**) del `a[3]`
- slicing, liste eterogene
- listele se pot folosi in for
- lista ca stivă(append, pop)
- folositi instructiunea help(list) pentru mai multe detalii despre operații posibile

```
# create
a = [1, 2, 'a']
print (a)
x, y, z = a
print(x, y, z)

# indices: 0, 1, ..., len(a) - 1
print a[0]
print ('last element = ', a[len(a)-1])

# lists are mutable
a[1] = 3
print a
```

```
# slicing
print a[:2]
b = a[:]
print (b)
b[1] = 5
print (b)
a[3:] = [7, 9]
print(a)
a[:0] = [-1]
print(a)
a[0:2] = [-10, 10]
print(a)
```

```
# lists as stacks
stack = [1, 2, 3]
stack.append(4)
print stack
print stack.pop()
print stack
```

```
# nesting
c = [1, b, 9]
print (c)
```

```
#generate lists using range
l1 = range(10)
print l1
l2 = range(0,10)
print l2
l3 = range(0,10,2)
print l3
l4 = range(9,0,-1)
print l4
```

```
#list in a for loop
l = range(0,10)
for i in l:
    print i
```

Tuple

Sunt secvențe inmutabile. Conține elemente, indexat de la 0
Operations:

- Crearea - packing `(23, 32, 3)`
- eterogen
- poate fi folosit in for
- unpacking

```
# Tuples are immutable sequences
# A tuple consists of a number of
values separated by commas

# tuple packing
t = 12, 21, 'ab'
print(t[0])

# empty tuple (0 items)
empty = ()
```

```
# tuple with one item
singleton = (12,)
print (singleton)
print (len(singleton))

#tuple in a for
t = 1,2,3
for el in t:
    print el
```

```
# sequence unpacking
x, y, z = t
print (x, y, z)
```

```
# Tuples may be nested
u = t, (23, 32)
print(u)
```

Dicționar

Un dicționar este o multime de perechi (cheie, valoare).
Cheile trebnuie sa fie imutabile.
Operations:

- creare `{}` sau `{'num': 1, 'denom': 2}`
- accesare valoare pe baza unei chei
- adaugare/modificare pereche (cheie, valoare)
- ștergere pereche (cheie, valoare)
- verificare dacă cheia există

`#create a dictionary`
`a = {'num': 1, 'denom': 2}`
`print(a)`
 	
`#get a value for a key`
`print(a['num'])`
	`#set a value for a key`
	`a['num'] = 3`
	`print(a)`
	`print(a['num'])`

`#delete a key value pair`
`del a['num']`
`print (a)`
	`#check for a key`
	`if 'denom' in a:`
	`    print('denom = ', a['denom'])`
	`if 'num' in a:`
	`    print('num = ', a['num'])`

Variables

Variablă:	Variablă in Python:

• nume
• valoare
• tip
  ◦ domeniu
  ◦ operații
• locație de memorie
	• nume
	• valoare
	  ◦ tip
	  ◦ domeniu
	  ◦ operații
	  ◦ locație de memorie

Introducerea unei variabile într-un program – asignare

Expressi

O combinație de valori, constante, variabile, operatori și funcții care sunt interpretate conform regulilor de precedență, calculate și care produc o altă valoare
Exemple:

- numeric : 1 + 2
- boolean: 1 < 2
- string : ‘1’ + ‘2’

Funcții utile:
help(instructiune) - ajutor
id(x) – identitatea obiectului
dir()
locals() / globals() - nume definite (variabile, funcții, module, etc)

Instrucțiuni

Operațiile de bază ale unui program. Un program este o secvență de instrucțiuni

- **Atribuire/Legare**
  - Instructiunea =.
  - Atribuirea este folosit pentru a lega un nume de o variabilă
  - Poate fi folosit și pentru a modifica un element dintr-o secvența mutabilă.
  - Legare de nume:
    - `x = 1 #x is a variable (of type int)`
  - Re-legare name:
    - `x = x + 2 #a new value is assigned to x`
  - Modificare secvență:
    - `y = [1, 2] #mutable sequence`
    - `y[0] = -1 #the first item is bound to -1`

- **Blocuri**
  - Parte a unui program care este executată ca o unitate
  - Secvență de instrucțiuni
  - Se realizează prin identarea liniilor (toate instrucțiunile identate la același nivel aparțin aceluiași bloc

Instrucțiuni - If, While

```
def gcd(a, b):
    """
    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


print gcd(7,15)
```

Instrucțiuni – For

```
#use a list literal
for i in [2,-6,"a",5]:
    print (i)

#using a variable
x = [1,2,4,5]
for i in x:
    print (i)

#using range
for i in range(10):
    print (i)

for i in range(2,100,7):
    print (i)

#using a string
s = "abcde"
for c in s:
    print (c)
```

Cum se scriu programe

Roluri în ingineria software

Programator/Dezvoltator

- Folosește calculatorul pentru a scrie/dezvolta aplicații

Client (stakeholders):

- Cel interesat/afectat de rezultatele unui proiect.

Utilizatori

- *Folosesc/rulează* programul.

Un proces de dezvoltare software este o abordare sistematică pentru construirea, instalarea, întreținerea produselor software. Indică:

- Pașii care trebuie efectuați.
- Ordinea lor

Folosim la fundamentele programării: un proces de dezvoltare incrementală bazată pe funcționalități (simple feature-driven development process)

Enunț (problem statement)

Enunțul este o descriere scurtă a problemei de rezolvat.

Calculator - Problem statement
Profesorul (client) are nevoie de un program care ajută *elevii* (users) sa învețe despre numere raționale. Programul ar trebui sa permite elevilor să efectueze operații aritmetice cu numere raționale

Cerințe (requirements)

Cerințele definesc în detaliu de ce este nevoie în program din perspectiva clientului. Definește:

- Ce dorește clientul
- Ce trebuie inclus în sistemul informatic pentru a satisface nevoile clientuli.

Reguli de elaborare a cerinţelor:

- **Cerințele exprimate corect asigură dezvoltarea sistemului conform așteptărilor clienților. (Nu se rezolvă probleme ce nu s-au cerut)**
- Descriu **lista de funcţionalităţi** care trebuie oferite de sistem.
- Funcţionalităţile trebuie să clarifice orice ambiguităţi din enunţ.

Funcţionalitate

- O funcţie a sistemului dorit de client
- descrie datele rezultatele şi partea sistemul care este afectat
- este de dimensiuni mici, poate fi implementat într-un timp relativ scurt
- se poate estima
- exprimată în forma acțiune rezultat obiect
  - Acțiunea – o funcție pe care aplicația trebuie să o furnizeze
  - Rezultatul – este obținut în urma execuției funcției
  - Obiect – o entitate în care aplicația implementează funcția

Calculator – Listă de Funcționalități
F1. **Adună** un *număr raţional* în calculator.
F2. **Sterge** calculator.
F3. **Undo** – reface ultima operaţie (utilizatorul poate repeta această operaţie).

Proces de dezvoltare incrementală bazată pe funcționalități

- Se crează lista de funcționalitați pe baza enunțului
- Se planifică iterațiile (o interație conține una/mai multe funcționalități)
- Pentru fiecare funcționalitate din iterație
    - Se face modelare – scenarii de rulare
    - Se crează o lista de tascuri (activități)
        - Se implementează și testează fiecare activitate

Iterație: O perioadă de timp în cadrul căreia se realizează o versiune stabilă și executabilă a unui produs, împreună cu documentația suport

La terminarea iterației avem un program funcțional care face ceva util clientului

Examplu: plan de iterații

Iteration	**Planned features**
I1	F1. **Adună** un *număr raţional* în calculator.
I2	F2. **Sterge** calculator.
I3	F3. **Undo** – reface ultima operaţie (utilizatorul poate repeta această operaţie).

Modelare - Iteration modeling

La fiecare început de iterație trebuie analizat funcționalitatea care urmeaza a fi implementată.
Acest proces trebuie sa sigure înțelegerea funcționalității si sa rezulte un set de pași mai mici (work item/task), activitați care conduc la realizarea funcționalității
Fiecare activitate se poate implementa/testa independent

Iterația 1 - Adună un *număr raţional* în calculator.

Pentru programe mai simple putem folosi **scenarii de rulare** (tabelară) pentru a înțelege problema și modul în care funcționalitatea se manifestă în program. Un scenariu descrie interacțiunea între utilizator și aplicație.

Scenariu pentru funcționalitatea de adaugare numar rațional

	Utilizator	Program	Descriere
a		0	Tipărește totalul curent
b	1/2		Adună un număr rațional
c		1/2	Tipărește totalul curent
d	2/3		Adună un număr rațional
e		5/6	Tipărește totalul curent
f	1/6		Adună un număr rațional
g		1	Tipărește totalul curent
h	-6/6		Adună un număr rațional
i		0	Tipărește totalul curent

Listă de activități

Recomandări:

- Definiți o activitate pentru fiecare operație care nu este implementata deja (de aplicație sa de limbajul Python), ex. T1, T2.
- Definiți o activitate pentru implementarea interacțiunii program-utilizator (User Interface), ex. T4.
- Definiți o activitate pentru a implementa operațiile necesare pentru interacțiune utilizator cu UI, ex. T3.
- Determinați dependențele între activități (ex. T4 --> T3 --> T2 -->T1, unde --> semnifică faptul ca o activitate depinde de o altă activitate).
- Faceți un mic plan de lucru (T1,T2,T3,T4)

T1	Determinare cel mai mare divizor comun (punctele g, I din scenariu)
T2	Sumă două numere raționale (c, e, g, i)
T3	Implementare calculator: init, add, and total
T4	Implementare interfață utilizator

Activitate 1. Determinare cel mai mare divizor comun

Cazuri de testare

Un **test case** conține un set de intrări și rezultatele așteptate pentru fiecare intrare.

Date: a, b	Rezultate: gcd (a, b): c, unde c este cel mai mare divizor comun
2 3	1
2 4	2
6 4	2
0 2	2
2 0	2
24 9	3
-2 0	ValueError
0 -2	ValueError

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 instructiuni
- are o ***documentație*** (specificație) care include:
    - o scurtă descriere
    - *tipul* și descriere parametrilor
    - condiții impuse paramterilor 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
    #if the first divider is the number itself than 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 identarea.
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 crează 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 folisind 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 incărcarea 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] = 1
    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))
```

Vizibilitatea variabilelor

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

- Variabilele definite într-o funcție au domenul 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 dupa 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:

- cand 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 interorul altei funcții)
  - spațiu global (nume definite în modul)
  - spațiul built-in
- operatorul = schimba/crează 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 (expressi 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 drijată de teste presupune crearea de teste automate, chiar înainte de implementare, care clarifică cerințele

Pașii TDD pentru crearea unei funcții:

- Addaugă 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 conentrăm la aspecte technice (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 inbunătățim codul, folosind refactorizări

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

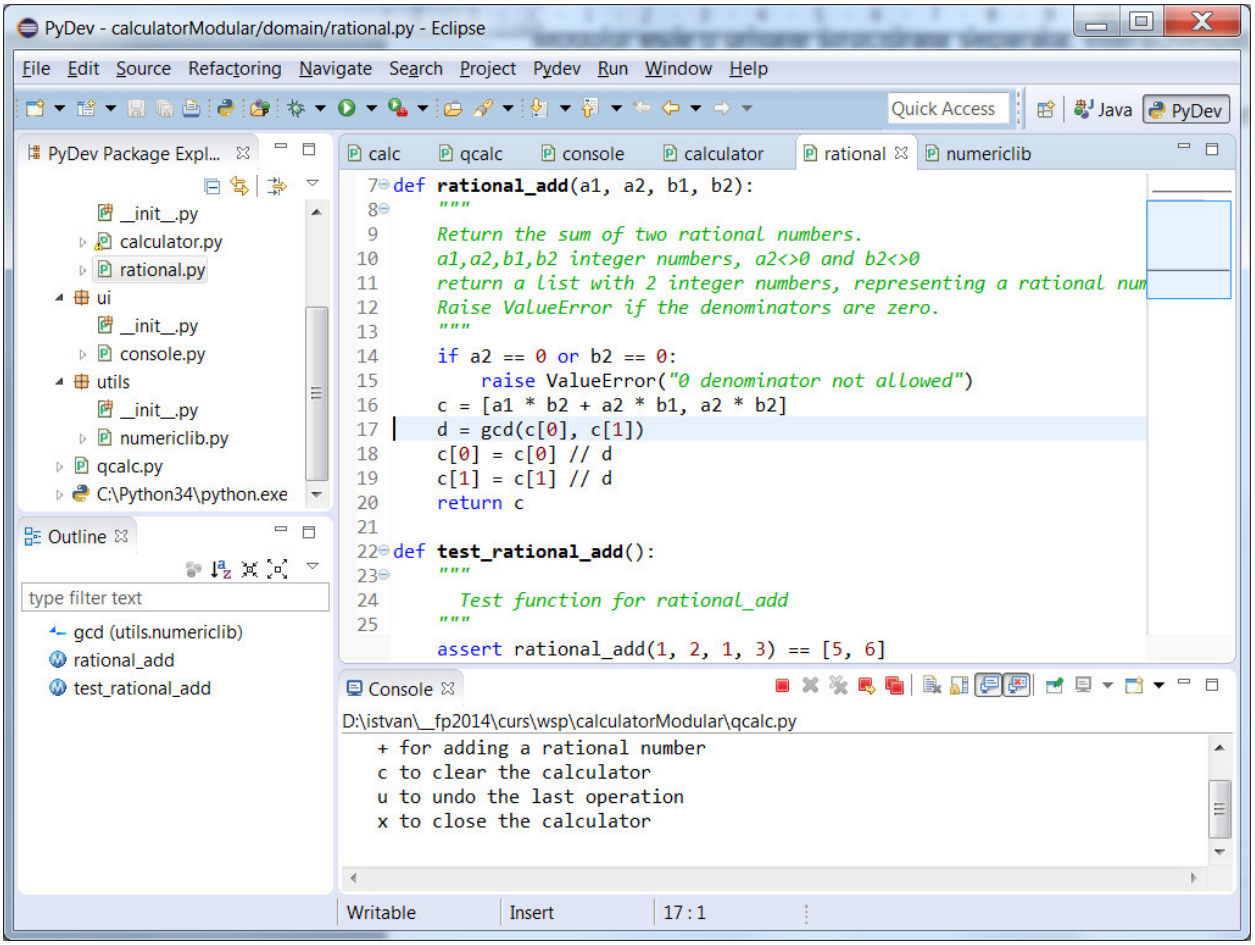
- **Proiect**
- **Editor Python**
- **ProjectExplorer: Pachete/Module**
- **Outline: Funcții**
- **Rulare/Depanare programe**

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



Organizarea aplicației pe funcții și module

Responsabilități

Responsabilitate – motiv pentru a schiba ceva

- responsabilitate pentru o funcție: efectuarea unui calcul
- responsabilitate modul: responsabilitățile tuturo funcțiilor din modul

Principul unei singure responsabilități - Single responsibility principle (SRP)

O funcție/modul trebuie să aibă o singură responsabilitate (un singur motiv de schimbare).

```
#Function with multiple responsibilities
#implement user interaction (read/print)
#implement a computation (filter)
def filterScore():
    st = input("Start score:")
    end = input("End score:")
    for c in l:
        if c[1]>st and c[1]<end:
            print c
```

Multiple responsibilități conduc la

- Dificultăți în înțelegere și utilizare
- Imposibilitatea de a testa
- Imposibilitatea de a refolosi
- Dificultăți la întreținere și evoluție

Separation of concerns

Principiu separării responsabilităților - Separation of concerns (**SoC**)
procesul de separare a unui program în responsabilități care nu se suprapun

```
def filterScoreUI():
   st = input("Start sc:")
   end = input("End sc:")
   rez = filtrareScore(l,st, end)
   for e in rez:
        print e


def filterScore(l,st, end):
    """
    filter participants
    l - list of participants
    st, end - integers -scores
    return list of participants
        filtered by st end score
    """
    rez = []
    for p in l:
        if p[1]>st and p[1]<end:
            rez.append(p)
    return rez
```

```
def testScore():
  l = [["Ana", 100]]
  assert filterScore(l,10,30)==[]
  assert filterScore(l,1,30)==l
  l = [["Ana", 100],["Ion", 40],["P", 60]]
 assert filterScore(l,3,50)==[["Ion", 40]]
```

Dependențe

- funcția: apelează o altă funcție
- modul: orice funcție din modul apelază o funcție din alt modul

Pentru a ușura întreținerea aplicației este nevoie de gestiunea dependențelor

Cuplare

Măsoară intensitatea legăturilor dintre module/funcții

Cu cât există mai multe conexiuni între module cu atât modulul este mai greu de înțeles, întreținut, refolosit și devine dificilă izolarea prolemelor ⇒cu cât gradul de cuplare este mai scăzut cu atât mai bine

Gradul de cuplare scăzut(Low coupling) facilitează dezvoltarea de aplicați care pot fi ușor modificate (interdependența între module/funcții este minimă astfel o modificare afectează doar o parte bine izolată din aplicație)

Coeziunea

Măsoară cât de relaționate sunt resposabilitățile unui element din program (pachet,modul, clasă)
Modulul poate avea:

- **Grad de coeziune ridicat (High Cohesion)**: elementele implementează responsabilități înrudite
- Grad de coeziune scăzut (Low Cohesion): implementează responsabilități diverse din arii diferite (fără o legatură conceptuală între ele)

Un modul puternic coeziv ar trebui să realizeze o singură sarcină și sa necesite interacțiuni minime cu alte părți ale programului.

Dacă elementele modulului implementeaza responsabilități disparate cu atât modulul este mai greu de înțeles/întreținut $\Rightarrow$Modulele ar trebui sa aibă grad de coeziune ridicat

Arhitectură stratificată (Layered Architecture)

Structurarea applicației trebuie să aibă în vedere :

- Minimizarea cuplării între module (modulele nu trebuie sa cunoască detalii despre alte module, astfel schimbările ulteroare sunt mai ușor de implementat)
- Maximizare coeziune pentru module (conținutul unui modul izoleaza un concept bine definit)

Arhitectură stratificată – este un șablon arhitectural care permite dezvoltarea de sisteme flexibile în care componentele au un grad ridicat de independență

- Fiecare strat comunică doar cu startul imediat următor (depinde doar doar de stratul imediat următor)
- Fiecare strat are o interfață bine definită (se ascun detaliile), interfață folosită de stratul imediat superior

Arhitectură stratificată

- Nivel prezentare (User interface / Presentation)
  - implementează interfața utilizator (funcții/module/clase)

- Nivel logic (Domain / Application Logic)
  - oferă funcții determinate de cazurile de utilizare
  - implementeaza concepte din domeniul aplicației
- Infrastructură
  - funcții/module/clase generale, utilitare

- Coordonatorul aplicației (Application coordinator)
  - asamblează și pornește aplicația

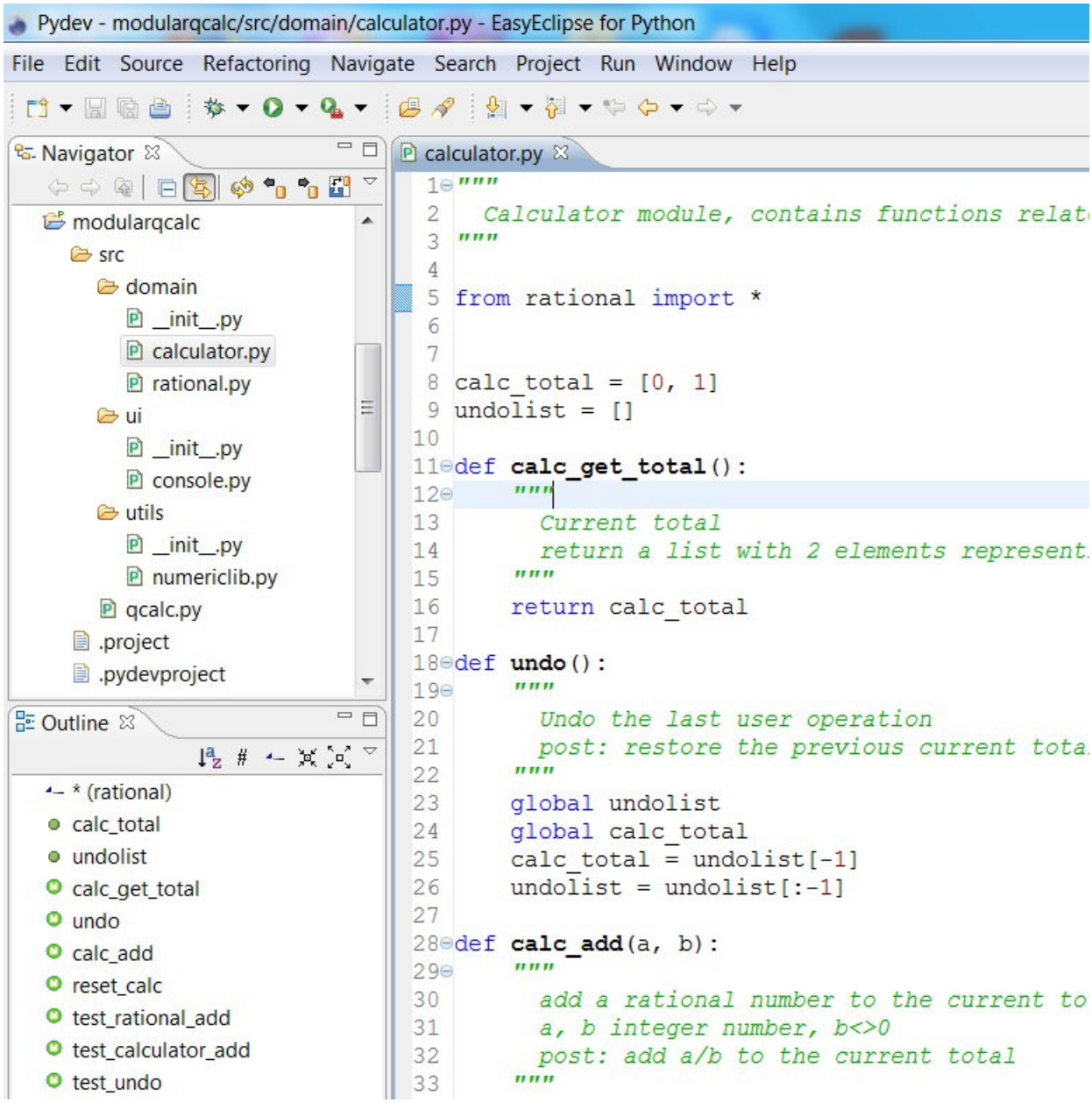
Layered Architecture – simple example

```
#Ui
def filterScoreUI():                                #manage the user interaction
    st = input("Start sc:")
    end = input("End sc:")
    rez = filterScoreDomain(st, end)
    for e in rez:
        print (e)
```

```
#domain
l = [["Ion",50],["Ana",30],["Pop",100]]
def filterScoreDomain(st, end):                     #filter the score board
    global l
    rez = filterMatrix(l, 1, st, end)
    l = rez
    return rez
```

```
#Utility function - infrastructure
def filterMatrix(matrice, col, st, end):            #filter matrix lines
    linii = []
    for linie in matrice:
        if linie[col]>st and linie[col]<end:
            linii.append(linie)
    return linii
```

Organizarea proiectelor pe pachete/module



Erori și excepții

Erori de sintaxă – erori ce apar la parsarea codului

```
while True print("Ceva"):
    pass
```

```
  File "d:\wsp\hhh\aa.py", line 1
    while True print("Ceva"):
                     ^
SyntaxError: invalid syntax
```

Codul nu e corect sintactic (nu respectă regulile limbajului)

Excepții

Erori detectate în timpul rulării.

Excepțiile sunt aruncate în momentul în care o eroare este detectată:

- pot fi aruncate de interpretorul python
- aruncate de funcții pentru a semnala o situație exceptională, o eroare
  - ex. Nu sunt satisfăcute precondițiile

```
>>> x=0
>>> print 10/x

Trace back (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print 10/x
ZeroDivisionError: integer division or modulo by zero
```

```
def rational_add(a1, a2, b1, b2):
    """
    Return the sum of two rational numbers.
    a1,a2,b1,b2 integer numbers, a2<>0 and b2<>0
    return a list with 2 int, representing a rational number a1/b2 + b1/b2
    Raise ValueError if the denominators are zero.
    """
    if a2 == 0 or b2 == 0:
        raise ValueError("0 denominator not allowed")
    c = [a1 * b2 + a2 * b1, a2 * b2]
    d = gcd(c[0], c[1])
    c[0] = c[0] / d
    c[1] = c[1] / d
    return c
```

Modelul de execuție (Execution flow)

Excepțiile intrerup execuția normală a instrucțiunilor
Este un mechanism prin care putem întrerupe execuția normală a unui bloc de instrucțiuni
Programul contiună execuția în punctul în care excepția este tratată (rezolvată) sau întrerupe de tot programul

```
def compute(a,b):
    print ("compute :start ")
    aux = a/b
    print ("compute:after division")
    rez = aux*10
    print ("compute: return")
    return rez

def main():
    print ("main:start")
    a = 40
    b = 1
    c = compute(a, b)
    print ("main:after compute")
    print ("result:",c*c)
    print ("main:finish")

main()
```

Tratarea excepțiilor (Exception handling)
Procesul sistematic prin care excepțiile apărute în program sunt gestionate, executân acțiuni necesare pentru remedierea situației.

```
try:
     #code that may raise exceptions
     pass
except ValueError:
     #code that handle the error
     pass
```

Excepțiile pot fi tratate în blocul de instrucțiuni unde apar sau în orice bloc exterior care in mod direct sau indirect a apelat blocul în care a apărut excepția (excepția a fost aruncată)
Dacă excepția este tratată, acesta oprește rularea programului

raise, try-except statements

```
try:
     calc_add (int(m), int(n))
     printCurrent()
except ValueError:
     print ("Enter integers for m, n, with n!=0")
```

Tratarea selectivă a excepțiilor

- avem mai multe clauze `except`,
- este posibil sa propagăm informații despre excepție
- clauza *`finally`* se execută în orice condiții (a apărut/nu a apărut excepția)
- putem arunca excepții proprii folosind `raise`

```
def f():
#    x = 1/0
    raise ValueError("Error Message") #   aruncăm excepție

try:
    f()
except ValueError as msg:
    print "handle value error:", msg
except KeyError:
    print "handle key error"
except:
    print "handle any other errors"
finally:
    print ("Clean-up code here")
```

Folosiți excepții doar pentru:

- A semnala o eroare – semnalam situația în care funcția nu poate respecta post condiția, nu poate furniza rezultatul promis în specificații
- Putem folosi pentru a semnala încălcarea precondițiilor

Nu folosiți excepții cu singurul scop de a altera fluxul de execuție

Specificații

- Nume sugestiv
- scurta descriere (ce face funcția)
- tipul și descrierea parametrilor
- condiții asupra parametrilor de intrare (precondiții)
- tipul, descrierea rezultatului
- relația între date și rezultate (postcondiții)
- **Excepții** care pot fi aruncate de funcție, și condițiile in care se aruncă

```
def gcd(a, b):
    """
    Return the greatest common divisor of two positive integers.
    a,b integer numbers
    return an integer number, the  greatest common divisor of a and b
    Raise ValueError if a<=0 or b<=0
    """
```

Review calculator modular

Cateva probleme:

- Starea calculatorului:
  - varianta cu variabilă globală:
    - avem mai multe variabile globale care pot fi cu ușurintă accesate din exterior (posibil stricand starea calculatorului)
    - variabila globală face testarea mai dificilă
    - nu este o legătură clară între aceste variabile (starea calculatorului este împrăștiat în cod)
  - varianta fară variabile globale:
    - starea calculatorului este expus (nu există garanții ca metodele se apeleaza cu un obiect care reprezintă calculatorul)
    - trebuie sa transmitem starea, ca parametru pentru fiecare funcție legată de calculator
- Numere raționale
  - reprezentarea numerelor este expusa: ex: `rez= suma(total[0],total[1],a,b)` , putem cu ușurința altera numărul rațional (ex. Facem `total[0] = 8` care posibil duce la încălcarea reguli cmmdc(a,b) ==1 pentru orice numărul rațional a/b
  - codul pentru adunare, înmultire, etc de numere rationale este diferit de modul in care facem operații cu numere intregi. Ar fi de preferat sa putem scrie r = r1+r2 unde r,r1,r2 sunt numere rationale

Programare orientată obiect

Este metodă de proiectare şi dezvoltare a programelor:

- Oferă o abstractizare puternică și flexibilă
- Programatorul poate exprima soluția în mod mai natural (se concentrează pe structura soluției nu pe structura calculatorului)
- Descompune programul într-un set de obiecte, obiectele sunt elementele de bază
- Obiectele interacționeaza pentru a rezolva problema, există relații între clase
- Clasele introduc tipuri noi de date, modeleaza elemente din spațiul problemei, fiecare obiect este o instanța a unui tip de data (clasă)

Clasă

Defineste in mod abstract caracteristicile unui lucru.
Descrie două tipuri de atribute:

- câmpuri (proprietati) – descriu caracteristicile
- metode (operații) – descriu comportamentul

Clasele se folosesc pentru crearea de noi tipuri de date (tipuri de date definite de utilizator)
Tip de date:

- domeniu
- operatii

Clasele sunt folosite ca un sablon pentru crearea de obiecte (instante), clasa defineste elementele ce definesc starea si comportamentul obiectelor.

Definitie de clasă în python

```
class MyClass:
    <statement 1>
    ....
    <statement n>
```

Este o instrucțiune executabilă, introduce un nou tip de date cu numele specificat.

Instrucțiunile din interiorul clasei sunt în general definiții de funcții, dar și alte instrucțiuni sunt permise

Clasa are un spațiu de nume propriu, definițiile de funcții din interiorul clasei (metode) introduc numele funcțiilor în acest spațiu de nume nou creat. Similar și pentru variabile

Obiect

Object (instanță) este o colecție de date și funcții care operează cu aceste date

Fiecare obiect are un tip, este de tipul clasei asociate: este instața unei clase

Obiectul:

- inglobează o stare: valorile campurilor
- folosind metodele:
  - putem modifica starea
  - putem opera cu valorile ce descriu starea obiectelor

Fiecare obiect are propiul spațiu de nume care conține campurile și metodele.

Creare de obiecte. Creare de instanțe a unei clase (__init__)

Instanțierea une clase rezulte in obiecte noi (instanțe). Pentru creara de obiecte se foloseste notație similară ca și la funcții.

```
x = MyClass()
```

Operația de instanțiere (“apelul” unei clase) crează un obiect nou, obiectul are tipul MyClass

O clasă poate defini metoda specială __init__ care este apelată în momentul instanțierii

```
class MyClass:
    def __init__(self):
        self.someData = []
```

__init__ :

- crează o instanță
- foloseste “self” pentru a referi instanța (obiectul) curent (similar cu “this” din alte limbaje orientate obiect)

Putem avea metoda __init__ care are și alți parametrii în afară de self

Campuri

```
x = RationalNumber(1,3)
y = RationalNumber(2,3)
x.m = 7
x.n = 8
y.m = 44
y.n = 21
```

```
class RationalNumber:
    """
       Abstract data type for rational numbers
       Domain: {a/b where a and b are integer numbers b!=0}
    """

    def __init__(self, a, b):
        """
          Creates a new instance of RationalNumber
        """
        #create a field in the rational number
        #every instance (self) will have this field
        self.n = a
        self.m = b
```

self.n = a vs n=a

1 Crează un atribut pentru instanța curentă
2 Crează o variabilă locală funcției

Metode

Metodele sunt funcții definite in interiorul clasei care au acces la valorile campurilor unei instanțe.

În Python metodele au un prim argument: instanța curentă
Toate metodele primesc ca prim parametru obiectul curent (self)

```
def testCreate():
    """
      Test function for creating rational numbers
    """
    r1 = RationalNumber(1,3)  #create the rational number 1/3
    assert r1.getNominator()==1
    assert r1.getDenominator()==3
    r1 = RationalNumber(4,3)  #create the rational number 4/3
    assert r1.getNominator()==4
    assert r1.getDenominator()==3

class RationalNumber:
    """
      Abstract data type rational numbers
      Domain: {a/b  where a,b integer numbers, b!=0, greatest common divisor
a, b =1}
    """
    def __init__(self, a, b):
        """
          Initialize a rational number
          a,b integer numbers
        """
        self.__nr = [a, b]


    def getDenominator(self):
        """
           Getter method
           return the denominator of the rational number
        """
        return self.__nr[1]

    def getNominator(self):
        """"
          Getter method
          return the nominator of the method
        """
        return self.__nr[0]
```

Metode speciale. Suprâncărcarea operatorilor. (Operator overloading)

__str__ - conversie in tipul string (print representation)

```
def __str__(self):
        """
          provide a string representation for the rational number
          return a string
        """
        return str(self.__nr[0])+"/"+str(self.__nr[1])
```

__lt__ , **__le__**, **__gt__**, **__ge__**- comparații (<,<=,>,>=)

```
def testCompareOperator():
    """
    Test function for < >
    """
    r1 = RationalNumber(1, 3)
    r2 = RationalNumber(2, 3)
    assert r2>r1
    assert r1<r2
```

```
def __lt__(self, ot):
        """
          Compare 2 rational numbers (less than)
          self the current instance
          ot a rational number
          return True if self<ot,False otherwise
        """
        if self.getFloat()<ot.getFloat():
            return True
        return False
```

__eq__ - verify if equals

```
def testEqual():
    """
      test function for ==
    """
    r1 = RationalNumber(1, 3)
    assert r1==r1
    r2 = RationalNumber(1, 3)
    assert r1==r2
    r1 = RationalNumber(1, 3)
    r1 = r1.add(RationalNumber(2, 3))
    r2 = RationalNumber(1, 1)
    assert r1==r2
```

```
def __eq__(self, other):
        """
        Verify if 2 rational are equals
        other - a rational number
        return True if the instance is
equal with other
         """
         return self.__nr==other.__nr
```

Operator overloading

__add__(self, other) - pentru a folosi operatorul “+”

```
def testAddOperator():
    """
     Test function for the + operator
    """
    r1 = RationalNumber(1,3)
    r2 = RationalNumber(1,3)
    r3 = r1+r2
    assert r3 == RationalNumber(2,3)
```

```
def __add__(self,other):
        """
        Overload + operator
          other  - rational number
        return a rational number,
          the sum of self and other
        """
        return self.add(other)
```

Metoda **__mul__(self, other)** - pentru operatorul “*”

Metoda **__setItem__(self,index, value)** – dacă dorim ca obiectele noastre sa se comporte similar cu liste/dicționare, sa putem folosi “[]”

```
a = A()
a[index] = value
```

__getItem__(self, index) – sa putem foloi obiecul ca si o secvență

```
a = A()
for el in a:
  pass
```

__len__(self) **-** pentru len

__getslice__(self,low,high) **-** pentru operatorul de slicing

```
a = A()
b = a[1:4]
```

__call__(self, arg) - to make a class behave like a function, use the “()”

```
a = A()
a()
```

Vizibilitate și spații de nume în Python

Spațiu de nume (*namespace)* este o mapare intre nume și obiecte
Namespace este implementat în Python folosind dictionarul
  Cheie: Nume
  Valoare – Object

Clasa introduce un nou spațiu de nume
Metodele, campurile sunt înt-un spațiu de nume sparat, spațiu de nume corespunzător clase.

Toate regulile (legare de nume, vizibilitate/scope, paramterii formali/actuali, etc.) legate de denumiri(funcțiion, variable) sunt acelasi pentru attributele clasei (methode, campuri) ca si pentru orice alt nume in python, doar trebuie luat în considerare ca avem un namespace dedicat clasei

Atribute de clasă vs atribute de instanțe

Variabile membre (câmpuri)

- atribute de instațe – valorile sunt unice pentru fiecare instanță (obiect)
- atribute de clasă – valoarea este partajata de toate instanțele clasei (toate obiectele de același tip)

```
class RationalNumber:
    """
       Abstract data type for rational numbers
       Domain: {a/b where a and b are integer numbers b!=0}
    """
    #class field, will be shared by all the instances
    numberOfInstances =0

    def __init__(self, a, b):
        """
          Creates a new instance of RationalNumber
        """
        self.n = a
        self.m = b
        RationalNumber.numberOfInstances+=1   # accessing class fields

    numberOfInstances =0
    def __init__(self,n,m):
        self.n = n
        self.m = m
        RationalNumber.numberOfInstances+=1


def testNumberInstances():
    assert RationalNumber.numberOfInstances == 0
    r1 = RationalNumber(1,3)
    #show the class field numberOfInstances
    assert  r1.numberOfInstances==1
    # set numberOfInstances from the class
    r1.numberOfInstances = 8
    assert r1.numberOfInstances==8  #access to the instance field
    assert RationalNumber.numberOfInstances==1  #access to the class field

testNumberInstances()
```

Class Methods

Funcții din clasă care nu opereaza cu o instanța.
Alte limbaje: metode statice

```
class RationalNumber:
    #class field, will be shared by all the instances
    numberOfInstances =0

    def __init__(self,n,m):
        """
         Initialize the rational number
         n,m - integer numbers
        """
        self.n = n
        self.m = m
        RationalNumber.numberOfInstances+=1

    @staticmethod
    def getTotalNumberOfInstances():
        """
          Get the number of instances created in the app
        """
        return RationalNumber.numberOfInstances

def testNumberOfInstances():
    """
     test function for getTotalNumberOfInstances
    """
    assert RationalNumber.getTotalNumberOfInstances()==0
    r1 = RationalNumber(2, 3)
    assert RationalNumber.getTotalNumberOfInstances()==1

testNumberOfInstances()
```

ClassName.attributeName – folosit pentru a accesa un atribut asociat clasei (camp,metoda)
Decoratorul **@staticmethod** este folosit pentru a marca o funcție statică. Aceste funcții nu au ca prim argument (self) obiectul curent.

Principii pentru crearea de noi tipuri de date

Încapsulare

Datele care reprezintă starea și metodele care manipuleaza datele sunt strâns legate, ele formează o unitate coezivă.
Starea si comportamentul ar trebui încapsulat în acelasi unitate de program (clasa)

Ascunderea informațiilor

Reprezentarea interna a obiectelor (a stării) trebuie protejat față de restul aplicației. Ascunderea reprezentării protejează integritatea datelor și nu permite modificarea starii din exteriorul clasei, astfel se evită setarea, accidentala sau voita, unei stari inconsistente.

Clasa comunica cu exteriorul doar prin interfața publică (mulțimea tuturor metodelor vizibile in exterior) și ascunde orice detalii de implementare (modul în care am reprezentat datele, algoritmii folosiși,etc).

De ce:

Definirea unei interfețe clare și ascunderea detaliilor de implementare asigură ca alte module din aplicație sa nu pot face modificări care ar duce la stari inconsistente. Permite evoluția ulterioară (schimbare reprezentare, algoritmi etc) fără să afectăm restul aplicației

Limitați interfața (metodele vizibile în exterior) astfel încât să existe o libertate în modificarea implementării (modificare fără a afecta codul client)

Codul client trebuie să depindă doar de interfața clasei, nu de detalii de implementare. Dacă folosiți acest principiu, atunci se pot face modificări fără a afecta restul aplicației

Membri publici. Membrii privați – Ascunderea implementării in Python

Trebuie sa protejăm (ascundem) reprezentarea internă a clasei (implementarea)

In Python ascunderea implementării se bazeaza pe convenții de nume.
_name sau __name pentru un atribut semnaleaza faptul ca atributul este “privat”

Un nume care incepe cu _ sau __ semnaleaza faptul ca atributul (camp, metode) ar trebui tratat ca fiind un element care nu face parte din interfața publică. Face parte din reprezentarea internă a clasei, nu ar trebui accesat din exterior.

Recomandări

- Creați metode pentru a accesa campurile clasei (getter)
- folositi convențiile de nume _,__ pentru a delimita interfața publică a clasei de detaliile de implementare
- Codul client ar trebui sa funcționeze (fără modificări) chiar daca schimbam reprezentarea internă, atâta timp cât interfața publică rămâne neschimbată. Clasa este o abstractizare, o cutie neagra (black box)
- Specificațiile funcțiilor trebuie sa fie independente de reprezentare

Cum creăm clase

Folosim Dezvoltare dirijată de teste

Specificațiile (doumentația) pentru clase includ:

- scurtă descriere
- domeniul – ce fel de obiecte se pot crea. În general descrie campurile clasei
- Constrângeri ce se aplică asupra datelor membre: Ex. Invariants – condiții care sunt adevărate pentru întreg ciclu de viața al obiectului

```
class RationalNumber:
    """
      Abstract data type rational numbers
      Domain:{a/b where a,b integer numbers, b!=0, greatest common divisor a, b =1}
      Invariant:b!=0, greatest common divisor a, b =1
    """
    def __init__(self, a, b):
```

Se creaza funcții de test pentru:

- Crearea de instanțe
- Fiecare metodă din clasă

Campurilie clasei (reprezentarea) se declară private (__nume). Se crează metode getter pentru a accesa câmpurile clasei

Tipuri abstracte de date (Abstract data types)

Tip abstract de date:

- operațiile sunt specificate independent de felul în care operația este implementată
- operațiile sunt specificate independent de modul de reprezentare a datelor

Un tip abstract de date este: Tip de date+ Abstractizarea datelor + Încapsulare

Review Calculator rational – varianta orientat obiect

Putem schimba cu usurința reprezentarea internă pentru clasa RationalNumber (folosim a,b în loc de lista [a,b])

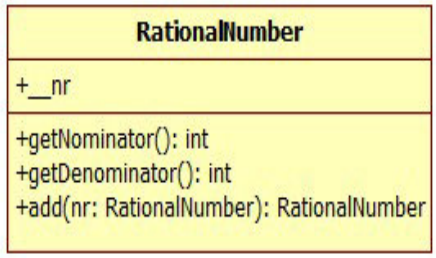
Diagrame UML

Unified Modeling Language (UML) - este un limbaj standardizat de modelare destinat vizualizării, specificării, modelării și documentării aplicațiilor.

UML include un set de notații grafice pentru a crea modele vizuale ce descriu sistemul.

Diagrame de clase

Diagrama UML de clase (UML Class diagrams) descrie structura sistemului prezentând clasele,atributele și relațiile intre aceste clase




```
class RationalNumber:
    def __init__(self, a, b):
        """
          Initialize a rational number
          a,b integer numbers
        """
        self.__nr = [a, b]
    def getDenominator(self):
        """
           Getter method return the
denominator
        """
        return self.__nr[1]
    def getNominator(self):
        """"
          Getter method return the nominator
        """
        return self.__nr[0]
    def add(self, a):
```

Clasele sunt reprezentate prin dreptunghiuri ce conțin trei zone:

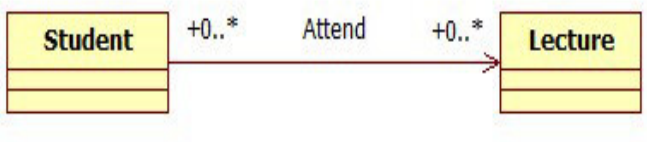
- Partea de sus – numele clasei
- Partea din mijloc – câmpurile/atributele clasei
- Partea de jos – metodele/operațiile

Relații UML

O relație UML este un termen general care descrie o legatură logică între două elemente de pe o diagramă de clase.
Un *Link* este relația între obiectele de pe diagramă. Este reprezentată printr-o linie care conecteaza două sau mai multe dreptunghiuri.

Associeri

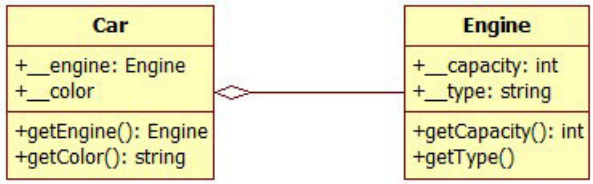
Asocierile binare se reprezintă printr-o linie între două clase.



O asociere poate avea nume, capetele asocieri pot fi adnotate cu nume de roluri, multiplicitate, vizibilitate și alte proprietăți. Asocierea poate fi uni-direcțională sau bi-direcțională

Agregare

Agregarea este o asociere specializată. Este o asociere ce reprezintă relația de parte-întreg (part-whole) sau apartenența (part-of).




```
class Car:
    def __init__(self,eng,col):
        """
          Initialize a car
          eng - engine
          col - string, ie White
        """
        self.__engine = eng
        self.__color = col
    def getColor(self):
        """
          Getter method for color
          return string
        """
        return self.__color
    def getEngine(self):
        """
          Getter method for engine
          return engine
        """
        return self.__engine
```

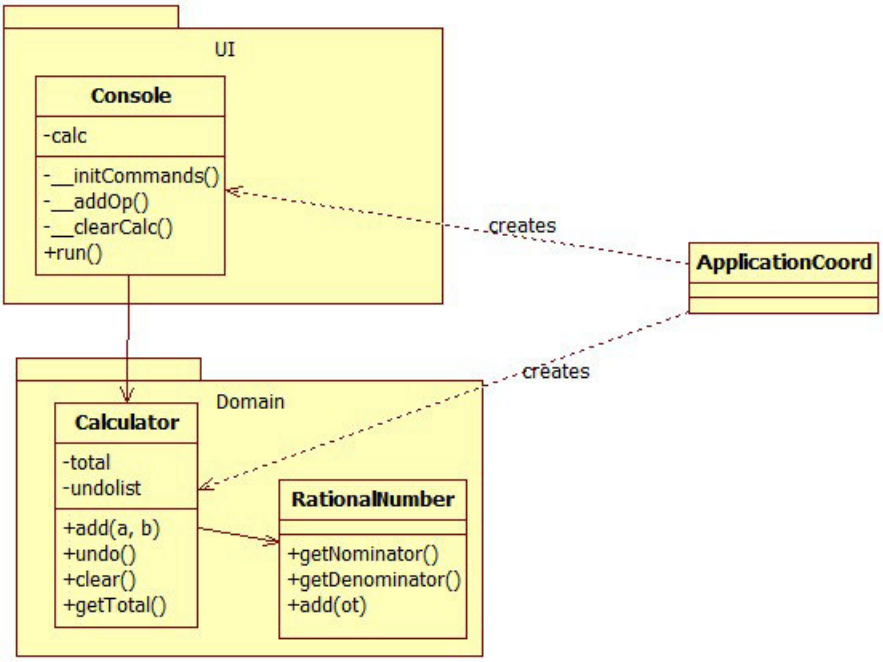
```
class Engine:
    def __init__(self,cap,type):
        """
          initialize the engine
          cap positive integer
          type string
        """
        self.__capacity = cap
        self.__type = type

    def getCapacity(self):
        """
       Getter method for the capacity
        """
        return self.__capacity
    def getType(self):
        """
         Getter method for type
         return string
        """
        return self.__type
```

Dependențe, Pachete

- Relația de dependență este o asociere în care un element depinde sau foloseste un alte element

Exemple de dependențe:

- crează instanțe
- are un parametru
- foloseste un obiect în interiorul unei metode



Principii de proiectare

Crează aplicații care:

Sunt ușor de înțeles, modificat, întreținut, testat

Clasele – abstracte, încapsulare, ascunderea reprezentării, ușor de testat, ușor de folosit și refolosit

Scop general: gestiunea dependențelor

- Single responsibility
- Separation of concerns
- Low Coupling
- High Cohesion

Enunț (Problem statement)
Scrieți un program care gestionează studenți de la o facultate (operații CRUD – **C**reate **R**ead **U**pdate **D**elete)

	Funcționalități (Features)
F1	Adauga student
F2	vizualizare studenți
F3	caută student
F4	șterge student

Plan de iterații

IT1 - F1; IT2 – F2; IT3 – F3; IT4 - F4

Scenariu de rulare (Running scenario)

user	app	description
‘a’		add a student
	give student id	
1		
	give name	
‘Ion’		
	new student added	
‘a’		add student
	give student id	
1		
	give name	
‘’		
	id already exists, name can not be empty	

Arhitectură stratificată (Layered architecture)

Layer (strat) este un mecanism de structurare logică a elementelor ce compun un sistem software
Într-o arhitectură multi-strat, straturile sunt folosite pentru a aloca responsabilități în aplicație.
Layer este un grup de clase (sau module) care au același set de dependențe cu alte module și se pot refolosi în circumstanțe similare.

- **User Interface Layer (View Layer, UI layer sau Presentation layer)**
- **Application Layer (Service Layer sau GRASP Controller Layer)**
- **Domain layer (Business Layer, Business logic Layer sau Model Layer)**
- **Infrastructure Layer (acces la date – modalităţi de persistenţă, logging, network I/O ex. Trimitere de email, sau alte servicii technice)**

Șabloane Grasp

General Responsibility Assignment Software Patterns (or **Principles**) conțin recomandări pentru alocarea responsabilităților pentru clase obiecte într-o aplicație orientat obiect.

- High Cohesion
- Low Coupling
- Information Expert
- Controller
- Protected Variations
- Creator
- Pure Fabrication

High Cohesion

Alocă responsabilitățile astfel încât coeziunea în sistem rămâne ridicată

High Cohesion este un principiu care se aplică pe pacursul dezvoltării în încercarea de a menține elementele în sistem:

- responsabile de un set de activități înrudite
- de dimensiuni gestionabile
- ușor de înțeles

Coeziune ridicată (High cohesion) înseamna ca responsabilitățile pentru un element din sistem sunt înrudite, concentrate în jurul aceluiași concept.

Înpărțirea programelor în clase și starturi este un exemplu de activitate care asigură coeziune ridicată în sistem.

Alternativ, coeziune slabă (low cohesion) este situația în care elementele au prea multe responsabilități, din arii diferite. Elementele cu coeziune slabă sunt mai greu de înțeles, reutilizat , întreținut și sunt o piedică pentru modificările necesare pe parcursul dezvoltării unui sistem

Low Coupling

Alocă responsabilități astfel încât cuplarea rămâne slabă (redusă)

Low Coupling încurajează alocarea de responsabilitați astfel încât avem:

- dependențe puține între clase;
- inpact scăzut în sistem la schimbarea unei clase;
- potențial ridicat de refolosire;

Forme de cuplare:

- TypeX are un câmp care este de TypeY.
- TypeX are o medodă care referă o instanță de tipul TypeY în orce formă (parameterii, variabile locale, valoare returnată, apel la metode)
- TypeX este derivat direct sau indirect din clasa TypeY.

Information Expert

Alocă responsabilitatea clasei care are toate informațiile necesare pentru a îndeplini sarcina

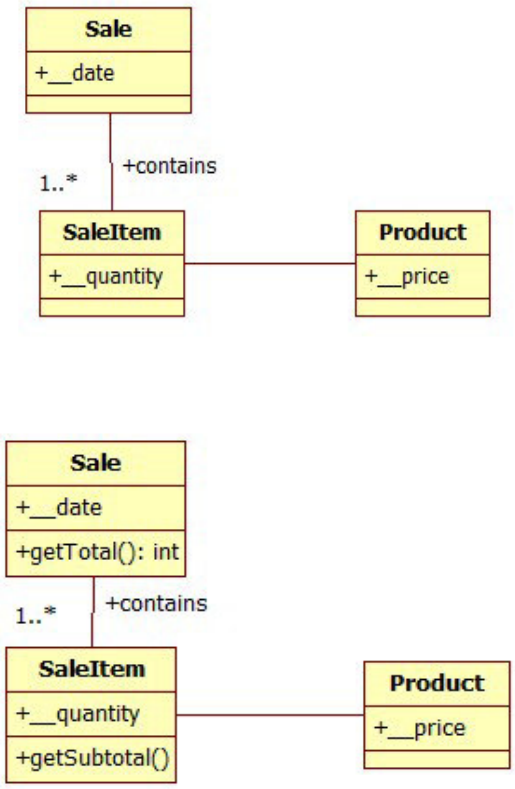
Information Expert este un principiu care ajută să determinăm care este clasa potrivită care ar trebui să primească responsabilitatea (o metodă noua, un câmp, un calcul).

Folosind principiu Information Expert încercăm sa determinăm care sunt informațiile necesare pentru a realiza ce se cere, determinăm locul în care sunt aceste informații și alocăm responsabilitatea la clasa care conține informațiile necesare.

Information Expert conduce la alocarea responsabilității în clasa care conține informația necesară pentru implementare. Ajută să răspundem la întrebarea Unde se pune – metoda, cămpul

Information Expert

Point of Sale application



Cine este responsabil cu calculul totalului?

Avem nevoie de toate SaleItems pentru a calcula totalul.

Information Expert → **Sale**

Conform Expert

SaleItem ar trebui sa fie responsabil cu calculul subtotalului (quantity * price)

1. Menține încapsularea
2. Promovează cuplare slabă
3. Promovează clase puternic coezive
4. Poate sa conducă la clase complexe - dezavantaj

Creator

Crearea de obiecte este o activitate importantă într-un sistem orientat obiect. Care este clasa responsabilă cu crearea de obiecte este o proprietate fundabentală care definește relația între obiecte de diferite tipuri.

Șablonul Creator descrie modul în care alocăm responsabilitatea de a crea obiecte în sistem

În general o clasa B ar trebui să aibă responsibilitatea de a crea obiecte de tip A dacă unul sau mai multe (de preferat) sunt adevărate:

- Instanța de tip B conține sau agregă instanțe de tip A
- Instanța de tip B gestionează instanțe de tip A
- Instanța de tip B folosește extensiv instanțe de tip A
- Instanța de tip B are informațiile necesare pentru a inițializa instanța A.

Work items

	Task
T1	Create Student
T2	Validate student
T3	Store student (Create repository)
T4	Add student (Create Controller)
T5	Create UI

Task: create Student

```
def testCreateStudent():
    """
      Testing student creation
    """
    st = Student("1", "Ion", "Adr")
    assert st.getId() == "1"
    assert st.getName() == "Ion"
    assert st.getAdr() == "Adr"
```

```
class Student:
    def __init__(self, id, name, adr):
        """
         Create a new student
         id, name, address String
        """
        self.id = id
        self.name = name
        self.adr = adr

    def getId(self):
        return self.id

    def getName(self):
        return self.name

    def getAdr(self):
        return self.adr
```

Protected Variations

Cum alocăm responsabilitatea astfel încât variațiile curente și viitoare nu vor afecta sistemul (nu va fi necesar o revizuire a sistemului, nu trebuie sa facem schimbări majore în sistem)?

Protected variations: Creăm o nouă clasă care încapsulează aceste variații.

Șablonul **Protected Variations** protejează elementele sistemului de variațiile/modificările altor elemente din sistem (clase, obiecte, subsisteme) încapsulând partea instabilă într-o clasă separată (cu o interfață publică bine delimitată care ulterior, folosind polimorfism, poate introduce variații prin noi implementări).

Task: Validate student

Design posibil pentru validare:

Algoritmul de validare:

- Poate fi o metoda in clasa student
- o metoda statica, o funcție
- încapsulat într-o clasă separată

Poate semnala eroarea prin:

- returnare true/false
- returnare lista de erori
- excepții care conțin lista de erori

Clasă Validator : aplică Principiu Protect Variation

```
def testStudentValidator():
    """
      Test validate functionality
    """
  validator = StudentValidator()
  st = Student("", "Ion", "str")
    try:
        validator.validate(st)
        assert False
    except ValueError:
        assert True
    st = Student("", "", "")
    try:
        validator.validate(st)
        assert False
    except ValueError:
        assert True
```

```
class StudentValidator:
    """
      Class responsible with validation
    """
    def validate(self, st):
        """
          Validate a student
          st - student
          raise ValueError
         if: Id, name or address is empty
        """
      errors = ""
      if (st.id==""):
         errors+="Id can not be empty;"
      if (st.name==""):
         errors+="Name can not be empty;"
      if (st.adr==""):
         errors+="Address can not be
empty"
      if len(errors)>0:
        raise ValueError(errors)
```

Pure Fabrication

Când un element din sistem încalcă primcipiul coeziunii ridicate și cuplare slabă (în general din cauza aplicării succesive a șablonului expert):
Alocă un set de responsabilități la o clasă artificială (clasă ce nu reprezintă ceva în domeniul problemei) pentru a oferi coeziune ridicată, cuplare slabă și reutilizare

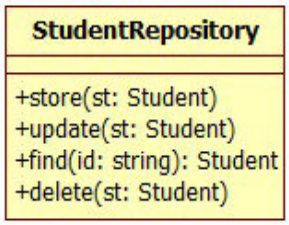
Pure Fabrication este o clasă ce nu reprezintă un concept din domeniul problemei este o clasă introdusă special pentru a menține cuplare slabă și coeziune ridicată în sistem.

Problema: Stocare **Student** (in memorie, fișier sau bază de date)
Expert pattern → Clasa Student este “expert”, are toate informațiile, pentru a realiza această operație

Pure Fabrication - Repository

Problema: Stocare **Student** (in memorie, fișier sau bază de date)
Expert pattern → Clasa Student este “expert”, are toate informațiile, pentru a realiza această operație
Dacă punem responsabilitatea persistenței in clasa Student, rezultă o clasă slab coeziva, cu potențial limitat de refolosire

Soluție – **Pure Fabrication**

	Clasă creată cu responsabilitatea de a salva/persista obiecte Student Clasa student se poate reutiliza cu ușurință are High cohesion, Low coupling Clasa StudentRepository este responsabil cu problema gestiunii unei liste de studenți (să ofere un depozit - persistent – pentru obiecte de tip student)

Șablonul Repository

Un **repository** reprezintă toate obiectele de un anumit tip ca si o mulțime de obiecte.
Obiecte sunt adăugate, șterse, modificate iar codul din repository insereaza, sterge obiectele dintr-un depozit de date persistent.

Task: Create repository

```
def testStoreStudent():
    st = Student("1", "Ion", "Adr")
    rep = InMemoryRepository()
    assert rep.size()==0
    rep.store(st)
    assert rep.size()==1
    st2 = Student("2", "Vasile", "Adr2")
    rep.store(st2)
    assert rep.size()==2
    st3 = Student("2", "Ana", "Adr3")
    try:
        rep.store(st3)
        assert False
    except ValueError:
        pass
```

```
class InMemoryRepository:
    """
     Manage the store/retrieval of students
    """
    def __init__(self):
        self.students = {}

    def store(self, st):
        """
          Store students
          st is a student
          raise RepositoryException if we have a student with the same id
        """
        if st.getId() in self.students:
            raise ValueError("A student with this id already exist")

        if (self.validator!=None):
            self.validator.validate(st)

        self.students[st.getId()] = st
```

GRASP Controller

Scop: decuplarea sursei de evenimente de obiectul care gestionează evenimentul. Decuplarea startului de prezentare de restul aplicației.

Controller este definit ca primul obiect după stratul de interfață utilizator. Interfața utilizator folosește un obiect controller, acest obiect este responsabil de efectuarea operațiilor cerute de utilizator.
Controller coordonează (controlează) operațiile necesare pentru a realiza acțiunea declanșată de utilizator.

Controlerul în general folosește alte obiecte pentru a realiza operația, doar coordonează activitatea.

Controllerul poate încapsula informații despre starea curentă a unui use-case. Are metode care corespund la o acțiune utilizator

Task: create controller

```
def tesCreateStudent():
    """
      Test store student
    """
    rep = InMemoryRepository()
    val = StudentValidator()
    ctr = StudentController(rep, val)
    st = ctr.createStudent("1", "Ion", "Adr")
    assert st.getId()=="1"
    assert st.getName()=="Ion"
    try:
        st = ctr.createStudent("1", "Vasile", "Adr")
        assert False
    except ValueError:
        pass
    try:
        st = ctr.createStudent("1", "", "")
        assert False
    except ValueError:
        pass
```

```
class StudentController:
    """
      Use case controller for CRUD Operations on student
    """
    def __init__(self, rep, validator):
        self.rep = rep
        self.validator = validator

    def createStudent(self, id, name, adr):
        """
          store a student
          id, name, address of the student as strings
          return the Student
          raise ValueError if a student with this id already exists
          raise ValueError if the student is invalid
        """
        st = Student(id, name, adr)
        if (self.validator!=None):
            self.validator.validate(st)
        self.rep.store(st)
        return st
```

Application coordinator

Dependency injection (DI) este un principiu de proiectare pentru sisteme orientat obiect care are ca scop reducerea cuplării între componentele sistemului.

De multe ori un obiect folosește (depinde de) rezultatele produse de alte obiecte, alte părți ale sistemului.

Folosind **DI**, obiectul nu are nevoie să cunoască modul în care alte părți ale sistemului sunt implementate/create. Aceste dependențe sunt oferite (sunt injectate), inpreună cu un contract (specificații) care descriu comportamentul componentei

```
#create validator
validator = StudentValidator()
#crate repository
rep = InMemoryRepository(None)
#create console provide(inject) a validator and a repository
ctr = StudentController(rep, validator)
#create console provide controller
ui = Console(ctr)
ui.showUI()
```

Review aplicația student manager – de revazut șabloanele ce apar

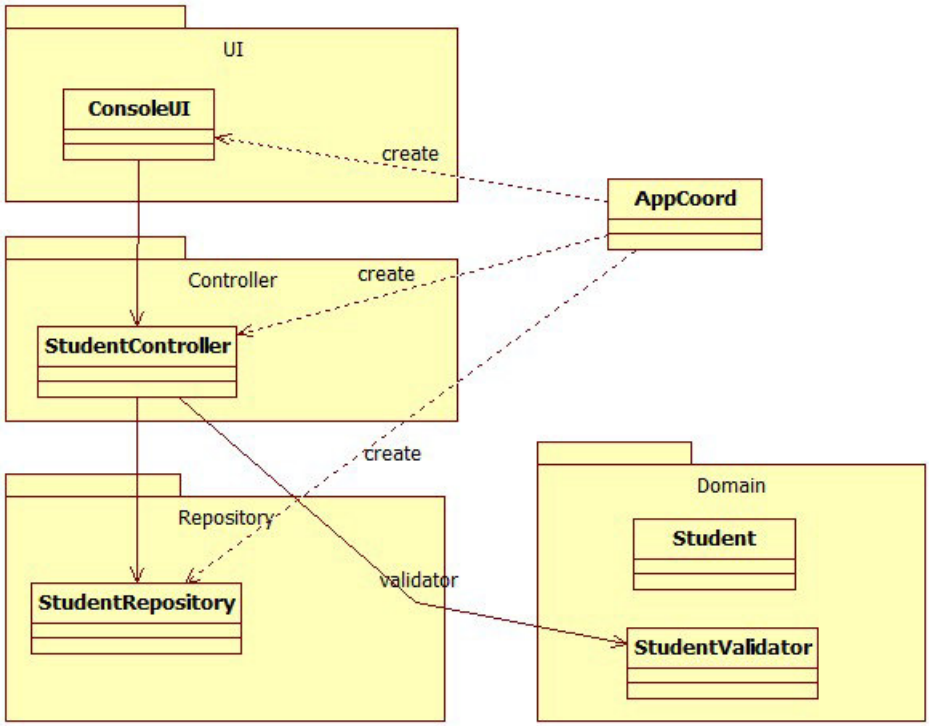
Arhitectură stratificată (Layered architecture)

Layer (strat) este un mecanism de structurare logică a elementelor ce compun un sistem software
Într-o arhitectură multi-strat, straturile sunt folosite pentru a aloca responsabilități în aplicație.
Layer este un grup de clase (sau module) care au același set de dependențe cu alte module și se pot refolosi în circumstanțe similare.

- **User Interface Layer (View Layer, UI layer sau Presentation layer)**
- **Application Layer (Service Layer sau GRASP Controller Layer)**
- **Domain layer (Business Layer, Business logic Layer sau Model Layer)**
- **Infrastructure Layer (acces la date – modalităţi de persistenţă, logging, network I/O ex. Trimitere de email, sau alte servicii technice)**

Aplicația StudentCRUD

Review aplicație



Entități

Entitate (Entity) este un obiect care este definit de identitatea lui (se identifică cu exact un obiect din lumea reală).

Principala caracteristică a acestor obiecte nu este valoarea atributelor, este faptul ca pe intreg existenta lor (in memorie, scris in fisier, incarcat, etc) se mentine identitatea si trebuie asigurat consistenta (sa nu existe mai multe entitati care descriu același obiect).

Pentru astfel de obiecte este foarte important sa se definească ce inseamnă a fi egale.

```
def testIdentity():
    #attributes may change
    st = Student("1", "Ion", "Adr")
    st2 = Student("1", "Ion", "Adr2")
    assert st==st2

    #is defined by its identity
    st = Student("1", "Popescu", "Adr")
    st2 = Student("2", "Popescu", "Adr2")
    assert st!=st2
```

```
class Student:
    def __init__(self, id, name, adr):
        """
         Create a new student
         id, name, address String
        """
        self.__id = id
        self.__name = name
        self.__adr = adr

    def __eq__(self,ot):
        """
          Define equal for students
          ot - student
          return True if ot and the current instance represent the same student
        """
        return self.__id==ot.__id
```

Atributele entității se poat schimba dar identitatea rămâbe același (pe întreg existența lui obiectul reprezintă același obiect din lumea reală)

O identitate greșită conduce la date invalide (data corruption) și la inposibilitatea de a implementa corect anumite operații.

Obiecte valoare (Value Objects)

Obiecte valoare: obiecte ce descriu caracteristicile unui obiect din lumea reala, conceptual ele nu au identitate.
Reprezintă aspecte descriptive din domeniu. Când ne preocupă doar atributele unui obiect (nu și identitatea) clasificam aceste obiecte ca fiind Obiecte Valoare (Value Object)

```
def testCreateStudent():
    """
      Testing student creation
      Feature 1 - add a student
      Task 1 - Create student
    """
    st = Student("1", "Ion", Address("Adr", 1, "Cluj"))
    assert st.getId() == "1"
    assert st.getName() == "Ion"
    assert st.getAdr().getStreet()=="Adr"

    st = Student("2", "Ion2", Address("Adr2", 1, "Cluj"))
    assert st.getId() == "2"
    assert st.getName() == "Ion2"
    assert st.getAdr().getStreet() == "Adr2"
    assert st.getAdr().getCity() == "Cluj"
```

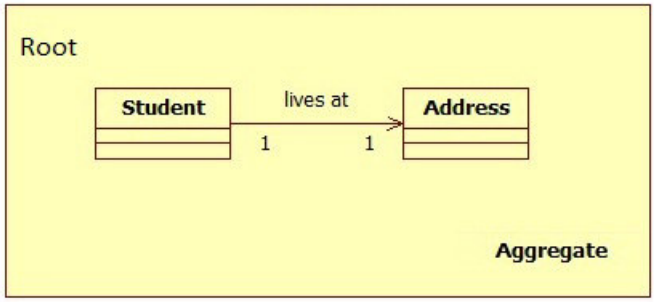
```
class Address:
    """
      Represent an address
    """
    def __init__(self,street,nr,city):
        self.__street = street
        self.__nr = nr
        self.__city = city

    def getStreet(self):
        return self.__street

    def getNr(self):
        return self.__nr

    def getCity(self):
        return self.__city
```

```
class Student:
    """
      Represent a student
    """
    def __init__(self, id, name, adr):
        """
         Create a new student
         id, name String
         address - Address
        """
        self.__id = id
        self.__name = name
        self.__adr = adr

def getId(self):
        """
          Getter method for id
        """
        return self.__id
```

Agregate și Repository

Grupați entități și obiecte valoare în agregate. Alegeți o entitate radăcină (root) care controlează accesul la toate elementele din agregat.

Obiectele din afara agregatului ar trebui să aibă referința doar la entitatea principală.



Repository – crează illuzia unei colecții de obiecte de același tip. Creați Repository doar pentru entitatea principală din agregat

Doar StudentRepository (nu și AddressRepository)

Fișiere text în Python

Funcția Built in: **open()** returneaza un obiect reprezentând fișierul

Cel mai frecvent se foloseste apelul cu două argumente: **open(filename,mode).**

Filename – un string, reprezintă calea către fișier(absolut sau relativ)

Mode:

"r" – open for read

"w" – open for write (overwrites the existing content)
"a" – open for append

Metode:
write(str) – scrie string în fișier
readline() - citire linie cu line, returnează string
read() - citește tot fișierul, returnează string
close() - închide fișier, eliberează resursele ocupate

Excepții:
IOError – aruncă această excepție daca apare o eroare de intrare/ieșire (no file, no disk space, etc)

Exemple Python cu fișiere text

```
#open file for write (overwrite if exists, create if not)
f = open("test.txt","w")
f.write("Test data\n")
f.close()
```

```
#open file for write (append if exist, create if not)
f = open("test.txt","a")
f.write("Test data line 2\n")
f.close()
```

```
#open for read
f = open("test.txt","r")
#read a line from the file
line = f.readline()
print line
f.close()
```

```
#open for read
f = open("test.txt","r")
#read a line from the file
line = f.readline().strip()
while line!="":
    print line
    line = f.readline().strip()
f.close()
```

```
#open for read
f = open("test.txt","r")
#read the entire content from the file
line = f.read()
print line
f.close()
```

```
#use a for loop
f  = open("etc/test.txt")
for line in f:
    print line
f.close()
```

Repository cu fișiere

```
class StudentFileRepository:
    """
      Store/retrieve students from file
    """
    def __loadFromFile(self):
        """
          Load students from file
        """
        try:
            f = open(self.__fName, "r")
        except IOError:
            #file not exist
            return []
        line = f.readline().strip()
        rez = []
        while line!="":
            attrs = line.split(";")
            st = Student(attrs[0], attrs[1], Address(attrs[2], attrs[3], attrs[4]))
            rez.append(st)
            line = f.readline().strip()
        f.close()
        return rez

    def store(self, st):
        """
          Store the student to the file.Overwrite store
          st - student
          Post: student is stored to the file
          raise DuplicatedIdException for duplicated id
        """
        allS = self.__loadFromFile()
        if st in allS:
            raise DuplicatedIDException()
        allS.append(st)
        self.__storeToFile(allS)

    def __storeToFile(self,sts):
        """
         Store all the students  in to the file
         raise CorruptedFileException if we can not store to the file
        """
        #open file (rewrite file)
        f = open(self.__fName, "w")
        for st in sts:
            strf = st.getId()+";"+st.getName()+";"
            strf = strf + st.getAdr().getStreet()+";"+str(st.getAdr().getNr())
+";"+st.getAdr().getCity()
            strf = strf+"\n"
            f.write(strf)
        f.close()
```

Dynamic Typing

Verificarea tipului se efectueaza în timpul execuției (runtime) – nu în timpul compilării (compile-time).

În general în limbajele cu dynamic typing valorile au tip, dar variabilele nu. Variabila poate referi o valoare de orice tip

Duck Typing

Duck typing este un stil de dynamic typing în care metodele și câmpurile obiectelor determină semantica validă, nu relația de moștenire de la o clasă anume sau implementarea unei interfețe.

Interfața publica este dată de multimea metodelor și câmpurilor accesibile din exterior. Două clase pot avea același interfața publică chiar dacă nu exista o relație de moștenire de la o clasă de bază comună

Duck test: When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck

```
class Student:
    def __init__(self, id, name):
        self.__name = name
        self.__id = id
    def getId(self):
        return self.__id

    def getName(self):
        return self.__name
```

```
class Professor:
    def __init__(self, id, name, course):
        self.__id = id
        self.__name = name
        self.__course = course

    def getId(self):
        return self.__id

    def getName(self):
        return self.__name

    def getCourse(self):
        return self.__course
```

```
l = [Student(1, "Ion"), Professor("1", "Popescu", "FP"), Student(31, "Ion2"),
Student(11, "Ion3"), Professor("2", "Popescu3", "asd")]

for el in l:
    print el.getName()+" id "+str(el.getId())

def myPrint(st):
    print el.getName(), " id ", el.getId()

for el in l:
    myPrint(el)
```

Duck typing – Repository

Fiindcă interfața publică a clasei:

- GradeRepository și GradeFileRepository
- StudentRepository și StudentFileRepository

sunt identice controllerul funcționeaza cu oricare obiect, fără modificări.

```
#create a validator
val = StudentValidator()
#create repository
repo = StudentFileRepository("students.txt")
#create controller and inject dependencies
ctr = StudentController(val, repo)
#create Grade controller
gradeRepo = GradeFileRepository("grades.txt")
ctrgr =  GradeController(gradeRepo, GradeValidator(), repo)
#create console ui and provide (inject) the controller
ui = ConsoleUI(ctr,ctrgr)
ui.startUI()
```

```
#create a validator
val = StudentValidator()
#create repository
repo = StudentRepository()
#create controller and inject dependencies
ctr = StudentController(val, repo)
#create Grade controller
gradeRepo = GradeRepository()
ctrgr =  GradeController(gradeRepo, GradeValidator(), repo)
#create console ui and provide (inject) the controller
ui = ConsoleUI(ctr,ctrgr)
ui.startUI()
```

Asocieri între obiecte din domeniu

În lumea reală, conceptual sunt multe relații de tip many-to-many dar modelarea acestor relații în aplicație nu este întodeauna fezabilă.

Când modelăn obiecte din lumea reală în aplicațiile noastre, asocierile complică implementarea și întreținerea aplicației.

- Asocierile bidirecționale de exemplu presupun ca fiecare obiect din asociere se poate folosi/înțelege/refolosi doar înpreună

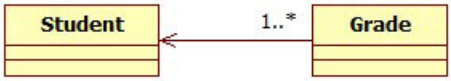
Este important să simplificăm aceste relații cât de mult posibil, prin:

- Impunerea unei direcții (transformare din bi-direcțional în unidirecțional)
- Reducerea mutiplicității
- Eliminarea asocierilor ne-esențiale

Scopul este sa modelăm lumea reală cât mai fidel dar în acelaşi timp sa simplificăm modelul pentru a nu complica implementare.

Asocieri

Exemplu Catalog

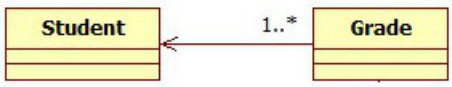


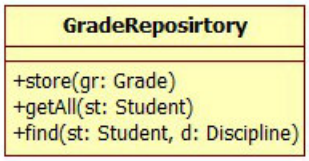

```
gr = ctr.assign("1", "FP", 10)
assert gr.getDiscipline()=="FP"
assert gr.getGrade()==10
assert gr.getStudent().getId()=="1"
assert gr.getStudent().getName()=="Ion"
```

```
st = Student("1", "Ion",
       Address("Adr", 1, "Cluj"))

rep = GradeRepository()
grades = rep.getAll(st)
assert grades[0].getStudent()==st
assert grades[0].getGrade()==10
```

Ascunderea detaliilor legate de persistență

Repository trebuie să ofere iluzia că obiectele sunt în memorie astfel codul client poate ignora detaliile de implementare.
În cazul în care repository salvează datele se în fișier, trebuie sa avem în vedere anumite aspecte.



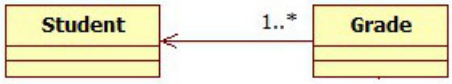


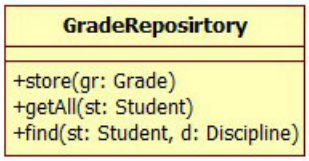
În exemplul de mai sus GradeRepository salvează doar id-ul studentului (nu toate campurile studentului) astfel nu se poate implementa o funcție getAll în care se returneaza toate notele pentru toți studenții. Se poate în scimb oferi metoda getAll(st) care returnează toate notele pentru un student dat

```
def store(self, gr):
        """
          Store a grade
          post: grade is in the repository
          raise GradeAlreadyAssigned exception if we already have a grade
                          for the student at the given discipline
          raise RepositoryException if there is an IO error when writing to
              the file
        """
        if self.find(gr.getStudent(), gr.getDiscipline())!=None:
            raise GradeAlreadyAssigned()

        #open the file for append
        try:
            f = open(self.__fname, "a")
            grStr = gr.getStudent().getId()+","+gr.getDiscipline()
            grStr =grStr+","+str(gr.getGrade())+"\n"
            f.write(grStr)
            f.close()
        except IOError:
            raise RepositorException("Unable to write a grade to the file")
```

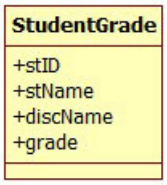
Obiecte de transfer (DTO - Data transfer objects)

Funcționalitate: Primi 5 studenți la o disciplină. Prezentați in format tabelar : nume student, nota la disciplina dată





Avem nevoie de obiecte speciale (obiecte de transfer) pentru acest caz de utilizare.
Funcțiile din repository nu ajung pentru a implementa (nu avem getAll()).
Se creaza o noua clasă care conține exact informațiile de care e nevoie.



În repository:

```
def getAllForDisc(self,disc):
        """
         Return all the grades for all the students from all disciplines
         disc - string, the discipline
         return list of StudentGrade's
        """
        try:
            f = open(self.__fname, "r")
        except IOError:
            #the file is not created yet
            return None
        try:
            rez = [] #StudentGrade instances
            line = f.readline().strip()
            while line!="":
                attrs = line.split(",")
                #if this line refers to the requested student
                if attrs[1]==disc:
                    gr = StudentGrade(attrs[0], attrs[1], float(attrs[2]))
                    rez.append(gr)
                line = f.readline().strip()
            f.close()
            return rez
        except IOError:
            raise RepositorException("Unable to read grades from the file")
```

DTO – Data transfer obiect

În controller:

```
def getTop5(self,disc):
        """
          Get the best 5 students at a given discipline
          disc - string, discipline
          return list of StudentGrade ordered descending on
the grade
        """
        sds = self.__grRep.getAllForDisc(disc)
        #order based on the grade
        sortedsds = sorted(sds, key=lambda studentGrade:
studentGrade.getGrade(),reverse=True)
        #retain only the first 5
        sortedsds = sortedsds[:5]
        #obtain the student names
        for sd in sortedsds:
            st = self.__stRep.find(sd.getStudentID())
            sd.setStudentName(st.getName())
        return sortedsds
```

Moștenire

Moștenirea permite definirea de clase noi (clase derivate) reutilizând clase existente (clasa de bază). Clasa nou creată moșteneste comportamentul (metode) și caracteristicile (variabile membre, starea) de la clasa de bază

Dacă A și B sunt două clase unde B moșteneste de la clasa A (B este derivat din clasa A sau clasa B este o specializare a clasei A) atunci:

- clasa B are toate metodele si variabilele membre din clasa A
- clasa B poate redefini metode din clasa A
- clasa B poate adauga noi membrii (variabile, metode) pe lângă cele moștenite de la clasa A.

Reutilizare de cod

Una din motivele pentru care folosim moștenire este reutilizarea codului existent într-o clasă (moștenire de implementare).

Comporamentul unei clase de baze se poate moșteni de clasele derivate.
Clasa dericvată poate:

- poate lăsa metoda nemodificată
- apela metoda din clasa de bază
- poate modifica (suprascrie) o metodă.

Moștenire în Python

Syntaxă:
class DerivedClassName(BaseClassName):

Clasa derivată moștenește:

- câmpuri
- metode

Dacă acessăm un membru (câmp, metodă) : se caută în clasa curentă, dacă nu se găsește atunci cautarea continuă în clasa de bază

```
class B(A):
    """
      This class extends A
      A is the base class,
      B is the derived class
      B is inheriting everything from class A
    """
    def __init__(self):
        #initialise the base class
        A.__init__(self)
        print "Initialise B"

    def g(self):
        """
          Overwrite method g from A
        """
        #we may invoke the function from the
base class
        A.f(self)
        print "in method g from B"
```

```
class A:
  def __init__(self):
    print "Initialise A"

  def f(self):
    print "in method f from A"

  def g(self):
    print "in method g from A"
```

```
b = B()
#f is inherited from A
b.f()
b.g()
```

Clasele Derivate pot suprascrie metodele clasei de baza.

Suprascrierea poate înlocui cu totul metoda din clasa de bază sau poate extinde funcționalitatea (se execută și metoda din clasa de bază dar se mai adaugă cod)

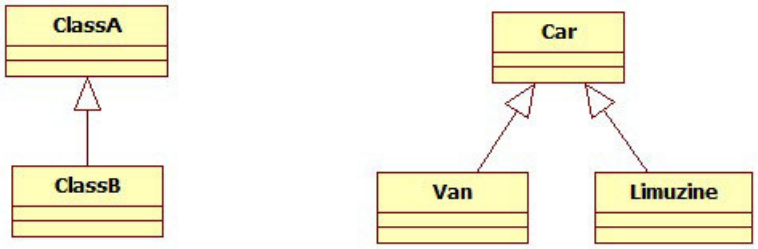
O metodă simplă să apelăm o metodă în clasa de bază:

BaseClassName.methodname (self,arguments)

Diagrame UML – Generalizare (moștenire)

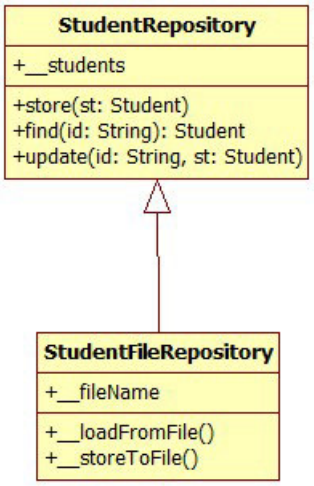
Relația de generalizare ("is a") indică faptul că o clasă (clasa derivată) este o specializare a altei clase (clasa de bază). Clasa de bază este generalizarea clasei derivate.

Orice instanță a clasei derivate este si o instanța a clasei de bază.



Repository cu Fișiere

```
class StudentFileRepository(StudentRepository):
    """
      Repository for students (stored in a file)
    """
   pass
```




```
class StudentFileRepository(StudentRepository):
    """
      Store/retrieve students from file
    """
    def __init__(self,fileN):
        #properly initialise the base class
        StudentRepository.__init__(self)
        self.__fName = fileN
        #load student from the file
        self.__loadFromFile()

    def __loadFromFile(self):
        """
          Load students from file
          raise ValueError if there is an error when reading from the file
        """
        try:
            f = open(self.__fName,"r")
        except IOError:
            #file not exist
            return
        line = f.readline().strip()
        while line!="":
          attrs = line.split(";")
          st = Student(attrs[0],attrs[1],Address(attrs[2], attrs[3], attrs[4]))
          StudentRepository.store(self, st)
          line = f.readline().strip()
        f.close()
```

Suprascriere metode

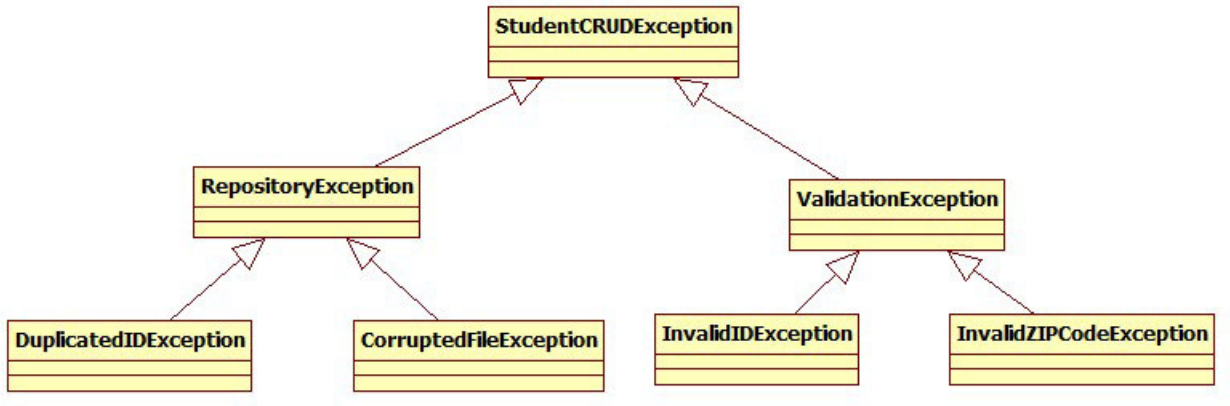
```
def testStore():
    fileName = "teststudent.txt"
    repo = StudentFileRepository(fileName)
    repo.removeAll()

    st = Student("1","Ion",Address("str",3,"Cluj"))
    repo.store(st)
    assert repo.size()==1
    assert repo.find("1") == st
    #verify if the student is stored in the file
    repo2 = StudentFileRepository(fileName)
    assert repo2.size()==1
    assert repo2.find("1") == st
```

```
def store(self,st):
        """
          Store the student to the file.Overwrite store
          st - student
          Post: student is stored to the file
          raise DuplicatedIdException for duplicated id
        """
        StudentRepository.store(self, st)
        self.__storeToFile()

def __storeToFile(self):
        """
         Store all the students  in to the file
         raise CorruptedFileException if we can not store to the file
        """
        f = open(self.__fName,"w")
        sts = StudentRepository.getAll(self)
        for st in sts:
            strf = st.getId()+";"+st.getName()+";"
            strf = strf + st.getAdr().getStreet()
+";"+str(st.getAdr().getNr())                    +";"+st.getAdr().getCity()
            strf = strf+"\n"
            f.write(strf)
        f.close()
```

Excepții

```
def __createdStudent(self):
        """
          Read a student and store in the apllication
        """
        id = input("Student id:").strip()
        name = input("Student name:").strip()
        street = input("Address - street:").strip()
        nr = input("Address - number:").strip()
        city = input("Address - city:").strip()
        try:
            self.__ctr.create(id, name,street,nr,city)
        except ValueError as msg:
            print (msg)


def __createdStudent(self):
        """
          Read a student and store in the apllication
        """
        id = input("Student id:").strip()
        name = input("Student name:").strip()
        street = input("Address - street:").strip()
        nr = input("Address - number:").strip()
        city = input("Address - city:").strip()
        try:
            self.__ctr.create(id, name,street,nr,city)
        except ValidationException as ex:
            print (ex)
        except DuplicatedIDException as ex:
            print (ex)
class ValidationException(Exception):
    def __init__(self,msgs):
        """
         Initialise
         msg is a list of strings (errors)
        """
        self.__msgs = msgs
    def getMsgs(self):
        return self.__msgs

    def __str__(self):
        return str(self.__msgs)
```

Ierarhie de excepții

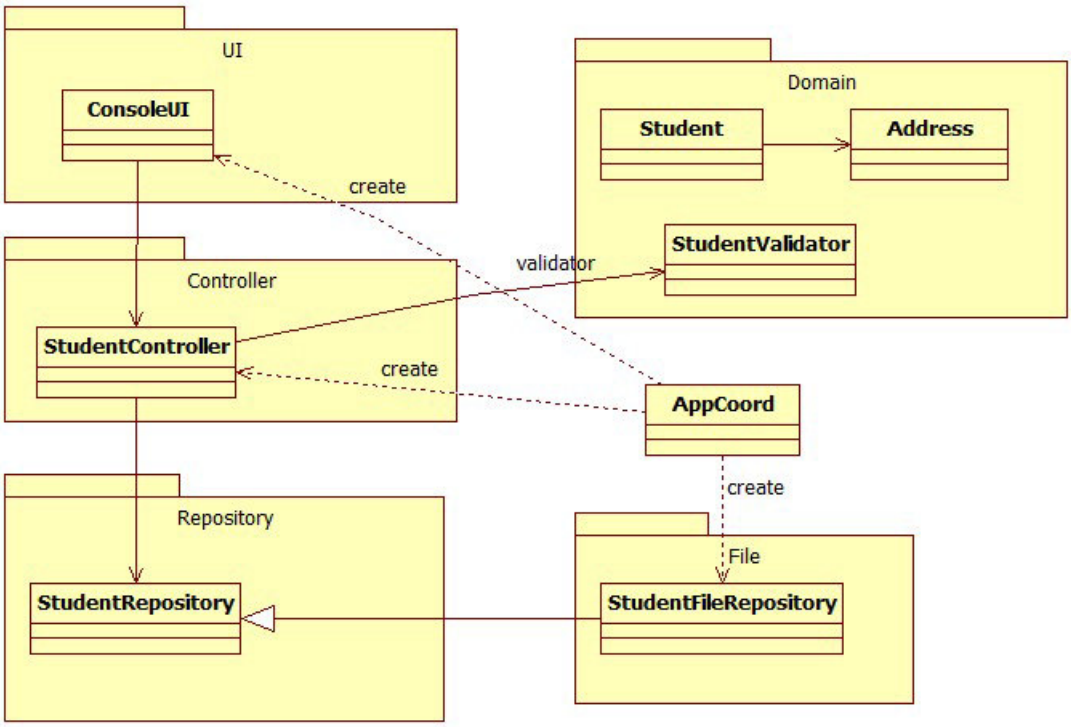



```
class StudentCRUDException(Exception):
    pass
```

```
class ValidationException(StudentCRUDException):
    def __init__(self,msgs):
        """
         Initialise
         msg is a list of strings (errors)
        """
        self.__msgs = msgs
    def getMsgs(self):
        return self.__msgs
    def __str__(self):
        return str(self.__msgs)

class RepositorException(StudentCRUDException):
    """
      Base class for the exceptions in the repository
    """
    def __init__(self, msg):
        self.__msg = msg
    def getMsg(self):
        return self.__msg
    def __str__(self):
        return self.__msg
class DuplicatedIDException(RepositorException):
    def __init__(self):
        RepositorException.__init__(self, "Duplicated ID")
```

```
def __createdStudent(self):
        """
          Read a student and store in the apllication
        """
        id = input("Student id:").strip()
        name = input("Student name:").strip()
        street = input("Address - street:").strip()
        nr = input("Address - number:").strip()
        city = input("Address - city:").strip()
        try:
            self.__ctr.create(id, name,street,nr,city)
        except StudentCRUDException as ex:
            print (ex)
```

Layered arhitecture – Structură proiect



Layered architecture – GUI Example

Tkinter este un toolkit GUI pentru Python (este disponibil pe majoritatea platformelor Unix , pe Windows și Mac)

Review - aplicația StudentCRUD cu GUI

Tkinter (sau orice alt GUI) nu se cere la examen

Testarea programelor

Testarea este observarea comportamentului unui program în multiple execuții.
Se execută programul pentru ceva date de intrare și se verifică daca rezultate sunt corecte în raport cu intrările.

Testarea nu demonstrează corectitudinea unui program (doar oferă o anumită siguranța , confidență). In general prin testare putem demonstra că un program nu este corect, găsind un exemplu de intrări pentru care rezultatele sunt greșite.

Testarea nu poate identifica toate erorile din program.

Metode de testare

Testare exhaustivă

Verificarea programului pentru toate posibilele intrări.
Imposibil de aplicat în practivă, avem nevoie de un număr finit de cazuri de testare.

Black box testing (metoda cutiei negre)

Datele de test se selecteaza analizând specificațiile (nu ne uităm la implementare).
Se verifică dacă programul respectă specificațiile.
Se aleg cazur de testare pentru: valori obișnuite, valori limite, condiții de eroare.

White box testing (metoa cutiei transparente)

Datele de test se aleg analizând coul sursă. Alegem datele astfel încât se acoperă toate ramurile de execuţie (în urma executări testelor, fiecare instrucţiune din program este executat măcar odată)

White box vs Black Box testing

```
def isPrime(nr):
    """
      Verify if a number is prime
      return True if nr is prime False if not
      raise ValueError if nr<=0
    """
    if nr<=0:
        raise ValueError("nr need to be positive")
    if nr==1:#1 is not a prime number
        return False
    if nr<=3:
        return True
    for i in range(2,nr):
        if nr%i==0:
            return False
    return True
```

Black Box

- test case pentru prim/compus
- test case pentr 0
- test case pentru numere negative

White Box (cover all the paths)

- test case pt 0
- test case pt negative
- test case pt 1
- test case pt 3
- test case pt prime (fără divizor)
- test case pt neprime

```
def blackBoxPrimeTest():
    assert (isPrime(5)==True)
    assert (isPrime(9)==False)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

```
def whiteBoxPrimeTest():
    assert (isPrime(1)==False)
    assert (isPrime(3)==True)
    assert (isPrime(11)==True)
    assert (isPrime(9)==True)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

Nivele de testare

Testele se pot categoriza în funcție de momentul în care se crează (în cadrul procesului de dezvoltare) sau în funcție de specificitatea testelor.

Unit testing

Se referă la testarea unei funcționalitați izolate, în general se referă la testarea la nivel de metode. Se testează funcțiile sau părți ale programului, independent de restul applicației

Integration testing

Consideră întreaga aplicație ca un întreg. După ce toate funcțiile au fost testate este nevoie de testarea comportamentului general al programului.

Testare automată (Automated testing)

Testare automată – presupune scrierea de programe care realizează testarea (în loc să se efectueze manual).
Practic se scrie cod care compara rezultatele efective pentru un set de intrări cu rezultatele așteptate.

TDD:

Pașii TDD:

- teste automate
- scrierea specificațiilor (inv, pre/post, excepții)
- implementarea codului

PyUnit - bibliotecă Python pentru unit testing

modulul **unittest** oferă:

- teste automate
- modalitate uniformă de pregatire/curațare (setup/shutdown) necesare pentru teste
  - fixture
- agregarea testelor
  - test suite
- independența testelor față de modalitatea de raportare

```
import unittest
class TestCaseStudentController(unittest.TestCase):
    def setUp(self):
        #code executed before every testMethod
        val=StudentValidator()
        self.ctr=StudentController(val, StudentRepository())
        st = self.ctr.create("1", "Ion", "Adr", 1, "Cluj")

    def tearDown(self):
        #cleanup code executed after every testMethod

    def testCreate(self):
        self.assertTrue(self.ctr.getNrStudents()==1)
        #test for an invalid student
        self.assertRaises(ValidationEx,self.ctr.create,"1", "", "", 1, "Cj")

        #test for duplicated id
        self.assertRaises(DuplicatedIDException,self.ctr.create,"1", "I",
                                                                "A", 1, "j")

    def testRemove(self):
        #test for an invalid id
        self.assertRaises(ValueError,self.ctr.remove,"2")

        self.assertTrue(self.ctr.getNrStudents()==1)

        st = self.ctr.remove("1")
        self.assertTrue(self.ctr.getNrStudents()==0)
        self.assertEquals(st.getId(),"1")
        self.assertTrue(st.getName()=="Ion")
        self.assertTrue(st.getAdr().getStreet()=="Adr")


if __name__ == '__main__':
    unittest.main()
```

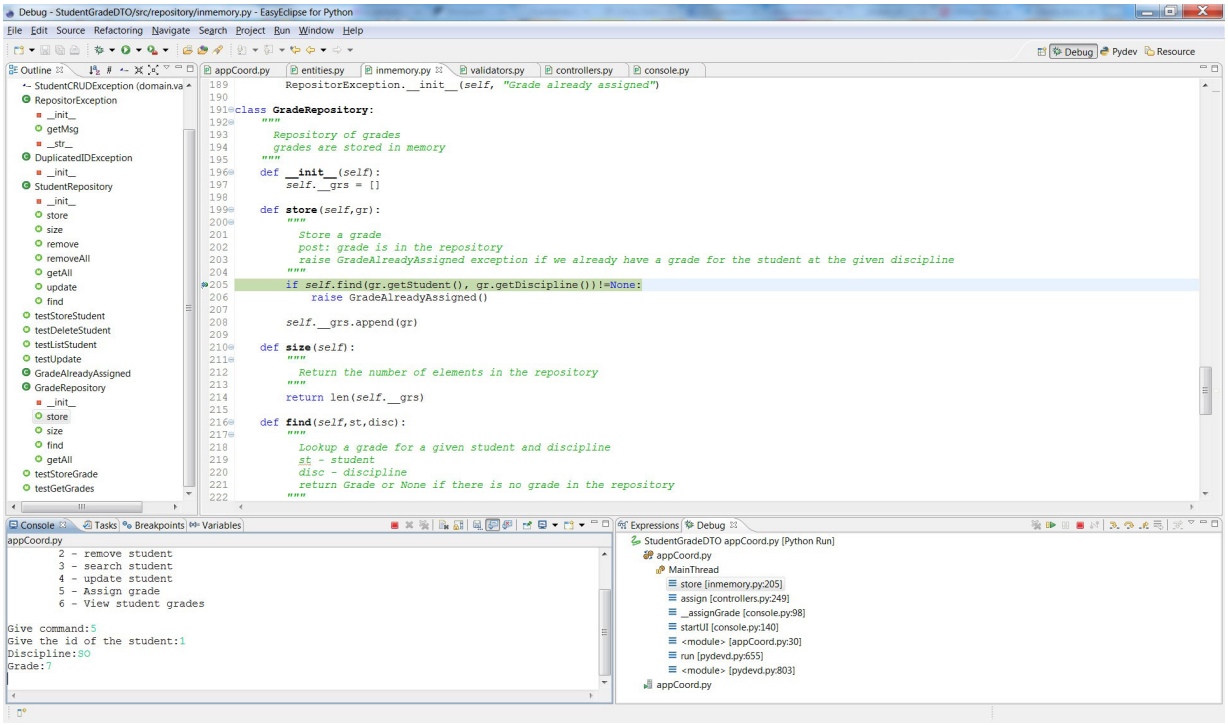
Depanare (Debugging)

DepanareaDebugging este activitatea prin care reparăm erorile găsite în urma testării. Dacă testarea indică prezența unei erori atunci prin depanare în cercăm să identificăm cauza erorii, modalități de rezolvare. Scopul este sa elminăm eroarea.
Se pate realiză folosind:

- instructiuni print
- instrumente specializate oferite de IDE

Depanarea este o activitate neplăcută, pe cât posibil, trebuie evitată**.**

Perspectiva Eclipse pentru depanare



Debug view

- prezintă starea curentă de execuție (stack trace)
- execuție pas cu pas, resume/pause

Variables view

- inspectarea variabilelor

Inspectarea programelor

Any fool can write code that a computer can understand. Good programmers write code that humans can understand

Prin stilul de programare înțelegem toate activitățile legate de scrierea de programe și modalitățile prin care obținem cod: ușor de citit, ușor de înțeles, ușor de intreținut.

Stil de programare

Principalul atribut al codului sursă este considerat ușurința de a citi (readability).
Un program, ca și orice publicație, este un text care trebuie citit și înțeles cu ușurință de orice programator.
Elementele stilului de programare sunt:

- comentarii
- formatarea textului (indentare, white spaces)
- specificații
- denumiri sugestive (pentru clase, funcții, variabile) din program
    - denumiri sugestive
    - folosirea convențiilor de nume

Convenții de nume (naming conventions):

- clase: Student, StudentRepository
- variabile: student, nrElem (nr_elem)
- funcții: getName, getAddress, storeStudent (get_name,get_address, store_student)
- constante: MAX

Este inportant sa folosiți același reguli de denumire in toată aplicația

Recursivitate

O noțiune e recursivă dacă e folosit în propria sa definiție.

O funcție recursivă: funcție care se auto-apelează.
Rezultatul este opținut apelând același funcție dar cu alți parametrii

```
def factorial(n):
    """
      compute the factorial
      n  is a positive integer
      return n!
    """
    if n== 0:
        return 1
    return  factorial(n-1)*n
```

- Recursivitate directă: P apelează P
- Recursivitate indirectă: P apelează Q, Q apelează P

Cum rezolvăm probleme folosind recursivitatea:

- Definim cazul de bază: soluția cea mai simplă.
  - Punctul în care problema devine trivială (unde se oprește apelul recursiv)
- Pas inductiv: înpărțim problema într-o variantă mai simplă al aceleași probleme plus ceva pași simplii
  - ex. apel cu n-1, sau doua apeluri recusive cu n/2

```
def recursiveSum(l):
    """
    Compute the sum of numbers
    l - list of number
    return int, the sum of numbers
    """
    #base case
    if l==[]:
        return 0
    #inductive step
    return l[0]+recursiveSum(l[1:])
```

```
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

Obs recursiveSum(l[1:]):

l[1:] - crează o copie a listei l

exercițiu: modificați funcția recursiveSum pentru a evita l[1:]

Recursivitate în python:

- la fiecare apel de metodă se crează o noua tabelă de simboluri (un nou namespace). Această tabelă conţine valorile pentru parametrii şi pentru variabilele locale
- tabela de simboluri este salvat pe stack, când apelul se termină tabela se elimină din stivă

```
def isPalindrome(str):
    """
      verify if a string is a palindrome
      str - string
      return True if the string is a palindrome False otherwise
    """
    dict = locals()
    print id(dict)
    print dict

    if len(str)==0 or len(str)==1:
        return True

    return str[0]==str[-1] and isPalindrome(str[1:-1])
```

Recursivitate

Avantaje:

- claritate
- cod mai simplu

Dezavantaje:

- consum de memorie mai mare
    - pentru fiecare recursie se crează o nouă tabelă de simboluri

Analiza complexității

Analiza complexități – studiul eficienței algoritmilor.

Eficienţa algoritmilor în raport cu:

- timpul de execuție – necesar pentru rularea programului
- spațiu necesar de memorie

Timp de execuție, depinde de:

- algoritmul folosit
- datele de intrare
- hardwareul folosit
- sistemul de operare (apar diferențe de la o rulare la alta).

Exemplu timp de execuție

```
def fibonacci(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

```
def fibonacci2(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    sum1 = 1
    sum2 = 1
    rez = 0
    for i in range(2, n+1):
        rez = sum1+sum2
        sum1 = sum2
        sum2 = rez
    return rez
```

```
def measureFibo(nr):
    sw = StopWatch()
    print "fibonacci2(", nr, ") =", fibonacci2(nr)
    print "fibonacci2 take " +str(sw.stop())+" seconds"

    sw = StopWatch()
    print "fibonacci(", nr, ") =", fibonacci(nr)
    print "fibonacci take " +str(sw.stop())+" seconds"


measureFibo(32)
```

```
fibonacci2( 32 ) = 3524578
fibonacci2 take 0.0 seconds
fibonacci( 32 ) = 3524578
fibonacci take 1.7610001564 seconds
```

Eficiența algoritmilor

- Eficienţa algoritmilor poate fi definită ca fiind cantitatea de resurse utilizate de algoritm (timp, memorie).

Măsurarea eficienţei:

- analiză matematică a algoritmului - **analiză asimptotică**
  Descrie eficienţa sub forma unei funcţii matematice.
  Estimeaza timpul de execuţie pentru toate intrările pisibile.
- o analiză **empirică** a algoritmului
  determinarea timpului exact de execuţie pentru date specifice
  nu putem prezice timpul pentru toate datele de intrare.

Timpul de execuţie pentru un algoritm este studiat în relaţie cu dimensiunea datelor de intrare.

- Estimăm timpul de execuţie în funcţie de dimensiunea datelor.
- Realizăm o **analiză *asymptotică***. Determinăm ordinul de mărime pentru resursa utilizată (timp, memorie), ne interesează în special pentru cazurile în care datele de inrare sunt mari

Complexitate

- **caz favorabil -** datele de intrare care conduc la timp de execuţie minim
  - *best-case complexity* (BC): $BC(A) = \min_{I \in D} E(I)$
- **caz defavorabil –** date de intrare unde avem cel mai mare timp de execuţie.
  - *worst-case complexity* (WC): $WC(A) = \max_{I \in D} E(I)$
- **caz mediu -** timp de execuție.
  - *average complexity* (AC): $AC(A) = \sum_{I \in D} P(I)E(I)$

A - algoritm; *E(I)* număr de operații; *P(I)* probabilitatea de a avea *I* ca și date de intrare

D – multimea tutoror datelor de intrare posibile pentru un n fixat

Obs. Dimensiunea datelor (n) este fixat (**un numar mare**) caz favorabil/caz defavorabil se referă la un **anumit aranjament al datelor** de intrare care produc timp minim/maxim

Complexitate timp de execuție

- **numărăm pași** (operații elementare) efectuați (de exemplu numărul de instrucțiuni, număr de comparații, număr de adunări).
- numărul de pași nu este un număr fixat, **este o funcție**, notat $T(n)$, este in funcție de dimensiunea datelor (n), nu rezultă timpul exact de execuție
- Se surprinde doar esențialul: cum crește timpul de execuție în funcție de dimensiunea datelor. Ne oferă **ordinea de mărime** pentru timpul de execuție (dacă $n \to \infty$, then $3 \cdot n^2 \approx n^2$).
- putem **ignora constante mici** – dacă $n \to \infty$ aceste constante nu afectează ordinea de mărime.

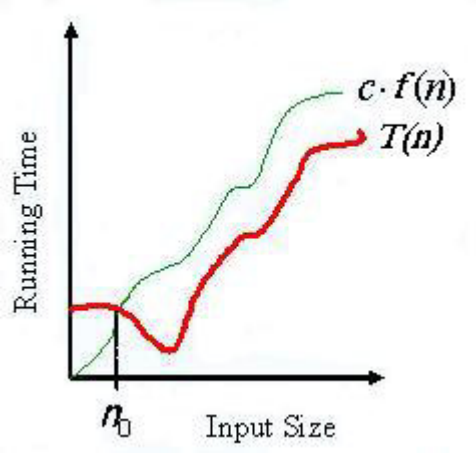
Ex : $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$

Fiindcă $0 < \log_2 n < n, \ \forall n > 1$ și $\sqrt{n} < n, \ \forall n > 1$, putem conclude că temenul n^3 domină această expresie cand n este mare

Ca urmare, timpul de execuţie a algoritmului creşte cu ordinul lui n^3, ceea se scrie sub forma $T(n) \in O(n^3)$ şi se citeşte "T(n) este de ordinul n^3

În continuare, vom nota prin *f* o funcţie $f : N \to \Re$ şi prin *T* funcţia care dă complexitatea timp de excuţie a unui algoritm, $T : N \to N$.

<u>Definiţia 1</u> (Notaţia O, "Big-oh"). Spunem că $T(n) \in O(f(n))$ dacă există **c** şi $\mathbf{n_0}$ constante pozitive (care nu depind de *n*) astfel încât $0 \le T(n) \le c \cdot f(n), \qquad \forall n \ge n_0.$



Cu alte cuvinte, notaţia *O* dă marginea superioară

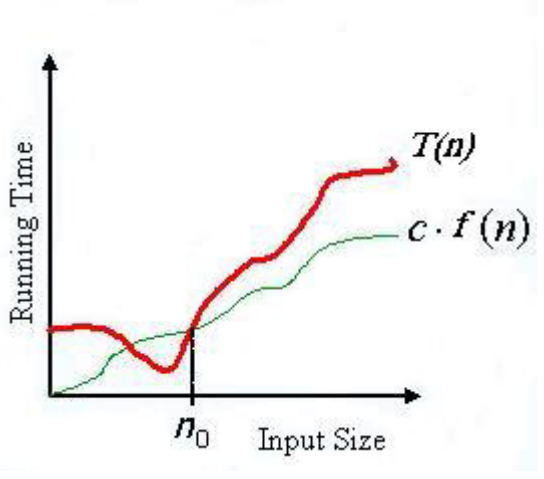
Definiţia alternativă: Spunem că $T(n) \in O(f(n))$ dacă $\lim_{n\to\infty} \frac{T(n)}{f(n)}$ este 0 sau o constantă, dar **nu** ∞.

Observaţii.

1. Dacă $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$, atunci $\lim_{n\to\infty} \frac{T(n)}{n^3} = 13$. Deci, putem spune că $T(n) \in O(n^3)$.

2. Notaţia *O* este bună pentru a da o limită superioară unei funcţii. Observăm, totuşi, că dacă $T(n) \in O(n^3)$, atunci este şi $O(n^4)$, $O(n^5)$, etc atâta timp cât limita este 0. Din această cauză avem nevoie de o notaţie pentru limita inferioară a complexităţii. Această notaţie este Ω.

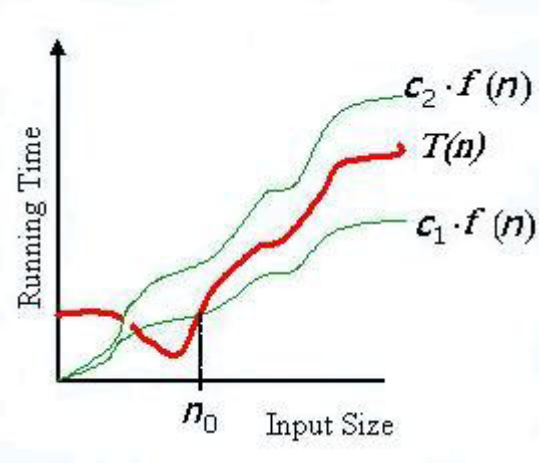
<u>Definiţia 2</u> (Notaţia Ω, "Big-omega"). Spunem că $T(n) \in \Omega(f(n))$ dacă există **c** şi $\mathbf{n_0}$ constante pozitive (care nu depind de *n*) astfel încât $0 \le c \cdot f(n) \le T(n), \quad \forall n \ge n_0$.



notaţia Ω dă marginea inferioară

Definiţia alternativă: Spunem că $T(n) \in \Omega(f(n))$ dacă $\lim_{n\to\infty} \frac{T(n)}{f(n)}$ este o constantă sau ∞, dar **<u>nu</u>** 0.

<u>Definiţia 3</u> (Notaţia θ, "Big-theta"). Spunem că $T(n) \in \theta(f(n))$ dacă $T(n) \in O(f(n))$ şi dacă $T(n) \in \Omega(f(n))$, altfel spus dacă există **c1, c2** şi $\mathbf{n_0}$ constante pozitive (care nu depind de n) astfel încât $c1 \cdot f(n) \le T(n) \le c2 \cdot f(n), \quad \forall n \ge n_0$.



notaţia θ mărgineşte o funcţie până la factori constanţi

Definiţia alternativă Spunem că $T(n) \in \theta(f(n))$ dacă $\lim_{n\to\infty} \frac{T(n)}{f(n)}$ este o constantă nenulă (dar **<u>nu</u>** 0 sau ∞).

Observaţii.

1. Timpul de execuţie al unui algoritm este $\theta(f(n))$ dacă şi numai dacă timpul său de execuţie în cazul cel mai defavorabil este $O(f(n))$ şi timpul său de execuţie în cazul cel mai favorabil este $\Omega(f(n))$.
2. Notaţia $O(f(n))$ este de cele mai multe ori folosită în locul notaţiei $\theta(f(n))$.
3. Dacă $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$, atunci $\lim_{n\to\infty} \frac{T(n)}{n^3} = 13$. Deci, $T(n) \in \theta(n^3)$. Acest lucru poate fi dedus şi din faptul că $T(n) \in O(n^3)$ şi $T(n) \in \Omega(n^3)$.

Sume

for i *in range(0, n):*

#some instructions

presupunând că ceea ce este în corpul structurii repetitive (*) se execută în *f(i)* paşi $\Rightarrow$ timpul de execuţie al întregii structuri repetitive poate fi estimat astfel

$$T(n) = \sum_{i=1}^{n} f(i)$$

Se poate observa că, în cazul în care se folosesc bucle imbricate, vor rezulta sume imbricate. În continuare, vom prezenta câteva dintre sumele uzuale:

Calculul se efectueaza astfel:

- se simplifică sumele – eliminăm constantele, separăm termenii in sume individuale
- facem calculul pentru sumele simplificate.

Exemple cu sume

Analizați complexitatea ca timp de execuție pentru următoarele funcții

```
def f1(n):
    s = 0
    for i in range(1,n+1):
        s=s+i
    return s
```

$T(n)=\sum_{(i=1)}^{n} 1=n \rightarrow T(n)\in\Theta(n)$

Complexitate (Overall complexity) $\Theta(n)$

Cazurile Favorabil/Mediu/Defavorabil sunt identice

```
def f2(n):
    i = 0
    while i<=n:
        #atomic operation
        i = i + 1
```

$T(n)=\sum_{(i=1)}^{n} 1=n \rightarrow T(n)\in\Theta(n)$

Overall complexity $\Theta(n)$

Cazurile Favorabil/Mediu/Defavorabil sunt identice

```
def f3(l):
    """
    l - list of numbers
    return True if the list contains
an even nr
    """
    poz = 0
    while poz<len(l) and l[poz]%2 !=0:
        poz = poz+1
    return poz<len(l)
```

Caz favorabil:

primul element e număr par: $T(n)=1\in\Theta(1)$

Caz defavorabil:

Nu avem numere pare în listă: $T(n)=n\in\Theta(n)$

Caz mediu:

While poate fi executat 1,2,..n ori (același probabilitate).

Numărul de pași = numărul mediu de iterații

$T(n)=(1+2+...+n)/n=(n+1)/2 \rightarrow T(n)\in\Theta(n)$

Complexitate $O(n)$

Exemple cu sume

```
def f4(n):
    for i in range(1,2*n-2):
        for j in range(i+2,2*n):
            #some computation
            pass
```

$$T(n)=\sum_{(i=1)}^{(2n-2)}\sum_{(j=i+2)}^{2n}1=\sum_{(i=1)}^{(2n-2)}(2n-i-1)$$

$$T(n)=\sum_{(i=1)}^{(2n-2)}2n-\sum_{(i=1)}^{(2n-2)}i-\sum_{(i=1)}^{(2n-2)}1$$

$$T(n)=2n\sum_{(i=1)}^{(2n-2)}1-(2n-2)(2n-1)/2-(2n-2)$$

...

$T(n)=2n^2-3n+1\in\Theta(n^2)$ Overall complexity $\Theta(n^2)$

```
def f5():
    for i in range(1,2*n-2):
        j = i+1
        cond = True
        while j<2*n and cond:
            #elementary operation
            if someCond:
                cond = False
```

Caz favorabil: While se execută odată

$$T(n)=\sum_{(i=1)}^{(2n-2)}1=2n-2\in\Theta(n)$$

Caz defavorabil: While executat $2n-(i+1)$ ori

$$T(n)=\sum_{(i=1)}^{(2n-2)}(2n-i-1)=...=2n^2-3n+1\in\Theta(n^2)$$

Caz mediu:

Pentru un i fixat While poate fi executat 1,2..2n-i-1 ori

număr mediu de pași:

$$C_i=(1+2+...+2n-i-1)/2n-i-1=...=(2n-i)/2$$

$$T(n)=\sum_{(i=1)}^{(2n-2)}C_i=\sum_{(i=1)}^{(2n-2)}(2n-i)/2=...\in\Theta(n^2)$$

Overall complexity $O(n^2)$

Formule cu sume:

$$\sum_{i=1}^{n} 1 = n$$ suma constantă.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$ suma liniară (progresia aritmetică)

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{2}$$ suma pătratică

$$\sum_{i=1}^{n} \frac{1}{i} = \ln(n) + O(1)$$ suma armonică

$$\sum_{i=1}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}, \quad c \neq 1$$ progresia geometrică (crește exponențial)

Complexităţi uzuale

$T(n) \in O(1)$ - **timp *constant*.** It is a great complexity. This means that the algorithm takes only constant time.

$T(n) \in O(\log_2 \log_2 n)$ - timp foarte rapid (aproape la fel de rapid ca un timp constant)

$T(n) \in O(\log_2 n)$ - complexitate *logaritmică*:
timp foarte bun (este ceea ce căutăm, în general, pentru orice algoritm);
$\log_2 1000 \approx 10, \quad \log_2 1.000.000 \approx 20$;
complexitate căutare binară, înălțimea unei arbore binar echilibrat

$T(n) \in O((\log_2 n)^k)$ - unde *k* este factor constant; se numeşte complexitate *polilogaritmică* (este destul de bună);

Complexităţi uzuale

$T(n) \in O(n)$ - complexitate *liniară;*

$T(n) \in O(n \cdot \log_2 n)$ - este o complexitate faimoasă, întâlnită mai ales la sortări (MergeSort, QuickSort);

$T(n) \in O(n^2)$ - *este complexitate pătratică (cuadratică);*
dacă n este de ordinul milioanelor, nu este prea bună;

$T(n) \in O(n^k)$ - *unde k este constant; este complexitatea polinomială (este practică dora daca k nu este prea mare);*

$T(n) \in O(2^n), O(n^3), O(n!)$ - complexitate *exponenţială* (algoritmii cu astfel de complexităţi sunt practici doar pentru valori mici ale lui *n*: $n \le 10, n \le 20$).

Recurențe

O **recurență** este o formulă matematică definită recursiv.

Ex. numărul de noduri (notat *N(h)*) dintr-un arbore ternar complet de înălţime *h* ar putea fi descris sub forma următoarei formule de recurenţă:

$$\begin{cases} N(0) = 1 \\ N(h) = 3 \cdot N(h-1) + 1, \qquad h \geq 1 \end{cases}$$

Explicaţia ar fi următoarea:

- Numărul de noduri dintr-un arbore ternar complet de înălţime 0 este 1.
- Numărul de noduri dintr-un arbore ternar complet de înălţime *h* se obţine ca fiind de 3 ori numărul de noduri din subarborele de înălţime *h-1*, la care se mai adaugă un nod (rădăcina arborelui).

Dacă ar fi să rezolvăm recurenţa, am obţine că numărul de noduri din arborele ternar complet de înălţime *h* este $N(h) = 3^h \cdot N(0) + (1 + 3^1 + 3^2 + ... + 3^{h-1}) = \sum_{i=0}^{h} 3^i$

Exemple

```
def recursiveSum(l):
    """
    Compute the sum of numbers
    l - list of number
    return int, the sum of numbers
    """
    #base case
    if l==[]:
        return 0
    #inductive step
    return l[0]+recursiveSum(l[1:])
```

Recurrence: $T(n)=\begin{cases}1 \text{ for } n=0\\ T(n-1)+1 \text{ otherwise}\end{cases}$

$$\begin{aligned}T(n)&=T(n-1)+1\\ T(n-1)&=T(n-2)+1\\ T(n-2)&=T(n-3)+1\\ &...=...\\ T(1)&=T(0)+1\end{aligned} \quad => \quad T(n)=n+1\in\Theta(n)$$

```
def hanoi(n, x, y, z):
    """
       n -number of disk on the x
stick
       x - source stick
       y - destination stick
       z - intermediate stick
    """
    if n==1:
      print "disk 1 from",x,"to",y
      return
    hanoi(n-1, x, z, y)
    print "disk ",n,"from",x,"to",y
    hanoi(n-1, z, y, x)
```

Recurrence: $T(n)=\begin{cases}1 \text{ for } n=1\\ 2T(n-1)+1 \text{ otherwise}\end{cases}$

$$\begin{aligned}T(n)&=2T(n-1)+1\\ T(n-1)&=2T(n-2)+1=\\ T(n-2)&=2T(n-3)+1\\ &...=...\\ T(1)&=T(0)+1\end{aligned} \quad => \quad \begin{aligned}T(n)&=2T(n-1)+1\\ 2T(n-1)&=2^2T(n-2)+2\\ 2^2T(n-2)&=2^3T(n-3)+2^2\\ &...=...\\ 2^{(n-2)}T(2)&=2^{(n-1)}T(1)+2^{(n-2)}\end{aligned}$$

$$T(n)=2^{(n-1)}+1+2+2^2+2^3+...+2^{(n-2)}$$

$$T(n)=2^n-1\in\Theta(2^n)$$

Complexitatea spaţiu de memorare

Complexitatea unui algoritm din punct de vedere al *spaţiului de memorare* estimează cantitatea de memorie necesară algoritmului pentru stocarea datelor de intrare, a rezultatelor finale şi a rezultatelor intermediare. Se estimează, ca şi *timpul de execuţie* al unui algoritm, în notaţiile O, Θ, Ω .

Toate obsevaţiile referitoare la notaţia asimptotică a complexităţii ca timp de execuţie sunt valabile şi pentru complexitatea ca spaţiu de memorare.

Exemplu

Analizați complexitatea ca spațiu de memorare pentru următoarele funcții

`def iterativeSum(l):`
`    """`
`    Compute the sum of numbers`
`    l - list of number`
`    return int, the sum of numbers`
`    """`
`    rez = 0`
`    for nr in l:`
`        rez = rez+nr`
`    return rez`
	Avem nevoie de spațiu pentru numerele din listă
	$T(n)=n \in \Theta(n)$

`def recursiveSum(l):`
`    """`
`    Compute the sum of numbers`
`    l - list of number`
`    return int, the sum of numbers`
`    """`
`    #base case`
`    if l==[]:`
`        return 0`
`    #inductive step`
`    return l[0]+recursiveSum(l[1:])`
	Recurență: $T(n)=\begin{cases} 0 \text{ for } n=1 \\ T(n-1)+1 \text{ otherwise} \end{cases}$

Analza complexității (timp/spațiu) pentru o funcție

1 Dacă există caz favorabil/defavorabil:

- descrie **Caz favorabil**
- calculează complexitatea pentru **Best Case**
- descrie **Worst Case**
- calculează complexitatea pentru **Worst case**
- calculează complexitatea **medie**
- calculează complexitatea **generală**

2 Dacă Favorabil = Defavorabil = Mediu – (nu avem cazuri favorabile/defavorabile)

- calculează complexitatea

Calculează complexitatea:

- dacă avem recurență
  - calculează folosind egalități
- altfel
  - calculează folosind sume

Algoritmi de căutare

- datele sunt în memorie, o *secvență de înregistrări* ($k_1, k_2, ..., k_n$)
- se caută o înregistrare având un câmp egal cu o valoare dată – *cheia de căutare*.
- Dacă am găsit înregistrarea, se returnează poziția înregistrării în secvență
- dacă cheile sunt ordonate atunci ne interesează poziția în care trebuie inserată o înregistare nouă astfel încât ordinea se menține

Specificații pentru căutare:

Date: a,n,(k_i, i=0,n-1);

Precondiții: n $\in$N, n$\geq$0;

Rezultate: p;

Post-condiții: (0$\leq$p$\leq$n-1 and a = k_p) or (p=-1 dacă cheia nu există).

Căutare secvențială – cheile nu sunt ordonate

```
def searchSeq(el,l):
    """
      Search for an element in a list
      el - element
      l - list of elements
      return the position of the element
         or -1 if the element is not in l
    """
    poz = -1
    for i in range(0,len(l)):
        if el==l[i]:
            poz = i
    return poz
```

$T(n)=\sum_{(i=0)}^{(n-1)} 1=n\in\Theta(n)$

```
def searchSucc(el,l):
    """
      Search for an element in a list
      el - element
      l - list of elements
      return the position of first occurrence
            or -1 if the element is not in l
    """
    i = 0
    while i<len(l) and el!=l[i]:
        i=i+1
    if i<len(l):
        return i
    return -1
```

Best case: the element is at the first position
$T(n)\in\theta(1)$
Worst-case: the element is in the n-1 position
$T(n)\in\theta(n)$
Average case: while can be executed 0,1,2,n-1 times
$T(n)=(1+2+...+n-1)/n\in\Theta(n)$
Overall complexity $O(n)$

Specificații pentru căutare – chei ordonate:

Date $a, n, (k_i, i=0,n-1)$;

Precondiții: $n \in N, n \geq 0$, *and* $k_0 < k_1 < .... < k_{n-1}$;

Rezultate p;

Post-condiții: $(p=0$ *and* $a \leq k_0)$ *or* $(p=n$ *and* $a > k_{n-1})$ *or*

$((0<p \leq n-1)$ *and* $(k_{p-1} < a \leq k_p))$.

Căutare secvențială – chei ordonate

```
def searchSeq(el,l):
    """
      Search for an element in a list
      el - element
      l - list of ordered elements
      return the position of first occurrence
             or the position where the element
             can be inserted
    """
    if len(l)==0:
        return 0
    poz = -1
    for i in range(0,len(l)):
        if el<=l[i]:
            poz = i
    if poz==-1:
        return len(l)
    return poz
```

```
def searchSucc(el,l):
    """
      Search for an element in a list
      el - element
      l - list of ordered elements
      return the position of first occurrence
             or the position where the element
             can be inserted
    """
    if len(l)==0:
        return 0
    if el<=l[0]:
        return 0
    if el>=l[len(l)-1]:
        return len(l)
    i = 0
    while i<len(l) and el>l[i]:
        i=i+1
    return i
```

$T(n)=\sum_{(i=0)}^{(n-1)} 1=n\in\Theta(n)$

Best case: the element is at the first position
$T(n)\in\theta(1)$
Worst-case: the element is in the n-1 position
$T(n)\in\theta(n)$
Average case: while can be executed 0,1,2,n-1 times
$T(n)=(1+2+...+n-1)/n\in\Theta(n)$
Overall complexity $O(n)$

Algoritmi de căutare

- *căutare secvențială*
  - se examinează succesiv toate cheile
  - cheile nu sunt ordonate
- *căutare binară*
  - folosește “divide and conquer”
  - cheile sunt ordonate

Căutare binară (recursiv)

```
def binaryS(el, l, left, right):
    """
      Search an element in a list
      el - element to be searched
      l - a list of ordered elements
      left,right the sublist in which we search
      return the position of first occurrence or the insert position
    """
    if left>=right-1:
        return right
    m = (left+right)/2
    if el<=l[m]:
        return binaryS(el, l, left, m)
    else:
        return binaryS(el, l, m, right)

def searchBinaryRec(el, l):
    """
      Search an element in a list
      el - element to be searched
      l - a list of ordered elements
      return the position of first occurrence or the insert position
    """
    if len(l)==0:
        return 0
    if el<l[0]:
        return 0
    if el>l[len(l)-1]:
        return len(l)
    return binaryS(el, l, 0, len(l))
```

Recurență căutare binară

$$T(n) = \begin{cases} \theta(1), & if\ n = 1 \\ T\left(\frac{n}{2}\right) + \theta(1), & otherwise \end{cases}$$

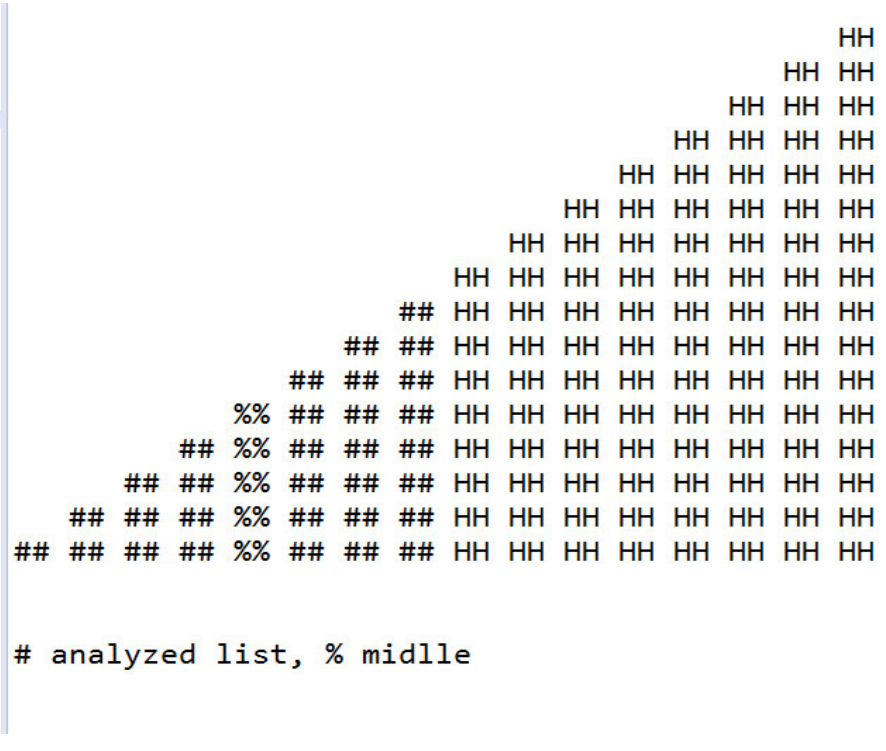
Căutare binară (iterativ)

```
def searchBinaryNonRec(el, l):
    """
      Search an element in a list
      el - element to be searched
      l - a list of ordered elements
      return the position of first occurrence or the position where the element can be
inserted
    """
    if len(l)==0:
        return 0
    if el<=l[0]:
        return 0
    if el>=l[len(l)-1]:
        return len(l)
    right=len(l)
    left = 0
    while right-left>1:
        m = (left+right)/2
        if el<=l[m]:
            right=m
        else:
            left=m
    return right
```

Complexitate

Algoritm	Timp de execuție			
	best case	worst case	average	overall
SearchSeq	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
SearchSucc	$\theta(1)$	$\theta(n)$	$\theta(n)$	$O(n)$
SearchBin	$\theta(1)$	$\theta(\log_2 n)$	$\theta(\log_2 n)$	$O(\log_2 n)$

Vizualizare cautări



Căutare in python - index()

```
l = range(1,10)
try:
    poz = l.index(11)
except ValueError:
    # element is not in the list
```

- __eq__

```
class MyClass:
    def __init__(self,id,name):
        self.id = id
        self.name = name

    def __eq__(self,ot):
        return self.id == ot.id


def testIndex():
    l = []
    for i in range(0,200):
        ob = MyClass(i,"ad")
        l.append(ob)

    findObj = MyClass(32,"ad")
    print "positions:" +str(l.index(findObj))
```

Searching in python- “in”

```
l = range(1,10)
found = 4 in l
```

– iterable (definiți __iter__ and __next__)

```
class MyClass2:
    def __init__(self):
        self.l = []

    def add(self,obj):
        self.l.append(obj)

    def __iter__(self):
        """
          Return an iterator object
        """
        self.iterPoz = 0
        return self
```

```
def __next__(self):
        """
          Return the next element in the iteration
     raise StopIteration exception if we are at the end
        """
        if (self.iterPoz>=len(self.l)):
            raise StopIteration()

        rez = self.l[self.iterPoz]
        self.iterPoz = self.iterPoz +1
        return rez

def testIn():
    container = MyClass2()
    for i in range(0,200):
        container.add(MyClass(i,"ad"))
    findObj = MyClass(20,"asdasd")
    print findObj in container
```

```
#we can use any iterable in a for
    container = MyClass2()
    for el in container:
        print (el)
```

Performanţă - căutare

```
def measureBinary(e, l):
    sw = StopWatch()
    poz = searchBinaryRec(e, l)
    print ("    BinaryRec in %f sec; poz=%i" %(sw.stop(),poz))

def measurePythonIndex(e, l):
    sw = StopWatch()
    poz = -2
    try:
        poz = l.index(e)
    except ValueError:
        pass #we ignore the error..
    print ("    PythIndex in %f sec; poz=%i" %(sw.stop(),poz))

def measureSearchSeq(e, l):
    sw = StopWatch()
    poz = searchSeq(e, l)
    print ("    searchSeq in %f sec; poz=%i" %(sw.stop(),poz))
```

```
search 200
    BinaryRec in 0.000000 sec; poz=200
    PythIndex in 0.000000 sec; poz=200
     PythonIn in 0.000000 sec
    BinaryNon in 0.000000 sec; poz=200
    searchSuc in 0.000000 sec; poz=200
```

```
search 10000000
    BinaryRec in 0.000000 sec; poz=10000000
    PythIndex in 0.234000 sec; poz=10000000
     PythonIn in 0.238000 sec
    BinaryNon in 0.000000 sec; poz=10000000
    searchSuc in 2.050000 sec; poz=10000000
```

Sortare

Rearanjarea datelor dintr-o colecţie astfel încât o cheie verifică o relaţie de ordine dată

- *internal sorting* - datele sunt în memorie
- *external sorting* - datele sunt în fişier

Elementele colecţiei sunt *înregistrări,* o înregistrare are una sau mai multe câmpuri
Cheia K este asociată cu fiecare înregistrare, în general este un câmp.

Colecţia este sortat:

- *crescător* după cheia *K :* if $K(i) \leq K(j)$ for $0 \leq i < j < n$
- *descrescător: if* $K(i) \geq K(j)$ *for* $0 \leq i < j < n$

Sortare internă – în memorie

Date n,K; $\{K=(k_1,k_2,\ldots,k_n)\}$

Precondiții: $k_i \in R$, $i=1,n$

Rezultate K';

Post-condiții: K' e permutare al lui K, având elementele sortate,

$$k'_1 \leq k'_2 \leq \ldots \leq k'_n.$$

Sortare prin selecție (Selection Sort)

- se determină elementul având cea mai mica cheie, interschimbare elementul cu elementul de pe prima poziție
- reluat procedura penru restul de elemente până când toate elementele au fost considerate.

Sortare prin selecție

```
def selectionSort(l):
    """
       sort the element of the list
       l - list of element
       return the ordered list (l[0]<l[1]<...)
    """
    for i in range(0,len(l)-1):
        ind = i
        #find the smallest element in the rest of the list
        for j in range(i+1,len(l)):
            if (l[j]<l[ind]):
                ind =j
        if (i<ind):
            #interchange
            aux = l[i]
            l[i] = l[ind]
            l[ind] = aux
```

Complexitate – timp de execuție

Numărul total de comparații este:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n \cdot (n-1)}{2} \in \theta(n^2)$$

Este independent de datele de intrare:

- caz favorabil/defavorabil/mediu sunt la fel, complexitatea este $\theta(n^2)$ **.**

Complexitate – spațiu de memorie

Sortare prin selecție – este un algoritm **In-place**:

memoria adițională (alta decăt memoria necesară pentru datele de intrare) este $\theta(1)$

- *In-place* . Algoritmul care nu folosțe memorie adițională (doar un mic factor constant).
- O*ut-of-place* sau *not-in-space*. Algoritmul folosește memorie adițională pentru sortare.

Selection sort is an *in-place* sorting algorithm.

Sortare prin selecție directă *(*Direct selection sort)

```
def directSelectionSort(l):
    """
       sort the element of the list
       l - list of element
       return the ordered list (l[0]<l[1]<...)
    """
    for i in range(0,len(l)-1):
        #select the smallest element
        for j in range(i+1,len(l)):
            if l[j]<l[i]:
                swap(l,i,j)
```

Overall time complexity: $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}1=\frac{n\cdot(n-1)}{2}\in\theta(n^2)$

Sortare prin inserție - Insertion Sort

- se parcurg elementele
- se inserează elementul curent pe poziția corectă în sub-secvența deja sortată.
- În sub-secvența ce conține elementele deja sortate se țin elementele sortate pe tot parcursul algoritmului, astfel după ce parcurgem toate elementele secvența este sortată în întregime

Sortare prin inserție

```
def insertSort(l):
    """
       sort the element of the list
       l - list of element
       return the ordered list (l[0]<l[1]<...)
    """
    for i in range(1,len(l)):
        ind = i-1
        a = l[i]
        #insert a in the right position
        while ind>=0 and a<l[ind]:
            l[ind+1] = l[ind]
            ind = ind-1
        l[ind+1] = a
```

Insertion Sort - complexitate – timp de execuție

Caz defavorabil: $T(n) = \sum_{i=2}^{n} (i-1) = \frac{n \cdot (n-1)}{2} \in \theta(n^2)$

Avem numărul maxim de iterații când lista este ordonat descrescător

Caz mediu: $\frac{n^2 + 3 \cdot n}{4} - \sum_{i=1}^{n} \frac{1}{i} \in \theta(n^2)$

Pentru un i fixat și un k, $1 \le k \le i$, probabilitatea ca x_i să fie al k-lea cel mai mare element în subsecvența $x_1, x_2, ..., x_i$ este $\frac{1}{i}$

Astfel, pentru *i* fixat, putem deduce:

Numarul de iterații **while**	Probabilitatea sa avem numărul de iterații **while** din prima coloană	caz
1	$\frac{1}{i}$	un caz în care **while** se execută odată: $x_i < x_{i-1}$
2	$\frac{1}{i}$	un caz în care while se execută de doă ori: $x_i < x_{i-2}$
...	$\frac{1}{i}$	**...**
i-1	$\frac{2}{i}$	un caz în care while se execută de **i-1** ori: $x_i < x_1$ and $x_1 \leq x_i < x_2$

Rezultă că numărul de iterații **while** medii pentru un i fixat este:

$$c_i = 1 \cdot \frac{1}{i} + 2 \cdot \frac{1}{i} + ... + (i-2) \cdot \frac{1}{i} + (i-1) \cdot \frac{2}{i} = \frac{i+1}{2} - \frac{1}{i}$$

Caz favorabil: $T(n) = \sum_{i=2}^{n} 1 = n - 1 \in \theta(n)$

lista este sortată

Sortare prin inserție

〉 complexitate generală este $O(n^2)$.

Complexitate – spațiu de meorie

complexitate memorie aditională este: $\theta(1)$.

〉 *Sortare prin inserție* este un algoritm *in-place*.

Sortare prin selecție directă

```
def directSelectionSort(l):
    """
       sort the element of the list
       l - list of element
       return the ordered list (l[0]<l[1]<...)
    """
    for i in range(0,len(l)-1):
        #select the smallest element
        for j in range(i+1,len(l)):
            if l[j]<l[i]:
                swap(l,i,j)
```

Overall time complexity: $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}1=\frac{n\cdot(n-1)}{2}\in\theta(n^2)$

Sortare prin inserție - Insertion Sort

- se parcurg elementele
- se inserează elementul curent pe poziția corectă în sub-secvența deja sortată.
- În sub-secvența ce conține elementele deja sortate se țin elementele sortate pe tot parcursul algoritmului, astfel după ce parcurgem toate elementele secvența este sortată în întregime

Sortare prin inserție

```
def insertSort(l):
    """
       sort the element of the list
       l - list of element
       return the ordered list (l[0]<l[1]<...)
    """
    for i in range(1,len(l)):
        ind = i-1
        a = l[i]
        #insert a in the right position
        while ind>=0 and a<l[ind]:
            l[ind+1] = l[ind]
            ind = ind-1
        l[ind+1] = a
```

Metoda bulelor - Bubble sort

- Compară elemente consecutive, dacă nu sunt în ordinea dorită, se interschibă.
- Procesul de comparare continuă până când nu mai avem elemente consecutive ce trebuie interschimbate (toate perechile respectă relația de ordine dată).

Sortare prin metoda bulelor

```
def bubbleSort(l):
    sorted = False
    while not sorted:
        sorted = True  #assume the list is already sorted
        for i in range(1,len(l)):
            if l[i-1]>l[i]:
                swap(l, i, i-1)
                sorted = False  #the list is not sorted yet
```

Complexitate metoda bulelor

Caz favorabil: $\theta(n)$. Lista este sortată

Caz defavorabil: $\theta(n^2)$. Lista este sortată descrescător

Caz mediu $\theta(n^2)$.

Coplexitate generală este $O(n^2)$

Complexitate ca spațiu adițional de memorie este $\theta(1)$.

〉 este un algoritm de sortare ***in-place*** .

QuickSort

Bazat pe “divide and conquer”

- **Divide**: se partiționează lista în 2 astfel încât elementele din dreapta pivotului sunt mai mici decăt elementele din stânga pivotului.
- **Conquer**: se sortează cele două subliste
- **Combine**: trivial – dacă partitionarea se face în același listă

Partiționare: re-aranjarea elementelor astfel încât elementul numit *pivot* ocupă locul final în secvență. Dacă poziția pivotului este i :

$$k_j \leq k_i \leq k_l \text{ , for } Left \leq j < i < l \leq Right$$

Quick-Sort

```
def partition(l,left,right):
    """
    Split the values:
             smaller pivot greater
    return pivot position
    post: left we have < pivot
          right we have > pivot
    """
    pivot = l[left]
    i = left
    j = right
    while i!=j:
        while l[j]>=pivot and i<j:
            j = j-1
        l[i] = l[j]
        while l[i]<=pivot and i<j:
            i = i+1
        l[j] = l[i]
    l[i] = pivot
    return i
```

```
def quickSortRec(l,left,right):

    #partition the list
    pos = partition(l, left, right)

    #order the left part
    if left<pos-1:
           quickSortRec(l, left, pos-1)
    #order the right part
    if pos+1<right:
            quickSortRec(l, pos+1, right)
```

QuickSort – complexitate timp de execuție

Timpul de execuție depinde de distribuția partiționării (căte elemente sunt mai mici decăt pivotul căte sunt mai mari)

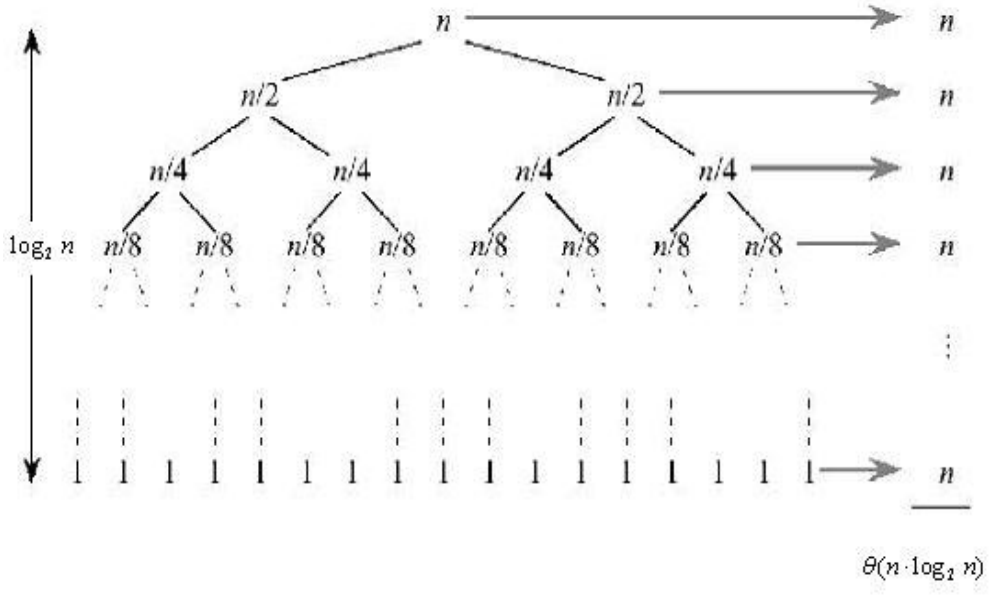
Partiționarea necesită timp linear.

Caz favorabil:, partițioanarea exact la mijloc (numere mai mici ca pivotul = cu numere mai mari ca pivotul):

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \theta(n)$$

Complexitatea este $\theta(n \cdot \log_2 n)$.

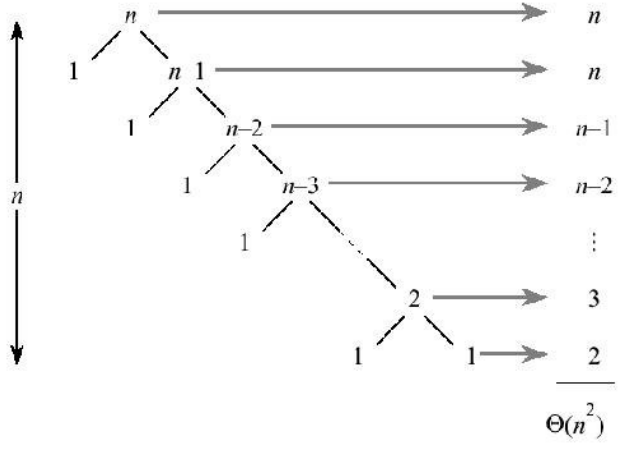
QuickSort – Caz favorabil



QuickSort – Caz defavorabil

Partiţionarea tot timpul rezultă într-o partiţie cu un singur element şi o partiţie cu n-1 elemente

$$T(n) = T(1) + T(n-1) + \theta(n) = T(n-1) + \theta(n) = \sum_{k=1}^{n} \theta(k) \in \theta(n^2) .$$



caz defavorabil: dacă elementele sunt în ordine inversă

QuickSort – Caz mediu

Se alternează cazurile:

- caz favorabil (lucky) cu complecitatea $\theta(n \cdot \log_2 n)$ (notăm cu L)
- caz defavorabil (unlucky) cu complexitatea $\theta(n^2)$ (notăm cu U).

Avem recurenţa:

$$\begin{cases} L(n) = 2 \cdot U\left(\frac{n}{2}\right) + \theta(n) & \textit{lucky case} \\ U(n) = L(n-1) + \theta(n) & \textit{unlucky case} \end{cases}$$

Rezultă

$$L(n) = 2 \cdot \left(L\left(\frac{n}{2} - 1\right) + \theta\left(\frac{n}{2}\right) \right) + \theta(n) = 2 \cdot L\left(\frac{n}{2} - 1\right) + \theta(n) = \theta(n \cdot \log_2 n),$$

Complexitatea caz mediu: $T(n) = L(n) \in \theta(n \cdot \log_2 n)$.

Coplexitatea ca timp de execuție pentru sortări:

Algorithm	Complexity	
	worst-case	average
SelectionSort	$\theta(n^2)$	$\theta(n^2)$
InsertionSort	$\theta(n^2)$	$\theta(n^2)$
BubbleSort	$\theta(n^2)$	$\theta(n^2)$
QuickSort	$\theta(n^2)$	$\theta(n \cdot \log_2 n)$

Python - Quick-Sort

```
def qsort(list):
    """
    Quicksort using list comprehensions

    """
    if list == []:
        return []
    else:
        pivot = list[0]
        lesser = qsort([x for x in list[1:] if x < pivot])
        greater = qsort([x for x in list[1:] if x >= pivot])
        return lesser + [pivot] + greater
```

List comprehensions – generatoare de liste

```
[x for x in list[1:] if x < pivot]
```

```
rez = []
for x in l[1:]:
    if x<pivot:
        rez.append(x)
```

- Variantă concisă de a crea liste
- crează liste unde elementele listei rezultă din operații asupra unor elemente dintr-o altă secvență
- paranteze drepte conținând o expresie urmată de o clauză for , apoi zero sau mai multe clauze for sau if

Python – Parametrii optionali parametrii cu nume

〉 Putem avea parametrii cu valori default;

```
def f(a=7,b = [],c="adsdsa"):
```

〉 Dacă se apelează metoda fără parametru actual se vor folosi valorile default

```
def f(a=7,b = [],c="adsdsa"):
    print (a)
    print (b)
    print (c)

f()
```

```
Console:
7
[]
adsdsa
```

〉 Argumentele se pot specifica în orice ordine

```
f(b=[1,2],c="abc",a=20)
```

```
Console:
20
[1, 2]
abc
```

〉 Parametrii formali se adaugă într-un dicționar (namespace)

〉 Trebuie oferit o valoare actuală pentru fiecare parametru formal - prin orice metodă: standard, prin nume, default value

Tipuri de parametrii – cum specificăm parametru actual

positional-or-keyword : parametru poate fi transmis prin poziție sau prin nume (keyword)

```
def func(foo, bar=None):
```

keyword-only : parametru poate fi transmis doar specificând numele

```
def func(arg, *, kw_only1, kw_only2):
```

Tot ce apare dupa * se poate transmite doar prin nume

var-positional : se pot transmite un număr arbitrar de parametri poziționali

```
def func(*args):
```

Valorile transmise se pot accesa folosind args, args este un tuplu

***var-keyword*:** un număr arbitrar de parametrii prin nume

```
def func(**args):
```

Valorile transmise se pot accesa folosind args, args este un dictionar

Sortare în python - list.sort() / funcție build in : sorted

sort(*, *key=None*,*reverse=None*)

Sortează folosind operatorul <. Lista curentă este sortată (nu se crează o altă listă)

key – o funcție cu un argument care calculeaza o valoare pentru fiecare element, ordonarea se face dupa valoarea cheii. În loc de o1 < o2 se face key(o1) < key(o2)

reverse – true daca vrem sa sortăm descrescător

```
l.sort()
print (l)
```

```
l.sort(reverse=True)
print (l)
```

sorted(*iterable[, key][, reverse]*)

Returnează lista sortată

```
l = sorted([1,7,3,2,5,4])
print (l)

l = sorted([1,7,3,2,5,4],reverse=True)
print (l)
```

```
def keyF(o1):
    return o1.name

ls = sorted(l,keyF)
```

Sort stability

Stabil (stable) – dacă avem mai multe elemente cu acelşi cheie, se menţine ordinea iniţială

Python – funcţii lambda (anonymous functions, lambda form)

〉 Folosind `lambda` putem crea mici funcţii

```
lambda x:x+7
```

〉 Funcţiile lambda pot fi folosite oriunde e nevoie de un obiect funcţie

```
def f(x):
    return x+7

print ( f(5) )
```

```
print ( (lambda x:x+7)(5) )
```

〉 Putem avea doar o expresie.

〉 Sunt o metodă convenientă de a crea funcţii mici.

〉 Similar cu funcțiile definite în interiorul altor funcții, funcțiile lambda pot referi variabile din namespace

```
l = []
l.append(MyClass(2,"a"))
l.append(MyClass(7,"d"))
l.append(MyClass(1,"c"))
l.append(MyClass(6,"b"))
#sort on name
ls = sorted(l,key=lambda o:o.name)
for x in ls:
    print (x)
```

```
l = []
l.append(MyClass(2,"a"))
l.append(MyClass(7,"d"))
l.append(MyClass(1,"c"))
l.append(MyClass(6,"b"))
#sort on id
ls = sorted(l,key=lambda o:o.id)
for x in ls:
    print (x)
```

TreeSort

Algoritmul crează un arbore binar cu proprietatea că la orice nod din arbore, arborele stâng conține doar elemente mai mici decât elementul din rădăcină iar arborele drept conține doar elemente mai mari decât rădăcina.

Dacâ parcurgem arborele putem lua elementele în ordine crescătoare/descrescătoare.

Arborele este construit incremental prin inserarea succesivă de elemente. Elementele se inserează astfel încât se menține proprietatea ca în stânga avem doar elemente mai mici în dreapta doar elemente mai mari decât elementul din rădăcină.

Elementul nou inserat tot timpul ajunge într-un nod terminal (frunză) în arbore.

MergeSort

Bazat pe “divide and conquer”.

Secvența este înpărțită în două subsecvențe și fiecare subsecvența este sortată. Dupa sortare se interclasează cele două subsecvențe, astfel rezultă secvența sortată în întregime.

Pentru subsecvențe se aplică același abordare până când ajungem la o subsecvență elementară care se poate sorta fără înpărțire (secvență cu un singur element).

Interclasare (Merging)

Date m, (x_i, i=1,m), n, (y_i, i=1,n);

Precondiții: {x_1 <= x_2...<=x_m} și {y_1 <= y_2 <=... <=y_n}

Rezultate k, (z_i, i=1,k);

Post-condiții: {k=m+n} și {z_1<=z_2<=...<=z_k} și (z_1,z_2,..., z_k) este o permutare a valorilor (x_1, ..., x_m,y_1,..., y_n)

complexitate interclasare: $\theta(m+n)$.

Spațiu de memorare adițională pentru merge sort $\theta(1)$

Technici de programare

- strategii de rezolvare a problemelor mai dificile
- algoritmi generali pentru rezolvarea unor tipuri de probleme
- de multe ori o problemă se poate rezolva cu mai multe technici – se alege metoda mai eficientă
- problema trebuie să satisfacă anumite criterii pentru a putea aplica technica
- descriem algoritmul general pentru fiecare technică

Divide and conquer – Metoda divizării - pași

- Pas 1 **Divide -** se împarte problema în probleme mai mici (de același structură)
  - împărțirea problemei în două sau mai multe probleme disjuncte care se poate rezolva folosind același algoritm
- Pas 2 **Conquer –** se rezolvă subproblemele recursiv
- Step3 **Combine –** combinarea rezultatelor

Divide and conquer – algoritm general

```
def divideAndConquer(data):
    if size(data)<a:
        #solve the problem directly
        #base case
        return rez
    #decompose data into d1,d2,..,dk
    rez_1 = divideAndConquer(d1)
    rez_2 = divideAndConquer(d2)
    ...
    rez_k = divideAndConquer(dk)
    #combine the results
    return combine(rez_1,rez_2,...,rez_k)
```

Putem aplica divide and conquer dacă:

O promblemă *P* pe un set de date *D* poate fi rezolvat prin rezolvarea aceleiaşi probleme P pe un alt set de date D"= d_1, d_2, ..., d_k, de dimensiune mai mică decăt dimensiunea lui *D*

Complexitatea ca timp de execuție pentru o problemă rezolvată folosind divide and conquer poate fi descrisă de recurența:

$$T(n) = \begin{cases} solving\ trivial\ problem, & if\ n\ is\ small\ enough \\ k \cdot T(n/k) + time\ for\ dividing + time\ for\ combining, & otherwise \end{cases}$$

Divide and conquer – 1 / n-1

Putem divide datele în: date de dimensiune 1 și date de dimensiune n-1

Exemplu: Caută maximul

```
def findMax(l):
    """
      find the greatest element in the list
      l list of elements
      return max
    """
    if len(l)==1:
        #base case
        return l[0]
    #divide into list of 1 elements and a list of n-1 elements
    max = findMax(l[1:])
    #combine the results
    if max>l[0]:
        return max
    return l[0]
```

Complexitate timp

Recurența: $T(n)=\begin{cases}1 \text{ for } n=1\\ T(n-1)+1 \text{ otherwise}\end{cases}$

$$\begin{aligned}T(n)&=T(n-1)+1\\ T(n-1)&=T(n-2)+1\\ T(n-2)&=T(n-3)+1\\ &...=...\\ T(2)&=T(1)+1\end{aligned} \Rightarrow T(n)=1+1+...+1=n\in\theta(n)$$

Divizare în date de dimensiune n/k

```
def findMax(l):
    """
      find the greatest element in the list
      l list of elements
      return max
    """
    if len(l)==1:
        #base case
        return l[0]
    #divide into 2 of size n/2
    mid = len(l) /2
    max1 = findMax(l[:mid])
    max2 = findMax(l[mid:])
    #combine the results
    if max1<max2:
        return max2
    return max1
```

Complexitate ca timp:

Recurența: $T(n)=\begin{cases}1 \text{ for } n=1\\ 2\,T(n/2)+1 \text{ otherwise}\end{cases}$

Notăm: $n=2^k$ => $k=\log_2 n$

$$\begin{aligned} T(2^k) &= 2\,T(2^{(k-1)})+1\\ 2\,T(2^{(k-1)}) &= 2^2\,T(2^{(k-2)})+2\\ 2^2\,T(2^{(k-2)}) &= 2^3\,T(2^{(k-3)})+2^2 \\ &\dots = \dots\\ 2^{(k-1)}\,T(2) &= 2^k\,T(1)+2^{(k-1)} \end{aligned} \Rightarrow$$

$$T(n)=1+2^1+2^2..+2^k=(2^{(k+1)}-1)/(2-1)=2^k 2-1=2n-1\in\theta(n)$$

Divide and conquer - Exemplu

Calculați x^k unde $k \geq 1$ număr întreg

Aborare simplă: $x^k = k * k * ... * k$ - k-1 înmulțiri (se poate folosi un for) $T(n) \in \theta(n)$

Rezolvare cu metoda divizării:

$$x^k = \begin{cases} x^{(k/2)} x^{(k/2)} \text{ for } k \text{ even} \\ x^{(k/2)} x^{(k/2)} x \text{ for } k \text{ odd} \end{cases}$$

```
def power(x, k):
    """
      compute x^k
      x real number
      k integer number
      return x^k
    """
    if k==1:
        #base case
        return x
    #divide
    half = k/2
    aux =  power(x, half)
    #conquer
    if k%2==0:
        return aux*aux
    else:
        return aux*aux*x
```

Divide: calculează k/2

Conquer: un apel recursiv pentru a calcul $x^{(k/2)}$

Combine: una sau doua înmulțiri

Complexitate: $T(n) \in \theta(\log_2 n)$

Divide and conquer

- **Căutare binară ($T(n) \in \theta(\log_2 n)$)**
  - **Divide – impărțim lista în două liste egale**
  - **Conquer – căutăm în stânga sau în dreapta**
  - **Combine – nu e nevoie**
- **Quick-Sort ($T(n) \in \theta(n \log_2 n)$ mediu)**
- **Merge-Sort**
  - **Divide – impărțim lista în două liste egale**
  - **Conquer – sortare recursivă pentru cele două liste**
  - **Combine – interclasare liste sortate**

Backtracking

- **se aplică la probleme de căutare unde se caută mai multe soluții**
- **generează toate soluțiile (dacă sunt mai multe) pentru problemă**
- **caută sistematic prin toate variantele de soluții posibile**
- **este o metodă sistematică de a itera toate posibilele configurații în spațiu de căutare**
- **este o technică generală – trebuie adaptat pentru fiecare problemă în parte.**
- **Dezavantaj – are timp de execuție exponențial**

Algoritm general de descoperire a tuturor soluțiilor unei probleme de calcul
Se bazează pe construirea incrementală de soluții-candidat, abandonând fiecare candidat parțial imediat ce devine clar că acesta nu are șanse să devină o soluție validă

Metoda generării și testării (Generate and test)

Problemă – Fie *n* un număr natural. Tipăriți toate permutările numerelor 1, 2, ..., *n*.

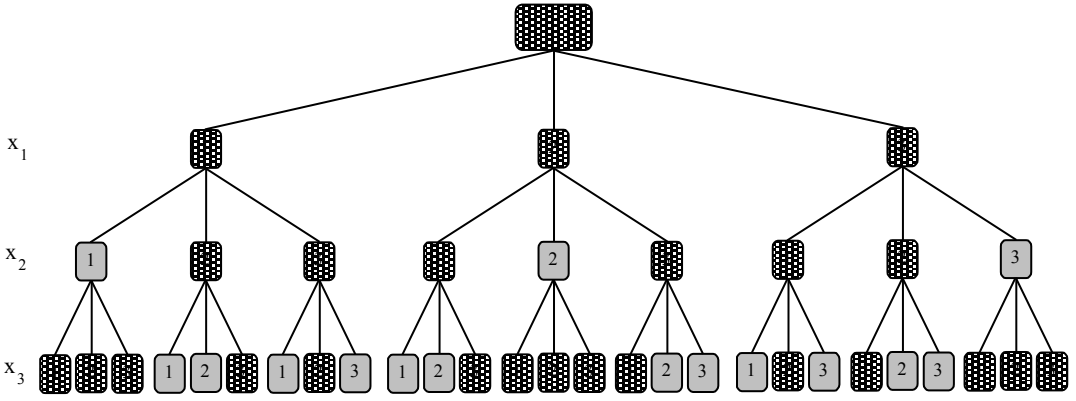
Pentru n=3

```
def perm3():
    for i in range(0,3):
        for j in range(0,3):
            for k in range(0,3):
                #a possible solution
                possibleSol = [i,j,k]
                if i!=j and j!=k and i!=k:
                    #is a solution
                    print possibleSol
```

```
[0, 1, 2]
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

- **Metoda generării și testării - *Generate and Test***
  - **Generare: se generează toate variantele posibile de liste de lungime 3 care conțin doar numerele 0,1,2**
  - **Testare: se testează fiecare variantă pentru a verifica dacă este soluție.**

Generare și testare – toate combinațiile posibile



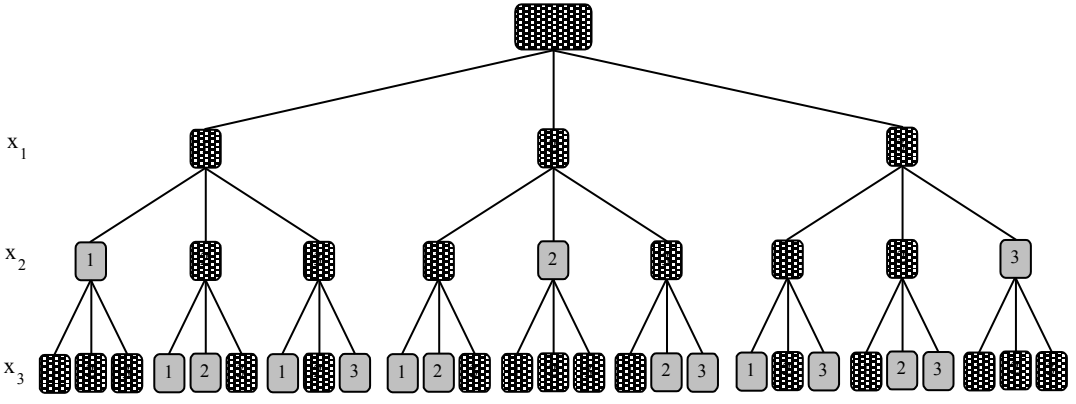
Probleme:

- Numărul total de **liste generate este 3^3**, în cazul general n^n
- inițial se generează toate componentele listei, apoi se verifica dacă lista este o permutare – in unele cazul nu era nevoie sa continuăm generarea (ex. Lista ce incepe cu 1,1 sigur nu conduce la o permutare
- Nu este general. Funcționează doar pentru n=3

În general: dacă n este afâncimea arborelui (numărul de variabile/componente în soluție) și presupunând că fiecare componentă poate avea k posibile valori, numărul de noduri în arbore este k^n. Înseamnă că pentru căutarea în întreg arborele avem o complexitate exponențială, $O(k^n)$.

Înbunătățiri posibile

- **să evităm crearea comăletă a soluției posibile în cazul în care știm cu siguranță că nu se ajunge la o soluție.**
  - **Dacă prima componentă este 1, atunci nu are sens să asignam 1 să pentru a doua componentă**



- **lucrăm cu liste parțiale (soluție parțială)**
- **extindem lista cu componente noi doar dacă sunt îndeplinite anumite condiții (*condiții de continuare)***
  - ***dacă lista parțială nu conține duplicate***

Generate and test - recursiv

folosim recursivitate pentru a genera toate soluțiile posibile (soluții candidat)

```
def generate(x,DIM):
    if len(x)==DIM:
        print x
    if len(x)>DIM:
        return
    x.append(0)
    for i in range(0,DIM):
        x[-1] = i
        generate(x[:],DIM)

generate([],3)
```

```
[0, 0, 0]
[0, 0, 1]
[0, 0, 2]
[0, 1, 0]
[0, 1, 1]
[0, 1, 2]
[0, 2, 0]
[0, 2, 1]
[0, 2, 2]
[1, 0, 0]
...
```

Testare – se tipărește doar soluția

```
def generateAndTest(x,DIM):
    if len(x)==DIM and isSet(x):
        print x
    if len(x)>DIM:
        return
    x.append(0)
    for i in range(0,DIM):
        x[-1] = i
        generateAndTest(x[:],DIM)
generateAndTest([],3)
```

```
[0, 1, 2]
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

- **În continuare se genereaza toate listele ex: liste care încep cu 0,0**
- **ar trebui sa nu mai generăm dacă conține duplicate Ex (0,0) – aceste liste cu siguranță nu conduc la rezultat – la o permutare**

Reducem spatiu de căutare – nu generăm chiar toate listele posibile

Un candidat e valid (merită să continuăm cu el) doar dacă nu conține duplicate

```
def backtracking(x,DIM):
    if len(x)==DIM:
        print x
    if len(x)>DIM:
        return  #stop recursion
    x.append(0)
    for i in range(0,DIM):
        x[-1] = i
        if isSet(x):
            #continue only if x can conduct to a solution
            backtracking(x[:],DIM)

backtracking([], 3)
```

```
[0, 1, 2]
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

Este mai bine decât varianta *generează și testează*, dar complexitatea ca timp de execuție este tot exponențial.

Permutation problem

- **rezultat:** $x=(x_0, x_1, .., x_n), x_i \in (0,1,.., n-1)$
- **e o solutie:** $x_i \neq x_j \; for \; any \; i \neq j$

8 Queens problem:

Plasați pe o tablă de sah 8 regine care nu se atacă.

- **Rezultat: 8 poziții de regine pe tablă**
- **Un rezultat partial e valid: dacă nu există regine care se atacă**
  - **nu e pe acelși coloana, linieor sau diagonală**
- **Numărul total de posibile poziții (atât valide cât și invalide):**
  - **combinări de 64 luate câte 8, C(64, 8) $\approx 4.5 \times 10^9$)**
- **Generează și testează – nu rezolvă problma în timp rezonabil**

Ar trebui sa generăm doar poziții care pot conduce la un rezultat (sa reducem spațiu de căutare)

- **Dacă avem deja 2 regine care se atacă nu ar trebui să mai continuăm cu această configurație**
- **avem nevoie de toate soluțiile**

Backtracking

- **spațiu de căutare**: $S = S_1 \times S_2 \times ... \times S_n$;
- ***x*** este un vector ce reprezintă **soluția**;
- $x[1..k]$ în $S_1 \times S_2 \times ... \times S_k$ este o **soluție candidat**; este o configurație parțială care ar putea conduce la rezultat; k este numărul de componente deja construită;
- **consistent** – o funcție care verifică dacă o soluție parțială este soluție candidat (poate conduce la rezultat)
- **soluție** este o funcție care verifică dacă o soluție candidat $x[1..k]$ este o soluție pentru problemă.

Algoritmul Backtracking – recursiv

```
def backRec(x):
    x.append(0) #add a new component to the candidate solution
    for i in range(0,DIM):
        x[-1] = i  #set current component
        if consistent(x):
            if solution(x):
                solutionFound(x)
            backRec(x[:]) #recursive invocation to deal with next components
```

Algoritm mai general (componentele soluției pot avea domenii diferite (iau valori din domenii diferite)

```
def backRec(x):
    el = first(x)
    x.append(el)
    while el!=None:
        x[-1] = el
        if consistent(x):
            if solution(x):
                outputSolution(x)
            backRec(x[:])
        el = next(x)
```

Backtracking

Cum rezolvăm problema folosind algoritmul generic:

- **trebuie sa reprezentăm soluția sub forma unui vector** $X=(x_{0,}x_{1,}\dots x_n)\in S_0 x S_1 x \dots x S_n$
- **definim ce este o soluție candidat valid (condiție prin care reducem spațiu de căutare)**
- **definim condiția care ne zice daca o soluție candidat este soluție**

```
def consistent(x):
    """
      The candidate can lead to an actual
      permutation only if there are no duplicate elements
    """
    return isSet(x)


def solution(x):
    """
      The candidate x is a solution if
      we have all the elements in the permutation
    """
    return len(x)==DIM
```

Metoda Greedy

- o strategie de rezolvare pentru probleme de optimizare
- aplicabil unde optimul global se poate afla selectând succesiv optime locale
- permite rezolvarea problemei fara a reveni asupra alegerilor facute pe parcurs
- folosit în multe probleme practice care necesite selecția unei mulțimi de elemente care satisfac anumite condiții și realizează un optim

Probleme

Problema rucsacului

Se dă un set de obiecte, caracterizate prin greutate și utilitate, și un rucsac care are capacitatea totala W. Se caută o submultime de obiecte astfel încât greutatea totală este mai mică decât W și suma utilității obiectelor este maximal.

Monede

Se da o sumă M și tipuri (ex: 1, 5, 25) de monede (avem un numar nelimitat de monede din fiecare tip de monedă). Să se găsească o modalitate de a plăti suma M de bani folosind cât mai puține monezi.

Forma generală a unei probleme Greedy-like

Având un set de obiecte C candidați pentru a face parte din soluție, se cere să se găsească un subset B ($B \subseteq C$) care indeplinește anumite condiții (condiții interne) și maximizează (minimizează) o funcție de obiectiv.

- Dacă un subset *X* îndeplinește condițiile interne atunci subsetul X este *acceptabil (*posibil)
- Unele probleme pot avea mai multe soluții acceptabile, caz în care se caută o soluție cât mai bună, daca se poate solutia ceea mai bună (cel care realizeaza maximul pentru o funcție obiectiv).

Pentru a putea rezolva o problema folosind metoda Greedy, problema trebuie să satisfacă proprietatea:

- dacă *B* este o soluție acceptabilă și $X \subset B$ atunci și *X* este o soluție acceptabilă

Algoritmul Greedy

Algoritmul Greedy găsește soluția incremetal, construind soluții acceptabile care se tot extind pe parcurs. La fiecare pas soluția este exstinsă cu cel mai bun candidat dintre candidații rămași la un moment dat. (Strategie greedy - lacom)

Principiu (strategia) Greedy :

- adaugă succesiv la rezultat elementul care realizează optimul local
- o decizie luată pe parcurs nu se mai modifică ulterior

Algoritmul Greedy

- Poate furniza soluția optimă (doar pentru anumite probleme)
  - alegerea optimului local nu garanteaza tot timpul optimul global
  - solutie optimă - dacă se găsește o modalitate de a alege (optimul local) astfel încat se ajunge la solutie optimă
  - în unele situații se preferă o soluție, chiar și suboptimă, dar obținut în timp rezonabil
- Construiește treptat soluția (fără reveniri ca în cazul backtracking)
- Furnizează o singură soluție
- Timp de lucru polinomial

Greedy – python

```
def greedy(c):
    """
       Greedy algorithm
       c - a list of candidates
       return a list (B) the solution found (if exists) using the greedy
strategy, None if the algorithm
       selectMostPromissing - a function that return the most promising
candidate
       acceptable - a function that returns True if a candidate solution can be
extended to a solution
       solution - verify if a given candidate is a solution
    """
    b = [] #start with an empty set as a candidate solution
    while not solution(b) and c!=[]:
        #select the local optimum (the best candidate)
        candidate = selectMostPromissing(c)
        #remove the current candidate
        c.remove(candidate)
        #if the new extended  candidate solution is acceptable
        if acceptable(b+[candidate]):
            b.append(candidate)

    if solution(b):
        return b
    #there is no solution
    return None
```

Algoritm Greedy - elemente

1. **Mulțimea candidat** (***candidate set)*** **–** de unde se aleg elementele soluției
2. **Funcție de selecție** (***selection function)*** **–** alege cel mai bun candidat pentru a fi adăugat la soluție;
3. **Acceptabil** (***feasibility function)*** **–** folosit pentru a determina dacă un candidat poate contribui la soluție
4. **Funcţie obiectiv** (***objective function)*** – o valoare pentru soluţie şi pentru orice soluţie parţială
5. **Soluţie (*solution function)***, - indică dacă am ajuns la soluţie

Exemplu

Se da o sumă M și tipuri (ex: 1, 5, 25) de monede (avem un numar nelimitat de monede din fiecare tip de monedă). Să se găsească o modalitate de a plăti suma M de bani folosind cât mai puține monezi.

Set Candidat:

Lista de monede - COINS = {1, 5, 25, 50}

Soluţie Candidat:

o listă de monede - $X=(X_0, X_1, .., X_k)$ **unde** $X_i \in COINS$ **– monedă**

Funcția de selecție:

candidate solution: $X=(X_0, X_1, .., X_k)$

alege moneda cea mai mare care e mai mic decăt ce mai e de plătit din sumă

Acceptabil (*feasibility function)*:

Soluție candidat: $X=(X_0, X_1, .., X_k)$ $S=\sum_{(i=0)}^{k} X_i \leq M$

suma monedelor din soluția candidat nu depășește suma cerută

Soluție:

Soluție candidat: $X=(X_0, X_1, .., X_k)$ $S=\sum_{(i=0)}^{k} X_i = M$

suma monedelor din soluția candidat este egal cu suma cerută

Monede – cod python

```
#Let us consider that we have a sum M of money and coins of 1, 5, 25 units (an unlimited number of coins).
#The problem is to establish a modality to pay the sum M using a minimum number of coins.
```

```
def selectMostPromissing(c):
    """
      select the largest coin from the remaining
      c - candidate coins
      return a coin
    """
    return max(c)
```

```
def acceptable(b):
    """
   verify if a candidate solution is valid
   basically verify if we are not over the sum M
    """
    sum = _computeSum(b)
    return sum<=SUM
```

```
def solution(b):
    """
   verify if a candidate solution is an actual solution
   basically verify if the coins conduct to the sum M
   b - candidate solution
    """
    sum = _computeSum(b)
    return sum==SUM

def _computeSum(b):
    """
   compute the payed amount with the current candidate
    return int, the payment
   b - candidate solution
    """
    sum = 0
    for coin in b:
        nrCoins = (SUM-sum) / coin
        #if this is in a candidate solution we need to
use at least 1 coin
        if nrCoins==0: nrCoins =1
        sum += nrCoins*coin
    return sum
```

```
def printSol(b):
 """
   Print the solution: NrCoinns1 * Coin1 +  NrCoinns2 *
Coin2 +...
 """
    solStr = ""
    sum = 0
    for coin in b:
        nrCoins = (SUM-sum) / coin
        solStr+=str(nrCoins)+"*"+str(coin)
        sum += nrCoins*coin
        if SUM-sum>0:solStr+=" + "
    print solStr
```

Greedy

1. Algoritmul Greedy are complexitate polinomială - $O(n^2)$ unde n este numărul de elemente din lista candidat C
2. Înainte de a aplica Greedy este nevoie să demonstrăm că metoda găsește soluția optimă. De multe ori demonstrația este netrivială
3. Există o mulțime de probleme ce se pot rezolva cu greedy. Ex: Algoritmul Kruskal – determinarea arborelui de acoperire, Dijkstra, Bellman-Kalaba – drum minim întrun graf neorientat
4. Există probleme pentru care Greedy nu găsește soluția optimă. În unele cazuri se preferă o soluție obținut în timp rezonabil (polinomial) care e aprope de soluția optimă, în loc de soluția optimă obținută în timp exponențial (*heuristics algorithms)*.

Programare Dinamică

Se poate folosi pentru a rezolva probleme de optimizare, unde:

- soluția este rezultatul unui șir de decizii, $d_1, d_2, ..., d_n$,
- *principiul optimalității* este satisfăcut.
- în general timp polinomial de execuție
- tot timpul găsește soluția optimă.
- Rezolvă problema combinân sub soluții de la subprobleme, calculează subsoluția doar o singură data (salvănd rezultatul pentru a fi refolosit mai târziu).

Fie stăritle $s_0, s_1, ..., s_{n-1}, s_n$, unde s_0 este starea inițială, s_n este starea finală, prin aplicariea succesivă a deciziilor $d_1, d_2, ..., d_n$ se ajunge la starea finală (decizia d_i duce din starea s_{i-1} în starea s_i, pentru i=1,n):

$$s_0 \xrightarrow{d_1} s_1 \xrightarrow{d_2} s_2 \rightarrow \dots \rightarrow s_{n-1} \xrightarrow{d_n} s_n$$

Programare Dinamică

Caracteristici:

- principiul optimalității;
- probleme suprapuse (overlapping sub problems);
- *memoization.*

Principiul optimalității

- *optimul general* implică *optimul parțial*
  - la greedy aveam *optimul local* implică *optimul global*
- într-o secvență de decizii optime, fiecare decizie este optimă.
- Principiul nu este satisfăcut pentru orice fel de problemă. In general nu e adevărat în cazul in care subsecvențele de decizii nu sunt independente și optimizarea uneia este în conflict cu optimizarea de la alta secvența de decizii.

Principiul optimalității

Dacă $d_1, d_2, ..., d_n$ este o secvență de decizii care conduce optim sistemul din starea inițială s_0 în starea finală s_n, atunci una din utmătoarele sunt satisfăcute:

1). $d_k, d_{k+1}, ..., d_n$ este o secvență optimă de decizii care conduce sistemul din starea s_{k-1} în starea s_n, $\forall k, 1 \le k \le n$. **(*forward* variant –** decizia d_k depinde de deciziile $d_{k+1}..d_n$**)**

2). $d_1, d_2, ..., d_k$ este o secvență optimă de decizii care conduce sistemul din starea s_0 în starea s_k, $\forall k, 1 \le k \le n$. **(*backward* variant)**

3). $d_{k+1}, d_{k+2}, ..., d_n$ și $d_1, d_2, ..., d_k$ sunt secvențe optime de decizii care conduc sistemul din starea s_k în starea s_n, respectiv, din starea s_0 în starea s_k, $\forall k, 1 \le k \le n$. **(*mixed* variant)**

Sub-Probleme suprapuse (Overlapping Sub-problems)

O problema are sub-probleme suprapuse daca poate fi inpărțit în subprobleme care se refolosesc de mai multe ori

Memorizare (Memorization)

salvarea rezultatelor de la o subproblemă pentru a fi refolosit

Cum aplicăm programare dinamică

- Principiul optimalității (oricare variantă: forward, backward or mixed) este demonstrat.
- Se definește structura solutiei optime.
- Bazat pe principiul optimalității, valoarea soluției optime se definește recursiv. Se definește o recurență care indică modalitatea prin care se opține optimul general din optime parțiale.
- Soluția optimă se calculează in manieră bottom-up, începem cu subproblema cea mai simplă pentru care soluția este cunoscută.

Cea mai lungă sublistă crescătoare

Se dă o listă $a_1, a_2, ..., a_n$. *Determinați cea mai lungă sublistă crescătoare* $a_{i_1}, a_{i_2}, ..., a_{i_s}$ *a listei date.*

Soluție:

- *Pricipiul optimalității*
  - varianta înainte *forward*
- *The structure of the optimal solution*
  - Construim două șiruri: $l =< l_1 . l_2, ... l_n >$ și $p =< p_1 . p_2, ... p_n >$.
    - l_k lungime sublistei care incepe cu elementul a_k.
    - p_k indexul elementului a care urmează după elementul a_k în sublista cea mai lungă care incepe cu a_k.
- *Definiția recursivă*

– $l_n = 1,\ p_n = 0$

– $l_k = \max\{1 + l_i \mid a_i \geq a_k,\ k+1 \leq i \leq n\}, \quad \forall k = n-1, n-2, ...1$

– $p_k = \arg\max\{1 + l_i \mid a_i \geq a_k,\ k+1 \leq i \leq n\}, \quad \forall k = n-1, n-2, ..., 1$

Cea mai lungă sublistă crescătoare– python

```
def longestSublist(a):
    """
      Determines the longest increasing sub-list
      a -  a list of element
      return sublist of x, the longest increasing sublist

    """
    #initialise l and p
    l = [0]*len(a)
    p = [0]*len(a)
    l[lg-1] = 1
    p[lg-1]=-1
    for k in range(lg-2, -1, -1):
        print p, l
        p[k] = -1
        l[k]=1
        for i in range(k+1, lg):
            if a[i]>=a[k] and l[k]<l[i]+1:
                l[k] = l[i]+1
                p[k] = i

    #identify the longest sublist
    #find the maximum length
    j = 0
    for i in range(0, lg):
        if l[i]>l[j]:
            j=i
    #collect the results using the position list
    rez = []
    while j!=-1:
        rez = rez+[a[j]]
        j = p[j]
    return rez
```