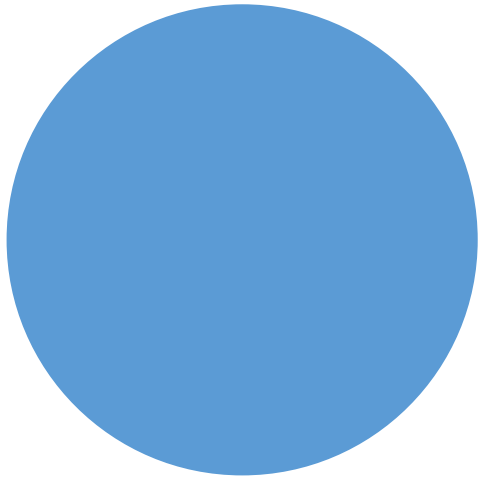
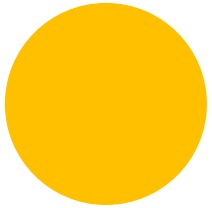



Programare 1

Functii, module

Cursul 6

Despre ce am discutat în cursul precedent?

- șiruri de caractere
- formatare șirurilor de caractere
- expresii regulate

Despre ce o să discutăm astăzi?

Funcții,

- apelul funcțiilor,
- context.
- Variabile locale și globale

Functii recursive

Module si pachete

Să considerăm un exemplu – o lanterna

Decompoziție

- Avem două perspective:

➢ Cum funcționează ?

- ❖ Din ce componente este formată ?
- ❖ Cum interacționează aceste componente între ele ?

➢ Cum o utilizăm ?

- ❖ Ce trebuie să știm ca să o putem utiliza ?
- ❖ Cum o pornim/oprim ?

Abstractizare

Decompoziție

- **Ideea:** mai multe componente funcționează împreună ca să obțină un rezultat (ex. o lanternă funcțională)
- Între toate aceste componente există convenții clar stabilite de interacțiune (ex: becul se aprinde după ce întrerupătorul închide circuitul cu bateria)
- Acest concept se transpune și în cod

Abstractizare

- **Ideea:** nu trebuie să știi cum funcționează o lanternă ca să o folosești
- O lanternă este un „black box” (o cutie neagră), nu știm cum funcționează
- Știm care este „interfața” unei lanterne: cum o pornim/oprim (apăsăm un buton)
- Cumva această „cutie neagră” face ca atunci când apăsăm butonul să se aprindă lumina!

Aplicați aceste concepte în programare!

Introduceți STRUCTURĂ folosind DECOMPOZIȚIE

- În programare ne structurăm codul în **module**
  - Sunt **autonome**
  - Folosite pentru a **descompune** codul
  - Menite a fi **reutilizate**
  - Folosite pentru a avea **cod organizat**
  - Pentru **cod coerent**
- Ajungem la decompoziție folosind funcții
- Revenim la module si pachete Python

Ascundeți DETALIILE folosind ABSTRACTIZARE

În exemplul nostru cu lanterna este suficient să avem un minim manual de utilizare, nu este necesar să avem schema lanternei

În programare, putem să gândim la o bucată de cod ca un „black box”

- Nu vedem detaliile
- Nu avem nevoie sa știm detaliile
- Nu ne dorim detaliile
- Ascunde detaliile de programare

Obținem abstractizare folosind **specificațiile de funcții** și **documentație**

Funcțiile

Ne dorim să scriem bucăți reutilizabile de cod numite funcții

Funcțiile nu sunt executate într-un program până când nu sunt **apelate** sau **invocate**

Funcțiile au următoarele caracteristici:

- Sunt referite printr-un **nume**
- Au **parametrii** (0 sau mai mulți)
- Au documentație/**docstring** (opțional dar recomandat)
- Au un **corp**
- **Returnează** ceva

Cum scriem și apelăm o funcție

Cuvânt cheie

Nume

Parametrii sau argumente

Specificație sau documentație/docstring

```
def este_par (i):
    """
    Intrare: i, un `int`
    Returnează True dacă i este par sau False dacă este impar
    """
    print ("in este_par")
    return i%2 == 0

este_par (3)
```

Corpul funcției

Mai târziu apelăm funcția și furnizăm valori pentru argumente

În corpul funcției

```
def este_par (i):
    """
    Intrare: i, un `int`
    Returnează True dacă i este par sau False dacă este impar
    """
    print ("in este_par")
    return i%2 == 0
```

Instrucțiuni

Cuvânt cheie

Expresie de evaluat și returnat rezultatul

Variabile – Domeniul de vizibilitate

Parametrii formali sunt legați la **parametrii efectivi** în momentul apelului

Un nou **context/domeniu de vizibilitate** este creat când execuția intră într-o funcție

Domeniul de vizibilitate/contextul reprezintă maparea obiectelor la obiecte

```
def f(x):
    x = x + 1
    print ("in f(x): x =", x)
    return x

x = 3
z = f(x)
```

Parametru formal

Parametru efectiv

Definiția funcției

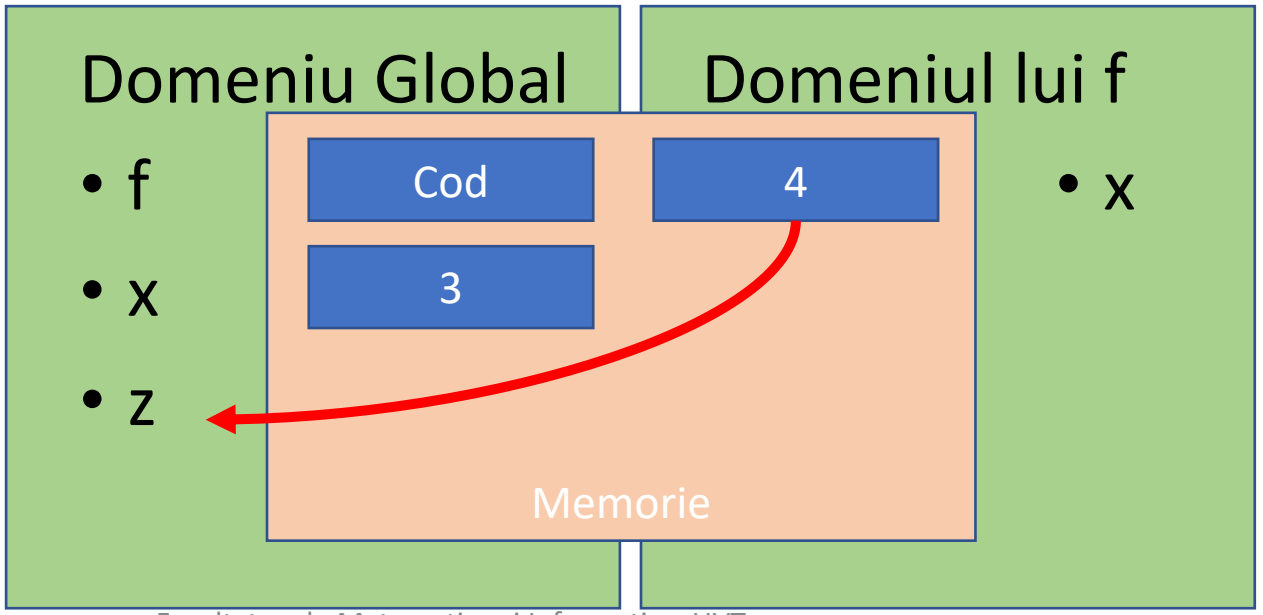
Programul principal:

- Inițializează variabile x
- Apelează funcția f
- Leagă rezultatul funcției cu variabila z

Variabile – Domeniul de vizibilitate

```
def f(x):
    x = x + 1
    print ("in f(x): x =", x)
    return x

x = 3
z = f(x)
```



Ce se întâmplă dacă nu returnăm nimic ?

Python returnează pentru noi valoarea **None** dacă nu am returnat noi explicit nimic

None este folosit pentru a reprezenta **absența** valorilor

Return vs print

return

- Are sens doar **într-o funcție**
- Doar **un singur return** este executat într-o funcție
- Codul din funcție situat după return nu este executat
- Are o valoare asociată care este **returnată apelantului**

print

- Poate fi folosit în afara funcțiilor
- Putem apela print de mai multe ori
- Codul din funcție poate fi executat după print
- Are o valoare asociată care este afișată pe consolă

Funcții ca argumente

- Argumentele pot să fie orice fel de obiect, **inclusiv funcții**

```
def a():
    print ("In functia a")
def b(y):
    print ("In functia b")
    return y
def c():
    print ("In functia c")
    return z()

print (a())
print (5+b(2))
print (c(a))
```

Functii recursive

- Sunt functii care se auto-apeleaza
- Trebuie sa avem grija la
  - conditia de oprire
  - Relatia de recurenta
- Ex: factorial n! = n x (n-1) x … x 1

```
def factorial1(n):
    return n * factorial1(n-1)
```

```
def factorial2(n):
    print(n)
    if n < 2:
        return 1
    else:
        return n * factorial2(n-1)
```

Factorial de cat?

7710530113353860041446393977750283605955564018160102391634109940339708518270930
6936709076979553903309264786122423067744465978515263974540148018465317490976250
4470638274259120173309701702610875092918816846985842150593623718603861642063078
8341172340985137252650454025230565756588606212388704126402196299710246868266247
1338366096312704819557227970771168835262025986914099490128789574729041072249610
6151954257267396322405556727354786893725785838732404646243357335918597747405776
3289247758975645195835913540808981170231327622507140572713441109481640299405888
2784778044231447320047952513831820830242772780313321930521095250760594899431434
5449325259594876385922128494560437296428386002940601874072732488897504223793518
3771806054417831166497082699460613802305310182919305107486655778030145232517977
9038861503375654483037490944016227018295230332909172043821063709710561625838705
1884030288933650309756289188364568672104084185529365727646234588306683493594765
2745594975437596517336998206397317021169129632474412942002978000870617258682238
8086524358336562348270439589365271184073541879977376305488758821994398467340105
1362280384187818611005035187862707840912942753454646054674870155072495767509778
5340592980383642040762990480729345010462551753783230082176707316495199556990844
8233079881104916627624925132654431258028935781292482589821746284829764834940083
8815410152872456707653654424335818651136964880049831580548028614922852377435001
5113776560157309592546471712909305173403672876570076061776754838305214997078734
4901684440239020374663308696974768067146854168726582363792200741384911859348771
0272883164905548707198762911703545119701275432473548172544699118836274377270607
4206521330926862820817773836744878816288008019281030158328210212863221204608749
4169719948775876973054492201238969450496000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
00

factorial(800)

Module în Python

Ce este un modul?

- un set de funcții grupate într-un fișier, care pot fi accesate extern

Exista module predefinite (vom învăța sa ne scriem propriile module în cursurile viitoare)

Exemple de module :

- **math** - funcții matematice (build-in)
- **random** - generare de numere aleatoare (build-in)
- **cv2** - procesare de imagini (bibliotecă externă)
- **numpy** - operații rapide pe array-uri (bibliotecă externă)

Cum se utilizează modulele în Python

import

- permite preluarea tuturor funcțiilor dintr-un modul

```
In [11]: import math

         v1 = math.sqrt(16)
         v2 = math.log10(100)
```

from

- permite preluarea individuala a funcțiilor dintr-un modul

```
In [ ]: from math import sqrt, log10

        v1 = sqrt(16)
        v2 = log10(100)
```

Redenumire

- folosind cuvântul **as**

import

```
In [12]: import math as m

         v1 = m.sqrt(16)
         v2 = m.log10(100)
```

from

```
In [13]: from math import sqrt as sq, log10 as lg

         v1 = sq(16)
         v2 = lg(100)
```

Exemplu

In [9]:
```
from random import random
from random import randint

r = random()
print(r)

r = random.randint(5,10)
print(r)
```

Functioneaza

Exemplu - alternative corecte

In [10]:
```
from random import random
from random import randint

r = random()
print(r)

r = randint(5,10)
print(r)
```

```
0.25200952128021614
10
```

In [12]:
```
import random


r = random.random()
print(r)

r = random.randint(5,10)
print(r)
```

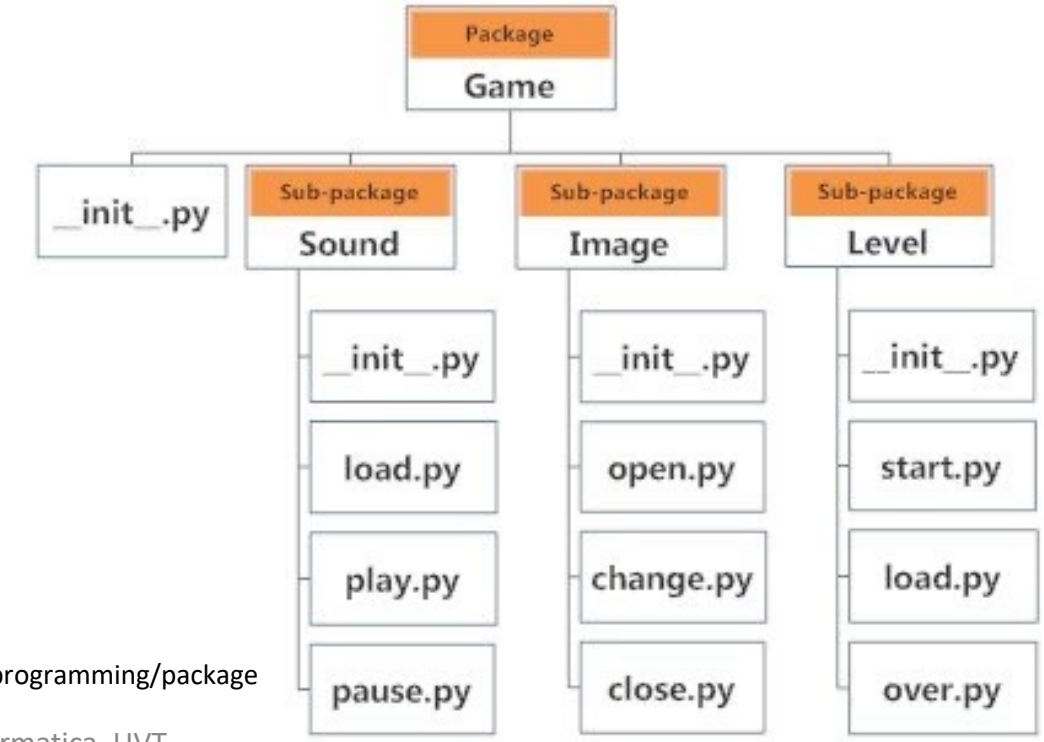
```
0.668119973038372
5
```

Package (pachete) în Python

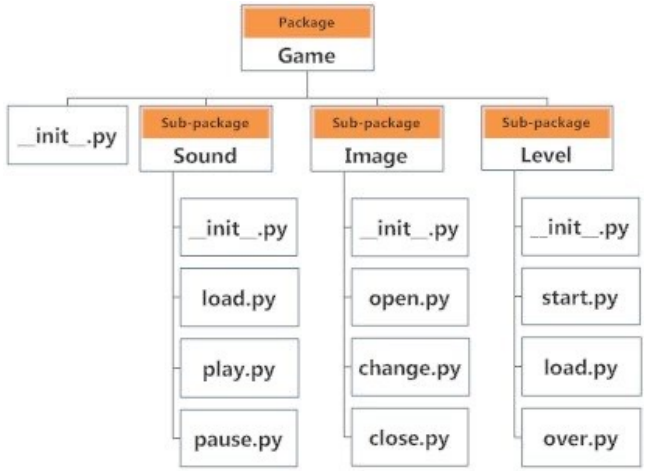
Ce este un pachet?

- un set de module grupate într-un director, care pot fi accesate extern

Modulele fiind fisiere care contin functii



from https://www.programiz.com/python-programming/package



Cum folosim pachetele

from - La fel ca si la module

Importam un anumit modul din pachet

```
from Game.Level import start

start.select_difficulty(2)
```

Sau doar o anumita functie

```
from Game.Level.start import select_difficulty

select_difficulty(2)
```