Tutoriatul 05

Funcții în Python, probleme + Recursivitate

Ce este aia o funcție în Python?

Ce este aia o funcție în Python?

- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de situații în care regăsim ideea de funcții:

Ce este aia o funcție în Python?

- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de situații în care regăsim ideea de funcții:

1) O funcție matematică $f : \mathbb{R} \to \mathbb{R}, f(x) = e^x$

Input: număr real
Output: număr pozitiv

Ce este aia o funcție în Python?

- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de situații în care regăsim ideea de funcții:

1) O funcție matematică $f : \mathbb{R} \to \mathbb{R}, f(x) = e^x$

Input: număr real
Output: număr pozitiv

2) Mașina de spălat

Input: rufe, modul de spălare
Output: rufe curate

Ce este aia o funcție în Python?

- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de situații în care regăsim ideea de funcții:

1) O funcție matematică

$$f : \mathbb{R} \to \mathbb{R}, f(x) = e^x$$

Input: număr real
Output: număr pozitiv

2) Mașina de spălat

Input: rufe, modul de spălare
Output: rufe curate

3) “Du și tu paharul la bucătărie…”

Input: pahar
Output? Nu avem, decât simpla confirmare că am făcut “ceva”

Ce este aia o funcție în Python?

- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de situații în care regăsim ideea de funcții:

1) O funcție matematică $f : \mathbb{R} \to \mathbb{R}, f(x) = e^x$

Input: număr real
Output: număr pozitiv

2) Mașina de spălat

Input: rufe, modul de spălare
Output: rufe curate

3) “Du și tu paharul la bucătărie…”

Input: pahar
Output? Nu avem, decât simpla confirmare că am făcut “ceva”

4) “Mestecă te rog în oală”

Input? (nu suntem trimiși cu lingura după noi)
Output? Confirmare

Dar care e ideea? De ce sunt utile funcțiile?

- Reduc din volumul codului
- Fac codul mai lizibil
- Împart un cod complex în mai multe funcții care contribuie la algoritm
- Reprezintă un scop fundamental pentru informatică

Ce este aia o funcție în Python?

- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de funcție:

Ce este aia o funcție în Python?

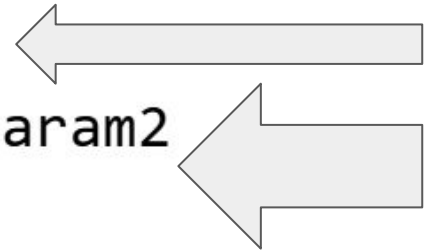
- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de funcție:

```
def schimbare(param1, param2):
    param2, param1 = param1, param2
    return param1, param2
```

Ce este aia o funcție în Python?

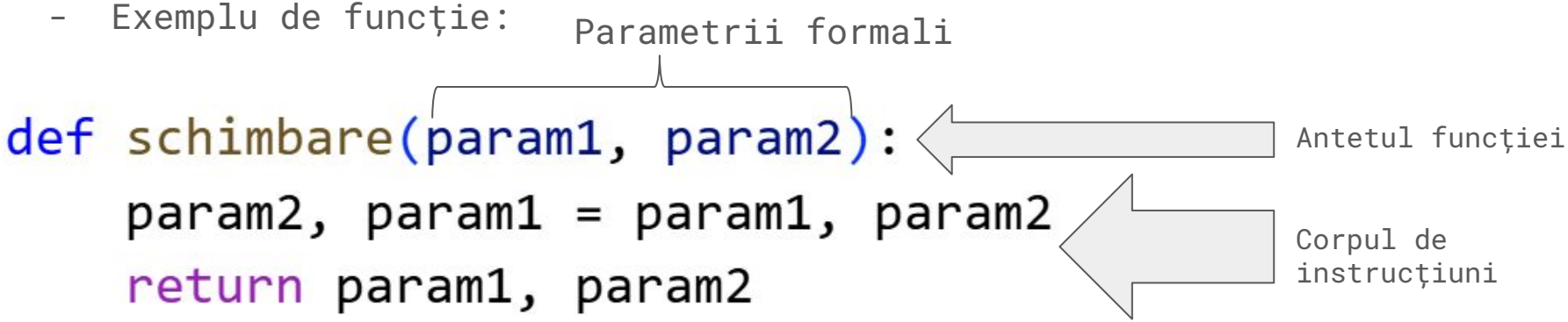
- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de funcție:

```
def schimbare(param1, param2):
    param2, param1 = param1, param2
    return param1, param2
```



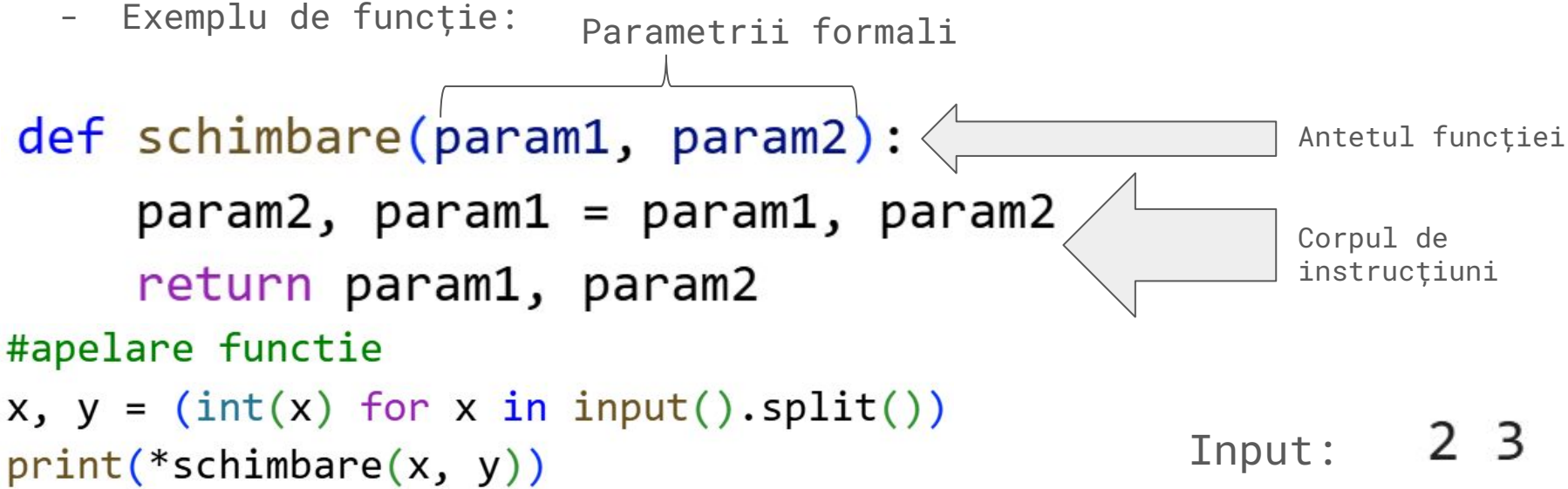
Ce este aia o funcție în Python?

- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de funcție:



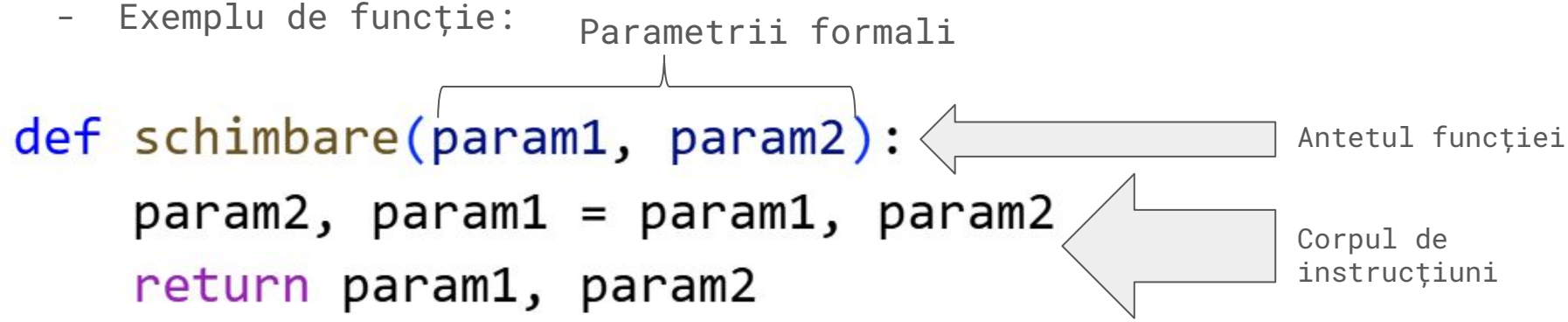
Ce este aia o funcție în Python?

- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de funcție:



Ce este aia o funcție în Python?

- O funcție este o secvență numită de instrucțiuni care prelucrează cu anumiți parametrii
- Exemplu de funcție:




```
#apelare functie
x, y = (int(x) for x in input().split())
print(*schimbare(x, y))
```

Input: 2 3

Output: 3 2

Funcții predefinite

- Există funcții în python care nu sunt incluse automat (trebuie “importat” modulul din care provin)

Funcții predefinite

- Există funcții în python care nu sunt incluse automat (trebuie “importat” modulul din care provin)

```
import math
print(math.sin(math.pi/2))
```

```
1.0
```

Funcții predefinite

- Există funcții în python care nu sunt incluse automat (trebuie “importat” modulul din care provin)

```
import math
print(math.sin(math.pi/2))
```

```
1.0
```

- Module importante pe care le vom întâlni: math (pentru funcții matematice uzuale, constante), copy (pentru deepcopy()), numpy, matplotlib (semestrul 2, ECS)
- 2 moduri de a accesa un modul:

Funcții predefinite

- Există funcții în python care nu sunt incluse automat (trebuie “importat” modulul din care provin)

```
import math
print(math.sin(math.pi/2))
```

```
1.0
```

- Module importante pe care le vom întâlni: math (pentru funcții matematice uzuale, constante), copy (pentru deepcopy()), numpy, matplotlib (semestrul 2, ECS)
- 2 moduri de a accesa un modul:

```
import math
print(math.sin(math.pi/2))
```

```
1.0
```

Funcții predefinite

- Există funcții în python care nu sunt incluse automat (trebuie “importat” modulul din care provin)

```
import math
print(math.sin(math.pi/2))
```

```
1.0
```

- Module importante pe care le vom întâlni: math (pentru funcții matematice uzuale, constante), copy (pentru deepcopy()), numpy, matplotlib (semestrul 2, ECS)
- 2 moduri de a accesa un modul:

```
import math
print(math.sin(math.pi/2))
```

```
1.0
```

```
import numpy as np
a = np.identity(3)
print(a)
```

Funcții predefinite

- Există funcții în python care nu sunt incluse automat (trebuie “importat” modulul din care provin)

```
import math
print(math.sin(math.pi/2))
```

```
1.0
```

- Module importante pe care le vom întâlni: math (pentru funcții matematice uzuale, constante), copy (pentru deepcopy()), numpy, matplotlib (semestrul 2, ECS)
- 2 moduri de a accesa un modul:

```
import math
print(math.sin(math.pi/2))
```

```
1.0
```

```
import numpy as np
a = np.identity(3)
print(a)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Funcții definite de utilizator

- Se folosește eticheta ‘def’ pentru a declara o nouă funcție, alături de numele funcției (după care va fi apelată) și parametrii formali

Funcții definite de utilizator

- Se folosește eticheta ‘def’ pentru a declara o nouă funcție, alături de numele funcției (după care va fi apelată) și parametrii formali

```
def f(parametrul_1, parametrul_2):
    corp_instructiuni()
```

Funcții definite de utilizator

- Se folosește eticheta ‘def’ pentru a declara o nouă funcție, alături de numele funcției (după care va fi apelată) și parametrii formali

```
def f(parametrul_1, parametrul_2):
    corp_instructiuni()
```

În general, corpul de instrucțiuni este menit să lucreze cu parametrii primiți

Funcții definite de utilizator

- Se folosește eticheta ‘def’ pentru a declara o nouă funcție, alături de numele funcției (după care va fi apelată) și parametrii formali

```
def f(parametrul_1, parametrul_2):
    corp_instructiuni()
    return x
```

În general, corpul de instrucțiuni este menit să lucreze cu parametrii primiți

Funcția se oprește la prima instrucțiune de return la care ajunge și returnează ce este specificat

Funcții definite de utilizator

- Se folosește eticheta ‘def’ pentru a declara o nouă funcție, alături de numele funcției (după care va fi apelată) și parametrii formali

```
def paritate(x):
    if x % 2 == 1:
        return 'x impar'
    return 'x par'
print(paritate(3))
print(paritate(2))
```

În general, corpul de instrucțiuni este menit să lucreze cu parametrii primiți

Funcția se oprește la prima instrucțiune de return la care ajunge și returnează ce este specificat

Output:

Funcții definite de utilizator

- Se folosește eticheta ‘def’ pentru a declara o nouă funcție, alături de numele funcției (după care va fi apelată) și parametrii formali

```
def paritate(x):
    if x % 2 == 1:
        return 'x impar'
    return 'x par'
print(paritate(3))
print(paritate(2))
```

În general, corpul de instrucțiuni este menit să lucreze cu parametrii primiți

Funcția se oprește la prima instrucțiune de return la care ajunge și returnează ce este specificat

Output:

```
x impar
x par
```

Funcții definite de utilizator

- Se folosește eticheta ‘def’ pentru a declara o nouă funcție, alături de numele funcției (după care va fi apelată) și parametrii formali

În general, corpul de instrucțiuni este menit să lucreze cu parametrii primiți

Funcția se oprește la prima instrucțiune de return la care ajunge și returnează ce este specificat

DAR poate să nu returneze o valoare, caz în care funcția returnează în mod implicit None

Funcții definite de utilizator

- Se folosește eticheta ‘def’ pentru a declara o nouă funcție, alături de numele funcției (după care va fi apelată) și parametrii formali

```
print(print("sir"))
```

Output:

În general, corpul de instrucțiuni este menit să lucreze cu parametrii primiți

Funcția se oprește la prima instrucțiune de return la care ajunge și returnează ce este specificat

DAR poate să nu returneze o valoare, caz în care funcția returnează în mod implicit None

Funcții definite de utilizator

- Se folosește eticheta ‘def’ pentru a declara o nouă funcție, alături de numele funcției (după care va fi apelată) și parametrii formali

```
print(print("sir"))
```

Output:

```
sir
None
```

În general, corpul de instrucțiuni este menit să lucreze cu parametrii primiți

Funcția se oprește la prima instrucțiune de return la care ajunge și returnează ce este specificat

DAR poate să nu returneze o valoare, caz în care funcția returnează în mod implicit None

Declarare

Apelare

Declarare

O funcție poate avea orice număr de parametrii

Apelare

Declarare

O funcție poate avea orice număr de parametrii

```
def f(x, y, z):
    return x+y
```

Apelare

Declarare

O funcție poate avea orice număr de parametrii

```
def f(x, y, z):
    return x+y
```

Apelare

Putem asocia fiecare parametru
- Fie prin poziție (ordinea din antet)

Declarare

O funcție poate avea orice număr de parametrii

```
def f(x, y, z):
    return x+y
```

Apelare

Putem asocia fiecare parametru
- Fie prin poziție (ordinea din antet)

```
print(f(1, 2, 3))
```

Declarare

O funcție poate avea orice număr de parametrii

```
def f(x, y, z):
    return x+y
```

Apelare

Putem asocia fiecare parametru

- Fie prin poziție (ordinea din antet)

```
print(f(1, 2, 3))
```

- Fie prin numire directă

Declarare

O funcție poate avea orice număr de parametrii

```
def f(x, y, z):
    return x+y
```

Apelare

Putem asocia fiecare parametru

- Fie prin poziție (ordinea din antet)

```
print(f(1, 2, 3))
```

- Fie prin numire directă

```
print(f(z=3, x=1, y=2))
```

Output:

Declarare

O funcție poate avea orice număr de parametrii

```
def f(x, y, z):
    return x+y
```

Apelare

Putem asocia fiecare parametru

- Fie prin poziție (ordinea din antet)

```
print(f(1, 2, 3))
```

- Fie prin numire directă

```
print(f(z=3, x=1, y=2))
```

Output:

```
3
3
```

Declarare

O funcție poate returna orice număr de variabile

Apelare

Declarare

O funcție poate returna orice număr de variabile (tuplu)

```
def f(x, y, z):
    return x+y, y*x, x+z
```

Apelare

Declarare

O funcție poate returna orice număr de variabile (tuplu)

```
def f(x, y, z):
    return x+y, y*x, x+z
```

Apelare

Putem accesa fiecare variabila returnată prin despachetare de tupluri

Declarare

O funcție poate returna orice număr de variabile (tuplu)

```
def f(x, y, z):
    return x+y, y*x, x+z
```

Apelare

Putem accesa fiecare variabila returnată prin despachetare de tupluri

```
s, p, s1 = f(1, 2, 3)
print(f(1, 2, 3))
print(s, p, s1)
```

Output:

Declarare

O funcție poate returna orice număr de variabile (tuplu)

```
def f(x, y, z):
    return x+y, y*x, x+z
```

Apelare

Putem accesa fiecare variabila returnată prin despachetare de tupluri

```
s, p, s1 = f(1, 2, 3)
print(f(1, 2, 3))
print(s, p, s1)
```

Output:

```
(3, 2, 4)
3 2 4
```

Declarare

O funcție poate avea parametrii cu valori implicite (parametri care, dacă nu primesc valori la apelarea funcției, vor prelua valoarea declarată în antetul funcției)

Apelare

Declarare

O funcție poate avea parametrii cu valori implicite (parametri care, dacă nu primesc valori la apelarea funcției, vor prelua valoarea declarată în antetul funcției)

```
def f(x, y, z=24):
    return x+y+z
```

Apelare

Declarare

O funcție poate avea parametrii cu valori implicite (parametri care, dacă nu primesc valori la apelarea funcției, vor prelua valoarea declarată în antetul funcției)

```
def f(x, y, z=24):
    return x+y+z
```

Apelare

Putem să apelăm funcția fără a specifica o valoare pentru parametrul specific (pot fi considerați drept parametrii opționali)

Declarare

O funcție poate avea parametrii cu valori implicite (parametri care, dacă nu primesc valori la apelarea funcției, vor prelua valoarea declarată în antetul funcției)

```
def f(x, y, z=24):
    return x+y+z
```

Apelare

Putem să apelăm funcția fără a specifica o valoare pentru parametrul specific (pot fi considerați drept parametrii opționali)

```
print(f(1, 2))
print(f(1, 2, 0))
```

Output:

Declarare

O funcție poate avea parametrii cu valori implicite (parametri care, dacă nu primesc valori la apelarea funcției, vor prelua valoarea declarată în antetul funcției)

```
def f(x, y, z=24):
    return x+y+z
```

Apelare

Putem să apelăm funcția fără a specifica o valoare pentru parametrul specific (pot fi considerați drept parametrii opționali)

```
print(f(1, 2))
print(f(1, 2, 0))
```

Output:

```
27
3
```

Declarare

O funcție poate avea parametrii cu valori implicite (parametri care, dacă nu primesc valori la apelarea funcției, vor prelua valoarea declarată în antetul funcției)

```
def f(x, y, z=24):
    return x+y+z
```

IMPORTANT! Parametrii opționali vin după cei obligatorii în declarare

Apelare

Putem să apelăm funcția fără a specifica o valoare pentru parametrul specific (pot fi considerați drept parametrii opționali)

```
print(f(1, 2))
print(f(1, 2, 0))
```

Output:

```
27
3
```

Declarare

O funcție poate avea un număr variabil de parametrii (pentru a specifica asta trebuie pus prefixul * pe parametrul care are un număr variabil de valori(tuple packing))

Apelare

Declarare

O funcție poate avea un număr variabil de parametrii (pentru a specifica asta trebuie pus prefixul * pe parametrul care are un număr variabil de valori(tuple packing))

```
def suma(*t):
    s = 0
    for termen in t:
        s = s + termen
    return s
```

Apelare

Declarare

O funcție poate avea un număr variabil de parametrii (pentru a specifica asta trebuie pus prefixul * pe parametrul care are un număr variabil de valori(tuple packing))

```
def suma(*t):
    s = 0
    for termen in t:
        s = s + termen
    return s
```

Apelare

Putem să apelăm funcția cu orice număr de parametrii

Declarare

O funcție poate avea un număr variabil de parametrii (pentru a specifica asta trebuie pus prefixul * pe parametrul care are un număr variabil de valori(tuple packing))

```
def suma(*t):
    s = 0
    for termen in t:
        s = s + termen
    return s
```

Apelare

Putem să apelăm funcția cu orice număr de parametrii

```
print(suma(1, 2, 3))
print(suma(2, 3, 5, 7, 11))
print(suma())
```

Output:

Declarare

O funcție poate avea un număr variabil de parametrii (pentru a specifica asta trebuie pus prefixul * pe parametrul care are un număr variabil de valori(tuple packing))

```
def suma(*t):
    s = 0
    for termen in t:
        s = s + termen
    return s
```

Apelare

Putem să apelăm funcția cu orice număr de parametrii

```
print(suma(1, 2, 3))
print(suma(2, 3, 5, 7, 11))
print(suma())
```

Output:

```
6
28
0
```

Declarare

O funcție poate avea un număr variabil de parametrii (pentru a specifica asta trebuie pus prefixul * pe parametrul care are un număr variabil de valori(tuple packing))

```
def suma(*t):
    s = 0
    for termen in t:
        s = s + termen
    return s
```

IMPORTANT! Parametrii normali vin înaintea parametrului cu număr variabil de valori (pentru a evita conflicte)

Apelare

Putem să apelăm funcția cu orice număr de parametrii

```
print(suma(1, 2, 3))
print(suma(2, 3, 5, 7, 11))
print(suma())
```

Output:

```
6
28
0
```

Variabile globale si variabile locale

- Toate variabilele create în blocul de instrucțiuni al unei funcții vor fi șterse după ce rulează

Variabile globale si variabile locale

- Toate variabilele create în blocul de instrucțiuni al unei funcții vor fi șterse după ce rulează

```
def loco():
    x = 78
    y = 'trei'
    return str(x)+y
loco()
```

Output:

Variabile globale si variabile locale

- Toate variabilele create în blocul de instrucțiuni al unei funcții vor fi șterse după ce rulează

```
def loco():
    x = 78
    y = 'trei'
    return str(x)+y
loco()
```

Output:

```
'78trei'
```

Variabile globale si variabile locale

- Toate variabilele create în blocul de instrucțiuni al unei funcții vor fi șterse după ce rulează

```
def loco():
    x = 78
    y = 'trei'
    return str(x)+y
loco()
print(y)
```

Output:

```
'78trei'
```

Variabile globale si variabile locale

- Toate variabilele create în blocul de instrucțiuni al unei funcții vor fi șterse după ce rulează. Aceste variabile temporare se numesc **variabile locale.**

```
def loco():
    x = 78
    y = 'trei'
    return str(x)+y
loco()
print(y)
```

Output:

```
'78trei'
---------------------------------------
NameError
/tmp/ipython-input-406592607.py in <cell
      4     return str(x)+y
      5 loco()
----> 6 print(y)

NameError: name 'y' is not defined
```

Variabile globale si variabile locale

- Toate variabilele create în afara unei funcții vor fi accesibile de către orice funcție din program, **chiar dacă nu o transmitem ca parametru**

Variabile globale si variabile locale

- Toate variabilele create în afara unei funcții vor fi accesibile de către orice funcție din program, **chiar dacă nu o transmitem ca parametru**

```
def rcd():
    print(x+1)

x = 7
rcd()
```

Output:

Variabile globale si variabile locale

- Toate variabilele create în afara unei funcții vor fi accesibile de către orice funcție din program, **chiar dacă nu o transmitem ca parametru**

```
def rcd():
    print(x+1)

x = 7
rcd()
```

Output:

```
8
```

Variabile globale si variabile locale

- Toate variabilele create în afara unei funcții vor fi accesibile de către orice funcție din program, **chiar dacă nu o transmitem ca parametru**
- Totuși, atenția tot este necesară

Variabile globale si variabile locale

- Toate variabilele create în afara unei funcții vor fi accesibile de către orice funcție din program, **chiar dacă nu o transmitem ca parametru**
- Totuși, atenția tot este necesară

```
def rcd():
    x = x + 1
    return x
x = 7
print(rcd())
```

Output:

Variabile globale si variabile locale

- Toate variabilele create în afara unei funcții vor fi accesibile de către orice funcție din program, **chiar dacă nu o transmitem ca parametru**
- Totuși, atenția tot este necesară

```
def rcd():
    x = x + 1
    return x
x = 7
print(rcd())
```

Output:

```
---------------------------------------------------------------------------
UnboundLocalError
/tmp/ipython-input-2780583115.py :
      3     return x
      4 x = 7
----> 5 print(rcd())

/tmp/ipython-input-2780583115.py :
      1 def rcd():
----> 2     x = x + 1
      3     return x
```

De ce se întâmplă asta? Funcția încearcă să creeze o variabilă nouă (vede că x nu a fost declarat local) și ajunge la un paradox

Variabile globale si variabile locale

- Toate variabilele create în afara unei funcții vor fi accesibile de către orice funcție din program, **chiar dacă nu o transmitem ca parametru**
- Totuși, atenția tot este necesară

```
def rcd():
    global x        ← Soluție
    x = x + 1
    return x
x = 7
print(rcd())
```

De ce se întâmplă asta? Funcția încearcă să creeze o variabilă nouă (vede că x nu a fost declarat local) și ajunge la un paradox

Output:

Variabile globale si variabile locale

- Toate variabilele create în afara unei funcții vor fi accesibile de către orice funcție din program, **chiar dacă nu o transmitem ca parametru**
- Totuși, atenția tot este necesară

```
def rcd():
    global x        ← Soluție
    x = x + 1
    return x
x = 7
print(rcd())
```

De ce se întâmplă asta? Funcția încearcă să creeze o variabilă nouă (vede că x nu a fost declarat local) și ajunge la un paradox

Output: 8

Variabile globale si variabile locale

- O variabilă locală devine accesibilă și pentru funcțiile din interiorul funcției.

Variabile globale si variabile locale

- O variabilă locală devine accesibilă și pentru funcțiile din interiorul funcției.

```
def f1():
    x = 7
    def f2():
        nonlocal x
        x = x + 1
        print(x)
    f2()
    print(x)
f1()
```

Output:

Variabile globale si variabile locale

- O variabilă locală devine accesibilă și pentru funcțiile din interiorul funcției.
- Aceste variabile se vor numi **nonlocal**

```
def f1():
    x = 7
    def f2():
        nonlocal x
        x = x + 1
        print(x)
    f2()
    print(x)
f1()
```

Output:

```
8
8
```

Transmiterea parametrilor

- Parametrii se transmit prin atribuire directă.
- Pentru parametrii imutabili, orice modificare în timpul rulării funcției nu se va reflecta asupra variabilei originale

Output:

```
29
29marian
('29marian', ['2', '9', 'm'
0
```

```
def g(x):
    x = x + 29
    print(x)
    x = str(x) + 'marian'
    print(x)
    x = (str(x), list(x))
    return x
x = 0
print(g(x))
print(x)
```

- Pentru parametrii mutabili, anumite instrucțiuni pot afecta valoarea variabilei originale (lucru nedorit de obicei)
- Soluții există pentru problema asta

Output:

```
[1, 2, 3]
[37, 3, 2, 1]
[37, 3, 2, 1]
```

```
def h(ls):
    ls.append(37)
    ls.reverse()
    print(ls)
ls = [1, 2, 3]
print(ls)
h(ls)
print(ls)
```

Transmiterea parametrilor

- Parametrii se transmit prin atribuire directă.
- Pentru parametrii imutabili, orice modificare în timpul rulării funcției nu se va reflecta asupra variabilei originale

- Pentru parametrii mutabili, anumite instrucțiuni pot afecta valoarea variabilei originale (lucru nedorit de obicei)
- Soluții există pentru problema asta

Output:

```
29
29marian
('29marian', ['2', '9', 'm'
0
```

```
def g(x):
    x = x + 29
    print(x)
    x = str(x) + 'marian'
    print(x)
    x = (str(x), list(x))
    return x
x = 0
print(g(x))
print(x)
```

Output:

```
[1, 2, 3]
[37, 3, 2, 1]
[1, 2, 3]
```

Locală ->

```
def h(ls):
    ls = ls.copy()
    ls.append(37)
    ls.reverse()
    print(ls)
ls = [1, 2, 3]
print(ls)
h(ls)
print(ls)
```

Funcții anonime (lambda)

- Unele funcții au o durată de viață foarte scurtă sau sunt prea simple pentru a-și justifica definirea cu eticheta ‘def’, așa că vor fi declarate local
- Le vom folosi des pentru funcția sorted()

Funcții anonime (lambda)

- Unele funcții au o durată de viață foarte scurtă sau sunt prea simple pentru a-și justifica definirea cu eticheta ‘def’, așa că vor fi declarate local
- Le vom folosi des pentru funcția sorted()

```
d = [['popescu', 9.56], ['ionescu', 6.44], ['andronescu', 8.37]]
print(sorted(d, key = lambda x: x[0]))
```

Output:

Funcții anonime (lambda)

- Unele funcții au o durată de viață foarte scurtă sau sunt prea simple pentru a-și justifica definirea cu eticheta ‘def’, așa că vor fi declarate local
- Le vom folosi des pentru funcția sorted()

```
d = [['popescu', 9.56], ['ionescu', 6.44], ['andronescu', 8.37]]
print(sorted(d, key = lambda x: x[0]))
```

Output:

```
[['andronescu', 8.37], ['ionescu', 6.44], ['popescu', 9.56]]
```

Funcții anonime (lambda)

- Unele funcții au o durată de viață foarte scurtă sau sunt prea simple pentru a-și justifica definirea cu eticheta ‘def’, așa că vor fi declarate local
- Le vom folosi des pentru funcția sorted()
- Exercițiu: Ordonați listele după inițiala șirului, iar dacă două liste au aceeași inițială, după ordine inversă a numerelor

```
d = [['apopescu', 9.56], ['ionescu', 6.44], ['andronescu', 9.57]]
```

Output:

Funcții anonime (lambda)

- Unele funcții au o durată de viață foarte scurtă sau sunt prea simple pentru a-și justifica definirea cu eticheta ‘def’, așa că vor fi declarate local
- Le vom folosi des pentru funcția sorted()
- Exercițiu: Ordonați listele după inițiala șirului, iar dacă două liste au aceeași inițială, după ordine inversă a numerelor

```
d = [['apopescu', 9.56], ['ionescu', 6.44], ['andronescu', 9.57]]
```

Output:

```
[['andronescu', 9.57], ['apopescu', 9.56], ['ionescu', 6.44]]
```

Funcții anonime (lambda)

- Unele funcții au o durată de viață foarte scurtă sau sunt prea simple pentru a-și justifica definirea cu eticheta ‘def’, așa că vor fi declarate local
- Le vom folosi des pentru funcția sorted()
- Exercițiu: Ordonați listele după inițiala șirului, iar dacă două liste au aceeași inițială, după ordine inversă a numerelor

Indiciu:tuplu

```
d = [['apopescu', 9.56], ['ionescu', 6.44], ['andronescu', 9.57]]
```

```
print(sorted(d, key = lambda x:
```

Output:

```
[['andronescu', 9.57], ['apopescu', 9.56], ['ionescu', 6.44]]
```

Funcții anonime (lambda)

- Unele funcții au o durată de viață foarte scurtă sau sunt prea simple pentru a-și justifica definirea cu eticheta ‘def’, așa că vor fi declarate local
- Le vom folosi des pentru funcția sorted()
- Exercițiu: Ordonați listele după inițiala șirului, iar dacă două liste au aceeași inițială, după ordine inversă a numerelor

```
d = [['apopescu', 9.56], ['ionescu', 6.44], ['andronescu', 9.57]]
```

```
print(sorted(d, key = lambda x: (x[0][0], -x[1])))
```

Output:

```
[['andronescu', 9.57], ['apopescu', 9.56], ['ionescu', 6.44]]
```

Recursivitate

Problema #1 - Fibonacci

Cerință: Să se scrie o funcție care returnează al n-lea termen al șirului Fibonacci unde n un număr natural nenul dat.

Șirul lui Fibonacci are următoarea definiție:

$$F(n) = \begin{cases} 0, & \text{dacă } n = 1 \\ 1, & \text{dacă } n = 2 \\ F(n-1) + F(n-2), & \text{altfel} \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Problema #1 - Fibonacci

Cerință: Să se scrie o funcție care returnează al n-lea termen al șirului Fibonacci unde n un număr natural nenul dat.

Î: Ce argument/argumente va avea funcția și ce va returna (dacă va returna ceva)?

Problema #1 - Fibonacci

Cerință: Să se scrie o funcție care returnează al n-lea termen al șirului Fibonacci unde n un număr natural nenul dat.

Î: Ce argument/argumente va avea funcția și ce va returna (dacă va returna ceva)?

R: argument: n = poziția din șirul fibonacci

return: numărul întreg aflat pe poziția n în șir

```
def fib(n):
    ...
    return termen
```

Problema #1 - Fibonacci

Vom arăta două soluții:

- Una **iterativă** (cu o structură repetitivă)
- Una ***recursivă***

Încercați să intuiți cum va arăta prima soluție (iterativă).

Î: Ce structură repetitivă vom folosi (for/while)?

Problema #1 - Fibonacci

Vom arăta două soluții:

- Una iterativă (cu o structură repetitivă)
- Una *recursivă*

Încercați să intuiți cum va arăta prima soluție (iterativă).

Î: Ce structură repetitivă vom folosi (for/while)?

R: Nu există o singură variantă corectă! Orice putem face cu un for, putem face cu un while.

-> Dar vom folosi un for totuși ☺

Problema #1 - Fibonacci

Iată soluția iterativă:

```
def fib(n):
    a = 0
    b = 1

    for _ in range(1, n):
        a, b = b, a + b

    return a

print(*[fib(x) for x in range(1, 15)])
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

Problema #1 - Fibonacci

Reamintim că a, b = b, b + a reprezintă o interschimbare.

Acum, varianta **recursivă** (= funcția se apelează singură).

-> Uitați-vă înapoi la definiția șirului Fibonacci și intuiți cum va arăta soluția.

Problema #1 - Fibonacci

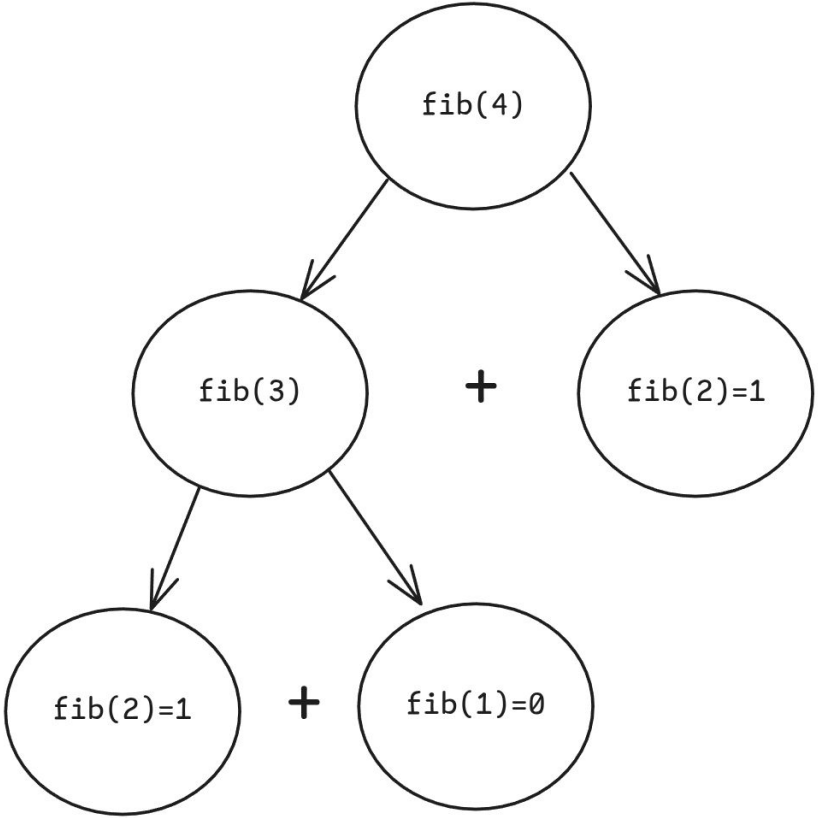
Soluția recursivă:

```
def fib_rec(n):
    if n == 1:
        return 0
    if n == 2:
        return 1

    return fib_rec(n - 1) + fib_rec(n - 2)

print(*[fib_rec(x) for x in range(1, 15)])
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

Problema #1 - Fibonacci

Cum se calculează fib(4):



Problema #1 - Fibonacci

Î: Ce algoritm credeți că este mai eficient ca timp de execuție? Cel iterativ (primul) sau cel recursiv (al doilea)?

Problema #1 - Fibonacci

Î: Ce algoritm credeți că este mai eficient ca timp de execuție? Cel iterativ (primul) sau cel recursiv (al doilea)?

R: primul este mai eficient; are complexitate O(n)

al doilea are complexitate $O(2^n)$

(veți auzi într-un curs viitor despre complexitatea algoritmilor)

Recursivitate

Recursivitatea este o tehnică de programare prin care o funcție se autoapelează pentru a rezolva o problemă, spărgând-o în subprobleme mai mici.

Recursivitate

O funcție recursivă trebuie să conțină:

- Condiția de oprire

```
def fib_rec(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
```

- Autoapelare (cu alte argumente)

```
return fib_rec(n - 1) + fib_rec(n - 2)
```

Problema #2 - Factorial

Cerință: Să se scrie o funcție recursivă care returnează n! (n factorial) pentru un n dat ca parametru.

Problema #2 - Factorial

Cerință: Să se scrie o funcție recursivă care returnează n! (n factorial) pentru un n dat ca parametru.

Idee: Știm că trebuie ca funcția să se autoapeleze. Deci,

factorial(n) = factorial(ceva), operații …

Problema #2 - Factorial

Cerință: Să se scrie o funcție recursivă care returnează n! (n factorial) pentru un n dat ca parametru.

Idee: Știm că trebuie ca funcția să se autoapeleze. Deci,

factorial(n) = factorial(ceva), operații …

factorial(n) = n * factorial(n-1)

$$n! = \begin{cases} 1, & \text{dacă } n \leq 1 \\ n \times (n-1)!, & \text{altfel} \end{cases}$$

Problema #2 - Factorial

Cerință: Să se scrie o funcție recursivă care returnează n! (n factorial) pentru un n dat ca parametru.

Idee: Știm că trebuie ca funcția să se autoapeleze. Deci,

factorial(n) = factorial(ceva), operații …

factorial(n) = n * factorial(n-1)

Î: Care-i condiția de oprire?

Hint: ne oprim când n = …

Problema #2 - Factorial

Cerință: Să se scrie o funcție recursivă care returnează n! (n factorial) pentru un n dat ca parametru.

Idee: Știm că trebuie ca funcția să se autoapeleze. Deci,

factorial(n) = factorial(ceva), operații …

factorial(n) = n * factorial(n-1)

Î: Care-i condiția de oprire?

R: **ne oprim când n <= 1 (ca în formula matematică)**

Problema #2 - Factorial

Soluție:

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

```
120
```

$$n! = \begin{cases} 1, & \text{dacă } n \leq 1 \\ n \times (n-1)!, & \text{altfel} \end{cases}$$

Problema #3 - Suma unei liste

Cerință: Să se scrie o funcție recursivă care returnează suma elementelor unei liste.

Exemplu: suma_listei([1, 2, 3, 4]) = 10

Problema #3 - Suma unei liste

Cerință: Să se scrie o funcție recursivă care returnează suma elementelor unei liste.

Exemplu: suma_listei([1, 2, 3, 4]) = 10

Idee: suma_listei([a, b, c, …]) = a + suma_listei([b, c, …])

Problema #3 - Suma unei liste

Cerință: Să se scrie o funcție recursivă care returnează suma elementelor unei liste.

Exemplu: suma_listei([1, 2, 3, 4]) = 10

Idee: suma_listei([a, b, c, …]) = a + suma_listei([b, c, …])

Î: Care-i condiția de oprire a funcției?

Problema #3 - Suma unei liste

Cerință: Să se scrie o funcție recursivă care returnează suma elementelor unei liste.

Exemplu: suma_listei([1, 2, 3, 4]) = 10

Idee: suma_listei([a, b, c, …]) = a + suma_listei([b, c, …])

R: ne oprim când lista e goală; verificăm asta prin

if ??? (completați)

Problema #3 - Suma unei liste

Cerință: Să se scrie o funcție recursivă care returnează suma elementelor unei liste.

Exemplu: suma_listei([1, 2, 3, 4]) = 10

Idee: suma_listei([a, b, c, …]) = a + suma_listei([b, c, …])

R: ne oprim când lista e goală; verificăm asta prin

if not lst SAU if lst == []

Iar suma va fi 0: suma_listei([]) == 0

Problema #3 - Suma unei liste

Cerință: Să se scrie o funcție recursivă care returnează suma elementelor unei liste.

Exemplu: suma_listei([1, 2, 3, 4]) = 10

Idee: **suma_listei([a, b, c, …]) = a + suma_listei([b, c, …])**

R: ne oprim când lista e goală; verificăm asta prin

if not lst SAU if lst == []

Iar suma va fi 0: suma_listei([]) == 0

Considerând lucrurile scrise cu portocaliu, stabiliți cum va arăta funcția.

Problema #3 - Suma unei liste

Soluție:

```
def suma_listei(lst):
    if not lst:
        return 0
    return lst[0] + suma_listei(lst[1:])

print(suma_listei([1, 2, 3, 4, 5, 6, 7]))
```

```
28
```

Problema #4 - print 1 to n

Cerință: Să se scrie o funcție recursivă care primește un parametru n nr. natural nenul și printează toate numerele de la 1 la n.

Problema #4 - print 1 to n

Cerință: Să se scrie o funcție recursivă care primește un parametru n nr. natural nenul și printează toate numerele de la 1 la n.

Î: Ce vrem ca funcția noastră să facă întâi: să printeze argumentul dat sau să se autoapeleze?

Problema #4 - print 1 to n

Cerință: Să se scrie o funcție recursivă care primește un parametru n nr. natural nenul și printează toate numerele de la 1 la n.

Î: Ce vrem ca funcția noastră să facă întâi: să printeze argumentul dat sau să se autoapeleze?

R: Vrem să se autoapeleze (să meargă întâi în adâncime), iar la întoarcere să printeze.

Problema #4 - print 1 to n

Cerință: Să se scrie o funcție recursivă care primește un parametru n nr. natural nenul și printează toate numerele de la 1 la n.

Soluție:

```
def print_1_to_n(n):
    if n == 0:
        return
    print_1_to_n(n - 1)
    print(n, end=" ")

print_1_to_n(10)
```

```
1 2 3 4 5 6 7 8 9 10
```

Căutarea binară

Căutarea binară este un mod eficient de a căuta o valoare într-o colecție *sortată* de valori.

Procedură:

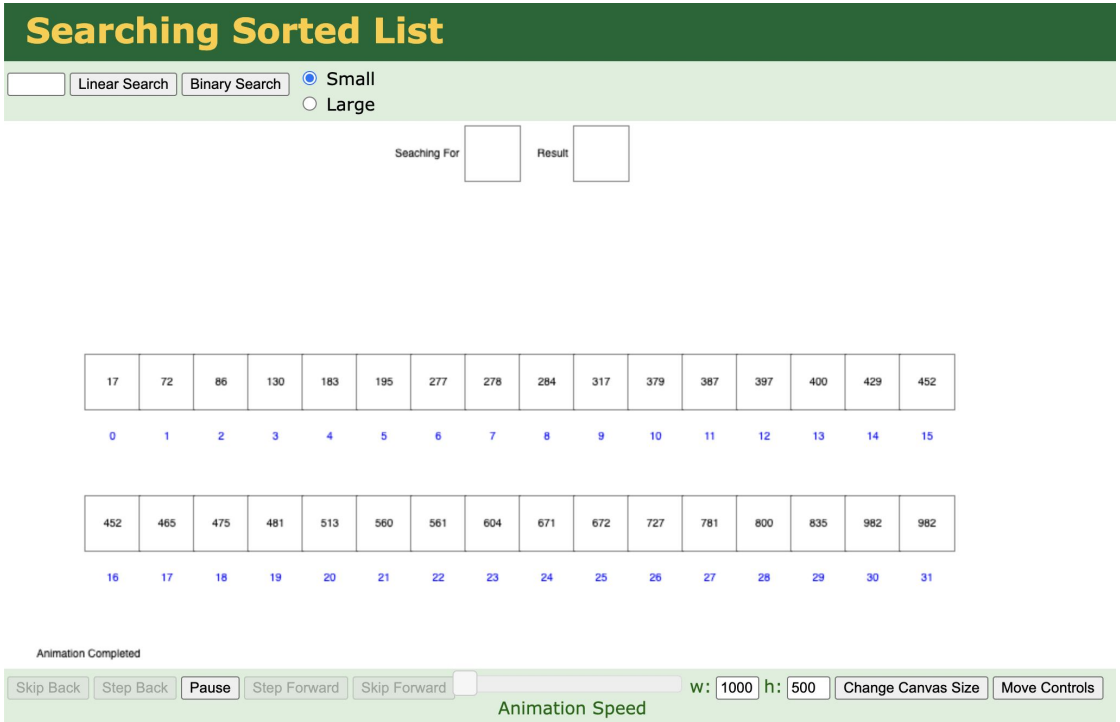
- comparăm valoarea căutată cu cea a elementului din mijlocul listei
- dacă valorile sunt egale, căutarea se încheie
- dacă valoarea căutată este mai mare, căutarea continuă în jumătatea din dreapta
- dacă este mai mică, se continuă în jumătatea din stânga.

Căutarea binară

Căutarea binară este un mod eficient de a căuta o valoare într-o colecție *sortată* de valori. **Iată algoritmul *iterativ*:**

```
def binary_search(lista, valoare):
    st = 0
    dr = len(lista) - 1
    while st <= dr:
        mij = st + (dr - st) // 2
        if lista[mij] == valoare:
            return mij
        elif lista[mij] < valoare:
            st = mij + 1
        else:
            dr = mij - 1

    return -1
```

Căutarea binară https://www.cs.usfca.edu/~galles/visualization/Search.html



Căutarea binară - recursiv

Hai să intuim cam cum ar arăta funcția pt. căutare binară folosind *recursivitate*.

Hint: seamănă mult cu varianta iterativă.

Căutarea binară - recursiv

Hai să intuim cam cum ar arăta funcția pt. căutare binară folosind *recursivitate*.

Hint: seamănă mult cu varianta iterativă.

```
def bin_search_rec(...):
    condiție de oprire
    calculăm mij
    dacă listă[mij] == valoare: return mij
    altfel dacă lista[mij] < valoare: caută la dreapta
    altfel: caută la stânga
```

Căutarea binară - recursiv

Am zis mai mai înainte că pentru scrierea funcțiilor recursive, trebuie să stabilim clar:

Î: Care va fi condiția de oprire pentru recursie?

Căutarea binară - recursiv

Am zis mai mai înainte că pentru scrierea funcțiilor recursive, trebuie să stabilim clar:

Î: Care va fi condiția de oprire pentru recursie?

R: st > dr

(caz în care nu am găsit valoarea căutată)

Căutarea binară - recursiv

Am zis mai mai înainte că pentru scrierea funcțiilor recursive, trebuie să stabilim clar:

Î: Care va fi condiția de oprire pentru recursie?

R: st > dr

(caz în care nu am găsit valoarea căutată)

Î: Ce returnăm în acest caz?

Căutarea binară - recursiv

Am zis mai mai înainte că pentru scrierea funcțiilor recursive, trebuie să stabilim clar:

Î: Care va fi condiția de oprire pentru recursie?

R: st > dr

(caz în care nu am găsit valoarea căutată)

Î: Ce returnăm în acest caz?

R: return -1 (return False nu e recomandat aici)

```python
def bin_search_rec(lista, valoare, st, dr):
    if st > dr:
        return -1

    mij = st + (dr - st) // 2

    if lista[mij] == valoare:
        return mij
    elif lista[mij] < valoare:
        return bin_search_rec(lista, valoare, mij + 1, dr)
    else:
        return bin_search_rec(lista, valoare, st, mij - 1)

v = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(bin_search_rec(v, 2, 0, len(v) - 1))
```

```
1
```