FUNCȚII RECURSIVE

În general, prin *recursivitate* se înțelege proprietatea unor noțiuni de a se defini prin ele însele. De exemplu, numerele naturale poate fi definite recursiv folosind următoarele două axiome ale lui Peano (https://en.wikipedia.org/wiki/Peano_axioms): "*Zero este un număr natural.*" şi "*Succesorul oricărui număr natural este tot un număr natural.*".

În programare, o *funcție recursivă* este o funcție care se autoapelează, direct sau indirect.

Deoarece recursivitatea indirectă (i.e., o funcție `f` apelează o funcție `g`, iar funcția `g` apelează, la rândul său, funcția `f`) este foarte rar utilizată în programare (de exemplu, pentru a calcula media aritmetico-geometrică: https://en.wikipedia.org/wiki/Arithmetic-geometric_mean), în continuare vom prezenta doar recursivitatea directă.

Un exemplu clasic de funcție recursivă îl reprezintă calculul factorialului unui număr natural n (i.e., $n! = 1 \cdot 2 \cdot \ldots \cdot n$), folosind următoarea relație de recurență:

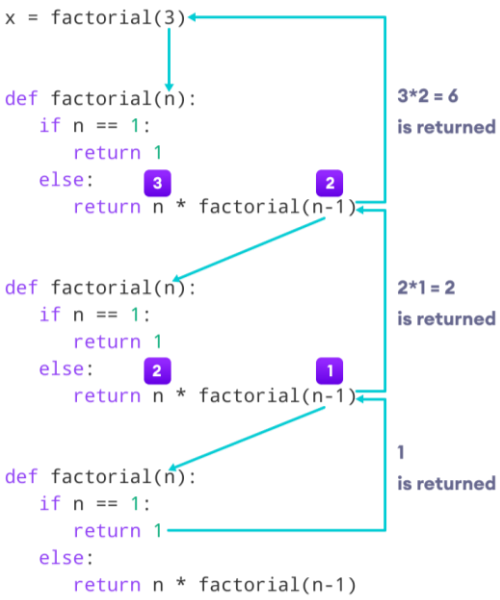
$$n! = \begin{cases} 1, & \text{dacă } n = 1 \\ n \cdot (n-1)!, & \text{dacă } n \geq 1 \end{cases}$$

O funcție care implementează în limbajul Python relația de recurență de mai sus este următoarea:

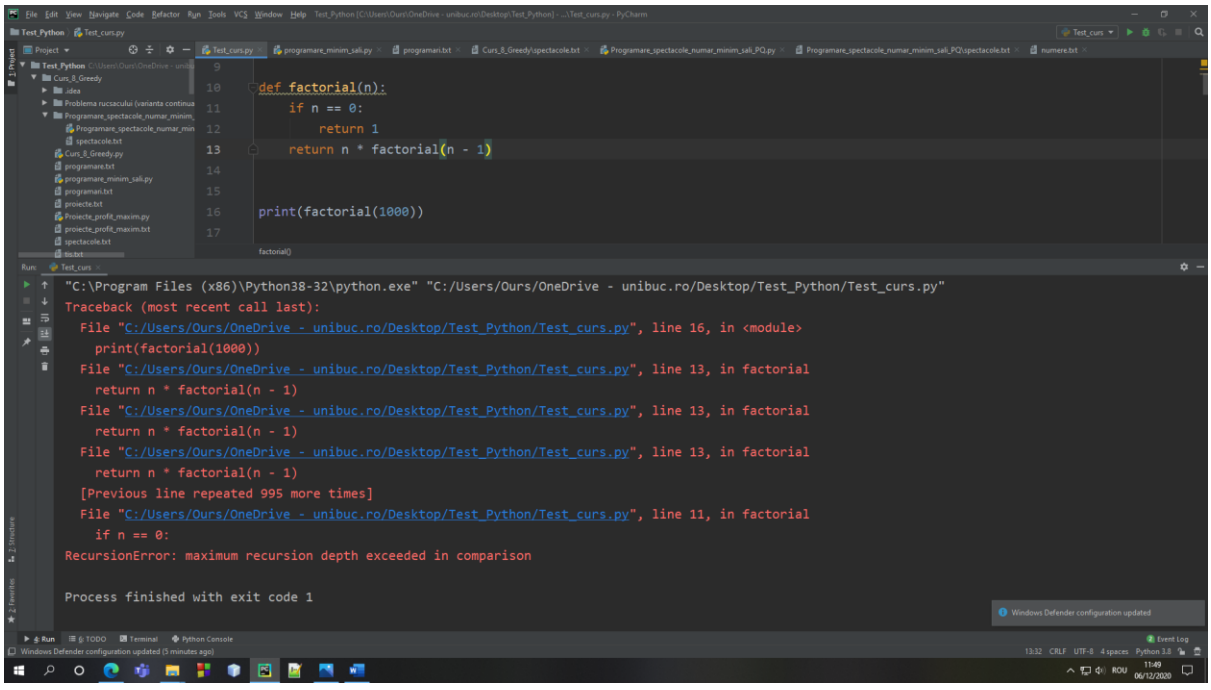
```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

În momentul apelării unei funcții, *contextul de apel* asociat (numele, variabilele locale, parametrii, adresa de revenire etc.) se salvează în stiva alocată programului, iar în momentul terminării executării sale, contextul de apel este eliminat din stivă.

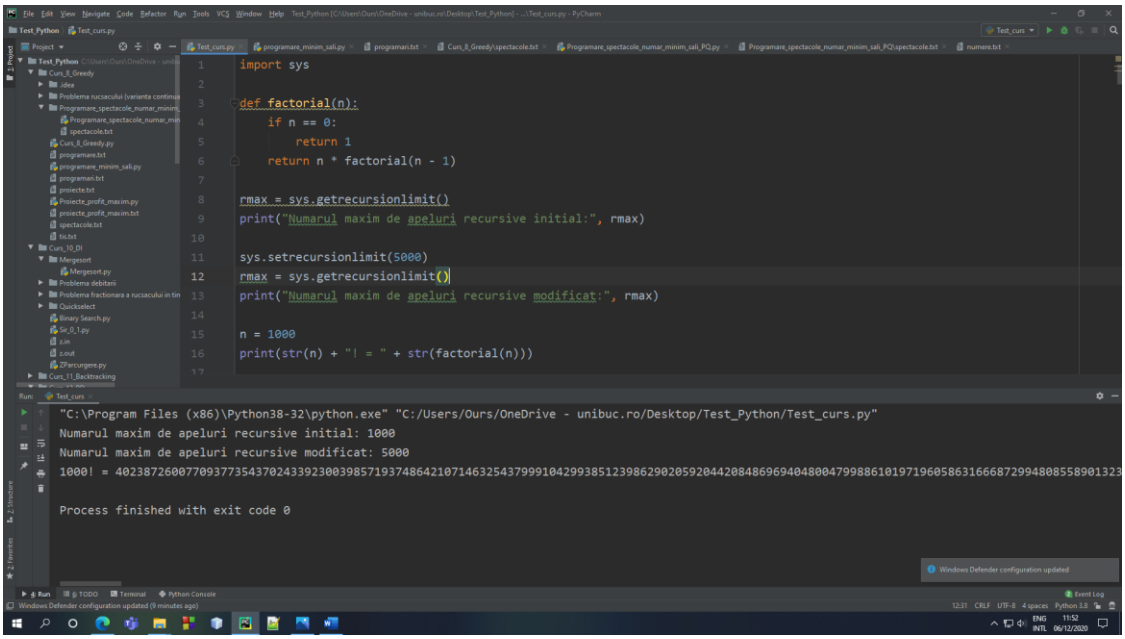
De exemplu, pentru apelul `f = factorial(3)`, stiva programului (reprezentată invers) va avea următoarea evoluție (sursa imaginii: https://www.programiz.com/python-programming/recursion):



Observaţie: Orice funcţie recursivă trebuie să conţină, pe lângă componenta recurentă (care conţine autoapelurile), şi o condiţie de oprire (care nu conţine niciun autoapel) realizabilă. În caz contrar, în momentul apelării sale, se va produce o recursivitate "infinită", care va conduce la depăşirea numărului maxim de apeluri recursive permis (implicit, acesta este egal cu 1000) şi la apariţia unei erori. De exemplu, în cazul apelului `f = factorial(1000)`, se va afişa următoarea eroare:



Numărul maxim de apeluri recursive care pot fi salvate pe stivă poate fi modificat astfel:



În continuare, vom prezenta câteva exemple clasice de funcţii recursive:

a) *Şirul lui Fibonacci* este definit prin următoarea relaţie de recurenţă:

$$f_n = \begin{cases} 0, & \text{dacă } n = 0 \\ 1, & \text{dacă } n = 1 \\ f_{n-2} + f_{n-1}, & \text{dacă } n \geq 2 \end{cases}$$

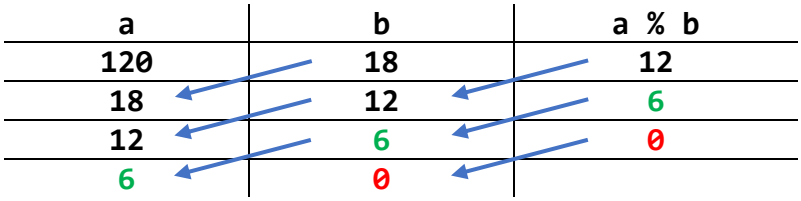
O implementare directă a acestei relaţii, care va furniza valoarea termenului de rang n a şirului lui Fibonacci, este următoarea funcţie recursivă:

```
def fibo(n):
    if n <= 1:
        return n
    return fibo(n-2) + fibo(n-1)
```

Pentru $n \geq 40$ se observă faptul că timpul de executare este destul de mare şi creşte în raport cu valoarea lui n. În capitolul următor, dedicat complexităţii computaţionale, se va demonstra faptul că numărul de apeluri recursive efectuate este exponenţial în raport cu n, aceasta fiind cauza timpului mare de executare. O implementare eficientă se poate obţine iterativ:

```
def fibo(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for i in range(n):
        a, b = b, a+b
    return a
```

b) *Algoritmul lui Euclid* permite determinarea celui mai mare divizor comun a două numere întregi nenule a şi b prin împărţiri repetate, respectiv: cât timp restul împărţirii lui a la b este nenul înlocuim a cu b şi b cu restul împărţirii lui a la b, iar ultimul rest nenul obţinut va fi $cmmdc(a, b)$. De exemplu, pentru $a = 120$ şi $b = 18$, cel mai mare divizor comun se va calcula astfel:



Ultimul rest nenul este egal cu 6, deci vom obţine $cmmdc(120, 18) = 6$. Se observă faptul că ultimul rest nenul este egal cu ultimul împărţitor, deci o variantă de implementare iterativă a acestui algoritm este următoarea:

```
def cmmdc(a, b):
    r = a % b
    while r != 0:
        a, b = b, r
        r = a % b
    return b
```

Analizând algoritmul, putem deduce foarte uşor următoarea formulă de recurenţa pentru calculul celui mai mare divizor comun a două numere întregi:

$$cmmdc(a, b) = \begin{cases} a, & \text{dacă } b = 0 \\ cmmdc(b, a\%b), & \text{dacă } b \neq 0 \end{cases}$$

Implementarea acestei relaţii de recurenţă printr-o funcţie recursivă este foarte simplă:

```
def cmmdc(a, b):
    if b == 0:
        return a
    return cmmdc(b, a%b)
```

Observăm faptul că funcţiile au complexităţi egale, respectiv numărul de iteraţii este aproximativ egal cu numărul de autoapeluri.

c) *Calculul sumei cifrelor unui număr natural* se poate realiza folosind următoarea relaţie de recurenţă:

$$sc(n) = \begin{cases} n, & \text{dacă } n < 10 \\ n\%10 + sc(n//10\ ), & \text{dacă } n \geq 10 \end{cases}$$

Implementarea acestei relații de recurență printr-o funcție recursivă este banală:

```
def sc(n):
    if n < 10:
        return n
    return n%10 + sc(n//10)
```

Ce relație există între numărul de autoapeluri din varianta recursivă și numărul de iterații din varianta iterativă?

d) *Suma elementelor dintr-o listă* se poate defini recursiv ca fiind suma dintre primul element al listei și suma elementelor din restul listei dacă lista este nevidă, respectiv 0 dacă lista este vidă:

```
def suma_lista(L):
    if len(L) == 0:
        return 0
    return L[0] + suma_lista(L[1:])
```

Într-un mod similar se poate calcula și suma elementelor strict pozitive dintr-o listă:

```
def sumapoz_lista(L):
    if len(L) == 0:
        return 0
    if L[0] > 0:
        return L[0] + sumapoz_lista(L[1:])
    else:
        return sumapoz_lista(L[1:])
```

e) *Frecvența unei litere într-un șir de caractere* se poate calcula recursiv astfel:

```
def frecventa(litera, sir):
    if len(sir) == 0:
        return 0
    return int(sir[0] == litera) + frecventa(litera, sir[1:])
```

f) Din numărul 4 se poate obține orice număr natural nenul aplicând în mod repetat următoarele operații:
1. împărțirea numărului la 2;
2. adăugarea cifrei 4 la sfârșitul numărului;
3. adăugarea cifrei 0 la sfârșitul numărului.

De exemplu, numărul 101 se poate obține prin șirul de operații $4 \to 40 \to 404 \to 202 \to 101$, iar numărul 133 prin șirul de operații $4 \to 2 \to 24 \to 12 \to 6 \to 64 \to 32 \to 16 \to 8 \to 84 \to 42 \to 424 \to 212 \to 106 \to 1064 \to 532 \to 266 \to 133$.

Pentru a afișa șirul de operații prin care se poate obține un număr natural nenul din numărul 4, vom aplica asupra sa operațiile inverse operațiilor date:
1'. înmulțirea numărului cu 2;
2'. eliminarea ultimei cifre, dacă aceasta este 4;
3'. eliminarea ultimei cifre, dacă aceasta este 0.

De exemplu, pentru numărul 101 vom obține următorul șir de operații $101 \rightarrow 202 \rightarrow 404 \rightarrow 40 \rightarrow 4$.

```
def numar4(n):
    if n != 4:
        if n % 10 == 0 or n%10 == 4:
            numar4(n//10)
        else:
            numar4(2*n)
        print(" ->", n, end="")
    else:
        print(4, end="")
```

Încheiem prezentarea funcțiilor recursive menționând faptul că, în general, acestea consumă multă memorie (pentru salvarea contextelor de apel), sunt mai greu de depanat decât funcțiile iterative și, de multe ori, necesită un timp de executare mai mare. Din aceste motive, se recomandă utilizarea unei funcții recursive doar în cazul în care complexitatea sa computațională este echivalentă cu cea a variantei iterative, dar implementarea sa este mai simplă.