CURS 7
Funcţii

Variabile locale şi globale

O variabilă definită în interiorul unei funcţii se numeşte *variabilă locală* şi este vizibilă (i.e., poate fi utilizată) doar în interiorul funcţiei respective.

O variabilă definită în afara oricărei funcţii se numeşte *variabilă globală* şi este vizibilă în tot modulul respectiv (i.e., poate fi utilizată în interiorul oricărei funcţii).

Exemplu:

```
def afisare():
    print("x = ", x)      # se va utiliza variabila globală x,
                          # deoarece nu există o variabilă locală x
x = 100
afisare()                 # x = 100
```

Dacă într-o funcţie există definită o variabilă locală având acelaşi nume cu o variabilă globală, atunci, implicit, se va utiliza variabila locală în interiorul funcţiei:

```
def afisare():
    x = 200
    print("x = ", x)        # se va utiliza variabila locală x

x = 100
afisare()                   # x = 200
print("x = ", x)            # x = 100 (variabila globală)
```

Dacă într-o funcţie există definită o variabilă locală având acelaşi nume cu o variabilă globală, atunci putem utiliza variabila globală precizând în interiorul funcţiei acest lucru:

```
def afisare():
    global x
    print("x = ", x)        # se va utiliza variabila globală x
    x = 200

x = 100
afisare()                   # x = 100
print("x = ", x)            # x = 200
```

Atenţie, dacă în exemplul de mai sus ar lipsi declararea `global x`, atunci ar fi generată eroarea `UnboundLocalError: local variable 'x' referenced before assignment`, deoarece definirea unei variabile locale `x` prin `x=200` va determina interpretatorul să nu

mai caute o variabilă globală cu numele x! Acelaşi lucru se va întâmpla şi în exemplul următor:

```
def f():
    x = x + 100             # eroare!
    print("x = ", x)

x = 200
f()
print("x = ", x)
```

În acest caz, eroarea apare deoarece interpretatorul consideră faptul că o instrucţiune de atribuire de forma "`x =`" reprezintă o declarare prin iniţializare a unei variabile locale `x`, dar, evident, expresia `x + 100` nu poate fi evaluată, variabila locală `x` nefiind iniţializată!

Funcţii imbricate

În limbajul Python putem defini o funcţie în interiorul altei funcţii, aşa cum se poate observa din exemplul următor:

```
def combinari(n, k):
    def factorial(x):
        p = 1
        for i in range(1, x+1):
            p = p * i
        return p

    return factorial(n) // (factorial(k) * factorial(n-k))

print(combinari(5, 3))
```

O funcţie `g` definită în interiorul unei funcţii `f` este locală funcţiei `f`, deci funcţia `g` poate fi apelată doar în interiorul funcţiei `f` (i.e., funcţia `g` este o funcţie auxiliară pentru `f`).

O funcţie imbricată are acces implicit la parametrii funcţiei în care este definită şi la variabilele sale locale:

```
def calcul(x, y):
    n = 2
    def medie(k):
        return (x**n + y**n) / k

    return medie(2)

print("m = ", calcul(3, 4))              # m = 12.5
```

Atenție, deși o funcție imbricată are acces implicit la variabilele locale ale funcției în care este definită, ea nu le poate modifica deoarece va fi generată o eroare de tipul `UnboundLocalError`. Pentru a utiliza o variabilă locală într-o funcție imbricată, variabila locală trebuie declarată în interiorul funcției imbricate folosind cuvântul cheie `nonlocal`:

```
def f():
    n = 100
    def aux():
        nonlocal n
        n = n * 2

    aux()
    return n

print(f())              # 200
```

Practic, prin utilizarea cuvântului cheie `nonlocal` pentru declararea unei variabile în interiorul unei funcții imbricate îi cerem interpretatorului să caute definirea variabilei respective în cel mai apropiat spațiu de nume exterior funcției imbricate, mai puțin în cel global (pentru a accesa variabilele globale se folosește cuvântul cheie `global`, așa cum deja am menționat):

```
incr = 100
def f():
    n = 7
    incr = 10

    def g():
        nonlocal incr, n
        n = n + incr
        return n

    def h():
        global incr
        nonlocal n
        n = n + incr
        return n

    print("nonlocal incr:", g())    # nonlocal incr: 17
    print("global incr:", h())      # global incr: 117

f()
```

În limbajul Python, o funcție poate să returneze o funcție imbricată, așa cum se poate observa din exemplul următor:

```
def putere(baza):


    def paux(exponent):
        return baza ** exponent

    return paux
putere10 = putere(10)

print(type(putere10))       # <class 'function'>
print(putere10.__name__)    # paux
print(putere10(3))          # 1000
```

Practic, `putere10` este o referință spre funcția `paux` particularizată pentru `baza = 10`, deci poate fi utilizată la fel ca orice altă funcție.

Transmiterea unei funcții ca parametru al altei funcții (callback)

Există mai multe situații în care este necesar să utilizăm *mecanismul de callback*, respectiv să transmitem o funcție `f` ca parametru al unei funcții `g`, astfel încât funcția `g` să poată apela funcția `f` când acest lucru este necesar. De exemplu, notificările utilizate în aplicațiile mobile utilizează un mecanism asemănător mecanismului de callback, respectiv o aplicație de tip server înregistrează faptul că un anumit utilizator a acceptat să primească notificări și în momentul în care apare un anumit eveniment pe server (de exemplu, când sunt depuși sau retrași bani dintr-un cont bancar), server-ul îi trimite utilizatorului respectiv o notificare. De asemenea, mecanismul de callback este intens utilizat în programarea interfețelor grafice, respectiv sistemul de operare primește o funcție pe care să o apeleze în momentul apariției unui anumit eveniment (e.g., apăsarea unui buton, închiderea unei ferestre, selectarea unei opțiuni dintr-o listă etc.).

Mecanismul de callback mai este utilizat și în *programarea generică*, respectiv în scrierea unor funcții care realizează prelucrări generice ale unei funcții a cărei expresie nu este cunoscută (e.g., reprezentarea grafică a unei funcții, calculul unei integrale etc.).

În continuare, vom prezenta o funcție generică pentru calculul unor sume. De exemplu, să considerăm următoarele 3 sume:

$$S_1 = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$$
$$S_2 = 1^2 + 2^2 + \cdots + n^2$$
$$S_3 = e^1 + e^2 + \cdots + e^n$$

Evident, putem să definim câte o funcţie pentru calculul fiecărei sume, dar această soluţie nu este scalabilă. Putem observa cu uşurinţă faptul că toate cele 3 sume au următoarea formă generală:

$$S_k = \sum_{i=1}^{n} f_k(i)$$

unde prin $f_k(i)$ am notat termenul de rang i al sumei S_k pentru $k \in \{1, 2, 3\}$, deci, pentru sumele de mai sus, termenii generali sunt $f_1(i) = \frac{1}{i}$, $f_2(i) = i^2$ şi $f_3(i) = e^i$. Astfel, putem scrie o funcţie generică pentru calculul unei astfel de sume, care va avea parametrii `n` şi `fk`, unde `fk` va fi o funcţie care implementează termenul general al unei anumite sume:

```
import math

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

def fk_1(i):
    return 1/i

def fk_2(i):
    return i**2


n = 10
s = suma_generica(n, fk_1)
print("Suma 1:", s)                 # Suma 1: 2.9289682539682538

s = suma_generica(n, fk_2)
print("Suma 2:", s)                 # Suma 2: 385

s = suma_generica(n, math.exp)
print("Suma 3:", s)                 # Suma 3: 34843.77384533132
```

Observaţi faptul că putem apela funcţia `suma_generica` utilizând pentru termenul general şi funcţii predefinite!

Funcţii anonime (lambda expresii)

O *funcţie anonimă* (*lambda expresie*) este o funcţie foarte simplă, fără nume, definită folosind cuvântul cheie lambda, astfel:

```
lambda parametrii: expresie
```

O funcţie anonimă poate să aibă unul sau mai mulţi parametrii, dar corpul său trebuie să fie format dintr-o singură expresie (e.g., nu poate conţine instrucţiuni sau mai multe expresii). În momentul apelării unei funcţii anonime, expresia asociată va fi evaluată, iar rezultatul obţinut va fi furnizat direct, fără a fi necesară utilizarea cuvântului cheie `return` în interiorul funcţiei anonime.

Exemple:

- *suma a două numere:* `lambda x, y: x+y`
- *testarea parităţii unui număr întreg:* `lambda x: x % 2 == 0`
- *testarea divizibilităţii:* `lambda x, y: False if y == 0 else x % y == 0`
- *suma cifrelor unui număr:* `lambda x: sum([int(cf) for cf in str(x)])`
- *numărul vocalelor dintr-un şir:* `lambda sir: len([lit for lit in sir if lit in "aeiouAEIOU"])`
- *numărul valorilor dintr-o listă* `L` *strict mai mari decât o valoare* `v`*:* `lambda L, v: len([x for x in L if x > v])`

O funcţie anonimă poate fi apelată direct, în momentul definirii sale, lucru care nu este posibil în cazul funcţiilor obişnuite:

```
print((lambda x, y: x+y)(5, 7))      # 12
print((lambda s: len([c for c in s if c in "aeiou"]))("examene"))# 4
print((lambda x, y: x % y == 0)(24, 6))      # True
```

O referinţă spre o funcţie anonimă poate fi atribuită unei variabile, pentru a fi utilizată ca o funcţie obişnuită, dar acest lucru nu se recomandă deoarece contrazice ideea de funcţie anonimă (se preferă definirea unei funcţii obişnuite, cu nume):

```
f = lambda x, y: x+y
s = f(5, 7)
print(f"s = {s}")    # s = 12
```

O funcţie poate returna o funcţie anonimă, permiţând astfel crearea unor funcţii particularizate în raport de anumite criterii:

```
def selectorFunctie(tip):
    if tip == "suma":
        return lambda x, y: x + y
    elif tip == "diferenta":
        return lambda x, y: x - y
    elif tip == "produs":
        return lambda x, y: x * y
    else:
        return None


f = selectorFunctie("suma")
s = f(5, 7)
print(f"s = {s}")   # s = 12
```

```
f = selectorFunctie("produs")
p = f(5, 7)
print(f"p = {p}")   # p = 35
```

O funcție anonimă poate fi utilizată în cadrul mecanismului de callback, pentru a elimina definițiile funcțiilor transmise ca parametrii (evident, dacă funcțiile respective sunt suficient de simple pentru a fi implementata ca nişte funcții anonime):

```
import math

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


n = 10
s = suma_generica(n, lambda x: 1 / x)
print("Suma 1:", s)                  # Suma 1:
2.9289682539682538

s = suma_generica(n, lambda x: x ** 2)
print("Suma 2:", s)                  # Suma 2: 385

s = suma_generica(n, math.exp)
print("Suma 3:", s)                  # Suma 3:
34843.77384533132
```

Încheiem prin a preciza faptul ca funcțiile anonime sunt utilizate, de obicei, în *programarea funcțională*, o altă paradigmă de programare implementată în limbajul Python, alături de programarea procedurală şi programarea orientată pe obiecte (https://realpython.com/python-functional-programming/).

Sortarea colecțiilor de date

În limbajul Python, colecțiile pot fi sortate crescător folosind funcția predefinită `sorted`, care furnizează o listă cu elementele colecției respective sortate crescător:

- `sorted([2, 1, 5, 2, 1, 4]) = [1, 1, 2, 2, 4, 5]`
- `sorted((2, 1, 5, 2, 1, 4)) = [1, 1, 2, 2, 4, 5]`
- `sorted({2, 1, 5, 2, 1, 4}) = [1, 2, 4, 5]`
- `sorted({"d": 2, "a": 1, "f": 0, "b": 3 }) = ['a', 'b', 'd', 'f']`

Observați faptul că, indiferent de tipul colecției sortate (i.e., listă, tuplu sau mulțime), rezultatul este furnizat sub forma unei liste, iar în cazul unui dicționar funcția va furniza doar o listă a cheilor dicționarului sortate crescător. Dacă dorim să obținem perechile

unui dicţionar sortate crescător în funcţie de chei, trebuie să sortăm o listă cu tuplurile corespunzătoare intrărilor sale:

```
sorted({"d": 2, "a": 1, "b": 3, "f": 0}.items()) = [('a', 1), ('b', 3), ('d', 2), ('f', 0)]
```

În cazul sortării unui şir de caractere, funcţia `sorted` va returna o listă formată din caracterele şirului, ordonate crescător:

```
sorted("exemplu") = ['e', 'e', 'l', 'm', 'p', 'u', 'x']
```

Dacă este necesar, putem să transformăm lista de caractere furnizată de metoda `sorted` înapoi într-un şir de caractere, folosind metoda `join`, astfel:

```
"".join(sorted("exemplu")) = "eelmpux"
```

Observaţie: O listă poate fi sortată direct, prin rearanjarea elementelor sale în ordine crescătoare, folosind metoda `sort` din clasa `list`:

```
L = [2, 1, 5, 2, 1, 4]
L.sort()
print("L =", L)             # L = [1, 1, 2, 2, 4, 5]
```

Dacă ulterior nu mai avem nevoie de lista iniţială, atunci se recomandă utilizarea metodei `sort` în locul funcţiei `sorted`, deoarece nu utilizează memorie suplimentară!

Deoarece funcţia `sorted` şi metoda `sort` din clasa `list` au aceiaşi parametrii opţionali, în continuare vom prezenta doar funcţia `sorted`, deoarece ea poate fi utilizată pentru orice structură de date iterabilă.

Pentru a realiza o sortare descrescătoare a elementelor unei structuri de date iterabile, parametrul opţional `reverse` al funcţiei `sorted` trebuie setat la valoarea `True`:

- `sorted([2, 1, 5, 2, 1, 4], reverse=True) = [5, 4, 2, 2, 1, 1]`
- `"".join(sorted("exemplu", reverse=True)) = "xupmlee"`

Evident, sortarea unei structuri de date iterabile se poate realiza doar în cazul în care elementele sale sunt comparabile, altfel se va genera o eroare de tipul `TypeError`. De exemplu, în cazul apelului `sorted([100, -10, "12345", 70])` se va genera eroarea `TypeError: '<' not supported between instances of 'str' and 'int'`!

Dacă o listă este formată din tupluri comparabile, atunci acestea vor fi sortate în ordine lexicografică. De exemplu, prin apelul `sorted([('g', 1), ('f', 7), ('b', 3), ('a', 2), ('f', 0), ('b', 1)])` se va obţine lista `[('a', 2), ('b', 1), ('b', 3), ('f', 0), ('f', 7), ('g', 1)]`, în care tuplurile au fost sortate în ordinea crescătoare a primelor componente, iar în cazul în care acestea erau egale în ordinea crescătoare a componentelor secundare. În cazul unei liste de şiruri de caractere, sortarea se va realiza folosind tot ordinea lexicografică. De exemplu, prin apelul

`sorted(["prune", "mere", "ananas", "pere", "mango"])` se va obține lista `["ananas", "mango", "mere", "pere", "prune"]`.

Pentru realizarea unor sortări complexe, eventual bazate pe mai multe criterii, putem să asociem fiecărui element al unei colecții o *cheie* pe bază căreia să se realizeze operația de sortare. Acest lucru se realizează folosind parametrul opțional `key` al funcției `sorted`, respectiv atribuindu-i acestuia numele unei funcții care asociază unui element al colecției cheia dorită. De exemplu, pentru a sorta crescător numerele dintr-o listă în funcție de sumele cifrelor lor, vom folosi pentru parametrul opțional `key` funcția `sumaCifre`, care calculează suma cifrelor unui număr natural `nr`:

```
def sumaCifre(nr):
    return sum([int(c) for c in str(nr)])

L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]
print(sorted(L, key=sumaCifre))
```

În urma executării programului, se va afișa următoarea listă (am evidențiat grupurile de numere care au aceeași sumă a cifrelor):

$$[\underbrace{101}_{2}, \underbrace{30, 111, 12}_{3}, \underbrace{202}_{4}, \underbrace{71, 107}_{8}, \underbrace{27, 81, 18}_{9}]$$

Practic, funcția `sumaCifre` a fost apelată pentru fiecare element al listei înainte ca acesta să fie comparat cu alte elemente, iar valoarea obținută (cheia elementului) a fost utilizată în comparări în locul numărului respectiv!

Deoarece funcția `sorted` implementează *o metodă de sortare stabilă*, în lista sortată se va păstra ordinea relativă din lista inițială a elementelor cu chei egale. De exemplu, în lista inițială, numărul 27 se afla înaintea numărului 81, iar numărul 81 se afla înaintea numărului 18, iar această ordine relativă este păstrată și în lista sortată.

Putem rescrie mai concis secvența de cod de mai sus, utilizând o funcția anonimă în locul funcției `sumaCifre`:

```
L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]
print(sorted(L, key=lambda n: sum([int(c) for c in str(n)])))
```

Dacă dorim să sortăm numerele din listă în ordinea crescătoare a sumelor cifrelor lor, iar în cazul în care sumele cifrelor sunt egale să le sortăm crescător după valorile lor, atunci vom asocia fiecărui număr un tuplu format din suma cifrelor sale și el însuși:

```
def sumaCifre(nr):
    sc = sum([int(c) for c in str(nr)])
    return sc, nr

L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]

print("L =", sorted(L, key=sumaCifre))
```

Astfel, se va afișa lista `[101, 12, 30, 111, 202, 71, 107, 18, 27, 81]`, deoarece cheile asociate elementelor listei inițiale `[30, 27, 111, 71, 101, 107, 12, 81, 202, 18]` au fost tuplurile `(3, 30)`, `(9, 27)`, `(3, 111)`, `(8, 71)`, `(2, 101)`, `(8, 107)`, `(3, 12)`, `(9, 81)`, `(4, 202)`, `(9, 18)`, iar sortarea elementelor listei a fost realizată comparând lexicografic aceste tupluri!
Putem rescrie mai concis secvența de cod de mai sus, utilizând o funcție anonimă care furnizează un tuplu, în locul funcției `sumaCifre`:

```
L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]
print(sorted(L, key=lambda n: (sum([int(c) for c in str(n)]), n)))
```

Observație: Pentru a sorta descrescător o colecție în funcție de o cheie numerică `k` este suficient să folosim cheia `-k`. De exemplu, pentru a sorta numerele dintr-o listă în ordinea crescătoare a sumelor cifrelor lor, iar în cazul în care sumele cifrelor sunt egale să le sortăm descrescător după valorile lor, atunci vom utiliza următoarea funcție pentru chei:

```
def sumaCifre(nr):
    sc = sum([int(c) for c in str(nr)])
    return sc, -nr
```

Funcția utilizată pentru a calcula cheia unui element poate fi și o funcție predefinită. De exemplu, putem utiliza funcția predefinită `len` pentru a sorta o listă de șiruri de caractere în ordinea crescătoare a lungimilor lor:

```
L = ["prune", "pere", "ananas", "mere", "mango"]
print(sorted(L, key=len))   # ['pere', 'mere', 'prune', 'mango', 'ananas']
```

În continuare, vom mai prezenta câteva exemple de sortări complexe:

a) Să se sorteze o listă de numere naturale astfel încât numerele pare sortate crescător să fie poziționate înaintea celor impare sortate descrescător.

   Pentru a realiza această sortare, vom asocia fiecărui număr natural `nr` cheia `(0, nr)` dacă el este par, respectiv cheia `(1, -nr)` dacă el este impar. Practic, prima componentă a cheii este chiar restul împărțirii numărului `nr` la 2 (i.e., paritatea sa), iar cea de-a doua componentă este utilizată pentru a sorta crescător sau descrescător numerele cu aceeași paritate!

```
def paritate(nr):
    return nr % 2, nr if nr % 2 == 0 else -nr


L = [int(x) for x in input("Lista: ").split()]
L = sorted(L, key=paritate)
print("Lista sortata:", L)
```

   De exemplu, dacă lista inițială este `[52, 27, 111, 71, 101, 17, 107, 12, 18]`, atunci, după sortare, se va obține lista `[12, 18, 52, 111, 107, 101, 71, 27, 17]`.

Putem utiliza o funcție anonimă în locul funcției `paritate`:

```
L = [int(x) for x in input("Lista: ").split()]
L = sorted(L, key=lambda n: (0, n) if n % 2 == 0 else (1, -n))
print("Lista sortata:", L)
```

b) Considerăm o listă care conține informații despre mai mulți studenți, respectiv pentru fiecare student se cunoaște numele, grupa și nota obținută la examenul de admitere. Să se sorteze studenții în ordinea crescătoare a grupelor, în fiecare grupă studenții să fie sortați descrescător după nota obținută la examenul de admitere, iar în cazul unor note egale studenții să fie sortați alfabetic.

Vom considera faptul că informațiile despre fiecare student sunt memorate într-un tuplu, astfel:

```
L = [( "Popescu Ion",  131, 9.25),
     ( "Ionescu Ana",  133, 8.75),
     ( "Popa Marian",  131, 9.85),
     ( "David Maria",  132, 8.95),
     ("Gheorghe Ana",  131, 9.85),
     ("Popescu Anca",  132, 9.15),
     ("Corbu Florin",  133, 8.05),
     ("Gheorghe Dan",  132, 9.15)]
```

Cheia asociată unui student va consta din componentele tuplului respectiv, în ordinea specificată în criteriile de sortare:

```
def cheie_student(t):
    return t[1], -t[2], t[0]

S = sorted(L, key=cheie_student)
print(*S, sep="\n")
```

După sortarea listei, studenții vor fi afișați în următoarea ordine:

```
("Gheorghe Ana",  131, 9.85)
( "Popa Marian",  131, 9.85)
( "Popescu Ion",  131, 9.25)
("Gheorghe Dan",  132, 9.15)
("Popescu Anca",  132, 9.15)
( "David Maria",  132, 8.95)
( "Ionescu Ana",  133, 8.75)
("Corbu Florin",  133, 8.05)
```

Putem simplifica secvența de cod utilizând o funcție anonimă în locul funcției `cheie_student`:

```
S = sorted(L, key=lambda t: (t[1], -t[2], t[0]))
print(*S, sep="\n")
```

c) Să se sorteze şirurile de caractere dintr-o listă `L` în ordinea descrescătoare a lungimilor prefixelor maximale comune cu un şir dat `s`. De exemplu, dacă lista este `L = ["apasator", "apartament", "exemplu", "ars", "test", "aparat", "amic"]` şi `s = "aparator"`, atunci lista sortată va fi `L = ["aparat", "apartament", "apasator", "ars", "amic", "exemplu", "test"]`.

Pentru a determina prefixul maximal comun a două şiruri, vom considera, pe rând, toate prefixele unuia dintre şiruri, în ordinea descrescătoare a lungimilor, şi vom verifica dacă el este prefix şi pentru cel de-al doilea şir:

```
def prefixMaxim(s, t):
    for i in range(len(s), 0, -1):
        if t.startswith(s[:i]):
            return i
    return 0
```

Evident, funcţia se poate optimiza din punct de vedere al complexităţii computaţionale considerând prefixele celui mai scurt şir dintre cele două!

Din păcate, funcţia `prefixMaxim` are 2 parametri, deci nu o putem utiliza pentru a furniza cheile asociate şirurilor din listă (funcţia ar trebui să aibă un singur parametru, respectiv un element al listei)! Totuşi, putem să rezolvăm această problemă folosind funcţii imbricate, respectiv definim funcţia `prefixMaxim` cu un singur parametru, şirul dat `s`, şi în interiorul său definim o funcţie auxiliară `pmax` cu un singur parametru `t`, care va determina prefixul maximal comun dintre şirurile `s` şi `t`, iar funcţia `prefixMaxim` va returna funcţia `pmax`:

```
def prefixMaxim(s):
    def pmax(t):
        for i in range(len(s), 0, -1):
            if t.startswith(s[:i]):
                return i, t
        return 0, t

    return pmax
```

Astfel, prin apelul `prefixMaxim(s)`, vom obţine o funcţie cu un singur parametru, care va determina lungimea prefixului comun maximal dintre un şir oarecare `t` (parametrul funcţiei) şi şirul dat `s`, iar această funcţie poate fi utilizată pentru a furniza cheia asociată unui element:

```
L = ["apasator", "apartament", "exemplu", "ars", "test",
     "aparat", "amic"]
s = "aparator"

L.sort(key=prefixMaxim(s), reverse=True)
print(L)
```

Cum ar trebui să modificăm funcţia `prefixMaxim` astfel încât şirurile având aceeaşi lungime a prefixului comun maximal să fie sortate alfabetic? În acest exemplu, nu putem înlocui funcţia `prefixMaxim` cu o funcţie anonimă. De ce?

În încheiere, menţionăm faptul că funcţia `sorted` şi metoda `sort` implementează algoritmul de sortare *Timsort*, care a fost creat special în 2002 de Tim Peters pentru a fi implementat în limbajul Python (https://en.wikipedia.org/wiki/Timsort). Algoritmul *Timsort* este derivat din algoritmii de sortare prin interclasare (*Mergesort*) şi sortare prin inserţie (*Insertion sort*), având complexitatea $\mathcal{O}(n\log_2 n)$ în cazul cel mai defavorabil.

Generatori

Un *generator* este un tip de funcţie a cărei executare nu se termină în momentul în care returnează o valoare, ceea ce îi permite furnizarea mai multor valori într-o manieră secvenţială (i.e., sub forma unui *iterator*). Un exemplu de generator predefinit în limbajul Python este `range`, care furnizează valorile întregi dintr-un anumit interval una câte una. Pentru a returna valori în mod repetat, un generator foloseşte instrucţiunea `yield` în locul instrucţiunii `return`:

```
def generator_numere_pare(n):
    x = 0
    while x <= n:
        yield x
        x += 2
```

Valorile furnizate de un generator pot fi accesate secvenţial, folosind, de exemplu, o instrucţiune `for`:

```
for x in generator_numere_pare(100):
    print(x, end=" ")
```

O altă posibilitate de accesare secvenţială a valorilor furnizate de un generator o reprezintă utilizarea funcţiei `next(iterator, [valoare_implicită])`, care permite, în general, accesarea următoarei valori dintr-un iterator sau furnizarea unei valori implicite, precizată prin parametrul opţional `valoare_implicită`, în momentul în care iteratorul nu mai conţine nicio valoare:

```
nr_pare = generator_numere_pare(100)
x = next(nr_pare, -1)
while x != -1:
    print(x, end=" ")
    x = next(nr_pare, -1)
```

Practic, pentru a putea furniza mai multe valori într-o manieră secvenţială, contextul de apel al unui generator nu este eliminat de pe stiva programului în momentul executării unei instrucţiuni `yield`. Astfel, după revenirea dintr-un apel, un generator îţi poate continua executarea din starea în care se afla înaintea apelului respectiv!

Instrucţiunea `return` vidă poate fi utilizată pentru a întrerupe executarea unui generator:

```
def generator_numere_pare(n):
    x = 0
    while True:
        yield x
        if x == n:
            return
        x += 2
```

Un generator poate rula "la infinit", aşa cum se poate observa din următorul exemplu:

```
def generator_numere_pare():
    x = 0
    while True:
        yield x
        x = x + 2
```

Evident, în acest caz nu se pot utiliza modalităţile de accesarea a valorilor furnizate menţionate anterior, ci accesarea acestora trebuie să fie întreruptă explicit:

```
for x in generator_numere_pare():
    print(x, end=" ")
    if x == 100:
        break
```

```
nr_pare = generator_numere_pare()
x = next(nr_pare, -1)
while x <= 100:
    print(x, end=" ")
    x = next(nr_pare, -1)
```

Întreruperea accesării valorilor furnizate de un generator infinit nu va conduce la oprirea sa, aşa cum se poate observa din următorul exemplu (se vor afişa numerele pare de la 0 la 20):

```
nr_pare = generator_numere_pare()
for x in nr_pare:
    print(x, end=" ")
    if x == 10:
        break

for x in nr_pare:
    print(x, end=" ")
    if x == 20:
        break
```

Pentru a întrerupe forţat executarea unui generator, infinit sau nu, trebuie apelată metoda `close` (se vor afişa doar numerele pare de la 0 la 10):

```
nr_pare = generator_numere_pare()
for x in nr_pare:
    print(x, end=" ")
```

```
    if x == 10:
        break

nr_pare.close()

for x in nr_pare:
    print(x, end=" ")
    if x == 20:
        break
```

În concluzie, un generator reprezintă o modalitate foarte simplă de creare a unui iterator în limbajul Python. În plus, generatorii permit o utilizare eficientă a memoriei, valorile fiind furnizate secvențial.