



Școala
informală
de IT

Python Development

Memory savers & files



Școala
informală
de IT

Cuprins

1. Memory savers
 - a. funcții lambda
 - b. funcția map
 - c. funcția filter
 - d. funcția zip
 - e. list comprehension
2. Lucrul cu fișiere



Memory savers

Memory savers & files



Memory savers

- În acest capitol ne vom juca cu câteva funcții și metode de a salva timp și memorie atunci când dezvoltăm o aplicație folosind Python.
- Chiar dacă memoria este gestionată de Python Memory Management, este de datoria noastră, ca developeri, să nu consumăm memoria inutil și să scriem cod astfel încât acesta să consume cât mai puține resurse și să fie cât mai eficient.
- Pentru a îndeplini această sarcină trebuie să știm de existența următoarelor noțiuni:
 - funcții **lambda**
 - funcția **map**
 - funcția **filter**
 - funcția **zip**
 - list **comprehension**



Memory savers - funcții lambda

- O funcție lambda este o funcție anonimă care îndeplinește o singură instrucțiune.
- Acest tip de funcție poate primi orice număr de parametri, dar poate executa o singură instrucțiune.
- Sintaxa este următoarea:

```
my_lambda = lambda x, y: x + y

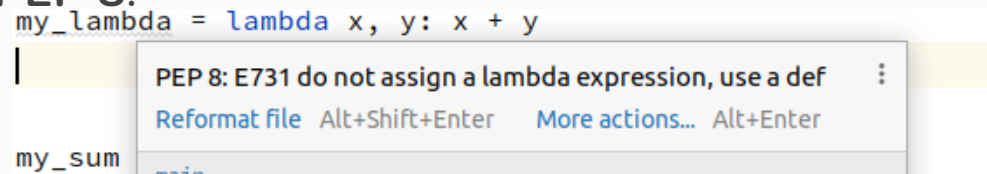
my_sum = my_lambda(2, 4)
print('my_sum =', my_sum)
```

- pentru folosirea unei funcții lambda avem nevoie să folosim keyword-ul lambda
- keyword-ul este urmat de lista de parametri - spre deosebire de o funcție normală se poate observa că parametri sunt doar separați prin virgulă, fără a folosi paranteze rotunde ().
- instrucțiunea ce urmează a fi rulată se află după caracterul :.



Memory savers - funcții lambda

- Deși exemplul anterior funcționează perfect, în consolă fiind afișată suma numerelor, codul nu este în conformitate cu standardul **PEP 8**.



- Conform PEP 8, o funcție lambda nu se asignează. Dacă vrem să etichetăm (numim / definim) o funcție trebuie să folosim o funcție normală, definită cu ajutorul keyword-ului **def**.
- Tocmai aici este avantajul în a folosi funcții lambda în detrimentul funcțiilor clasice. Odată definită o funcție aceasta va ocupa locație în memorie, în timp ce rolul unei funcții lambda este exact acela de a salva memoria și a nu o ocupa inutil.
- Cu alte cuvinte, o funcție lambda ocupă memorie doar în momentul rulării și este ștearsă imediat după.

Memory savers - funcții lambda

- Această afirmație poate fi demonstrată ușor dacă vom folosi funcția `id()` și verificăm valoarea returnată de aceasta pentru o funcție lambda asignată (variantă nerecomandată) și folosirea unei funcții lambda de sine stătătoare (variantă recomandată).

```
a = lambda: 1

print(f'id(a) = {id(a)}')
print(f'unassigned lambda: {id(lambda: 2)}')
print(f'id(a) = {id(a)}')
print(f'unassigned lambda: {id(lambda: 2)}')
print(f'id(a) = {id(a)}')
print(f'unassigned lambda: {id(lambda: 3)}')
print(f'id(a) = {id(a)}')
```

```
id(a) = 139780462704840
unassigned lambda: 139780462704160
id(a) = 139780462704840
unassigned lambda: 139780462704160
id(a) = 139780462704840
unassigned lambda: 139780462704160
id(a) = 139780462704840
```

- În exemplul anterior se poate observa că funcția asignată lui `a` își păstrează locația de memorie pe tot parcursul programului, în timp ce restul funcțiilor lambda ocupă, rând pe rând, aceeași locație de memorie.



Memory savers - funcții lambda

- Un exemplu elocvent de avantaj în folosirea funcțiilor lambda o reprezintă sortarea unei liste de forma:

```
players = [{  
    "first_name": "John",  
    "last_name": "Doe",  
    "rank": 3  
}, {  
    "first_name": "Kevin",  
    "last_name": "McDonald",  
    "rank": 1  
}, {  
    "first_name": "Bradd",  
    "last_name": "Kelvin",  
    "rank": 2  
}]
```

- În acest exemplu urmărim să sortăm această listă în funcție de proprietatea **rank** a fiecărui player.
- Pentru a realiza acest lucru putem folosi două metode:
 - list.sort()** care va face sortarea in-place - lista inițială va fi modificată
 - sorted(list)** care va returna o nouă listă sortată.



Memory savers - funcții lambda

- Ambele funcții dispun de un parametru numit **key** al cărui rol este să specificăm cheia după care să se facă sortarea.
- Acest parametru trebuie să fie o funcție ce va fi rulată O SINGURĂ dată pentru fiecare valoare întâlnită în listă.
- Pentru a evita definirea unei funcții pentru a o trimite ca parametrul key, și să ocupe memoria până la terminarea

programului, putem folosi o funcție lambda astfel:

```
sorted_players = sorted(players, key=lambda player: player["rank"])  
print(sorted_players)
```

- În momentul acesta ne-am folosit, într-adevăr, de avantajele oferite de funcțiile lambda,
- Acest lucru o să fie confirmat și de PyCharm prin intermediul standardului PEP 8 - nu o să mai apară nici un warning la utilizarea funcției.



Memory savers - funcția map

- Funcția map are rolul de a modifica fiecare element al unei liste.
- Sintaxa acesteia este:

```
players_with_top_3_value = map(check_top_3_player, players)
```

- primul parametru primit de funcție este o funcție care primește fiecare element din iterabil, pe rând, și trebuie să întoarcă un alt element pe baza acestuia.
 - al doilea parametru este un iterabil pe care vrem să acționeze map-ul.
- Având în vedere lista noastră anterioară de jucători, în exemplul următor ne folosim de funcția map pentru a adăuga fiecărui element din listă proprietatea *is_top_3* care va fi True sau False în funcție de rank.

```
def check_top_3_player(player):  
    updated_player = copy.deepcopy(player)  
    updated_player["is_top_3"] = True if updated_player["rank"] <= 3 else False  
    return updated_player  
  
players_with_top_3_value = map(check_top_3_player, players)  
print('players_with_top_3_value =', list(players_with_top_3_value))
```

- Atenție! Rezultatul returnat de **map** este un iterabil de tip **map**.

```
print(type(players_with_top_3_value)) # <class 'map'>
```



Memory savers - funcția filter

- Funcția filter are rolul de a filtra elementele dintr-un iterabil.
- Sintaxa acesteia este:

```
all_mcdonalds = filter(filter_all_mcdonalds, players)
```

- primul parametru primit de funcție este o funcție care primește fiecare element din iterabil, pe rând, și returnează True dacă acesta va face parte din secvența finală sau False în caz contrar.
- al doilea parametru este un iterabil pe care vrem să acționeze iterabilul.
- Având în vedere lista anterioară de jucatori, în exemplul următor ne vom folosi de funcția filter pentru a obține un iterabil doar cu jucătorii a căror proprietate *last_name* are valoarea *McDonald*:

```
def filter_all_mcdonalds(player):  
    if player["last_name"] == "McDonald":  
        return True  
  
    return False  
  
all_mcdonalds = filter(filter_all_mcdonalds, players)  
print('all_mcdonalds', list(all_mcdonalds))
```

- ⇒ Pentru a beneficia la maxim de eficientizarea memoriei, funcția folosită în exemplul alăturat poate fi redusă la o funcție anonimă care va produce exact același rezultat.

```
all_mcdonalds = filter(lambda player: True if player["last_name"] == "McDonald" else False, players)  
print('all_mcdonalds', list(all_mcdonalds))
```

- Atenție! Rezultatul returnat de **filter** este un iterabil de tip **filter**.

```
print(type(all_mcdonalds))
```



Memory savers - funcția zip

- Funcția zip primește două sau mai multe structuri iterabile și returnează un iterabil de tip **zip** format din tuple-uri care conțin elemente grupate din structurile inițiale.
- Având în vedere că structurile inițiale pot avea lungimi diferite, lungimea finală a iterabilului rezultat în urma funcției zip va avea lungimea egală cu lungimea celei mai scurte structuri inițiale.
- Teoria poate fi cel mai bine evidențiată prin următorul exemplu:

```
for zip_item in zip(list_1, list_2, list_3):  
    # print(zip_item) - va afișa, pe rand, (1, 10, 100), (2, 20, 200), (3, 30, 300)  
    list_1_element, list_2_element, list_3_element = zip_item  
    print(list_1_element, list_2_element, list_3_element)
```

- din câte se observă, structurile iterabile sunt trimise ca parametrii poziționali funcției zip. Ordinea în care sunt trimiși va reprezenta ordinea în tuple-urile inițiale.
- în exemplul anterior avem de a face cu 3 liste de dimensiuni diferite astfel că lungimea iterabilului returnat de funcția zip va avea lungimea egală cu 3 (lungimea listei **list_1** - aceasta fiind cea mai scurtă)



Memory savers - list comprehension

- List comprehension este un concept folosit pentru obținerea unei liste noi pe baza unei liste deja existente.

```
my_numbers = [1, 2, 3, 4, 5]
squared_numbers = [item ** 2 for item in my_numbers]
print('my_numbers', my_numbers)
print('squared_numbers', squared_numbers)
```

- Sintaxa list comprehension se bazează pe iterarea unei liste existente și crearea unei liste noi.
- În această metodă poate fi folosit și un inline if, ca în exemplul următor:

```
my_numbers = [1, 2, 3, 4, 5]
even_squared_numbers = [item ** 2 for item in my_numbers if item % 2 == 0]
print('my_numbers', my_numbers)
print('squared_numbers', even_squared_numbers)
```



Lucrul cu fișiere

Memory savers & files



Lucrul cu fișiere

- Atunci când vorbim de Python și lucrul cu fișiere ne putem gândi la cam tot ce ne trece prin cap despre fișiere:
 - putem să “ne plimbăm” prin structura de directoare
 - putem să creăm și/sau să ștergem, directoare și sau fișiere
 - putem arhiva și dezarhiva fișiere .zip și/sau .tar.
 - ș.a.
- Pentru lucrul cu fișierele există mai multe module built-in care ne pot ajuta, cum ar fi:
 - os
 - os.path
 - shutil
 - pathlib



Lucrul cu fișiere

- Pentru deschiderea unui fișier vom folosi instrucțiunea **with**.
- Această instrucțiune este folosită pentru gestionarea excepțiilor astfel încât codul să fie mai lizibil. Are rolul de a simplifica gestionarea resurselor comune cum ar fi fluxul de fișiere.
- În exemplul de mai jos puteți observa modalitatea de deschidere a unui fișier folosind metoda clasică, fără folosirea instrucțiunii with:

```
file = open('data.txt', 'w')  
file.write('hello, world!')  
file.close()
```

```
file = open('data.txt', 'w')  
try:  
    file.write('hello, world!')  
finally:  
    file.close()
```

- Următorul exemplu este echivalent cu cele prezentate mai sus, singura diferență fiind eficientizarea codului prin neapelarea manuală a metodei **close()**.

```
with open('data.txt', 'w') as file:  
    file.write('hello, world!')
```

- Folosind instrucțiunea with, Python se va ocupa de gestionarea fluxului necesar operațiilor cu fișiere.



Lucrul cu fișiere

- Funcția **open()** primește următorii parametri:
 - (string) → reprezintă numele fișierului ce urmează a fi deschis.
 - (string - opțional) → reprezintă modul în care fișierul va fi deschis:
 - ⇒ **r** - deschide fișierul în mod read-only. Din fișier doar se pot citi date. Această valoare este default.
 - ⇒ **w** - deschide fișierul cu drepturi de scriere. În fișier doar se poate scrie.
 - ⇒ **a** - deschide fișierul cu drepturi de adăugare. În fișier se poate scrie, dar datele deja existente vor rămâne.
 - ⇒ **r+** - deschide fișierul cu drepturi atât de scriere cât și de citire.
- Pentru scrierea într-un fișier vom folosi metoda `write()`:

```
with open('data.txt', 'w') as file:  
    file.write('hello, world!')
```

- Pentru a citi din fișier vom folosi una din metodele:

- citirea tuturor liniilor în același timp

```
with open('data.txt', 'r') as file:  
    for line in file.readlines():  
        print('line', line)
```

```
with open('data.txt', 'r') as file:  
    for line in list(file):  
        print('line', line)
```

- citirea linie cu linie folosindu-ne de `while`.

```
with open('data.txt', 'r') as file:  
    while True:  
        line = file.readline()  
  
        if not line:  
            break  
  
        print('line', line)
```



Lucrul cu fișiere

- Fișierele CSV (**C**omma **S**eparated **V**alues) sunt niște fișiere aparte dar foarte întâlnite în gestionarea datelor în domeniul programării.
- Teoretic, aceste fișiere conțin date separate prin virgulă, astfel încât fiecare rând poate fi reprezentat ca un rând dintr-un tabel, iar tot fișierul poate fi reprezentat sub forma unui tabel. În realitate poate fi folosit orice caracter cu rol delimitator.
- Considerând următorul tabel (imaginea stângă) acesta poate fi reprezentat printr-un fișier CSV (imaginea dreaptă) și vice-versa:

Brand	Model	An	CP
Opel	Astra	2002	101
Mazda	6	2012	120
Ford	Focus	2009	105

```
Brand,Model,An,CP
Opel,Astra,2002,101
Mazda,6,2012,120
Ford,Focus,2009,105
```

- Citirea și scrierea unui fișier CSV se poate face asemănător cu fișierele text, dar pentru o mai bună gestionare avem la dispoziție pachetul built-in **csv**.

```
import csv

with open('data.csv', 'r') as csv_file:
    rows = csv.reader(csv_file, delimiter=',')
    for row in rows:
        print(row)
```

```
import csv

new_cars = [
    ['Dacia', 'Logan', 2005, 75],
    ['Renault', 'Clio', 2005, 75]
]

with open('data.csv', 'a') as csv_file:
    csv_writer = csv.writer(csv_file, delimiter=',')

    for new_car in new_cars:
        csv_writer.writerow(new_car)
```



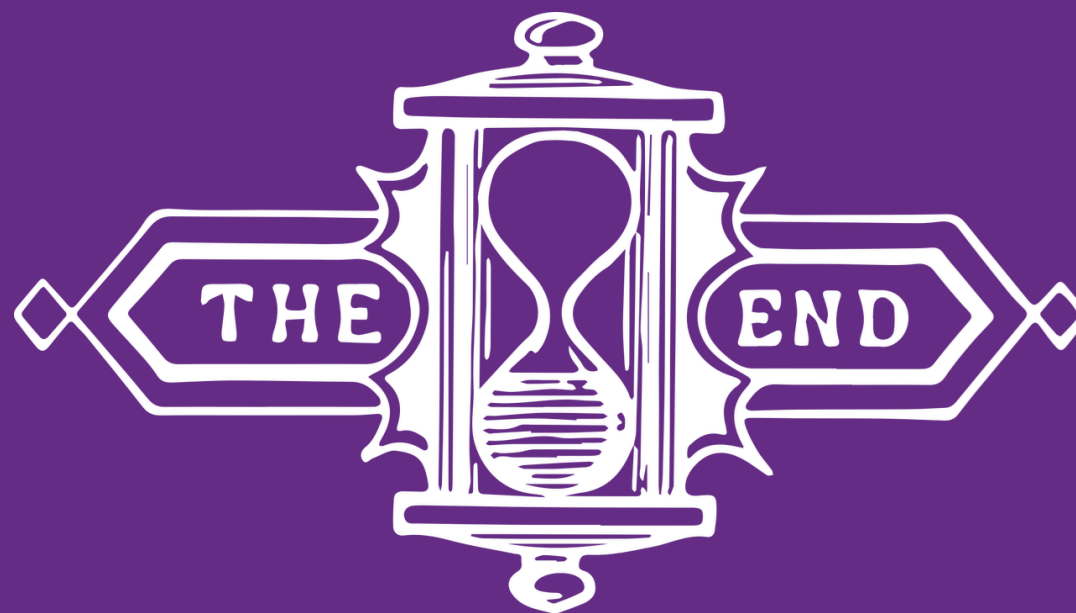
Lucrul cu fișiere

- Modulele **os** și **os.path** oferă suport pentru lucrul cu fișiere specifice dependente de sistemul de operare.
- Cu ajutorul acestor module se poate controla tot ce ține de sistemul de fișiere.
- În exemplul de mai jos:
 - vom obține un iterabil cu toate fișierele din directorul curent
 - le vom parcurge
 - vom afișa dacă este fișier sau director folosindu-ne de modulele amintite anterior.

```
import os

for dir_entry in os.scandir():
    if os.path.isfile(dir_entry):
        print(f'{dir_entry.name} is file')
    else:
        print(f'{dir_entry.name} is directory.')
```





Vă mulțumesc!

