Tutoriatul 06

Lambda, map, filter, zip + Recapitulare.

Reamintire funcții

- bucăți de algoritm (cod), *de obicei etichetate*, care reduc volumul codului și ușurează procesul de codare
- pot returna

Reamintire funcții

- bucăți de algoritm (cod), *de obicei etichetate*, care reduc volumul codului și ușurează procesul de codare
- pot returna o valoare, un tuplu de valori

Reamintire funcții

- bucăți de algoritm (cod), *de obicei etichetate*, care reduc volumul codului și ușurează procesul de codare
- pot returna o valoare, un tuplu de valori

  O variabilă, un tuplu de variabile

Reamintire funcții

- bucăți de algoritm (cod), *de obicei etichetate*, care reduc volumul codului și ușurează procesul de codare
- pot returna o valoare, un tuplu de valori

  O variabilă, un tuplu de variabile

  None

Orice fel de retur confirmă faptul că funcția și-a încheiat rularea.

Transmiterea parametrilor

- Parametrii sunt transmiți prin atribuire
- **Variabile imutabile:**

Transmiterea parametrilor

- Parametrii sunt transmiți prin atribuire
- **Variabile imutabile:**

- **Variabile mutabile**

Transmiterea parametrilor

- Parametrii sunt transmiți prin atribuire
- **Variabile imutabile:**

- **Variabile mutabile**

Transmiterea parametrilor

- Soluție pentru mutabile?

```
def liste(ls):
    ls = ls.copy()
    #sau
    ls = ls.deepcopy()
```

Funcții ca obiecte

- Funcțiile au, precum variabilele, o etichetă(id()) și un tip **deci** putem să le atribuim unor variabile.

Funcții ca obiecte

- Funcțiile au, precum variabilele, o etichetă(id()) și un tip **deci** putem să le atribuim unor variabile.

```
def lege(x, y):
    return x*y+2*x+2*y + 2

L = lege
print(L(123465862, -2))
```

Funcții ca obiecte

- Funcțiile au, precum variabilele, o etichetă(id()) și un tip **deci** putem să le atribuim unor variabile.

```
def lege(x, y):
    return x*y+2*x+2*y + 2


L = lege
print(L(123465862, -2))
```

Output:

Funcții ca obiecte

- Funcțiile au, precum variabilele, o etichetă(id()) și un tip **deci** putem să le atribuim unor variabile.

```
def lege(x, y):
    return x*y+2*x+2*y + 2


L = lege
print(L(123465862, -2))
```

Output: -2

Idee din curs: dicționar de funcții (atribuite unor variabile)

Funcții ca obiecte

- Funcțiile au, precum variabilele, o etichetă(id()) și un tip **deci** putem să le atribuim unor variabile.

```
def lege(x, y):
    return x*y+2*x+2*y + 2


L = lege
print(L(123465862, -2))
```

Output: -2

Idee din curs: dicționar de funcții (atribuite unor variabile)

În plus, funcțiile pot fi argumentele altor funcții.

Funcții anonime(lambda)

- Pentru funcții care sunt simple sau au o durată de viață scurtă (adică sunt folosite într-un singur cadru)
- Exemplu:

Funcții anonime(lambda)

- Pentru funcții care sunt simple sau au o durată de viață scurtă (adică sunt folosite într-un singur cadru)
- Exemplu: `sorted()`

Sorted are ca parametrii obligatorii un iterabil omogen și ca parametrii impliciți sensul (*Reverse*) și o *cheie*.

Funcții anonime(lambda)

- Pentru funcții care sunt simple sau au o durată de viață scurtă (adică sunt folosite într-un singur cadru)
- Exemplu: `sorted()`

Sorted are ca parametrii obligatorii un iterabil omogen și ca parametrii impliciți sensul (*Reverse*) și o *cheie*.

Cheia are scopul de a “ghida” funcția sorted() în felul în care vrem să sortăm iterabilul.

Funcții anonime(lambda)

Exemplu: Vrem să sortăm o listă de cuvinte în funcție de lungimea lor

Funcții anonime(lambda)

Exemplu: Vrem să sortăm o listă de cuvinte în funcție de lungimea lor

Apelarea de mai jos ne va ordona lista în ordine alfabetică.

```
lista_cuvinte=["marginire", "carnat", "sabie", "mar", "derivabilitate"]
lista_lungime = sorted(lista_cuvinte)
print(lista_lungime)
```

:Output

Funcții anonime(lambda)

Exemplu: Vrem să sortăm o listă de cuvinte în funcție de lungimea lor

Apelarea de mai jos ne va ordona lista în ordine alfabetică.

```
lista_cuvinte=["marginire", "carnat", "sabie", "mar", "derivabilitate"]
lista_lungime = sorted(lista_cuvinte)
print(lista_lungime)
```

```
['carnat', 'derivabilitate', 'mar', 'marginire', 'sabie']
```

:Output

Funcții anonime(lambda)

Exemplu: Vrem să sortăm o listă de cuvinte în funcție de lungimea lor

```
lista_cuvinte=["marginire", "carnat", "sabie", "mar", "derivabilitate"]
lista_lungime = sorted(lista_cuvinte, key =
print(lista_lungime)
```

:Output

sorted() iterează în funcție de elementele listei
Vrem să sorteze doar în funcție de lungimea acestora

Funcții anonime(lambda)

Exemplu: Vrem să sortăm o listă de cuvinte în funcție de lungimea lor

```
lista_cuvinte=["marginire", "carnat", "sabie", "mar", "derivabilitate"]
lista_lungime = sorted(lista_cuvinte, key = lambda x: len(x))
print(lista_lungime)
```

:Output

sorted(),implicit, iterează cu elementele listei
Vrem să sorteze doar în funcție de lungimea acestora

sorted(), în cazul descris mai sus, ia fiecare element din listă, îl “trece” prin funcția lambda, și apoi le sortează în funcție de returul funcției anonime.

Funcții anonime(lambda)

Exemplu: Vrem să sortăm o listă de cuvinte în funcție de lungimea lor

```
lista_cuvinte=["marginire", "carnat", "sabie", "mar", "derivabilitate"]
lista_lungime = sorted(lista_cuvinte, key = lambda x: len(x))
print(lista_lungime)
```

```
['mar', 'sabie', 'carnat', 'marginire', 'derivabilitate']
```
:Output

Funcții anonime(lambda)

Exemplu bis: Avem o listă de cuvinte și vrem să o sortăm în funcție de lungime, iar dacă două elemente au aceeași lungime vom sorta alfabetic.

```
lista_cuvinte = ["margine", "lume", "muci", "harta", "cinci"]
```

Idee: În general când vrem să sortăm niște obiecte pe baza mai multor criterii care vin într-o anumită ordine putem să ne folosim de ordonarea tuplurilor
În cazul de față, vom avea un tuplu de forma (lungimea cuvântului, cuvântul)

Funcții anonime(lambda)

Exemplu bis: Avem o listă de cuvinte și vrem să o sortăm în funcție de lungime, iar dacă două elemente au aceeași lungime vom sorta alfabetic.

```
lista_cuvinte = ["margine", "lume", "muci", "harta", "cinci"]
```

Idee: În general când vrem să sortăm niște obiecte pe baza mai multor criterii care vin într-o anumită ordine putem să ne folosim de ordonarea tuplurilor
În cazul de față, vom avea un tuplu de forma (lungimea cuvântului, cuvântul)

Soluție: o funcție poate returna un tuplu de valori

Funcții anonime(lambda)

Exemplu bis: Avem o listă de cuvinte și vrem să o sortăm în funcție de lungime, iar dacă două elemente au aceeași lungime vom sorta alfabetic.

```
lista_cuvinte = ["margine", "lume", "muci", "harta", "cinci"]
lista_noua = sorted(lista_cuvinte, key = lambda x: (len(x), x))
print(lista_noua)
```

:Output

Idee: În general când vrem să sortăm niște obiecte pe baza mai multor criterii care vin într-o anumită ordine putem să ne folosim de ordonarea tuplurilor
În cazul de față, vom avea un tuplu de forma (lungimea cuvântului, cuvântul)

Soluție: o funcție poate returna un tuplu de valori

Funcții anonime(lambda)

Exemplu bis: Avem o listă de cuvinte și vrem să o sortăm în funcție de lungime, iar dacă două elemente au aceeași lungime vom sorta alfabetic.

```
lista_cuvinte = ["margine", "lume", "muci", "harta", "cinci"]
lista_noua = sorted(lista_cuvinte, key = lambda x: (len(x), x))
print(lista_noua)
```

```
['lume', 'muci', 'cinci', 'harta', 'margine']
```
:Output

Idee: În general când vrem să sortăm niște obiecte pe baza mai multor criterii care vin într-o anumită ordine putem să ne folosim de ordonarea tuplurilor
În cazul de față, vom avea un tuplu de forma (lungimea cuvântului, cuvântul)

Soluție: o funcție poate returna un tuplu de valori

Funcții anonime(lambda)

Ce au în comun cele două exemple este că am folosit un singur sens de ordine.

Exemplu diferit(din tutoriatul 5):

Ordonați listele după inițiala șirului, iar dacă două liste au aceeași inițială, după ordine inversă a numerelor

Funcții anonime(lambda)

Ce au în comun cele două exemple este că am folosit un singur sens de ordine.

Exemplu diferit(din tutoriatul 5):

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)

Ce au în comun cele două exemple este că am folosit un singur sens de ordine.

Exemplu diferit(din tutoriatul 5):

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]]
```

Sortăm în funcție de două elemente, din nou, dar avem un conflict de sensuri.

Soluție? Schimbăm semnul forțat prin înmulțire cu -1 a tipului numeric

Output:

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

Funcții anonime(lambda)

Ce au în comun cele două exemple este că am folosit un singur sens de ordine.

Exemplu diferit(din tutoriatul 5):

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:

*sensul de parcurgere este același

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

map

map, filter, zip - exemplul #1

map: - “treci” o funcție peste unul sau mai mulți iterabili

- sintaxa: **map(func, *iterabili)**

```
def functie(n):
  return len(n)

x = map(functie, ('mere', 'pere', 'prune'))
print(x)
```

```
<map object at 0x7b89b611d0c0>
```

map, filter, zip - exemplul #1

map: - “treci” o funcție peste unul sau mai mulți iterabili

- sintaxa: **map(func, *iterabili)**

```
def functie(n):
  return len(n)

x = tuple(map(functie, ('mere', 'pere', 'prune')))
print(x)
```

```
(4, 4, 5)
```

map, filter, zip

Observatii:

- primește un număr variabil de iterabili (dați prin poziție)
  - *1 iterabil*: aplică funcția *func* pe fiecare dintre elementele iterabilului
  - *2 sau mai multi iterabili*: funcția aplicată trebuie sa primească tot atâția parametri câți iteratori a primit map în apel; în acest caz, *func* se aplică pe fiecare element al iteratorilor în paralel, până la epuizarea celui mai scurt dintre aceștia

- returnează un obiect de tip *map* ce poate fi iterat

map, filter, zip - exemplul #2

Exemplu map:

```
fructe = ["mere", "pere", "prune"]

print(list(map(list, fructe)))
```

Î: Ce se va afișa?

map, filter, zip - exemplul #2

Exemplu map:

```
fructe = ["mere", "pere", "prune"]

print(list(map(list, fructe)))
```

```
[['m', 'e', 'r', 'e'], ['p', 'e', 'r', 'e'], ['p', 'r', 'u', 'n', 'e']]
```

Î: Cum mai putem ajunge la acest rezultat? (indiciu: comprehensiune)

map, filter, zip - exemplul #2

Exemplu map:

```
fructe = ["mere", "pere", "prune"]

print(list(map(list, fructe)))
```

```
[['m', 'e', 'r', 'e'], ['p', 'e', 'r', 'e'], ['p', 'r', 'u', 'n', 'e']]
```

```
fructe = ["mere", "pere", "prune"]
lista = [list(fruct) for fruct in fructe]
print(lista)
```

```
[['m', 'e', 'r', 'e'], ['p', 'e', 'r', 'e'], ['p', 'r', 'u', 'n', 'e']]
```

map, filter, zip - exemplul #3

Avem o matrice cu 3 coloane. Vrem suma de pe fiecare coloană.

```
matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

sume_pe_coloane = tuple(map(lambda *c: sum(c), matrice[0], matrice[1], matrice[2]))
print("Sumele pe coloane:", sume_pe_coloane)
```

```
Sumele pe coloane: (12, 15, 18)
```

map, filter, zip - exemplul #4

Avem o matrice cu 190 de coloane. Vrem suma de pe fiecare coloană.

```
matrice = [list(range(j, j+10)) for j in range(100, 2000, 10)]
print(matrice)
```

```
[[100, 101, 102, 103, 104, 105, 106, 107, 108, 109], [110, 111, 112, 113, 114, 115, 116,
 …
 1986, 1987, 1988, 1989], [1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999]]
```

map, filter, zip - exemplul #4

Avem o matrice cu 190 de coloane. Vrem suma de pe fiecare coloană.

```
matrice = [list(range(j, j+10)) for j in range(100, 2000, 10)]

print(list(map(lambda *col: sum(col), *matrice)))
```

```
[198550, 198740, 198930, 199120, 199310, 199500, 199690, 199880, 200070, 200260]
```

map, filter, zip - exemplul #4

Ce înseamnă ***matrice**? = Despachetarea listei matrice:

```
map(..., matrice[0], matrice[1], matrice[2], ..., matrice[189])
```

Cum funcționează map cu mai multe iterabile?

```
map(func, a, b, c, ...)
```

La fiecare pas, map ia al i-lea termen din fiecare iterabil si le trimite funcției func.

```
func(a[0], b[0], c[0], ...)
func(a[1], b[1], c[1], ...)
...
```

map, filter, zip - exemplul #4

```
matrice = [list(range(j, j+10)) for j in range(100, 2000, 10)]

print(list(map(lambda *col: sum(col), *matrice)))
```

```
[198550, 198740, 198930, 199120, 199310, 199500, 199690, 199880, 200070, 200260]
```

Ce înseamnă **lambda *col: sum(col)**?

Parametrul *col înseamnă ”primește oricâte argumente și pune-le într-un tuplu numit col”.

La fiecare apel, funcția primește un tuplu cu valorile unei coloane:

```
col = (100, 110, 120, ..., 1990)   # prima coloană
sum(col)

col = (101, 111, 121, ..., 1991)   # a doua coloană
sum(col)
```

map, filter, zip - exemplul #5

Exemplu map (înmulțirea pe componente a două liste):

```
lista1 = [4, 7, 11, 12, 60]
lista2 = [3, 2, 99, 15, 6]
print(list(map(lambda x, y: x * y, lista1, lista2)))
```

```
[12, 14, 1089, 180, 360]
```

Î: Cum mai putem ajunge la acest rezultat? (indiciu: comprehensiune)

map, filter, zip - exemplul #5

Exemplu map (înmulțirea pe componente a două liste):

```
lista1 = [4, 7, 11, 12, 60]
lista2 = [3, 2, 99, 15, 6]
print(list(map(lambda x, y: x * y, lista1, lista2)))
```

```
[12, 14, 1089, 180, 360]
```

Î: Cum mai putem ajunge la acest rezultat? (indiciu: comprehensiune)

R: Soluție: `print([x * y for x in lista1 for y in lista2])`

map, filter, zip - exemplul #5

Exemplu map (înmulțirea pe componente a două liste):

```
lista1 = [4, 7, 11, 12, 60]
lista2 = [3, 2, 99, 15, 6]
print(list(map(lambda x, y: x * y, lista1, lista2)))
```

```
[12, 14, 1089, 180, 360]
```

Î: Cum mai putem ajunge la acest rezultat? (indiciu: comprehensiune)

Țeapă! Această comprehensiune nu funcționează cum doream noi.

```
lista1 = [4, 7, 11, 12, 60]
lista2 = [3, 2, 99, 15, 6]
print([x * y for x in lista1 for y in lista2])
```

```
[12, 8, 396, 60, 24, 21, 14, 693, 105, 42, 33, 22, 1089, 165, 66, 36, 24, 1188, 180, 72, 180, 120, 5940, 900, 360]
```

map, filter, zip - exemplul #5

Exemplu map (înmulțirea pe componente a două liste):

Soluția corectă cu comprehensiune:

```
lista1 = [4, 7, 11, 12, 60]
lista2 = [3, 2, 99, 15, 6]
print([x * y for x, y in zip(lista1, lista2)])
```

```
[12, 14, 1089, 180, 360]
```

zip

map, filter, zip - exemplul #1

zip: - ”lipește” elementele a doi sau mai mulți iterabili

- sintaxa: **zip(*iterabili)**

```
a = ("Gigel", "Sorin", "Speranta")
b = ("Militaru", "Dascalescu", "Vladoiu")

myzip = zip(a, b)

print(myzip)
print(tuple(myzip))
```

```
<zip object at 0x788372790f00>
(('Gigel', 'Militaru'), ('Sorin', 'Dascalescu'), ('Speranta', 'Vladoiu'))
```

map, filter, zip - exemplul #1

zip: - ”lipește” elementele a doi sau mai mulți iterabili

- sintaxa: **zip(*iterabili)**

```
a = ("Gigel", "Sorin", "Speranta")
b = ("Militaru", "Dascalescu", "Vladoiu")

myzip = zip(a, b)

for tuplu in myzip:
    print(tuplu)
```

```
('Gigel', 'Militaru')
('Sorin', 'Dascalescu')
('Speranta', 'Vladoiu')
```

map, filter, zip - exemplul #2

Iterabilii sunt zip-uiți până se epuizează unul dintre ei.

```
lista = list(zip(range(3), 'abcdef', (10, 20, 30, 40, 50, 60, 70, 80, 90)))
print(lista)
```

```
[(0, 'a', 10), (1, 'b', 20), (2, 'c', 30)]
```

filter

map, filter, zip - exemplul #1

filter: - filtrează elementele unui iterabil

- sintaxa: **filter(func, iterabil)**

```
lista_nume = ["Ana", "Andrei", "Gigel", "Agigel"]
incep_cu_A = filter(lambda nume: nume[0] == 'A', lista_nume)

print(incep_cu_A)
print(list(incep_cu_A))
```

```
<filter object at 0x78835bf06fe0>
['Ana', 'Andrei', 'Agigel']
```

map, filter, zip - exemplul #2

filter(None, iterabil) înseamnă filter(bool, iterabil) (bool = implicit)

-> filter(None, iterabil) filtreaza iterabilul pentru elemente care nu sunt “**falsy**”. Elemente falsy sunt:

None, False, 0, 0.0, ‘’, [], (), {} etc.

```
lista = [0, 1, 2, '', 'abc', None, [], [1], False, True]
print(list(filter(None, lista)))
```

```
[1, 2, 'abc', [1], True]
```