Materiale utile laborator PA

1. In caz ca vreti sa reluati elementele de sintaxa ale limbajului Python, puteti sa urmariti tutorialul de pe https://www.w3schools.com/python/. In partea stanga aveti un meniu pentru navigare rapida (de ex: variabile/string-uri/valori de tip bool etc.). De asemenea puteti face quiz-uri pentru testarea cunostintelor. (PS: Nu vom studia la laborator toate notiunile prezentate pe acest site, de aceea este bine sa invatati mai intai notiunile predate la curs si la seminar).

2. Avand in vedere ca la laborator vom implementa probleme de algoritmica, aveti mai jos o lista de notiuni care sunt folositoare atunci cand rezolvati exercitiile.

3. In caz ca o notiune nu este clara, este indicat sa o executati in REPL (Windows + R => cmd [Enter] => python [Enter]; apare >>>). Daca tot nu este evident cum functioneaza puteti sa o cautati in documentatia limbajului Python (https://docs.python.org/3/ => dreapta sus - Quick search).

Operatii elementare:

1. Variabilele sunt asemanatoare cu **pointerii/referintele** din C/C++ (ceea ce inseamna ca arata catre o valoare):
   - a = 12 # a arata catre locatia din memorie care contine numarul 12
   - a = "ana" # a "uita" ca arata catre valoarea 12 si acum arata catre "ana"
   - # am pierdut legatura catre 12 si nu o mai putem accesa… trebuia sa facem o
     # copie inainte de a asigna alta valoare daca voiam sa o pastram.

2. Control flow (if/while/for):
   - nu trebuie sa mai puneti parantezele pentru conditii
   - **trebuie** sa puneti **:** dupa instructiuni si sa **aliniati** restul instructiunilor
   - for i in range(n): # for (int i = 0; i < n; i++) ← mai mic **strict**
   - for i in range(in, sf): # for (int i = in; i < sf; i++) ← mai mic **strict**
   - for i in range(0, n, p): # for (int i = 0; i < n; i += p) ← mai mic **strict**

3. Conversie la un tip de date existent (int, string, bool etc.):
   - int(string)
   - ex: n = int('123'); s = str(12); b = bool(0); b = bool('0')
   - unii constructori primesc mai multe argumente; ex: int("101", base=2) == 5

4. Limbajul Python este strong typed ceea ce inseamna ca nu face conversie automat intre tipuri (exista cateva exceptii - comoditate):
   - a = "ana" + 3 # eroare nu se poate aduna un string cu un int
   - in schimb, in principiu se poate face conversie de la orice tip la string:
   - a = "ana" + str(3) # functioneaza: "ana" + **"**3**"** = "ana3"

Operatii pe biti

1. Operatiile elementare (tabel de adevar):
    - https://en.wikipedia.org/wiki/Boolean_algebra#Basic_operations **&, |, ~**
    - https://en.wikipedia.org/wiki/Boolean_algebra#Secondary_operations **->,^,≡**

2. Declaratie numar intr-o baza a (a = b - binar, a = o - octal, a = x - hexazecimal):
    - 0**a**1001 (in loc de **a** inlocuiti cu una din literele de mai sus)
    - ex: n = 0b1110; print(n) => 14
    - cand specificati o baza puteti pune zero-uri nesemnificative (0b0010 = 2)

3. Conversie numar din baza 10 in baza (2, 8, 16) in string:
    - bin(n), oct(n), hex(n)
    - ex: bin(11) = '0b1011' # il primiti ca string
    - bin(11)[2:] == '1011' # tot ca string, dar fara 0b din fata

4. Numerele negative sunt salvate in 2's complement ceea ce inseamna ca numerele negative nu vor fi afisate "corect" cand sunt convertite in baza 2. Pentru numerele cu semn (+/-) primul bit indica semnul (+ = 0, - = 1) :
    - pentru numere pe 4 biti, o metoda simpla este sa setam primul bit la 0 daca numarul este pozitiv si la 1 daca este negativ si sa pastram restul bitilor; de ex: 0b**0**110 = **+6** si 0b**1**011 = **-3**; bit-ul ingrosat nu se foloseste pentru numar ci doar pentru semn => putem salva astfel numere intre -7 si 7 in loc de 0 si 15.
    - 2's complement = nu este eficienta reprezentarea de mai sus cand vrem sa calculam a + (-a) = 0, deci 5 + -5 = 0, deci 0101 + 1101 = 0; 5-ul il pastram dar am vrea atunci cand adunam -5 ca rezultatul sa fie **1**0000 (pe **5** biti**!)**
    - -5 = 1011; ***intrebare*****:** cum putem ajunge de 5 (0101) la -5 (1011)?
    - de exemplu operatia **not:** ~5 = -6; (not-ul inversa toti bitii, pentru 101 avem 010 cand inversam; putem testa acest lucru: (-5) & (1 << 1) == 2, deci not-ul functioneaza si in plus: 0110 + 1010 = **1**0000 (tot pe **5** biti**!)**

Operatii string-uri:

1. Operatii uzuale siruri de caractere:
    - s = "Ana are mere"
    - for c in s: # parcurgere sir
        print(c, end=" ")
    - dimensiunea unui sir de caractere: len(s) == 12
    - concatenare: s + "!" == "Ana are mere!"
    - indexare: s[0] == 'A' si s[**-1**] = 'e'
    - extrage cuvinte: ss = s.split(" ") # ss =["Ana", "are", "mere"]
    - partitionare: i, p, s = "ana:32".partition(':') # i="ana" s="32"
    - slice-uri:
        i. s[1:4] = "na " # sir-ul s de la indexul 1 pana la 4, exclusiv
        ii. s[:4] = "Ana" # nu specificam inceputul => 0:4
        iii. s[::-1] = 'erem era anA' # Al treilea parametru reprezinta pasul
2. String-urile sunt tipuri de date de tip immutable ceea ce inseamna ca nu se mai pot modifica dupa ce au fost initializate (s + "1" construieste un **nou** string); daca vrem sa modificam un string il redeclaram: s = s +"1".
3. Functii de conversie intre **un caracter**(string format dintr-un singur caracter) si ASCII
    - ord('a') = 97
    - chr(98) = 'b'
4. Ordine relativa intre caractere:
    - vrem sa stim numarul de ordine al literei c (c este a treia litera din alfabet - sau a **doua** daca indexam de la 0):
        i. nr_c = ord('c')
        ii. nr_a = ord('a')
        iii. nr_ordine = nr_c - nr_a # **2**
5. Avand in vedere faptul ca string-urile sunt immutable, operatia de concatenare ("+") are complexitate O(n). Daca aveti m string-uri de concatenat si faceti *rezultat = rezultat + str_curent* complexitatea este O(m * n). De aceea, este indicat sa folositi operatia join:
    "".join(["Ana", "are", "mere"]) = "Anaaremere"
    ", ".join(["Ana", "are", "mere"]) = "Ana, are, mere"
6. Formatare:
    s = "ana are {} mere si {} pere".format(2, 3)
    == "ana are 2 mere si 3 pere"
    s = "Inaltimea este {height}, lungime este {length} si latimea este {width}".format(height=5, length=8, width=4)
    s = "Pi este {math.pi:**.4f**}".format(math=math)

Operatii tupluri

1. Operatii uzuale:
   a. construirea unui tuplu cu 2 elemente: t = **(**1, 2**)**
   b. extragerea unui element folosind idx: x = t[1] # x== 2
   c. construirea unui tuplu cu 1 singur elem: u = **(**"unu"**,)**
   d. construirea unui tuplu cu 0 elemente: z = **()**
   e. dimensiunea tuplului: len(t)
   f. parcurgerea tuplului:
      - for x in t:
        print(x)
   g. verificarea existentei (/lipsei) unui element in t:
      - 2 **in** t == True
      - 3 **not in** t == True
      - **not** 3 **in** t == True

2. Tuplurile sunt obiecte de tip **immutable** (ca string-urile) deci odata ce au fost create nu mai pot fi modificate:

```
t = (1, 2, 3)
t[1] = 4 # Eroare
```

3. Se pot folosi daca vreti ca o functie sa returneze mai multe valori:

```
# f : R -> RxR
# f(x) = (x + 1, x^2 +1)
def f(x):
    return (x + 1, x**2 + 1)

(y1, y2) = f(1)
```

4. Prin tuple unpacking se poate face identificare:

```
(a, b, (c, d), e) = (1, 2, ("ana", "mere"), 5)
c == "ana"
e == 5
```

5. Inversarea ordinii:

```
t = (1, 2, 3)
it = t[::-1]
it == (3, 2, 1)
```

Operatii liste

1. Operatii uzuale pe liste:
   a. lista vida: xs = []
   b. adaugare element: xs.append(123)
   c. extragere ultimul el: x = xs.pop()
   d. extragere el de pe **idx**: x = xs.pop(**idx**)
   e. parcurgere lista:
      ```
      for x in xs:
          print(x)
      ```
   f. dimensiune lista: len(xs)
   g. slice-uri (ca la string-uri):
      xs[inc:sf:pas] = [inc, inc + pas, inc + 2*pas, … sf - 1]
      ! daca lipseste inc, se considera inc=0 (xs[:3]==xs[0:3])
      ! daca lipseste sf, se considera sf=len(xs) xs[1:]
      ! daca lipseste si inc si sf se intampla ca mai sus ^
      ! puteti folosi indexare negativa xs[**-1**] == xs[len(xs)**-1**]
      ! pentru a inversa lista xs.reverse() sau xs[::-1]
      ! pentru a copia o lista care contine elemente
      ! immutable folositi ys = xs[:]
2. Tineti minte de la **Operatii elementare 1)** variabilele sunt referinte:
   a. xs = [1, 2, 3]
   b. ys = xs
   c. ys.append(4)
   d. print(ys) # [1, 2, 3, 4]
   e. print(**xs**) # [1, 2, 3, **4**] **! important !**
3. Alte operatii interesante:
   a. stergerea tuturor elementelor din xs: xs.clear()
   b. gasirea indexului unui element din xs: xs.index(element)
      ! daca nu exista aparea eroarea **ValueError**;
      ! trebuie sa o tratati cu try: …. except ValueError: …
      ! sau va asigurati ca nu apare: if elem in xs: i= xs.index(elem)
   c. sortarea listei:
      i. inplace: xs.sort(reversed=False) # poate lipsi
      ii. alta lista: ys = sorted(xs)
      ! in caz ca aveti tupluri ca elem, sortarea se face mai intai
      ! dupa primul element si apoi dupa cel de-al doilea; daca vreti
      ! sa definiti alt tip de sortare: xs.sort(key=f(x)) unde f
      ! este o functie definita de voi care primeste un elem si
      ! returneaza ordinea relativa: ex: def f(t): return (t[1], t[0])
   d. list comprehension:
      i. [i for i in range(n)] == [0, 1, 2, … n-1]
      ii. [0 for i in range(n)]==[0, 0, 0, … 0] (n elem. idx de la 0)
      iii. [i*i for i in range(10) if i%2==1] == [1, 9, 25, 49, 81]
4. Copierea listelor care contin elemente mutable:

a. lista_0 = [1, 2, 3]
b. lista_1 = [i for i in range(4, 7)]
c. lista_2 = [i for i in range(7, 10)]

d. xs = [lista_0, lista_1, lista_2]
e. ys = xs[:] # ys contine referintele catre aceleasi liste

f. xs is ys == False # cele doua variable nu sunt identice
g. (xs == ys) == True # cele doua variabile contin aceleasi el.

h. xs[0] = [0, 0, 0]
i. print(xs, '\n', ys) # xs s-a modificat, ys nu s-a modificat

j. lista_1.append(0) # adaugam 0 la finalul listei 1
(echivalent cu xs[1]. append(0) ***sau*** ys[1].append(0))
k. print(xs, '\n', ys) # s-au modificat in ambele parti

l. lista_2.clear()
m. print(xs, '\n', ys) # s-au sters elementele din ambele parti

n. del xs[2]
o. print(xs, '\n', ys) # am sters doar lista_2 din xs

p. ! daca vreti sa nu se mai intample cazul **j** trebuie sa faceti ***deep copy*** ceea ce inseamna ca trebuie sa copiati fiecare lista in parte; in cazul nostru:

```
xss = [[1,2,3], [4,5,6], [7,8,9]]
yss = []
for xs in xss:
        yss.append(xs[:])

print(xss, '\n',  yss)

xss[0].insert(0, 0)          # pe pozitia 0 inseram elem. 0
print(xss,  yss)
```

Evident, metoda de mai sus nu functioneaza daca aveti 3 liste imbricate. Daca vreti sa functioneze in cazul general trebuie sa verificati daca elementele sunt mutabile. Alternativ, folositi **from copy import deepcopy** si **yss = deepcopy(xss)**.

Operatii multimi

1. Operatii uzuale pe multimi:
    a. multimea vida: s = set()
    b. multime plecand de la lista: s = set([1,2,3]) *sau* s={1,2,3}
    c. adaugare element: s.add(4)
    d. cautare element: 4 in s
    e. stergere element s.remove(4)
    f. parcurgere multime:
    ```
    for x in s:
        print(x)
    ```
    g. dimensiune multime: len(s)

2. Set-urile sunt structuri de date de tip **mutable** (la fel ca listele), dar spre deosebire de liste, **ordinea nu conteaza**. Ceea ce insemna ca atunci cand parcurgeti un set, puteti primi elemente care nu sunt in ordinea uzuala (incercati ***set([1, 5, 10])***).

3. Set-urile sunt obiecte de tip **abstract data type** (ADT) ceea ce inseamna ca pot retine orice tip de date de tip immutable (set-uri de string-uri, tupluri, int). **Nu** putem salva **liste/set-uri**!

4. Set-urile sunt multimi din punct de vedere matematic (nu au duplicate). Daca incercati sa adaugati un element care exista deja, nu se va intampla nimic.

5. Complexitatea operatiilor:
    a. adaugare element in set: O(1)
    b. cautare element in set: O(1)
    c. stergere element din set: O(1)

6. Folosind set-uri puteti elimina duplicatele dintr-o lista:
```
xs = [1, 2, 1, 1, 3, 1]
s = set(xs)
ys = list(s)          # aplicati sorted(s) daca le vreti sortate
print(ys)
```

Operatii dictionare

1. Operatii uzuale pe dictionare:
   a. dictionar vid: d = dict() *sau* d = {}
   b. dictionar - constructor: d = dict([(1,2), (3,4), (5,6)])
   c. elementele sunt perechi de tipul cheie - valoare
      i. adaugare element: d["ana"] = 1
      ii. cautare cheie: "ana" in d == True
      iii. extragere valoare: cnt = d["ana"]
      iv. stergere element: del d["ana"]
   d. parcurgere:
      i. pe chei:
```
for k in d:
    print(k, "=>", d[k])
```
      ii. pe valori:
```
for v in d.values():
    print(v)
```
      iii. pe chei si pe valori:
```
for k, v in d.items():
    print(k, "=>", v)
```
   e. dimensiune dictonar: len(d)

2. La fel ca listele si set-urile, dictionarele sunt obiecte de tip **mutable** (se modifica in place). De asemenea, nu puteti salva in *chei* elemente de tip **mutable** (liste/set-uri/dictionare), ci doar elemente de tip **immutable**(string-uri, tupluri, int-uri etc.). In schimb nu aveti nicio restrictie pentru *valori*!
```
d[1] = [1,2,3]
d[1].append(4)
d[2] = dict()
d[2][1] = (1, 2)
```

3. Daca incercati sa folositi o cheie care nu exista veti primi o eroare de tipul KeyError. Puteti sa o prindeti (try: …. except KeyError: …) sau sa va asigurati ca nu apare (if k in d: …) (sau folosind dictionare cu valoare implicita). Exemplu (vectori de frecventa):
```
xs = [1, 2, 1, 1, 3, 2]          # import collections
f = {}                           # f = collections.defaultdict(int)
for x in xs:                             # for x in xs:
    if x in f:                           #     f[x] += 1
        f[x] += 1                        #
    else:                                # for k in sorted(f.keys()):
        f[x] = 0                         #     print(k, "de", f[k], "ori")
```
4. Complexitati - la fel ca la set-uri: O(1) / operatie