Cunoștințe necesare de Python

Valori booleene

Valorile boolean din Python sunt True și False. Tipul bool este un tip numeric, prin urmare, True este egal cu 1 și False e gal cu 0

```
>>> True==1
True
>>> False==0
True
>>> True+True
2
>>> False*5
0
```

Mai există și alte tipuri de valori care pot fi evaluate la valori boolene în cadrul unor instrucțiuni, precum if sau while, care cer expresii boolene :

- None e evaluat la False
- orice număr nenul e evaluat la True
- obiectele de tip secvență (listă, tuplu, dicționar, mulțime, șir etc) sunt evaluate la False dacă sunt vide, și True dacă sunt nevide.

Operatori booleeni:

- `not expresie_booleana` - returnează opusul valorii boolene a expresiei
- `expresie1 and expresie2` - returnează True dacă ambele expresii sunt evaluate la True și False în caz contrar. Din acest motiv, dacă `expresie1` e evaluata la False, `expresie2` nu mai este evaluată, deoarece rezultatul e deja cunoscut ca fiind fals.
- `expresie1 or expresie2` - returnează True dacă măcar una dintre expresii este evaluată la True și False în caz contrar. Din acest motiv, dacă `expresie1` e evaluata la True , `expresie2` nu mai este evaluată, deoarece rezultatul e deja cunoscut ca fiind adevărat.

```
>>> def f():
    print("ceva")
    return True

>>> True and f()
ceva
True
>>> False and f()
False
>>> True or f()
```

```
True
>>> False or f()
ceva
True
```

Operatorul ternar

Acest operator permite o scriere mai facilă a unei expresii condiționale și are sintaxa:
`val_true if conditie else val_false`
cu sensul că valoarea expresiei e `val_true` dacă avem condiția adevărată și `val_false` în caz contrar.

```
>>> 10 if 3 in [2,4,7] else 11
11
>>> 10 if 4 in [2,4,7] else 11
10
```

Operații cu șiruri de caractere

Șirurile de caractere se scriu între ghilimele sau apostrofuri, de exemplu, "sir".Putem scrie un șir pe mai multe rânduri daca folosim 3 apostrofuri sau 3 ghilimele la inceput și la finalul lui. Atenție dacă am deschis șirul cu apostrof sau ghlimele trebuie să îl închidem cu același tip de semn. Putem accesa caracterele unui șir direct prin indici. **Șirurile sunt *immutable***, putem accesa un caracter prin indice, dar nu îl putem modifica. Prin urmare **metodele șirurilor nu modifică șirul pe care se aplică ci returnează un șir nou.**

Dacă adunăm două șiruri obținem concatenarea lor:

```
>>> "py"+"thon"
'python'
```

Dacă înmulțim un șir cu un număr *n*, obținem un șir nou în care se repetă șirul inițial de *n* ori:

```
>>> "abc"*5
'abcabcabcabcabc'
```

Metode utile

Metodele lstrip(), rstrip(), strip()

Eliminarea spațiilor din capete:

```
>>> sir="  abc   "
>>> sir.lstrip()
```

```
'abc '
>>> sir.rstrip()
'    abc'
>>> sir.strip()
'abc'
```

Metoda split()

Creează o listă cu subșirurile șirului inițial

```
>>> "ab#cd#efgh#".split("#")
['ab', 'cd', 'efgh', '']
```

Metoda join()

Este opusul metodei split(). Pentru un șir conținând "separatorul" și o listă de șiruri puteți folosi metoda join pentru a concatena toate șirurile din listă (rezultând un singur șir) punând între ele șirul separator:

```
>>> "#".join(['ab', 'cd', 'efgh', ''])
'ab#cd#efgh#'
```

Funcția sorted()

Se poate folosi pentru a obține o listă cu literele sortate (crescător sau descrescător) dintr-un șir:

```
>>> sorted("bcdabdf")
['a', 'b', 'b', 'c', 'd', 'd', 'f']
>>> sorted("bcdabdf", reverse=True)
['f', 'd', 'd', 'c', 'b', 'b', 'a']
```

Operații cu liste

Liste sunt seturi ordonate de elemente. Se enumeră între paranteze pătrate.

Crearea unei liste

Se enumeră elementele între paranteze drepte, sau se folosește constructorul list care primește un obiect prin care se poate itera (precum un sir de caractere).

```
l1=[1,2,3]
l2=list("abc")
print(l1)
print(l2)
```

```
[1, 2, 3]
['a', 'b', 'c']
```

Lungimea unei liste

Lungimea unei liste se calculează cu len(lista). De exemplu, len([10,5,1]) este 3.

Numărarea elementelor dintr-o listă

Folosim metoda count(element) pentru a număra de câte ori apare elementul în listă:

```
>>> l=[1,2,5,0,2,1,1,3]
>>> l.count(1)
3
```

Concatenarea listelor

Dacă adunăm două liste obținem concatenarea lor:

```
>>> [1,2]+[3,4]
[1, 2, 3, 4]
```

Dacă înmulțim o listă cu un număr *n*, obținem o listă nouă în care se repetă șirul inițial de *n* ori:

```
>>> [1,2]*4
[1, 2, 1, 2, 1, 2, 1, 2]
```

Indicii unei liste

Indicii unei liste încep de la 0. Pentru lista l=[10,5,1], l[0] este 10, iar l[2] este 1.
Putem folosi și indici negativi. Indicii negativi îi putem considera pornind de la dreapta spre stânga începând cu -1, și tot descrescând cu 1 spre stânga. De exemplu pentru lista de mai sus, l[-1] este 1, l[-2] este 5 și l[-3] este 10.

Subliste

Pentru a obține o sublistă dintr-o listă putem folosi notația l[inidiceStart:indiceFinal] care va oferi sublista cu elementele din lista inițială cuprinse între pozițiile indicestart inclusiv și indiceFinal exclusiv.
Dacă dorim un mod de parcurgere a listei diferit de cel implicit, pentru a crea sublista, putem adăuga și parametrul de iterare: l[inidiceStart:indiceFinal:iterator]. Astfel se va porni de la indicele de Start, punând elementul corespunzător în sublistă, si la indiceStart se va aduna apoi iteratorul, generând urmatorul element din sublistă. Procedeul se va repeta până se ajunge la un indice mai mare sau egal cu indicele final (pentru această ultimă valoare care depășește limita dată de indicele final nu se mai generează element în sublistă).
Oricare dintre cele trei argumente ale scrierilor l[inidiceStart:indiceFinal] și l[inidiceStart:indiceFinal:iterator] poate lipsi, caz în care se iau valorile implicite (pentru indiceStart se ia valoarea 0, pentru indiceFinal, lungimea listei și pentru iterator 1).
Exemple de utilizări ale acestor scrieri, în consola Python:

```
>>> l=list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:8]
[2, 3, 4, 5, 6, 7]
>>> l[2:8:2]
[2, 4, 6]
>>> l[-1:-7]
[]
>>> l[-1:-7:-1]
[9, 8, 7, 6, 5, 4]
>>> l[8:2:-1]
[8, 7, 6, 5, 4, 3]
>>> l[:5]
[0, 1, 2, 3, 4]
>>> l[4:]
[4, 5, 6, 7, 8, 9]
>>> l[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> l[2:1234]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> l[9:-3:-1]
[9, 8]
```

Observați că putem folosi și indici negativi, dar dacă vrem o parcurgere de la dreapta la stânga, trebuie să folosim un iterator negativ, așa cum rezultă din rândurile marcate cu galben.

Iterarea printr-o listă

Putem itera în mai multe moduri.
Operatorul in permite iterarea prin fiecare element al listei fără a cunoaște indicele elementului:

```
for elem in l:
    proceseaza(elem)
```

Dacă avem nevoie și de indicele elementului putem parcurge lista folosindu-ne de accesul direct al elementului prin indice (insă nu e recomandată această parcurgere când modificăm lista (adăugăm sau ștergem elemente) în acest mod, deoarece trebuie să fim atenți la actualizarea indicelui.

```
for i in range(len(l)):
    proceseaza(l[i])
```

Putem itera printr-o listă folosind enumerate(lista) care este un generator prin care obținem pe rând tupluri de forma (indice, lista[indice]) (adică indicele din listă și elementul corespunzător acestuia).

```
for i, elem in enumerate(l):
    proceseaza(elem)
```

Suma elementelor unei liste (funcția sum())

Funcția sum() primește ca parametru o listă de elemente ce pot fi adunate și eventual o valoare de inițializare pentru sumă (parametrul start).

```
>>> sum([2,5,1,4])
12
>>> sum([2,5,1,4],start=100)
112
```

Căutarea unei element (metoda index(element))

Metoda index(element) returnează indicele primei apariții a unui eelement și aruncă o eroare dacă elementul nu se găsește în listă.

```
>>> l=[10,4,1,2,22,1]
>>> l.index(1)
```

```
2
>>> l.index(7)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    l.index(7)
ValueError: 7 is not in list
```

Minimul și maximul unei liste

Metoda min(lista) returnează elementul minim al unei liste. Metoda min poate primi și argumentul key sub formă de funcție care se va aplica fiecărui element din listă, evaluând minimul după valoarea returnată de funcția din key pentru fiecare element. Metoda max(element) returnează mximul dintr-o listă și acceptă de asemenea un parametru key care funcționează după aceleași principii ca și în cazul lui min().

```
>>> min([23,16,100,44])
16
>>> min([23,16,100,44], key=lambda x:x%10)
100
>>> max(["abc","def","aaaaaa","z"])
'z'
>>> max(["abc","def","aaaaaa","z"], key=lambda x: len(x))
'aaaaaa'
```

Filtrarea unei liste

Se poate folosi funcția `filter(functie_booleana, lista)`. Se va aplica funcția booleană fiecărui element din lista dată ca argument. Valorile pentru care funcția booleană returnează True se vor regăsi în rezultatul lui filter. Atenție, filter nu returnează o listă ci un obiect iterabil.

```
>>> list(filter(lambda x: x%2==0,[20,3,11,12,28]))
[20, 12, 28]
```

Reducerea unei liste la o valoare

O listă poate fi redusă la o valoare (de obicei, scalar), cu funcția reduce (functie, lista [, initial]). Funcția data ca argument trebuie să aibă 2 parametri(îi vom nota a și b), a fiind rezultatul parțial calculat pentru primele valori din listă (de dinaintea lui b), iar b fiind elementul curent din listă.

Funcția reduce se va aplica întâi primelor două elemente ale listei, apoi va aplica funcția între rezultatul anterior și al treilea element din listă și tot așa.

```
from functools import reduce
import math

l=[10,22,8,408,32]
x=reduce(lambda a,b: math.gcd(a,b), l)
print(x) # 2
```

Matrici

În Python nu avem în mod direct o structură pentru matrici (printre modulele implicite), în schimb puem simula o matrice printr-o listă de liste.
De exemplu:

```
l=[[1,2,3],[4,5,6]]
```

ar fi o matrice de 2 linii și 3 coloane.

Adăugarea elementelor într-o lista

La finalul listei (metoda append(element)):

```
l=[10,7,3,5]
l.append(2)
print(l)
```

```
[10, 7, 3, 5, 2]
```

La o poziție dată (metoda insert(indice, element)):

```
l=[10,7,3,5]
l.insert(1,122)
print(l)
```

```
[10, 122, 7, 3, 5]
```

Ștergerea elementelor dintr-o listă

Se folosește metoda pop(indice) care șterge din listă elementul de pe poziția dată. Dacă nu o folosim cu argumente, este echivalentul folosirii cu indicele -1, prin urmare sterge ultimul element.

```
l=[10,7,3,5]
l.pop(2)
print(l)
```

```
[10, 7, 5]
```

Fară parametri:

```
l=[10,7,3,5]
l.pop()
print(l)
```

```
[10, 7, 3]
```

Putem șterge un anumit element, cu metoda remove(element)

```
l=[10,7,3,5]
l.remove(7)
print(l)
```

```
[10, 3, 5]
```

Sortarea unei liste

Sortarea unei liste se poate face simplu prin metoda sort().

```
l=[2,1,10,4,100,17,23]
l.sort()
print(l) #[1, 2, 4, 10, 17, 23, 100]
```

Sortarea implicită este cea crescătoare. Dacă dorim să sortăm descrescător (practic să inversăm lista după sortare), putem folosi parametrul *reverse* al funcției sort, cu valoarea *True*:

```
l=[2,1,10,4,100,17,23]
l.sort(reverse=True)
```

```
print(l) #[100, 23, 17, 10, 4, 2, 1]
```

Putem să dorim sortări speciale (diferite de cea implicită). De exemplu, putem dori sortarea după ultima cifră a numerelor:

```
l=[2,1,10,4,100,17,23]
l.sort(key= lambda x: x%10)
print(l) #[10, 100, 1, 2, 23, 4, 17]
```

Alternativ, se poate folosi funcția sorted(obiect_iterabil), care de asemenea primește parametrii key și reverse:

```
l=[2,10,3,1,7000,9010,200,111111]
print(sorted(l)) #[1, 2, 3, 10, 200, 7000, 9010, 111111]
print(l) #[2, 10, 3, 1, 7000, 9010, 200, 111111]
print(sorted(l, reverse=True)) #[111111, 9010, 7000, 200, 10, 3, 2,
1]
print(sorted(l, key=lambda x: str(x).count("0"))) #[2, 3, 1, 111111,
10, 9010, 200, 7000]
```

Spre deosebire de metoda sort() funcția sorted() nu schimbă lista primită ca parametru, ci returnează o listă nouă.

Modificarea listei în timpul parcurgerii

Dacă dorim să modificăm o lista, de exemplu, vrem să ștergem toate elementele pare, se întâmpla des să se facă **greșeala următoare**:

```
l=[3,2,4,5,10,12]
l=[3,2,4,5,10,12]
for i in range(len(l)):
    if l[i]%2==0:
        l.pop(i)
        i-=1 #degeaba fiindca i nu e de fapt incrementat ci se ia
urmatorul din range
```

Vom primi o eroare penru metoda pop() care va da un "IndexError: pop index out of range" pentru că i-ul nu este decrementat la ștergerea elementului (necesar deoarece la ștergere, toate elementele se mută cu o poziție în stânga).

O altă greșeală:

```
l=[3,2,4,5,10,12]
#tot gresit fiindca se sare peste elemente
```

```
for elem in l:
    if elem%2==0:
        l.remove(elem)
print(l)
#rezultatul va fi: [3, 4, 5, 12]
```

Modul corect. Sunt mai multe moduri prin care putem realiza cerința de mai sus, având grijă să nu sărim elementele, de exemplu, folosind while (in felul acesta limita pentru i nu mai e rigidă, deja calculată cum era în cazul lui range:

```
l=[3,2,4,5,10,12]
i=0
while i<len(l):
    if l[i]%2==0:
        l.pop(i)
        i-=1
    i+=1
print(l) # va afisa [3, 5]
```

Modul corect și elegant. Folosind comprehensions:

```
l=[3,2,4,5,10,12]
rez=[x for x in l if x%2==1]
print(rez) # va afisa [3, 5]
```

Funcțiile all() și any()

Funcțiile all() și any() se folosesc în contextul în care primesc o listă de elemente (fiecare element putând fi evaluat la o valoare booleană). Funcția all() returnează True, dacă toate elementele din listă pot fi evaluate la True și False în caz contrar. Funcția any() returnează True, dacă măcar un element din listă poate fi evaluat la True și False în caz contrar.

```
>>> all([True,True, True])
True
>>> all([True,True, False,True])
False
```

```
>>> any([False,True,False])
True
>>> any([False,False,False])
```

```
False
```

Dicționare

Dicționarele reprezintă un set de perechi de chei și valori (asociate cheilor). Cheile sunt unice în dictionar și trebuie să fie de tip immutable (de exemplu pot fi stringuri , numere, tupluri, dar nu pot fi liste).
Pentru a crea un disctionar vid, folosim:

```
d={}
```

Pentru a adăuga chei noi în dicționar, putem pur și simplu să le folosim pentru prima oară, atribuindu-le o valoare. Sintaxa este: dictionar[cheie]=valoare.

```
d["a"]=100
d["b"]=200
d["c"]=300
print(d)
```

```
{'a': 100, 'b': 200, 'c': 300}
```

Pentru a itera printr-un dicționar, putem folosi operatorul in:

```
for k in d:
    print(k,d[k])
```

sau să folosim metoda items() care returnează o listă cu tupluri de forma (cheie,valoare):

```
for k,v in d.items():
    print(k,v)
```

Ambele au output de mai jos:

```
a 100
b 200
c 300
```

Pentru a obține lista de chei putem folosi metoda keys() iar pentru lista de valori metoda values().

Pentru a verifica dacă o cheie se găsește într-un dicționar, putem folosi operatorul *in*.

Mulțimi

Mulțimile reprezintă seturi de elemente **neordonate** care nu acceptă duplicate.

Crearea unei mulțimi

```
multime_vida=set()
multime={2,3,10,8}
print(multime)
```

```
{10, 8, 2, 3}
```

Observați că nu s-a păstrat ordinea din inițializarea mulțimii, fiindcă mulțimile sunt **neordonate**.

Cardinalul unei mulțimi

Pentru a obține cardinalul unei mulțimi se folosește funcția len(). De exemplu, len({2,4,10}) va fi 3.

Compararea secvențelor

Toate obiectele de tip secvență ordonată (liste, tupluri, șiruri etc) se supun aceluiași mod de comparare. Când două obiecte a și b de tip secvență sunt comparate, întâi se compară primul element din a, a[0] cu primul element din b, b[0]. Dacă nu sunt egale, atunci relația de ordine dintre a[0] și b[0] este și relația de ordine dintre a și b. Dacă a[0] și b[0] sunt egale se trece la compararea următoarelor două elemente: a[1] și b[1] și se repetă raționamentul. În general pentru prefixe maximale egale a[0:i], b[0:i], cu primul set de valori distincte în cele două secvențe, a[i+1] și b[i+1] relația de ordine dintre a și b e aceeași cu relația de ordine dintre a[i+1] și b[i+1]. Dacă o secvența a este prefix a unei secvențe b, atunci a < b.
Atenție, obiectele de tip secvență trebuie să fie de tipuri compatibile (în general, să fie de același tip).

```
>>> [1,2,3]  < [1,2,4]
True
>>> [1,2]<[1,2,3]
True
>>> (1,2)<[1,2,3]
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    (1,2)<[1,2,3]
```

```
TypeError: '<' not supported between instances of 'tuple' and 'list'
```

Comprehensions

List comprehensions

Au sintaxa de forma
`[expresie for element in obiect_iterabil]` - caz in care se va genera o lista cu acelasi număr de elemente precum obiectul iterabil
sau
`[expresie for element in obiect_iterabil if conditie]` - caz în care va avea în listă doar elementele din obiectul iterabil care îndeplinesc condiția, prin urmare vor fi mai puține elemente sau în număr egal.

Exemple:

```
l=[2,7,5,23,10]

#lista cu dublul elementelor lui l
l1=[2*x for x in l]
#[4, 14, 10, 46, 20]

#lista cu perechile de vecini din l
l2=[[l[i], l[i+1]] for i in range(len(l)-1)]
#[[2, 7], [7, 5], [5, 23], [23, 10]]

#lista cu produsul cartezian
l3=[[x,y] for x in l for y in l]
#[[2, 2], [2, 7], [2, 5], [2, 23], [2, 10], [7, 2], [7, 7], [7, 5],
[7, 23], [7, 10], [5, 2], [5, 7], [5, 5], [5, 23], [5, 10], [23, 2],
[23, 7], [23, 5], [23, 23], [23, 10], [10, 2], [10, 7], [10, 5], [10,
23], [10, 10]]

#lista cu elementele pare
l4=[x for x in l if x%2==0]
#[2, 10]
```

Crearea unei matrici folosind *comprehensions*

Putem folosi un comprehension cu *for* dublu.
De exemplu dacă dorim o matrice formată doar din 0-uri:

```
l=[ [0]*5 for _ in range(10)]
print(l)
```

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Atentie, frecvent se face **greșeala** următoare:

```
l=[ [0]*5] *10
print(l)
l[0][0]=111
print(l)
```

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
[[111, 0, 0, 0, 0], [111, 0, 0, 0, 0], [111, 0, 0, 0, 0], [111, 0, 0,
0, 0], [111, 0, 0, 0, 0], [111, 0, 0, 0, 0], [111, 0, 0, 0, 0], [111,
0, 0, 0, 0], [111, 0, 0, 0, 0], [111, 0, 0, 0, 0]]
```

Observați cum s-a schimbat primul element în fiecare listă. atunci cand facem lista*n, unde n e un număr natural nenul, se copiază elemente din listă de n ori. Problema apare când avem o listă de obiecte, deoarece se copiază referențele către acele obiecte. Practic am avut [lista_de_0] * 10 care a dus la o listă cu 10 referințe către aceeași lista de 0-uri, deci când am schimbat primul element din prima listă am văzut modificarea în toate cele 10 liste fiindcă de fapt **sunt toate același obiect**.

Operații cu fișiere

Pentru a deschide un fișier folosim metoda open(cale_fisier,mod_deschidere).
De exemplu, pentru a citi fisierul input.txt, putem folosi:

```
f=open("input.txt","r")
sir=f.read()
```

caz în care vom avea în șir tot conținutul fișierului
O altă variantă este să folosim metoda readlines() care returnează o listă de stringuri cu liniile fișierului.

Pentru a scrie într-un fișier putem deschide fișierul cu "w" pentru a fi suprascris, sau cu "a" pentru a adăuga la final de fișier.
Pentru a scrie într-un fișier putem folosi metoda write().

Clase

Pentru a defini o clasă folosim cuvântul cheie class, urmat de numele clasei.
Clasele au o metodă specială prin care se construiesc instanțele clasei, numită __init__. In funcția init vom trimite argumentele necesare pentru a completa proprietățile noii instanțe a clasei.
Orice metodă proprie instanțelor are ca prim parametru chiar o referință către instanță respectivă (metoda __init__ nu face excepție). De obicei acest prim parametru este numit *self*, dar nu este obligatoriu. Parametrul self nu va avea un argument corespunzător în apelul metodei, practic metodele se apelează cu argumente corespunzătoare tuturor parametrilor, mai puțin self.
Proprietățile de instanță nu se definesc direct în clasă ci sunt create în constructor atunci când le folosim numele prima oară, de exemplu, facem o inițializare de genul self.proprietate=valoare.

```
class Cls:
    def __init__(self, aa, bb):
        self.a=aa
        self.b=bb
    def incrementeaza_a(self):
        self.a+=1

c1=Cls(2,5)
c1.incrementeaza_a()
print(c1.a)
```

Observați cum în exemplul de mai jos, la crearea instanței c1, s-au dat valori doar pentru parametrii aa și bb din __init__ primul parametru fiind self, pentru care nu se oferă argument. De asemenea, metoda incrementeaza_a() nu se apelează cu argumente deoarece are ca unic parametru self, adică instanța.

Pentru a asigura o afișare frumoasă a elementelor dintr-o clasă putem defini metodele __str__ și __repr__ ambele având ca rol returnarea unui string reprezentativ pentru instanța curentă. **Când apelăm print(obiect) se scrie ce returnează __str__. Când apelăm print(lista_de_obiecte) se afișează lista aplicând repr pentru fiecare obiect.** Dacă apelăm str(obiect) obținem stringul returnat de __str__, iar cu repr(obiect) stringul returnat de __repr__.

```
class Cls:
```

```
    def __init__(self, aa, bb):
        self.a=aa
        self.b=bb

    def __str__(self):
        return "a={} b={}".format(self.a,self.b)
    def __repr__(self):
        return "({}, {})".format(self.a,self.b)

c1=Cls(2,5)
print(c1) #a=2 b=5
print(str(c1)) #a=2 b=5
print(repr(c1)) #(2, 5)
c2=Cls(3,3)
c3=Cls(4,1)
print([c1,c2,c3]) #[(2, 5), (3, 3), (4, 1)]
```

Operatori

Se pot defini și operatori pentru elementele unei clase. De exemplu putem defini operatori de egalitate sau care să determine că un element se află într-o relație de ordine față de altul. Pentru clasa de mai sus, am putea considera că elementele întâi se ordonează după proprietatea *a*, apoi după *b*. Am definiti operatorii:

- __eq__ operația de egalitate
- __lt__ operatorul "<"
- __le__ operatorul "<="
- __gt__ operatorul ">"
- __ge__ operatorul pentru ">="

```
class Cls:
    def __init__(self, aa, bb):
        self.a=aa
        self.b=bb
    def __eq__(self,elem):
        return (self.a, self.b)==(elem.a,elem.b)
    def __lt__(self,elem):
        return (self.a, self.b)<(elem.a,elem.b)
    def __le__(self,elem):
```

```
        return (self.a, self.b)<=(elem.a,elem.b)
    def __gt__(self,elem):
        return (self.a, self.b)>(elem.a,elem.b)
    def __ge__(self,elem):
        return (self.a, self.b)>=(elem.a,elem.b)

c1=Cls(2,5)
c2=Cls(2,5)
c3=Cls(2,4)
print(c1<c3) #False
print(c1 >= c3) #True
```

Proprietăți și metode de clasă

Proprietățile clasei se definesc direct în clasă, de obicei la inceput (dar nu e obligatoriu).
Metodele clasei vor fi precedate de decoratorul @classmethod.
Instanțele pot accesa proprietăți și metode de clasă.
În momentul în care o instanță încearcă să modifice o proprietate de clasă prin scrierea instanta.proprietate_clasa=valoare, nu se va modifica proprietatea clasei ci se va crea o proprietate a instanței cu acelasi nume. Din acel moment instanța nu mai poate accesa (In mod direct) decât propia proprietate cu acel nume:

```
class Cls:
    n = 100
    def __init__(self, aa, bb):
        self.a=aa
        self.b=bb

    @classmethod
    def incrementeaza_n(cls):
        cls.n+=1


print(Cls.n) #100
c1=Cls(2,5)
print(c1.n) #100
c1.n=17
print(c1.n) #17
c2=Cls(2,5)
```

```
print(c2.n) #100
print(Cls.n) #100
Cls.incrementeaza_n()
print(Cls.n) #101
print(c1.n) #17
c1.incrementeaza_n()
print(Cls.n) #102
print(c1.n) #17
```

Module utile

Modulul math

Modulul math este folosit pentru funcții matematice necesare frecvent:

- floor(numar) - returneaza partea întreagă inferioară
- ceil(numar) - returnează partea întreagă superioară
- sqrt(numar) - returnează rădăcina pătrată a numărului
- funcții trigonometrice sin(numar), cos(numar) etc.

Modulul time

Uneori avem nevoie să calculăm cât a durat o anumită zonă de cod. Pentru asta putem folosi funcția time() din modulul time. E important de știut că funcția doar returnează câte secunde au trecut de la o dată de referință (1 Ianuarie 1970, ora 0, UTC+0). Funcția time() returnează un număr rațional (deci nu se referă doar la secunde întregi). Se poate folosi pentru a calcula cât timp durează o anumită operație:

```
import time
t1=time.time()
functie_de_evaluat()
t2=time.time()
print(t2-t1)
```

Modulul queue

Cu ajutorul acestui modul se pot crea diverse tipuri de cozi. Cozile pe care le vom folosi sunt

- Queue - coada obișnuită - ordinea în care sunt băgate elementele în coadă este si ordinea în care sunt extrase

- PriorityQueue - coada de priorități. Elementele sunt extrase din coadă în ordinea priorității (cu cât elementul este mai mic în coadă, cu atât e considerat cu prioritate mai mare)

Pentru obiectele de tip coadă există următoarele metode:

- constructorul, cu parametrul numeric maxsize, care indică numărul maxim de elemente în coadă
- empty() - returnează True dacă este goală coada și False în caz contrar
- full() - returnează True dacă este plină coada (a ajuns la maxsize) și False în caz contrar
- qsize() - returnează dimensiunea cozii
- put(element, block=True, timeout=None) - adaugă un element în coadă (nu vom folosi ceilalți 2 parametri). Dacă nu se dau și alte argumente (precum block, timeout), metoda put() poate bloca firul de execuție, dacă în acel moment coada este plină
- get(block=True, timeout=None) - extrage un element din coadă (nu vom folosi ceilalți 2 parametri). Dacă nu se dau și alte argumente (precum block, timeout), metoda get() poate bloca firul de execuție, dacă în acel moment coada este goală

```
import queue
coada= queue.Queue(maxsize=3)
print(coada.empty()) # True
coada.put(10)
coada.put(7)
print(coada.qsize()) # 2
coada.put(5)
print(coada.full()) # True
#coada.put(9) - va bloca programul deoarece coada este plina
print(coada.get()) # 10
print(coada.qsize()) # 2
print(coada.get()) # 7
print(coada.get()) # 5
```

Exemplu cu PriorityQueue (observați ordinea în care sunt extrase elementele; primul extras este mereu cel mai mic):

```
import queue
coada=queue.PriorityQueue()
coada.put(5)
coada.put(1)
coada.put(7)
print(coada.get()) # 1
print(coada.get()) # 5
print(coada.get()) # 7
```

Modulul heapq

Cu ajutorul modulului heapq puteam crea o structură heap pe baza unei liste, folosind metoda heapify(lista). Metoda heapify poate schimba ordinea elementelor în listă pentru a reprezenta un heap. Putem folosi structura heap ca pe o coadă de priorități. Cu heappush(lista,element), adăugăm un element în listă fără a strica structura de heap. Metoda heappop(lista) șterge și returnează cel mai mic element din listă.

```
import heapq
l=[5,10,1,9,4,7]
heapq.heapify(l)
print(l)
heapq.heappush(l,55)
print(l)
elem=heapq.heappop(l)
print("Am scos ", elem)
print(l)
```

Afișează:

```
[1, 4, 5, 9, 10, 7]
[1, 4, 5, 9, 10, 7, 55]
Am scos  1
[4, 9, 5, 55, 10, 7]
```

Prin urmare, **heapq poate fi folosit pentru a simula o coadă de priorități**.

Type hints

Python nu este un limbaj orientat pe tip și nu este necesar să specificăm tipul de date al variabilelor. Totuși, pentru a fi mai ușor de înțeles codul, uneori se simte nevoia indicării tipului variabilelor sau parametrilor unei funcții.
Pentru a specifica un type hint, după numele variabilei/parametrului se scriu ":" urmat de numele tipului.

```
x:float = 2.5
```

În cazul obiectelor de tip secvența, se pun între paranteze drept tipurile elementelor, de exemplu:

```
l1:list[int]=[1,4,10,5]
d1:dict[str,int]={}
d1["A"]=5
```

Atenție! Type hint nu aruncă erori în cazul în car în variabilele respective se pun valori diferite de tipul așteptat. De exemplu codul de mai jos nu aruncă nicio eroare:

```
#valoare gresita conform type hint; dar codul nu arunca erori
l2:list[int]=["abc",14,23]
x2:float=[1,2]
```

Pentru a specifica tipul returnat de funcții, după paranteza parametrilor, se scrie simbolul "->" urmat de tipul returnat:

```
def nrVoc(sir: str) -> int:
    s=0
    for x in sir:
        s+= x in "aeiou"
    return s

print(nrVoc("abceaa")) # 4

def prefixe(sir: str) -> list[str]:
    return [sir[:i] for i in range(1,len(sir))]

print(prefixe("abcdef")) # ['a', 'ab', 'abc', 'abcd', 'abcde']
```

Dacă funcția nu returnează nimic, se scrie după "->" None.

```
def afisare() -> None:
    print("un text frumos")
```

Pentru o variabila în care vrem să punem o instanță a unei clase, folosim drept type hint chiar numele clasei. Dacă variabila sau parametrul se găsește în cadrul unei metode in clasă, atunci type hint-ul cu numele clasei trebuie pus ca șir (altfel va da eroare fiindcă el consideră că nu e definită încă clasa):

```
class Clasa1:
    n=0
    def __init__(self, a:int, b:str) -> None:
        self.a=a
        self.b=b
        Clasa1.n+=1

    def __eq__(self, val: 'Clasa1') -> bool:
        return self.a==val.a and self.b==val.b
```

```
a:Clasa1=Clasa1(10,"abc")
```

În cazul în care avem o variabilă chiar de tip clasă atunci trebuie să importăm Type din pachtul typing, ca în exemplul de mai jos (pentru clasa Clasa1 definită mai sus):

```
from typing import Type
def afisInfo(c: Type[Clasa1]):
    print(c.n)
    aux=c(1,"abc")
    print(c.n)

afisInfo(Clasa1)
```

va afișa:

```
0
1
```

Prinderea erorilor

Erorile se aruncă cu instrucțiunea `raise`, urmată de tipul erorii (poate fi și o eroare definită de programator).
Codul care poate arunca erori se scrie într-un bloc `try`.
Dacă blocul de cod din `try` aruncă o eroare atunci se caută blocul `except` cu tipul de eroare care se potrivește cu eroarea aruncată si se excută codul din acel bloc `except`.
În cazul în care blocul `try` nu a aruncat vreo eroare, și blocul `try` are un bloc `else`, asociat, se execută codul din blocul `else`.
Dacă blocul `try` are și un bloc `finally` asociat, codul din blocul `finally` se execută întotdeauna (indiferent dacă s-a aruncat vreo eroare sau nu).

```
class Nu100(Exception):
    """Eroare definita de programator"""
    pass

def impartire(l, ind, y):
    try:
        if ind==100:
            raise Nu100
        x=l[ind]/y
    except ZeroDivisionError:
        print("Impartire la zero")
```

```
    except IndexError:
        print("Nu exista indicele")
    except Nu100:
        print("Nu se accepta indicele 100")
    else:
        print(x)
    finally:
        print("Se afișeaza mereu\n ---------")

impartire([1,4,3], 1, 0)
impartire([1,4,3], 10, 2)
impartire([1,4,3], 100, 2)
impartire([1,4,3], 1, 2)
```

va afișa:

```
Impartire la zero
Se afișeaza mereu
 ---------
Nu exista indicele
Se afișeaza mereu
 ---------
Nu se accepta indicele 100
Se afișeaza mereu
 ---------
2.0
Se afișeaza mereu
 ---------
```

Tehnici de căutare

Se folosesc pentru probleme care pot fi abstractizate la un graf (orientat sau neorientat). Presupun existența unui nod de început (nodul start) și unul sau mai multe noduri scop (la care vrem să ajungem din nodul start).

BreadthFirst

Pași:

1. Se pune nodul start într-o coadă.
2. Repetitiv:

a. dacă nu este coada vidă, se extrage primul nod din coadă și se verifică dacă este scop, caz în care se returnează o soluție. Dacă am ajuns la numărul de soluții dorit, ne oprim.
b. expandăm nodul și adăugăm succesorii săi în coadă, cu condiția ca succesorii să nu fi fost deja vizitați pe ramurile din arbore corespunzătoare lor (atenție e posibil să fi fost vizitați pe o altă ramură din arbore, dar acest lucru nu contează, vizitarea nu se consideră la nivel global ci doar pe drumul curent al acelui succesor).

DepthFirst

1. Se pune nodul start într-o stivă.
2. Repetitiv:
    a. Dacă vârful stivei este nod scop, afișăm o soluție. Dacă am ajuns la numărul de soluții dorit, ne oprim.
    b. Pentru vârful stivei dacă nu i s-au generat succesorii deja, adăugăm succesorii săi nevizitați **pe drumul curent** din stivă și verificăm dacă e nod scop, caz în care se returnează o soluție. Dacă am ajuns la numărul de soluții dorit, ne oprim.
    c. Dacă vârful stivei nu are succesori sau au fost deja generați și procesați, este eliminat din stivă.

Exerciții laborator 1

Atenție! Acestea sunt niște exerciții generale care servesc drept ghid și pot fi schimbate sau înlocuite de profesorul de laborator. Este necesar să verificați ce variantă de exerciții ați primit în oră la semigrupa la care ați participat.

1. Definiți o clasa NodArbore, care va reprezenta un nod dintr-un arbore de cautare, cu câmpurile: informatie, parinte (care e un obiect de tip NodArbore). In constructor, vom defini pentru parinte ca valoare implicită None. Clasa NodArbore va avea următoarele metode:
    a. drumRadacina(self) care va returna o listă cu toate nodurile ca obiecte (nu informatia nodurilor) de la rădăcină până la nodul curent
    b. vizitat(self) care verifică dacă nodul a fost vizitat pe drumul curent (deci nu în tot arborele) caz în care returnează True, altfel (dacă nu a fost vizitat) False. Reformulat: dacă informația se mai găsește în drumul de dinaintea nodului curent, returnăm True.
    c. funcția __repr__ care va returna un string continand informatia nodului, urmata de un spatiu, urmat de o paranteză în care se află tot drumul de la rădăcină până

la acel nod). De exemplu "c (a->b->c)" unde *c* e informația nodului curent și *a,b,c* sunt informațiile nodurilor din drumul de la rădăcină(*a*) către el.

d. funcția __str__ care va returna un string doar cu informația nodului

Definiți o clasă Graf, în care se va memora un graf (alegeți voi dacă prin listă de vecini, matrice de adiacența, sau listă de noduri și muchii), inclusiv informația nodului start și nodurile scop (date ca lista de informații scop). Veți defini pentru ea:

a. un constructor prin care se transmit informațiile grafului.
b. o metoda scop(self, informatieNod) care primește o informație de nod și verifică dacă e nod scop
c. o metoda (care va fi folosită în algoritmi pentru generarea arborelui de căutare), numită **succesori(self, *nod*)** care primește un nod al arborelui de parcurgere și parcurge nodurile adiacente din graf, returnand o lista de obiecte de clasa Nod ce reprezinta succesori direcți ai nodului (care vor fi fii în arborele de căutare), care nu au fost vizitati pe ramura curentă. Toți succesorii vor avea, evident, parintele egal cu *nod*

2. Implementați tehnica de căutare Breadthfirst, folosind clasele Nod și Graf definite mai sus. Se va citi de la tastatură (sau se da intr-o variabila) un număr NSOL de soluții. Se vor afișa primele NSOL soluții.
3. Implementați tehnica de căutare DepthFirst, folosind clasele Nod și Graf definite mai sus, în mod recursiv. Se va citi de la tastatură un număr NSOL de soluții. se vor afișa primele NSOL soluții.
4. Implementați tehnica de căutare DepthFirst, folosind , folosind clasele Nod și Graf definite mai sus, în mod nerecursiv. Se va citi de la tastatură un număr NSOL de soluții. se vor afișa primele NSOL soluții.