Pe calculatoarele personale trebuie să vă instalați o versiune cât mai recentă de **Python 3** (https://www.python.org/downloads/).

Ca IDE recomandăm să vă instalați **PyCharm** (https://www.jetbrains.com/pycharm/download/).

Cuprins

Despre sintaxa Python:

- Blocurile de cod se grupează folosind același nivel de indentare pentru toate instrucțiunile din acel bloc (în loc de folosirea acoladelor {} cum era în C/C++). Pentru indentare se folosesc 4 spații (recomandabil) sau un tab. Trebuie ca în întreg fișierul să se folosească aceeași indentare (fie câte 4 spații, fie câte un tab).
- Nu se folosește ”;” la finalul instrucțiunilor (ca în C/C++), ci doar între instrucțiuni, atunci când se scriu mai multe pe același rând. Dar pentru claritatea codului, nu este recomandat acest lucru, ci fiecare instrucțiune se scrie pe un alt rând.
- Python este case-sensitive (se face diferența dintre literă mică/mare).

Comentarii:

- Pe un singur rând: se folosește ”**#**” (în loc de ”//” ca în C/C++) și tot ce este în dreapta pe acel rând se consideră comentariu.
- Pe mai multe rânduri: se include tot comentariul între câte 3 ghilimele/apostrofuri (**”””**comentariu…**”””** sau **’’’**comentariu… **’’’**) (în loc de: /* comentariu… */ ca în C/C++).
- În PyCharm, pentru a (de)comenta rapid o porțiune de cod, selectați acele rânduri și apăsați tastele ”ctrl + /”.

Citirea datelor de la tastatură:

- Se apelează funcția ”**input**” fără parametru sau cu un parametru de tip *string*, care va fi afișat pe ecran (stringurile se includ între ghilimele/apostrofuri). De la tastatură se va introduce inputul dorit, apoi se apasă Enter.
- Funcția ”input” returnează mereu un *string*, deci rezultatul trebuie apoi transformat în alt tip de date, atunci când este nevoie.

```
x = input()
y = input(”y= ”)          # în C/C++: cout<<”y= ”; cin>>y;
z = input(’z= ’)
nr_intreg = int(input(”numarul intreg este: ”))
nr_real = float(input(”numarul real este: ”))
```

Afișarea datelor pe ecran:

- Se apelează funcția ”**print**”: `print(*objects, sep=' ', end='\n')`
- În loc de ”`*objects`” se pun unul sau mai multe obiecte (cu virgulă între ele) care trebuie afișate.
- Parametrul ”**sep**” (de la ”separator”) este opțional. Dacă se afișează mai multe obiecte cu un singur apel al funcției ”print”, între ele se va afișa implicit câte un spațiu. Dacă doriți altceva, trebuie să-i atribuiți o valoare (neapărat de tip *string*) parametrului ”sep”.
  Exemplu: `sep=", "` (virgulă și spațiu), `sep=";"`, `sep="\n"` (rând nou), `sep=""` (nimic) etc.
- Parametrul ”**end**” este opțional. Implicit, la finalul afișarii se va trece la un rând nou. Dacă doriți ca după afișare să pună altceva în loc de rând nou, trebuie să-i atribuiți o valoare (neapărat de tip *string*) parametrului ”end”.
  Exemplu: `end=", "` (virgulă și spațiu), `end=";"` etc.

```
print(3, 4.5, True, "ana")    # se vor afișa cu spațiu între ele
print("alt print")            # se va afișa pe rând nou
```

sau

```
print(3, 4.5, True, "ana", sep=",", end=";")
     # se vor afișa cu ”,” între ele, iar la final cu ”;”
print("alt print")  # se va afișa pe același rând, după ”;”
```

Variabile:

- Variabilele nu trebuie declarate (nu au tip de date static), ci doar **inițializate** cu o valoare (tipul de date se stabilește automat la atribuire). Atribuirea se face cu instrucțiunea ”**=**”.
- Orice dată (valoare) este un *obiect*. Variabilele sunt **referințe** către obiecte.
- **Numele unei variabile** poate conține litere mari, litere mici, simbolul underscore (“_”) și cifre (dar NU poate să înceapă cu o cifră). ***Exemplu:*** var, Var, VAR, _var, myVar, my_Var, var3.
- Pentru a afla *adresa* din memorie a unei variabile ”var”: `id(var)`
- Pentru a afla *tipul de date* al unei variablile ”var”: `type(var)`

Tipuri de date:

- Tipurile de date sunt **clase**.

1) **Clasa ”NoneType”** conține doar valoarea ”**None**” (se folosește pentru a verifica existența unui obiect).
2) *Tipuri numerice:*

a) **Clasa ”bool”** conține doar valorile ”**True**” și ”**False**” (atenție, se scriu mereu cu prima literă mare).

b) **Clasa ”int”** conține numere întregi de dimensiune oricât de mare (cât permite memoria calculatorului).

c) **Clasa ”float”** conține numere reale (corespunde tipului ”double” din C/C++). Se folosește ”.” între partea întreagă și zecimale (ex: 3.75). Se permite și scrierea: 17e3 (adică $17 * 10^3$).

d) **Clasa ”complex”** conține numere complexe: a+bj, unde a și b sunt numere (întregi sau reale), a reprezintă partea reală, iar b reprezintă partea imaginară. Atenție, nu se pune ”*” între b și j, iar valoarea lui b trebuie obligatoriu specificată, chiar daca este 1. Ex: 4+5j, 3+1j, 2j, complex(4,5).

3) *Tipuri secvențiale:*
   - tupluri (clasa “tuple”), liste (clasa “list”), șiruri de caractere (clasa “str”)
   - bytes / bytesarray
4) *Tipuri mulțime:* set, frozenset
5) *Tipul dicționar*: dict
6) *Tipuri funcționale*

Operatori (numerici, relaționali, logici, pe biți, condițional):

a) operatori numerici:

+, -, *, / (împărțire cu virgulă), // (împărțire ”întreagă”), %, ** (ridicare la putere).

Atenție! În Python nu există operatorii ++ și --. Dar există +=, -=, *= etc.

a//b = cel mai mare întreg care este mai mic sau egal decât a/b

a%b = a – (a//b)*b

a**b = a^b

x = y ** 0.5 # x este radical din y

Obs: Pentru a folosi funcția radical, trebuie la începutul fișierului cu codul (.py) să aveți instrucțiunea

```
import math
```

apoi pentru a folosi functia:

```
x = math.sqrt(y)
```

b) operatori relaționali:

<, <=, >=, > (NU pentru numere complexe)

==, != (compară *valorile*)

is, is not (compară *adresele* din memorie) (x is y) <=> (id(x) == id(y))

in, not in (pentru testarea apartenenței unei valori la o colecție)

- Se permite înlănțuirea operatorilor relaționali: `if a <= y < b:` …
- Pentru compararea numerelor reale: `abs(x-y)<=1e-9` (NU folosiți x==y)

c) operatori logici (booleeni): not, and, or

False <=> 0, 0.0, 0+0j; colecție_vidă: (), [], {}, “”, ‘’ etc

True <=> restul

- Operatorii logici se evaluează prin scurtcircuitare (adică nu se evaluează toate condițiile dacă se știe deja rezultatul: False pentru “and” și True pentru “or”).
- NU folosiți apeluri de funcții în condiții!
- $x\ or\ y = \begin{cases} x, daca\ x\ este\ True \\ y, daca\ x\ este\ False \end{cases}$ 5 or 3.7 == 5
- $x\ and\ y = \begin{cases} y, daca\ x\ este\ True \\ x, daca\ x\ este\ False \end{cases}$ 5 and 0.0 == 0.0

d) operatori pe biți:

~ (negare), & (și), | (sau), ^ (XOR/sau exclusiv)

<<, >> (deplasări/shiftări pe biți către stânga sau dreapta)

~	0	1
	1	0

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

- Proprietăți ^ (XOR):
a) x ^ x = 0, x este bit
b) x ^ 0 = x, x este bit sau număr

x = (x<<b) <=> x = x * (2**b) b = număr natural

x = (x>>b) <=> x = x // (2**b)

- Operatorii pe biți acționează asupra reprezentării binare interne; se execută rapid, pe procesor.

- Aplicații:

1) Testarea parității unui număr (mai rapid decât cu %)

```
if x & 1 == 1:
    print("Numar impar.")
else:
    print("Numar par.")
```

2) Interschimbare variabile

```
x = x ^ y
y = x ^ y    # = (x^y)^y = x^(y^y) = x^0 = x
x = x ^ y    # = (x^y)^x = y^(x^x) = y^0 = y
```

e) operatorul condițional:

`valoare1 if conditie==True else valoare2`

max = a if a > b else b

Un operator are următoarele proprietăți:

- **aritatea** (numărul de operanzi) 1/2/3 -> operator unar/binar/ternar
- **prioritatea/precedența**
  https://docs.python.org/3/reference/expressions.html#operator-precedence
  Atenție, excepție: `2**-1 = 0.5` (nu necesită paranteze: 2**(-1))
  `x == (not y)` (obligatoriu cu paranteze, altfel dă eroare de sintaxă)
- **asociativitatea** -> majoritatea au de la stânga la dreapta
  Atenție, operatorul ** are asociativitate de la dreapta la stânga.
  2**3**2 = 2**(3**2) = 2**9 = 512 (NU (2**3)**2=8**2=64)

Instrucțiuni (if, for, while, …):

1) instrucțiunea de atribuire:

```
x = 100
x = y = 100
a, b, c = 1, 2, 3
```
=> interschimbare: `x, y = y, x`

2) instrucțiunea de decizie: cuvinte cheie "**if**", "**elif**", "**else**".

- Atentie la indentări și la ":" de la finalul acelor instrucțiuni
- Nu este nevoie să puneți condițiile între paranteze.
- Pot fi 0 sau oricât de multe ramuri cu "elif", iar ramura cu "else" poate lipsi.
- În Python NU există instrucțiunea "switch/case".

```
if conditie_1:
    bloc_instructiuni_1
elif conditie_2:
    bloc_instructiuni_2
elif conditie_3:
    bloc_instructiuni_3
else:
    bloc_instructiuni_4
```

3) instrucțiunea repetitivă cu test inițial: cuvinte cheie "**while**".

- Atenție la indentare și la ":" de la finalul instrucțiunii.
- În Python NU există instrucțiunea "do… while".

```
while conditie:
    bloc_instructiuni
```

4) instrucțiunea repetitivă cu număr fix de iterații/pași: cuvinte cheie ”**for**”, “**in**”.

- Atenție la indentare și la ”:” de la finalul instrucțiunii.

```
for variabila in colectie_iterabila:
    bloc_instructiuni
```

a) parcurgere caractere dintr-un string

```
s = "test"
for x in s:
    print(x)
```

b) parcurgere elemente dintr-o listă

```
L = [1,2,3]
for x in L:
    print(x)
```

c) parcurgere interval de numere întregi cu funcția “**range**”

```
range(b) <=> range(0,b)     => 0, 1, ..., b-1
range(a,b) <=> range(a,b,1) => a, a+1, a+2, ..., b-1
                                sau nimic daca a>=b
range(a,b,pas)  => a, a+pas, a+2*pas, ..., (a+k*pas)!=b
    cu a<b si pas>0, sau a>b si pas<0, sau nimic altfel
```

```
for i in range(10, 20, 2):        #10, 12, 14, 16, 18
  print(i)
for j in range(5, -5, -1):        #5, 4, 3, 2, 1, 0, -1, -2, -3, -4
  print(j)
```

5) instrucțiunea ”continue” aflată în cadrul unui bloc “for” sau “while”:

- Se ignoră instrucțiunile de după și se trece la următoarea iterație din for/while.

```
for x in range(n+1):
    if x%2 == 0:
        continue # se ignora instructiunile de dupa ”continue”,
                 # se trece la urmatoarea iteratie din for
    print(x) # se vor afisa doar nr impare: 1,3,5, ... <=n
```

6) instrucțiunea “break” aflată în cadrul unui bloc “for” sau “while”:

- Se iese din ciclul for/while cel mai apropiat.

```
for x in range(5):
    for y in range(10):
        if y > x:
            print("\nx =", x, "\n")
            break # se iese din for-ul cu y
        print("y="+str(y), end = " ")
```

7) instrucțiunea “pass”:

- Instrucțiunea vidă, nu face nimic.

```
if conditie_1:
    if conditie_2:
        bloc_instructiuni_1
    else:
        pass
else:
    bloc_instructiuni_2
```

8) clauza ”else” după un ciclu “for” sau ”while”:

- Se execută dacă ciclul a fost parcurs integral, fără să fie intrerupt de instrucțiunea “break”.

```
a, b = 20, 50
for x in range(a, b+1):
    d = 2
    while d <= x//2:
        if x % d == 0:
            break # intrerupe while-ul
        d = d + 1
    else: # daca x nu a avut niciun divizor d
        print(f"Cel mai mic numar prim intre {a} si {b} este: {x}.")
        break # intrerupe for-ul
else:
    print(f"Nu exista niciun numar prim intre {a} si {b}.")
```

Șiruri de caractere (clasa “str”):

- Se scriu între una sau 3 perechi de ghilimele/apostrofuri:
`"text", 'text', """text""", '''text'''` (sunt șiruri echivalente)
- Secvențe escape: “\n” (rând nou), “\t” (tab/4 spații), “\\” (backslash “\”).

- Dacă șirul conține ghilimele/apostrofuri, atunci preferabil se încadrează între celălalt simbol sau se folosește escape (“\”) înaintea simbolului din interiorul șirului.

```
"It's Monday." # in loc de 'It\'s Monday.'
'She said "hello" to me.' # in loc de "She said \"hello\" to me."
```

- Șirul se încadrează între 3 perechi de ghilimele/apostrofuri dacă în interior are și ghilimele și apostrofuri și nu dorim să folosim “\”, sau atunci când lucrăm cu șiruri lungi care ocupă mai multe rânduri (*atenție*, dacă dorim ca șirul să nu conțină “\n”, trebuie să punem “\” la finalul rândului ca să ignore linia nouă).

```
"""It's "sir" to you.""" # in loc de "It's \"sir\" to you."
                         # sau de 'It\'s "sir" to you.'
#s1 si s2 sunt echivalente cu 'test simplu'
s1 = "test "\
     "simplu"   # obligatoriu cu "\", altfel eroare de sintaxa
s2 = """test \
simplu"""   # fara "\" => s2 == 'test \nsimplu'
```

- Șirurile sunt **imutabile** => după crearea obiectului, *valoarea* conținută la acea adresă nu mai poate fi modificată, dar *referința* poate fi modificată (variabilei i se atribuie un alt obiect, creat la altă adresă, cu noua valoare).

```
s = "test"; print(id(s))
print(s.upper(), s, id(s)) # s NU isi modifica nici valoarea, nici adresa
s = s.upper(); print(s, id(s)) # s isi modifica valoarea si adresa
```

- Șirurile sunt **iterabile** => caracterele din șir au atribuite poziții (**indecși**). Dacă n este lungimea șirului, de la stânga la dreapta pozițiile se numerotează cu 0, 1, 2, …, n-1, iar de la dreapta la stânga pozițiile se numerotează cu -1, -2, -3, …, -n. Deci poziția k≥0 este aceeași cu poziția k-n = -(n-k).

	-9	-8	-7	-6	-5	-4	-3	-2	-1	<--
-->	0	1	2	3	4	5	6	7	8	9
“	H	a	l	l	o	w	e	e	n	“

*Gândiți un index ca fiind un cursor plasat **înaintea (la stânga)** caracterului de pe acea poziție. (Observați alinierea din tabel.)*

- Accesarea elementelor dintr-un șir:

a) prin **index** pozitiv sau negativ

b) prin "**slice**" (secvență dintr-un șir)

sir[poz] => caracterul de pe poziția poz (între indecșii poz și poz+1)
sir[a:b] <=> sir[a:b:1] => subșirul care începe pe poziția a și se termină *înainte* de poziția b
(sau șirul vid dacă a≤b)
sir[a:] => subșirul care începe pe poziția a și se termină la finalul șirului
sir[:b] => subșirul care începe la începutul șirului și se termină *înainte* de poziția b
sir[:] <=> sir => întreg șirul
sir[a:b:pas] => pozițiile a, a+pas, a+2*pas…., până *înainte* de poziția b
cu a<b si pas>0 sau a>b si pas<0 (sau șirul vid altfel)
sir[::-1] => întreg șirul inversat, de la sfârșit la început

```
s = "Halloween"
print(s[5], s[-4], s[5:6], s[-4:-3], s[5:-3], s[-4:6]) # toate echivalente: "w"
print(s[::2]) # din 2 in 2: "Hloen"
print(s[::-1]) # sirul invers: "neewollaH"
print(s[1:7:-2], len(s[1:7:-2])) # sirul vid, lungime 0
    # (pt ca nu exista intervalul de la 1 la 7 cu pas negativ)
```

- Modificare / ștergere / inserare într-un șir:

```
s = "Halloween"; s1 = "_LOVE_"; s2 = "..."; a=3; b=7; print(s[a:b]) # "lowe"
# --> Inlocuire subsir dintre indecsi a si b cu s1
# s[a:b]=s1 # asa NU => TypeError: 'str' object does not support item assignment
t1 = s[:a] + s1 + s[b:]; print(t1) # asa DA --> "Hal_LOVE_en"

# --> Stergere subsir dintre indecsi a si b (inlocuire cu sirul vid)
# s[a:b]="" # asa NU => TypeError: 'str' object does not support item assignment
t2 = s[:a] + s[b:]; print(t2) # asa DA --> "Halen"

# --> Inserare s2 intre indecsi a si a (adica intre pozitiile a-1 si a)
# s[a:a]=s2 # asa NU => TypeError: 'str' object does not support item assignment
t3 = s[:a] + s2 + s[a:]; print(t3) # asa DA --> "Hal...loween"
```

- Operatori:

+ (concatenare), ***** (multiplicare = concatenare repetată)
in, **not in** (testarea apartenenței unui subșir într-un șir)

```
s = "abcd" + "123"; print(s) # "abcd123"
s = "abc." * 3; print(s) # "abc.abc.abc."
print("abc" in "bcabcd", "c" not in "ab", "" in "abc") # True True True
```

- Funcții predefinite:

Funcția	Ce tip de date și valoare returnează	Explicații și exemple
`len(sir)`	`int` (numărul de caractere din șir)	`len("") => 0` `len("test") => 4`

`str(valoare)`	`str` (transformă valoarea în șir)	`str(123) => "123"`
		`str(True) => "True"`
		
		`x=3 ; print("x=" + str(x))`
		`# print("x="+x)` Eroare: str + int

`min(sir)`
`max(sir)`
	`str` (cel mai mic/mare caracter din șir, conform codurilor ASCII)	`sir = "abcABC"`
		`print(min(sir),max(sir)) => "A" "c"`
		`# ord("A") < ord("a")`

`ord(caracter)` `chr(int)`	`int` (codul ASCII) `str` (caracter pt acel cod)	`print(ord("A")) # 65` `print(chr(65)) # "A"`

- Metode specifice șirurilor de caractere (funcții din clasa "str"):

Atenție, deoarece șirurile sunt imutabile, apelarea oricărei metode NU modifică stringul apelant.

1) *Formatarea șirurilor:*

→ *(metoda veche, nerecomandabilă)* cu %-formatare, asemănător ca în C/C++. Se recomandă folosirea celor două metode de mai jos, de la caz la caz în funcție de avantajele fiecăreia.

→ *(începând cu Python 2.6)* cu metoda "**șir.format()**": Șirul apelant conține perechi de acolade în locul porțiunilor care vor fi înlocuite cu valorile trimise ca parametri funcției "format".

a) formatare pozițională (valorile între {} se completează în ordinea parametrilor)
```
s = "Ana are {} mere {} si {} pere {}.".format(5, "rosii", 5, "galbene")
print(s) # "Ana are 5 mere rosii si 5 pere galbene."
```
b) cu menționarea indexului parametrului (începând cu 0)
```
s = "Ana are {1} mere {0} si {1} pere {2}.".format("rosii", 5, "galbene")
print(s) # "Ana are 5 mere rosii si 5 pere galbene."
```
c) cu menționarea numelui parametrului
```
s = "Ana are {nr} mere {culoare_mere} si {nr} pere {culoare_pere}."\
    .format(nr=5, culoare_mere="rosii", culoare_pere="galbene")
print(s) # "Ana are 5 mere rosii si 5 pere galbene."
```

→ *(începând cu Python 3.6)* cu **f-șiruri** (**f"..... {}..... {} ....."**): Observați caracterul **f** lipit înaintea șirului. Direct între perechile de acolade se scriu valorile/expresiile sau variabilele ale căror valori trebuie puse în acele porțiuni ale șirului.

```
nr = 5; culori = ["rosii", "galbene"]
s = f"Ana are {nr} mere {culori[0]} si {1+2*2} pere {culori[1]}."
print(s) # "Ana are 5 mere rosii si 5 pere galbene."
```

2) Pentru *transformări la nivel de caracter*: metodele **lower()**, **upper()**, **title()**, **capitalize()**, **swapcase()** returnează *șiruri* (obiecte diferite, aflate la altă adresă, față de șirul apelant).

```
s = "proGrAMaRea alGOriTMilor"
print("lower: " + s.lower()) # "programarea algoritmilor"
print("upper: " + s.upper()) # "PROGRAMAREA ALGORITMILOR"
print("title: " + s.title()) # "Programarea Algoritmilor"
print("capitalize: " + s.capitalize()) # "Programarea algoritmilor"
print("swapcase: " + s.swapcase()) # "PROgRamArEA ALgoRItmILOR"
print("sir nemodificat: " + s) # "proGrAMaRea alGOriTMilor"
```

3) Pentru *clasificare*: metodele returnează o valoare booleană (*True* / *False*).

- **islower()**, **isupper()** verifică dacă șirul apelant are toate literele mici, respectiv mari și dacă există cel puțin o literă în șir; **istitle()** verifică dacă în fiecare cuvânt al șirului apelant prima literă (dacă există vreuna) este mare iar restul literelor sunt mici și dacă există cel puțin o literă în șir.

```
print("_ana face 3ani maine #3".islower()) # True
print("_ANA FACE 3ANI MAINE #3".isupper()) # True
print("_Ana Face 3Ani Maine #3".istitle()) # True
```

- **isalpha()** verifică dacă toate caracterele din șirul apelant sunt litere și dacă există cel puțin una; **isdecimal()**, **isdigit()**, **isnumeric()** verifică dacă toate caracterele sunt cifre/numerice și există cel puțin unul; **isalnum()** verifică dacă șirul apelant conține doar caractere alfanumerice (litere și cifre), cel puțin un caracter.

```
print("Nota".isalpha(), "1234".isdigit()) # True True
print("nota 10".isalnum()) # False (contine si spatiul)
```

- mai există și **isspace()** (șirul conține doar: " ", "\t", "\n"), **isidentifier()** (dacă șirul respectă regulile denumirii variabilelor), **iskeyword()** (dacă șirul este un cuvânt cheie din Python: “if”, “else”, “for”, “in”, “while”, “def”, “None”, “True”, “and”, etc.), **isprintable()**.

4) Pentru *căutarea unui subșir* în șirul apelant:

a) șir.**count**(subșir, start = 0, end = len(șir)) --> returnează numărul de apariții *disjuncte* (fără suprapuneri) ale lui “subșir” în slice-ul “șir[start:end]”.

```
print("aaaaaa".count("aa")) # 3
print("abcdabcdab".count("abcd")) # 2
print("abc.abc.abc".count("abc",2)) # "c.abc.abc" --> 2
print("abc.abc.abc".count("abc",2, 9)) # "c.abc.a" --> 1
```

b) → șir.**find**(subșir, start = 0, end = len(șir)) sau **rfind** --> returnează indexul din “șir” de unde începe *prima* (sau *ultima* pt “rfind”) apariție a lui “subșir” în slice-ul “șir[start:end]” sau **-1** dacă nu apare.

→ șir.**index**(subșir, start = 0, end = len(șir)) sau **rindex** --> returnează indexul din “șir” de unde începe *prima* (sau *ultima* pt “rindex”) apariție a lui “subșir” în slice-ul “șir[start:end]” sau **ValueError** dacă nu apare.

```
s = "alabala"; t = "ala"; u = "sala"
print(s.find(t), s.find(t,3), s.find(t,2,7), s.find(u)) # 0 4 4 -1
```

c) → șir.**startswith**(prefix, start = 0, end = len(șir)) --> returnează True/False dacă slice-ul “șir[start:end]” *începe* cu “prefix” sau nu.

→ șir.**endswith**(sufix, start = 0, end = len(șir)) --> returnează True/False dacă slice-ul “șir[start:end]” *se termină* cu “sufix” sau nu.

```
s = "alabala"; t = "ala"; u = "sala"
print(s.startswith(t), s.startswith(t,4,7), s.startswith(u)) # True True False
print(s.endswith(t), s.endswith(t,0,3), s.endswith(u)) # True True False
```

5) Pentru *înlocuirea*/ștergerea *unui subșir*:

șir.**replace**(old, new, max = -1) --> returnează un nou șir (obiect diferit față de cel apelant), obținut prin înlocuirea cu “new” a *primelor* maxim “max” apariții ale lui “old” în “șir”. Dacă max = -1, se înlocuiesc *toate* aparițiile.

```
s = "aaaa"; old = "a"; new = "A"
for max in [-1,2,100]: print(s.replace(old, new, max)) # AAAA AAaa AAAA
```

Obs: Dacă “new” = “” (șirul vid), se *șterg* aparițiile.

```
print("abc.abc.abc.".replace("a","",2)) # "bc.bc.abc."
```

6) Pentru eliminarea anumitor caractere de la începutul/sfârșitul șirului:

șir.**strip**(chars = ” ”) sau **lstrip** sau **rstrip**--> returnează un nou șir (obiect nou), obținut prin eliminarea tuturor aparițiilor caracterelor din șirul dat ca parametru (acest șir de caractere este considerat o *mulțime de caractere*; implicit este caracterul spațiu) de la *începutul* și *sfârșitul* lui “șir” (sau doar de la *început*/*sfârșit* pentru lstrip/rstrip).

```
print(['  ab '.lstrip(), ' cd '.rstrip(), ' ef '.strip()]) #['ab  ', ' cd', 'ef']
print(['++--+-ab+'.lstrip('+'), '++--+-ab+'.lstrip('+-')]) #['--+-ab+', 'ab+']
```

7) Pentru *separarea*/*alipirea* șirurilor:

→ șir.**split**(sep = None, maxsplit = -1) --> returnează o *listă de șiruri* obținută prin spargerea lui “șir” la separatorul “sep” de maxim “maxsplit” ori (sau de șir.count(sep) ori dacă maxsplit = -1). Dacă

sep=None, implicit spargerea șirului se face după *spațiu*, iar în listă NU vor apărea *șirurile vide*. Metoda **rsplit** face primele maxsplit spargeri numărând aparițiile lui "sep" de la *finalul* lui "șir".

```
s = "Ana numara: 1,  2,  3."
for L in [s.split(), s.split(" "), s.split(","),\
          s.split(", ", 1), s.rsplit(" ", 1)]:
    print(len(L),L)
# 5 ['Ana', 'numara:', '1,', '2,', '3.']
# 7 ['Ana', 'numara:', '1,', '', '2,', '', '3.']
# 3 ['Ana numara: 1', '  2', '  3.']
# 2 ['Ana numara: 1', ' 2,  3.']
# 2 ['Ana numara: 1,  2, ', '3.']
```

→ sep.**join**(colecție_iterabilă_șiruri) --> returnează un *șir de caractere* obținut prin inserarea separatorului "sep" între toate elementele șiruri din colecția iterabilă și concatenarea lor. Dacă există în colecție vreun element care nu e de tip **str**, metoda join returnează **TypeError**.

```
sir = "Ana are mere"; sep = " "
print(sir == sep.join( sir.split(sep) )) # True
sir = ".".join(str(x) for x in range(5)); print(sir) # "0.1.2.3.4"
sir = ", ".join(["1","2","3"]); print(sir) # "1, 2, 3"
sir = ";".join(("1","2","3")); print(sir) # "1;2;3"
sir = "--".join("abcd"); print(sir) # "a--b--c--d"
```

Liste (clasa "list"):

- Listele se scriu între paranteze drepte. Elementele listei se scriu cu virgulă între ele și pot avea tipuri de date diferite.

- Listele sunt **mutabile** (au dimensiune dinamică, putem adăuga/modifica/șterge elemente), **iterabile** (prin indecși, elemente), **indexate** (de la 0).

a) Crearea unei liste

```
L = [] # lista vida
L = [1, 2, "Ion", 3.14, 7, 8, True] # elemente cu tipuri de date diferite
L = [[1, 2], [3, 4, 5]] # L[0]==[1,2], L[1]==[3,4,5], L[0][1]==2
L = ["Popescu Ion", 131, [9,10,10]]; nr_note = len(L[2]) # nr_note == 3
```

→ Creare listă folosind secvențe de inițializare (**"list comprehensions"**):

- Forma generală: **L = [element for …]** sau **L = [element for … if …]**
- Între [] punem întâi o expresie care reprezintă forma elementelor care vor face parte din lista creată. Apoi pot exista una sau mai multe clauze "for" una după alta. Eventual pot exista și clauze "if" (care pot conține mai multe condiții grupate cu "and", "or"), care pot fi intercalate cu clauzele "for". *Atenție:* rezultatul ar fi același dacă am scrie clauzele (de la stânga la dreapta) una sub alta, imbricate și indentate corespunzător, iar apoi am adăuga "element" la finalul listei L, inițial vidă.
- Expresia poate fi oricât de simplă/complexă: o constantă, o variabilă, o expresie aritmetică, o colecție (listă, tuplu, set, dicționar), o altă listă formată tot cu "list comprehension" etc.

```
L = [a**2 for a in range(6)]; print(L) # [0, 1, 4, 9, 16, 25]
L = [a for a in range(10) if a%2 == 0]; print(L) # [0, 2, 4, 6, 8]

L1 = [2, 3, 4, 5, 6, 7]; L2 = [11, 12, 13, 14, 15]
L3 = [float(str(a)+"."+str(b)) for a in L1 if a%2==0 for b in L2 if b%a!=0]
print(L3) # [2.11, 2.13, 2.15, 4.11, 4.13, 4.14, 4.15, 6.11, 6.13, 6.14, 6.15]
L4 = [(a,b) for a in L1 for b in L1 if a<=b and a+b==10]
print(L4) # [(3, 7), (4, 6), (5, 5)]

Lin = 3; Col = 4;
M = [[x*Col+y+1 for y in range(Col)] for x in range(Lin)]
print(M) # [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

```
L1 = [M[i][j] for i in range(Lin) for j in range(Col)]
print(L1) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
L2 = [elem for linie in M for elem in linie]
print(L2) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
# atentie, "list comprehensions" imbricate pt T
T = [[linie[k] for linie in M] for k in range(Col)] # obtinem matr. transpusa
print(T) # [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

b) Accesarea elementelor unei liste

(la fel ca la șiruri, pentru detalii vedeți pag.8)

- prin indecși pozitivi/negativi
- prin secvență de indecși (slice)

→ Modificare / ștergere / inserare într-o listă:

```
a = [1,2,3,4,5,6,7]; print(a) # [1, 2, 3, 4, 5, 6, 7]
b = a[1:5]; print(b) # [2,3,4,5]
a[0]=10; a[1:3]=[16,17,18,19]; print(a) # [10,16,17,18,19,4,5,6,7] modificare
a[1:5]=[]; print(a) # [10, 4, 5, 6, 7] stergere
a[2:2]=[100, 200, 300]; print(a) # [10, 4, 100, 200, 300, 5, 6, 7] inserare
```

→ Instrucțiunea “del”:

```
del a[1]; print(a) # [10, 100, 200, 300, 5, 6, 7] stergere element
del a[:4]; print(a) # [5, 6, 7] stergere slice
del a # stergere obiect => "print(a)" da eroare NameError
```

c) Operatori

d) Funcții predefinite

e) Metode specifice pentru liste (funcții din clasa “list”)

Rezumat proprietăți tipuri de date

bool, int, float --> imutabil

str --> imutabil, iterabil (prin indecși, caractere), indexat

list --> mutabil, iterabil (prin indecși, elemente), indexat
tuple --> imutabil, iterabil (prin indecși, elemente), indexat

set --> mutabil, iterabil (prin elemente), neindexat (nu păstrează ordinea elementelor)
frozenset --> imutabil, iterabil (prin elemente), neindexat (?? nu păstrează ordinea elementelor)

dict --> mutabil, iterabil (prin chei, valori, elemente), indexat_pe_chei (nu păstrează ordinea elementelor)

→ Parcurgeri colecții:

……..

```
from random import randint, choice
print(randint(1,100))
print(choice([10,20,30,40,50]))
```