

CURS 2

Instrucțiuni

Pentru a putea controla fluxul unui program (ordinea în care se vor executa operațiile dorite), majoritatea limbajelor de programare folosesc *instrucțiuni de control*. Aceste instrucțiuni pot fi, de exemplu, *instrucțiuni de decizie* (cu ajutorul cărora se stabilește dacă o anumită operație se efectuează sau nu în funcție de o anumită condiție), *instrucțiuni repetitive* (cu ajutorul cărora se efectuează de mai multe ori o anumită operație) etc.

În limbajul Python nu există delimitatori pentru *blocurile de instrucțiuni* (cum sunt acoladele în limbajele C/C++), ci gruparea mai multor instrucțiuni se realizează prin indentarea lor în raport de instrucțiunea căreia i se subordonează.

În limbajul Python sunt definite următoarele instrucțiuni de control:

1. *instrucțiunea de atribuire*

- Spre deosebire de limbajele C/C++, atribuirea nu este un operator, ci este o instrucțiune!
- Instrucțiunea de atribuire poate avea următoarele forme:
 - *atribuire simplă* ($x = 100$);
 - *atribuire multiplă* ($x = y = 100$);
 - *atribuire compusă* ($x, y, z = 100, 200, 300$).
- Două variabile se pot interschimba prin atribuirea compusă $x, y = y, x$!
- O atribuire de forma $x = x \text{ operator expresie}$ poate fi scrisă prescurtat sub forma $x \text{ operator} = \text{expresie}$, unde operator este un operator aritmetic sau pe biți binar. De exemplu, instrucțiunea $x = x + y * 10$ poate fi scrisă prescurtat sub forma $x += y * 10$.
- În limbajul Python nu sunt definiți operatorii ++/-- din limbajele C/C++!

2. *instrucțiunea de decizie / alternativă if*

- Instrucțiunea de decizie este utilizată pentru a executa o instrucțiune (sau un bloc de instrucțiuni) doar în cazul în care o expresie logică este adevărată:

```
if expresie_logică:
    instrucțiune
```

Exemplu (maximul dintre două numere):

```
a = int(input("a = "))
b = int(input("b = "))
maxim = a
if b > maxim:
    maxim = b
print("Maximul dintre", a, "si", b, "este", maxim)
```

- Instrucțiunea alternativă este utilizată pentru a alege executarea unei singure instrucțiuni (sau a unui bloc de instrucțiuni) dintre două posibile, în funcție de valoarea de adevăr a unei expresii logice:

```

if expresie_logică:
    instrucțiune_1
else:
    instrucțiune_2

```

Exemplu (maximul dintre două numere):

```

a = int(input("a = "))
b = int(input("b = "))
if a > b:
    maxim = a
else:
    maxim = b
print("Maximul dintre", a, "si", b, "este", maxim)

```

- Instrucțiunile alternative imbricate se pot scrie mai concis folosind instrucțiunea `elif`, așa cum se poate observa din exemplul următor:

<pre> a = int(input("a = ")) if a < 0: print("Strict negativ") else: if a == 0: print("Zero") else: print("Strict pozitiv") </pre>	<pre> a = int(input("a = ")) if a < 0: print("Strict negativ") elif a == 0: print("Zero") else: print("Strict pozitiv") </pre>
---	---

- În limbajul Python nu este definită o instrucțiune alternativă multiplă, cum este, de exemplu, instrucțiunea `switch` din limbajele C/C++!

3. instrucțiunea repetitivă `while`

- Instrucțiunea `while` este o instrucțiune repetitivă cu test inițial, fiind utilizată pentru a executa o instrucțiune (sau un bloc de instrucțiuni) cât timp o expresie logică este adevărată:

```

while expresie_logică:
    instrucțiune

```

Exemplu (suma cifrelor unui număr natural):

```

n = int(input("n = "))
aux = n
sc = 0
while aux != 0:
    sc = sc + aux % 10
    aux = aux // 10
print("Suma cifrelor numarului", n, "este", sc)

```

- În limbajul Python nu este definită o instrucțiune repetitivă cu test final, cum este, de exemplu, instrucțiunea `do...while` din limbajele C/C++!

4. instrucțiunea repetitivă for

- Instrucțiunea for este utilizată pentru a accesa, pe rând, fiecare element dintr-o secvență (de exemplu, un șir de caractere, o listă etc.), elementele fiind considerate în ordinea în care apar în secvență:

for *variabilă în secvență:*
 instrucțiune

Exemple:

```
sir = "test"
for c in sir:
    print(c, end=" ")

#Se va afișa: t e s t
```

```
lista = [1, 2, 3]
for x in lista:
    print(x, end=" ")

#Se va afișa: 1 2 3
```

- Pentru a genera secvențe numerice de numere întregi asemănătoare unor progresii aritmetice se poate utiliza funcția `range([min], max, [pas])`, care va genera, pe rând, numerele întregi cuprinse între valorile `min` (inclusiv) și `max` (exclusiv!!!) cu rația `pas`. Parametrii scriși între paranteze drepte sunt opționali, iar parametrul opțional `pas` se poate specifica doar dacă se specifică și parametrul opțional `min`. Dacă pentru parametrul `min` nu se specifică nicio valoare, atunci el va fi considerat în mod implicit ca fiind 0.

Exemple:

```
range(6)          => 0, 1, 2, 3, 4, 5
range(2, 6)       => 2, 3, 4, 5
range(2, 11, 3)   => 2, 5, 8
range(2, 12, 3)   => 2, 5, 8, 11
range(7, 2)       => secvență vidă (deoarece 7 > 2)
range(7, 2, -1)   => 7, 6, 5, 4, 3
```

5. instrucțiunea continue

- Instrucțiunea continue este utilizată în cadrul unei instrucțiuni repetitive pentru a termina forțat iterația curentă (dar nu și instrucțiunea repetitivă!), continuându-se direct cu următoarea iterație.

Exemplu:

```
for i in range(1, 11):
    if i%2 == 0:
        continue
    print(i, end=" ")

#Se va afișa: 1 3 5 7 9
```

6. instrucțiunea break

- Instrucțiunea break este utilizată în cadrul unei instrucțiuni repetitive pentru a termina forțat executarea instrucțiunii respective.

Exemplu: Se citește un șir de numere care se termină cu valoarea 0 (care se consideră că nu face parte din șir, ci este doar un marcaj al sfârșitului său). Să se afișeze suma numerelor citite.

```
s = 0
while True:
    x = int(input("x = "))
    if x == 0:
        break
    s += x
print("Suma numerelor citite: ", s)
```

Atenție, în acest program instrucțiunea `s += x` ar fi putut fi scrisă și înaintea instrucțiunii `if` fără a-i afecta corectitudinea! Totuși, în alte cazuri (de exemplu, dacă s-ar fi cerut produsul numerelor citite), acest lucru ar fi dus la afișarea unui rezultat eronat!

7. instrucțiunea else

- Instrucțiunea `else` poate fi adăugată la sfârșitul unei instrucțiuni repetitive, instrucțiunile subordonate ei fiind executate doar în cazul în care instrucțiunea repetitivă se termină natural (condiția dintr-o instrucțiune `while` devine falsă sau o instrucțiune `for` a parcurs toate elementele unei secvențe), ci nu din cauza întreruperii sale forțate (utilizând o instrucțiune `break`).

Exemple:

- a) Se citește un șir format din n numere întregi. Să se verifice dacă toate numerele citite au fost pozitive sau nu.

```
n = int(input("n = "))
for i in range(n):
    x = int(input("x = "))
    if x < 0:
        print("A fost citit un număr negativ!")
        break
else:
    print("Toate numerele citite au fost pozitive!")
```

- b) Să se afișeze cel mai mic număr prim cuprins între două numere naturale a și b sau un mesaj corespunzător în cazul în care nu există niciun număr prim cuprins între a și b .

```
a = int(input("a = "))
b = int(input("b = "))

for x in range(a, b+1):
    for d in range(2, x//2+1):
        if x % d == 0:
            break
```

```

else:
    #instrucțiunea for d in ... s-a terminat natural,
    #deci numărul x nu are divizori proprii,
    #ceea ce înseamnă că x este un număr prim
    print("Cel mai mic numar prim cuprins între", a,
          "și", b, "este", x)

    #numărul x este cel mai mic număr prim cuprins între
    #a și b, deci nu are rost să mai continuăm căutarea
    #altuia și opresc forțat instrucțiunea for x in ...
    break
else:
    #instrucțiunea for x in ... s-a terminat natural, deci
    # nu a fost găsit niciun număr prim cuprins între a și b
    print("Nu exista niciun numar prim cuprins între", a,
          "și", b)

```

8. instrucțiunea pass

- Instrucțiunea pass este o instrucțiune care nu are niciun efect în program (este similară unei instrucțiuni vide). Această instrucțiune se utilizează în cazurile în care sintactic ar fi necesară o instrucțiune vidă, deoarece în limbajul Python aceasta nu este definită.

Exemplu:

```

varsta = 10
if varsta <= 18:
    print("Junior")
elif varsta < 60:
    #nu prelucrăm informațiile despre persoanele
    #cu vârste cuprinse între 19 și 59 de ani
    pass
else:
    print("Senior")

```

Șiruri de caractere

Un *șir de caractere* este o secvență de caractere indexată de la 0, memorată folosind un obiect de tipul clasei str.

Limbajul Python folosește setul de caractere Unicode (<https://home.unicode.org/>), ceea ce permite utilizarea într-un program a caracterelor din aproape orice alfabet existent pe Glob (*internaționalizare*) și a simbolurilor specifice multor domenii (pentru mai multe detalii consultați pagina <https://docs.python.org/3/howto/unicode.html>).

Constantele de tip șir de caractere (*literal*) se reprezintă în două moduri:

- prin 'șir' sau "șir", dacă dorim ca șirul respectiv să fie considerat ca fiind scris pe o singură linie (sunt ignorate caracterele newline);
- prin '''șir''' sau """șir""", dacă dorim ca șirul respectiv să fie considerat ca fiind scris pe mai multe linii (nu sunt ignorate caracterele newline).

Exemplu:

```

1 sir_1 = "Acesta este, de fapt, \"
2         \"un sir scris \" \"
3         \" \"
4         \"pe o singura linie!\"
5
6 print(sir_1)
7
8 sir_2 = """Acesta este
9           un sir scris
10
11           pe mai multe linii linie!"""
12
13 print(sir_2)
14

```

Run: Test_curs

```

"C:\Program Files (x86)\Python38-32\python.exe" "C:/Users/Ours/One
Acesta este, de fapt, un sir scris pe o singura linie!
Acesta este
    un sir scris
    pe mai multe linii linie!
Process finished with exit code 0

```

Într-un șir de caractere se pot insera *secvențe escape*, la fel ca în limbajele C/C++: \n (linie nouă), \t (tab), \\ (backslash) etc. (pentru mai multe detalii consultați pagina <http://www.python-ds.com/python-3-escape-sequences>).

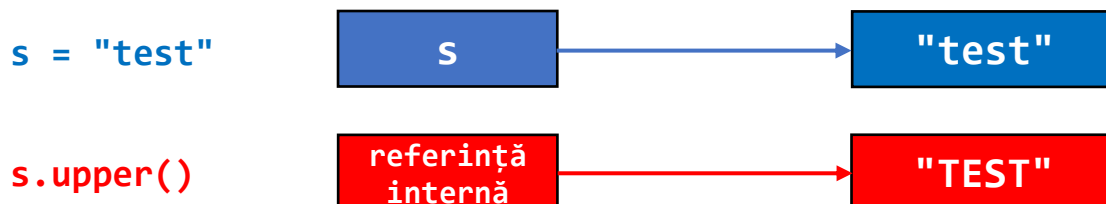
O proprietate foarte importantă a șirurilor de caractere o constituie faptul că sunt *imutabile*, respectiv valoarea unui obiect de tip șir de caractere nu mai poate fi modificată după crearea sa, ci se poate modifica doar o referință spre el. Din acest motiv, nicio metoda din clasa str nu va modifica efectiv șirul curent (cel care apelează metoda respectivă), ci va crea un nou șir! De exemplu, să considerăm următoarea secvență de cod, care va afișa șirul test:

```

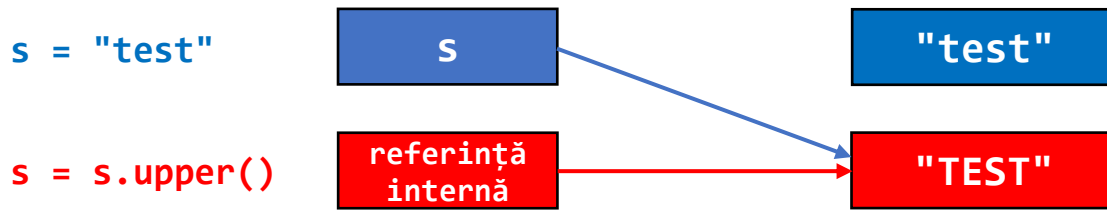
s = "test"
s.upper()
print(s)    #test

```

Practic, metoda upper() nu va transforma literele mici ale șirului s în litere mari, ci va genera un nou șir, obținut prin transformarea literelor mici ale șirului s în litere mari:

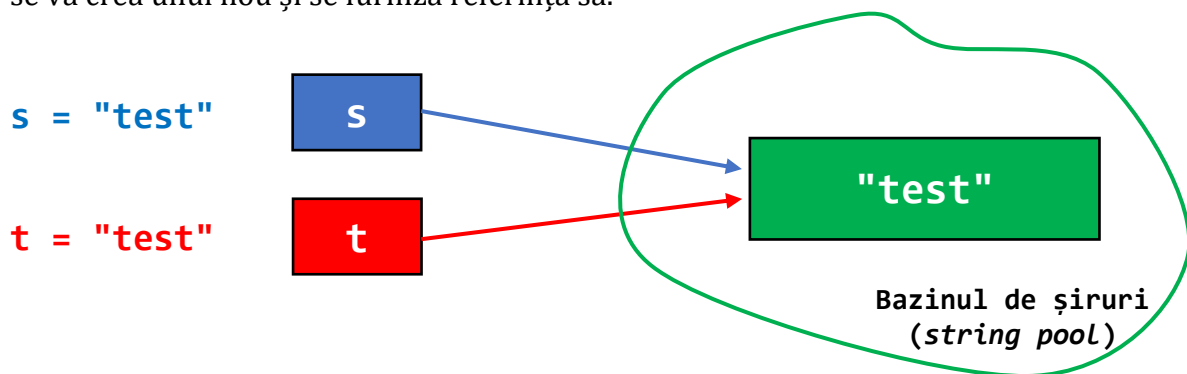


Pentru a modifica efectiv șirul s, trebuie să modificăm referința conținută de el în referința internă asociată șirului generat prin apelul s.upper():



Obiectul conținând șirul `"test"` va fi șters automat din memorie de *Garbage Collector* după ce nu va mai exista nicio referință activă spre el.

Pentru stocarea șirurilor de caractere în memoria internă, limbajul Python utilizează mecanismul *string interning*, prin care se stochează în memorie o singură copie a fiecărui șir de caractere distinct utilizat în cadrul unui program. Astfel, obiectele de tip șir de caractere sunt păstrate într-un bazin de șiruri (*string pool*), iar în momentul în care o variabilă este inițializată cu un șir constant se verifică dacă acesta există deja în bazin sau nu. În caz afirmativ, nu se mai creează un nou obiect de tip șir, ci variabila de tip șir va primi direct referința șirului existent din bazin, iar în cazul în care șirul nu există în bazin se va crea unul nou și se furniza referința sa.



Funcționarea corectă a mecanismul *string interning* se bazează pe imutabilitatea șirurilor de caractere, respectiv pe faptul că nu putem să modificăm conținutul unui șir după crearea sa. În caz contrar, dacă șirurile de caractere ar fi fost *mutable*, modificarea unui obiect de tip șir din bazinul de șiruri ar fi condus la modificarea implicită a tuturor șirurilor care conțin referința sa!

Utilizarea mecanismului *string interning* prezintă mai multe avantaje, cele mai importante fiind micșorarea spațiului de memorie utilizat pentru stocarea șirurilor și creșterea vitezei de comparare a lor. Fără utilizarea bazinului de șiruri, compararea a două șiruri de caractere `s` și `t` prin `s == t` necesită compararea caracterelor aflate pe aceleași poziții, deci complexitatea computațională va fi $\mathcal{O}(\text{len}(s))$. În cazul utilizării bazinului de șiruri, compararea `s == t` a două șiruri de caractere poate fi înlocuită cu compararea `s is t` a referințelor obiectelor asociate, ceea ce revine la compararea a două numere întregi (i.e., `id(s) == id(t)`), deci complexitatea computațională va fi doar $\mathcal{O}(1)$!!!

În *mod implicit*, un șir este salvat în bazinul de șiruri dacă face parte din sintaxa limbajului Python (i.e., șirul este un cuvânt cheie, denumirea unei funcții/clase/variabile etc.) sau are lungimea cel mult 1 (i.e., este șirul vid sau este format dintr-un singur caracter). Orice alt șir de caractere este salvat în bazinul de șiruri dacă îndeplinește următoarele condiții:

- lungimea sa este cel mult 4096;

```
s = "a" * 4096
t = "a" * 4096
print(s == t)    #True
print(s is t)    #True

s = "a" * 4097
t = "a" * 4097
print(s == t)    #True
print(s is t)    #False
```

- valoarea sa poate fi determinată în momentul compilării;

```
s = "test"
t = "te" + "st"
print(s == t)    #True
print(s is t)    #True

s = "test"
x = "st"
t = "te" + x
print(s == t)    #True
print(s is t)    #False
```

- toate caracterele sunt caractere ASCII (condiția nu este obligatorie în toate implementările Python).

În *mod explicit*, un șir poate salvat în bazinul de șiruri dacă se utilizează metoda `intern(șir)` din modulul `sys`:

```
import sys

s = "test"
x = "st"
t = "te" + x
print(s is t)    #False

s = "test"
x = "st"
t = sys.intern("te" + x)
print(s is t)    #True
```

Atenție, metoda `sys.intern(șir)` este lentă, deci trebuie utilizată doar dacă acest lucru este absolut necesar (de exemplu, dacă în programul respectiv se vor efectua multe comparații de șiruri care nu sunt implicit supuse procesului de *string interning*)!

Mai multe detalii referitoare la mecanismul *string interning* din limbajul Python găsiți în pagina <https://stackabuse.com/guide-to-string-interning-in-python/>.

Accesarea elementelor unui șir de caractere

Elementele unui șir de caractere pot fi accesate în mai multe moduri, astfel:

a) *prin indici pozitivi sau negativi*

În limbajul Python, oricărei secvențe (*mulțime iterabilă*) de lungime n îi sunt asociați atât indici pozitivi, cuprinși între 0 și $n - 1$ de la stânga spre dreapta, cât și indici negativi, cuprinși între $-n$ și -1 de la stânga la dreapta.

Exemplu: pentru șirul $s = \text{"programare"}$ avem asociați următorii indici:

	0	1	2	3	4	5	6	7	8	9
s	'p'	'r'	'o'	'g'	'r'	'a'	'm'	'a'	'r'	'e'
	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Astfel, al patrulea caracter din șir (litera 'g'), poate fi accesat atât prin $s[3]$, cât și prin $s[-7]$. Atenție, accesarea unui caracter va fi de tip *read-only* (doar în citire), deoarece șirurile sunt imutabile! De exemplu, instrucțiunea $s[6] = 'd'$ o să genereze, în momentul executării programului, eroarea "TypeError: 'str' object does not support item assignment". Totuși, caracterul aflat într-un șir s pe o poziție p validă (cuprinsă între 0 și $\text{len}(s) - 1$) poate fi modificat sau șters construind un nou șir a cărui referință va înlocui referința inițială:

```
s = s[:p] + "caracter_nou" + s[p+1:]      (modificare)
s = s[:p] + s[p+1:]                       (ștergere)
```

b) *prin secvențe de indici pozitivi sau negativi (slice)*

Expresia $\text{șir}[st:dr]$ extrage din șirul dat subșirul cuprins între pozițiile st și $dr-1$, dacă $st \leq dr$, sau șirul vid în caz contrar.

Exemple:

```
s[1:4] = s[-9:-6] = "rog"
s[:4] = s[0:4] = "prog"
s[4:] = "ramare"
s[5:2] = "" (deoarece 5 > 2)
s[5:2:-1] = "arg"
s[:] = "programare" (întregul șir)
s[::-1] = "eramargorp" (șirul oglindit)
s[-9:4] = "rog"
```

Operatori pentru șiruri de caractere

În limbajul Python sunt definiți următorii operatori pentru manipularea șirurilor de caractere:

a) *operatorul de concatenare: +*

Exemplu: $\text{"un"} + \text{" " + "exemplu"} = \text{"un exemplu"}$

- b) *operatorul de multiplicare* (concatenare repetată): `*`
Exemplu: `"test" * 3 = 3 * "test" = "testtesttest"`
- c) *operatorii pentru testarea apartenenței*: `in`, `not in`
Exemplu: expresia `"est" in "atestat"` va avea valoarea `True`
- d) *operatorii relaționali*
Exemplu: expresia `"Popa" <= "Popescu"` va avea valoarea `True`, deoarece șirurile se vor compara lexicografic și `"a" < "e"`.

Funcții predefinite pentru șiruri de caractere

În limbajul Python sunt predefinite mai multe funcții (*built-in functions* – <https://docs.python.org/3/library/functions.html>), dintre care unele pot fi utilizate pentru mai multe tipuri de date. De exemplu, funcția `len(secvență)` va furniza numărul de elemente dintr-o secvență (iterabil), indiferent dacă aceasta este o listă sau un șir de caractere. Funcțiile predefinite care se pot utiliza pentru șiruri de caractere sunt următoarele:

- a) `len(șir)`: furnizează numărul de caractere unui șir (lungimea șirului)
Exemplu: `len("test") = 4`
- b) `str(expresie)`: transformă valoarea expresiei într-un șir de caractere
Exemplu: `str(123) = "123", str(1+2==3) = "True"`
- c) `min(șir) / max(șir)`: furnizează caracterul minim/maxim în sens lexicografic (alfabetic) din șirul respectiv (atenție, literele mari sunt mai mici, din punct de vedere lexicografic, decât literele mici!)
Exemplu: `min("Examene") = "E", max("examene") = "x"`
- d) `ord(caracter)`: furnizează codul Unicode asociat caracterului respectiv
Exemplu: `ord("A") = 65`
- e) `chr(număr)`: furnizează caracterul având codul Unicode respectiv
Exemplu: `chr(65) = "A"`

Metode pentru prelucrarea șirurilor de caractere

Metodele pentru prelucrarea șirurilor de caractere sunt, de fapt, metodele încapsulate în clasa `str`. Așa cum am precizat anterior, șirurile de caractere sunt *imutabile*, deci metodele respective nu vor modifica șirul curent, ci vor genera un nou șir care va conține rezultatul prelucrării șirului inițial. Din acest motiv, metodele clasei `str` se vor apela, de obicei, în cadrul unei expresii de forma următoare: `șir = șir.metodă(parametrii)`. De exemplu, pentru a transforma toate literele mici ale unui șir în litere mari, vom utiliza expresia `șir = șir.upper()`.

În continuare, vom prezenta mai multe metode pentru prelucrarea șirurilor de caractere, cu observația că parametrii scriși între paranteze drepte sunt opționali:

a) metode pentru formatare

- `strip([caractere])`: furnizează șirul obținut din șirul curent prin eliminarea celui mai lung prefix și celui mai lung sufix formate doar din caracterele indicate prin parametrul opțional de tip șir.

Exemplu:

```
s = "www.example.org"
s = s.strip("...misgrown?!")
print(s)                                #example
```

Dacă metoda este apelată fără parametru, atunci metoda va furniza șirul obținut din șirul curent prin eliminarea celui mai lung prefix și celui mai lung sufix formate doar din spații albe:

```
s = "    www.python.org    "
s = s.strip()
print(s)                                #www.python.org
```

- `center(lățime, [caracter])`: furnizează șirul obținut prin centrarea șirului inițial folosind caracterul indicat.

Exemplu:

```
s = "www.python.org"
s = s.center(30, ".")
print(s)                                #.....www.python.org.....
```

Dacă metoda este apelată fără parametru, atunci metoda va furniza șirul obținut prin centrarea șirului inițial folosind spații:

```
s = "www.python.org"
s = s.center(30)
print(s)                                #          www.python.org
```

- `format(număr variabil de parametrii)`: furnizează șirul obținut prin înlocuirea câmpurilor variabile din șirul curent (marcate prin { }) cu parametrii metodei.

Parametrii metodei `format(...)` pot fi accesați de câmpurile variabile din șirul curent prin mai multe modalități:

- *accesare secvențială* (primul câmp variabil din șir va fi înlocuit cu primul parametru al metodei, al doilea câmp va fi înlocuit cu al doilea parametru etc.):

```
s = "Ana are {} mere {}!"
s = s.format(5, "roșii")
print(s)                                #Ana are 5 mere roșii!
```

- *accesare pozițională* (fiecare câmp variabil din șir va conține numărul de ordine al parametrului metodei cu care va fi înlocuit):

```
s = "Ana are {1} mere {0}!"
s = s.format("roșii", 3+4)
print(s)                                #Ana are 7 mere roșii!
```

- *accesare prin parametri cu nume* (fiecare câmp variabil din șir va conține numele parametrului metodei cu care va fi înlocuit):

```
s = "Ana are {numar} mere {culoare}!"
s = s.format(culoare="roșii", numar=3)
print(s)                                     #Ana are 3 mere roșii!
```

Metoda `format(...)` permite realizarea unor formătări complexe ale șirurilor de caractere, descrise în mod detaliat în pagina <https://docs.python.org/3.8/library/string.html#formatstrings>.

b) *metode pentru transformări la nivel de caracter*

- `lower()`: furnizează șirul obținut din șirul inițial prin transformarea tuturor literelor mari în litere mici;
- `upper()`: furnizează șirul obținut din șirul inițial prin transformarea tuturor literelor mici în litere mari;
- `swapcase()`: furnizează șirul obținut din șirul inițial prin transformarea tuturor literelor mici în litere mari și invers;
- `title()`: furnizează șirul obținut din șirul inițial prin transformarea primei litere a fiecărui cuvânt în literă mare, restul literelor fiind transformate în litere mici;
- `capitalize()`: furnizează șirul obținut din șirul inițial prin transformarea primei sale litere în literă mare, restul literelor fiind transformate în litere mici.

Example:

```
s = "amINTiri DIN CopilăRIE..."
print(s)                                     #amINTiri DIN CopilăRIE...
print(s.lower())                             #amintiri din copilărie...
print(s.upper())                             #AMINTIRI DIN COPILĂRIE...
print(s.swapcase())                          #AMinTIRI dIn cOPILĂrie...
print(s.title())                             #Amintiri Din Copilărie...
print(s.capitalize())                        #Amintiri din copilărie...
```

c) *metode pentru clasificare*

Metodele din această categorie verifică dacă toate caracterele șirului curent sunt de un anumit tip și returnează `True` sau `False`:

- `isascii()`: verifică dacă toate caracterele șirului sunt caractere ASCII (au codul cuprins între 0 și 127);
- `isalpha()`: verifică dacă toate caracterele șirului sunt litere;
- `isdigit()`: verifică dacă toate caracterele șirului sunt cifre;
- `isnumeric()`: verifică dacă toate caracterele șirului sunt caractere numerice;
- `isalnum()`: verifică dacă toate caracterele șirului sunt alfanumerice (i.e., sunt litere sau cifre);
- `isspace()`: verifică dacă toate literele din șir sunt spații albe (blank, tab etc.);
- `islower()`: verifică dacă toate literele din șir sunt litere mici, iar șirul poate să mai conțină și alte caractere care nu sunt litere (de exemplu, cifre sau semne de punctuație);

- `isupper()`: verifică dacă toate literele din șir sunt litere mari, iar șirul poate să mai conțină și alte caractere care nu sunt litere;
- `istitle()`: verifică dacă prima literă a fiecărui cuvânt este literă mare, iar restul literelor sunt mici.

Exemple:

```
s = "Ana are 123 de mere!!!"
print(s.isascii())           #True
print(s.isalpha())           #False

s = "Anu12020"
print(s.isalnum())           #True

s = "Ana Are 123 De Mere!!!"
print(s.istitle())           #True

s = "123\u00BD"
print(s.isdigit())           #False
print(s.isnumeric())         #True

s = "ANA ARE  mere și pere!"
print(s[3:10].islower())     #False
print(s[0:7].isupper())      #True
print(s[7:9].isspace())      #True
```

d) *metode pentru căutare*

- `count(subșir, [start], [stop])`: furnizează numărul aparițiilor disjuncte ale subșirului indicat în șirul curent. Parametrii opționali `start` și `stop` pot fi utilizați pentru a specifica poziții din șir între care să fie căutat subșirul respectiv (se va considera inclusiv poziția `start` și exclusiv poziția `stop`).

Exemplu:

```
s = "Hahahahahahaha..."
print(s.count("haha"))       #3

s = "Ana are mere în camerele casei sale."
print(s.count("mere"))       #2
print(s.count("mere", 15))   #1
```

- `find(subșir, [start], [stop])`: furnizează cel mai mic indice la care începe subșirul dat în șirul curent sau -1 dacă subșirul dat nu apare în șirul curent. Parametrii opționali `start` și `stop` pot fi utilizați pentru a specifica poziții din șir între care să fie căutat subșirul respectiv (se va considera inclusiv poziția `start` și exclusiv poziția `stop`).

Exemplu:

```
s = "Ana are mere în camerele casei sale."
print(s.find("mere"))        #8
print(s.find("pere"))        #-1
```

- `rfind(subșir, [start], [stop])`: furnizează cel mai mare indice la care începe subșirul dat în șirul curent sau -1 dacă subșirul dat nu apare în șirul curent. Parametrii opționali `start` și `stop` pot fi utilizați pentru a specifica poziții din șir între care să fie căutat subșirul respectiv (se va considera inclusiv poziția `start` și exclusiv poziția `stop`).

Exemplu:

```
s = "Ana are mere în camerele casei sale."
print(s.rfind("mere"))           #18
print(s.rfind("pere"))          #-1
```

- `startswith(prefix, [start], [stop])`: verifică dacă șirul curent are prefixul indicat sau nu.

Exemple:

```
s = "programare"
print(s.startswith("pro"))       #True
print(s.startswith("gram", 2))   #False
print(s.startswith("gram", 3))   #True
print(s.startswith("gram", 3, 6)) #False
```

- `endswith(sufix, [start], [stop])`: verifică dacă șirul curent are sufixul indicat sau nu.

Exemple:

```
s = "programare"
print(s.endswith("are"))         #True
print(s.endswith("mare", 7))     #False
print(s.endswith("mare", 4))     #True
print(s.endswith("mare", 4, 8))  #False
```

- `replace(subșir, subșir_nou, [max])`: furnizează șirul obținut din șirul curent prin înlocuirea tuturor aparițiilor subșirului indicat cu noul subșir. Parametrul opțional `max` poate fi utilizat pentru a înlocui doar primele `max` apariții ale subșirului dat cu noul subșir.

Exemple:

```
s = "Ana are ore de floretă cu Dorel!"
s = s.replace("ore", "in")
print(s)                                #Ana are in de flintă cu Dinl!

s = "Ana are ore de floretă cu Dorel!"
s = s.replace("ore", "in", 2)
print(s)                                #Ana are in de flintă cu Dorel!

s = "Ana are ore de floretă cu Dorel!"
s = s.replace("are", "va avea")
print(s)                                #Ana va avea ore de floretă cu Dorel!
```

```
s = "Ana are ore de floretă cu Dorel!"
s = s.replace("ore", "")
print(s)           #Ana are  de fltă cu D!
```

e) *metode pentru împărțirea/construirea unui șir în/din subșiruri:*

- `split(separator)`: furnizează o listă care conține subșirurile obținute din șirul curent considerând separatorul indicat (implicit, spațiile albe sunt considerate separatori).

Exemple:

```
s = "Ana are ore de floretă cu Dorel!"
w = s.split()
print(w)    #['Ana', 'are', 'ore', 'de', 'floretă', 'cu', 'Dorel!']

s = "Ana are ore de floretă cu Dorel!"
w = s.split("e")
print(w)    #['Ana ar', ' or', ' d', ' flor', 'tă cu Dor', 'l!']

w = s.split("ore")
print(w)    #['Ana are ', ' de fl', 'tă cu D', 'l!']
```

- `join(listă_subșiruri)`: furnizează șirul obținut prin concatenarea subșirurilor date, folosind ca separator șirul curent.

Exemple:

```
s = " ".join(["Ana", "are", "mere", "!"])
print(s)    #Ana are mere !

s = "Ana are ore de floretă cu Dorel!"
w = "...".join(s.split())
print(w)    #Ana...are...ore...de...floretă...cu...Dorel!
```

În afara metodelor prezentate mai sus, clasa `str` conține și alte metode utile pentru prelucrarea șirurilor de caractere. O prezentare detaliată a lor găsiți în pagina <https://docs.python.org/3/library/stdtypes.html?#string-methods>.