Șiruri de caractere – clasa str

Despre șiruri de caractere în Python. Creare, constante

Un tip de secvență în Python sunt șirurile de caractere, care se memorează folosind tipul de date (clasa) str.

Fiind o secvență, pentru un șir de caractere se aplică principiile, operatorii și funcțiile comune secvențelor prezentate deja pentru secvențe, dar există și o serie de funcții proprii clasei str. În cele ce urmează le vom detalia pe acestea, amintind însă și operațiile comune.

Constantele (literalii) de tip șir de caractere pot fi delimitate de:

- apostrof:

```
s = 'acesta este un sir'
```

- ghilimele:

```
t = "acesta este un alt sir"
```

- 3 apostrofuri sau 3 ghilimele, și atunci șirul se pot întinde pe mai multe linii (amintim ca astfel se puteau comenta mai multe linii in Python):

```
versuri = """A fost odata ca in povesti
A fost ca niciodata"""
print(versuri)
```

Un obiect de tip str poate fi creat și folosind constructorul **str**(), care permite și conversia unui alt tip de date la șir de caractere:

```
x = str(3.1415)
str(123) => "123", str(1+2==3) => "True"
```

Un obiect de tip str (un șir de caractere) este imutabil, ceea ce înseamnă că valoarea sa nu poate fi modificată. Astfel:

- o încercare de a modifica un caracter prin atribuiri de tipul

```
s[i] = c
```

va da eroare (**TypeError: 'str' object does not support item assignment**). De exemplu:

```
s = "acest sir"
s[0] = 'A'  #eroare, nu mai ajunge la afisare
print(s)
```

- metodele din clasa str care au ca scop efectuarea unor modificări (spre exemplu: conversie în litere mari, înlocuirea unui subșir) **nu modifică** valoarea șirului (obiectului) care apelează metoda, ci **returnează un șir nou**, de aceea se apelează de obicei sub forma:

```
s_nou = s.metoda()
```

Evident, **s_nou** poate fi chiar **s**, dar atunci nu se modifica obiectul referit inițial de s, ci variabila **s** va referi un obiect nou (iar vechiul obiect, dacă nu mai este referit, va fi șters din memorie de Garbage Collector).

Exemplu:

```
s = "acest sir"
id_initial_s = id(s)
s.replace("acest", "Alt")
print(s) #nemodificat
s = s.replace("acest", "Alt")
print(s)#modificat,
print( id(s), id_initial_s) #de fapt s-a creat obiect nou, cu
alt id
```

Limbajul Python folosește setul de caractere Unicode (https://home.unicode.org/), având la dispoziție astfel o gamă largă de caractere.

Pentru a insera caractere speciale într-un șir putem folosi secvențe escape, ca și în C/C++: https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals

```
\n newline
\t tab
\\
\'
\"
```

Exemple:

1. Tab

```
s = 'Programarea\talgoritmilor'
print(s)
```

2. Șir care conține în interior și apostrof sau ghilimele:

```
#s = 'That's it' ?!?!?!?!
#variante corecte:
s = "That's it"
print(s)
s = "That's it"
print(s)
s = 'That\'s it'
print(s)
s = """I say "That's it" again"""
print(s)
```

3. Folosirea caracterelor unicode speciale într-un șir: prefixam cu \u (cod hexa 16) sau \U (cod hexazecimal 32 biti) sau \N{numele lor}

```
s = "Aceasta este o  pisic\u0103 \N{Cat}"
print(s)
```

Accesarea elementelor. Parcurgere

Accesarea elementelor unui șir de caractere se face cu metodele deja discutate la secvențe.

Pentru un șir s amintim:

- primul element este s[0]
- un element din șir se poate accesa prin s[i], unde i este un indice care poate fi si negativ (s[-1] este ultimul caracter din șir, s[-len(s)] este primul caracter din șir)
- prin feliere (slicing) s[i:j:k] se pot accesa subsecvențe sau subșiruri cu pas k

Exercițiu. Care dintre următoarele expresii sunt adevărate pentru **`s = "Programarea"`**?

```
s[0] == s[0:1]
s[0] is s[0:1]
s[1] == s[1:3:2]
s[4] == s[-3]
s[4] is s[-3] #vezi sectiunea memorarea sirurilor
```

Pentru a parcurge elementele unui șir putem folosi instrucțiunea for, după cum am amintit și în secțiunea dedicată lucrului cu secvențe:

```
s = "un sir"
for x in s:
    print(x)
for i in range(len(s)):
    print(s[i])
```

Operatori

Pentru șiruri de caractere se pot folosi toți operatorii specifici secvențelor:

- de concatenare și multiplicare **`+`**, **`*n`**
- **`in`**, **`not in`**
- **`< , <=, >=, >, ==, is`**

Exemplu: **`print("un sir" < "alt sir")`** va afișa False (șirurile se compară lexicografic, nu după lungime)

Funcții și metode

În continuare vom prezenta o parte dintre metodele existente în clasa str. Mai multe detalii și o listă completă a metodelor din clasa str se găsesc în documentație: https://docs.python.org/3/library/stdtypes.html?#string-methods.

Funcții uzuale

Amintim funcțiile uzuale comune pentru subsecvențe și cele pentru lucrul cu codul unui caracter:

- **`len(s)`** – lungimea șirului s
- **`min(s)`, `max(s)`** – compararea caracterelor se face lexicografic (!, literele mari sunt mai mici lexicografic decât literele mici)

Exemple: **`min("Examene") = "E"`**, **`max("examene") = "x"`**

- **`ord(s)`**: primește ca parametru un sir de lungime unu (un caracter) și returnează codul Unicode asociat caracterului respectiv (TypeError dacă șirul are mai multe caractere sau este vid)

Exemplu: **`ord("A") = 65`**

- **`chr(număr)`**: returnează caracterul (ca șir de lungime 1) având codul Unicode primit ca parametru

Exemplu: **`chr(65) = "A"`**

Exemplu: Pentru o literă mică c (diferită de "z"), următoarea literă în alfabet este **`chr(ord(c) + 1))`**

Căutare și înlocuire

Amintim metodele comune secvențelor pentru **căutarea** unui cuvânt:

- **`s.index(x[,start[,end]])`** => returnează poziția primei apariții a lui **`x`** în **`s`**; dacă se precizează și parametri opționali, aceștia sunt interpretați ca la feliere, căutarea făcându-se începând cu poziția **`start`** dacă se transmite valoare pentru **`start`**, respectiv de la poziția **`start`** la **`end`** (exclusiv **`end`**) dacă au valori ambii parametri opționali; metoda aruncă eroarea **`ValueError`** dacă valoarea lui **`x`** nu este element în s
- **`s.count(sub[, start[, end]]))`** => returnează numărul de apariții ale șirului primit ca parametru în **`s`**; dacă se precizează și parametri opționali, aceștia sunt interpretați ca la feliere, numărarea făcându-se în subsecvența **`s[start:]`** dacă se transmite valoare pentru **`start`**, respectiv **`s[start:end]`** dacă au valori ambii parametri opționali

Detaliem în cele ce urmează funcții specifice clasei **`str`** pentru **căutare** într-un șir.

Pentru funcțiile care au ca parametri opționali **`start`** și **`end`** nu vom mai detalia semnificația acestora, fiind similară cu cea de la metodele **`index`** și **`count`**.

- **`s.rindex(x[,start[,end]])`** => similar cu **`index`**, dar determină ultima apariție (Caută de la dreapta la stânga)

- **`s.find(x[,start[,end]])`** => ca și **`s.index`** , dar **returnează `-1`** dacă nu găsește valoarea **`x`** în listă (nu dă eroare)
- **`s.rfind(x[,start[,end]])`** - "right" find (căutare de la dreapta) – ca și **`find`,** dar returnează indicele ultimei apariții

  Exemplu

```
s = 'Programarea'
print(s.find("a"))
print(s.find("z")) #nu da eroare, ca s.index
print(s.rfind('a'))
print(s.index("a"))
print(s.index("z")) #da eroare
```

- **`s.startswith(prefix [, [start [, [stop]])`** – verifică dacă șirul **`prefix`** este prefix al lui s (a lui **`s[start: stop]`** dacă se specifică și parametri opționali) și returnează **`True`**/**`False`**. Parametrul **`prefix`** poate fi și un tuplu de prefixe
- **`s.endswith`**

  Exemplu

```
if s.endswith('nt'):
    print('Fazan!')
if s.startswith(('a','e','i','o','u') ):
    print('incepe cu o vocala’)
if s.startswith(tuple("aeiouAEIOU") ):
    print('incepe cu o vocala')
```

Pentru **modificarea conținutului** unui șir (! se returnează șirul obținut, nu se modifică șirul care apelează metoda) în clasa str există, printre altele, următoarele metode:

- **`s.replace(old, new[, count])`** => returnează o **copie a lui s** în care toate aparițiile subsecvței **`old`** sunt înlocuite cu **`new`**; dacă este dat si parametrul opțional **`count`**, atunci sunt înlocuite doar primele **`count`** apariții ale lui old (!!! nu se modifică s, este imutabil)

  Exemplu:

```
t = s = 'Programarea algoritmilor'
s.replace('a', 'aaa')
print(s)
s = s.replace('a', 'aaa')
print(s)
print(t)
s = s.replace('o', '', 1)
print(s)
s = s.replace('rea', 're')
print(s)
```

- **`s.translate`** – pentru a face mai multe înlocuiri simultan – v. seminar și laborator

Transformare la nivel de caracter

- **`s.lower()`** - returnează șirul obținut din șirul inițial prin transformarea tuturor literelor mari în litere mici
  Exemplu:
  ```
  s = ' Programarea Algoritmilor'
  s.lower()
  print(s) #nu s-a modificat
  s = s.lower()
  print(s) #' programarea algoritmilor'
  ```
- **`s.upper()`** – returnează șirul obținut din șirul inițial prin transformarea tuturor literelor mici în litere mari
- **`s.swapcase()`**: furnizează șirul obținut din șirul inițial prin transformarea tuturor literelor mici în litere mari și invers;
- **`s.title()`**: furnizează șirul obținut din șirul inițial prin transformarea primei litere a fiecărui cuvânt în literă mare, restul literelor fiind transformate în litere mici;
  Exemplu:
  ```
  s = 'programarea algoritmilor'
  s = s. title()
  print(s) #' Programarea Algoritmilor'
  ```
- **`s.capitalize()`** - returnează șirul obținut din șirul inițial prin transformarea primei sale litere în literă mare, restul literelor fiind transformate în litere mici
  Exemplu:
  ```
  s = 'programarea algoritmilor'
  s = s. capitalize ()
  print(s) #' Programarea algoritmilor'
  ```
- **`s.strip([chars])`** - elimină caracterele din șirul **`chars`** primit ca parametru care se află la începutul și sfârșitul șirului s (și returnează șirul obținut); implicit **`chars=None`** și atunci elimina caracterele albe
  Exemplu:
  ```
  s = '    Programarea algoritmilor!!**     '
  s = s.strip()
  print(s)
  print(len(s))
  print(s.rstrip("*!"))
  ```
- **`s.rstrip() / s.lstrip()`** – acționează ca și **`strip`**, dar doar la sfârșitul șirului (la capătul din dreapta – **`rstrip`**), respectiv dar doar la începutul șirului (la capătul din stânga – **`lstrip`**),
- **`s.center(width, fillchar=')`**, **`s.ljust (width, fillchar=' )`** / **`rjust`** - pentru alinierea textului în centru, dreapta, respectiv stânga, pe o dimensiune dată. Spațiile care nu sunt ocupate de șir se completează cu **`fillchar:`**
  ```
  print('Programarea algoritmilor'.center(40))
  print('Programarea algoritmilor'.center(40,"*"))
  print('Programarea algoritmilor'.rjust(30,">"))
  ```

Testarea unor proprietăți legate de caractere

Pentru metodele de testare vom preciza doar cazul în care returnează **`True`** (altfel, evident, returnează False)

- **`s.islower()`** – returnează True dacă șirul este nevid și toate literele din șir sunt litere mici, False altfel
- **`s.isupper()`**
- **`s.isdecimal()`**, **`s.isdigit()`** , **`s.isnumeric()`**

  https://docs.python.org/3/library/stdtypes.html#str.isdecimal

- **`s.isalnum()`:** returnează True dacă toate caracterele șirului sunt alfanumerice (adică sunt litere sau cifre);

Exemplu :

```
s = "exemplu"
print(s.islower(),s.isalpha(),s.isupper())
s = "un sir. 2 siruri"
print(s.islower(),s.isalpha(),s.isupper())

#SUPLIMENTAR
s = '23'
print(s.isdigit(),s.isdecimal(),s.isnumeric())
s = "23\u00BD" #s = 123½
print(s)
print(s.isdigit(),s.isdecimal(),s.isnumeric())
s = '2' + '\u00B2'
print(s)
print(s.isdigit(),s.isdecimal(),s.isnumeric())
s = '2.3'
print(s.isdigit(),s.isdecimal(),s.isnumeric())
s = '-23'
print(s.isdigit(),s.isdecimal(),s.isnumeric())
```

Parsare, divizare și unificare cu separatori

- **`s.split(sep = None, maxsplit =-1)`**

  împarte s în cuvinte folosind **`sep`** ca separator (delimitator) și returnează o lista acestor cuvinte. Dacă parametrul opțional **`maxsplit`** este specificat, sunt făcute cel mult **`maxsplit`** împărțiri (se obține o listă de maxim **`maxsplit+1`** cuvinte). Dacă nu este specificat **`sep`**, atunci caracterele albe consecutive sunt considerate un separator. Între delimitatori consecutivi se consideră că există cuvinte vide. Nu se pot specifica o listă de delimitatori, ci doar unul (suplimentar: pentru a specifica mai mulți delimitatori se poate folosi funcția **`split`** din modulului **`re`**, folosind expresii regulat)

Exemple:

1. Împărțirea unei propoziții în cuvinte

```
s = "acesta   este un sir"
print(s.split()) #['acesta', 'este', 'un', 'sir']
print(s.split(" ")) #['acesta', '', '', 'este', 'un', 'sir']
print(s.split(maxsplit=1)) #['acesta', 'este un sir']
```

2. Se dau pe o linie numere separate cu spații. Să se afișeze suma lor.

```
s = input("numere pe o linie\n")
ls = s.split()
print(ls)
s = 0
for x in ls:
    s = s + int(x)
print(s)
```

- `s.join(iterable)`
  returnează șirul obținut prin concatenarea șirurilor din parametrul **`iterable`** (care este o colecție de șiruri, de exemplu o listă de cuvinte), folosind șirul s ca separator între șirurile concatenate.

  Exemplu:

```
ls = ["2","3","5"]
s = "*".join(ls)
print(s)
ls = ["ab"]*4
print(ls, id(ls[0]),id(ls[1]))
s = "".join(ls)
print(s)
```

Formatare

În Python există mai multe modalități de a formata un șir (care să conțină și variabile, aliniate după anumite reguli, scrise cu un format specificat, de exemplu cu un număr dat de zecimale.

Metoda format

Prima variantă este folosirea metodei format pentru un șir template în care pot fi inserate constante și variabile formatate după anumite specificații:

```
template.format(<positional_arguments>,<keyword_arguments>)
```

unde:

- **`template`** – este un șir care conține secvențe speciale cuprinse între **{}** care vor fi înlocuite cu parametri ai metodei format (numite **câmpuri de formatare**), de tipul

  `{[<camp>][!<fct_conversie>][:<format_spec>]}`

- parametri pot fi atât poziționali (care se identifică prin poziție și se apelează ca în C), dar și numiți (cu nume) = care se pot identifica și prin nume, putând fi apelați sub forma **`template.format(num_parametru=valoare)`**

- se returnează șirul template formatat, înlocuind câmpurile de formatare cu valorile parametrilor (formatate conform cu specificațiilor din câmpuri)

  Există mai multe variante de a specifica ce parametru pozițional se folosește într-un câmp de formatare:

  a. Nu specificăm nimic între {} => parametrii poziționali sunt considerați în ordine (numerotare automată)

  ```
  s = "Nota la {} = {}".format("PA",10)
  ```

  b. Specificăm numărul parametrului pozițional pe care îl folosim (numerotarea parametrilor începe de la 0)

  ```
  s = "Nota la {0} = {1}".format("PA",10)
  ```

  Nu se pot combina cele două abordări.

  Exemple

  ```
  x = 3; y = 4
  print("x={},y={}".format(x,y))
  print("x={0},y={1}".format(x,y))
  print("x={1},y={0},x+y={1}+{0}={2}".format(y,x,x+y))
  print("x={0},y={1}, s={2}".format(x,y))
  #eroare IndexError: Replacement index 2 out of range #for positional args tuple
  ```

  Dacă utilizăm parametri **cu nume**, pentru a indica ce parametru numit (cu nume) se folosește într-un câmp de formatare putem folosi direct numele parametrului

  ```
  x = 3
  y = 4
  print("x={p1},y={p2},s={suma}".format(suma=x+y,p1=x,p2=y))
  ```

  Se pot combina parametri poziționali cu cei numiți, dar cei numiți **se dau la final**

  ```
  x = 3; y = 4
  print("{p1}+{p2}={suma}, {p1}*{p2}={}".
            format(x*y,suma=x+y,p1=x,p2=y))
  ```

  Pentru a include acolada in șirul template fără a fi interpretat ca delimitator de câmp se dublează:

  ```
  x = 3; y = 4; z = 5
  s = "Multimea {{{},{},{}}}".format(x,y,z)
  print(s)
  ```

Se pot apela și câmpuri ale parametrilor:

```
z = 1 + 3j
print('z = {0.real}+ {0.imag}i'.format(z))
ls = [10,20,30]
print("primul si al doilea element din lista: {0[0]} si
{0[1]}".format(ls))
```

Specificarea formatului de afișare (de exemplu: lungimea ocupată la afișare, precizie, aliniere, baza în care se afișează etc) se face folosind sintaxa:

```
{[<camp>][!<fct_conversie>][:<format_spec>]}
```

unde

- **<fct_conversie>** poate fi:
  - **!s** – convertește cu str()
  - **!r** – convertește cu repr()
  - **!a**– convertește cu ascii()

  Exemplu

  ```
  s = "Programarea\talgoritmilor Ă"
  print('{!s}'.format(s))
  print('{!r}'.format(s))
  print('{!a}'.format(s))
  ```

- **<format_spec>** - poate include (printre altele):

  ```
  [0][<dimensiune>][.<precizie>][<tip>]
  ```

  cu **<tip>** având ca posibile valori (printre altele)

  - **d** : întreg zecimal
  - **b,o x,X** : întreg în baza 2,8 repectiv 16 cu litere mici/mari
  - **s** șir de caractere
  - **f** : număr real în virgulă mobilă (afișat implicit 6 cifre) ... etc

  Exemplu

  ```
  z = 1.1 + 2.2
  print('{}'.format(z))
  print('{:f}'.format(z))
  print('{:.2f}'.format(z))
  z = 12
  print('{}'.format(z))
  print('{:8}'.format(z))
  print('{:8b}'.format(z))
  print('{:08b}'.format(z))
  ```

Formatare - Interpolarea șirurilor: f-stringuri

Din versiunea 3.6 există un tip special de șiruri, numite **f-stringuri**, care pot include câmpuri de formatare în care se pot folosi direct variabile, chiar și expresii. Un astfel de șir este precedat de **`f`** sau **`F`**.

Exemple

```
nume = 'Ionescu'
prenume = 'Ion'
varsta = 20
#f-string
s = f"{nume} {prenume}: {varsta} ani"  #f-string
print(s) #Ionescu Ion: 20 ani

#acelasi sir se putea obține si folosind metoda format
#cu parametri pozitionali
s = "{} {}: {} ani".format(nume,prenume,varsta)
print(s)

#f-string cu numere si formatari
x = 3; y = 4
print(f"{x}")
print(f"{x}+{y}={x+y}, {x}*{y}={x*y}")
print(f'{x:08b}') #00000011
```

Un articol interesant despre formatare este:

https://realpython.com/python-formatted-output/

Formatare cu operatorul %

În versiunile mai vechi de Python exista o metodă de formatare similară cu cea din limbajul C (funcția printf), folosind operatorul %. Nu se mai recomandă folosirea ei, dar este util să o cunoaștem pentru a citi versiuni mai vechi de cod și a le traduce folosind noile metode de formatare:

```
<template> % (<values>)
```

Exemplu

```
s = "Nota la %s = %d" %  ("PA",10)
print(s)
x=3.1415
print("%d %.2f" % (x,x))
```

Memorarea șirurilor de caractere. String interning

Amintim că operatorul **`is`** testează dacă două obiecte sunt identice (sunt același obiect, cu aceeași zonă de memorie), în timp ce **`==`** testează dacă două obiecte sunt egale.

Amintim de asemenea că efectul evaluării unei expresii este, de obicei, crearea unui obiect nou cu valoarea egală cu a rezultatului evaluării. De asemenea, în urma unei atribuiri **`v = expresie`**, variabila **`v`** va referi obiectul rezultat în urma evaluării expresiei (expresia poate fi și o constantă).

Ce credeți că va afișa următoarea secvență de cod?

```
s1 = "sir"
s2 = "s" + "ir"
s3 = input()  # vom da ca input tot sirul sir
print(s1 == s2, s1 == s3)
print(s1 is s2, s1 is s3)
```

Dacă pentru fiecare valoare de tip **`str`** utilizată în program s-ar crea un obiect nou, atunci vor exista în memorie mai multe obiecte cu aceeași valoare, iar pentru astfel de obiecte expresia **`ob1 == ob2`** va fi **`True`**, iar **`ob1 is ob2`** va fi **`False`**. După cum am văzut în exemplul anterior, sunt însă și cazuri în care expresia **`ob1 is ob2`** este adevărată.

Pentru a evita crearea de obiecte egale care să ocupe memorie, în cazul șirurilor de caractere (dar și a altor tipuri imutabile), în Python este folosit un mecanism de string interning, numit în continuare SI, cu scopul de a memora o singură copie a unui șir având o anumită valoare (sau cât mai puține). Această optimizare este făcută pentru expresii **care se pot evalua la compilare** (în care operanzii sunt constante de tip str) și se bazează pe imutabilitatea șirurilor de caractere.

Modul de funcționare al mecanismului de SI poate să difere în funcție de mediul în care se execută programul (de nivelul acestuia de optimizare).

Pentru a înțelege cum funcționează mecanismul de SI să considerăm următoarele exemple. În primul considerăm două variabile ls1, ls2 inițializate cu aceeași valoare, lista [1, 2]:

```
ls1 = [1, 3]
ls2 = [1, 3]
print(id(ls1), id(ls2))
print(ls1 == ls2)
print(ls1 is ls2)
```

Când se inițializează **`ls1`** cu **`[1,3]`** se creează în memorie un obiect nou de tip **`list`**, cu elementele 1 și 3 și **`ls1`** referă acel obiect. Același lucru se întâmplă și la atribuirea **`ls2 = [1, 3]`**, deci **`ls1`** și **`ls2`** referă două obiecte diferite, care au însă aceeași valoare. Astfel, expresia **`ls1==ls2`** este **`True`** (obiectele au aceeași valoare), iar expresia **`ls1 is ls2`** este **`False`**.

Dacă modificăm însă programul anterior astfel încât variabilele să fie două șiruri inițializate cu aceeași valoare, rezultatul va fi diferit:

```
s1 = "sirul"
s2 = "sirul"
print(id(s1), id(s2))
print(s1 == s2)
print(s1 is s2)
```

Atât expresiile **`s1 == s2`**, cât și **`s1 is s2`** sunt **`True`**, deoarece prin mecanismul de SI se menține un bazin de șiruri (string pool, numit în continuare SP) de caractere cu șirurile deja create.
Când se ajunge la atribuirea **`s2 = "sirul"`** compilatorul verifică dacă există în SP un obiect cu valoarea "**`sirul`**" si, în caz afirmativ, returnează o referință către acesta (nu se mai creează un nou obiect), **`s2`** referind astfel același obiect ca și **`s1`**. Dacă șirul nu este în bazin, se creează un nou șir care se adaugă în bazin și variabila va referi acest obiect nou. Reamintim că mecanismul de SI se bazează pe faptul important că valoarea unui șir de caractere nu se poate modifica (elementele unei liste se pot modifica, un obiect de tip list este mutabil).

Nu orice valoare de tip **`str`** se memorează în SP. Memorarea implicită în SP se realizează **doar pentru expresii ale căror valori se pot determina la compilare** (deci nu vor fi memorate în SP șiruri a căror valoare nu se poate determina la compilare).

Să considerăm următoarea secvență de cod care extinde pe cea anterioară.

```
s1 = "sirul"   #=> un obiect ob cu valoarea "sirul" se adaugă în bazin
s2 = "sirul"  # există deja ob cu valoarea "sirul" în bazin, s2 va referi ob
s3 = "sir" + "ul" #suma de siruri constante => se cauta in bazin rezultatul;
                    #exista => s3 va referi spre ob
x = "sir"
s4 = x + "ul" #nu se poate evalua la compilare => nu se cauta in bazin,
              #se creeaza obiect nou
print(s1 is s2, s1 is s3, s1 is s4, s1==s4)
print(id(s1),id(s2),id(s3), id(s4)) #doar id(s4) este diferit
```

În general de memorează în SP doar valorile de tip **`str`** din program care se pot evalua la compilare, care în plus sunt șiruri de cel mult **4096** caractere (20 până la versiunea 3.7), care conțin doar caractere ASCII (condiția nu este obligatorie în toate implementările Python, în altele caracterele trebuie să poată fi utilizate în identificatori). Sunt memorate în SP și șiruri care fac parte din sintaxa limbajului (numele funcțiilor, claselor, variabilelor, argumentelor etc) și șirurile de lungime cel mult 1.

Exercițiu: Justificați rezultatele afișate de următoarea secvență de cod:

```
a = "programarea"
b = "programator"
print(a[0:3] == b[0:3] )
print(a[0:3] is b[0:3] )
print(a[-1] is b[5])
```

```
s = "a" * 4096
t = "a" * 4096
print(s == t)
print(s is t)
s = "a" * 4097
t = "a" * 4097
print(s == t)
print(s is t)
```

Utilizarea mecanismului SI prezintă și alte avantaje, pe lângă cele legate de spațiul de memorie utilizat. Pentru șirurile din SP testul de egalitate **`s == t`** se poate face în **O(1)**, comparând doar **`id(s)`** și **`id(t)`** (referințele obiectelor corespunzătoare). De asemenea, căutările în dicționare după chei din SP sunt mai rapide (vom explica în cursurile următoare modul de funcționare al acestora).

Un șir poate salvat și explicit în bazinul de șiruri folosind metoda intern(șir) din modulul **`sys`**:

```
import sys
s1 = "sirul"
x = "sir"
s2 = x + "ul"
print(s1 is s2)
s2 = sys.intern(x + "ul")
print(s1 is s2) #True
```

Metoda **`sys.intern(s)`** este însă lentă, deci trebuie utilizată doar dacă acest lucru este absolut necesar (de exemplu, dacă în program se vor efectua multe comparări de șiruri care nu sunt memorate de mecanismul de string interning).

Mai multe detalii referitoare la mecanismul string interning din limbajul Python găsiți în articolele:

- https://stackabuse.com/guide-to-string-interning-in-python/.
- https://medium.com/@bdov_/https-medium-com-bdov-python-objects-part-iii-string-interning-625d3c7319de
- http://guilload.com/python-string-interning/