

ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ Python

Урок 3

Рядки, списки

Зміст

1. Рядки	4
1.1. Кодування ASCII, Unicode, UTF-8, Byte-code	4
1.2. Рядок — незмінна послідовність символів	13
1.3. Методи рядків	15
1.4. Особливості роботи з рядками	20
1.5. Зріз рядка	29
1.6. Екрановані послідовності	34
1.7. «Сирі» рядки	38
1.8. Форматований вивід	42
1.9. Модуль string	49
1.10. Регулярні вирази, модуль re	54
2. Списки	70
2.1. Поняття класичного масиву	70
2.2. Поняття колекції об'єктів	71
2.3. Посилальний тип даних list	73
2.4. Створення списків	74

2.5. Генератори списків	76
2.6. Робота зі списками	79
2.7. Методи списків	83
2.8. Оператор приналежності in.....	89
2.9. Особливості списків, посилання та клонування.....	90
2.10. Пошук елемента	93
2.11. Матриці.....	93

1. Рядки

1.1. Кодування ASCII, Unicode, UTF-8, Byte-code

Багато інформації, що обробляється програмами, перебуває у текстовому вигляді. Текст сприймається людиною легко, але поняття рядка та символу для неї є досить абстрактними. Для програміста важливо конкретно і чітко розуміти, як такі дані зберігаються і обробляються комп'ютером.

Процесор комп'ютера працює лише з числами, причому представленими у двійковому форматі (з бітами). Згадаймо, що біт — це мінімальна одиниця інформації у пам'яті комп'ютера, яка може приймати лише одне із двох значень: нуль або одиницю. Відповідно, всі дані (числові, текстові, відео, аудіо-інформація, зображення), що зберігаються та обробляються комп'ютером — це послідовність нулів та одиниць.

Процесор працює з двійковим представленням даних, але може перетворювати їх на інші системи числення, наприклад, на шістнадцяткову (яка легше і зручніше сприймається людиною).

У шістнадцятковій системі числення кожен байт представляється двома цифрами (наприклад, число 0 — 0x00 у шістнадцятковому вигляді, префікс «0x» — ознака шістнадцяткової форми).

Текстові дані складаються із символів, які є елементарними об'єктами. Символи у мові є скінченною множиною (алфавіт мови). Кількість символів у алфавіті мови — потужність алфавіту.

Для того, щоб дізнатися, скільки інформації можна надати деяким алфавітом, використовується формула:

$$N = 2^b,$$

де N — потужність алфавіту;

b — кількість біт для представлення символу.

Якщо для одного символу виділяється один байт (8 біт), то максимальна кількість символів алфавіту — це $2^8 = 256$, яка є достатньою для представлення всіх необхідних символів мови. Таким чином, кожен символ тексту є восьмирозрядним двійковим кодом.

Процес кодування полягає в тому, що кожному символу алфавіту ставиться у відповідність унікальний код (від 0 до 255 в десятковій системі або його двійковому представленню від 00000000 до 11111111). Комп'ютер розрізняє (розпізнає) символи з їхнього коду.

Для того, щоб розуміти, якому символу відповідає код, використовують так звані таблиці кодування, у яких кожному символу алфавіту (алфавітно-цифровому символу) поставлено відповідний набір числових значень (код).

Першою таблицею кодувань стала ASCII (англ. *American Standard Code for Information Interchange* — Американський стандартний код для інформаційного обміну). Перша версія ASCII використовувала лише 7 біт для представлення символу, відповідно потужність алфавіту була 127 символів мови. Перші 32 коди в таблиці називаються керуючими (використовуються для форматowanego виведення тексту та інших службових цілей). Коди символів з літер і цифр знаходяться в діапазоні з 32 до 126 (код 127 не зайнятий жодним символом).

Крім символів латинського алфавіту, таблиця містила цифри, арифметичні та розділові знаки, символи пробілу, дужки тощо. Наприклад, код латинського символу «А» — 65, а «В» — 66. Оскільки символ «В» йде в алфавіті після символу «А», то його числовий код більший.

Коди малих символів відрізняються від кодів великих, наприклад, код «а» — це 97. Якщо символ є цифрою, це означає, що його код збігається з його значенням. Наприклад, код цифри «0» — це 48.

Однак, крім англійської мови у світі є й інші, алфавіти яких також потрібно представляти у вигляді таблиць кодувань. Тоді 127 символів явно недостатньо, щоб можна було працювати з іншими мовами. Для вирішення цієї проблеми кодування ASCII розвивалося і паралельно з'являлися інші стандарти. Кожен символ став кодуватися восьмибітним кодом і, відповідно, потужність алфавіту вже становила 256 символів.

Перша половина таблиці кодування ASCII залишалася незмінною (див. рис. 1).

Ця частина ASCII таблиці — це загальноприйнятий світовий стандарт (текст, який використовує лише ці символи, відображатиметься на будь-якому комп'ютері, тобто текст, що складається тільки з латинських літер і цифр буде правильно прочитаний).

Друга (верхня) частина таблиці (128 кодів, з 128 по 255) була задіяна для символів інших мов та псевдографіки. Однак, мов багато, і для кодування всіх символів усіх мов 128 кодів не вистачить. У зв'язку з цим, при використанні однобайтового кодування для кожної мови існує своя верхня частина таблиці ASCII.

Нижня часть кодировочной ASCII – таблицы.							
0	16 ►	32 пробел	48 0	64 @	80 P	96 `	112 p
1 ☺	17 ◀	33 !	49 1	65 A	81 Q	97 a	113 q
2 ☹	18 ↑	34 «	50 2	66 B	82 R	98 b	114 r
3 ♥	19 !!	35 #	51 3	67 C	83 S	99 c	115 s
4 ♦	20 ¶	36 \$	52 4	68 D	84 T	100 d	116 t
5 ♣	21 §	37 %	53 5	69 E	85 U	101 e	117 u
6 ♠	22 —	38 &	54 6	70 F	86 V	102 f	118 v
7 •	23 ↓	39 '	55 7	71 G	87 W	103 g	119 w
8 ■	24 ↑	40 (56 8	72 H	88 X	104 h	120 x
9 ○	25 ↓	41)	57 9	73 I	89 Y	105 i	121 y
10 ■	26 →	42 *	58 :	74 J	90 Z	106 j	122 z
11 ♂	27 ←	43 +	59 ;	75 K	91 [107 k	123 {
12 ♀	28 L	44 ,	60 <	76 L	92 \	108 l	124
13 ♪	29 ↔	45 -	61 =	77 M	93]	109 m	125 }
14 🎵	30 ▲	46 .	62 >	78 N	94 ^	110 n	126 ~
15 ☀	31 ▼	47 /	63 ?	79 O	95 _	111 o	127

Рисунок 1

Проблеми були не в тому, що для кожної мови доводилося розробляти свій стандарт, а в тому, що цей стандарт виявлявся несумісним з іншими (наприклад, у випадках, коли в межах одного документа використовується кілька мов, то встановлення одного кодування на весь документ може призвести до невірної інтерпретації одного чи кількох таких фрагментів). Також часто для однієї мови могло бути розроблено кілька

різних стандартів. Наприклад, для російської мови були створені стандарти CP866 (Code 866), KOI8-R, KOI8-U, ISO-8859-5. І в цьому хаотичному наборі стандартів було досить важко розібратися.

Отже, виникла необхідність переходу до стандартів, у яких для кодування символу буде використовуватися більше одного байта. У 1991 р. був розроблений стандарт Юнікод (англ. *Unicode*), із застосуванням якого можна було в одному документі використовувати різні мови, знаки математичних формул тощо.

Кожен символ у стандарті Unicode кодується за допомогою 31 біта (4 байти з вирахуванням одного біта). Таким чином, отримуємо $2^{31} = 2147483684$ (тобто більше двох мільярдів), що цілком достатньо для створення унікального коду для кожного символу кожної мови.

Unicode є стандартом, а не кодуванням, тобто це таблиця символів, що складається з 1114112 позицій, більшість із яких не заповнені. Цей простір поділено на 17 блоків, по 65 536 символів у кожному. Кожен блок містить певну групу символів, наприклад, в нульовому (базовому) блоці містяться найуживаніші символи усіх сучасних алфавітів. У Unicode перші 128 кодів збігаються з таблицею ASCII.

Записуються символи у шістнадцятковому вигляді з префіксом «U+». Наприклад, базовий блок включає символи від U+0000 до U+FFFF (від 0 до 65535).

Відображення кодів на рисунку 2.

U+ вказує на те, що це стандарт Unicode, а номер — число, в яке переведено двійковий код для даного символу. У Юнікодi використовується шістнадцаткова система.


```
"Hello World"
```

```
U+0048 : LATIN CAPITAL LETTER H
U+0065 : LATIN SMALL LETTER E
U+006C : LATIN SMALL LETTER L
U+006C : LATIN SMALL LETTER L
U+006F : LATIN SMALL LETTER O
U+0020 : SPACE [SP]
U+0057 : LATIN CAPITAL LETTER W
U+006F : LATIN SMALL LETTER O
U+0072 : LATIN SMALL LETTER R
U+006C : LATIN SMALL LETTER L
U+0064 : LATIN SMALL LETTER D
```

Рисунок 2

Одним із основних принципів у філософії Unicode є чітке розмежування між символами, їх представлення у пам'яті комп'ютера та їх зовнішнім відображенням на пристрої виведення.

Наприклад, юнікод-символ **U+041F** — це велика кирилична літера **П**. Існує кілька можливостей представлення даного символу в пам'яті комп'ютера (залежно від того, скільки біт виділяється під його код) так само, як і величезна кількість способів відображення його на екрані монітора. Але при цьому ми завжди розуміємо символ «**П**», як, власне, символ «**П**» (рис. 3).

Якщо до Unicode кожному символу відповідав код довжиною один байт, то Unicode-символ може бути закодований різною кількістю байтів. Для вирішення цих питань було створено Unicode-кодування (формати перетворення Unicode), наприклад, UTF-8.

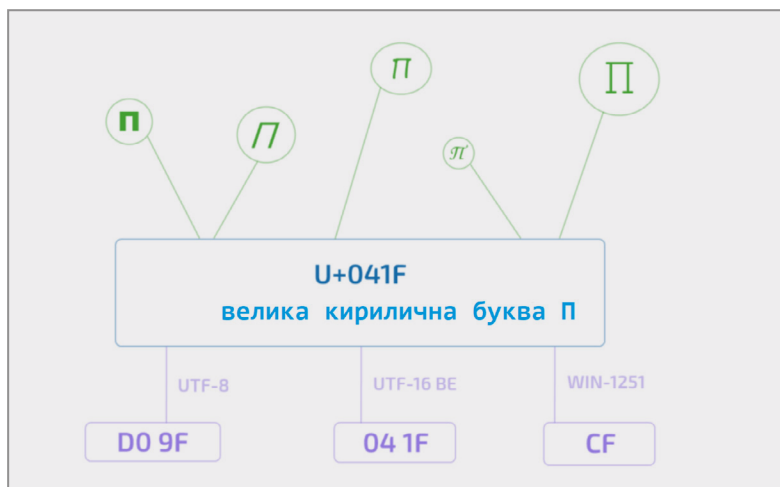


Рисунок 3

UTF (*Unicode Transform Format*) — це спосіб представлення Unicode. Кодування UTF визначено стандартом Unicode і дозволяють закодувати будь-який потрібний кодовий пункт Unicode.

Існує кілька різних видів UTF-стандартів, що відрізняються кількістю байтів, використовуваних для кодування одного кодового пункту. У UTF-8 використовується один байт на кодовий пункт, у UTF-16 — 2 байти, у UTF-32 — 4 байти. Як було зазначено, UTF-8 використовує для коду символу 1 байт (як і ASCII), тобто якщо програміст при розробці програми використовував кодування ASCII, а користувачі — UTF-8, то відображення символів не буде відрізнятись.

Однак є символи Юнікод, для кодування яких потрібно більше байт. Зрозуміло, що це залежить від мови. Символ англійського алфавіту подається одним байтом, тоді як іврит, арабські символи — вже двома байтами. Для

кодування символів багатьох мов (наприклад, китайської чи японської) необхідно 3 байти.

Отже, особливістю UTF-8 є те, що є можливість кодувати символи у такий спосіб, що найпоширеніші символи займають 1-2 байти, а для кодування рідко використовуваних символів — 3-4 байти. Якщо потрібно вказати, що символ займає більше одного байта, використовується спеціальна комбінація біта (знак продовження), і це вказує на те, що код даного символу продовжується в наступних байтах.

Рядок у Python — це послідовність знаків, тобто послідовність кодів, наприклад, з використанням юнікод-символів. Як було вже розглянуто вище, будь-яка послідовність символів зберігається та обробляється комп'ютером як послідовність байтів (байт-рядок, Byte-code).

Ця послідовність утворюється в результаті застосування кодування, спосіб якого залежить від кодування.

Для перетворення байт-рядка на звичайний (сприйманий людиною) рядок необхідно виконувати декодування за допомогою спеціальних методів (`.decode()`). Але щоб процес декодування був виконаний правильно, нам необхідно знати кодування (яке було використано для отримання цієї послідовності байтів), оскільки інше кодування може відображати одні й ті самі байти у вигляді інших символів рядка.

Кодування можна умовно представляти як ключ шифрування, який допомагає:

- «зашифрувати» (виконати кодування) рядок у байти: зазвичай для цього використовується метод `encode`;
- «розшифрувати» байти (декодувати) у рядок: наприклад, за допомогою методу `decode`.

Тобто для виконання зазначених перетворень має використовуватися те саме кодування.

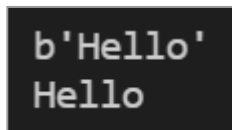
Наприклад:

```
userMessage = 'Hello'
userMessageEnc=userMessage.encode('utf-8')
print(userMessageEnc)

userMessageDec=userMessageEnc.decode('utf-8')
print(userMessageDec)
```

Ми створили рядок `userMessage`, який спочатку перевели в послідовність байт (`userMessageEnc`), використовуючи метод `encode` та кодування `utf-8`. А потім декодували назад у рядок `userMessageDec`, використовуючи метод `decode` з тим самим кодуванням (`utf-8`).

Оскільки наш рядок складається з символів латиниці (входить до нижньої частини ASCII таблиці), то в результаті першої команди `print(userMessageEnc)` ми бачимо ASCII символи, а не коди цих символів в Unicode (такі особливості роботи функції `print()`).



```
b'Hello'
Hello
```

Рисунок 4

Префікс «`b`» перед першим рядком говорить про те, що фактично це байт-рядок.

У випадку, якщо наш рядок буде містити символи, відмінні від латиниці (наприклад, деякі символи німецького алфавіту), то ми побачимо коди символів в Unicode:

```

userMessage = 'Können Sie mir bitte helfen?'
userMessageEnc=userMessage.encode('utf-8')
print(userMessageEnc)

userMessageDec=userMessageEnc.decode('utf-8')
print(userMessageDec)

```

```

b'K\x3c3\xb6nnen Sie mir bitte helfen?'
Können Sie mir bitte helfen?

```

Рисунок 5

1.2. Рядок — незмінна послідовність символів

В багатьох мовах програмування є такі структури даних, які дозволяють зберігати набір об'єктів, доступних за індексом (номером входження). Наприклад, статичний масив C++ має фіксований розмір і містить групу об'єктів одного типу.

Послідовності в Python дозволяють зберігати набір об'єктів як одного типу даних, так і різних. Причому деякі з цих наборів (наприклад, списки) можуть збільшуватися або зменшуватися в розмірах.

Одним із найпоширеніших видів послідовностей є рядок — послідовність символів.

Приклади рядків:

```

'Python'
'n'
'''
"%s is number %d"
"""hey there"""

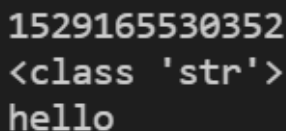
```

Як вам вже відомо, в Python існують змінні та незмінні типи даних. Рядок (`str`) є незмінним (`immutable`) типом даних.

Як було розглянуто раніше, під час створення змінної спочатку буде створено об'єкт, до складу якого входить унікальний ідентифікатор, тип і значення. Тільки після цього змінна може посилатися на вже створений об'єкт.

Незмінність типу даних означає, що створений об'єкт більше не змінюється. Наприклад, якщо ми оголосимо змінну `myStr = "hello"`, то буде створено об'єкт зі значенням `"hello"`, типу `str` та ідентифікатором, який можна впізнати за допомогою функції `id()`.

```
myStr="hello"  
print(id(myStr))  
print(type(myStr))  
print(myStr)
```



```
1529165530352  
<class 'str'>  
hello
```

Рисунок 6

На перший погляд, це може здатися обмеженням або недоліком, але насправді це не так. На прикладах роботи з рядками ми побачимо, чому це не є обмеженням. У цьому прикладі ми використовували літерали рядків.

Рядковий літерал — це деяка послідовність символів, укладена в одинарні або подвійні лапки.

```
myStr1="hello"  
myStr2='hello'
```

Усі пробіли та знаки табуляції збережуться у рядку так, як є.

```
myStr3="What's your name?"
```

Причина наявності двох варіантів рядкових літералів полягає в можливості додавати всередину рядка символи лапок або символ апострофа (як у прикладі вище) без використання екранування (яке ми розглянемо далі).

Також можна вказувати «багаторядкові» літерали, використовуючи потрійні лапки («» або '''). У межах потрійних лапок можна використовувати подвійні або потрійні лапки.

Наприклад:

```
myStr4='''This is a multi-line string (text).  
         This is the first line of text.  
         This is the second line of text.  
         This is the third line of text with 'quotes'  
         '''
```

Таким чином, однією з переваг рядкових літералів є можливість запису багаторядкових текстів, у тому числі з лапками, апострофами всередині.

1.3. Методи рядків

Для роботи з рядками в Python передбачено велику кількість вбудованих функцій і методів рядків.

Перш ніж переходити до вивчення цих різноманітних методів, необхідно розглянути особливості роботи з рядками, а саме індексацію рядків. Оскільки рядок є впорядкованим набором символів (є послідовністю), кожен елемент рядка (символ рядка) має свій унікальний індекс (порядковий номер). Нумерація індексів починається з нуля. Індекс другого елемента рядка — 1 і так далі. Індекс останнього символу визначається як довжина рядка мінус один.

Розподіл індексів елементів рядка «**foobar**» відбувається так:

f	o	o	b	a	r
0	1	2	3	4	5

Рисунок 7

Для доступу до окремих символів рядка потрібно вказати ім'я рядка, за яким йде індекс потрібного символу в квадратних дужках [].

Ми можемо звернутися до потрібного символу рядка наступним чином:

```
myStr="foobar"

print(myStr[0]) #'f'
print(myStr[1]) #'o'
print(len(myStr)-1) #'r'
```

В Python можливий також і доступ по від'ємному індексу, причому відлік йде від кінця рядка, тобто останній

елемент має індекс, який дорівнює **-1**, передостанній — індекс, який дорівнює **-2** і т. д.

```
myStr="foobar"  
print(myStr[-1]) #r  
print(myStr[-2]) #a
```

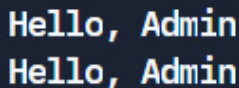
Тепер перейдемо до вивчення базових операцій під час роботи з рядками.

1.3.1. Конкатенація (об'єднання) рядків

Якщо необхідно об'єднати два (або більше) рядків в один (новий, тому що пам'ятаємо, що рядки незмінні), тоді це можна виконати за допомогою символу «+», розташованого між об'єднуваними об'єктами рядків.

```
str1 = "Hello,"  
str2 = "Admin"  
  
print(str1 + str2)  
str3 = str1 + str2  
print(str3)
```

Результат:



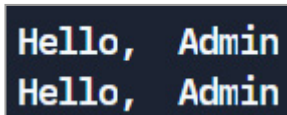
```
Hello, Admin  
Hello, Admin
```

Рисунок 8

Як було зазначено, в результаті конкатенації рядків **str1** і **str2** буде створено новий об'єкт (який можна вивести в консоль або зберегти в окрему змінну — **str3**).

Розглянемо приклад конкатенації трьох рядків:

```
str1 = "Hello,"  
str2 = "Admin"  
  
print(str1 + str2)  
str3 = str1 + str2  
print(str3)
```



```
Hello, Admin  
Hello, Admin
```

Рисунок 9

Конкатенацію також називають оператором «додавання» рядків. Аналогічну можливість надає метод рядків `.join()`, який буде розглянуто далі.

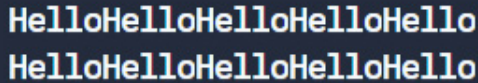
1.3.2. Дублювання рядка (оператор «множення» рядків)

Якщо нам необхідно повторити рядок на задану кількість разів (продублювати його), то для цього скористаємося символом «*» (оператор множення), вказавши рядок як один операнд, а другий — це кількість повторень.

Як і у разі конкатенації рядків, результатом буде новий об'єкт.

Наприклад:

```
myStr="Hello"  
print(myStr*5)  
  
myBigStr=myStr*5  
print(myBigStr)
```



HelloHelloHelloHelloHello
HelloHelloHelloHelloHello

Рисунок 10

1.3.3. Визначення довжини рядка

Для визначення довжини рядка (кількості символів, що входять до його складу) передбачено вбудовану функцію `len()`, аргументом якої є сам рядок.

```
myStr="Hello"
print(len(myStr)) # 5
```

Ця функція є універсальною (тобто використовується з усіма) типами, що ітеруються: рядками, списками, кортежами та словниками, повертаючи кількість елементів такої послідовності. Тепер розглянемо методи об'єкта `str` (рядки), `str` (скорочення від «*string*») — це назва типу даних (об'єкту) в Python, що представляє собою послідовність символів (рядок).

Спочатку згадаємо загальний синтаксис для виклику методу деякого об'єкту:

```
objName.methodName(args)
```

Цей код викликає метод `.methodName()` об'єкта `objName`., а `args` — це список аргументів (вказуються через кому), які передаються методу (якщо такі аргументи є).

Рядки в Python є об'єктами (`str`), отже, цей синтаксис буде таким:

```
str.methodName(args)
```

Вбудовані методи об'єкта `str`, які є в Python для роботи з рядками, зручно розділити на групи в залежності від типу завдання, які вони реалізують.

1.4. Особливості роботи з рядками

1.4.1. Методи для зміни регістру рядка

- `str.capitalize()` переводить перший символ рядка `str` у верхній регістр, решта — у нижній, результат, що повертається — перетворена копія оригінального рядка `str` (при цьому оригінальний рядок у результаті роботи методу не зміниться, як і у разі використання розглянутих нижче методів перетворення регістру).
- `str.lower()` переводить усі символи літер оригінального рядка `str` в нижній регістр, результат, що повертається — перетворена копія рядка `str`.
- `str.upper()` перетворює усі символи літер рядка `str` у верхній регістр, результат, що повертається — перетворена копія рядка `str`.
- `str.title()` перетворює перші літери кожного слова рядка `str` у верхній регістр (а решту літер слів переводить у нижній регістр), результат, що повертається — перетворена копія рядка `str`.
- `str.swapcase()` перетворює символи літер рядка `str`, змінюючи їх регістр на протилежний, результат, що повертається — перетворена копія рядка `str`.

Розглянемо перелічені методи на прикладі:

```
myStr="python was created in the late 1980's  
by Guido van Rossum."
```

```
print(myStr.capitalize()) # Python was created in
    the late 1980's by guido van rossum.
print(myStr.lower()) #python was created in the late
    1980's by guido van rossum.
print(myStr.upper()) #PYTHON WAS CREATED IN THE LATE
    1980'S BY GUIDO VAN ROSSUM.

print(myStr.title()) #Python Was Created In The Late
    1980'S By Guido Van Rossum.

print(myStr.swapcase()) #PYTHON WAS CREATED IN
    THE LATE 1980'S BY gUIDO VAN rOSSUM.
```

1.4.2. 2. Методи пошуку підрядка у рядку

Ці методи надають різні способи пошуку в цільовому рядку зазначеного підрядка (фрагменту).

Кожен метод у цій групі, крім обов'язкового аргументу підрядка, (фрагмента оригінального рядка, який шукається) включає два необов'язкові аргументи (**startIndex** та **endIndex**) для вказівки діапазону пошуку.

При використанні цих параметрів пошук відбувається не у всьому рядку, а в його частині, починаючи з індексу **startIndex** до індексу **endIndex-1** включно. Якщо аргумент **startIndex** вказано, а **endIndex** — ні, то пошук у рядку відбувається від індексу **startIndex** до кінця рядка.

- **str.count(pattern [, startIndex [, endIndex]])** — визначає кількість входжень фрагмента **pattern** у рядок **str** (або її частину при заданні діапазону пошуку (з індексу **startIndex** ... по індекс **endIndex**)).

```
myStr = "Python was created in the late 1980's
    by Guido van Rossum. Python- cool!"
```

```
print(myStr.count('Python')) #2
print(myStr.count('Python',20,65)) #1
print(myStr.count('Python',10)) #1
```

- `str.find(pattern [, startIndex [, endIndex]])` — використовується для пошуку в рядку `str` потрібного фрагмента `pattern`, результат, що повертається — індекс початку першого входження фрагмента `pattern` у рядок `str` або `-1` у випадку, якщо фрагмент `pattern` не входить до складу `str`.

Також можна обмежити діапазон пошуку за допомогою параметрів `startIndex` та `endIndex`.

```
myStr="Python was created in the late 1980's by Guido
van Rossum. Python- cool!"

print(myStr.find('a')) #8
print(myStr.find('a',2,5)) #-1
```

- `str.index(pattern [, startIndex [, endIndex]])` — робота методу аналогічна методу `.find()`, відмінність — у виклику виключення `ValueError` у разі, коли фрагмент `pattern` не входить до складу `str`).

```
print(myStr.index('a')) #8
print(myStr.index('a',2,5)) #ValueError - substring
                             not found
```

- `str.rfind(pattern [, startIndex [, endIndex]])` — використовується для пошуку в рядку `str` потрібного фрагмента `pattern`, починаючи з кінця рядка `str`, результат, що

повертається — індекс початку останнього входження фрагмента `pattern` у рядок `str` або `-1` у випадку, якщо фрагмент `pattern` не входить до складу `str`. Для обмеження діапазону пошуку можна використовувати параметри `startIndex` та `endIndex`.

- `str.rindex(pattern [, startIndex [, endIndex]])` — робота методу аналогічна методу `.rfind()`, відмінність — у виклику виключення `ValueError` у разі, коли фрагмент `pattern` не знайдено (не входить до складу `str`).

```
print(myStr.rfind('a')) #48
print(myStr.rfind('a',2,20)) #14
print(myStr.rindex('a')) #48
print(myStr.rindex('a',2,6)) #ValueError -
                             substring not found
```

1.4.3. Методи перевірки початку (закінчення) рядків

Методи цієї групи повертають `True` у разі, коли оригінальний рядок задовольняє умову, `False` — навпаки. Для обмеження діапазону перевірки можна використовувати параметри `startIndex` і `endIndex`.

- `str.endswith(pattern [, startIndex [, endIndex]])` — визначає, чи рядок `str` закінчується вказаним фрагментом `pattern`.
- `str.startswith(pattern [, startIndex [, endIndex]])` — визначає, чи рядок `str` починається вказаним фрагментом `pattern`.

```
myStr="Python was created in the late 1980's by Guido
      van Rossum. Python- cool!"
```

```

print(myStr.startswith('P'))    #True
print(myStr.startswith('p'))    #False

print(myStr.startswith('w',7))  #True

print(myStr.endswith('!'))      #True
print(myStr.endswith('n',2,6))  #True

```

1.4.4. Методи перевірки рядків

Іноді потрібно перевірити, чи складається рядок із певних символів, наприклад цифр. Для вирішення такого роду завдань передбачені наступні методи (усі вони, як результат, повертають значення **True** — якщо рядок містить потрібні символи, **False** — навпаки).

- **str.isalnum()** — перевіряє, чи рядок **str** складається лише з літер та цифрових символів.
- **str.isalpha()** — перевіряє, чи рядок **str** складається лише з літерних символів.
- **str.isdigit()** — перевіряє, чи рядок **str** складається лише з цифрових символів (використовується для перевірки, чи є рядок **str** числом).
- **str.islower()** перевіряє, чи всі літерні символи рядка **str** в нижньому регістрі (символи рядка **str**, які не є літерою алфавіту — ігноруються даною перевіркою).
- **str.isspace()** перевіряє, чи до складу рядка **str** входять лише пробілові символи, до яких відносяться символи пробілу ' ', табуляції '\t' та переходу на новий рядок '\n'.
- **str.istitle()** перевіряє, чи починається кожне слово рядка **str** із символу у верхньому регістрі.

- `str.isupper()` визначає, чи усі літери рядка `str` належать до верхнього регістру.
- `str.islower()` визначає, чи усі літери рядка `str` належать до нижнього регістру.

Розглянемо наведені методи на прикладах:

```
myStr="Python2021"
print(myStr.isalnum()) #True
print(myStr.isalpha()) #False

myAge="16"
print(myAge.isdigit()) #True

myStr1="it was created in the late 1980's"
print(myStr1.islower()) #True

myStr2=" \t \n \t\t"
print(myStr2.isspace()) #True

myName1= "Guido van Rossum"
print(myName1.istitle()) #False

myName2= "GUIDO VAN ROSSUM"
print(myName2.isupper()) #True
```

1.4.5. Методи форматування рядків

Методи цієї групи змінюють вивід (форматують) рядки.

- `str.center(width [, fillchar])` доповнює (розширює) рядок `str` до вказаної довжини `width`, результат, що повертається — розширена копія рядка `str`. Якщо параметр `fillchar` вказано, він буде використаний, як символ заповнення, інакше — відступи заповнюються пробілами.

У випадку, якщо параметр `width` менше або дорівнює довжині рядка `str`, то рядок не змінюється.

```
myStr="Python2021"
print(myStr.center(30))
print(myStr.center(30, '*'))
print(myStr.center(5))
```

Результат:

```
          Python2021
*****Python2021*****
Python2021
```

Рисунок 11

- `str.expandtabs(tabsize=8)` повертає копію рядка `str`, в якому кожен символ табуляції (`'\t'`) замінено на пробіл, кількість яких задається через параметр `tabsize`.

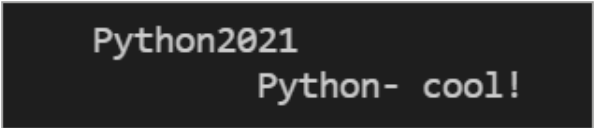
```
myStr="Python2021\n\t\tPython- cool!"
print(myStr.expandtabs(tabsize=4))
```

```
Python2021
      Python- cool!
```

Рисунок 12

Якщо параметр `tabsize` не заданий, кожен символ табуляції замінюється на 8 пробілів.

```
myStr="Python2021\n\t\tPython- cool!"
print(myStr.expandtabs())
```



```
Python2021
Python- cool!
```

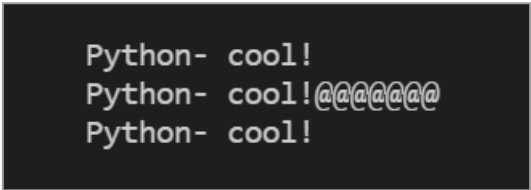
Рисунок 13

- `str.ljust(width [, fillchar])` повертає вирівняну по лівому краю копію рядка `str` зазначеної ширини `width`.

Якщо параметр `fillchar` заданий, то він використовується для заповнення недостатньої кількості символів, в іншому випадку використовується символ пробілу.

У випадку, коли параметр `width` менше або дорівнює довжині рядка `str`, рядок не змінюється.

```
myStr="Python- cool!"
print(myStr.ljust(20))
print(myStr.ljust(20, '@'))
print(myStr.ljust(5))
```



```
Python- cool!
Python- cool!@@@@@@@@
Python- cool!
```

Рисунок 14

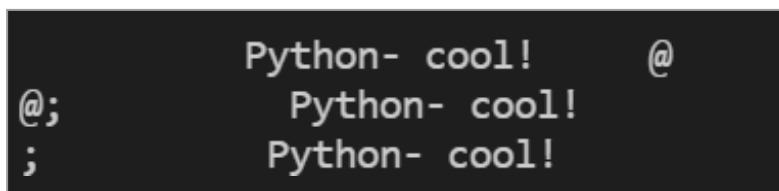
- Для аналогічного вирівнювання рядка з правого краю в полі вказаної ширини використовується метод `str.rjust(width [, fillchar])`.
- `str.lstrip([characters])` повертає копію рядка `str`, видаляючи початкові символи (ліворуч), вказані як аргумент

characters. Якщо параметр **characters** не вказаний, то видаляються символи пробілів.

- Для аналогічного видалення кінцевих символів (або пробілів) у рядку праворуч передбачено метод **str.rstrip([characters])**.
- Якщо необхідно видалити пробіли (або вказані символи) з лівого і з правого боку рядка, використовується метод **str.strip([characters])**.

```
myStr="          Python- cool!          "
print(myStr.lstrip())
print(myStr.rstrip())
print(myStr.strip())

myStr="@;          Python- cool!          @"
print(myStr.lstrip('@;'))
print(myStr.rstrip('@'))
print(myStr.strip('@'))
```



```
Python- cool!  }
@; Python- cool!  }
; Python- cool!  }
```

Рисунок 15

Також для роботи з рядками передбачено цікавий метод **str.zfill(width)**, який доповнює рядок ліворуч символами «0» ширини **width**.

Якщо рядок **str** містить будь-який символ перед цифрами, він залишається в лівій частині рядка і заповнення нулями відбувається після нього.

```
myStr="123"
print(myStr.zfill(5))

myStr="+123"
print(myStr.zfill(5))
```

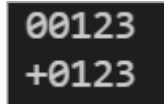


Рисунок 16

1.5. Зріз рядка

Часто в різних завданнях з обробки текстової інформації нам необхідно добувати певну частину рядка. Наприклад, якщо рядок містить ім'я та прізвище студента «*John Smith*», а нам для подальшої роботи потрібне лише прізвище студента «*Smith*», тобто фрагмент рядка.

У Python для вилучення фрагмента рядка можна скористатися одним із наступних способів:

- індексуванням — для отримання одного символу рядка;
- зрізом рядка — для отримання як окремого символу, так і фрагмента рядка;
- функціями стандартної бібліотеки Python.

Особливості індексування рядків ми вже розглянули раніше. Зріз рядка — спосіб вилучення фрагмента рядка (підрядка) за одну дію. Варіанти формування зрізів можна поділити на дві категорії:

- базові зрізи із зазначенням меж зрізу (обох або одного з);
- розширені зрізи із зазначенням кроку індексування.

Розглянемо синтаксис базових зрізів:

- `str[a:b]` — обидві межі (`a` — ліва та `b` — права) зрізу задані явно.

У цьому випадку з рядка `str` витягується підрядок, починаючи з позиції `a` до позиції `b` (але не включаючи `b`) (рис. 17).

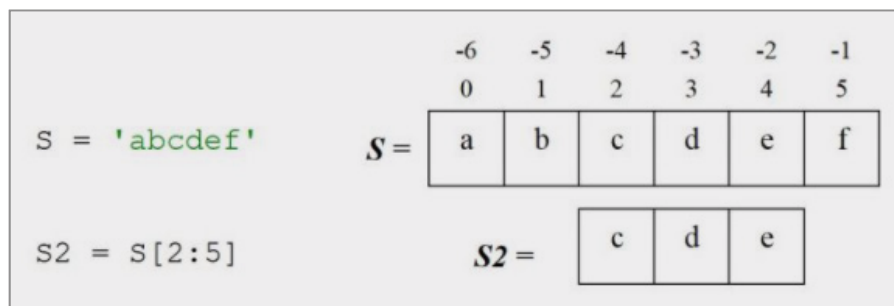


Рисунок 17

На рисунку наведено склад рядка із зазначенням індексів. Перший зріз (з використанням додатних індексів) `S2[2:5]` витягує з рядка `S` фрагмент, починаючи з символу рядка `S` з індексом 2 («`c`») і до символу з індексом 5, але не включаючи його. Останній символ фрагмента буде «`e`», тобто за індексом 4 (5-1).

Можливе також використання від'ємних індексів. Згадаймо, що з індексації рядків ми можемо також використовувати від'ємні індекси, які зменшуються зправа наліво від `-1` до `-довжини рядка`: останній елемент рядка має індекс `-1`, передостанній — на один менше (`-2`) тощо.

Також можливе поєднання від'ємних та додатних індексів в одному зрізі.

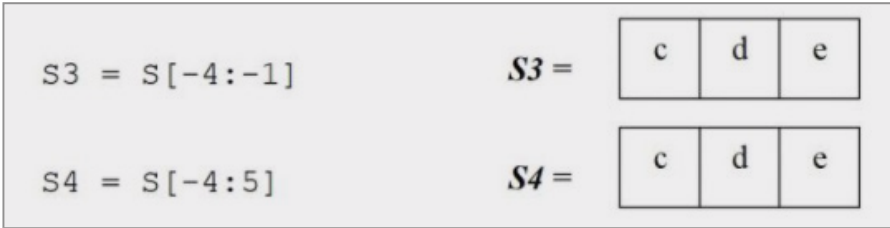


Рисунок 18

Зріз **S3** витягує із рядка **S** фрагмент, починаючи із символу рядка **S** з індексом **-4** («**c**») і до символу з індексом **-1**, але не включаючи його. Останній символ фрагмента буде «**e**», тобто за індексом **-2** (**-1-1**).

- `str[:b]` — вказана лише межа зрізу праворуч.

У цьому випадку з рядка **str** витягується підрядок, починаючи з початку рядка і до позиції **b** (але не включаючи **b**).

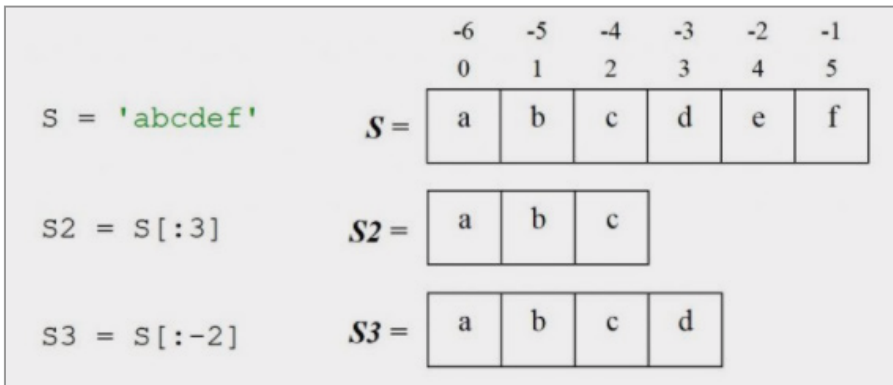


Рисунок 19

- `str[a:]` — вказана лише права межа зрізу. У цьому випадку з рядка **str** витягується підрядок, починаючи з позиції **a** і до кінця рядка.

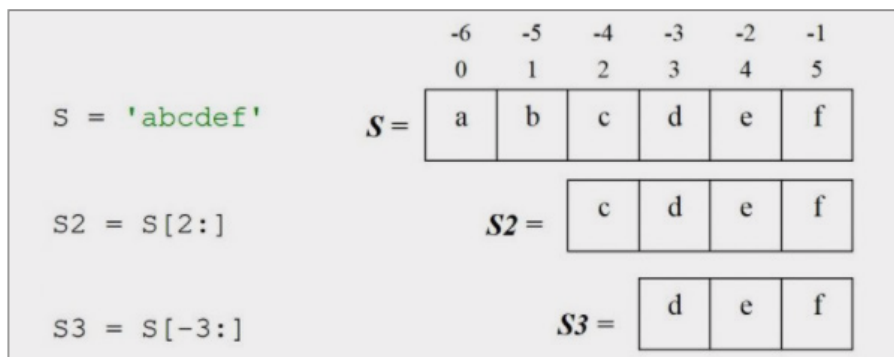


Рисунок 20

- `str[:]` — не вказана межа зрізу. У цьому випадку витягнутий підрядок буде копією рядка `str`, починаючи від початку до кінця.

Розглянемо ще приклади таких зрізів:

```
myStr="Python-cool!"
print(myStr[1:3]) #yt
print(myStr[-5:-2]) #coo
print(myStr[-5:11]) #cool
print(myStr[:6]) #Python
print(myStr[:-1]) #Python-cool
print(myStr[6:]) #-cool!
print(myStr[-5:]) #cool!
```

У розглянутих способах формування зрізів у зріз потрапляє кожен символ з оригінального рядка від лівої до правої межі (але не включаючи його), тобто крок зрізу дорівнює одиниці.

Розширені зрізи дозволяють задати величину кроку. Такий варіант зрізів має наступний синтаксис:

`str[a : b : k],`

де a і b — ліва і права межа зрізу;
 k — крок за індексом.

При формуванні такого зрізу величина кроку k додається до індексу кожного елемента, що витягується з рядка `str`. При цьому крок k може бути як позитивним, так і негативним:

- якщо $k \geq 0$, то зріз формується зліва направо (від a до b);
- якщо $k < 0$, то рядок зрізу формується справа наліво (від b до a).

Розглянемо можливі варіанти формування таких зрізів.

- `str[a : b : k]` — вилучаються усі елементи рядка `str`, починаючи з індексу a і завершуючи індексом $b - 1$ включно, зі зміщенням (кроком) k .
- `str[a : : k]` — вилучаються усі елементи рядка `str`, починаючи з індексу a і до кінця рядка зі зміщенням (кроком) k .
- `str[: b : k]` — вилучаються усі елементи рядка `str`, починаючи з індексу a і завершуючи індексом $b - 1$ включно, зі зміщенням (кроком) k .
- `str[: : k]` — вилучаються усі елементи рядка `str`, починаючи з початку рядка та до кінця зі зміщенням (кроком) k .

Розглянемо приклади таких зрізів:

```
myStr="1234567890"
print(myStr[2:8:2]) #357
print(myStr[8:2:-2]) #975
print(myStr[::-1]) #0987654321
```

```
print(myStr[5::2]) #680
print(myStr[-1::-2]) #08642
print(myStr[:len(myStr):3]) #1470
```

1.6. Екрановані послідовності

Припустимо, що нам потрібно відобразити (згідно з деяким макетом) наступний текстовий фрагмент:

```
Python books:
  'Python Programming: An Introduction to Computer Science'
  'The Python Guide for Beginners'
```

Рисунок 21

Якби ми набирали його в якомусь текстовому редакторі, то після першого рядка ми би натиснули клавішу «**Enter**» після першого та другого рядка тексту, щоб перейти на новий, а на початку другого та третього використовували б клавішу «**Tab**».

Таким чином, при створенні такого текстового фрагмента у вигляді рядка нам знадобляться символи-коди, які відповідають таким діям, як перехід на новий рядок та табуляція. Тобто механізму-обробнику символівної послідовності потрібно особливим чином вказати, що в цьому місці не слід відображати символ, і що це взагалі не символ, а лише вказівка виконати певні дії.

Для реалізації такого підходу у мовах програмування існують екрановані послідовності.

Екрановані послідовності є набором текстових символів, що починаються (і механізмом, що розпізнається, — обробником) з символу «\» (*backslash*), «\» є ознакою того,

що починається екранована послідовність, наступні за ним символи (і сам символ «\») не друкуються, а сприймаються обробником як код дії.

Окремі символи такої послідовності не мають сенсу для обробника, тому і називаються «послідовністю», тому що розпізнається саме набір символів у вказаному порядку після «\».

Екрановані послідовності також називають керуючими послідовностями, або **Escape** — послідовностями (англ. *Escape Sequence*).

Загальний синтаксис для представлення екранованих послідовностей:

```
"\СИМВОЛ (СИМВОЛИ)"
```

Для того, щоб зрозуміти, як екрановані послідовності впливають на відображення рядків, розглянемо найбільш поширені з них.

- **\n** — перехід на початок нового рядка (*newline*). За допомогою даної комбінації ми можемо переносити наступний за ним фрагмент на новий рядок.

Наприклад:

```
myStr="There is no shortage of material to learn
      Python.\nThe following books might serve
      as a starting point, in the order specified:
      \n'Python Programming: An Introduction to
      Computer Science' by John M. Zelle, Ph.D.
      \n'The Python Guide for Beginners' by Renan
      Moura.\n'Effective Python' by Brett Slatkin"
print(myStr)
```

Результат:

```
There is no shortage of material to learn Python.
The following books might serve as a starting point, in the order specified:
'Python Programming: An Introduction to Computer Science' by John M. Zelle, Ph.D.
'The Python Guide for Beginners' by Renan Moura.
'Effective Python' by Brett Slatkin
```

Рисунок 22

- `\t` — горизонтальна табуляція (виведення наступних символів розпочнеться з відступом по горизонталі).

Наприклад:

```
print("Python books:")
print("\t'Python Programming: An Introduction
      to Computer Science'")
print("\t'The Python Guide for Beginners'")
```

Результат:

```
Python books:
      'Python Programming: An Introduction to Computer Science'
      'The Python Guide for Beginners'
```

Рисунок 23

- `\x` — ознака шістнадцяткового коду символу, наприклад, `\x2C` — це уявлення шістнадцяткового коду символу коми.

Наприклад:

```
myStr="\x2C"
print(myStr) # ,
```

- `\ooo` — використовується для представлення вісімкового коду символів (`ooo` — умовне позначення трьох знакомісць для цифр коду), наприклад, `\053` — це представлення шістнадцяткового код символу «плюс».

Наприклад:

```
myStr="\053"
print(myStr) # +
```

Символ «`\`» використовується (зайнятий) екранованими послідовностями, проте є ситуації, коли нам потрібно його відобразити у рядку саме як символ «`\`».

Також є інші символи, які використовуються синтаксисом Python для інших цілей, наприклад, символ подвійних лапок або символ апострофа.

Для відображення (друкування) таких символів використовується підхід, званий екрануванням. При цьому безпосередньо перед спеціальним символом, який необхідно відобразити, вставляється символ «`\`»:

- `\\` — для відображення символу «`\`»;
- `\'` — для відображення символу апострофа;
- `\»` — для відображення символу подвійних лапок.

Наприклад:

```
print('\ Python Programming: An Introduction to
      Computer Science\')
print("\ Python Programming: An Introduction to
      Computer Science\"")
print("15\\2")
```

Результат:

```
'Python Programming: An Introduction to Computer Science'
"Python Programming: An Introduction to Computer Science"
15\2
```

Рисунок 24

1.7. «Сирі» рядки

Ми вже знаємо, що в Python escape-послідовності завжди починаються з символу «\» ([backslash](#)). Це зручно для форматування, але як діяти, якщо ми хочемо відобразити цей символ у рядку? Для цього, як було розглянуто раніше, використовується механізм екранування, при якому перед спеціальним символом, який потрібно відобразити, вставляється символ «\».

Наприклад:

```
myStr="Backslash symbol: \\"
print(myStr) # Backslash symbol: \
```

Тепер припустимо, що всередині рядка нам необхідно відобразити два таких символи «\n» і «\t». Набір символів «\n» і «\t» — це escape-послідовності, які реалізують перехід на новий рядок і табуляцію, відповідно, ці символи не відображаються. Знову застосуємо механізм екранування для їх відображення:

```
myStr="In Python strings, the backslash '\\' is
      a special character. It is used in representing
      certain whitespace characters: '\\t' is a tab,
      '\\n' is a newline"
print(myStr)
```

Зауважимо, що цей рядок важко сприймається, тому що потрібно пам'ятати про кожен позицію рядка (наприклад, «\»), яка буде друкуватися не так («\»), як вона виглядає в коді.

Ситуація ускладнюється, якщо в рядку є кілька символів «\», що йдуть поспіль, і доводиться точно вираховувати, який з «\» використовується для екранування, а який буде фактично відображатися при виведенні.

Для вирішення цієї проблеми в Python передбачені «Сирі» рядки (*raw strings*). *Raw strings* «розглядають» зворотну скісну риску (\) як буквальный символ. Це корисно у випадках, коли нам необхідно працювати з рядком, що містить зворотну скісну риску (\), і ми не хочемо, щоб вона розглядалася як escape-послідовність.

«Сирі» рядки Python мають префікс «r» або «R». Просто додайте до рядка префікс «R» або «r», і він буде розглядатися як «*raw strings*». Розглянемо на прикладах:

1)

```
normalStr = "This is a \nnormal string"
print(normalStr)

rawStr = r"This is a \n raw string"
print(rawStr)
```

Результат:

```
This is a
normal string
This is a \n raw string
```

Рисунок 25

2)

```

normalText='''Python Arithmetic Operators:\n
    Arithmetic operators are used to perform
    mathematical operations like addition,
    subtraction, multiplication and division.\n
    \tThere are 7 arithmetic operators in Python:
    \t\tAddition +\n
    \t\tSubtraction -\n
    \t\tDivision /\n
    \t\tModulus %\n
    \t\tExponentiation **\n
    \t\tFloor division //\n'''
rawText=r'''Python Arithmetic Operators:\n
    Arithmetic operators are used to perform
    mathematical operations like addition,
    subtraction, multiplication and division.\n
    \tThere are 7 arithmetic operators in Python:
    \t\tAddition +\n
    \t\tSubtraction -\n
    \t\tDivision /\n
    \t\tModulus %\n
    \t\tExponentiation **\n
    \t\tFloor division //\n
    '''
print(normalText)
print(rawText)

```

Результат:

```

Python Arithmetic Operators:
Arithmetic operators are used to perform mathematical operations like addition,
subtraction, multiplication and division.

    There are 7 arithmetic operators in Python:
        Addition +

```

Рисунок 26


```

Subtraction -

Division /

Modulus %

Exponentiation **

Floor division //

Python Arithmetic Operators:\n
Arithmetic operators are used to perform mathematical operations like addition,
subtraction, multiplication and division.\n
\tThere are 7 arithmetic operators in Python:
\t\tAddition +\n
\t\tSubtraction -\n
\t\tDivision /\n
\t\tModulus %\n
\t\tExponentiation **\n
\t\tFloor division //\n

```

Рисунок 26 (продолжение)

Однак не кожен рядок (послідовність символів у лапках) є коректним «сирим» рядком.

«Сирий» рядок, що містить лише один символ «\» неприпустимий. Так само «сирі» рядки з непарним числом символів «\».

Наприклад:

```

errRawStr1 = r'\ '
errRawStr2 = r'123\ '
errRawStr3 = r'abc\\ '

```

У цьому випадку інтерпретатор видасть таку помилку:

```

errRawStr1 = r'\ '
              ^
SyntaxError: unterminated string literal (detected at line 1)

```

Рисунок 27

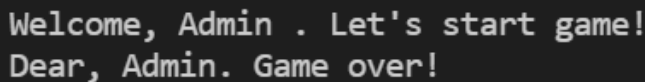
1.8. Форматований вивід

Ми вже знаємо, що для виведення (з'єднання) в одному рядку деяких текстових фрагментів (рядкових літералів) та рядкових змінних (нешвидких даних, що утворюються та змінюються в ході роботи програми) можна використовувати або конкатенацію, або (якщо потрібен лише висновок) функцію `print()` з кількома аргументами.

Наприклад:

```
userLogin=input("Your login: ")
print("Welcome,", userLogin, ". Let's start game!")
strMsg="Dear, "+userLogin+". Game over!"
print(strMsg)
```

Результат:



```
Welcome, Admin . Let's start game!
Dear, Admin. Game over!
```

Рисунок 28

Крім не дуже зручного способу підстановки у разі використання функції `print()`, ми бачимо зайвий символ пробілу після логіна користувача (це пов'язано з особливостями роботи функції `print()`, яка при заданні кількох аргументів поділяє їх значення при виведенні пробілом). Звичайно, таку поведінку за замовчуванням функції `print()` можемо змінити, задавши їй додатковий атрибут `sep=""`, але це ще більше знижує читання та розуміння коду.

```
print("Welcome,", userLogin,
      ". Let's start game!", sep="")
```

Більше того, використання атрибуту `sep=""` видалило пробіл між «Welcome!» та логіном користувача.

Більш зручним для такого роду форматування рядків є використання методу `format()`.

У випадку, коли в рядковий літерал потрібно підставити значення лише однієї змінної, можна використовувати такий синтаксис:

```
str.format(value)
```

- `str` — рядок-шаблон із заповнювачем `{}` (місце у шаблоні, куди слід помістити значення, передане аргументом методу).
- `value` — аргумент-значення, яке буде підставлятися у вказане місце рядка-шаблону.

Наприклад:

```
userLogin=input("Your login: ")
strMsg="Welcome, {}. Let's start game!".format(userLogin)
print(strMsg)
```

Результат:

```
Your login: Admin
Welcome, Admin. Let's start game!
```

Рисунок 29

Якщо змінних, значення яких потрібно вставити в шаблон, є декілька, то кількість заповнювачів повинна відповідати кількості аргументів методу `format()`.

При цьому заповнювачі можна ідентифікувати за допомогою іменованих індексів `{indexName}`, нумерованих індексів `{index}` або залишити порожніми `{}`, як у прикладі вище.

Розглянемо усі ці випадки на прикладі.

```
strMsg = "My name is {}, I'm {}".format("Student",25)
print(strMsg) # My name is Student, I'm 25
```

У разі використання порожніх заповнювачів `{}`, у них будуть підставлені аргументи методу `format()` чітко у тому порядку, як вони перелічені у методі.

Тобто, якщо поміняти місцями літерали «*Student*» та 25, то рядок втратить будь-який сенс:

```
strMsg = "My name is {}, I'm {}".format(25,"Student")
print(strMsg) # My name is 25, I'm Student
```

Для запобігання таким ситуаціям рекомендується ідентифікувати заповнювачі за допомогою іменованих індексів `{indexName}` або нумерованих індексів `{index}`.

Наприклад, використовуємо нумеровані індекси:

```
strMsg1 = "My name is {0}, I'm {1}".format("Student",25)
print(strMsg1) # My name is Student, I'm 25

strMsg2 = "I'm {1}. My name is {0}".format("Student",25)
print(strMsg2) # I'm 25. My name is Student
```

Або іменовані індекси, які забезпечують ще більш чітке позиціонування навіть за зміни порядку аргументів:

```
strMsg3 = "My name is {name}, I'm {age}".
          format(name="Student",age=25)
print(strMsg3) # My name is Student, I'm 25

strMsg4 = "My name is {name}, I'm {age}".
          format(age=25,name="Student")
print(strMsg4) # My name is Student, I'm 25
```

Також метод `format()` забезпечує нам додаткові можливості для виведення дійсних чисел, дозволяючи чітко задати кількість цифр після коми та загальну кількість знакомісць під число. Розглянемо на прикладі:

```
strMsg = "Your salary is {0:9.2f}".format(200.846)
print(strMsg) # Your salary is 200.85
```

Заповнювач `{0}`, який отримує значення з першого аргументу методу (бо індекс `0`), замінюється на значення `200.846`, яке попередньо піддається форматуванню за шаблоном `9.2f`:

- `f` — це специфікатор, який вказує на тип аргументу (`float`, речовий);
- кількість цифр після коми скорочується до двох (якщо в оригінального числа цифр після коми більше, то відбувається округлення, у прикладі до `200.85`);
- число у шаблоні `9.2f` до точки (цифра `9`) визначає загальну кількість знакомісць (позицій), що відводиться під виведення всього числа;
- якщо число знакомісць більше довжини числа (якщо довжина числа разом із символом «.» після округлення `6`), то вільні позиції заповнюються пробілом

і число наче відсувається вправо (у нашому прикладі було використано 3 пробіли перед числом);

- якщо число знакомісць менше довжини числа, то ця частина формату ігнорується (дана ситуація показана у прикладі нижче).

```
strMsg = "Your salary is {0:3.2f}".format(200.846)
print(strMsg) # Your salary is 200.85
```

Крім дійсних чисел також передбачені можливості форматованого виведення десяткових чисел, значень у двійковому, вісімковому, шістнадцятковому форматі та ін., для чого передбачені відповідні специфікатори.

Найбільш поширені типи специфікаторів наведено у наступній таблиці.

Таблиця 1

Специфікатор	Опис
:b	Двійковий формат
:c	Перетворює значення у відповідний символ Юнікоду
:d	Десятковий формат
:o	Вісімковий формат
:x	Шістнадцятковий формат, нижній регістр

Специфікатори поміщаються при цьому всередину заповнювачів {}, як у прикладах вище.

Розглянемо їхню дію на прикладах.

```
userNumber = int(input("Your number? ")) #255
myStrB = "The binary representation of a number {n}
         is {n:b}".format(n=userNumber)
```

```

print(myStrB) #The binary representation of a number
                255 is 11111111

myStrO="The octal representation of a number {n}
        is {n:o}".format(n=userNumber)
print(myStrO) #The octal representation of a number
                255 is 377

myStrH="The Hex representation of a number {n}
        is {n:x}".format(n=userNumber)
print(myStrH) #The Hex representation of a number
                255 is ff

```

Додаткові можливості форматування призначені для чисел зі знаками:

Таблиця 2

Тип форматування	Опис
:-	Відображає знак «мінус» тільки для від'ємних значень
:+	Відображає знак «плюс» лише для додатних значень
:символ пробілу	Вставляє додатковий пробіл перед додатними числами (і знак «мінус» перед від'ємними числами)

Приклади:

```

myStr1="The number1 range from {0:-}
        to {1:-}".format(-10,10)
print(myStr1) #The number1 range from -10 to 10

myStr2="The number2 range from {0:+} to {1:+"}.
        format(-20,50)
print(myStr2) #The number2 range from -20 to +50

```

```
myStr3="The number3 range from {0: }
      to {1: }".format(-30,30)
print(myStr3) #The number3 range from -30 to 30
```

Тепер розглянемо, які можливості надає нам метод `format()` для вирівнювання чисел та рядків.

Таблиця 3

Тип форматування	Опис
:<	Вирівнює по лівому краю (у межах доступного простору)
:>	Вирівнює по правому краю (у межах доступного простору)
:^	Вирівнює по центру (у межах доступного простору)

Наприклад:

```
#установим доступное пространство для значения
до 10 символов.
myStr1 = "You have {:<10} points."
print(myStr1.format(12)) #You have 12           points.

myStr2 = "You have {:>10} points."
print(myStr2.format(12)) #You have           12 points.

myStr3 = "You have {:^10} points."
print(myStr3.format(12)) #You have           12 points.
```

При вирівнюванні чисел після типу форматування можна вказати специфікатор:

```
myStr1 = "Number is {:<10.2f}!"
print(myStr1.format(34.8256)) #Number is 34.83      !
```



```
myStr2 = "Number is {:>10.2f}!"
print(myStr2.format(34.8256)) #Number is      34.83!

myStr3 = "Number is {:^10.2f}!"
print(myStr3.format(34.8256)) #Number is      34.83!
```

Розглянуті типи форматування також корисні під час вирівнювання рядків:

```
myStr1 = "Your login is {:<20}!"
print(myStr1.format("Admin")) #Your login is Admin  !

myStr2 = "Your password is {:>20}!"
print(myStr2.format("12345")) #Your password is  12345!

myStr3 = "Your secret word is {:^15}!"
print(myStr3.format("IT")) #Your secret word is   IT   !
```

1.9. Модуль string

У попередніх розділах ми познайомилися з багатьма корисними методами рядка (**str**). Більшість з них ми реалізували раніше в окремому модулі **string**. Однак, у зв'язку з високою частотою їх використання вони були перенесені в методи об'єкта **str**, щоб не потрібно було виконувати попередній імпорт модуля. Зараз модуль **string** містить набір констант, утиліт та класів для роботи з об'єктами **str** (наприклад, можливості створення власних шаблонів для налаштування власного форматування рядків).

Для використання можливостей модуля **string** спочатку необхідно виконати його імпорт наступним чином:

```
import string
```

Розглянемо спочатку перелік констант, що надаються модулем:

Таблиця 4

Ім'я константи	Опис	Значення
<code>string.ascii_letters</code>	Символи літер латинського алфавіту (верхнього та нижнього регістра)	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ
<code>string.ascii_lowercase</code>	Символи літер латинського алфавіту (у нижнього регістра)	abcdefghijklmnopqrstuvwxyz
<code>string.ascii_uppercase</code>	Символи літер латинського алфавіту (верхнього регістра)	ABCDEFGHIJKLMNOPQRSTUVWXYZ
<code>string.digits</code>	Символи цифр десяткової системи числення	0123456789
<code>string.hexdigits</code>	Символи цифр шістнадцяткової системи числення	0123456789 abcdefABCDEF
<code>string.octdigits</code>	Символи цифр вісімкової системи числення	01234567
<code>string.punctuation</code>	Символи пунктуації	!"#\$%&'()*+,-./ :;<=>?@[\] ^ _ { } ~
<code>string.whitespace</code>	Символи, які вважаються пробільними: перехід на новий рядок, табуляція і т.д.	
<code>string.printable</code>	Друковані символи (об'єднання множин <code>digits</code> , <code>ascii_letters</code> , <code>punctuation</code> та <code>whitespace</code>)	

Кожна з розглянутих констант є рядком, тобто `string.digits` це рядок «0123456789».

Розглянемо приклад практичного застосування цих констант.

Припустимо, що нам необхідно згенерувати випадковим чином логін користувача заданої довжини (наприклад, 6 символів), який має складатися лише із символів латинського алфавіту в нижньому регістрі, та пароль (8 символів), який має складатися із символів латинського алфавіту та десяткових цифр.

Для цього ми скористаємося методом `sample()` модуля `random`, який отримує випадкові елементи з послідовності в заданій кількості. Для об'єднання отриманого набору символів можна використовувати метод рядка `join()`.

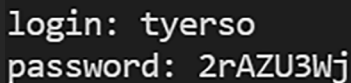
```
import string
import random

userLogin = "".join(random.
                    sample((string.ascii_lowercase), 6))

userPass = "".join(random.
                   sample((string.ascii_letters +
                           string.digits), 8))

print("login:", userLogin)
print("password:", userPass)
```

Результат:



```
login: tyerso
password: 2rAZU3Wj
```

Рисунок 30

До складу модуля `string` входить єдина функція `capwords()`:

```
capwords(strObj, sep = None)
```

де

- `strObj` — це рядок, який піддається обробці;
- `sep` — необов'язковий аргумент, що задає символ-розділювач у рядку-результаті.

Ця функція спочатку поділяє рядок `strObj` на слова, потім перший символ кожного слова переводить у верхній регістр, після — з'єднує перетворені слова у новий рядок, використовуючи вказаний роздільник.

Якщо аргумент `sep` не вказано (його значення за замовчуванням `None`), тоді початкові та кінцеві пробіли (якщо вони є) будуть видалені з рядка `strObj`, а слова нового рядка будуть з'єднані символом одного пробілу.

```
import string
myStr = " We have BEEN happy\n to welcome\n\n back
        OLD friends, \n\n\nand to make new ones  "

#We Have Been Happy To Welcome Back Old Friends,
And To Make New Ones
print(string.capwords(myStr))
```

Також модуль `string` надає нам два класи: `Formatter` та `Template`.

Функціональність першого (класу `Formatter`) практично збігається з можливостями, які надає розглянутий метод рядків `format()`, тобто цей клас є альтернативним рішенням для створення та налаштування форматування рядків так, як це необхідно розробнику.

```
from string import Formatter
formatter = Formatter()
```

```
print(formatter.format('{userLog}',
    userLog = 'Admin')) #Admin

print(formatter.format('{} {userLog}', 'Welcome, ',
    userLog = 'Admin')) #Welcome, Admin

print('{} {userLog}'.format('Welcome, ',
    userLog = 'Admin')) #Welcome, Admin
```

Як ми бачимо з прикладу, перший аргумент методу `format()` класу `Formatter` — це рядок-шаблон із заповнювачами, а наступні аргументи — набір значень (або змінних), які будуть підставлені до заповнювачів.

В останньому рядку нашого коду наведено приклад використання методу `format()` для рядка-шаблону, який забезпечив такий самий результат.

Другий клас `Template` корисний при створенні рядків-шаблонів з наступним внесенням до них необхідних даних.

Після його імпорту з модуля `string` шаблон створюється конструктором класу, якому, як аргумент, передається рядок із іменованими заповнювачами у потрібних позиціях.

```
from string import Template

t = Template('$UserName has the rights to $UserRights
    in the $appName')
```

У нашому прикладі у рядку три позиції із заповнювачами: `$UserName`, `$UserRights`, `$appName`. Далі необхідно застосувати шаблон, використовуючи метод `.substitute()`, якому передаються дані для вставки у заповнювачі:

```
resStr = t.substitute(userName='Admin',  
                        userRights = 'edit',  
                        appName='SuperApp.')
```

```
print(resStr) #Admin has the rights to edit in  
              the SuperApp.
```

1.10. Регулярні вирази, модуль re

Одним з найпоширеніших завдань при роботі з текстовими даними є завдання пошуку та перевірки на відповідність деяким умовам.

Раніше ми вже розглядали набір методів об'єкта рядків, за допомогою яких можна реалізувати пошук (методи `.find()`, `.index()` та їх варіації). Однак їх можливості обмежені тим фактом, що ключем пошуку є чітко визначений патерн (підрядок). Також, якщо хоча б один із символів у патерні знаходиться не в тому регістрі порівняно з рядком, то результат пошуку буде невдалим. А якщо патерн заздалегідь неможливо визначити? І є лише набір правил, якому рядок має відповідати. Наприклад, логін користувача має не просто складатися тільки з латинських символів та цифр, а містити хоча б одну літеру у верхньому регістрі.

Для вирішення подібних завдань можна на основі наявних правил (обмежень) створити шаблон та виконати його пошук у рядку, який необхідно перевірити на відповідність.

Саме для створення та використання таких шаблонів у багатьох мовах програмування (у тому числі й у Python) передбачені регулярні вирази.

Фактично, регулярні вирази (*regular expressions*, *regex*, *regex*, *RE*) — це спеціальна послідовність символів (рядків), яка задає шаблон. Такий шаблон може застосовуватися для пошуку, заміни, впорядкування, вилучення даних із рядка (або текстового файлу).

До складу шаблону, який ми створюємо з допомогою регулярного виразу, входять як звичайні символи, так і спеціальні символи (або їх послідовності).

Припустимо, що нам потрібно визначити, чи входить слово «*cat*» в склад фрази «*My Supercat*». У цьому випадку шаблон пошуку буде лише «*cat*».

А тепер у фразі «*My1 S123supercat*» знайдемо слова, до складу яких входять дві цифри підряд. Шаблон буде «*\d\d*», який виявить слово «*S123supercat*». У цьому прикладі, для представлення правила «одна будь-яка цифра», використовується послідовність «*\d*».

Робота з регулярними виразами в Python відбувається засобами модуля *re* (який спочатку необхідно імпортувати). Однак спочатку нам необхідно познайомитися з основами синтаксису *RE*.

Наприклад, якщо нам потрібно визначити, чи входить фрагмент «*student*» у логін користувача, ми будемо використовувати символи тільки зі слова «*student*» і в потрібній послідовності. А якщо правила для логіну такі: починається зі слова «*student*» і далі йдуть три будь-які цифри? В цьому випадку нам знадобиться дещо, що представить ситуацію «будь-яка цифра» і можливість задавати потрібне кількість таких екземплярів ситуації (три цифри в нашому прикладі). Для вирішення саме таких проблем у синтаксисі регулярних виразів є метасимволи.

До складу RE входять як окремі спеціальні символи (метасимволи), а й їх послідовності. До метасимволів відносяться символи точки «.», символ «^», знак долара «\$», символ «*», плюс «+», знак питання «?», фігурні «{}», квадратні «[]», що відкривається, закривається, круглі «()» дужки, зворотний слеш «\», вертикальна риса «|».

Метасимволи в RE є деякими керуючими конструкціями. Розглянемо призначення кожного з них.

Таблиця 5

Метасимвол	Призначення
.	Завдання (представлення) одного довільного символу (крім символу нового рядка)
^	Ознака початку послідовності
\$	Ознака закінчення послідовності
*	Позначає будь-яку кількість повторень одного символу (0 або більше), що передує символу «*»
+	Позначає будь-яку кількість повторень одного символу (1 або більше), попереднього до символу «+»
?	Позначає нуль або одне повторення одного символу, попереднього до символу «?»
{n}	Позначає задане число (n) повторень одного символу, попереднього до символу «{»
[]	Використовується для задання будь-якого символу з перелічених всередині []
\	Використовується для екранування метасимволів
	Відповідає логічному АБО (значення до або після символу « »)
()	Для створення групи символів (вираз усередині) розглядається як один елемент)

Наприклад, є умова, що логін користувача з правами адміністратора повинен містити слово «*admin*» (тобто логіни «*admin12*», «*admin12a*» «*54admin*», «*adminadmin*» — припустимі, а логіни «*user_adm23*», «*adm60*» — ні).

Регулярний вираз, що описує цю вимогу, буде таким:

```
'(admin)+ '
```

Оскільки нам потрібно перевірити наявність цілої групи символів і в зазначеному порядку (слово «*admin*»), ми використовували круглі дужки («*(admin)*»), а для вказівки того, що слово «*admin*» має зустрітися хоча б один раз — знак «*+*» після групи.

Розглянемо наступний приклад: пароль користувача має містити лише символи латиниці нижнього регістра (довжина пароля — довільна, тобто пароль із одного символу також підходить).

Раніше ми вже вивчали можливості та функціональність модуля `string`, в якому є корисна для даної задачі константа — `string.ascii_lowercase`, яка є рядком із символів латинського алфавіту нижнього регістра.

Оскільки до складу пароля може входити будь-який із цих символів, нам знадобиться метасимвол `[]`. Шаблон зберемо за допомогою конкатенації рядків таким чином:

```
'['+string.ascii_lowercase+'] '+'+'
```

Тепер розглянемо шаблони (які входять до складу RE) для представлення одного символу.

Таблиця 6

Шаблон	Призначення
<code>\d</code>	Відповідає одній десятковій цифрі
<code>\D</code>	Відповідає одному будь-якому символу, крім десяткової цифри
<code>\s</code>	Відповідає одному (будь-якому) пробільному символу
<code>\S</code>	Відповідає одному (будь-якому) символу, який не відноситься до пробільних
<code>\w</code>	Відповідає будь-якому символу з літер і цифр або символу нижнього підкреслення («_»)
<code>\W</code>	Відповідає будь-якому символу не з літер та цифр, та не символу нижнього підкреслення («_»)
<code>[..]</code>	Відповідає будь-якому з символів, перелічених у дужках, можна також вказувати діапазони символів (наприклад, <code>[0-5]</code> — будь-яка цифра від 0 до 5). Увага! Метасимволи всередині <code>[]</code> втрачають своє спеціальне значення і позначають просто символ. Наприклад, точка всередині <code>[]</code> позначатиме саме точку, а не будь-який символ.
<code>[^..]</code>	Відповідає будь-якому одному символу, крім перелічених у дужках або крім тих, що потрапляють до зазначеного діапазону.
<code>\b</code>	Відповідає початку або кінцю слова (тобто ліворуч від <code>\b</code> пробіл або не літерний символ, праворуч буква і навпаки – для кінця слова). На відміну від попередніх шаблонів, вона відповідає позиції, а не символу.
<code>\B</code>	Відповідає «внутрішньому» (неграничному) символу слова (тобто ліворуч і праворуч від <code>\B</code> літерні символи, або зліва і праворуч від <code>\B</code> не літерні символи).

Розглянемо приклади шаблонів з метасимволами та результати їх застосування до рядків.

Таблиця 7

Приклад шаблону	Опис шаблону	Приклади рядків, що відповідають шаблону
<code>course. topic.part.a</code>	Підходять усі рядки, що містять фрагменти <code>course</code> , <code>topic</code> , <code>part</code> , а у зазначеному порядку. Між фрагментами має бути один довільний символ, крім символу нового рядка	<code>course1topic2part7a</code> <code>courseAtopic1partDa</code> <code>mycourse1topic5partCapital</code>
<code>course\d\d</code>	Підходять усі рядки, що містять фрагмент <code>course</code> , одразу після якого мають йти дві десяткові цифри	<code>course25</code> <code>mycourse01</code> <code>mycourse01part23</code>
<code>part\D001</code>	Підходять усі рядки, що містять фрагмент <code>part</code> , відразу після якого має бути один будь-який символ, крім десяткової цифри, після якого йде фрагмент 001	<code>part-001</code> <code>part@001</code>
<code>user\s24</code>	Підходять усі рядки, що містять фрагмент <code>user</code> , відразу після якого має бути один будь-який пробільний символ, після якого йде фрагмент 24	<code>user 24</code> <code>user</code> <code>24</code>
<code>user-\S007</code>	Підходять усі рядки, що містять фрагмент <code>user-</code> , відразу після якого повинен бути один будь-який не пробільний символ, після якого йде фрагмент 007	<code>user-A007</code> <code>my user-1007</code>
<code>\w\w555</code>	Підходять усі рядки, що містять фрагмент із двох будь-яких символів з літер і цифр (або символу нижнього підкреслення), одразу після якого йде фрагмент 555	<code>aa555</code> <code>12555</code> <code>a_555</code>

Приклад шаблону	Опис шаблону	Приклади рядків, що відповідають шаблону
Python\W	Підходять усі рядки, що містять фрагмент Python, одразу після якого йде один символ, що не відноситься до символів з літер і цифр (і не є символами нижнього підкреслення)	Python! Python?
[0-5] [0-7A-Ca-c]	Підходять усі рядки, що містять фрагмент із двох символів, перший з яких це цифра в діапазоні від 0 до 5, а другий – це цифра від 0 до 7, або символ латинського алфавіту від «A» до «C», або від «a» до «c»	1a 2B 4c
[^B-Db-d]	Підходять усі рядки, що містять фрагмент одного символу, поміщеного у круглі дужки. Це має бути символ латинського алфавіту, крім символів у діапазонах від «B» до «D» і від «b» до «d»	(1) (a) (A) (e)

При створенні шаблону нам часто потрібно не тільки сформулювати конструкцію, яка описує правило відповідності, але й вказати кількість символів, які мають бути в аналізованому рядку, щоб відповідність була зафіксована.

У регулярних виразах для цього призначені квантифікатори. Один із квантифікаторів ми вже розглянули вище — це число у фігурних дужках {*n*}. Квантифікатор вказується після символу (або набору [...]), визначаючи, скільки таких екземплярів нам потрібно.

Таблиця 8

Квантифікатор	Кількість повторень, що задається
{n}	дорівнює n
{m,n}	від m до n (включно)
{m,}	від m і більше (не менше, ніж m)
{,n}	до n (не більше, ніж n)
?	нуль або одне
*	нуль або більше нуля (від нуля включно)
+	одне або більше одного (як мінімум одне)

Розглянемо приклади використання квантифікаторів у шаблонах.

Таблиця 9

Приклад шаблону	Опис шаблону	Приклади рядків, що відповідають шаблону
<code>student\d{5}</code>	Підходять усі рядки, що містять фрагмент <code>student</code> , одразу після якого йде п'ять десятичних цифр	<code>student12345</code> <code>student80567</code>
<code>student\d{3,5}</code>	Підходять усі рядки, що містять фрагмент <code>student</code> , одразу після якого йдуть від трьох до п'яти десятичних цифр	<code>student12345</code> <code>student0000</code> <code>student805</code>
<code>user\d{3,}</code>	Підходять усі рядки, що містять фрагмент <code>user</code> , відразу після якого йдуть десятичні цифри у кількості не менше трьох	<code>user111</code> <code>user1111111111</code>
<code>user\d{2}</code>	Підходять усі рядки, що містять фрагмент <code>user</code> , відразу після якого йдуть десятичні цифри у кількості не більше двох	<code>user1</code> <code>user99</code>

Приклад шаблону	Опис шаблону	Приклади рядків, що відповідають шаблону
<code>come?</code>	Підходять усі рядки, що містять фрагмент <code>come</code> , причому останній символ «e» може бути як присутнім, так і ні	<code>come</code> <code>coming</code> <code>com.</code>
<code>user\d*</code>	Підходять усі рядки, що містять фрагмент <code>user</code> , після якого може йти одна десяткова цифра (але її наявність необов'язкова)	<code>user</code> <code>user1</code> <code>user2345</code>
<code>user\d+</code>	Підходять усі рядки, що містять фрагмент <code>user</code> , після якого має йти як мінімум одна десяткова цифра	<code>user1</code> <code>user99</code> <code>user9999999999999999</code>

Як зазначено вище, робота з регулярними виразами в Python передбачає використання можливостей модуля `re`.

Найбільш поширеними та використовуваними функціями модуля `re` для виявлення збігів є:

- `re.search(pattern, strObj)` — шукає у рядку `strObj` перший збіг із шаблоном `pattern`;
- `re.findall(pattern, strObj)` — шукає у рядку `strObj` усі збіги (які не перетинаються) з шаблоном `pattern` (результат — список рядків, що збіглися з шаблоном);
- `re.match(pattern, strObj)` — шукає на початку рядка `strObj` збіг із шаблоном `pattern`.

Крім функцій пошуку збігів, модуль `re` надає нам ще такі корисні функції, як `re.sub()` для заміни знайдених збігів на новий фрагмент і `re.split()` для розбиття рядка по фрагментах, які збігаються з шаблоном.

Деякі з перелічених функцій повертають об'єкт `Match` (у разі виявлення збігу з шаблоном в рядку, що аналізується). Розглянемо його особливості з прикладу.

Припустимо, що у рядку необхідно знайти перше слово, що складається лише з чотирьох символів літер і цифр. Для цього нам знадобиться такий шаблон: `'\w{4}'`.

```
import re

userStr="abcd abc efgh"
match = re.search(r'\w{4}', userStr)
```

Шаблон подаємо як «сирий» рядок для обліку (за потреби) всіх символів (у тому числі і недрукованих). Якщо збіг з шаблоном не знайдено, то функція `search()` поверне `None`.

Інакше для подальшої роботи з результатами пошуку ми можемо скористатися одним із методів об'єкту `Match`:

- `group()` — повертає підрядок, який збігся з усім шаблоном або з виразом у групі шаблону (розглянуто далі).

Якщо потрібно отримати весь підрядок, який збігається з шаблоном, то метод викликається без аргументів або з аргументом `0`:

```
print(match.group()) # abcd
print(match.group(0)) # abcd
```

Тепер спробуємо знайти перше входження в рядок послідовності (фрагменту) завдовжки в три символи, який складається лише з цифр.

Шаблон: `\d{3}`

```
import re

userStr="abcd abc 123 efgh 456"
match = re.search(r'\d{3}', userStr)
print(match.group()) # 123
```

А тепер розглянемо, як працювати з групами в шаблонах. Припустимо, що нам необхідно проаналізувати рядок на наявність у ньому номерів мобільних телефонів двох операторів (відомо код оператора).

```
import re

userStr="My cell phone numbers: Vodafone +38(095)1234567;
        Cellcom +38(067)9875612";
match1 = re.search(r'Vodafone \+38\ (095\
                  (\d\d\d\d\d\d\d\d); Cellcom \+38\ (067\
                  (\d\d\d\d\d\d\d\d))', userStr)
```

Як бачимо з прикладу, символи «+», а також символи круглих дужок, що відкриваються «(» і закриваються «)», у номерах телефонів необхідно екранувати, тому що вони мають спеціальне значення у регулярних висловлюваннях. Так, за допомогою `()` ми створили дві групи (для кожного з номера телефону), щоб далі можна було працювати з кожним із номерів окремо.

```
print(match1.group()) # Vodafone +38(095)1234567;
                      Cellcom +38(067)9875612
print(match1.group(0)) # Vodafone +38(095)1234567;
                      Cellcom +38(067)9875612
```


Для отримання підрядків, що збіглися з певною групою, потрібно викликати метод `group()` з її номером у якості аргументу:

```
print(match1.group(1)) # 1234567
print(match1.group(2)) # 9875612
```

Якщо до якоїсь групи нічого не потрапило, буде порожній рядок.

Якщо вказати номер групи більше, ніж загальна кількість груп, інтерпретатор видасть помилку «*no such group*».

Метод `group()` також може приймати кілька аргументів (можна ставити кілька номерів груп). У цьому випадку результат — кортеж, який складається з підрядків, що відповідають збігам із групами.

```
print(match1.group(1,2)) # ('1234567', '9875612')
```

Також у об'єкта `Match` є два корисні методи для отримання індексів початку і кінця, що збігаються з шаблоном фрагмента: `start()` та `end()`. Дані методи, за аналогією з методом `group()`, можуть працювати без аргументів (для отримання індексів фрагмента з повним збігом) або номерами груп.

Так для нашого прикладу:

```
print(match1.start(), match1.end()) #23 73
print(match1.start(0), match1.end(0)) #23 73
print(match1.start(1), match1.end(1)) #40 47
print(match1.start(2), match1.end(2)) #66 73
```

У розглянутих методах `start()` і `end()` є аналог — метод `span()`, який повертає кортеж, що складається з

індексу початку та індексу кінця, збігається з шаблоном фрагмента:

```
print(match1.span()) # (23, 73)
print(match1.span(0)) # (23, 73)
print(match1.span(1)) # (40, 47)
print(match1.span(2)) # (66, 73)
```

Одночасно з об'єктом `Match` ми розглянули приклад роботи з функцією `re.search(pattern, strObj)`, при виклику якої, якщо пошук у рядку `strObj` першого збігу із шаблоном `pattern` вдалий, то результат — об'єкт `Match`, інакше — `None`.

Ця функція корисна, якщо потрібно знайти лише один збіг у рядку. Якщо нам необхідно знайти усі збіги, то можна використовувати функцію `re.findall(pattern, strObj)`.

Розглянемо з прикладу. Припустимо, що нам потрібно вивести всі дати (у форматі `дд.мм.рррр`) із тексту з розкладом змагань.

```
import re

userStr="2021-2022 Competition Calendar:30.11.2021 —
        2021 Grand Prix Series; 14.01.2022 —
        Grand Premio D'Italia"

match2=re.findall(r'\d{2}\.\d{2}\.\d{4}', userStr)

print(match2) # ['30.11.2021', '14.01.2022']
```

Залежно від того, чи є в шаблоні групи, результат, який повертається функцією `findall()`, може бути:

- Список рядків, які відповідають шаблону (якщо у шаблоні немає груп, як у прикладі);

- Список рядків, які відповідають фрагменту шаблону у групі (якщо у шаблоні лише одна група);
- Список кортежів, що складаються з рядків, які відповідають фрагментам шаблону в групах (якщо груп у шаблоні декілька).

У ситуації, коли нам необхідно перевірити, що початок рядка відповідає заданому шаблону, підійде функція `re.match(pattern, strObj)`, яка аналогічно до функції `search()` повертає об'єкт `Match` при збігу і `None` — навпаки.

```
import re

userStr1="My cell phone numbers: Vodafone
        +38(095)1234567; Cellcom +38(067)9875612";
userStr2="Vodafone +38(095)1234567; Cellcom
        +38(067)9875612 — my cell phone numbers";

match3 = re.match(r'Vodafone \+38\.(095\.)
                (\d\d\d\d\d\d\d); Cellcom \+38\.(067\.)
                (\d\d\d\d\d\d\d)', userStr1)
match4 = re.match(r'Vodafone \+38\.(095\.)
                (\d\d\d\d\d\d\d); Cellcom \+38\.(067\.)
                (\d\d\d\d\d\d\d)', userStr2)

print(match3) #None
print(match4.group()) # Vodafone +38(095)1234567;
                      Cellcom +38(067)9875612
```

Наприклад, якщо застосувати її для виявлення розглянутого вище патерну «`Vodafone \+38\.(095\.) (\d\d\d\d\d\d\d); Cellcom \+38\.(067\.) (\d\d\d\d\d\d\d)`» для перевірки рядка `userStr1` «My cell phone numbers: Vodafone +38(095)1234567; Cellcom +38(067)9875612», то результат буде `None`, тому

що рядок не починається з даного шаблону, а містить його в середині, а результат перевірки `userStr2` успішний, отриманий об'єкт `Match`.

Іноді в завданнях обробки текстової інформації ми стикаємося з необхідністю заміни окремих фрагментів контенту (які задовольняють певній умові) на нову інформацію.

Модуль `re` надає нам функцію `re.sub(pattern, strRepl, strObj)`, яка замінює знайдені у рядку `strObj` збіги із шаблоном `pattern` на новий фрагмент `strRepl`.

Припустимо, що нам необхідно замінити у нашому розкладі змагань символи мінуса та крапки з комою на символ пробілу.

```
import re

userStr="2021-2022 Competition Calendar:
        30.11.2021 — 2021 Grand Prix Series;
        14.12.2021 — Grand Pemio D'Italia"
newStr=re.sub(r'[-;]', ' ', userStr)
print(newStr) #2021/2022 Competition
               Calendar:30.11.2021 /
               2021 Grand Prix Series/ 14.12.2021 /
               Grand Pemio D'Italia
```

Ми вже ознайомилися з методом `.split()`, який розбиває рядок на частини, раніше при вивченні методів об'єкта рядка.

В модуль `re` включена функція `split`, яка працює подібно, але дозволяє виконувати розбиття на підставі більш складних умов. Отриманий результат, — набір (список) рядків.

Наприклад, рядок «30.11.2021 — 2021 Grand Prix Series, 14.12.2021 — Grand Pemio D'Italia; 27.12.2021 — Cup of Austria by IceChallenge» треба розбити на окремі рядки (для кожного змагання). Як ми бачимо, інформація про змагання розділяється в одному випадку комою, а в іншому — крапкою з комою.

```
import re

userStr="30.11.2021 — 2021 Grand Prix Series,
        14.12.2021 — Grand Pemio D'Italia;
        27.12.2021 — Cup of Austria by IceChallenge"

strList=re.split(r'[;,]+', userStr)

print(strList) #['30.11.2021 — 2021 Grand Prix Series',
# ' 14.12.2021 — Grand Pemio D'Italia',
# ' 27.12.2021 — Cup of Austria by IceChallenge']
```

2. Списки

2.1. Поняття класичного масиву

Часто для зберігання інформації про деякий об'єкт предметної області нам недостатньо одного значення.

Базові типи даних (цілочисленні, речові та рядки) можуть одночасно зберігати лише одне значення, відповідно, де вони підходять у цій ситуації, тому що при привласненні змінної нового значення попереднє «затирається» і для його зберігання потрібна ще одна змінна.

А якщо можливих значень такої змінної не два, а набагато більше? Наприклад, нам потрібно зберігати оцінки студентів групи, у групі 30 осіб. Тоді нам доведеться створити 30 окремих змінних для збереження оцінки кожного студента групи? А якщо груп декілька?

І тут ми підходимо до необхідності створення такої змінної (та такого типу даних), яка може зберігати декілька значень. При цьому логічно перенумерувати їх за порядком послідовності та отримати до них доступ далі за цим порядковим номером. Саме в такий спосіб і організована класична структура даних під назвою «масив», що відноситься до складових типів даних.

Адреса	n	$n+k$	$n+2k$	$n+k(q-1)$	
Значення	$a[0]$	$a[1]$	$a[2]$...	$a[q-1]$
Індекс	0	1	2	$q-1$	

Рисунок 31

У деяких мовах програмування (наприклад, C++) масив — це впорядкований набір елементів однакового типу, які послідовно розташовані в пам'яті (рис. 31).

Представлений на цьому малюнку масив містить q елементів (одного типу даних) з індексами (порядковими номерами елементів в масиві) від 0 до $q-1$. Кожен елемент займає у пам'яті комп'ютера однакове число (k) байт, при цьому елементи розміщені у пам'яті один за одним (безперервна область пам'яті).

У Python класичного масиву, як окремого типу даних, немає. Натомість нам надається можливість використання різних колекцій, що мають більш різноманітну функціональність у порівнянні з класичними масивами.

2.2. Поняття колекції об'єктів

Уявімо, що нам необхідно зберігати та обробляти жанри-категорії фільмів: комедія, пригод, вестерн, тобто з однією змінною `category` буде пов'язаний цілий набір значень: «Drama», «Comedy», «Fantasy», «Western» (у цьому прикладі усі значення одного типу даних — рядки). Також можливі ситуації, коли потрібно, щоб у змінної було декілька значень різних типів, наприклад, список предметів може містити, як назви предметів, так і їх цілочисельні ідентифікатори («Algorithms», 2345, 7009, «Networks», «Databases»).

Саме для таких ситуацій нам потрібні колекції. У Python під колекцією розуміється об'єкт, який може зберігати кілька значень. При цьому, ці значення можуть бути як одного, так і різних типів даних (у тому числі й іншими об'єктами).

Залежно від типу колекції:

- можуть бути змінними або незмінними;
- мати певні методи;
- бути впорядкованими (індексованими) або неупорядкованими;
- складатися з набору тільки унікальних елементів або в них можливі повторення значень.

Розглянемо перелічені вище характеристики колекцій докладніше.

Змінність — над колекцією допустимі операції додавання нових та видалення існуючих значень.

Упорядкованість — кожен елемент колекції характеризується не лише своїм значенням, а й індексом (порядковим номером елемента в колекції). З поняттями індексу, індексацією та зрізами ми вже познайомилися раніше під час роботи з рядками.

Унікальність — колекція складається тільки з унікальних елементів.

Як було сказано вище, у кожної колекції є набір власних методів для роботи з її елементами або колекцією в цілому. Однак, також є і набір операцій, які є допустимими та корисними при роботі з будь-якою колекцією незалежно від її типу:

1. Визначення довжини колекції (кількості її елементів) за допомогою вбудованої функції `len()`;
2. Перевірка приналежності деякого елемента до колекції за допомогою оператора приналежності `in`;
3. обхід колекції за допомогою циклу `for in`;
4. Виведення усіх елементів колекції за допомогою вбудованої функції `print()`.

2.3. Посилальний тип даних list

Перший тип колекцій у Python, з якою ми познайомимося, — це список. Практично у будь-якій задачі, незалежно від предметної області, ми стикаємося з поняттям списку: список студентів, список інгредієнтів рецепту, список книг, список фільмів тощо.

У Python список (*list*) — це змінена, впорядкована (тобто індексується) колекція значень будь-яких (зокрема і різних) типів даних. При цьому в одному списку значення можуть повторюватися (тобто список не має властивість унікальності).

Список (*list*) є об'єктом, тобто після створення змінної типу `list` вона зберігатиме адресу об'єкта класу «список». За цією адресою в пам'яті резервується область, де зберігаються адреси (посилання) на елементи списку в пам'яті.

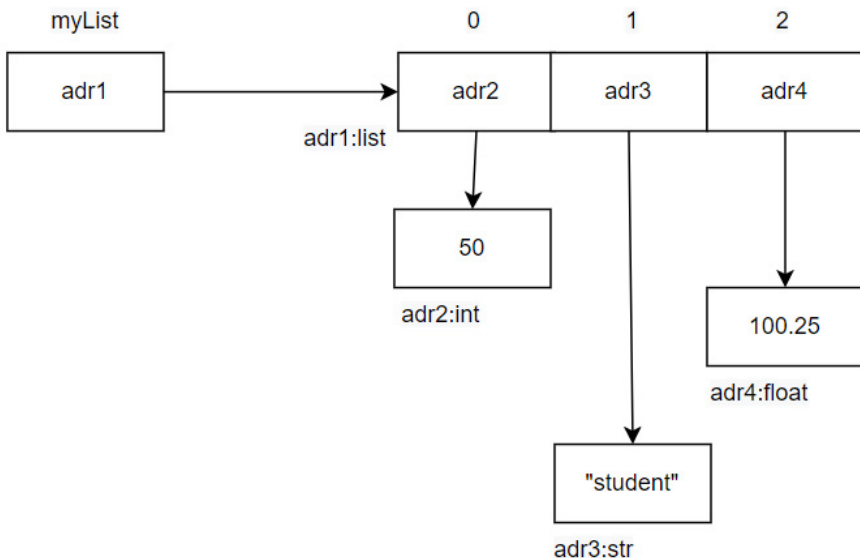


Рисунок 32

Наприклад, після створення змінної-списку `myList`, що містить три значення (ціле число `50`, рядок «`student`», дійсне число `100.25`), у пам'яті, за адресою `adr1`, було виділено місце для зберігання трьох адрес (посилань): `adr2`, `adr3`, `adr4`. За адресою `adr2` у пам'яті розташовується значення першого елемента списку — ціле число `50` і т. д. Таким чином, кожен елемент списку зберігає адресу іншого об'єкта (посилання на інший об'єкт).

2.4. Створення списків

Для того, щоб використовувати у своїх завданнях функціональність списків, нам необхідно попередньо створити його (визначити).

Визначити списки в Python можна двома способами:

1. Можна створити змінну та ініціалізувати її набором елементів, укладених у квадратні дужки:

```
category = ["Drama", "Comedy", "Fantasy"]
```

2. А можна скористатися вбудованою функцією `list`, передавши їй значення елементів у вигляді набору:

```
courses = list(("Math", "Algorithms", "Databases"))
```

Примітка: звертаємо увагу на подвійні круглі дужки, що відкриваються і закриваються! Функція `list` може викликатись або без аргументів (для створення порожнього), або з одним аргументом. Тому, для передачі кількох значень ми укладаємо набір у власні круглі дужки. В обох випадках результат буде однаковим — буде створено об'єкт-список.

Скористайтесь функцією `print()` для виведення елементів списку:

```
print(category) #['Drama', 'Comedy', 'Fantasy']
print(courses) #['Math', 'Algorithms', 'Databases']
```

Передаючи як аргумент функції `print()` ім'я списку ми виводимо всі його значення (разом із символами `[]`, знаками апострофа, тому що елементи — це рядки, і комами у вигляді роздільників елементів списку).

```
['Drama', 'Comedy', 'Fantasy']
['Math', 'Algorithms', 'Databases']
```

Рисунок 33

Якщо нам необхідно створити порожній список (без елементів), наприклад, коли перелік значень ще невідомий, це також можна реалізувати двома способами:

```
studentScores=[]
students=list()

print(studentScores) #[]
print(students) #[]
```

Спроба вивести порожній список призведе до відображення порожніх квадратних дужок:

```
[]
[]
```

Рисунок 34

Тепер припустимо, що у нашому списку мають бути елементи різних типів:

```
myCourses= ["Algorithms", 2345, 7009, "Networks",
            "Databases"]
print(myCourses) #['Algorithms', 2345, 7009,
                  'Networks', 'Databases']
```

Також елементом списку може бути інший список:

```
nestedList=[1,2.5,[45, "Example"]]
print(nestedList) #[1, 2.5, [45, 'Example']]
```

Елементи списку можуть повторюватися:

```
customers=['Bob', 'Anna', 'Joe', 'Bob', 'Nick']
```

Також, якщо передати функції `list()` рядок, результатом її роботи буде список, що складається з символів рядка:

```
mySymbols=list("abcdef")
print(mySymbols) #['a', 'b', 'c', 'd', 'e', 'f']
```

2.5. Генератори списків

Крім розглянутих підходів до створення списків, існує додатковий спосіб — за допомогою генераторів списків.

List comprehensions (спискові включення, генератори списків) — це спосіб створення списку на основі значень, що генеруються (за заданим правилом).

Розглянемо загальний синтаксис такого генератора:

```
newList = [expression for item in sequence]
```

expression — вираз (обчислення), яке виконується над кожним елементом **item** у послідовності **sequence**. Результат роботи генератора — новий перелік **newList**.

Розглянемо роботу генератора на прикладах.

Припустимо, що нам необхідно згенерувати список із квадратів чисел у діапазоні від 0 до 5:

```
list1=[i*i for i in range(6)]
print(list1) #[0, 1, 4, 9, 16, 25]
```

У цьому прикладі:

- **i*i** — це вираз, який використовується для отримання квадрата числа;
- **i** — кожен елемент із послідовності 0, 1, 2, 3, 4, 5 (sequence), яка була отримана в результаті роботи функції **range(6)**.

У якості послідовності може бути використаний і рядок.

Припустимо, що ми маємо рядок «**example**», з елементів якого потрібно створити список, приєднуючи до них символ *****.

```
list2=[i+"*" for i in "example"]
print(list2) #['e*', 'x*', 'a*', 'm*', 'p*', 'l*', 'e*']
```

Або, наприклад, потрібно створити список із продубльованих 5 разів символів рядка:

```
list3=[i*5 for i in "abcdf"]
print(list3) #['aaaaa', 'bbbbb', 'ccccc', 'dddd',
              'ttttt', 'fffff']
```

У генератори списків можна додавати умову, виходячи з якої вибираються елементи для обробки виразом. У цьому випадку синтаксис буде таким:

```
newList = [expression for item in sequence if condition]
```

expression — вираз (обчислення), яке виконується над таким елементом **item** у послідовності **sequence**, для якого **condition==True**. Умова схожа на деякий фільтр, який відбирає для обробки та поміщення в новий список тільки ті елементи, для яких ця умова виконується.

Припустимо, що нам необхідно згенерувати список із квадратів парних чисел у діапазоні від 1 до 10:

```
list4=[i*i for i in range(1,11) if i%2==0]
print(list4) #[4, 16, 36, 64, 100]
```

Або вибрати із вже наявного списку усіх клієнтів, крім *Bob* та *Joe*:

```
customers=['Bob', 'Anna', 'Joe', 'Bob', 'Nick']
list5=[i for i in customers if i!='Bob' and i!='Joe']
print(list5) #['Anna', 'Nick']
```

В Python також передбачена можливість створення генератора з кількома циклами.

```
list6 = [x*y for x in range(1, 4) for y in range(1, 4)]
print(list6) #[1, 2, 3, 2, 4, 6, 3, 6, 9]
```

У цьому прикладі таблиця множення на три представлена у вигляді списку. Тут цикл **for x in range(1, 4)** є

вкладеним у цикл `for y in range(1, 4)`, а дія, що створює елемент нового списку, `x*y` — перебуває у внутрішньому циклі, тобто повторюється 9 разів.

Генератори списків також корисні, якщо потрібно створити вкладений перелік. Наприклад, щоб представити ту саму таблицю множення на три у вигляді матриці:

```
list7 = [[x*y for x in range(1, 4)] for y in range(1, 4)]
print(list7) #[[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

Цикл, який наповнює елементами кожен рядок нового списку, поміщений разом із виразом у власні квадратні дужки.

2.6. Робота зі списками

Раніше нами вже зазначали таку особливість списків, як «упорядкованість» або «індексованість», тобто до окремих елементів списку ми можемо отримувати доступ за його індексом.

Подібно до рядків та інших упорядкованих колекцій, індексація елементів списку починається з нуля, і можливе звернення до елементів за допомогою від'ємних індексів.

Припустимо, що ми маємо список:

```
myList ["user", 12, 200.34, False, True]
```

Таблиця 10

0	1	2	3	4
"user"	12	200.34	False	True
-5	-4	-3	-2	-1

Перший елемент списку зі значення «user» ми можемо отримати за індексом 0 або -5 (як і в рядках, від'ємний індекс останнього елемента списку -1, а індекс першого — від'ємне значення довжини списку):

```
myList[0], myList[-5].
```

Розглянемо на прикладах:

```
myList = ["user", 12, 200.34, False, True]
print(myList [1]) #12
print(myList [-1]) #True
print(myList [len(L)-1]) #True
print(myList [-2]) #False
```

Спроба звернення до елемента за неіснуючим індексом (наприклад, `myList[5]` або `myList[-7]`) викличе помилку: *IndexError: list index out of range*.

Також нам доступна можливість використовувати зрізи (спосіб отримання безперервного фрагмента списку за одну дію). Синтаксис зрізів списку аналогічний розглянутим раніше зрізам рядків. Ми можемо формувати як базові зрізи із зазначенням меж зрізу (обох або однієї з), так і розширені зрізи із зазначенням кроку індексування.

Розглянемо роботу з різними варіантами зрізів списку на прикладах.

```
myCourses= ["Algorithms", 2345, 7009, "Networks",
            "Databases"]

print(myCourses[1:3])    #[2345, 7009]
print(myCourses[-4:-2])  #[2345, 7009]
print(myCourses[1:-1])   #[2345, 7009, 'Networks']
```



```
print(myCourses[:-1])    #['Algorithms', 2345, 7009,
                        'Networks']
print(myCourses[3:])    #['Networks', 'Databases']
print(myCourses[::2])   #['Algorithms', 7009, 'Databases']
print(myCourses[::-1])  #['Databases', 'Networks',
                        7009, 2345, 'Algorithms']
print(myCourses[-4::-1]) #[2345, 'Algorithms']
```

На відміну від рядків, списки — це колекція, що змінюється, тобто з використанням індексу ми можемо змінювати значення елемента:

```
category=["Drama", "Comedy", "Fantasy"]
print(category) #['Drama', 'Comedy', 'Fantasy']
category[-1]="Action"
print(category) #['Drama', 'Comedy', 'Action']
```

Можлива зміна значень цілих фрагментів списку, використовуючи зрізи. Наприклад, необхідно замінити логіни через одного користувача (тобто у користувачів «*admin*», «*teacher*», «*user*») на логіни “*newUser1*”, “*newUser2*” і т. д.

```
userLogs=["admin","student","teacher","HR","user"]
print(userLogs) #['admin', 'student', 'teacher',
                'HR', 'user']
userLogs[::2]=["newUser1","newUser2","newUser3"]
print(userLogs) #['newUser1', 'student', 'newUser2',
                'HR', 'newUser3']
```

Примітка! Кількість нових значень має дорівнювати довжині зрізу (у нашому прикладі в результаті зрізу ми отримаємо 3 елементи, відповідно список нових значень має містити також 3 елементи).

Для роботи зі списками можна використовувати такі вбудовані функції Python:

- `len(listObj)` — повертає довжину списку `listObj` (кількість елементів у списку);
- `max(listObj)` — повертає максимальний елемент у списку `listObj`;
- `min(listObj)` — повертає мінімальний елемент у списку `listObj`;
- `sum(listObj)` — повертає суму значень у списку `listObj`;
- `sorted(listObj)` — повертає копію списку `listObj`, у якому елементи впорядковані за зростанням (у разі числових значень) або за алфавітом. Не змінює оригінальний список `listObj`.

Функції `max(listObj)`, `min(listObj)`, `sum(listObj)` застосовуються для списків, що містять числові значення.

Розглянемо застосування перелічених функцій на прикладах:

```
userLogs = ["admin", "student", "teacher", "hr", "user"]
print(len(userLogs)) #5
print(sorted(userLogs)) #['admin', 'hr', 'student',
                        'teacher', 'user']

prices = [100, 250.45, 1200, 20.78]
print(sum(prices)) #1571.23
print(max(prices)) #1200
print(min(prices)) #20.78
print(sorted(prices)) #[20.78, 100, 250.45, 1200]
```

Операції конкатенації (символ «+») та дублювання (символ «*») зі списками працюють так само, як і з рядками:

```
category1 = ["Drama", "Comedy"]
category2 = ["Action", "Fantasy"]
print(category1+category2) #['Drama', 'Comedy',
                           'Action', 'Fantasy']
print(category1*2) #['Drama', 'Comedy', 'Drama',
                   'Comedy']
```

2.7. Методи списків

Оскільки список — це колекція, обробку його елементів можна виконувати в циклах з використанням конструкції `for ... in`, так і з рядками.

Наприклад, виведемо значення кожного елемента списку:

```
category = ["Drama", "Comedy", "Mystery", "Romance"]
for film in category:
    print(film)
```

Тут, на кожному кроці циклу, в змінну `film` потрапляє елемент списку `category`.

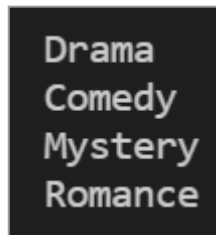


Рисунок 35

Також можна організувати цикл з використанням функції `range()`, використовуючи довжину циклу:

```
category = ["Drama", "Comedy", "Mystery", "Romance"]

for i in range(len(category)):
    print(category[i])
```

У цьому циклі значення змінної *i* пройнуть у діапазоні від 0 до `len(category)-1`, тобто: 0, 1, 2, 3.

Результат буде аналогічний попередньому:

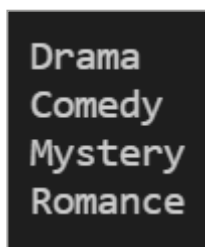


Рисунок 36

Проте, більшість операцій, проведених з елементами списків, здійснюється за допомогою методів об'єкта «список». Розглянемо найбільш популярні та корисні з них.

Спочатку вивчимо методи розширення списку, тобто, як виконувати додавання, вставку елементів, тощо.

Метод `listObj.append(item)` дозволяє додати ще один елемент (аргумент методу, *item*) до кінця списку `listObj`.

```
category1 = ["Drama", "Comedy"]
print(category1) #['Drama', 'Comedy']

category1.append("Fantasy")
print(category1) #['Drama', 'Comedy', 'Fantasy']
```

Результат:

```
['Drama', 'Comedy']
['Drama', 'Comedy', 'Fantasy']
```

Рисунок 37

Метод `listObj.append(iterableItem)` схожий на метод `append()` у тому, що дозволяє нам додавати до списку `listObj`, проте його відмінність у тому, що ми можемо додати відразу кілька елементів, в тому числі і з іншого списку.

```
category1=["Drama", "Comedy"]
category2=['Action', 'Fantasy']
print(category1) #['Drama', 'Comedy']
category1.extend(category2)
print(category1) #['Drama', 'Comedy', 'Action',
                  'Fantasy']
category1.extend(["Mystery", "Romance"])
print(category1) #['Drama', 'Comedy', 'Action',
                  'Fantasy', 'Mystery', 'Romance']
```

Результат:

```
['Drama', 'Comedy']
['Drama', 'Comedy', 'Action', 'Fantasy']
['Drama', 'Comedy', 'Action', 'Fantasy', 'Mystery', 'Romance']
```

Рисунок 38

Метод `listObj.insert(itemIndex, item)` вставляє вказаний елемент `item` у список `listObj` за вказаним індексом `itemIndex`.

```
category1=["Drama", "Comedy"]
print(category1) #['Drama', 'Comedy']
category1.insert(1,"Fantasy")
```

```
print(category1) #['Drama', 'Fantasy', 'Comedy']
category1.insert(2,"Action")
print(category1) #['Drama', 'Fantasy', 'Action', 'Comedy']
category1.insert(-1,"Romance")
print(category1) #['Drama', 'Fantasy', 'Action',
                  'Romance', 'Comedy']
```

Результат:

```
['Drama', 'Comedy']
['Drama', 'Fantasy', 'Comedy']
['Drama', 'Fantasy', 'Action', 'Comedy']
['Drama', 'Fantasy', 'Action', 'Romance', 'Comedy']
```

Рисунок 39

Якщо в якості `itemIndex` вказати `0`, то елемент буде вставлено на початок списку. Якщо вказаний `itemIndex` знаходиться за межами списку, то елемент буде додано до кінця списку.

Тепер розглянемо методи видалення елементів списку.

Метод `listObj.remove(item)` видаляє перше входження вказаного значення (`item`) у списку.

Метод `listObj.pop(itemIndex)` видаляє елемент за вказаним (`itemIndex`) значенням індексу. При використанні даного методу без аргументів буде видалено останній елемент у списку `listObj`.

```
category =["Drama", "Comedy", "Mystery", "Romance",
           "Comedy"]
print(category) #['Drama', 'Comedy', 'Mystery',
                  'Romance', 'Comedy']

category.remove("Comedy")
```

```
print(category) #['Drama', 'Mystery', 'Romance',
                  'Comedy']
category.pop(2)
print(category) #['Drama', 'Mystery', 'Comedy']
category.pop()
print(category) #['Drama', 'Mystery']
```

Результат:

```
['Drama', 'Comedy', 'Mystery', 'Romance', 'Comedy']
['Drama', 'Mystery', 'Romance', 'Comedy']
['Drama', 'Mystery', 'Comedy']
['Drama', 'Mystery']
```

Рисунок 40

Метод `listObj.clear()` видаляє всі елементи зі списку `listObj`.

```
category=["Drama", "Comedy", "Mystery", "Romance",
          "Comedy"]
print(category) #['Drama', 'Comedy', 'Mystery',
                  'Romance', 'Comedy']
category.clear()
print(category) #[]
```

Результат:

```
['Drama', 'Comedy', 'Mystery', 'Romance', 'Comedy']
[]
```

Рисунок 41

Якщо потрібно визначити позицію елемента у списку `listObj` за його значенням (`item`), то використовується метод `listObj.index(item)`.

```
category = ["Drama", "Comedy", "Mystery", "Romance",
            "Comedy"]
print(category.index("Mystery")) #2
print(category.index("Comedy")) #1
```

Примітка: метод `index()` повертає лише перше входження елемента.

Якщо необхідно визначити, скільки разів певне значення `item` зустрічається у списку `listObj`, то використовуємо метод `listObj.count(item)`.

```
category = ["Drama", "Comedy", "Mystery", "Romance",
            "Comedy"]
print(category.count("Comedy")) #2
```

Раніше ми вже розглядали вбудовану функцію `sorted()`, яка повертала відсортовану копію списку. Також є метод списку `listObj.sort(reverse=False)` з аналогічною функціональністю, який за замовчуванням сортує список за зростанням (бо у параметра `reverse` значення за промовчанням `False`). Якщо необхідно змінити напрямок сортування (сортувати за спаданням), слід встановити `reverse= True`.

Метод `listObj.reverse()` змінює порядок сортування елементів на зворотній.

```
category=["Drama", "Comedy", "Mystery", "Romance",
          "Comedy"]

category.sort()
print(category) #['Comedy', 'Comedy', 'Drama',
                'Mystery', 'Romance']
```



```
category.sort(reverse=True)
print(category) #['Romance', 'Mystery', 'Drama',
                'Comedy', 'Comedy']

prices=[100, 250.45, 1200, 20.78]
prices.sort()
print(prices) # [20.78, 100, 250.45, 1200]

prices.sort(reverse=True)
print(prices) # [1200, 250.45, 100, 20.78]

prices.reverse()
print(prices) # [20.78, 100, 250.45, 1200]
```

Результат:

```
['Comedy', 'Comedy', 'Drama', 'Mystery', 'Romance']
['Romance', 'Mystery', 'Drama', 'Comedy', 'Comedy']
[20.78, 100, 250.45, 1200]
[1200, 250.45, 100, 20.78]
[20.78, 100, 250.45, 1200]
```

Рисунок 42

2.8. Оператор приналежності in

Досить часто під час роботи зі списками трапляється завдання перевірити належність певного елемента деякому списку. Наприклад, чи входить до складу клієнтів, які зберігаються у списку `customers = ['Bob', 'Anna', 'Joe', 'Bob', 'Nick']`, клієнт з ім'ям 'Bob'?

Для таких ситуацій у Python є можливість перевірити наявність елемента у списку за допомогою оператора `in`.

Розглянемо на прикладі.

```
customers=['Bob','Anna','Joe','Bob','Nick']
print('Bob' in customers) #True
```

Одне й теж значення може бути у списку більше разу (як клієнт *'Bob'* у прикладі). Доки воно буде в списку хоча б в одному екземплярі, оператор **in** повертатиме значення **True**.

Цей оператор зручний для формування умов:

```
customers=['Bob','Anna','Joe','Bob','Nick']
if ('Bob' in customers):
    print("Bob is our customer")
else:
    print("Sorry")
```

2.9. Особливості списків, посилання та клонування

Однією з найважливіших особливостей списків є створення псевдонімів при операції присвоєння списку іншій змінній.

Псевдоніми — це змінні, які мають різні імена, але містять однакові адреси пам'яті. Ця особливість важлива і її необхідно враховувати, тому що можна випадково, працюючи з однією змінною, зіпсувати значення, що зберігаються в іншій.

Розглянемо ці моменти докладніше на такому прикладі:

```
list1=[1,2,3,4,5]
print(list1) #[1, 2, 3, 4, 5]
list2=list1
```

```
print(list2) #[1, 2, 3, 4, 5]
list2[1]="Hello"
print(list2) #[1, 'Hello', 3, 4, 5]
print(list1) #[1, 'Hello', 3, 4, 5]
```

Спочатку ми маємо список `list1=[1,2,3,4,5]`. Далі ми створюємо новий список `list2` та присвоюємо йому список `list1`. Після цієї операції змінна `list2` містить ту саму адресу пам'яті, як і змінна `list1`, тобто фактично посиляється на той самий список.

Тому, коли ми, використовуючи змінну `list2`, змінили другий елемент списку на слово «*Hello*», то при виведенні списку як через змінну `list2`, так і через змінну `list1`, ми бачимо оновлений список.

Якщо необхідно перевірити, чи посиляються дві різні змінні на той самий список, можна використовувати наступний підхід за допомогою оператора `is`, який перевіряє, чи є дві змінні одним і тим самим об'єктом:

```
list1=[1,2,3,4,5]
list2=list1
list3=[6,7,8]

print(list2 is list1) #True
print(list3 is list1) #False
```

Якщо ж нам необхідно скопіювати елементи з існуючого списку в новий (тобто створити новий об'єкт з такими ж значеннями), то можна використовувати один з наступних способів:

- використовувати функцію `copy()`;
- використовувати функцію-конструктор списку `list()`;

- використовувати зріз всього списку[:].

```
list1=[1,2,3,4,5]
print(list1)  #[1, 2, 3, 4, 5]

list2=list1.copy()
list2[1]="Hello"
print(list2)  #[1, 'Hello', 3, 4, 5]
print(list1)  #[1, 2, 3, 4, 5]

list3=list(list1)
list3[2]="Hello"
print(list3)  #[1, 2, 'Hello', 4, 5]
print(list1)  #[1, 2, 3, 4, 5]

list4=list1[:]
list4[3]="Hello"
print(list4)  #[1, 2, 3, 'Hello', 5]
print(list1)  #[1, 2, 3, 4, 5]
```

У прикладі ми створили копії списку `list1` трьома вищенаведеними способами, тобто `list2`, `list3`, `list4` — це нові об'єкти, які мають власні значення і не пов'язані зі значеннями списку `list1`. Тому після виконання змін у цих трьох нових списках, вміст списку `list1` залишився незмінним.

Так само, якщо ми внесемо зміни до списку `list1`, це не вплине на вміст списків `list2`, `list3`, `list4`:

```
list1[-1]=0
print(list1)  #[1, 2, 3, 4, 0]
print(list2)  #[1, 'Hello', 3, 4, 5]
print(list3)  #[1, 2, 'Hello', 4, 5]
print(list4)  #[1, 2, 3, 'Hello', 5]
```

2.10. Пошук елемента

Ми вже розглянули оператор `in`, який можна використувати для того, щоб дізнатися, чи є потрібний елемент у списку. Результат такої перевірки є `True` або `False`. А якщо нам потрібно не просто перевірити наявність елемента, а дізнатися, де саме він знаходиться (його позицію в списку)?

Саме для таких ситуацій і призначений метод `listObj.index(item)`, який повертає позицію елемента `item` при його першій появі у списку `listObj`.

```
customers=['Bob', 'Anna', 'Joe', 'Nick']
print(customers.index('Joe')) #2
category=["Drama", "Comedy", "Mystery", "Romance",
          "Comedy"]
print(category.index('Comedy')) #1
```

Якщо нам необхідно вибрати всі елементи, які дорівнюють вказаному, то пошук можна організувати вручну, перебираючи список.

```
customers=['Bob', 'Anna', 'Joe', 'Bob', 'Nick']
for i in range(len(customers)):
    if customers[i]=='Bob':
        print(i)
#0
#3
```

2.11. Матриці

Як ми знаємо, елементами списку можуть бути будь-які типи даних, зокрема інші списки, тобто у списку можуть бути списки, які називаються вкладеними. Подібні

структури називають матрицями. Матриці корисні для зберігання даних, які зазвичай представлені у вигляді таблиць. Наприклад, таблиця з успішністю групи студентів (перший стовпець — студент, а далі — його оцінки з предметів, кожен рядок відповідає одному студенту):

Таблиця 11

Bob	11	8	10	12	12
Jane	12	11	11	11	12
Kate	7	8	9	9	10

Розглянемо особливості роботи з матрицями докладніше. Наприклад, для зберігання даних такої таблиці:

Таблиця 12

111	112	113
221	222	223

нам знадобиться такий список:

```
myTbl = [ [111, 112, 113], [221, 222, 223] ]
```

Таким чином, кожен елемент такого списку має два індекси: порядковий номер вкладеного списку, в який він входить (номер «рядка»), і його порядковий номер усередині цього списку (номер «стовпця»). Як нам відомо, індексація в Python починається з нуля.

Таблиця 13

111	112	113
[0][0]	[0][1]	[0][2]
221	222	223
[1][0]	[1][1]	[1][2]

```
myTbl=[ [111,112,113], [221,222,223]]
print(myTbl[1][1]) #222
print(myTbl[0]) #[111, 112, 113]
```

Для перебору всіх елементів таких списків можна використовувати вкладені цикли:

```
for i in range(2):
    for j in range(3):
        print(myTbl[i][j])
#111
#112
#113
#221
#222
#223
```

або

```
for i in range(len(myTbl)):
    for j in range(len(myTbl[i])):
        print(myTbl[i][j])
```

Для створення матриць також можна використовувати генератори списків:

```
myTbl2 = [[j for j in range(2)] for i in range(3)]
print(myTbl2) #[[0, 1], [0, 1], [0, 1]]
```

Тут цикл по стовпцях `for j in range(2)` є вкладеним у цикл за рядками `for i in range(3)`, а дія, що створює новий елемент списку `j` — розміщується у внутрішньому циклі, тобто повторюється 6 разів.

Вкладений цикл вміщається у власні квадратні дужки. Розглянемо приклади роботи з матрицями.

Припустимо, що ми маємо дані про успішність 3 студентів у групі з 4 предметами. Для кожного студента необхідно знайти максимальну оцінку.

Таблиця 14

Bob	11	8	10	12
Jane	12	11	11	11
Kate	7	8	9	9

Як бачимо, кожному студенту відповідає один рядок таблиці, а дані про його оцінки знаходяться в стовпцях (починаючи з першого до кінця, нульовий стовпець зберігає ім'я студента).

Створимо таку матрицю і виведемо її в рядок (кожен студент з нового рядка):

```
studScores = [['Bob', 11, 8, 10, 12],
               ['Jane', 12, 11, 11, 11],
               ['Kate', 7, 8, 9, 9]]
for student in studScores:
    print(student)
```

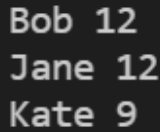
```
['Bob', 11, 8, 10, 12]
['Jane', 12, 11, 11, 11]
['Kate', 7, 8, 9, 9]
```

Рисунок 43

Тепер організуємо аналогічний цикл по рядках (студентах) і в кожному рядку, починаючи з першого стовпця

(за допомогою зрізу), знайдемо максимальну оцінку, використовуючи вбудовану функцію `max()`:

```
for student in studScores:
    print(student[0],max(student[1:]))
```



Bob 12
Jane 12
Kate 9

Рисунок 44

Примітка: функція `max()` знаходить перший максимальний елемент.

Розглянемо ще один приклад. Припустимо, що у нас є дані, до категорії якої належить певний фільм.

```
films=[['Catch Me If You Can', 'Biography', 'Crime', 'Drama'],
        ['Mrs. Doubtfire', 'Comedy', 'Drama', 'Family']]
```

Користувач вводить назву категорії, яка його цікавить, а програма виводить назви фільмів, що належать до цієї категорії:

```
userCategory=input("Input film category: ")

for film in films:
    if userCategory in film[1:]:
        print(film[0])
```

Оскільки в нашому списку лише один фільм відноситься до категорії **Crime**, то результат запиту такий:

```
Input film category: Crime  
Catch Me If You Can
```

Рисунок 45

А на запит категорії **Drama** у нас вже є два фільми:

```
Input film category: Drama  
Catch Me If You Can  
Mrs. Doubtfire
```

Рисунок 46



Урок 3

Рядки, списки

© STEP IT Academy, www.itstep.org

© Анна Егошина

Усі права на фото-, аудіо- і відеотвори, що охороняються авторським правом і фрагменти яких використані в матеріалі, належать їх законним власникам. Фрагменти творів використовуються в ілюстративних цілях в обсязі, виправданому поставленим завданням, у рамках учбового процесу і в учбових цілях, відповідно до законодавства про вільне використання твору без згоди його автора (або іншої особи, яка має авторське право на цей твір). Обсяг і спосіб цитованих творів відповідає прийнятим нормам, не завдає збитку нормальному використанню об'єктів авторського права і не обмежує законні інтереси автора і правовласників. Цитовані фрагменти творів на момент використання не можуть бути замінені альтернативними аналогами, що не охороняються авторським правом, і відповідають критеріям добросовісного використання і чесного використання.

Усі права захищені. Повне або часткове копіювання матеріалів заборонене. Узгодження використання творів або їх фрагментів здійснюється з авторами і правовласниками. Погоджене використання матеріалів можливе тільки якщо вказано джерело.

Відповідальність за несанкціоноване копіювання і комерційне використання матеріалів визначається чинним законодавством.