

## РОЗДІЛ 2. БАЗОВІ ПОНЯТТЯ МОВИ PYTHON

### 2.1. Базовий синтаксис

Мова Python має багато спільного з такими мовами як Perl, C та Java. Однак, є і деякі певні відмінності.

Існує два основних способи запустити програму, написану на мові Python.

Інтерактивний інтерпретатор, який поставляється разом з Python, дає можливість експериментувати з невеликими програмами. Вводячи команди рядок за рядком і миттєво отримуючи результат кожної з них.

Проте, як правило програми містять дуже велику кількість рядків коду, тож їх зберігають у вигляді текстових файлів з розширенням `.py`, а потім запускають.

#### *Інтерактивний інтерпретатор*

Працювати в інтерактивному режимі можна в консолі. Для цього слід виконати команду **python**. Запуститься інтерпретатор, де спочатку виведеться інформація про інтерпретатор. Далі, послідує запрошення до вводу (`>>>`).

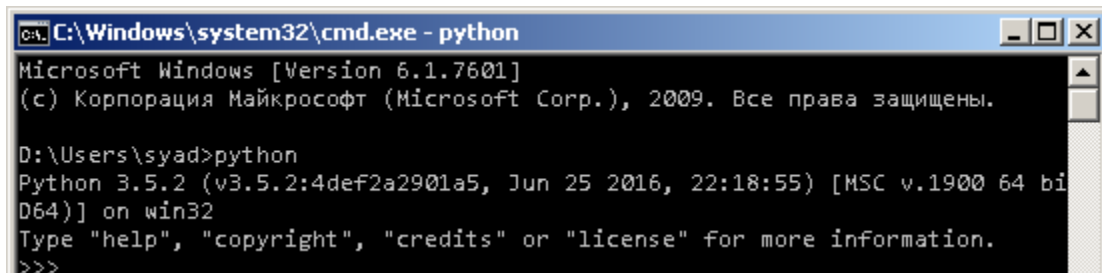
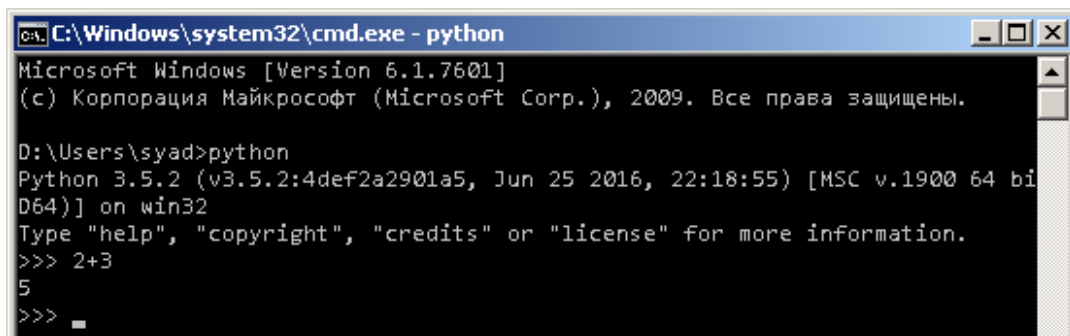


Рис. 2.1. Консоль. Запуск інтерпретатора

Такий режим можна використовувати для вводу інформації або для розрахунків. Вводиться команда та натискається клавіша Enter (завершення вводу команди). Після чого відразу з'являється відповідь.

Автоматичне виведення значення — це особливість інтерактивного інтерпретатора, що економить час.



```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

D:\Users\syad>python
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bi
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> _
```

Рис. 2.2. Консоль. Виконання простих математичних дій

Буває, що в процесі уведення була допущена помилка або потрібно повторити раніше використовувану команду. Щоб не писати рядок спочатку, в консолі можна прокручувати список команд, використовуючи для цього стрілки на клавіатурі.

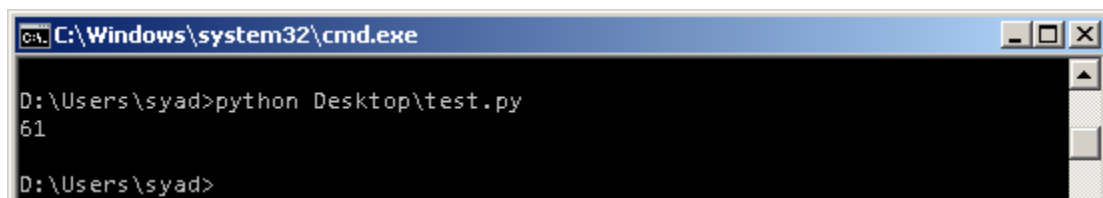
### ***Вихідний програмний код у вигляді файлу***

Незважаючи на зручності інтерактивного режиму роботи при написанні програм на Python, звичайно потрібно зберігати вихідний програмний код для подальшого використання. В такому випадку зберігаються файли, які передаються потім інтерпретатору на виконання. В інтерпретованих мовах програмування, часто вихідний код називають скриптом.

Підготувати скрипти можна у вбудованому середовищі IDLE або будь-якому редакторі редакторі. Крім того, існують спеціальні програми для розробки.

Запускати підготовлені файли можна в IDLE та в консолі за допомогою команди

**python адреса\ім'я\_файлу.py**



```
C:\Windows\system32\cmd.exe
D:\Users\syad>python Desktop\test.py
61
D:\Users\syad>
```

Рис. 2.3. Результат виконання команди, збереженої у файлі, в консолі

## 2.2. Лексеми та ідентифікатори

**Ідентифікатор** Python – це ім'я, яке використовується для ідентифікації змінної, функції, класу, модуля або іншого об'єкту.

Ідентифікатор може містити тільки такі символи:

- літери в нижньому регістрі (від "a" до "z");
- літери у верхньому регістрі (від "A" до "Z") (Python є регістро-чутливою мовою програмування);
- цифри (від 0 до 9);
- нижнє підкреслення (\_);
- не можуть співпадати з зарезервованими словами (табл. 2.2);

Ідентифікатор не може починатися з цифри.

Таким чином, *Al* та *al* – два різних ідентифікатори в Python.

Коректними є такі імена: *a*; *al*; *a\_b\_c\_\_95*; *\_abc*; *\_1a*.

Наступні імена є некоректними: *l*; *1a*; *l\_*.

Як правило великими літерами в Python позначаються константи.

Таблиця 2.1. Зарезервовані слова Python

false	class	finally	is	return
none	continue	for	lambda	try
true	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	Pass	
break	except	in	raise	

### **Відступи**

Код написаний мовою програмування Python не потребує фігурних дужок для позначення блоків коду для визначення класів та функцій або регулювання потоку. Блоки коду відокремлюються за допомогою рядкового відступу, якого необхідно жорстко дотримуватися.

Кількість пробілів у відступах є змінною, але всі оператори в блоці повинні бути відокремлені відступими однакової розмірності.

Наприклад:

```
if True:
    print ("True")
else:
    print ("False")
```

Однак, наступний блок коду згенерує помилку:

```
if True:
    print ("Answer")
    print ("True")
else:
    print ("Answer")
    print ("False")
```

### ***Багаторядкові речення***

Python приймає одиничні ('), подвійні (") і потрійні (' " або ") лапки, щоб позначити строкові літерали. Необхідною умовою є те що рядок має починатися і закінчуватися одним типом лапок. Однак, всередині рядка можна використовувати інші види лапок.

Потрійні лапки використовуються для розбиття рядка на кілька рядків. Наприклад:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

### ***Коментарі в Python***

По мірі збільшення розмірів програм рано чи пізно код стане складніше читати. Для підвищення зрозумілості коду, його корисно доповнювати коментарями на природній мові, і більшість мов програмування, у тому числі Python, надають таку можливість.

Коментування коду вважається правилом "хорошого тону". Коли над програмою працює один програміст, то відсутність коментарів компенсується хорошим знанням коду, але при роботі в команді, за рідкісними винятками, коментарі просто необхідні.

**Коментар** – це фрагмент тексту у програмі, який буде проігнорований інтерпретатором Python. Можна використовувати коментарі, щоб дати пояснення до коду, зробити якісь позначки для себе, або для чогось ще. Коментар позначається символом #; все, що знаходиться після # до кінця поточного рядка, є коментарем. Зазвичай коментар розташовується на окремому рядку:

```
# Підрахунок процентного співвідношення двох величин: 20  
та 80  
print (100 * 20 / 80, "%")
```

Отримаємо:

```
25.0 %
```

Або на тому самому рядку, що і код, який потрібно пояснити:

```
print (100 * 20 / 80, "%")          # Підрахунок процентного  
співвідношення двох величин: 20 та 80
```

Отримаємо:

```
25.0 %
```

Символ # має багато імен: хеш, шарп, фунт або октоторп.

Коментар діє тільки до кінця рядка, на якому він розташовується.

Однак якщо символ # знаходиться всередині текстового рядка, він стає простим символом #:

```
print("Без коментарів #")
```

Отримаємо:

```
Без коментарів #
```

### **Уведення даних**

Уведення даних з клавіатури в програму (починаючи з версії Python 3.0) здійснюється за допомогою функції **input()**. Якщо ця функція виконується, то потік виконання програми зупиняється в очікуванні даних, які користувач повинен ввести за допомогою клавіатури. Після уведення даних і натискання *Enter*, функція

*input()* завершує своє виконання і повертає результат, який є рядком символів, введених користувачем.

```
>>> input ()
1234
'1234'
>>> input ()
Hello World!
'Hello World!'
```

Коли програма, що виконується пропонує користувачеві щось ввести, то користувач може не зрозуміти, що від нього хочуть. Треба якось повідомити, введення яких саме даних очікує програма. З цією метою функція *input()* може приймати необов'язковий аргумент-запрошення строкового типу; при виконанні функції повідомлення буде з'являтися на екрані і інформувати користувача про запитовані дані.

```
>>> input("Введіть значення змінної:")
Введіть значення змінної: 25
'25'
```

Дані повертаються у вигляді рядка, навіть якщо було введено число. Якщо потрібно отримати число, то результат виконання функції *input()* змінюють за допомогою функцій *int()* або *float()*.

```
>>> input('Введіть число:')
Введіть число: 10
'10'
>>> int(input('Введіть число:'))
Введіть число: 10
10
>>> float(input('Введіть число:'))
Введіть число: 10
10.0
```

Результат, що повертається функцією *input()*, зазвичай привласнюють змінній для подальшого використання в програмі.

## **Виведення даних**

Функція *print* () має кілька "прихованих" аргументів, які задаються за замовчуванням в момент виклику:

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

**`print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)`**

де *value* – перераховуються об'єкти через кому, що необхідно вивести на екран,

*sep* – рядок-розділювач між рядками. По замовчуванню стоїть пробіл,

*end* – рядок, що розміщено після останнього об'єкту. По замовчуванню – перехід на новий рядок,

Отже, функція *print* () додає пробіл між кожним виведеним об'єктом, а також символ нового рядка в кінці:

```
>>> print(1, 2, 3, 4, 5)
1 2 3 4 5
>>> print(1, 6, 7, 8, 9, sep=':')
1:6:7:8:9
```

## **2.3. Змінні**

Мови програмування також дозволяють визначати **змінні**.

**Змінна** – це не що інше, як зарезервоване місце пам'яті для зберігання значень. Це означає, що при створенні змінної виділяється деякий простір.

Виходячи з типу даних змінної, інтерпретатор виділяє пам'ять і вирішує, що можна зберегти в зарезервованій пам'яті. Тому, призначаючи різні типи даних змінним, можна зберігати цілі числа, десяткові значення або символи в цих змінних.

Змінні в Python – це просто імена, які посилаються на значення в пам'яті комп'ютера. Можна визначити їх для використання в своїй програмі. В Python символ = застосовується для присвоювання значення змінній.

Операнд ліворуч від оператора = ім'я змінної, а операнд справа від оператора = це значення, що зберігається в змінній. Наприклад:

```
var_int = 100                # Цілочисельна змінна
var_float = 1000.0          # Десятковий дріб
var_str = "Hello"           # Рядок

print (var_int)
print (var_float)
print (var_str)
```

Тут *100*, *1000.0* та *"Hello"* – привласнені значення для назв змінних *var\_int*, *var\_float* та *var\_str*, відповідно. Результатом будуть наступні значення:

```
100
1000.0
Hello
```

Присвоєння не копіює значення, воно прикріплює ім'я об'єкта, який містить дані (рис. 2.4).

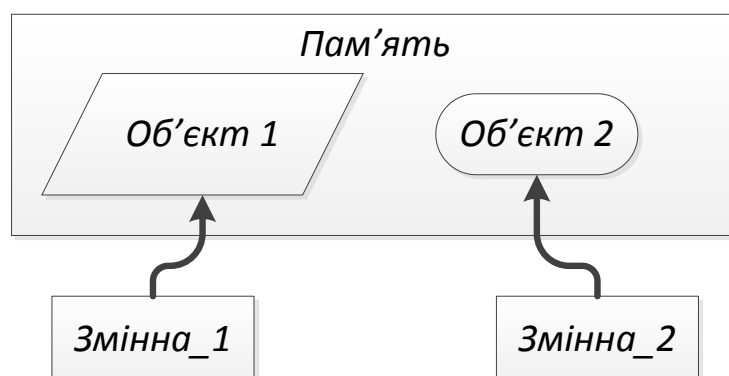


Рис. 2.4. Змінна – як посилання на об'єкт

## 2.4. Типи даних

В статичних мовах необхідно вказувати тип кожної змінної, який визначає, скільки місця змінна займе в пам'яті і що з нею



можна зробити. Комп'ютер використовує цю інформацію, щоб скомпілювати програму в дуже низькорівневу машинну мову. Оголошення типів змінних допомагає комп'ютеру знайти деякі помилки і працювати швидше, але це вимагає попереднього продумування і набору коду. Велика частина мов програмування, наприклад C, C++ і Java, вимагають оголошення типів змінних.

**Тип даних** – множина значень та множина операцій на цих значеннях. Тобто визначає можливі значення та їх сенс, операції над значеннями та способи зберігання.

**Типізація** – операція призначення типу інформаційним сутностям. Для різних мов програмування виділяють різні види типізації: статична/динамічна, сильна/слабка (табл. 2.2).

Таблиця 2.2. Групування мов програмування за типізацією

	Статична	Динамічна
<b>Сильна</b>	C#, Java	Python, Ruby
<b>Слабка</b>	C	JavaScript, PHP

При **статичній типізації** тип даних визначається на етапі компіляції. Змінна не може змінити тип, вони статичні. Ціле число – це ціле число, раз і назавжди.

При **динамічній типізації** – тип змінної визначається при призначенні їй значення. Якщо написати  $x = 5$ , динамічна мова визначить, що 5 – це ціле число, тому змінна  $x$  має тип *int*. Ці мови дозволяють досягти більшого, написавши меншу кількість рядків коду.

Динамічні мови зазвичай повільніше, ніж статичні, але їх швидкість підвищується, оскільки інтерпретатори стають більш оптимізованими. Довгий час динамічні мови використовувалися для коротких програм (сценаріїв), які часто призначалися для того, щоб підготувати дані для оброблення більш довгими програмами, написаними на статичних мовах.

**Сильна типізація** не допускає виконання операцій при несумісності типів.

**Слабка типізація** допускає виконання операцій при несумісності типів, в результаті чого можна отримати непередбачуваний результат.

Відносний лаконізм мови Python дозволяє створити програму, яка буде набагато коротшою свого аналога, написаного на статичній мові.

В Python все (цілі числа, числа з плаваючою точкою, булеві значення, рядки і різні інші структури даних, функції і програми) реалізовано як **об'єкт**. Це дозволяє Python бути стабільним, чого не вистачає деяким іншим мовам.

Python є сильно типізованою мовою – тип об'єкта не зміниться, навіть якщо можна змінити його значення.



Рис. 2.5. Типи даних

## 2.5. Прості типи даних. Числа

Числа бувають різними: **цілими, дробовими, комплексними**. Вони можуть мати величезне значення або дуже довгу дробову частину:

- **цілі числа (*int*)** – додатні і від'ємні цілі числа, а також 0 (наприклад, 4, 687, -45, 0).

- **числа з плаваючою точкою (*float*)** – дробові числа (наприклад, 1.45, -3.789654, 0.00453). Роздільником цілої і дробової частини служить точка.

- **комплексні числа (*complex*)** – зберігає пару значень типу *float*, одне з яких представляє дійсну частину комплексного числа, а інше – уявну (наприклад, 1+2j, -5+10j, 0.44+0.08j)

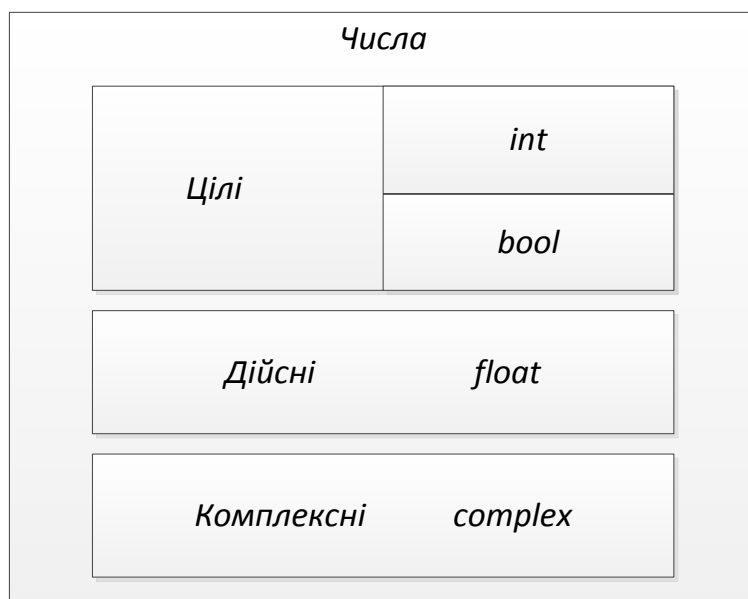


Рис. 2.6. Числові типи даних

### Операції над числами

Відомо, що **операція** – це виконання якихось дій над даними (операндами). Для виконання конкретних дій потрібні спеціальні інструменти – **оператори**, що наведені у таблиці 2.3.

Таблиця 2.3. Оператори, що застосовуються у мові програмування Python

Оператор	Опис	Приклад	Результат
+	Додавання	$5 + 8$	13
-	Віднімання	$90 - 10$	80
*	Множення	$4 * 7$	28
/	Ділення з плаваючою точкою	$7/2$	3,5
//	Цілочисельне ділення (Truncating)	$7//2$	3
%	Залишок	$7\%3$	1
**	Піднесення до степеню	$3^4$	81

Нехай змінна  $a$  має значення 20, а змінна  $b$  – значення 10:

```

a = 20
b = 10

```

```
c = a + b  
print ("1. Значення c = ", c)
```

```
c = a - b  
print ("2. Значення c = ", c)
```

```
c = a * b  
print ("3. Значення c = ", c)
```

```
c = a / b  
print ("4. Значення c = ", c)
```

```
c = a % b  
print ("5. Значення c = ", c)
```

```
c = a**b  
print ("6. Значення c = ", c)
```

```
c = a//b  
print ("7. Значення c = ", c)
```

Операція ділення поділяється на два підвиди:

- за допомогою оператора `/` виконується ділення з плаваючою точкою (десятькове ділення);
- за допомогою оператора `//` виконується цілочисельне ділення (ділення із залишком).

Якщо ділити ціле число на ціле число, оператор `/` дасть результат з плаваючою точкою.

```
1. Значення c = 30  
2. Значення c = 10  
3. Значення c = 200  
4. Значення c = 2.0  
5. Значення c = 0  
6. Значення c = 10240000000000  
7. Значення c = 2
```

Можна працювати з довільною кількістю операндів та операторів.

Вставляти пробіл між кожним числом і оператором не обов'язково. Зайві пробіли погіршують сприйняття коду:

```
print (5+4  + 7)
```

Результатом запуску даного коду буде:

```
17
```

Ділення на нуль за допомогою будь-якого оператора згенерує виняток:

```
print (a//0)
```

Результатом запуску даного коду буде:

```
Traceback (most recent call last):  
  File "G:\lab.py", line 1, in <module>  
    print (a//0)  
NameError: name 'a' is not defined
```

Якщо в одному і тому ж виразі зустрічаються декілька типів, то результат має самий сильний тип зі всіх чисел що у виразі. Самим сильним типом вважається комплексний, за ним йде дійсний, а потім цілий тип даних.

Якщо змішати чисельні значення, Python буде намагатися автоматично перетворити їх:

```
a = 20  
b = 10.8  
  
c = a + b  
print ("1. Значення c = ", c)
```

Результатом запуску даного коду буде:

```
1. Значення c = 30.8
```

Булеве значення *False* розглядається як 0 або 0.0, коли воно змішується з цілими числами або числами з плаваючою точкою, а *True* – як 1 або 1.0:

```
a = 20  
b = 10.8
```

```
c = a + True  
print ("1. Значення c = ", c)
```

```
c = b + False  
print ("2. Значення c = ", c)
```

Результатом запуску даного коду буде:

```
1. Значення c = 21  
2. Значення c = 10.8
```

### ***Оператори присвоєння***

Можна поєднувати арифметичні оператори з привласненням, розміщуючи оператор перед знаком =.

Вираз `a -= 3` аналогічний виразу `a = a - 3`.

Замість знаку – можуть стояти інші оператори: `+`, `*`, `/`, `//`, `%`

```
a = 20  
b = 10
```

```
c = a + b  
print ("1. Значення c = ", c)
```

```
c += a  
print ("2. Значення c = ", c)
```

```
c *= a  
print ("3. Значення c = ", c)
```

```
c /= a  
print ("4. Значення c = ", c)
```

```
c = 2  
c %= a  
print ("5. Значення c = ", c)
```

```
c **= a  
print ("6. Значення c = ", c)
```

```
c //= a  
print ("7. Значення c = ", c)
```

Результатом запуску даного коду буде:

```
1. Значення c = 30  
2. Значення c = 50  
3. Значення c = 1000  
4. Значення c = 50.0  
5. Значення c = 2  
6. Значення c = 1048576  
7. Значення c = 52428
```

### ***Пріоритет операцій***

Коли вираз містить більше одного оператора, послідовність виконання операцій залежить від порядку їх слідування у виразі, а також від їх пріоритету. Пріоритети операторів в Python збігаються з пріоритетами математичних операцій.

```
d = 2 + 3 * 4  
  
print ("Значення d =", d)
```

Результатом запуску даного коду буде:

```
14
```

Найвищий пріоритет мають дужки, потім піднесення до степеню, множення та ділення і лише після них додавання, віднімання, далі кон'юнкція, диз'юнкція та все інше.

У випадку рівних пріоритетів розрахунок йде справа наліво. Для зміни цього порядку використовують дужки. Краще їх використовувати у всіх сумнівних ситуаціях:

```
d = (2 + 3) * 4  
print ("Значення d =", d)
```

Результатом запуску даного коду буде:

```
20
```

Це спрощує візуальне сприймання виразу.

## ***Перетворення типів***

Для того щоб змінити одні типи даних на інші використовуються певні стандартні функції.

Так, для зміни чогось на цілочисельний тип, слід використовувати функцію *int()*. Вона зберігає цілу частину числа і відкидає залишок.

```
a = 20
b = 10.8

q = int(a)
print ("1. Значення q = ", q)

q = int(b)
print ("2. Значення q = ", q)
```

Результатом запуску даного коду буде:

```
1. Значення q = 20
2. Значення q = 10
```

Текстовий рядок теж можна змінити на цілочисельний, якщо він буде містити цифрові символи і, можливо, знаки + і -:

```
a = "-15"
q = int(a)

print ("1. Значення q = ", q)
```

Результатом запуску даного коду буде:

```
1. Значення q = -15
```

Якщо перетворити щось несхоже на число – буде згенеровано виняток:

```
a = "xyz"
q = int(a)

print ("1. Значення q = ", q)
```



Результатом запуску даного коду буде:

```
Traceback (most recent call last):  
  File "G:\lab.py", line 3, in <module>  
    q = int(a)  
ValueError: invalid literal for int() with base 10: 'xyz'
```

За допомогою винятків Python сповіщає про те, що сталася помилка, замість того щоб перервати виконання програми, як роблять деякі інші мови.

Останній рядок *ValueError: invalid literal for int() with base 10: 'xyz'* інформує про те, що текстовий рядок починається тими символами, які функція *int()* обробити не може.

Функція *int()* буде створювати цілі числа з чисел з плаваючою точкою або рядків, що складаються з цифр, але вона не буде обробляти рядки, що містять порожній рядок, десяткові точки або експоненти.

Для того щоб перетворити інші типи в тип *float*, слід використовувати функцію *float()*.

Перетворення значення типу *int* в тип *float* лише створить десяткову кому:

```
a = 98  
b = '99'  
  
q = float(a)  
print ("1. Значення q = ", q)  
  
q = float(b)  
print ("2. Значення q = ", q)
```

Результатом запуску даного коду буде:

```
1. Значення q = 98.0  
2. Значення q = 99.0
```

Можна перетворювати рядки, що містять символи, які є коректним числом з плаваючою точкою (цифри, знаки, десяткова кома чи *e*, за якою слідує експонента):

```
a = '98.6'  
b = '-1.5'
```

```
c = '1.0e4'
d = True

q = float(a)
print ("1. Значення q = ", q)

q = float(b)
print ("2. Значення q = ", q)

q = float(c)
print ("3. Значення q = ", q)

q = float(d)
print ("4. Значення q = ", q)
```

Результатом запуску даного коду буде:

```
1. Значення q = 98.6
2. Значення q = -1.5
3. Значення q = 10000.0
4. Значення q = 1.0
```

## 2.6. Прості логічні вирази та логічний тип даних

В усіх мовах програмування високого рівня є можливість розгалуження програми; при цьому виконується одна з гілок програми в залежності від істинності чи хибності умови.

**Логічними виразами** називають вирази, результатом яких є істина (True) або хибність (False). У найпростішому випадку будь-яке твердження може бути істинним або хибним. Наприклад, "2 + 2 дорівнює 4" – істинний вираз, а "2 + 2 дорівнює 5" – хибний.

Розмовляючи на природній мові (наприклад, українській) порівняння позначається словами "рівно", "більше", "менше". У мовах програмування використовуються спеціальні знаки, подібні до тих, які використовуються в математичних виразах: > (більше), < (менше), >= (більше або дорівнює), <= (менше або дорівнює).

В Python використовуються наступні оператори порівняння:

- рівність (`==`);
- нерівність (`!=`);
- менше (`<`);
- менше або дорівнює (`<=`);
- більше (`>`);
- більше або дорівнює (`>=`);
- включення (`in ...`).

Ці оператори повертають булеві значення *True* або *False*.

Для перевірки на рівність використовуються два знака "дорівнює" (`==`) (один знак "дорівнює" застосовується для надання значення змінній).

```
x = 2 + 2
print('1.Результат роботи логічного виразу:', x == 4)

print('2.Результат роботи логічного виразу:', x == 5)

print('3.Результат роботи логічного виразу:', x != 5)
```

Результатом запуску даного коду буде:

```
1.Результат роботи логічного виразу: True
2.Результат роботи логічного виразу: False
3.Результат роботи логічного виразу: True
```

Результат порівняння двох значень можна записати в змінну:

```
y = x == 5
print (y)
```

Отримаємо:

```
False
```

Пріоритет операцій порівняння менший пріоритету арифметичних операцій, але більший, ніж у операції присвоювання. Це означає, що спочатку вираховується результат фрагментів, а потім вони порівнюються.

Значення *False* не обов'язково явно означає *False*. Наприклад, до *False* прирівнюються всі наступні значення:

- булева змінна *False*;

- значення *None*;
- ціле число 0;
- число з плаваючою точкою 0.0;
- порожній рядок (' ');
- порожній список ([]);
- порожній кортеж (());
- порожній словник ({});
- порожня множина (set ()).

Всі інші значення прирівнюються до *True*.

## 2.7. Логічні оператори

Логічні вирази типу  $x \geq y$  є простим. Однак, на практиці не рідко використовуються більш складні. Може знадобитися отримати відповіді "Так" або "Ні" в залежності від результату виконання двох простих виразів. Для об'єднання простих виразів в більш складні використовуються **логічні оператори**: *and*, *or* і *not*.

Значення їх повністю збігаються зі значенням англійських слів, якими вони позначаються.

Щоб отримати істину (*True*) при використанні оператора *and*, необхідно, щоб результати обох простих виразів, які пов'язує цей оператор, були істинними. Якщо хоча б в одному випадку результатом буде *False* (хибність), то і весь складний вираз буде хибним.

Щоб отримати істину (*True*) при використанні оператора *or*, необхідно, щоб результати хоча б одного простого виразу, що входить до складу складного, був істинним. У разі оператора *or* складний вираз стає хибним лише тоді, коли хибні всі складові його прості вирази.

Оператор *not* унарний, тобто він працює тільки з одним операндом.

Результатом застосування логічного оператора *not* (не) відбудеться заперечення операнда, тобто якщо операнд істинний, то *not* поверне – хибність, якщо хибний, то – істину.

```
y = 6 > 8
```

```
print('y =', y)
```

```
print('1.Результат роботи логічного виразу "not y":', not y)
print('2.Результат роботи логічного виразу "not None":', not
None )
print('3.Результат роботи логічного виразу "not 2":', not 2)
```

Результатом запуску даного коду буде:

```
y = False
1.Результат роботи логічного виразу "not y": True
2.Результат роботи логічного виразу "not None": True
3.Результат роботи логічного виразу "not 2": False
```

Логічний оператор *and* поверне *True* або *False*, якщо його операндами є логічні вирази.

```
print(2>4 and 45>3)
```

Отримаємо:

```
False
```

Якщо операндами оператора *and* є об'єкти, то в результаті Python поверне об'єкт:

```
print(" and 2)
```

Отримаємо:

```
"
```

Для обчислення оператора *and* Python обчислює операнди зліва направо і повертає перший об'єкт, який має хибне значення.

```
print(0 and 3)
```

Отримаємо:

```
0
```

Якщо Python не вдається знайти хибний об'єкт-операнд, то він повертає крайній правий операнд.

```
print(5 and 4)
```

Отримаємо:

```
4
```

Логічний оператор *or* діє схожим чином, але для об'єктів-операндів Python повертає перший об'єкт, який має істинне значення. Python припинить подальші обчислення, як тільки буде знайдений перший об'єкт, який має істинне значення.

```
print(2 or 3)
```

Отримаємо:

```
2
```

Таким чином, кінцевий результат стає відомий ще до обчислення решти виразу.

```
print(None or 5)      # Повертає другий об'єкт, тому що  
                        перший завжди хибний  
print(None or 0)      # Повертає об'єкт, що залишився
```

Отримаємо:

```
5  
0
```

Логічні вирази можна комбінувати:

```
>>> 1+3 > 7  
False  
>>> 1+(3>7)  
1
```

В Python можна перевіряти приналежність інтервалу:

```
x=0  
print(-5<x<10)          # Еквівалентно: x > -5 and x<10
```

Отримаємо:

```
True
```

Рядки в Python теж можна порівнювати по аналогії з числами. Символи, як і все інше, представлено в комп'ютері у вигляді чисел. Є спеціальна таблиця, яка ставить у відповідність кожному символу деяке число 20. Визначити, яке число відповідає символу можна за допомогою функції *ord()*:

```
q=ord('L')
```

```
print ("1. Значення 'L' =", q)
```

```
q=ord ('Ф')
```

```
print ("2. Значення 'Ф' =", q)
```

```
q=ord ('A')
```

```
print ("3. Значення 'A' =", q)
```

```
q=ord ('a')
```

```
print ("4. Значення 'a' =", q)
```

Результатом запуску даного коду буде:

```
1. Значення 'L' = 76
```

```
2. Значення 'Ф' = 1060
```

```
3. Значення 'A' = 65
```

```
4. Значення 'a' = 97
```

Порівняння символів зводиться до порівняння чисел, які їм відповідають.

```
q='A' > 'L'
```

```
print ("Порівняння 'A' > 'L' =", q)
```

Отримаємо:

```
Порівняння 'A' > 'L' = False
```

Для порівняння рядків Python їх порівнює посимвольно:

```
q='Aa' > 'Ll'
```

```
print ("Порівняння 'Aa' > 'Ll' =", q)
```

Отримаємо:

```
Порівняння 'Aa' > 'Ll' = False
```

Оператор *in* перевіряє входження підрядка в рядок:

```
q='a' in 'abc'
```

```
print("Результат входження 'a' in 'abc' -", q)
```

```
q='A' in 'abc'
```

```
print("Результат входження 'A' in 'abc' -", q)
```

# Великої літери A немає в рядку 'abc'

```
q="" in 'abc'          # Порожній рядок міститься в будь-
якому рядку
print("Результат входження " in 'abc' "-", q)

q=" in "
print("Результат входження " in " "-", q)
```

Отримаємо:

```
Результат входження 'a' in 'abc' - True
Результат входження 'A' in 'abc' - False
Результат входження " in 'abc' - True
Результат входження " in " - True
```

## 2.8. Складні структури даних. Рядки

Завдяки підтримці стандарту Unicode Python 3 може містити символи будь-якої мови світу, а також багато інших символів. Необхідність роботи з цим стандартом була однією з причин зміни Python 2. Іноді використовуються рядки формату ASCII.

Рядки представляють собою послідовності символів.

На відміну від інших мов, в Python рядки є незмінними. Не можна змінити сам рядок, але можна скопіювати частини рядків в інший рядок, щоб отримати той же ефект.

Рядок в Python створюється заключенням символів в одинарні або подвійні лапки.

```
a = 'Hello'
b = "Hi"

print ("1. Значення a:", a)
print ("2. Значення b:", b)
```

Результатом запуску даного коду буде:

```
1. Значення a: Hello
2. Значення b: Hi
```

Два види лапок дозволяють створювати рядки, що містять лапки. У середині одинарних лапок можна розташувати подвійні і навпаки:



```
a = "використаємо апостроф у слові подвір'я"
```

```
print ("Значення a:", a)
```

Результатом запуску даного коду буде:

```
Значення a: використаємо апостроф у слові подвір'я
```

Будь-яка програма стає більш зрозумілою, якщо її рядки відносно короткі. Рекомендована (але не обов'язкова) максимальна довжина рядка дорівнює 80 символам. Якщо є необхідність надрукувати рядок що містить більше 80 символів, використовують символ відновлення `\`, що розміщується в кінці рядка, і далі Python буде діяти так, ніби це все той же рядок.

```
alphabet = 'abcdefg' \
           'hijklmnop' \
           'qrstuv' \
           'wxyz'
```

```
print ("Значення alphabet:", alphabet)
```

Результатом запуску даного коду буде:

```
Значення alphabet: abcdefghijklmnopqrstuvwxyz
```

### ***Створення керуючих символів***

Крім цього зворотний слеш (`\`) дозволяє створювати керуючі послідовності всередині рядків. Найбільш поширена послідовність `\n`, яка означає перехід на новий рядок. З її допомогою можна створити багаторядкові рядки з однорядкових:

```
alphabet = "\nabcdefg\nhijklmnop\nqrstuv\nwxyz"
```

```
print ("Значення alphabet:", alphabet)
```

Результатом запуску даного коду буде:

```
Значення alphabet:
abcdefg
hijklmnop
qrstuv
wxyz
```

Найбільш уживаними є:

`\t` – знак табуляції  
`\\` – похила риса вліво  
`\'` – символ одинарних лапок  
`\''` – символ подвійних лапок

```
print('\tabc')
print('a\tbc')
print('ab\tc')
```

Отримаємо:

```
      abc
a      bc
ab     c
```

Якщо потрібен зворотний слеш, необхідно надрукувати два:

```
print('abc\\')
```

Отримаємо:

```
abc\
```

### ***Перетворення типів***

Подібно функціям `int()` та `float()`, можна перетворювати інші типи даних Python в рядки за допомогою функції `str()`:

```
a = 98.6
b = -1.5
c = 1.0e4
d = True

q = str(a)
print ("1. Значення q = ", q)

q = str(b)
print ("2. Значення q = ", q)

q = str(c)
print ("3. Значення q = ", q)
```

```
q = str(d)
print ("3. Значення q = ", q)
```

Результатом запуску даного коду буде:

```
1. Значення q = 98.6
2. Значення q = -1.5
3. Значення q = 10000.0
3. Значення q = True
```

### ***Об'єднання рядків***

Можна об'єднувати рядки або рядкові змінні в Python за допомогою оператора +, або розташувавши їх послідовно один за одним:

```
q='ab'+ 'cd'
q='ab"cd'

print("1. Результат виконання 'ab'+ 'cd':", q)
print("2. Результат виконання 'ab"cd':", q)
```

Отримаємо:

```
1. Результат виконання 'ab'+ 'cd': abcd
2. Результат виконання 'ab'+ 'cd': abcd
```

Python не додає пробілів при конкатенації рядків, тому потрібно явно додати пробіли:

```
a = 'Python'
b = "'s"
c = ' philosophy'
q = a + b + c

print("Результат виконання 'a + b + c':", q)
```

Отримаємо:

```
Результат виконання 'a + b + c': Python's philosophy
```

### ***Розмноження рядків***

Оператор `*` можна використовувати для того, щоб розмножити рядок.

```
a='abc'  
q = 3*a  
  
print("Результат виконання '3*a:", q)
```

Отримаємо:

```
Результат виконання '3*a: abcabcabc
```

### *Звернення до символу*

Для того щоб отримати один символ рядка, задається зміщення всередині квадратних дужок після імені рядка. Зсув першого (крайнього зліва) символу дорівнює 0, наступного – 1 і т.д. Зсув останнього (крайнього праворуч) символу може бути виражено як -1, тому не потрібно рахувати, в такому випадку зміщення подальших символів дорівнюватиме -2, -3 і т.д.:

```
string= 'abcdefghijklmnopqrstuvwxy'  
  
print (string)           # Виведення всього рядку  
print (string [0])       # Виведення першого символу рядку  
print (string [1])       # Виведення другого символу рядку  
print (string [-1])      # Виведення останнього символу рядку
```

Результатом запуску даного коду буде:

```
abcdefghijklmnopqrstuvwxy  
a  
cde  
cdefghijklmnopqrstuvwxy
```

Якщо вказати зміщення, яке дорівнює довжині рядка або більше (зміщення повинно лежати в діапазоні від 0 до довжини рядка -1), буде згенеровано виняток:

```
string= 'abcdefghijklmnopqrstuvwxy'  
print (string [100])
```

Отримаємо:

```
Traceback (most recent call last):
```

```
File "G:\lab.py", line 8, in <module>
    print (string [100])
IndexError: string index out of range
```

Оскільки рядки незмінні – не можна вставити символ безпосередньо в рядок або змінити символ за заданим індексом.

```
string= 'abcdefghijklmnopqrstuvwxyz'
string [1] = 'ello'
```

Отримаємо:

```
Traceback (most recent call last):
  File "G:\lab.py", line 10, in <module>
    string[1] = 'ello'
TypeError: 'str' object does not support item assignment
```

З рядка можна вилучати підрядок (частину рядка) за допомогою функції *slice*. Визначається *slice* за допомогою квадратних дужок, зміщення початку підрядка *start* і кінця підрядка *end*, а також розміру кроку *step*.

**[start: end: step]**

Деякі з цих параметрів можуть бути відсутні. У підрядок будуть включені символи, розташовані починаючи з точки, на яку вказує зміщення *start*, і закінчуючи точкою, на яку вказує зміщення *end*.

- Оператор **[:]** дозволяє взяти зріз всієї послідовності від початку до кінця.

- Оператор **[start:]** дозволяє взяти зріз послідовності з точки, на яку вказує зміщення *start*, до кінця.

- Оператор **[: end]** дозволяє взяти зріз послідовності від початку до точки, на яку вказує зміщення *end* - 1.

- Оператор **[start: end]** дозволяє взяти зріз послідовності з точки, на яку вказує зміщення *start*, до точки, на яку вказує зміщення *end* - 1.

- Оператор **[start: end: step]** дозволяє взяти зріз послідовності з точки, на яку вказує зміщення *start*, до точки, на

яку вказує зміщення *end* мінус 1, опускаючи символи, чие зміщення всередині підрядка кратне *step*.

Зміщення зліва направо визначається як 0, 1 і т.д., а справа наліво – як -1, -2 і т.д. Якщо не вказати *start*, функція буде використовувати в якості його значення 0 (початок рядку). Якщо не вказати *end*, функція буде використовувати кінець рядка. Python не включає символ, розташований під номером, який вказаний останнім.

```
string= 'abcdefghijklmnopqrstuvwxyz'

print (string [2:5])      # Виведення символів починаючи з 3-го
                           # до 5-го
print (string [2:])       # Виведення рядку починаючи з 3-го
                           # символу
print (string [:])        # Вся послідовність від початку до кінця
print (string [20:])      # Всі символи, починаючи з 20-го і до
                           # кінця
print (string [-3:])      # Останні три символи
print (string [18:-3])    # Починаючи з 18-го і закінчуючи 4 з
                           # кінця
print (string [-6:-2])    # Закінчуючи 3 з кінця
```

Результатом запуску даного коду буде:

```
cde
cdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
uvwxyz
xyz
stuvw
uvwx
```

Щоб збільшити крок, необхідно вказати його після другої двокрапки.

```
string= 'abcdefghijklmnopqrstuvwxyz'
print (string [::7])      # Кожен сьомий символ з початку
                           # до кінця
```

<code>print (string [4:20:3])</code>	# Кожен 3 символ, починаючи з 4 та закінчуючи 19-м
<code>print (string [19::4])</code>	# Кожен 4 символ, починаючи з 19-го:
<code>print (string [:21:5])</code>	# Кожен п'ятий символ від початку до 20-го:

Отримаємо:

```
ahov
ehknqt
tx
afkpu
```

Значення *end* має бути на одиницю більше, ніж реальне зміщення.

Якщо задати від'ємний крок, Python буде рухатися у зворотний бік.

<code>string= 'abcdefghijklmnopqrstuvwxy'</code>	
<code>print (string [-1::-1])</code>	# Всі символи, починаючи з кінця і закінчуючи на початку
<code>print (string [::-1])</code>	# Аналогічно

Отримаємо:

```
zyxwvutsrqponmlkjihgfedcba
zyxwvutsrqponmlkjihgfedcba
```

### ***Строкові методи та функції***

Функція *len()* підраховує символи в рядку:

<code>string= 'abcdefghijklmnopqrstuvwxy'</code>	
<code>q=len(string)</code>	
<code>print ('Довжина рядка string =', q)</code>	

Отримаємо:

```
Довжина рядка string = 26
```

Довжина порожнього рядка = 0. Функцію *len()* можна застосовувати до інших послідовностей (кортежі, словники, списки).

На відміну від функції *len()* деякі функції характерні лише для рядків.

Для того щоб використовувати строкову функцію, необхідно ввести ім'я рядка, крапку, ім'я функції і аргументи, які потрібні функції:

### **рядок.функція(аргументи)**

Однією з таких вбудованих функцій є функція *split()*, що розбиває рядок на список невеликих рядків, спираючись на роздільник.

### **рядок.split('роздільник')**

**Список** – це послідовність значень, розділених комами і оточених квадратними дужками:

```
string= 'abcdefghijklmnopqrstuvwxy'
q = string.split(',')
print (q)

q = string.split('k')
print (q)
```

Результатом запуску даного коду буде:

```
['a', ' b', ' c', ' d', ' e']
['abcdefghijklmnopqrstuvwxy']
['abcdefghij', 'lmnopqrstuvwxy']
```

Рядок має ім'я *letters*, а строкова функція називається *split()* і отримує один аргумент *' '*. Якщо не вказати роздільник, функція *split()* буде використовувати будь-яку послідовність пробілів, а також символи нового рядка і табуляцію:

```
letters = 'a, b, c, d, e'
q = letters.split()
```



```
print (q)
```

Отримаємо:

```
['a,', 'b,', 'c,', 'd,', 'e']
```

Але в будь-якому випадку навіть при виклику функції *split()* без аргументів, все одно потрібно додавати круглі дужки – саме так Python дізнається, що викликається функція.

### **Об'єднання рядків за допомогою функції *join()***

Функція *join()* є протилежністю функції *split()*: об'єднує список рядків в один рядок.

Для виклику функції спочатку вказується рядок, який об'єднує інші, а потім – список рядків для об'єднання:

#### **рядок.join(список)**

Для того щоб об'єднати список рядків *lines*, розділивши їх символами нового рядка, потрібно написати *'\n'.join(lines)*.

```
q = ['abcdefgh', 'ijklmnopqr', 'stuvwxyz']
string = ''.join(q)           # Об'єднання 3 послідовностей
                               літер без розділення

print(string)
string = ', '.join(q)         # Об'єднання 3 послідовностей
                               літер з розділенням їх комами та
                               пробілом

print(string)
```

Результатом запуску даного коду буде:

```
abcdefghijklmnopqrstuvwxyz
abcdefgh, ijklmnopqr, stuvwxyz
```

### **Регістр і вирівнювання**

В Python є велика множина строкових методів (можуть бути використані з будь-яким об'єктом *str*) і модуль *string*, що містить корисні визначення.

```
string = "abcdefghijklmnopqrstuvwxyz"
string = ".abcdefghi jklmnopqrs tuvxyz..."
```

<code>q = string.strip('.') print(q)</code>	# Видалення символів '.' з обох кінців рядка:
<code>q = string.capitalize() print(q)</code>	# Перше слово з великої літери
<code>q = string.title() print(q)</code>	# Всі слова з великої літери
<code>q = string.upper() print(q)</code>	# Всі слова великими літерами
<code>q = string.lower() print(q)</code>	# Всі слова маленькими літерами
<code>q = string.swapcase() print(q)</code>	# Зміна регістру літер

Результатом запуску даного коду буде:

```

abcdefghi jklmnopqrs tuvwxyz
.abcdefghi jklmnopqrs tuvwxyz...
.Abcdefghi Jklmnopqrs Tuvwxyz...
.ABCDEFGHI JKLMNOPQRS TUVWXYZ...
.abcdefghi jklmnopqrs tuvwxyz...
.ABCDEFGHI JKLMNOPQRS TUVWXYZ...

```

### ***Форматування рядків***

Python дозволяє виконати вирівнювання рядків [5, 8].

<code>string = ".abcdefghi jklmnopqrs tuvwxyz..."</code>	
<code>q=string.center(60)</code>	# Рядок вирівнюється всередині заданої кількості пробілів (30) по центру
<code>print(q)</code>	
<code>q=string.ljust(60)</code>	# Рядок вирівнюється по лівому краю
<code>print(q)</code>	

```
q=string.rjust(60)           # Рядок вирівнюється по
                              правому краю
print(q)
```

Результатом запуску даного коду буде:

```
.abcdefghijklmnpqrs tuvwxyz...
.abcdefghijklmnpqrs tuvwxyz...
.abcdefghijklmnpqrs tuvwxyz...
```

Python дозволяє *інтерполювати дані в рядки* – розмістити значення всередині рядків, – застосовуючи різні формати. Можна використовувати цю можливість, щоб створювати звіти та інші документи, яким необхідно зробити певний зовнішній вигляд.

Python пропонує два способи форматування рядків.

### **Форматування рядків з використанням символу %**

Форматування рядків з використанням символу % має форму:

**рядок % дані**

Усередині рядка знаходяться інтерполяційні послідовності. У табл. 2.5 показано, що найпростіша послідовність – це символ %, за яким слідує буква, що представляє тип даних, який повинен бути відформатований.

Таблиця 2.5. Типи перетворення

%s	Рядок
%d	Ціле число в десятковій системі числення
%x	Ціле число в шістнадцятковій системі числення
%o	Ціле число в вісімковій системі числення
%f	Число з плаваючою крапкою в десятковій системі числення
%e	Число з плаваючою крапкою в шістнадцятковій системі числення
%g	Число з плаваючою крапкою у вісімковій системі числення
%%	Символ %

Послідовність %s всередині рядка означає, що в неї потрібно інтерполювати рядок. Кількість використаних символів %

повинно збігатися з кількістю об'єктів, які розташовуються після %.

Ціле число:

```
q = '%s' % 42  
print(q)
```

```
q = '%d' % 42  
print(q)
```

```
q = '%x' % 42  
print(q)
```

```
q = '%o' % 42  
print(q)
```

Отримаємо:

```
42  
42  
2a  
52
```

Число з плаваючою крапкою:

```
q = '%s' % 7.03  
print(q)
```

```
q = '%f' % 7.03  
print(q)
```

```
q = '%e' % 7.03  
print(q)
```

```
q = '%g' % 7.03  
print(q)
```

Отримаємо:

```
7.03  
7.030000  
7.030000e+00
```

### 7.03

Ціле число і символ %:

```
q = '%d%%' % 100  
print(q)
```

Отримаємо:

```
100%
```

Інтерполяція деяких рядків і цілих чисел:

```
breed = 'British Shorthair'  
cat = 'Lola'  
weight = 4  
  
q = "My favorite cat breed is %s" % breed  
print(q)  
  
q = "My cat %s weighs %s kg" % (cat, weight)  
print(q)
```

Результатом запуску даного коду буде:

```
My favorite cat breed is British Shorthair  
My cat Lola weighs 4 kg
```

Один об'єкт на зразок *breed* розташовується відразу після символу %. Якщо таких об'єктів кілька, вони повинні бути згруповані в кортеж (потрібно оточити їх дужками і розділити комами) на зразок (*cat, weight*).

Незважаючи на те що змінна *weight* цілочисельна, послідовність *%s* всередині рядка перетворює її в рядок.

Можна додати інші значення між % і визначенням типу, щоб вказати мінімальну і максимальну ширину, вирівнювання і заповнення символами.

```
n = 42  
f = 7.03  
s = 'string'  
  
q = '%d %f %s' % (n, f, s)
```

```
print(q)
```

Отримаємо:

```
'42 7.030000 string'
```

Можна встановити мінімальну довжину поля для кожної змінної і вирівняти їх по правому (лівому) краю, заповнюючи невикористане місце пробілами:

```
n = 42
f = 7.03
s = 'string'
q = '%10d %10f %10s' % (n, f, s)    # Вирівнювання по
                                     # правому краю,
                                     # мінімальна довжина
                                     # поля = 10 знаків

print(q)

q = '%-10d %-10f %-10s' % (n, f, s)  # Вирівнювання по
                                     # лівому краю, мінімальна
                                     # довжина поля = 10
                                     # знаків

print(q)
```

Отримаємо:

```
    42  7.030000  string
42      7.030000  string
```

Можна вказати довжину поля та максимальну кількість символів (вирівнювання по правому краю). Таке налаштування обрізає рядок і обмежує число з плаваючою точкою чотирма цифрами після десяткової коми:

```
n = 42
f = 7.03
s = 'string'

q = '%10.4d %10.4f %10.4s' % (n, f, s)

print(q)
```

```
q = '%.4d %.4f %.4s' % (n, f, s)
print(q)
```

Отримаємо:

```
0042  7.0300  stri
0042 7.0300 stri
```

### ***Форматування за допомогою символів {} і функції format***

В Python 3 рекомендується застосовувати новий стиль форматування за допомогою методу *format()*, що має наступний синтаксис:

**рядок\_спеціального\_формату.format(\*args, \*\*kwargs)**

У параметрі *рядок\_спеціального\_формату* всередині символів {} можуть бути вказані деякі специфікатори.

Всі символи, розташовані поза фігурних дужок, виводяться без перетворень. Якщо всередині рядка необхідно використовувати символи {}, то ці символи слід подвоїти, інакше збуджується виняток *ValueError*.

```
n = 42
f = 7.03
s = 'string'
q = '{} {} {}'.format(n, f, s)

print(q)
```

Отримаємо:

```
42 7.03 string
```

Аргументи старого стилю потрібно надавати в порядку появи їх наповнювачів з символами % в оригінальній рядку. За допомогою нового стилю можна вказувати будь-який порядок:

```
n = 42
f = 7.03
s = 'string'
```

```
q = '{2} {0} {1}'.format(f, s, n)
print(q)
```

Отримаємо:

```
42 7.03 string
```

Значення *0* відноситься до першого аргументу, *f*, *1* відноситься до рядка *s*, а *2* – до останнього аргументу, цілого числа *n*.

Аргументи можуть бути словником або іменованими аргументами, а специфікатори можуть включати їх імена:

```
q = '{n} {f} {s}'.format(n=42, f=7.03, s='string')
print(q)
```

Отримаємо:

```
42 7.03 string
```

Старий стиль дозволяє вказати специфікатор типу після символу %, а новий стиль – після `:`.

```
n = 42
f = 7.03
s = 'string'

q = '{0:d} {1:f} {2:s}'.format(n, f, s)
print(q)

# Для іменованих аргументів
q = '{n:d} {f:f} {s:s}'.format(n=42, f=7.03, s='string')
print(q)
```

Отримаємо:

```
42 7.030000 string
42 7.030000 string
```

Інші можливості (мінімальна довжина поля, максимальна ширина символів, зміщення і т.д.) також підтримуються.

```
n = 42
f = 7.03
s = 'string'
```



```
# Мінімальна довжина поля 10
q = '{0:10d} {1:10f} {2:10s}'.format(n, f, s)
print(q)

# Вирівнювання по правому краю
q = '{0:>10d} {1:>10f} {2:>10s}'.format(n, f, s)
print(q)

# Вирівнювання по лівому краю
q = '{0:<10d} {1:<10f} {2:<10s}'.format(n, f, s)
print(q)

# Вирівнювання по центру
q = '{0:^10d} {1:^10f} {2:^10s}'.format(n, f, s)
print(q)
```

Результатом запуску даного коду буде:

```
42  7.030000 string
42  7.030000  string
42    7.030000 string
42   7.030000  string
```

Значення точності (після десяткової коми) означає кількість цифр після десяткової коми для дробових чисел і максимальне число символів рядка, але не можна використовувати його для цілих чисел:

```
n = 42
f = 7.03
s = 'string'

# Без використання для цілих чисел
q = '{0:>10d} {1:>10.4f} {2:>10.4s}'.format(n, f, s)

print(q)

# Використано для цілих чисел
q = '{0:>10.4d} {1:>10.4f} {2:10.4s}'.format(n, f, s)
```

```
print(q)
```

Отримаємо:

```
42    7.0300    stri
```

Traceback (most recent call last):

File "G:\lab.py", line 5, in <module>

```
q = '{0:>10.4d} {1:>10.4f} {2:10.4s}'.format(n, f, s)
```

ValueError: Precision not allowed in integer format specifier

Якщо необхідно заповнити поле виведення чимось крім пробілів, можна розмістити необхідний символ відразу після двокрапки, але перед символами вирівнювання (<, >, ^) або специфікатором ширини:

```
q = '{0:!!^20s}'.format('ВЕЛИКІ ЛІТЕРИ')
```

```
print(q)
```

Отримаємо:

```
!!!ВЕЛИКІ ЛІТЕРИ!!!!
```

### ***Заміна символів***

Можна використовувати функцію ***replace()*** для того, щоб замінити один підрядок іншим. В функцію передається старий підрядок, новий підрядок і кількість включень старого підрядка, яку потрібно замінити. Якщо опустити останній аргумент, будуть замінені всі включення.

```
x='a a aaa a b ba ba ab cb bc'
```

```
q = x.replace('a', 'y')
```

```
print(q)
```

```
q = x.replace('a', 'y', 5)
```

# Заміна 5 входжень

```
print(q)
```

Отримаємо:

```
y y ууу y b by by yb cb bc
```

```
y y ууу a b ba ba ab cb bc
```