МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ PYTHON

КОНСПЕКТ ЛЕКЦІЙ

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний посібник для здобувачів ступеня бакалавра за освітньою програмою «Мікро- та наноелектроніка» спеціальності 153 «Мікро- та наносистемна техніка»

Київ
КПІ ім. Ігоря Сікорського
2021

Об'єктно-орієнтоване програмування мовою PYTHON : Конспект лекцій [Електронний ресурс] : навч. посіб. для студ. спеціальності 153 «Мікро- та наносистемна техніка» освітньої програми «Мікро- та наноелектроніка» / КПІ ім. Ігоря Сікорського ; уклад.: Д. Д. Татарчук, Ю. В. Діденко. – Електронні текстові данні (1 файл: 1,3 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2021. – 129 с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 7 від 13.05.2021 р.) за поданням Вченої ради факультету електроніки (протокол № 04/2021 від 26.04.2021 р.)

Електронне мережне навчальне видання

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ PYTHON

КОНСПЕКТ ЛЕКЦІЙ

Укладачі: *Татарчук Дмитро Дмитрович*, д-р техн. наук, доц.
Діденко Юрій Вікторович, канд. техн. наук, доц.

Відповідальний редактор *Свєчніков Г.С.,* канд. фіз.-мат. наук, доцент кафедри мікроелектроніки, КПІ ім. Ігоря Сікорського

Рецензент *Мельник І. В.*, доктор технічних наук, професор, професор кафедри електронних пристроїв та систем, КПІ ім. Ігоря Сікорського

У посібнику викладено основи програмування мовою Python, розглянуто вбудовані типи даних, оператори та базові модулі мови. Особливу увагу приділено засобам об’єктно-орієнтованого програмування мови Python. Окремо розглянуто графічні бібліотеки для розробки сучасного графічного інтерфейсу користувача.

Посібник призначений для студентів-бакалаврів, які навчаються за освітньо-професійною програмою «Мікро- та наноелектроніка», а також всім хто бажає вивчити основи мови програмування Python.

ЗМІСТ

ВСТУП

Python - це високорівнева універсальна інтерпретована мова програмування загального призначення. Вона з одного боку досить проста для вивчення, а з іншого боку має великий набір різноманітних бібліотечних функцій і вбудованих типів даних, що робить її зручним інструментом для вирішення широкого класу задач. Python належить до вільного і відкритого програмного забезпечення.

Мова програмування Python була розроблена Гвідо Россумом [1]. Вона з самого початку розроблялась як об’єктно-орієнтована мова програмування. Мова програмування Python підтримує структурне, об'єктно-орієнтоване, функціональне, імперативне і аспектно-орієнтоване програмування. Характерними рисами мови програмування Python є динамічна типізація, автоматичне керування пам'яттю, наявність механізму обробки виключень, підтримка багатопотокових обчислень та розвинена багаторівнева структура даних.

Структурно програма на мові програмування Python складається з модулів, які можна об'єднувати у пакети, що дозволяє значно спростити розробку програм.

Мова має чіткий і послідовний синтаксис, що робить код написаних на Python програм зрозумілим і відносно легким для сприйняття. Крім того Python реалізовано майже для всіх існуючих платформ.

Все вищевказане робить Python однією з найпопулярніших натепер мов програмування.

1. ЕЛЕМЕНТИ МОВИ PYTHON

1.1 Основні поняття

Елементами мови програмування – це її базові конструкції, з яких формується програма, а саме:

- коментарі;
- константи;
- ідентифікатори;
- зарезервовані слова;
- операції;
- спеціальні символи;
- оператори;
- типи даних.

З елементів мови формуються лексеми. Лексема – це елементарна одиниця тексту програми, яка має самостійний зміст для мови програмування. Лексема не може містити інших лексем.

Розглянемо основні елементи мови Python.

1.2 Коментарі у мові Python

Коментарем у мові Python є будь-яка послідовність символів, розміщена після символу # і до кінця строки. Коментарі не інтерпретуються мовою і не впливають на виконання програми. Коментарі пишуть для того, щоб пояснити (коментувати програму), зробити її більш зрозумілою для людини, яка буде читати текст програми. Зазвичай у коментарях пояснюють які дії виконує даний блок операторів, навіщо, а також описують важливі деталі, які необхідні для розуміння деякого блоку коду, або програми в цілому.

Наведемо кілька прикладів.

 # Це коментар від символа гратки до кінця строки
printf(“Hello!”) # це також коментар від символа гратки до кінця строки

1.3 Константи

Як і в будь-якій мові програмування у мові Python можуть використовуватись константи. Константи це дані, значення яких не змінюються протягом виконання програми. У мові Python можна використовувати константи різних типів: числові, рядкові і т.д.

Числові константи можуть бути цілими, з плаваючою крапкою, комплексними. Цілі константи можуть бути представлені у різних системах числення: десятковій, двійковій, вісімковій і шістнадцятковій.

Наведемо приклади різних констант.

```
#числові константи
11 # ціле число 11 у десятковому форматі
0b11 # ціле число 3 у двійковому форматі
0o11 # ціле число 9 у вісімковому форматі
0h11 # ціле число 17 у шістнадцятковому форматі
7.5 # число з плаваючою крапкою
0.1e-7 # число з плаваючою крапкою у інженерному форматі
5+4j # комплексне число 5-дійсна частина, 4-уявна частина
#рядкова константа 1 в одинарних лапках
‘Доброго дня!‘
#рядкова константа 2 в подвійних лапках
“Доброго дня!”
```

А для рядкових даних, які треба записати на кількох рядках програми можна використовувати потрійні лапки:

```
#багаторядкова константа
’’’ Доброго дня,
шановні ! ’’’
#а можна і так
”””Доброго дня,
шановні ! ”””
```

1.4 Ідентифікатори

Ідентифікатори – це імена функцій, змінних, констант і т.і. При формуванні ідентифікаторів необхідно виконувати ряд правил[2]:

- ідентифікатор може містити символи Unicode (включаючи їх підмножину ASCII символи) та цифри (0-9);

- першим символом ідентифікатору може бути лише літера з алфавіту (символ Unicode, включаючи ASCII символи), а також символ підкреслювання - ("_");
- ідентифікатор не може містити символ пробілу, а також спеціальні символи (лапки, слеш, ">", "<" тощо);
- у якості ідентифікаторів не можна використовувати зарезервовані слова.

Наведемо кілька прикладів.

Правильні ідентифікатори: *моя_змінна*, *myvar*, *_myvar*. Неправильні ідентифікатори: *1_myvar*, *"моя_змінна"*, *моя змінна*, *моя_змінна#.*

Ідентифікатори чутливі до регістру. Так наприклад *MyVaR*, *MyVar* і *myvar* – це три різних ідентифікатори.

1.5 Зарезервовані слова

Зарезервовані слова – це слова, які в мові програмування Python мають спеціальне значення. Їх неможна використовувати в якості ідентифікаторів.

Список зарезервованих слів наведено нижче [3, 4, 5]:

and	*as*	*assert*	*break*	*class*	*continue*
def	*del*	*elif*	*else*	*except*	*finally*
for	*from*	*global*	*if*	*import*	*in*
is	*lambda*	*nonlocal*	*not*	*or*	*pass*
raise	*return*	*try*	*while*	*with*	*yield*
True	*False*	*None*			

Повний список зарезервованих слів знаходиться у модулі *keyword*. Там же знаходиться функція *keyword.iskeyword(рядок_символів)*, яка дозволяє перевірити чи являється вказаний рядок символів зарезервованим словом.

1.6 Операції , операнди та вирази

Як і в будь-якій мові програмування у мові програмування Python для реалізації елементарних дій над даними використовують операції. Операції – це елементарні дії, які можна виконувати над даними (операндами). Кожна операція має своє позначення у вигляді спеціального символу або групи символів. Операції разом з операндами формують вирази. Наведемо приклад.

У виразі x*a ‘x’ та ‘a’ - це операнди, а знак ‘*’ – це символ, що означає операцію множення. Якщо операція застосовується одночасно до двох операндів (додавання, віднімання, множення і т.і.), то вона називається бінарною, а якщо тільки до одного оператора (мінус у позначенні від’ємного числа), то вона називається унарною). Кожна операція може застосовуватись лише до операндів певних типів. Дозволені в мові програмування Python операції умовно можна розділити на кілька груп : арифметичні операції, операції відношення, логічні (булевські) операції, інші операції.

1.6.1. Арифметичні операції

До арифметичних операцій відносяться операції додавання, віднімання, множення, ділення піднесення до степеня, унарний мінус, тощо. Найбільш використовувані арифметичні операції наведено у таблиці 1.1.

Таблиця 1.1. Арифметичні операції мови програмування Python [2]

Оператор	**Назва**	**Пояснення**	**Приклад** (вираз # результат)
+	Додавання	Додає два операнди, як результат повертає їх суму	*5+3 # 8*
-	Віднімання	Віднімає від лівого операнда правий, як результат повертає їх різницю	*2-1 # 1*
*	Множення	Перемножує операнди, як результат повертає добуток операндів	*5*2 # 10*
/	Ділення	Ділить лівий операнд на правий, повертає результат ділення.	*3/5 # 0.6*
//	Цілочислен не ділення	Ділить лівий операнд на правий, повертає цілу частину від результату	*8/5 # 1*
%	Ділення по модулю	Повертає залишок від цілочисленного ділення лівого операнда на правий	*8/5 # 3*
**	Піднесення до степеня	Підносить лівий операнд до степеня вказаного в правому операнді	*2**3 # 8*

1.6.2. Операції відношення

Операції відношення використовують для порівняння величин. Це необхідно, наприклад, для формування умовних виразів в операторах циклів,

в умовному операторі і т.і. Результатом операцій відношення є булевське значення істина або брехня (True або False). Основні операції відношення наведено у таблиці 1.2.

Таблиця 1.2. Операції відношення у мові програмування Python [4]

Оператор	**Назва**	**Пояснення**	**Приклад** (вираз # результат)
>	Більш	Повертає значення True, якщо лівий операнд більший за правий. У іншому випадку повертає False.	5>2 # True
<	Менше	Повертає значення True, якщо лівий операнд менший за правий. У іншому випадку повертає False.	5<2 # False
>=	Більше або дорівнює	Повертає значення True якщо лівий операнд не менший за правий. У іншому випадку повертає False.	5>=2 # True
<=	Менше або дорівнює	Повертає значення True якщо лівий операнд не більший за правий. У іншому випадку повертає False.	5<=2 # False
==	Дорівнює	Повертає значення True якщо лівий операнд дорівнює правому. У іншому випадку повертає False.	5==2 # False
!=	Не дорівнює	Повертає значення True якщо лівий операнд не дорівнює правому. У іншому випадку повертає False.	5!=2 # True

1.6.3. Булевські операції

Булевські (логічні операції), як і операції відношення, використовують для умовних виразів в операторах циклів, в умовному операторі і т.і. Результатом операцій відношення є булевське значення True або False.

Основні булевські операції наведено у таблиці 1.3, а таблиці істинності у таблиці 1.4.

Таблиця 1.3. Булевські операції у мові програмування Python [2]

Оператор	**Назва**	**Пояснення**	**Приклад** (вираз # результат)
and	Логічне І (логічне множення)	Повертає значення True лише у випадку, коли і правий і лівий операнди одночасно мають значення True. У іншому випадку повертає False.	(5>2) and(1<3) # True
or	Логічне АБО (логічне додавання)	Повертає значення True лише у випадку, коли хоча б один з операндів має значення True. Лише у випадку, коли обидва операнди одночасно мають значення False, повертає значення False.	(5>2) or (1>3) # True
not	Логічне заперечення	Унарний оператор – змінює значення виразу на протилежне. Якщо операнд має значення False – повертає значення True. Якщо операнд має значення – результатом буде False.	not (1>3) # True

Таблиця 1.4. Таблиці істинності логічних операцій [5]

and	**T**	**F**		**or**	**T**	**F**			**T**	**F**
T	**T**	**F**		**T**	**T**	**T**		**not**	**F**	**T**
F	**F**	**F**		**F**	**T**	**F**				

T – True, **F** – False.

1.6.4. Бітові операції

До бітових операцій (таблиця 1.5) належать операції зсуву, побітове І, побітове АБО, побітове виключне АБО і операція побітового заперечення (побітова інверсія).

Операції зсуву – бінарні операції. Перший операнд визначає величину, у якій потрібно виконати побітовий зсув, а другий операнд вказує величину цього зсуву. При цьому біти, які звільнюються заповнюються нулями, а значення бітів, що вийшли за межі початкової величини втрачається.

Інші побітові операції аналогічні відповідним логічним операціям. Тільки виконуються над окремими бітами. При цьому замість True і False фігурують 0 та 1 (таблиця 1.6).

Таблиця 1.5. Бітові операції у мові програмування Python [2]

Оператор	**Назва**	**Приклад** (вираз # результат)
<<	Побітовий зсув вліво	0b011<<1 #0b110
>>	Побітовий зсув вправо	0b011>>1 #0b001
&	Побітове логічне І	0b111&0b101 #0b101
|	Побітове логічне АБО	0b100 | 0b101 #0b101
^	Побітове виключне логічне АБО	0b100 ^ 0b101 #0b001
~	Побітове заперечення (побітова інверсія)	----------

Таблиця 1.6. Таблиці істинності побітових операцій [5]

&	**1**	**0**		**|**	**1**	**0**		**^**	**1**	**0**			**1**	**0**
1	**1**	**0**		**1**	**1**	**1**		**1**	**0**	**1**				
0	**0**	**0**		**0**	**1**	**0**		**0**	**1**	**0**		**~**	**0**	**1**

Ми розглянули далеко не всі операції мови програмування Python, деякі специфічні операції будуть розглянуті нижче.

1.6.5. Пріоритет операцій

Операнди і операції формують вирази. У найпростішому випадку вираз може складатися лише з однієї унарної операції, але частіше вираз складається з кількох операцій. Часто результат буде залежати від того, в якому порядку виконуються операції. Для того щоб не виникало неоднозначності при обчисленні виразів вводиться поняття пріоритету операцій. Операції у виразі виконуються у порядку їхнього пріоритету (таблиця 1.7), Чим вищий рівень пріоритету тим раніше виконується дана операція (у таблиці 1.7 пріоритет зростає зверху-вниз), а операції з однаковим пріоритетом виконуються у порядку зліва-направо. За необхідності змінити порядок виконання операцій – це можна зробити за допомогою округлих дужок.

Таблиця 1.7. Пріоритет операцій в мові програмування Python [2]

Оператор	**Опис**
lambda	Лямбда-вираз
or	Логічне “АБО”
and	Логічне “І”
not x	Логічне “НІ”
in, not in	Перевірка належності
is, is not	Перевірка тотожності
<, <=, >, >=, !=, ==	Операції відношення
|	Побітове “АБО”
^	Побітове “ВИКЛЮЧНЕ АБО ”
&	Побітове “І”
<<, >>	Операції зсуву
+, -	Додавання і віднімання
***, /, //, %**	Операції множення і ділення
+x, -x	Унарний мінус і унарний плюс
~x	Побітове “НІ”
**	Піднесення до степеня
x.attribute	Посилання на атрибут
x[індекс]	Звернення по індексу
x[індекс1:індекс2]	Вирізка
f(аргументи ...)	Виклик функції
(вирази, ...)	Зв’язка або кортеж [2]
[вирази, ...]	Список
{ключ:дані, ...}	Словник

1.7 Спеціальні символи

Спеціальні символи – це символи або групи символів, які мають у мові програмування спеціальне призначення. Це можуть бути пробільні символи, невідображувані символи і т.і. Спеціальні символи наведено у таблиці 1.8.

Таблиця 1.8. Спеціальні символи мови програмування Python [3]

Спеціальний символ	**Призначення**
\n	Перехід на наступний рядок
\a	Звуковий сигнал
\b	Забій попереднього символу
\f	Перехід на наступну сторінку
\r	Перехід до початку поточного рядка
\t	Горизонтальна табуляція
\v	Вертикальна табуляція
\N{id}	Ідентифікатор ID бази даних Юникода
\uhhhh	16-бітовий символ Юникода у 16-шістнадцятковому представленні представлении
\Uhhhh. . .	32-бітовий символ Юникода у 32-му представленні
\xhh	шістнадцяткове значення символу
\ooo	вісімкове значення символу
\0	Символ Null (не є ознакою кінця рядка)

Примітка: деякі спеціальні символи при виконанні програми можуть бути проігноровані операційною системою.

1.8 Оператори мови програмування Python

Для опису дій, які необхідно виконати, у програмі використовують оператори. Будь-яка мова програмування має певний набір операторів, які дозволяють виконувати всі необхідні дії з обробки даних. На відміну від багатьох інших мов програмування у мові Python після оператора точка з крапкою не ставиться. Розглянемо оператори мови програмування Python.

1.8.1. Умовні оператори

Умовні оператори використовують тоді, коли є необхідність вибору певних дій з кількох альтернативних варіантів. Вибір здійснюється на основі певної логічної умови. Для цього у мові Python використовують оператор if-elif-else. Синтаксис даного оператора має наступний вигляд:

```
if умова1:
        блок операторів 1
elif умова2:
        блок операторів 2
else:
        блок операторів 3
```

Працює оператор наступним чином. Якщо умова 1 істинна, то виконується блок операторів 1, якщо умова 1 хибна, то перевіряється умова 2, якщо вона істинна, то виконується блок операторів 2, якщо вона хибна, то виконується блок операторів 3.

Розглянемо приклад. В даному прикладі програма просить ввести з клавіатури рейтинговий бал. Функція int перетворює введений з клавіатури рядок символів у ціле число (а потім за допомогою оператора if обирає, яке повідомлення вивести на екран.

```
a=int(input('введіть ваш рейтинговий бал a ='))
if a < 60:
   print('\n a менше за 60 балів ')
   print('\n це погано')
elif  a>60 and a< 75:
   print('\n a між 60 за 75 балами')
   print('\n це непогано, але може бути краще')
else:
   print('\n a більше за 75')
   print('\n це найкраще')
```

Якщо ввести значення 76 на екран буде виведено повідомлення:

```
a більше за 75
це найкраще
```

Зверніть увагу, що у мові програмування Python відповідні блоки операторів відокремлюються лише певним числом пробілів. Ніякі дужки для цієї мети не використовують. Тому треба бути дуже уважним.

За необхідності крім основної форми оператора if у мові програмування Python можуть бути використані скорочені форми оператора. Перша скорочена форма має синтаксис:

```
if умова1:
        блок операторів 1
else:
        блок операторів 2
```

Працює вона наступним чином. Якщо умова 1 істинна, то виконується блок операторів 1, якщо умова 1 хибна, то виконується блок операторів 2

Наприклад:

```
if(n!=0):
        x=x/n
else:
        x=n
```

В даному прикладі за умови, що змінна n не дорівнює 0 буде виконано оператор x=x/n у протилежному випадку буде виконано оператор x=n.

Друга скорочена форма має синтаксис:

```
if умова1:
        блок операторів 1
```

Працює вона наступним чином. Якщо умова 1 істинна, то виконується блок операторів 1, якщо умова 1 хибна, то блок операторів 1 не виконується.

Наприклад:

```
if(n!=0):
        x=x/n
```

Крім оператора if в якості умовного оператора у мові програмування Python можна використовувати так званий тримісний оператор. Він має наступний синтаксис:

змінна = значення1 if умова else значення2

Працює даний оператор наступним чином. Якщо умова істинна змінна отримає значення 1, а в противному випадку значення2.

Наприклад:

```
a=int(input(' введіть ваш рейтинговий бал a = '))
b= ' відрахувати ' if a<60 else ' залишити '
print(b)
```

В даному випадку, якщо значення змінної *a* буде менше 60, то на екран буде виведено напис ' відрахувати ', в противному випадку буде виведено напис ' залишити '.

1.8.2. Оператори циклів

У мові програмування Python є два оператори циклів – оператор циклу з передумовою (while) і оператор покрокового циклу (for)

Оператор while має наступний синтаксис.

```
while умова :
        блок операторів 1
else:
        блок операторів 2
```

Даний оператор працює наступним чином. Якщо умова істинна виконується блок операторів 1, коли умова стає хибною виконується блок операторів 2 після чого виконання циклу завершується.

Розглянемо приклад:

```
a=int(input('введіть ціле число a ='))
f=1
if (a==0) or (a==1):
   f=1
while a>1:
   f=f*a
    a=a-1
else:
   print('\n a!=', f)
```

В даному прикладі вводиться з клавіатури ціле число a і розраховується факторіал цього числа. Після закінчення циклу виводиться на екран отримане значення. При чому блок операторів 2 відпрацює навіть якщо цикл не буде виконано жодного разу, що дозволяє вивести на екран значення факторіалів чисел 0 та 1 без використання додаткового коду.

Можливе також використання скороченої форми циклу з передумовою:

```
while умова :
        блок операторів 1
```

Наприклад:

```
a=int(input('введіть ціле число a ='))
f=1
if (a==0) or (a==1):
   f=1
while a>1:
   f=f*a
    a=a-1
```

Даний приклад працює так само, як і попередній, але отримане значення на екран не виводиться.

Оператор покрокового циклу використовується для обробки всіх елементів деякої заданої послідовності даних і має наступний синтаксис:

```
for індекс in послідовність даних:
        блок операторів 1
else:
        блок операторів 2
```

Для прикладу розрахуємо факторіал числа за допомогою покрокового циклу і виведемо отримане значення на екран.:

```
a=int(input('введіть ціле число a ='))
f=1
if (a==0) or (a==1):
   print('\n a!=', f)
else:
   for i in range(2,a+1):
      f=f*i
```

```
else:
    print('\n a!=', f)
```

Відмітимо, що для покрокового циклу довелось верхню межу діапазону значень дати на одиницю більше ніж значення введеного числа *a* оскільки для останнього елемента із заданого діапазону значень цикл не виконується. Крім того зазначимо, що на відміну від цилу *while* у покроковому циклі блок операторів 2 виконується після виконання циклу лише тоді, коли було виконано хоча б одну ітерацію. Якщо ж цикл не було виконано жодного разу блок операторів 2 не буде виконуватись, тому для відображення значень факторіалу від 0 та 1 довелось до блоку операторів 1 оператора *if* додати оператор *print('\n a!=', f)*.

Оператор покрокового циклу також має скорочену форму без розділу *else*.

1.8.3. Оператори break та continue

За допомогою циклів зручно обробляти великі масиви даних, але часто бувають ситуації, коли немає необхідності обробляти весь масив даних. Наприклад, коли під час пошуку необхідної інформації вона вже знайдена, і продовжувати цикл нема потреби. В цьому випадку необхідно перервати циклічну обробку даних (перервати виконання циклу). Для виконання такої дії у мові програмування Python є оператор *break*. Оператор *break* перериває виконання циклу, в якому він знаходиться. Слід зазначити, що коли цикли *for* або *while* перервані оператором *break*, то відповідні їм блоки *else* виконуватися не будуть.

Наведемо приклад. Нехай змінна *a* містить речення, нам необхідно виділити перше слово у реченні, записати його у змінну *c* і вивести на екран. Ознакою кінця слова будемо вважати перший пробіл після значущого символу. Це можна зробити за допомогою програми наведеної нижче. В результаті її виконання на екрані з’явиться слово *Hello*.

```
a='Hello  a world!'
c=''
for i in a:
    if i == ' ':
        break
    c=c+i
print(c)
```

Іноді виникають ситуації, коли цикл переривати нема потреби, але є необхідність пропустити одну ітерацію циклу або кілька ітерацій для уникнення небажаних наслідків. Наприклад треба уникнути ділення на 0.

Розглянемо приклад. Нехай у нас є масив значень *a*, серед яких можуть бути нулі, а нам необхідно вивести на екран результат ділення одиниці на кожен з елементів масиву *a*. Це можна зробити за допомогою наступної програми:

```
a=[1, 2, 0, 3, 0, 6, 7, 9]
c=1
for i in a:
    if i == 0:
        print('Дію виконати неможливо!')
        continue
    c=1/i
    print(c)
```

В результаті її виконання на ерані буде відображено наступне:

```
1.0
0.5
Дію виконати неможливо!
0.3333333333333333
Дію виконати неможливо!
0.16666666666666666
0.14285714285714285
0.1111111111111111
```

1.8.4. Оператор присвоювання

У мові програмування Python, як і в більшості інших мов програмування, існує простий оператор присвоювання. При виконанні

оператора присвоювання змінна отримує значення (тобто у вказану комірку пам'яті записується значення). Наприклад:

```
# змінна x отримує значення 1
x=1
#змінна y отримує значення, що є результатом виконання операції x+2
y=x+2
```

Крім простого присвоювання у мові Python є велика кількість складних операторів присвоювання (таблиця 1.9), які дозволяють сумістити в одну дію деяку операцію мови Python і операцію присвоювання значення.

Таблиця 1.9. Складні оператори мови програмування Python

Оператор	**Призначення**	**Приклад**
+=	Додавання з присвоюванням	*x=1; x+=1 # x=2*
-=	Віднімання з присвоюванням	*x=3; x-=1 # x=2*
*=	Множення з присвоюванням	*x=3; x*=2 # x=6*
/=	Ділення з присвоюванням	*x=7; x/=2 # x=3.5*
%=	Залишок від ділення з присвоюванням	*x=7; x%=2 # x=1*
>>=	Зсув вліво з присвоюванням	*x=7; x>>=2 # x=1*
<<=	Зсув вправо з присвоюванням	*x=3; x<<=2 # x=12*
^=	Побітове виключне АБО з присвоюванням	*x=7; x^=5 # x=2*
|=	Побітове АБО з присвоюванням	*x=5; x|=2 # x=7*
&=	Побітове І з присвоюванням	*x=5; x&=4 # x=4*
**=	Піднесення до степеня з присвоюванням	*x=3; x**=2 # x=9*

1.8.5. Оператор return

Оператор return використовується для виходу з функції і для повернення результату. Даний оператор має вигляд:

return результат;

Результатом може бути константа, значення змінної або результат виразу.

Операторів return у функції може бути кілька. Коли управління потоком передається на оператор return, функція завершується, навіть якщо не досягнуто її кінця. Управління передається оператору, який знаходиться безпосередньо за оператором виклику функції.

Оператор return може не повертати ніякого значення або й зовсім бути відсутній. За відсутності оператора return функція виконується до кінця. Після завершення функції управління потоком передається оператору, який знаходиться безпосередньо за оператором виклику функції. При цьому значення результату виконання функції невизначено і його не можна використовувати.

Контрольні запитання

1. З яких елементів складається мова програмування?
2. Що таке коментар? Як оформлюють коментарі у мові програмування Python? Наведіть приклад.
3. Що таке константа? Які константи можна використовувати у мові програмування Python? Наведіть приклади.
4. Що таке ідентифікатор? Які правила формування ідентифікаторів у мові програмування Python? Наведіть приклади правильних і неправильних ідентифікаторів.
5. Що таке зарезервовані слова? Наведіть приклади зарезервованих слів мови програмування Python.
6. Що таке операції і операнди? Які групи операцій допустимі у мові програмування Python?
7. Перерахуйте і опишіть арифметичні операції мови програмування Python.
8. Перерахуйте і опишіть операції відношення мови програмування Python.
9. Перерахуйте і опишіть булевські операції мови програмування Python. Наведіть таблиці істинності булевських операцій.
10. Перерахуйте і опишіть бітові операції мови програмування Python. Наведіть таблиці істинності бітових операцій
11. Опишіть порядок пріоритетності операцій мови програмування Python.
12. Що таке спеціальні символи? Перерахуйте і опишіть спеціальні символи мови програмування Python.

13. Що таке умовний оператор? Який синтаксис умовного оператора мови програмування Python? Наведіть приклади.
14. Який синтаксис оператора циклу *while* мови програмування Python? Наведіть приклади.
15. Який синтаксис оператора покрокового циклу мови програмування Python? Наведіть приклади.
16. Який синтаксис оператора *break*? Для чого він використовується? Наведіть приклади.
17. Який синтаксис оператора *continue*? Для чого він використовується? Наведіть приклади.
18. Який синтаксис оператора присвоювання? Для чого він використовується? Наведіть приклади.
19. Перерахуйте складні оператори мови програмування Python. Наведіть приклади.
20. Для чого використовують оператор *return*? Який синтаксис оператора *return*? Наведіть приклади.

2. ФУНКЦІЇ І МОДУЛІ У МОВІ PYTHON

2.1. Функції у мові Python

Функція – це іменований блок операторів, який виконує певну завершену дію і може бути використаний у програмі багато разів шляхом звернення до його імені. Функція може виконувати якісь необхідні дії або обчислювати і повертати якесь значення, яке потім може бути використано у подальшій роботі програми. Визначити функцію у мові програмування Python можна за допомогою зарезервованого слова *def*. Функція у мові програмування Python може бути визначена у будь-якому місці програми, навіть всередині іншої функції. Функція може приймати і використовувати вхідні параметри, а може і не мати параметрів. Це залежить від призначення функції. Розглянемо приклад.

```
#дана функція виводить на екран повідомлення, яке передане їй як параметр
def print_message(message):
print(message) # блок операторів функції
# закінчення функції
```

Визвати дану функцій у програмі можна наступним чином:

```
print_message('Доброго дня!')
```

В результаті роботи цієї простої функції на екран буде виведено повідомлення - *Доброго дня!*

Як уже було сказано вище функція може повертати результат. Для того, щоб використати цей результат у програмі, його треба зберегти у змінній. Наведемо приклад.

```
#дана функція з'єднує два текстових рядка, отримані як параметри
# у один текстовий рядок і повертає отриманий рядок як результат
def add_strings(a,b):
    return a+b
```

Визвати дану функцію і використати результат можна наступним чином.

```
c=add_strings('Доброго ','Дня!')
```

print(c)

В результаті виконання наведених операторів рядок *'Доброго '* і рядок *'Дня!'* будуть об'єднані і збережені у змінній *c* після чого на екрані буде надруковано значення змінної *c - 'Доброго дня!'*.

Крім звичайних функцій у мові програмування Python можна використовувати ще так звані анонімні функції. На відміну від звичайних функцій анонімні функції можуть містити лише один вираз і не потребують використання оператора *return*. Їх перевага в тому, що вони швидше виконуються. Анонімні функції задають за допомогою зарезервованого слова *lambda* [3]:

```
#дана функція обчислює і повертає як результат різницю двох значень
func = lambda x, y: x - y
```

Після визначення їх можна використовувати як звичайні функції:

```
c=func(5,2) # в результаті c=3
```

Цікавою особливістю функцій мови Python є те що їх імена можна використовувати як аргументи інших функцій. Наприклад можна визначити функцію наступним чином:

```
def fun_as_argument(a,b,c):
    return a(b,c)
```

Тепер якщо виконати команду:

```
e=fun_as_argument(func,7,2)
```

При цьому фактично буде виконано команду:

```
e =func(7,2) # в результаті e =5
```

В деяких мовах програмування для аргументів функцій можна використовувати значення за замовчуванням, якщо програміст не вкаже власних значень. В мові програмування Python також є така можливість.

Значення за замовчуванням для аргументу функції можна вказати, у визначенні функції. Для цього достатньо при описі функції присвоїти йому потрібне значення за допомогою оператора присвоювання (=). Тоді при виклику функції можна задавати не всі аргументи.

Розглянемо приклад. Визначимо функцію з двома параметрами за замовчуванням.

```
def fun_default_parameters(m1='Доброго дня, ', m2='Світ!'):
    print(m1+m2)
```

Викличемо її без аргументів.

```
fun_default_parameters()
```

На екрані з’явиться повідомлення:

Доброго дня, Світ!

Задамо лише перший аргумент:

```
fun_default_parameters('До побачення,')
```

або

```
fun_default_parameters(m1='До побачення,')
```

На екрані з’явиться повідомлення:

До побачення, Світ!

Задамо лише другий аргумент:

```
fun_default_parameters(m2='Василь!')
```

На екрані з’явиться повідомлення:

Доброго дня, Василь!

Іноді виникає необхідність створити функцію, яка могла б приймати будь-яку кількість аргументів. У мові програмування Python є така можливість. Це можна зробити двома способами. Для неіменованих аргументів це можна зробити наступним чином (зверніть увагу на використання символу ‘*’) [6]:

```
def fun_numbers_sum(*args):
    sum=0
    for n in args:
```

```
        sum+=n
    return sum

c=fun_numbers_sum(1,2) # c=3
c=fun_numbers_sum(1,2,3,4,5) # c=15
```

Для іменованих аргументів це роблять трохи інакше (зверніть увагу на використання символу ‘**’) [6]:

```
def fun_numbers_sum2(**args):
    sum=0
    for key, value in args.items():
        sum+=value
    return sum

c=fun_numbers_sum2(x=1, y=2, z=3) #c=6
c=fun_numbers_sum2(x=1, y=2, z=3, v=0, w=4) #c=10
```

2.2. Вбудовані функції

Інтерпретатор Python містить велику кількість вбудованих функцій і типів, які доступні завжди. Наведемо їх в алфавітному порядку.

abs()	*dict()*	*help()*	*min()*	*setattr()*
all()	*dir()*	*hex()*	*next()*	*slice()*
any()	*divmod()*	*id()*	*object()*	*sorted()*
ascii()	*enumerate()*	*input()*	*oct()*	*staticmethod()*
bin()	*eval()*	*int()*	*open()*	*str()*
bool()	*exec()*	*isinstance()*	*ord()*	*sum()*
bytearray()	*filter()*	*issubclass()*	*pow()*	*super()*
bytes()	*float()*	*iter()*	*print()*	*tuple()*
callable()	*format()*	*len()*	*property()*	*type()*
chr()	*frozenset()*	*list()*	*range()*	*vars()*
classmethod()	*getattr()*	*locals()*	*repr()*	*zip()*
compile()	*globals()*	*map()*	*reversed()*	*__import__()*
complex()	*hasattr()*	*max()*	*round()*	
delattr()	*hash()*	*memoryview()*	*set()*	

Розглянемо трохи детальніше ті з них, які можуть знадобитись у наших лабораторних роботах [8].

abs(x) – Обчислює модуль числа (цілого, дійсного або комплексного.

all(iterable) – повертає логічне значення *True*, коли змінна *iterable* порожня або, коли всі її єлементи мають значення *True*.

any(iterable) – повертає логічне значення *True*, коли хоч один елемент *iterable* є істинним. Для порожньої змінної *iterable* повертає значення *False*.

ascii(object) – замінює не -ASCII символи на ESC - послідовності. Наприклад символ *å* буде замінено на послідовність *\xe5*.

bin(x) – перетворює ціле число в у рядок, що має префікс *'0b'* і є символьним відображенням двійкового запису даного числа.

*bool(*x*)* – повертає еквівалент змінної *x* у булевському представленні. Наприклад для будь-якого цілого ненульового числа повертає *True*, а для нуля повертає *False*.

callable(object) – повертає логічне значення *True*, якщо object може бути викликаний (наприклад це функція або метод об'єкту), *False*, якщо ні (наприклад змінна або константа). Якщо повертається *True*, все ще можливо, що виклик не вдасться, але якщо це *False*, виклик *object* ніколи не вдасться.

chr(i) – повертає рядок з одного символу, який відповідає числу *i* (коду у таблиці UniCod).

*complex([*real*[,* imag*]])* – повертає комплексне число з заданими дійсною і уявною частинами.

delattr(object, name) – видаляє атрибут з ім'ям *name* із об'єкту *object*, якщо даний об'єкт це дозволяє.

dir ([object]) – без аргументів повертає список імен в поточній локальній області видимості. З аргументом намагатися повернути список допустимих атрибутів для зазначеного об'єкта.

divmod(a, b) – приймає два аргументи (цілі або дійсні числа) і повертає два числа – результат цілочисленного ділення *a* на *b* та залишок від цілочисленного ділення.

eval(expression) – перетворює рядок символів на число, якщо це можливо.

exec(object) – виконує команду задану рядком або об’єктом *object*, якщо це можливо.

filter(function, iterable) – створює ітератор з тих елементів *iterable*, для яких *function* повертає *True*. Змінна *iterable* повинна бути послідовністю, контейнером, який підтримує ітерацію, або ітератором.

float([x]) – повертає число з плаваючою крапкою створене із змінної *x*, якщо це можливо.

format(value[, format_spec]) – повертає форматований рядок отриманий форматуванням змінної *value* у відповідності до заданої специфікації *format_spec*. Якщо формат не заданий, то використовується формат за замовчуванням.

getattr(object, name [, default]) – повертає значення атрибута *object* з вказаним ім'ям. Ім’я *name* повинно бути рядком.

hasattr(object, name) – повертає значення *True*, якщо вказаний об’єкт має атрибут, заданий рядком *name*. У противному випадку повертає значення *False*.

help([object]) – виводить на екран коротку довідку по вказаному об’єкту, якщо це можливо.

hex (x) – перетворює ціле число в рядкове шістнадцяткове представлення з префіксом "*0x*".

input([prompt]) – функція виводить на екран повідомлення, яке міститься у рядку *prompt*, а потім зчитує рядок з вхідного потоку.

int(x=0) – перетворює значення, задане змінною *x* на ціле число у десятковому представленні, якщо це можливо. У випадку, коли змінна *x* не задана, функція повертає 0.

int(x, base=10) – перетворює рядкове представлення цілого числа *x* із системи числення заданої змінною *base*, на ціле число у десятковому представленні, якщо це можливо.

isinstance(object, classinfo) – повертає логічне значення *True*, якщо *object* є екземпляром класу *classinfo* або *False* у протилежному випадку.

issubclass(class, classinfo) – повертає логічне значення *True*, якщо *object* є підкласом класу *classinfo*.

len(s) – повертає довжину(кількість елементів об’єкту *s*).

*min(iterable, *[, key, default])* – повертає мінімальне значення із масиву, послідовності, списку або іншого ітерованого об’єкту.

*min(arg1, arg2, *args[, key])* – повертає мінімальне значення із заданих аргументів.

next(iterator[, default]) – повертає наступний елемент ітератора, якщо це можливо.

oct(x) – повертає вісімкове рядкове представлення цілого числа *x*.

ord(c) – повертає код (Unicode) символу *c*.

pow(x, y) – за відсутності змінної *z* повертає x^y.

print (objects, sep = '', end = '\n', file = sys.stdout, flush = False)* – виводить об'єкти в текстовий потік *file*, використовуючи як розділовий символ між обєктами *sep* (за замовчуванням пробіл) і як кінцевий символ *end* (за замовчуванням \n).

range(start, stop[, step]) – генерує числову послідовність від числа *start* до числа *stop-1* з кроком, *step*.

repr(object) – повертає рядок, який містить друковану версію об'єкту.

reversed(seq) – повертає послідовність *seq* в оберненому порядку.

round(number[, ndigits]) – округлює число *number* до *ndigits* цифр після десяткової крапки. За відсутності *ndigits* округлює до цілого числа.

set([iterable]) – повертає об’єкт множина, створений із заданої послідовності *iterable*. За відсутності об’єкту *iterable* повертає порожню множину.

setattr(object, name, value) – привласнює значення *value* атрибуту *name* об’єкта *object*, якщо це можливо.

*sorted (iterable, *, key = None, reverse = False)* – повертає новий відсортований список з елементів послідовності *iterable*.

sum(iterable) – повертає суму елементів послідовності *iterable*.

Повний опис вбудованих функцій можна знайти у [3, 8].

2.3. Локальні і глобальні змінні у мові Python

Всі програми так чи інакше використовують змінні. Під змінними розуміють іменовані величини, які містять необхідні для роботи програми дані. З назви зрозуміло, що їх значення може змінюватись в процесі виконання програми. Всі змінні величини, які використовуються у програмах можна умовно розділити на локальні змінні та глобальні змінні. Змінні, описані всередині функцій, – це локальні змінні. Такі змінні існують у пам’яті комп’ютера лише під час виконання функцій, у яких вони визначені. Як тільки виконання функції, у якій вони визначені, закінчується локальні змінні видаляються із пам’яті, а звільнена пам'ять використовується для збереження інших змінних. , Змінні, описані у основній програмі – глобальні. Глобальні змінні зберігаються у пам’яті до завершення програми.

В одній програмі можуть одночасно існувати глобальні і локальні змінні з однаковими іменами. Вони ніяк не пов’язані одна з одною і розміщені у різних комірках пам’яті. Локальні змінні екранують глобальні змінні з тим же ім’ям роблячи їх недоступними всередині функції. Для того, щоб глобальна змінна стала доступною у функції, необхідно вказати Python, що дана змінна глобальна. Це можна зробити за допомогою зарезервованого слова *global*. Але тоді в даній функції не можна створювати локальну змінну з такою ж назвою.

Наведемо приклад.

```
x = 10
print('\n x = ', x)
def fun1():
    global x
    x += 2 # працюємо з глобальною змінною x
    print('\n Всередині функції fun1 ')
    print('\n x = ', x) #кінець функції fun1

def fun2():
    x=25 # працюємо з локальною змінною x
    print('\n Всередині функції fun2 ')
    print('\n x = ', x) #кінець функції fun2
```

```
fun1()
print('\nПісля виконання функції fun1 ')
print('\n x = ', x) # значення глобальної змінної змінилось x = 12
fun2()
print('\nПісля виконання функції fun2 ')
print('\n x = ', x) # значення глобальної змінної без змін x = 12
```

Слід зазначити, що змінити значення глобальної змінної всередині функції без використання зарезервованого слова *global* неможливо. Але можливо використати значення глобальної змінної.

Наприклад:

```
x = 10

def fun3():
    a=x # використовуємо значення глобальної змінної x
    print('\n a = ', a) #кінець функції fun3

fun3() #на екран буде виведено a =10
```

Однак використовувати такий стиль програмування не рекомендується, оскільки це ускладнює аналіз програмного коду.

У мові програмування Python крім локальних і глобальних змінних існують ще так звані нелокальні змінні, які є проміжними між локальними і глобальними змінними. Для їх створення використовують зарезервоване слово *nonlocal*. Їх використовують у випадку, коли одна функція визначена всередині іншої, і внутрішній функції потрібен доступ до локальних змінних зовнішньої функції. Наведемо приклад [2].

```
def func_outer():
    x = 2
    print('x = ', x)
    def func_inner():
        nonlocal x
        x = 5
    func_inner()
    print('Значення локальної змінної x змінилось на ', x)
func_outer()
```

2.4. Модулі у мові Python

Як було показано вище, функція – це зручний засіб для реалізації дій, які використовуються в програмі багато разів. А чи можна зробити так, щоб функцію можна було використати не тільки в одній програмі. Звісно можна скопіювати код функції і перенести його у потрібну програму, але це не дуже зручно. Є значно ефективніший спосіб – використання модулів. Модуль – це файл, у якому зібрано набір функцій певного призначення [7]. Наприклад математичні функції мови програмування Python зібрані у файлі *math.py*, який включено до стандартної бібліотеки мови програмування Python.

Підключити модуль до програми можна за допомогою зарезервованого слова *import*:

```
import math #підключаємо до програми стандартний модуль math
# тепер можна користуватись функціями з даного модуля
#наприклад визвати функцію обчислення квадратного кореня
math.sqrt(9) #зверніть увагу комбіноване ім'я – модуль.функція
```

Можна підключати не весь модуль а окремі функції або змінні з модуля:

```
#імпортуємо математичні функції корінь квадратний, синус та косинус
from math import sqrt, cos, sin
#тепер можна звертатись до цих функцій
#зверніть увагу, що в такому разі ім'я модуля не потрібне
cos(3.14)
```

Такий спосіб має один недолік, якщо у програмі вже визначено функцію чи змінну з таким ім'ям, то вона буде замінена на імпортовану функцію, що може викликати небажані ефекти в програмі.

Якщо назва модуля дуже довга, то для спрощення набору можна задати для модуля псевдонім:

```
import math as m #тепер замість math можна використовувати псевдонім m
m.sqrt(9) #m.sqrt(9) замість math.sqrt(9)
```

Аналогічно можна задати псевдоніми для імпортованих функцій або змінних:

```
# задаємо псевдонім для функції визначення квадратного кореня
from math import sqrt as sq
sq(9) #використовуємо псевдонім
```

Підключати можна, як бібліотечні модулі мови Python, так і модулі створені програмістом. Для того, щоб модуль можна було підключити, він повинен знаходитись у доступному місці. Шляхи пошуку модулів вказані у змінній *sys.path*. Переглянути *sys.path* можна за допомогою наступних команд:

```
import sys
sys.path
```

У шляхи пошуку за замовчуванням включено поточну директорію (з якої запущено програму), а також системні директорії Python. Змінну *sys.path* можна модифікувати вручну в головній програмі, що дозволяє додати до шляху будь-який каталог:

```
import sys
sys.path.append("d:\\python_work")
```

Слід відзначити, що в разі коли модуль не буде знайдено, буде згенеровано помилку *ImportError*, а якщо не буде знайдено у модулі задану функцію чи змінну, то буде згенеровано виняткову ситуацію *AttributeError*.

Крім того слід розуміти, що оскільки Python – це інтерпретатор, тому при виконанні команди спочатку відбувається її перетворення у байт-код, а потім вона виконується. Це доволі повільний процес, особливо, якщо завантажується багато великих модулів. Тому для прискорення завантаження модулів можна завантажувати уже сформований байт-код. Це значно швидше. Зважаючи на це Python при першому використанні модуля створює файл з байт-кодом і зберігає його на диску. Цей файл має таке ж ім’я як і модуль, але інше розширення (*.pyc* замість *.py*). При подальших запусках

використовуються ці файли. Якщо вносяться якісь зміни до модуля, то ці файли також змінюються при наступному запуску. Коли ж модуль уже налагоджений і не буде змінюватись, то можна замість нього використовувати одразу файл байт-коду. Для цього їх можна збирати у спеціальні каталоги, створюючи свої власні бібліотеки.

2.5. Бібліотечні модулі мови Python

Мова програмування Python має велику кількість бібліотечних модулів, призначених для вирішення різноманітних задач. Однак за браком часу ми розглянемо лише ті модулі, які знадобляться нам у наших лабораторних роботах.

2.5.1. Математичні функції. Модуль math

Модуль *math* - один з найважливіших для нас бібліотечних модулів мови програмування *Python*. Даний модуль має великий набір функцій і констант для розв'язку математичних задач. Розглянемо їх детальніше[9].

Функції.

math.ceil (x) – округлення до найближчого більшого числа.

math.copysign (x, y) – повертає число, що має модуль такий, як у числа *x*, а знак – як у числа *y*.

math.fabs(x) – повертає модуль числа *x*.

math.factorial(x) – повертає факторіал числа *x*.

math.floor(x) – округлення до найближчого меншого числа.

math.fmod (x, y) – повертає залишок від ділення *x* на *y*.

math.frexp (x) – повертає мантису і експоненту числа *x*.

math.ldexp (x, i) – повертає *x * 2i*. Функція, обернена до функції *math.frexp().*

math.fsum(послідовність) – повертає суму всіх членів послідовності.

math.isfinite(x) – повертає логічне значення *True* , якщо *x* – скінченне число.

math.isinf(x) – перевіряє чи є *x* нескінченністю. Повертає логічне значення *False*, якщо *x* не є нескінченністю.

math.isnan (x) – перевіряє чи є *x* NaN (Not a Number – не числом). Повертає логічне значення *False*, якщо *x* – число.

math.modf(x) – повертає дробову і цілу частину числа *x*. Обидва числа мають той же знак, що і *x*.

math.trunc(x) – відкидає дробову частину числа *x*.

math.exp(x) – обчислює e^x.

math.expm1(x) – обчислює $e^x - 1$.

math.log(x, [base]) – обчислює логарифм числа *x* за основою *base*. Якщо *base* не вказано, обчислюється натуральний логарифм.

math.log1p(x) – обчислює натуральний логарифм від $(1 + x)$.

math.log10(x) – обчислює десятковий логарифм *x*.

math.log2(x) – обчислює логарифм числа *x* за основою 2 (починаючи з Python 3.3).

math.pow(x, y) – обчислює значення x^y.

math.sqrt(x) – обчислює квадратний корінь з *x*.

math.acos(x) – обчислює *x*. Результат повертає в радіанах.

math.asin(x) – обчислює арксинус *x*. Результат повертає в радіанах.

math.atan(x) – обчислює арктангенс *x*. Результат повертає в радіанах.

math.atan2(y, x) – обчислює арктангенс від величини y/x. Результат повертає в радіанах. Розрахунок проводиться з урахуванням чверті, в якій знаходиться точка (*y, x*).

math.cos(x) – обчислює косинус кута *x* (*x* вказується в радіанах).

math.sin(x) – обчислює синус кута *x* (*x* вказується в радіанах).

math.tan(x) – обчислює тангенс кута *x* (*x* вказується в радіанах).

math.hypot(x, y) – обчислює гіпотенузу трикутника з катетами *x* і *y*.

math.degrees(x) – конвертує радіани у градуси.

math.radians(x) – конвертує градуси у радіани.

math.cosh(x) – обчислює гіперболічний косинус кута *x* (*x* вказується в радіанах).

math.sinh(x) – обчислює гіперболічний синус кута *x* (*x* вказується в радіанах).

math.tanh(x) – обчислює гіперболічний тангенс кута *x* (*x* вказується в радіанах).

math.acosh(x) – обчислює зворотній гіперболічний косинус від *x*. Результат повертає в радіанах.

math.asinh(x) – обчислює зворотній гіперболічний синус від *x*. Результат повертає в радіанах.

math.atanh(x) – обчислює розраховує зворотній гіперболічний тангенс від *x*. Результат повертає в радіанах.

math.erf(x) – обчислює функцію помилок.

math.erfc(x) – обчислює додаткову функцію помилок (*1 - math.erf (x)*).

math.gamma(x) – обчислює гамма-функцію від *x*.

math.lgamma(x) – обчислює натуральний логарифм гамма-функції від *x*.

Константи.

math.pi – pi = 3,1415926...

math.e – e = 2,718281...

2.5.2. Засоби для роботи комплексними числами. Модуль cmath

Модуль *cmath* надає доступ до функцій, які можуть виконувати математичні дії над комплексними числами. Крім комплексних чисел, аргументами даних функцій можуть бути як цілі так і дійсні (з плаваючою крапкою) числа. Функції модуля *cmath* зажди повертають комплексне значення. Комплексні числа можуть бути представлені у прямокутних або у полярних координатах. Для переходу від одного представлення до іншого використовують спеціальні функції.

Деякі функції комплексної змінної є багатозначними, тому для них необхідно визначити ту область в якій їх значення буде однозначним, а сама функція неперервна.

У модулі *cmath* визначено ті ж функції, що і у модулі *math*, але адаптовані до дій над комплексними числами (за винятком функцій цілочислової математики, функцій округлення, функцій помилок, гамма

функцій і ще деяких функцій). Використання цих функцій відрізняється лише назвою модуля у визові функції. Наприклад для виклику тригонометричних функцій комплексного аргументу необхідно набрати *cmath.cos(x), cmath.sin(x)* і т.д. Крім того у модулі *cmath* визначено деякі специфічні для комплексної математики засоби. Розглянемо їх.

Функції.

cmath.phase(x) – повертає фазу комплексного числа. Ця функція еквівалентна команді *math.atan2(x.imag, x.real)*, тобто це арктангенс, який повертає значення з інтервалу *[-π;π]* (в радіанах) причому, отримане значення фази буде знаходитися в правильному квадраті, оскільки знаки *x.imag* і *x.real* відомі.

cmath.polar(x) – повертає полярні координати комплексного числа у вигляді *(r, phi)*, де *r* – це модуль комплексного числа, а *phi* – це його фазовий кут.

cmath.rect(r, phi) – обчислює дійсну і уявну частини комплексного числа по його полярних координат і повертає комплексне отримане число.

*cmath.isclose(a, b, *, rel_tol = 1e-09, abs_tol = 0.0)* – повертає *True* якщо в межах зазначеної точності (*rel_tol, abs_tol*), числа a і b близькі настільки, що їх можна вважати рівними.

Константи.

cmath.tau – 6.283185307179586.

cmath.inf – додатна нескінченість з плаваючою крапкою. Еквівалентна *float('inf')*.

cmath.infj – комплексне число з нульовою реальною частиною і позитивною нескінченною уявною частиною. Еквівалентна *complex(0.0, float('inf'))*.

cmath.nan – значення "не число" (*NaN*) з плаваючою комою. Еквівалентна *float('nan')*.

cmath.nanj – комплексне число з нульовою дійсною частиною та уявною частиною *NaN*. Еквівалентна *complex(0.0, float('nan'))*.

2.5.3. Засоби для роботи з рядковими даними. Модуль string

У модулі *string* реалізовано ряд спеціальних констант, необхідних для роботи з рядковим типом. У ранніх версіях мови програмування Python у модулі string було реалізовано також ряд функцій для роботи з рядковим типом. В останніх же версіях функції реалізовані як методи об'єктів рядкового типу даних, тому вони будуть розглянуті у розділі «Типи даних мови Python».

string.ascii_letters – містить літери латинського алфавіту.

 # 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

string.ascii_lowercase – містить маленькі літери латинського алфавіту. *# 'abcdefghijklmnopqrstuvwxyz'*

string.ascii_uppercase – містить великі літери латинського алфавіту.

 # 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

string.digits – містить цифри. *# '0123456789'*

string.printable – містить 100 друкованих символів, використовуваних у мові програмування Python

'"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'"*

string.punctuation – містить символи пунктуації та розділові символи.

 # '!"#$%&\'()+,-./:;<=>?@[\\]^_`{|}~'*

2.5.4. Модуль sys

Даний модуль забезпечує високорівневу взаємодію з операційною системою [10]. Треба зауважити, що деякі функції даного модуля працюють не в усіх операційних системах. Розглянемо найбільш використовувані функції і змінні модуля *sys*.

Функції

sys._clear_type_cache() – очищає внутрішній кеш типу.

sys.exc_info() - повертає інформацію про виняткові ситуації, які обробляються в даний момент.

sys.exit ([arg]) – здійснює вихід з поточного скрипта Python. Збуджує виключну ситуацію *SystemExit*, яка може бути перехоплена і оброблена.

sys.getdefaultencoding() - повертає використовуване в системі кодування (наприклад – *'utf-8'*).

sys.getfilesystemencoding() – повертає кодування файлової системи (наприклад – *'utf-8'*).

sys.getrefcount(object) - повертає кількість посилань на об'єкт *object*. Аргумент функції *getrefcount* – це теж ще одне посилання на об'єкт *object*.

sys.getrecursionlimit() – повертає максимальну допустиму кількість рекурсивних викликів функцій.

sys.getsizeof(object [, default]) – повертає розмір об'єкта (в байтах).

sys.getswitchinterval() – повертає час необхідний для перемикання потоків (у секундах).

sys.getwindowsversion() – повертає інформацію про поточну версію Windows (лише в операційних системах Windows).

sys.setrecursionlimit(межа) – дозволяє встановити максимальну допустиму кількість рекурсивних викликів.

sys.setswitchinterval(інтервал) – дозволяє встановити інтервал перемикання потоків (у секундах).

Змінні і константи

sys.argv – дозволяє отримати список аргументів командного рядка, які передані поточному сценарію Python при запуску. Аргумент *sys.argv*[0] містить ім’я скрипта.

sys.byteorder – визначає порядок запису-зчитування байтів. Матиме значення '*big*' при порядку проходження байтів від старшого до молодшого, і *'little*', якщо навпаки (молодший байт перший).

sys.builtin_module_names – містить перелік імен всіх доступних модулів.

sys.copyright – повертає рядок, що містить авторські права, які відносяться до інтерпретатора Python.

sys.dllhandle - ціле число, яке визначає дескриптор DLL Python (лише у операційній системі Windows).

sys.exec_prefix – містить каталог установки Python.

sys.executable – містить шлях до інтерпретатора Python.

sys.flags – містить прапори командного рядка (тільки для читання).

sys.float_info – містить інформацію про тип даних *float*.

sys.float_repr_style – містить інформацію про застосування вбудованої функції *repr()* для типу float.

sys.hexversion – містить поточну версію Python у вигляді шістнадцяткового числа (для 3.2.2 final це буде 30202f0).

sys.implementation – містить інформацію про запущену версію Python.

sys.int_info – містить інформацію про тип int.

sys.path – містить список шляхів пошуку модулів.

sys.platform – містить інформацію про операційну систему.

sys.dont_write_bytecode – якщо містить *True*, python не писатиме .pyc файли.

sys.stdin – стандартний потік введення.

sys.stdout – стандартний потік виведення.

sys.stderr – стандартний потік помилок.

2.5.5. Модуль time

Модуль *time* надає засоби для обробки часу і дати [11].

Функції.

time.asctime([t]) - перетворює кортеж або *struct_time* в рядок виду *'День тжня місяць число час рік'* (наприклад *'Sun May 31 12:33:02 2020'*). Якщо аргумент не вказано, використовується поточний час.

time.clock() – у операційній системі Unix, повертає поточний час. У Windows, повертає час, що минув з моменту першого виклику даної функції.

time.ctime ([сек]) – перетворює час, виражений в секундах з початку епохи комп'ютерної техніки (*'Thu Jan 1 02:00:00 1970'*) в рядок виду *'День тжня місяць число час рік'.*

time.gmtime([сек]) - перетворює час, виражений в секундах з початку епохи комп'ютерної техніки (*'Thu Jan 1 02:00:00 1970'*) у тип даних *struct_time*.

time.mktime(t) – перетворює кортеж або *struct_time* в число секунд з початку епохи комп'ютерної техніки.

time.sleep(сек) – затримує виконання програми на задану кількість секунд.

time.strftime(формат, [t]) – перетворює кортеж або *struct_time* в рядок згідно заданого формату (див документацію на модуль).

time.strptime(рядок [, формат]) – перетворює рядок, який представляє час відповідно до формату. Повертає значення типу *struct_time*.

Константи і змінні.

time.altzone – змінна, яка містить зміщення часового поясу в секундах від нульового меридіана. Якщо часовий пояс знаходиться на захід від нульового меридіана, то зміщення додатне, якщо на схід, то зміщення від'ємне.

time.daylight – змінна, яка містить не 0, якщо визначено, зимовий або літній час.

2.5.6. Модуль random

Одним з важливих модулів є модуль random. У цьому модулі знаходиться ряд функцій для роботи з генератором випадкових чисел. Розглянемо ці функції.

seed() – ініціалізація генератора випадкових чисел.

getstate() – повертає поточний стан (*state*) генератора випадкових чисел.

setstate(state) – відновлює внутрішній стан (*state*) генератора випадкових чисел.

getrandbits(bit_num) – повертає ціле число, яке може бути представлене заданим у *bit_num* числом біт.

randrange(start,stop,step) – повертає випадкове число в межах заданого проміжку (*start:stop:step*).

randint(start,stop) – повертає випадкове число в межах заданого проміжку.

choice(послідовність) – повертає вибраний випадковим чином елемент із заданої послідовності.

choices(sequence, weights=None, cum_weights=None, k=1) – повертає вибрану випадковим чином послідовність елементів розміром *k* із заданої послідовності *sequence* . За замовчуванням параметр *k=1*. Більш докладну інформацію дивіться у [2, 5].

shuffle(послідовність) – перетасовує елементи заданої *послідовності* випадковим чином. Результат зберігається цій самій послідовності. Наприклад:

b=[1,2,5,4,7]

random.shuffle(b) # b=[1, 4, 2, 7, 5] (порядок випадковий)

sample(послідовність, кількість_вибраних_елементів) – повертає послідовність розміром *кількість_вибраних_елементів* вибрану випадковим чином із заданої *послідовності*.

random() – повертає випадкове дійсне число в проміжку від 0 до 1.

uniform(start,stop) – повертає випадкове дійсне число в зазначеному проміжку (*start,stop*).

betavariate(alpha,beta) – повертає випадкове дійсне число в проміжку між 0 і 1, грунтуючись на бета-розподілі, який використовують у статистиці (*alpha* і *beta* – параметри розподілу).

expovariate(параметр_розподілу) – повертає випадкове дійсне число в проміжку між 0 і 1, або ж між 0 і -1, коли параметр негативний. За основу береться експоненціальний розподіл, який використовується в статистиці.

Також є ряд функцій, які повертають випадкове дійсне число в проміжку між 0 і 1 базуючись на інших відомих розподілах (гамма-розподілі, гауссовому розподілі, нормальному розподілі і т.д.), які ми за браком часу розглядати не будемо.

2.5.7. Робота з регулярними виразами, модуль re

Регулярний вираз - це послідовність символів, яка використовується для пошуку і заміни тексту в рядку або файлі [17]. Для роботи з ними у мові програмування Python використовують функції модуля re. Розглянемо ці функції.

re.match(pattern, string) – перевіряє чи починається заданий рядок *string* з виразу *pattern*. Як результат повертає знайдений вираз. Якщо ні, то повертає значення *None*.

re.search(pattern, string) – здійснює пошук виразу *pattern* у рядку *string*. Як результат повертає перший знайдений вираз. Якщо не знаходить, то повертає значення *None*. Знаходить лише перше входження.

re.findall(pattern, string) – здійснює пошук виразу *pattern* у рядку *string*. Як результат повертає список всіх знайдених об'єктів. Якщо не знаходить, то повертає значення *None*.

re.split(pattern, string, [maxsplit=0]) – розділяє заданий рядок *string*, на кілька рядків використовуючи як межу розділу вираз *pattern*. Повертає список отриманих об'єктів. При цьому вираз *pattern* видаляється. Аргумент *maxsplit* задає кількість розбиттів. За замовчуванням дорівнює 0, що передбачає поділ рядка на максимально можливу кількість об'єктів. Якщо вказано число не рівне нулю, то буде зроблено розбиття на кількість об'єктів, яка не більше значення аргументу *maxsplit+1*.

re.sub(pattern, repl, string) – здійснює пошук у рядку *string* виразу *pattern* і замінює всі знайдені вирази виразом *repl*.

re.compile(pattern, repl, string) – дозволяє заздалегідь скомпілювати регулярний вираз для подальшого його використанн. Це особливо корисно в тих випадках, коли регулярний вираз використовується багато раз.

```
#наприклад
pattern = re.compile(r'AV' r'123')
print(patern)
```

На екрані буде надруковано

re.compile('AV123')

А можна так:

```
a='123'
b='dfg'
pattern = re.compile(a+b)
print(pattern)
```

На екрані буде надруковано

re.compile('123dfg')

Більш докладну інформацію можна знайти у [17, 18, 19].

Контрольні запитання

1. Дайте визначення функції. Наведіть приклади.
2. Що таке анонімна функція? Який синтаксис анонімної функції?
3. Як визначити функцію? Наведіть приклади.
4. Яким чином можна використати ім’я функції у якості аргумента іншої функції? Наведіть приклад.
5. Поясніть, що таке аргументи за замовчуванням. Наведіть приклади.
6. Поясніть, що таке іменовані аргументи. Як їх використовують? Наведіть приклади.
7. Що таке вбудовані функції? Наведіть приклади.
8. Що таке локальні змінні? Чим вони відрізняються від глобальних? Наведіть приклади.
9. Для чого використовують ключове слово *global*? Наведіть приклади.
10. Що таке нелокальна змінна? Наведіть приклади.
11. Що таке модуль? Для чого їх використовують? Наведіть приклади.
12. Як підключити до програми модуль? Наведіть приклади.
13. Опишіть бібліотечний модуль *math*.
14. Опишіть бібліотечний модуль *cmath*.

15. Опишіть бібліотечний модуль *string*.
16. Опишіть бібліотечний модуль *sys*.
17. Опишіть бібліотечний модуль *time*.
18. Опишіть бібліотечний модуль *random*.
19. Опишіть бібліотечний модуль *re*.

3. ТИПИ ДАНИХ МОВИ PYTHON

Про Python можна сказати, що він відноситься до мов з неявною сильною динамічною типізацією. Це означає, що по-перше при створенні змінної необов’язково описувати її тип, а по-друге, що тип змінної може змінюватись в процесі виконання програми.

В Python типи даних можна умовно розділити на вбудовані в інтерпретатор і не вбудовані, які можна використовувати при імпортуванні відповідних модулів.

До основних вбудованих типів відносяться:

- int – ціле число;
- float – число з плаваючою крапкою;
- complex – комплексне число;
- логічні змінні (Boolean Type);
- рядки;
- list – список;
- tuple – кортеж;
- range – діапазон;
- set – множина;
- dict – словник;
- bytes – байти;
- bytearray – масиви байт;
- memoryview – спеціальні об'єкти для доступу до внутрішніх даних об'єкта;
- *None* (невизначене значення змінної).

В мові програмування Python всі типи даних реалізовані як класи. Крім власне самих значень вони мають деякі методи для обробки цих значень.

3.1. Числові типи

3.1.1. Цілі числа (int)

Цілі числа в мові програмування Python підтримують всі елементарні операції допустимі над цілими числами такі як арифметичні операції, бітові операції, тощо. Але крім цього вони підтримують ряд додаткових методів.

int.bit_length() – повертає кількість біт необхідних для відображення числа у двійковій системі числення без урахування знаку і лідуючих нулів.

Наприклад:

int.bit_length(3) #2

 # або

(3).bit_length()#2

*int.to_bytes(length, byteorder, *, signed=False)* – повертає рядкове шістнадцяткове представлення числа, де *length* – це кількість байт у рядку, *byteorder* – порядок байт (*'big'* або 'little')), *signed* вказує чи враховувати знак (*False* – ні, *True* - да).

Наприклад:

(15).to_bytes(3, byteorder='big') #b'\x00\x00\x0f'

(15).to_bytes(3, byteorder='little') #b'\x0f\x00\x00'

(-15).to_bytes(3, byteorder='big', signed=True) #b'\xff\xff\xf1'

int.from_bytes(bytes, byteorder, *, signed=False) – перетворює на число його рядкове шістнадцяткове представлення.

Приклад:

int.from_bytes(b'\x00\x10', byteorder='big') #16

3.1.2. Дійсні числа (float)

Дійсні числа підтримують всі арифметичні операції, а також кілька додаткових методів:

float.as_integer_ratio() – повертає пару цілих чисел, відношенням яких є дане число.

Приклад:

float.as_integer_ratio(0.5) # (1, 2)

float.is_integer(number) – повертає *True*, якщо *number* – ціле число, в противному випадку повертає *False*.

Приклад:

float.is_integer(5.0) #True

float.is_integer(5.1) #False

3.1.3. Комплексні числа (complex)

Комплексні числа є вбудованим у Python типом даних. Вони підтримують усі вбудовані функції, призначені для роботи зними (див. розділ 2.2), а також всі функції модуля *cmath* (див. розділ 2.5.2).

3.2. Логічний (булевський) тип

Логічний тип даних може приймати лише одне з двох значень: істина або неправда. В Python є дві константи для позначення цих значень: *True* (істина) і *False* (неправда). Лише їх можна присвоювати змінним логічного типу. Ці ж значення є результатом логічних операцій. Допустимі логічні операції розглянуто у розділі 1.6. Крім логічних операцій над логічними значеннями можна виконувати операції додавання, віднімання, множення та ділення. У цьому випадку *True=1*, а *False=0*. Наприклад:

a=True+True #a=2

*b=True*3 #b=3*

*c=True*False #c=0*

d=True-False #d=1

e=False –True #e=-1

a=5-True #a=4

3.3. Рядкові типи

Рядок у Python – це послідовність символів. Як правило - це просто деякі слова або набори слів. Ці набори слів можуть бути на будь-якій мові, яка підтримується стандартом Unicode.

У мові програмування Python 3 немає окремих ASCII-рядків, тому що множина символів ASCII є підмножиною множини символів Unicode.

Рядки незмінні. Це означає, що після створення рядка його більше не можна змінювати. Однак це не додає незручностей при програмуванні.

В мові програмування Python нема окремого типу даних char (символ). Замість нього використовують рядок, який складається з одного символу. Кількість символів у рядку можна визначити за допомогою вбудованої функції *len(рядок)*. Кожен символ рядка має свій порядковий номер (індекс). Нумерація символів починається з нуля. Допустимі також від'ємні індекси. Від'ємний індекс інтерпретується, як сума даного від'ємного числа і довжини рядка. Наприклад:

s='Hello World!'
s[0] #'H'
s[-2] # 'd'
s[len(s)-2] #'d'
len(s)-2 #10
s[10] #'d'

При зверненні до елементів рядка можна використовувати діапазон індексів (зріз). Зріз має синтаксис:

Строка[початковий елемент зрізу:кінцевий елемент зрізу:крок вибору]

За замовчуванням *початковий елемент зрізу = початковому елементу рядка*, *кінцевий елемент зрізу = кінцевому елементу рядка*, *крок вибору=1*.

Розглянемо приклади:

s='Hello World!'
s[0:5] #'Hello'

 # або

s[:5] #'Hello'

 # або весь рядок

s[:] #'Hello World!'

 # або кожен другий елемент рядка (вибір з кроком 2)

s[::2] #'HloWrd'

Рядки можна додавати і множити на число:

d='Hello! '

k='World!'

d+k #'Hello! World!'

*d*3 #'Hello! Hello! Hello! '*

Рядки можна порівнювати. Порівнюють посимвольно коди символів до першого неспівпадання:

d='Hello! '

k='World!'

d==k #False

d>k #False

k>d #True

c=d[0:6] +'!!' #'Hello!!!'

d>c #False

d<c #True

Як і всі інші типи даних мови програмування Python рядки є об’єктами і містять ряд методів для роботи з ними. Розглянемо їх.

s.find(str, [start],[end]) – здійснює пошук рядка *str* у зрізі рядка *s* від позиції *start* до позиції *end*. Повертає позицію першого входження або -1, якщо шуканий рядок не входить до даного зрізу.

s.rfind(str, [start],[end]) – здійснює пошук рядка *str* у зрізі рядка *s* від позиції *start* до позиції *end*. Повертає позицію останнього входження або -1, якщо шуканий рядок не входить до даного зрізу.

s.index(str, [start],[end]) – здійснює пошук рядка *str* у зрізі рядка *s* від позиції *start* до позиції *end*. Повертає позицію першого входження або генерує помилку *ValueError*, якщо шуканий рядок не входить до даного зрізу.

s.rindex(str, [start],[end]) – здійснює пошук рядка *str* у зрізі рядка *s* від позиції *start* до позиції *end*. Повертає позицію останнього входження або генерує помилку *ValueError*, якщо шуканий рядок не входить до даного зрізу.

s.replace (шаблон, заміна) – шукає у рядку *s* рядок, заданий змінною *шаблон* і замінює його на рядок, заданий змінною *заміна*. Якщо знаходить кілька входжень, то заміняє їх всі.

s.split (символ) – розділяє рядок на частини, вважаючи вказаний *символ* межею розділу.

s.isdigit() – перевіряє, чи всі символи рядка – цифри, чи ні. Повертає логічне значення *True*, якщо всі символи рядка – цифри. У противному випадку повертає *False*.

s.isalpha() – перевіряє, чи всі символи рядка – літери, чи ні. Повертає логічне значення *True*, якщо всі символи рядка – літери. У противному випадку повертає *False*.

s.isalnum() – перевіряє, чи всі символи рядка є літерами або цифрами. Повертає логічне значення *True*, якщо це так. У противному випадку повертає *False*.

s.islower() – перевіряє чи всі символи рядка є літерами нижнього регістру. Повертає логічне значення *True*, якщо це так. У противному випадку повертає *False*.

s.isupper() – перевіряє чи всі символи рядка є літерами верхнього регістру. Повертає логічне значення *True*, якщо це так. У противному випадку повертає *False*.

s.isspace() – перевіряє чи всі символи рядка є не відображуваними символами (пробіл, табуляція, перхід на інший рядок, тощо). Повертає логічне значення *True*, якщо це так. У противному випадку повертає *False*.

s.istitle() – перевіряє чи є перший символ рядка літерою верхнього регістру. Повертає логічне значення *True*, якщо це так. У противному випадку повертає *False*.

s.upper() – перетворює всі літери рядка у літери верхнього регістру.

s.lower() – перетворює всі літери рядка у літери нижнього регістру.

s.startswith(str) – перевіряє чи починається рядок з послідовності літер вказаної у рядку *str*. Повертає логічне значення *True*, якщо це так. У противному випадку повертає *False*.

s.endswith(str) – перевіряє чи закінчується рядок послідовністю літер вказаною у рядку *str*. Повертає логічне значення *True*, якщо це так. У противному випадку повертає *False*.

s.join(список) – об'єднує елементи *списку* у рядок, розділяючи їх у рядку символом, вказаним у *s*. Наведемо приклад:

```
A = ['red', 'green', 'blue']
g=':'.join(A) # g= 'red:green:blue'
```

s.capitalize() – переводить перший символ рядка у верхній регістр, а всі інші символи рядка у нижній.

s.center(width, [fill]) – центрує рядок, доповнюючи його символами *fill* (за замовчуванням пробілами) зліва і справа до довжини *width* символів. Наведемо приклад:

```
c='a123WERT!'
x=c.center(15, '#') # x= '###a123WERT!###'
```

s.count(str, [start],[end]) – підраховує кількість входжень рядка *str* у зріз рядка *s*, обмежений номерами елементів рядка від *start* до *start*. За замовчуванням зріз охоплює весь рядок. При підрахунку не враховують рядки, що перетинаються. Наприклад:

```
c='rterterbnbrter'
x= c.count('rter') # x=2
```

s.expandtabs([tabsize]) – створює копію рядка *s*, у якій замінює кожну табуляцію (*tabsize пробілів підряд*) одним пробілом. За замовчуванням *tabsize=8*.

s.lstrip([chars]) – створює копію рядка *s*, у якій видалені всі пробіли від початку рядка до першого не пробільного символу. Символ, який функція вважає пробільним задається у аргументі *chars.* За замовчуванням – це символ пробілу.

s.rstrip([chars]) – створює копію рядка *s*, у якій видалені всі пробіли від останнього не пробільного символу до кінця рядка. Символ, який функція вважає пробільним задається у аргументі *chars.* За замовчуванням – це символ пробілу.

s.strip([chars]) – видаляє всі пробільні символи на початку і в кінці рядка. Символ, який функція вважає пробільним задається у аргументі *chars.* За замовчуванням – це символ пробілу.

s.swapcase() – переводить всі символи рядка верхнього регістру до нижнього і навпаки.

s.title() – у кожному слові рядка *s* переводить першу літеру до верхнього регістру, а всі інші до нижнього.

s.z1ll(width) – доповнює рядок *s* зліва нулями до кількості символів, заданої у аргументі *width*.

s.ljust(width, fillchar=" ") – доповнює, до заданої у аргументі *width* кількості символів, рядок *s* зліва символами вказаними у аргументі *fillchar*.

s.rjust(width, fillchar=" ") – доповнює, до заданої у аргументі *width* кількості символів, рядок *s* справа символами вказаними у аргументі *fillchar*.

s.format(аргументи) – функція, яка допомагає від форматувати рядок заданим чином. Більш докладну інформацію можна знайти за посиланням [12]. Розглянемо приклади:

'Привіт, {}, {}, {}!'.format('Василь','Петро','Степан')

#результат – 'Привіт, Василь, Петро, Степан!'

'Привіт, {0}, {1}, {2}!'.format('Василь','Петро','Степан')

#результат – 'Привіт, Василь, Петро, Степан!'

'Привіт, {2}, {1}, {0}!'.format('Василь','Петро','Степан')

#результат – 'Привіт, Степан, Петро, Василь!'

'Мене звуть {name} {last_name}!'.format(name='Василь',last_name='Петров')

#результат – 'Мене звуть Василь Петров!'

'Мене звуть {last_name} {name}!'.format(name='Василь',last_name='Петров')

#результат – 'Мене звуть Петров Василь!'

3.4. Списки

Списки в Python - впорядковані колекції об'єктів довільних типів. Списки схожі на масиви, але кожен елемент списку може відрізнятись за типом від інших елементів списку. Можна також створювати багаторівневі списки, елементами яких є списки.

Задати список можна двома способами – за допомогою ключового слова *list* або за допомогою квадратних дужок. Далі наведено приклад у якому створюють два однакових списки різними способами.

z=[7,1,2,7,9,0,'a'] # z =[7, 1, 2, 7, 9, 0, 'a']

c=list((7, 1, 2, 7, 9, 0, 'a')) # c =[7, 1, 2, 7, 9, 0, 'a']

d=[[1,2],[0,3,4],[9,5]] #багаторівневий список

#звертатись до елементів такого списку можна за індексами

#нумерація елементів починається з нуля

#перший індекс вказує номер вкладеного списку,

#а другий номер елемента у вкладеному списку.

#Наприклад: d[2][1] дорівнює 5, d[1][2] дорівнює 4

Як і всі інші типи даних мови Python список це – клас, який має ряд методів. Потрібно відзначити, що методи списків, на відміну від строкових методів, змінюють сам список, а тому нема потреби без нагальної необхідності необхідності зберігати результат у іншій змінній. Розглянемо методи списків:

list.append(x) – додає елемент *x* у кінець списку.

list.extend(L) – розширює список *list*, додаючи у кінець списку *list* всі елементи списку *L*.

list.insert(i, x) – додає у *i*-ту позицію списку *list* елемент із значенням *x*.

list.remove(x) – видаляє із списку *list* перший знайдений елемент, який має значення *x*. Генерує помилку *ValueError*, якщо такого елементу не існує.

list.pop([i]) – видаляє *i* -ий елемент із списку *list* і повертає його значення як результат. Якщо індекс не вказано, видаляється останній елемент.

list.index(x, [start [, end]]) – проводить пошук елемента зі значенням *x* у зрізі *[start : end]*. Як результат повертає його положення у списку *list*. Якщо таких значень у зрізі кілька, то повертає значення першого знайденого у зрізі елементу. Генерує помилку *ValueError*, якщо такого елементу у вказаному зрізі не існує.

list.count(x) – повертає кількість елементів списку, які мають значення *x*.

list.sort ([key = функція], [reverse=False]) – сортує список на основі заданої функції. Якщо параметр *reverse=False*, то сортує за зростанням, якщо *True* – за спаданням. За замовчуванням сортує за зростанням. Сортувати можна лише списки з однотипними елементами (всі елементи числа, або всі елементи символи, тощо).

list.reverse() – розміщує елементи списку *list* у порядку зворотному до початкового.

list.copy() – повертає як результат копію списку *list*.

list.clear() – очищує список *list*.

3.5. Тип кортеж (tuple)

Кортежі служать для зберігання кількох об'єктів разом. Їх можна розглядати як аналог списків, але з обмеженою функціональністю. Одна з найважливіших відмінностей кортежів від списків полягає в тому, що вони незмінні. Друга їх відмінність полягає в тому, що вони займають менший об’єм пам’яті.

Кортежі зазвичай використовують в тих випадках, коли для коректної роботи оператора або користувацької функції необхідно забезпечити незмінність набору значень, з якими вони працюють.

Кортежі позначають зазначенням елементів, розділених комами; за бажанням їх можна ще укласти в круглі дужки [2]. Всі операції над списками, що не змінюють список (додавання, множення на число, методи index () і count () і деякі інші операції) допустимі при роботі з кортежами. Можна також по-різному змінювати елементи місцями і так далі. Якщо до складу кортежу входить список, то можна змінювати елементи цього списку, не дивлячись на те, що кортеж - це незмінний тип даних. Розглянемо приклади.

```
b = ('1', 2, '4') # визначаємо кортеж b
c=tuple(('1', 2, '4')) # ще один спосіб визначити кортеж (c і b еквівалентні)
d='1', 2, '4' # можна і так кортежі c. b і d – еквівалентні
e=() # e – порожній кортеж
a=(1) #це не кортеж, а ціле число
a=1 #а це також не кортеж, а ціле число
a=(1,) #а це кортеж з одним елементом
#або так
a=1,
#порівняємо розміри списку і кортежу
a = (1, 2, 3, 4, 5, 6) #кортеж
b = [1, 2, 3, 4, 5, 6] #список
print('розмір кортежу - ',a.__sizeof__()) # 36 - байт
print('розмір списку - ',b.__sizeof__()) # 44 – байти
#за допомогою кортежів можна присвоювати значення одночасно
#кільком змінним:
a=(1,2,3)
(x,y,z)=a # x=1,y=2,z=3
#або та ж дія в інший спосіб
(x,y,z)= (1,2,3)
# можна також поміняти місцями вміст кількох змінних
x,y,z=y,z,x # x=2,y=3,z=1
#визначимо кортеж, який містить список (список – [3,4,5])
a=(1,[3,4,5],2)
#змінимо перший елемент списку – замінимо 4 на 7
#пам'ятаємо, що нумерація елементів починається з нуля
#список – перший елемент кортежу, а 4-ка перший елемент списку, тому:
a[1][1]=7 # тепер a=(1,[3,7,5],2)
```

3.6. Тип діапазон (range)

Тип діапазон – це незмінна послідовність цілих чисел [13]. Перевагою даного типу в порівнянні зі звичайним списком або кортежем, є те, що він займає завжди однакову невелику кількість пам'яті незалежно від того, якої довжини діапазон. Це можливо тому, що в пам'яті зберігаються тільки параметри діапазону, а значення обчислюються в міру необхідності. Фактично діапазон представляє собою список членів геометричної прогресії. Створити діапазон можна за допомогою функції range.

range(start, stop[, step]), де *start* – це початкове значення послідовності (арифметичної прогресії), *stop* – кінцеве значення, а *step* крок (різниця арифметичної прогресії). За замовчуванням *start=0*, а *step=1*. При цьому треба мати на увазі, що *stop* – це не останній член послідовності а межа діапазону. Це значення не входить до послідовності згенерованих чисел. Так реалізовано, тому, що генерується арифметична прогресія, а значення *stop*, в залежності від значення *step*, може не бути членом прогресії. Тому для однозначності і для спрощення алгоритму за рахунок зменшення перевірок, значення *stop* не враховують, навіть коли воно є членом прогресії. Розглянемо приклади:

```
x=range(1,5,1) # буде згенеровано послідовність чисел 1,2.3,4
x=range(5) # буде згенеровано таку саму послідовність чисел 1,2.3,4
x=range(2,5,2) # буде згенеровано послідовність чисел 2,4
range(5,2) # буде згенеровано порожню послідовність,
#бо початкове значення більше за кінцеве, а крок за замовчуванням – 1
range(5,2,-1) # буде згенеровано послідовність чисел 5,4,3
range(-5,-1,1) # буде згенеровано послідовність чисел -5,-4,-3,-2
range(-1,-5,-1) # буде згенеровано послідовність чисел -1,-2,-3,-4
```

Діапазони можна перевіряти на рівність. Перевірка діапазонів на рівність за допомогою операцій == і *!=* порівнює їх як послідовності. Це означає, що два діапазони рівні, якщо вони представляють однакову послідовність значень. Наприклад:

```
x=range(5,2,-1)
z=range(5,3,-1)
```

```
z==x #False
z!=x#True
```

Крім того можна перевіряти входження числа в діапазон за допомогою команди *in*:

```
x=range(5,2,-1)
4 in x #True
6 in x #False
```

Тип діапазон є ітерованим типом даних і може бути використаний у операторах циклів;

```
for i in range(-5,-1,1):
    print(i)
else:
    print('\n')
```

На екран буде виведено послідовність:

```
-5
-4
-3
-2
```

```
for i in range(-1,-5,-1):
    print(i)
else:
    print('\n')
```

На екран буде виведено послідовність:

```
-1
-2
-3
-4
```

```
for i in range(5,2):
    print(i)
else:
    print('\n')
```

Останній цикл не буде виконано жодного разу, оскільки задано порожній діапазон.

Також до елементів діапазону можна звертатись за індексом. При цьому необхідно слідкувати, щоб не вийти за межі діапазону:

```
x=range(5,2,-1)
c=x[0] # c=5
l=len(x) # l=3 допустимі індекси 0, 1, 2
```

У випадку, коли індекс виходить за межі діапазону генерується помилка.

3.7. Тип множина (set)

Множина у Python - це тип даних, який представляє собою невпорядкований набір елементів [5]. Всі елементи множини – унікальні (повторів немає). Оскільки множини не впорядковані, вони не підтримують операцій зрізу і операцій індексування. Створити множину можна двома способами – за допомогою фігурних скобок або за допомогою функції *set()* [14]:

```
a={1,4,2,7,3,1,2} # створено множину чисел 1,4,2,7,3
b=set([1,4,2,7,3,1,2]) # створено аналогічну множину чисел 1,4,2,7,3 зі списку
c= set() # створено порожню множину
# а так неможна, бо буде створено порожній словник
# замість порожньої множини
e=={}
```

Елементами множини можуть бути числа, рядки, кортежі, тощо. В одній множині можуть знаходитись одночасно елементи різних типів:

```
names={'Василь','Петро','Степан'}
# змішана множина чотирьох елементів
# містить рядок, кортеж і два числових елементи
mix={'string',(1,2,3),3,4}
a=len(mix) # a=4
```

Елементи множини незмінні, а сама множина змінна величина. До множини можна додавати елементи, а можна і вилучати елементи із множини.

c={1,2,3} # с містить елементи 1,2,3
c.add(5) # до с додано елемент 5, тепер с містить елементи 1,2,3,5
c.discard(2) # з с вилучено елемент 2, тепер с містить елементи 1,3,5
c.remove(1) # з с вилучено елемент 1, тепер с містить елементи 3,5

Різниця між *remove()* і *discard()* полягає в тому, що уразі відсутності вказаного елемента у множині функція *remove()* згенерує помилку, а функція *discard()* – ні.

До множин можна використовувати логічні операції ==, *!*= >, < та інші результати при цьому визначаються правилами математичної теорії множин:

- вираз A==B істинний тільки тоді, коли A і B містять однакові елементи;
- вираз A!=B істинний, якщо хоч одна із заданих множин містить хоч один елемент, відсутній в іншій;
- вираз A<=B істинний, якщо кожен елемент множини A являється також елементом множини B;
- вираз A>=B істинний, якщо кожен елемент множини B являється також елементом множини A. Розглянемо приклади:

a={1,2,3}
b={3,4,5}
e={1,2,3}
a==b # результат False
a!=b # результат True
a==e # результат True

За допомогою операції in можна перевірити наявність елемента у множині:

a={1,2,3}
2 in a # результат True
5 in a # результат False

Множина – це клас і він має ряд вбудованих методів. Розглянемо ці методи [14].

Методи, що не змінюють множину:

set.isdisjoint(other) – повертає логічне значення *True*, якщо *set* і *other* не мають спільних елементів.

set.issubset(other) – повертає логічне значення *True*, якщо *set* <= *other* (всі елементи *set* належать *other*).

set.issuperset(other) – повертає логічне значення *True*, якщо set> = other.

set.union (other, ...) – об'єднує множини прелічені у дужках з множиною *set*.

set.intersection(other, ...) – формує перетин множини *set* і перелічених у дужках множин (формує множину, яка містить елементи спільні для всіх вказаних множин).

set.difference (other, ...) – формує множину елементів множини *set*, які не належать жодній з перелічених у дужках множин.

set.symmetric_difference(other) – містить елементи множин *set* і *other*, які не є спільними для цих множин.

set.copy() – створює копію множини *set*.

Методи, що змінюють множину:

set.update(other, ...) – замінює множину *set* на об’єднання множини *set* і перелічених в дужках множин.

set.intersection_update(other, ...) – замінює множину *set* на перетин множини *set* і перелічених в дужках множин.

set.difference_update(other, ...) – замінює множину *set* на різницю множини *set* і перелічених в дужках множин.

set.symmetric_difference_update(other) – замінює множину *set* на множину, яка складається з елементів множин *set* і *other*, які не є спільними для цих множин.

set.add(elem) – додає елемент *elem* до множини *set*.

set.remove(elem) – видаляє елемент *elem* з множини *set*. Генерує *KeyError*, якщо такого елемента не існує.

set.discard(elem) – видаляє з множини *set* елемент *elem*, якщо він знаходиться в множині. За відсутності такого елемента не генерує помилку.

set.pop() – видаляє перший елемент з множини *set*. Оскільки множини не впорядковані, не можна точно сказати, який елемент буде першим.

set.clear() – очищує множину *set* (спорожнює її).

Крім звичайних множин у мові програмування Python існують також так звані незмінні множини (*frozenset*). Єдина відмінність *set* від *frozenset* полягає в тому, що *set* – змінюваний тип даних, а *frozenset* – ні.

3.8. Тип словник (dict)

Тип даних словник – це невпорядкована колекція довільних об'єктів з доступом по ключу. Тобто певні ключі пов'язані з певною інформацією. Ключ повинен бути унікальним. Пари ключ-значення не впорядковані. Тому, якщо вам необхідний деякий порядок, необхідно самотужки впорядкувати словник перед зверненням до нього. Також необхідно зауважити, що в словниках в якості ключів можуть використовуватися тільки незмінні об'єкти [5,15,16]. Задати словник можна наступним чином:

```
personal_information={ 'last name': 'Петренко',
                       'first name': 'Василь',
                       'date of Birth':'12.05.70'}
```

Для того, щоб отримати потрібну інформацію із словника необхідно ввести запис:

ім'я_словника[ключ]

Наприклад:

```
print('Прізвище - ', personal_information['last name'])
print("Ім`я - ", personal_information['first name'])
print('Дата народження - ', personal_information['date of Birth'])
```

В результаті на екрані з'явиться повідомлення:

```
Прізвище -  Петренко
Ім`я -  Василь
Дата народження -  12.05.70
```

Спроба отримати інформацію за неіснуючим ключем приведе до генерації помилки.

Додати інформацію до словника можна наступним чином:

ім'я_словника[новий_ключ]=нове_значення

Наприклад:

```
#додаємо новий ключ і пов'язану з ним інформацію
personal_information['profession']='інженер'
#а тепер виведемо на екран вміст словника
for i in personal_information:
   print(i,' - ',personal_information[i])
```

На екрані з'явиться повідомлення:

```
last name  -  Петренко
first name  -  Василь
date of Birth  -  12.05.70
profession  -  інженер
```

Якщо замість нового ключа вказати існуючий, то буде перезаписана (змінено) інформація, пов'язана з цим ключем.

Як і всі типи даних мови програмування Python тип даних словник є об'єктом і має ряд власних методів:

dict.clear() – спорожнює словник.

dict.copy() – повертає як результат копію словника.

dict.fromkeys(послідовність_ключів [, послідовність_значень]) – створює словник з ключами, які містить *послідовність_ключів* і значеннями, які містить *послідовність_значень* (за замовчуванням *None*).

dict.get(key [, default]) – повертає значення, яке відповідає заданому ключу. Якщо такий ключ відсутній, то не формує виняткову ситуацію, а повертає значення задане у змінній *default* (за замовчуванням *None*).

dict.items() – повертає перелік ключів і відповідних їм значень у вигляді – (ключ, значення).

dict.keys() - повертає перелік ключів словника.

dict.pop(key [, default]) – видаляє ключ зі словника і повертає як результат пов'язане з ним значення. Якщо такого ключа немає, то повертає значення змінної *default*. Якщо змінна *default* не задана, то генерує виняткову ситуацію.

dict.popitem() – видаляє зі словника і повертає як результат пару (ключ, значення). Якщо словник порожній, формує виняткову ситуацію *KeyError*.

dict.setdefault(key [, default]) – повертає інформацію зв’язану із заданим ключем. Якщо такого ключа не існує, то не генерує виняткову ситуацію, а створює ключ із значенням *default* (за замовчуванням *None*).

dict.update([other]) – оновлює словник, додаючи пари (ключ, значення) задані у змінній *other*. Існуючі ключі перезаписує.

dict.values() – повертає інформацію, яка міститься у словнику (без ключів лише інформацію).

3.9. Типи bytes, bytearray і memoryview

У мові програмування Python існують два основних вбудованих типи двійкових послідовностей: незмінний тип *bytes*, який з'явився в Python 3, і змінюваний тип *bytearray*. Кожен елемент типу *bytes* або *bytearray* це – ціле число від 0 до 255. Даний тип використовується, як правило для запису інформації у файли і зчитування інформації з файлів.

Обидва типи *bytes* і *bytearray* підтримують всі методи типу *str* крім тих, що відносяться до форматування (*format, format_map*), і ще декількох, які безпосередньо залежать від особливостей Unicode, в тому числі *casefold*, *isdecimal*, *isidentifier*, *isnumeric*, *isprintable* і *encode*. При роботі з двійковими послідовностями можна використовувати такі рядкові методи, як: *endswith*, *replace*, *strip*, *translate*, *upper* і десятками інших, тільки аргументи повинні мати тип *bytes*, а не *str*. До двійкових послідовностей можна застосовувати функції для роботи з регулярними виразами з модуля *re*, якщо регулярний вираз відкомпільований з двійкової послідовності, а не з *str*.

Розглянемо приклади:

```
#створимо байт
b'bytes'
#створимо рядок байтів ‘байти’ у кодуванні utf-8
'Байти'.encode('utf-8')
# результат – b'\xd0\x91\xd0\xb0\xd0\xb9\xd1\x82\xd0\xb8'
#створимо рядок байтів із числового списку
bytes([50, 100, 76, 72, 41]) # результат – b'2dLH)'
#кожне число перетворилося на символ (як при використанні функції chr)
#перетворимо bytearray у рядок символів з кодуванням utf-8
b'\xd0\x91\xd0\xb0\xd0\xb9\xd1\x82\xd0\xb8'.decode('utf-8')
#результат – 'Байти'
#створиму масив байтів із рядка за допомогою функції bytearray
#в даному випадку символи повинні належати до множини ASCII
b = bytearray(b'hello world!')
```

Тип *memoryview* – один із типів для роботи з бінарними даними. Об'єкти *memoryview* дозволяють Python-коду звертатися до внутрішніх даних об'єкта, який підтримує *protocol buffer*, і працювати з цими даними без створення додаткової копії (*protocol buffer* описує взаємодію об'єктів). Розглянемо тип *memoryview* детальніше.

Методи:

memoryview(obj) – дозволяє створити об’єкт *memoryview* для доступу до об’єкту *obj*.

tobytes() – повертає у вигляді рядка дані, які розміщені в буфері.

tolist() – повертає у вигляді списку дані, які розміщені в буфері.

Властивості:

format – повертає рядок, який містить формат для кожного елемента у представленні.

itemsize – повертає розмір у байтах кожного елемента у представленні.

ndim – повертає ціле число, яке вказує, скільки вимірів багатовимірного масиву представляє пам'ять.

лише для читання

readonly – повертає булеве значення, яке вказує, статус атрибуту *readonly*.

shape – повертає кортеж цілих чисел довжиною *ndim*, надаючи форму пам'яті як N-мірний масив.

strides – повертає кортеж цілих чисел довжиною *ndim*, надаючи розмір у байтах, щоб отримати доступ до кожного елемента для кожного виміру масиву.

Розглянемо приклад:

```
#створимо масив байтів
random_byte_array = bytearray('ABC', 'utf-8')
#створимо об'єкт memoryview для роботи з ним
mv = memoryview(random_byte_array)
print('елемент списку з індексом 0 - ',mv[0])
print('діапазон байтів з індексами 0...2 - ',bytes(mv[0:2]))
print(list(mv[0:3]))
a=mv.tobytes()#a=b'ABC'
print('a = ',a)
b=mv.tolist() #b=[65, 66, 67]
print('b = ',b)
#модифікуємо елемент з індексом 1
mv[1] = 90
print('після модифікації - ', random_byte_array)
print('формат об`єкта random_byte_array - ',mv.format)
print('розмір елементів об`єкта random_byte_array - ',mv.itemsize)
print('кількість вимірів об`єкта random_byte_array - ',mv.ndim)
print('статус атрибута redonly об`єкта random_byte_array - ',mv.readonly)
print('shape об`єкта random_byte_array - ',mv.shape)
print('strides об`єкта random_byte_array - ',mv.strides)
```

В результаті на екран буде виведено наступні повідомлення:

```
елемент списку з індексом 0 -  65
діапазон байтів з індексами 0...2 -  b'AB'
[65, 66, 67]
a =  b'ABC'
b =  [65, 66, 67]
після модифікації -  bytearray(b'AZC')
формат об`єкта random_byte_array -  B
розмір елементів об`єкта random_byte_array -  1
кількість вимірів об`єкта random_byte_array -  1
статус атрибута redonly об`єкта random_byte_array -  False
shape об`єкта random_byte_array -  (3,)
strides об`єкта random_byte_array -  (1,)
```

Контрольні запитання

1. Перерахуйте основні вбудовані типи мови Python.
2. Опишіть тип *int* мови Python.
3. Опишіть тип *float* мови Python.
4. Опишіть тип *complex* мови Python.
6. Опишіть булевський тип мови Python.
7. Опишіть рядковий тип мови Python.
8. Опишіть тип список мови Python.
9. Опишіть тип кортеж мови Python.
10. Опишіть тип діапазон мови Python.
11. Опишіть тип множина мови Python.
12. Опишіть тип словник мови Python.
13. Опишіть тип *bytes* мови Python.
14. Опишіть тип *bytearray* мови Python.
15. Опишіть тип *memoryview* мови Python.

4. ОБРОБКА ВИНЯТКОВИХ СИТУАЦІЙ У МОВІ PYTHON

У будь-якій, програмою можуть виникати помилки. Помилки можуть бути синтаксичні або помилки виконання.

Синтаксичні помилки легко виявляються у процесі інтерпретації (компіляції) програми. При наявності синтаксичних помилок програму не вдасться запустити.

Помилки виконання проявляються на етапі виконання програми. Такі помилки можуть призводити як до повної непрацездатності програми так і до неправильної роботи програми. Часто такі помилки досить важко виявити, оскілки вони проявляються лише при певних умовах.

У мові програмування Python прийнято називати помилками лише синтаксичні помилки. Всі інші помилки прийнято називати винятками (винятковими ситуаціями). При виникненні виняткової ситуації інтерпретатором мови програмування Python генерується спеціальне повідомлення, яке може бути опрацьоване програмою. За замовчуванням при виникненні виняткової ситуації інтерпретатор просто виводить на екран відповідне повідомлення і завершує виконання програми. Але у мові Python є засоби, які дають змогу програмісту запрограмувати іншу (більш адекватну) реакцію на виникнення виняткової ситуації. Для цього у мові програмування Python передбачено спеціальний оператор, який дозволяє перехопити повідомлення, ідентифікувати його і визначити дії, необхідні для ліквідації наслідків виникнення виняткової ситуації:

try:
    оператори, які можуть викликати генерацію виняткової ситуації
except ситуація_1:
    оператори, які виключення ситуація_1
except ситуація_2:
    оператори, які виключення ситуація_2
…
else :
    оператори, які виконуються за відсутності виняткової ситуації
finally :
    оператори, які завжди виконуються вкінці блоку обробки ситуацій

Розділів *except* може бути кілька (кожен з цих розділів може обробляти певну ситуацію) або це може бути один розділ на всі випадки. У даному операторі завжди повинен бути як мінімум один розділ *except*.

Розділи *else* та *finally* можуть бути відсутні. Наведемо приклад:

```
try:
    a=int(input('введіть ціле число ='))
    x=1/a
except ValueError:
    print('Це не число!!!')
except ZeroDivisionError :
   print('Недопустима операція! Ділення на нуль!')
else:
    print('Помилки відсутні!  x = ', x)
finally:
    print('Кінець блоку')
```

У мові програмування Python передбачено велику кількість повідомлень про виняткові ситуації. Розглянемо деякі з них [20, 21].

ArithmeticError – виникає при будь-якій арифметичній помилці.

AssertionError – виникає, коли вираз у функції *assert* помилковий.

AttributeError – генерується при зверненні до атрибута об’єкту, якщо такий атрибут у об’єкта відсутній.

BaseException – генерується при виникненні будь-якої виняткової ситуації.

BlockingIOError – виникає, коли операція блокує об'єкт (наприклад, *сокет*), встановлений для неблокуючої операції.

BrokenPipeError – виникає при спробі записати інформацію у зачинений канал або сокет.

BufferError – виникає, коли пов'язана з буфером операція не може бути виконана.

ChildProcessError – виникає при невдалій операції з дочірнім процесом.

EOFError – функція натрапила на кінець файлу і не змогла виконати операцію зчитування.

ConnectionAbortedError – виникає, коли спробу з’єднання перервано.

ConnectionError – виникає при будь-якій помилці, пов'язаній з підключеннями.

ConnectionRefusedError – виникає, коли спробу з’єднання відхилено.

ConnectionResetError – виникає, коли з’єднання пере запущене приймаючою стороною.

Exception – генерується при виникненні будь-якої виняткової ситуації.

FileExistsError – виникає при спробі створення файлу або директорії, які вже існують.

FileNotFoundError – виникає, коли шуканий файл або директорія не існують.

FloatingPointError – породжується при невдалому виконанні операції з плаваючою крапкою.

GeneratorExit – породжується при виклику методу *close* об'єкта *generator*.

InterruptedError – виникає, коли системний виклик перервано запитом на виконання переривання.

IOError – генерується при виникненні будь-якої помилки введення-виведення.

ImportError – виникає, коли оператор *import* не може знайти файл, який повинен бути імпортований.

IndexError – виникає, коли індекс знаходиться поза межами допустимого діапазону.

IsADirectoryError – виникає, коли заданий файл виявився директорією.

KeyError – виникає, коли ключ (*dictionary key*) не знайдений в наборі існуючих ключів.

KeyboardInterrupt – виникає, коли користувач натискає кнопку переривання процесу (зазвичай *Delete* або *Ctrl + C*).

LookupError – виникає, коли задано некоректний індекс або ключ.

MemoryError – генерується у разі, коли недостатньо пам'яті.

NameError – виникає, коли не знайдено локальне чи глобальне ім'я.

NotADirectoryError – виникає, коли задана директорія виявляється файлом.

OSError - виникає, коли функція отримує помилку пов'язану з системою.

PermissionError – виникає, коли не вистачає прав доступу.

ProcessLookupError – виникає, коли зазначеного процесу не існує.

StopIteration – породжується вбудованою функцією *next*, якщо в об’єкті *Імператор* більше немає елементів.

SystemExit – породжується функцією sys.exit при виході з програми.

TimeoutError – виникає, закінчився час очікування.

TypeError – генерується, коли операція або функція застосовується до об'єкту невідповідного типу.

UnboundLocalError – у функції зроблено посилання на неіснуючу локальну змінну.

ValueError - виникає, коли вбудована операція або функція отримує аргумент, тип якого правильний, але неправильно значення. Наприклад коли за допомогою функції int(*рядок*) намагаються перетворити на ціле число рядок, який не може бути перетворений на ціле число.

ZeroDivisionError – виникає при спробі ділення на нуль.

Повний список виняткових ситуацій, можна знайти за посиланням [22].

Контрольні запитання

1. Яким чином у Python обробляють помилки і виняткові ситуації? Наведіть приклади.
2. Опишіть синтаксис оператора *try*. Наведіть приклади.
3. Наведіть кілька прикладів виняткових ситуацій. В яких випадках вони виникають?

5. ОПЕРАЦІЇ ВВЕДЕННЯ-ВИВЕДЕННЯ, РОБОТА З ФАЙЛАМИ

Досить часто в процесі роботи з програмою необхідно обмінюватись з нею даними, тобто передавати дані від користувача у програму та із програми користувачу. Один з найпоширеніших способів вивести дані в Python – це надрукувати їх в консолі. Для виведення даних в консоль використовується функція *print*. Функція *print* має наступний синтаксис [23]:

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

де

objects – об'єкт, який потрібно вивести * позначає, що об'єктів може бути кілька;

sep – символ, який розділяє об'єкти (за замовчуванням це – пробіл);

end – уставиться після всіх об'єктів;

file – потік з методом *write(string)*, у який проводиться запис, за замовчуванням використовується файл *sys.stdout* (консоль);

flush – визначає режим букерування запису (буферизований чи ні) якщо задано значення *True*, потік примусово записується безпосередньо у файл (без проміжного буфера). значення за замовчуванням: *False*.

Розглянемо приклади:

```
#запишемо у файл python.txt рядок - "Доброго дня!"
#для цього відкриємо файл, запишемо рядок і закриємо файл
sFile = open('python.txt', 'w')
print("Доброго дня!", file = sFile)
sourceFile.close()
#а тепер виведемо той же рядок на екран
#потік пов'язаний з консоллю відкривати не треба
#він відкривається автоматично при запуску програми
print("Доброго дня!")
```

Для зчитування даних з клавіатури використовують функцію *input()*. Синтаксис даної функції наступний [24]:

input([prompt])

Дана функція виводить на екран повідомлення *prompt* і очікує введення даних. Повертає введені дані у вигляді рядка, який необхідно перетворити на потрібний тип даних за допомогою відповідної функції. Наприклад:

```
my_str = input('Введіть ціле цисло ')
my_num= int(my_str)
```

У розглянутому вище прикладі показано, як можна використати функцію *print* для запису у файл. Однак є інший спосіб, який більше відповідає логіці мови програмування Python. Це – використання класу файл. Оскільки у мові Python всі дані є об'єктами, то і файли реалізовано, як об'єкти, які мають свої властивості і методи [25]. Розглянемо файловий об'єкт. Для роботи з файлом спочатку необхідно створити файловий об'єкт. Для цього використовують спеціальну функцію – *open(file, mode)* .Дана функція створює файловий об'єкт, зв'язує його з файлом вказаним у змінній *file* і задає режим доступу до файлу, вказаний у змінній *mode*. Як результат функція повертає створений файловий об'єкт. При роботі з файлами доступні наступні режими доступу:

r – файл відкривається для читання. Якщо файл не знайдений, то генерується виключення *FileNotFoundError*.

w – файл відкривається для запису. Якщо файл відсутній, то він створюється. Якщо подібний файл вже є, то він створюється заново, і відповідно старі дані в ньому стираються.

a – файл відкривається для доповнення. Якщо файл відсутній, то він створюється. Якщо подібний файл вже є, то дані записуються в його кінець.

b – використовується для роботи з бінарними файлами. Застосовується разом з іншими режимами – *w* або *r*.

Після створення файлу можна використовувати його методи і властивості.

Методи:

file.read() – зчитує вміст файлу у рядок;

file.readline() – зчитує файл по рядках;

file.readlines() – зчитує рядки файлу і створює список із зчитаних рядків;

file.write(string) – записує заданий рядок у відкритий файл. Рядок може містити не тільки текст, але і бінарні дані. При запису метод *write()* не додає символ кінця рядка у кінець файлу;

file.writelines(list) – записує елементи списку *list* у файл. Де будуть вставлені елементи списку, залежить від режиму доступу до файлу, якщо "*a*" то вони будуть вставлені в поточну позицію файлу (за замовчуванням в кінець файлу). Якщо "*w*", то вміст файлу буде видалено до того, як елементи списку будуть вставлені в поточну позицію файлу (за замовчуванням у позицію 0).

file.close() – закриває файл. Файл стає недоступним для операцій читання-записування;

file.flush() – ініціює запис даних із буфера безпосередньо у файл.

Властивості:

file.closed – повертає *True* якщо файл був закритий;

file.mode – повертає режим доступу, з яким був відкритий файл;

file.name – повертає ім'я файлу;

Наведемо приклад:

```
f = open("prob_file.txt", "w")
print('Відкрито файл - ',f.name)
f.write('Всім привіт!')
print('У файл записано:')
print('Всім привіт!')
f.flush()
f.close()
print('Файл закрито!\n\n')
f = open("prob_file.txt", "r")
print('Відкрито файл - ',f.name)
print('Режим доступу до файлу - ',f.mode)
print('З файлу прочитано:')
print(f.read())
f.close()
print('Файл закрито!\n\n')
```

На екрані з’явиться наступний текст:

Відкрито файл - prob_file.txt
У файл записано:
Всім привіт!
Файл закрито!

Відкрито файл - prob_file.txt
Режим доступу до файлу - r
З файлу прочитано:
Всім привіт!
Файл закрито!

Контрольні запитання

1. Який синтаксис функції *print*? Як її використовують? Наведіть приклади.

2. Який синтаксис функції *input*? Як її використовують? Наведіть приклади.

3. Опишіть клас файл. Як його використовують? Наведіть приклади.

6. ООП У МОВІ PYTHON

6.1. Базові поняття

Як уже було зазначено раніше Python – це об'єктно-орієнтована мова програмування. Основною ідеєю об'єктно-орієнтованого програмування (ООП) є об'єднання в єдине ціле даних і функціоналу для їх обробки в певний єдиний тип даних. Такий комбінований тип даних називається класом.

ООП базується на чотирьох основних поняттях: абстрагування, інкапсуляція, успадкування і поліморфізм.

Абстрагування – це виділення самих істотних характеристик об'єкта, які відрізняють його від всіх інших об'єктів і чітко визначають його з точки зору подальшого розгляду та аналізу.

Інкапсуляція – це об’єднання в одному класі даних і функцій для їх обробки.

Успадкуванням називається властивість класів створювати нові (породжені) класи на основі вже існуючих (базових). Породжені класи успадковують властивості базових класів. При цьому успадковані властивості можуть бути модифіковані. Крім того у породжені класи можуть бути добавлені нові властивості.

Поліморфізм – це властивість споріднених класів, породжених від загального базового класу, вирішувати схожі за змістом задачі різними способами.

В програмі можна створювати змінні, які є екземплярами того чи іншого класу. Такі екземпляри прийнято називати об’єктами класу. Кожен об’єкт повторює структуру свого класу, тобто має такі ж змінні (поля) і такі ж функції (методи). Змінні касу називають полями, а функції класу – методами. Поля і методи разом називають атрибутами класу. Поля можуть бути двох типів: вони можуть локальними для кожного конкретного екземпляру класу, а можуть бути загальними для всіх екземплярів даного класу. Їх називають змінними екземпляру і змінними класу відповідно.

6.2. Створення власного класу

6.2.1. Створення власного класу без успадкування

При програмуванні можна використовувати стандартні класи, які є складовою частиною мови програмування Python, а можна створювати свої власні класи. Створити клас можна за допомогою ключового слова *class*:

```
#визначаємо клас з іменем point
class point:
#поля класу – координати крапки у просторі за замовчуванням (0,0,0)
   x=0
   y=0
   z=0
#методи класу:
#get_coord – дозволяє прочитати координати
   def get_coord(self):
      return [self.x, self.y, self.z]
#set_coord – дозволяє встановити координати
   def set_coord(self,x,y,z):
       self.x=x
       self.y=y
       self.z=z
```

Створимо екземпляр класу *point* і попрацюємо з ним:

```
a=point()
#переконаємось, що за замовчуванням координати крапки (0,0,0)
a.get_coord()
```

На екрані з’явиться повідомлення:

```
[0, 0, 0]
```

А тепер змінимо координати і виведемо їх значення на екран:

```
a.set_coord(1,2,3)
a.get_coord()
```

На екрані з’явиться повідомлення:

```
[1, 2, 3]
```

Зверніть увагу на те, що кожен метод має аргумент *self*, який є покажчиком на конкретний екземпляр класу. Він є обов’язковим для всіх методів класу. Це аналог покажчика *this* із мови C++. При визові методу його

вказувати не треба, транслятор підставить його автоматично. Також зверніть увагу на те, що дякуючи покажчику *self* не виникає ніякого конфлікту між внутрішніми змінними об'єкту (*x, y, z*) і параметрами методу *set_coord* з такими самими іменами.

6.2.2. Створення власного класу на базі вже існуючого. Композиція та успадкування

Створення власних класів на базі вже існуючих може значно прискорити процес розробки програми. У мові програмування Python для реалізації власних класів на базі вже існуючих є два механізми: композиція і успадкування.

Композиція – це коли до складу створюваного класу входять екземпляри інших класів. Використовувані класи повинні бути визначені в програмі до їх першого використання. Наприклад можна створити клас *section* (відрізок) на основі композиції двох екземплярів класу *point*:

```
class section():
   p1=point()
   p2=point()
   def get_coords(self):
      return [self.p1.get_coord(), self.p2.get_coord()]
   def set_coords(self,x1,y1,z1,x2,y2,z2):
      self.p1.set_coord(x1,y1,z1)
      self.p2.set_coord(x2,y2,z2)
   def length(self):
      tmp=self.get_coords()
      len=(tmp[0][0]-tmp[1][0])**2+(tmp[0][1]-tmp[1][1])**2+(tmp[0][2]-
tmp[1][2])**2
      return len**(0.5)
```

При успадкуванні клас будується на основі базового класу наслідуючи його властивості. При цьому є можливість модифікувати успадковані властивості відповідно до потреб створюваного класу.

Наведемо приклад. Побудуємо на базі класу *point*, який моделює точку у тривимірному просторі, клас *four_dimentional_point*, моделює точку у

чотиривимірному просторі, де четверта координата, це час. При розробці класу використаємо механізм успадкування.

```
#задаємо ім'я класу – four_dimentional_point,
#а в дужках вказуємо базовий клас point
#новий клас успадкував структуру базового класу
class four_dimentional_point(point):
#додамо ще одну координату до структури нового класу
   t=0
#оскільки змінилась кількість координат,
#нам необхідно модифікувати методи,
   def get_coord(self):
      return [self.x, self.y, self.z, self.t]
   def set_coord(self,x,y,z,t):
      self.x=x
      self.y=y
      self.z=z
      self.t=t
```

Класс *four_dimentional_point* можна реалізувати дещо по іншому. У попередньому прикладі методи даного класу було повністю переписано. Якщо методи дуже великі і складні це може бути недоцільно. Тоді викликати аналогічний метод базового класу, а потім виконати додаткові дії.

```
class four_dimentional_point(point):
   t=0
#даний метод реалізуємо так само, як і в попередньому прикладі
   def get_coord(self):
      return [self.x, self.y, self.z, self.t]
#а цей метод реалізуємо через виклик
#аналогічного методу базового класу
   def set_coord(self,x,y,z,t):
      point.set_coord(self,x, y, z)
      self.t=t
```

У мові програмування Python можливе множинне наслідування від кількох класів одночасно. В цьому випадку їх імена перераховуються в дужках через кому:

```
class person:
   name='unknown'
class adress:
```

```
    adress='unknown'
class full_inform(person, adress):
pass
```

Тепер виконаємо кілька команд :

```
s=full_inform()
s.name # на екрані зявиться напис – 'unknown'
s.adress # на екрані зявиться напис – 'unknown'
```

6.2.3. Ініціалізація об'єктів. Метод __init__

При використанні ООП часто виникає ситуація, коли необхідно виконати деякі обов'язкові при створенні об'єкту. Для реалізації такої можливості у мові програмування Python передбачений метод *__init__*. Цей метод виконується першим при створенні об'єкту. Щоб скористатися цією можливістю достатньо при описі класу визначити метод *__init__*.

Модифікуємо наш клас *point* таким чином, щоб можна було створювати екземпляри не тільки із значенням координат за замовчуванням, але і з наперед заданими значеннями координат. Для цього введемо до класу метод *__init__* з аргументами за замовчуванням.

```
class point:
    def __init__(self,x=0,y=0,z=0):
        self.x=x
        self.y=y
        self.z=z
    def get_coord(self):
        return [self.x, self.y, self.z]
    def set_coord(self,x,y,z):
        self.x=x
        self.y=y
        self.z=z
```

Створимо екземпляр класу і виконаємо кілка команд:

```
#створюємо екземпляр класу із значеннями за замовчуванням
n=point()
#перевіримо значення координат
n.get_coord() #на екрані зявиться – [0, 0, 0]
#а тепер створимо екземпляр з координатами (1, 2, 3)
d=point(1, 2, 3)
```

```
# перевіримо значення координат
d.get_coord()#на екрані зявиться – [1, 2, 3]
```

А ось так можна, використовуючи метод *__init__*, модифікувати приклад з наслідування від двох класів із попереднього підрозділу:

```
class person:
   #name='unknown'
   def __init(self, name='unknown'):
      self.name=name

class adress:
   #adress='unknown'
   def __init(self, adress='unknown'):
      self.adress=adress

class full_inform(person, adress):
    def __init__(self,name='unknown', adress='unknown'):
      self.name=name
      self.adress=adress
```

Тепер можна створювати екземпляри класу *full_inform* із заданими значеннями полів:

```
s=full_inform('Василь Пупкін','Україна, Київ, вул. Хрещатик 26, кв. 7')
s.name #на екрані зявиться – 'Василь Пупкін'
s.adress #на екрані зявиться – 'Україна, Київ, вул. Хрещатик 26, кв. 7'
```

6.2.4. Поліморфізм.

Фактично поліморфізм – це можливість обробки різних типів даних (які належать до різних класів) за допомогою однієї функції, або методу. У попередньому підрозділі ми вже спостерігали поліморфізм між класами, пов'язаними успадкуванням. У кожного класу може бути, наприклад, своя функція ініціалізації. Крім того ми змінювали методи виведення на екран інформації про об'єкт. Все це приклади поліморфізму. Однак класи не обов'язково повинні бути пов'язані успадкуванням. Поліморфізм як один з ключових елементів ООП існує незалежно від успадкування. Класи можуть бути не родинними, але мати однакові методи, як в наступному прикладі [26]:

```
#визначимо два різних класи, які мають метод з однаковим імям – total
class T1:
    n=10
    def total(self, N):
        self.total = int(self.n) + int(N)

class T2:
    def total(self,s):
        self.total = len(str(s))
#створимо по екземпляру кожного з цих класів
t1 = T1()
t2 = T2()
#звернемось до методу total для кожного з об’єктів і порівняємо результат
t1.total(45)
t2.total(45)
print(t1.total) # на екрані зявиться – 55
print(t2.total) # на екрані зявиться – 2
```

Як бачимо результат різний для різних типів, що власне і є проявом поліморфізму. Поліморфізм дає можливість реалізовувати одноманітні інтерфейси для об'єктів різних класів. Наприклад, різні класи можуть передбачати різний спосіб виведення на екран інформації об'єктів. При цьому вони можуть мати однакову назву методу виведення на екран, що спрощує аналіз програми, робить код більш зрозумілим [26].

6.2.5. Перенавантаження операторів.

Одним з прикладів поліморфізму можуть слугувати математичні оператори, які на основі однакового інтерфейсу по різному обробляють цілі числа, числа з плаваючою крапкою та комплексні числа, що досягається за рахунок перенавантаження цих операторів.

Перенавантаження операторів і функцій – це ефективний засіб реалізації поліморфізму. Мова програмування Python має ряд засобів для спрощення процесу перенавантаження операторів і функцій.

Для прикладу спробуємо перенавантажити функцію *print* для відображення на екрані повідомлення про об’єкт створеного програмістом (не вбудованого в мову) класу[26].

```
#опишемо клас
class A:
    def __init__(self, v1, v2):
        self.field1 = v1
        self.field2 = v2
#створимо екземпляр класу
a = A(3, 4)
#спробуємо вивести його на екран за допомогою функції print
print(a)
```

На екрані з’явиться, щось на кшталт наведеного нижче напису:

<__main__.A object at 0x7f840c8acfd0>

Це нас не влаштовує ми хочемо отримати більш інформативне повідомлення. Тому перенавантажимо функцію *print*. Для цього треба додати до класу спеціальний метод *__str __()*. Цей метод буде формувати рядок, який буде виводити функція *print()*:

```
class A:
   def __init__(self, v1, v2):
      self.field1 = v1
      self.field2 = v2
   def __str__(self):
      return str(self.field1) + " " + str(self.field2)
```

А тепер створимо екземпляр класу і виведемо його на екран за допомогою функції *print()*:

```
a = A(3, 4)
print(a) # на екрані зявиться напис – (3 4)
```

Як бачимо, механізм перенавантаження у мові Python реалізовано інакше, ніж у C++. Оскільки у мові Python всі дані – це об’єкти, а всі функції – це методи, то для перенавантаження методів і операторів використовують спеціальні вбудовані у мову функції, які також є методами. Розглянемо деякі з них. Повний перелік можна знайти за посиланням [27].

__add__(self, other) – метод перенавантаження оператора складання. *x + y* викликає *x.__add__(y)*;

__and__(self, other) – викликається при виконанні операції побітове І (*x & y*);

__bool__(self) – викликається при перевірці істинності. Якщо цей метод не визначений, викликається метод __len__ (об'єкти, що мають ненульову довжину, вважаються дійсними);

__bytes __(self) – викликається функцією *bytes* при перетворенні до типу байт;

__call__(self [, args ...]) – здійснює виклик екземпляра класу як функції;

__contains__(self, item) – здійснює перевірку на приналежність елементу до контейнера (*item in self*);

__del__() – деструктор об'єктів класу, викликається при видаленні об'єктів;

__delattr__(self, name) – викликається при видаленні атрибута (*del obj.name*);

__delitem__(self, key) – викликається при видаленні елемента за індексом;

__divmod__(self, other) – здійснює обчислення частки і залишку від цілочислового ділення (*divmod(x, y)*);

__eq__(self, other) – викликається при виконанні операції *x == y* викликає метод *x.__eq__(y);*

__floordiv__(self, other) – викликається, коли має місце операція цілочисельного ділення (*x // y*);

__format__(self, format_spec) – використовується функцією *format* (а також методом *format* у рядків);

__ge__(self, other) – викликається при виконанні операції $x \geq y$ викликає метод *x.__ge__(y)*;

__getattr__(self, name) – викликається, коли атрибут реалізації класу не знайдений в звичайних місцях (наприклад, у примірника немає методу з такою назвою);

__getitem__(self, key) – викликається, коли має місце доступ за індексом (або за ключем);

__gt__(self, other) – викликається при виконанні операції *x > y* викликає метод *x.__gt__(y)*;

__hash__(self) – викликається при отриманні хеш-суми об'єкту, наприклад, для додавання в словник;

__init __() – конструктор об'єктів класу, викликається при створенні об'єктів;

__iter__(self) – повертає ітератор для контейнера;

__le__(self, other) – викликається при виконанні операції $x \leq y$ викликає метод *x.__le__(y)*;

__len__(self) – викликається при визначенні довжини об'єкта;

__lshift__(self, other) – викликається при виконанні операції бітовий зсув вліво (*x << y*);

__lt__(self, other) – викликається при виконанні операції $x < y$ викликає метод *x.__lt__ (y)*;

__mul__(self, other) – викликається при виконанні операції множення *(x * y)*;

__mod__(self, other) – викликається при виконанні операції залишок від ділення (*x% y*);

__ne__(self, other) – викликається при виконанні операції *x! = y* викликає метод *x.__ne__(y);*

__new __(cls [...]) – керує створенням екземпляру класу. В якості обов'язкової аргументу приймає клас. Повертає екземпляр класу для його подальшої його передачі методу *__init__*;

__or__(self, other) – викликається, коли має місце операція побітове АБО (*x | y*);

__pow__(self, other [, modulo]) – викликається, коли має місце операція піднесення до степеня (*x ** y*, *pow(x, y [, modulo])*);

__reversed__(self) – змінює порядок елементів ітератора на зворотній;

__rshift__(self, other) – викликається при виконанні операції бітовий зсув вправо (*x >> y*);

__setattr__(self, name, value) – викликається, коли атрибуту об'єкта виконується присвоювання;

__setitem__(self, key, value) – викликається, коли має місце призначення елемента за індексом;

__str__() – перетворення об'єкта до рядкового представлення, викликається, коли об'єкт передається функцій *print()* і *str()*;

__sub__(self, other) – викликається, коли має місце операція віднімання *(x - y)*;

__truediv__(self, other) – викликається, коли має місце операція ділення (*x / y*);

__xor__(self, other) – викликається, коли має місце операція побітове виключає Або (*x ^ y*);

6.2.6. Декоратори

В Python все є об'єктом. У цьому сенсі Python повністю відповідає ідеям ООП. Функції також є об’єктами. Це дає додаткові можливості для роботи з ними. Так, наприклад функції можна передавати в якості аргументів, присвоювати функції змінним. Так, наприклад, можна привласнити ім’я функції змінній і викликати функцію, звертаючись до даної змінної:

```
#нехай маємо функцію
def fun_hello(a='Світ'):
   print('Доброго дня, '+a+'!')
#запустимо функцію шляхом звернення до неї
fun_hello('студент') #на екрані зявиться напис: Доброго дня, студент!
#привласнимо ім’я функції змінній
hello=fun_hello
#запустимо функцію звернувшись до змінної
hello('студент') #на екрані зявиться напис: Доброго дня, студент!
```

Як бачимо результат однаковий в обох випадках. Можна також передавати функцію як аргумент в іншу функцію:

```
#спочатку визначимо функцію,
яка буде використовуватись як аргумент за замовчуванням
#def pass_fun():
   Pass #ця функція не виконує ніяких дій
#а тепер визначимо функцію,
#яка приймає іншу функцію як аргумент
def my_fun(a=pass_fun()):
   return a
```

#а тепер запустимо функцію *my_fun* без аргументу і з аргументом:

```
my_fun() #за замовчуванням функція не виконує ніяких дій
```

```
my_fun(fun_hello())#на екрані зявиться напис: Доброго дня, Світ!
my_fun(fun_hello('студент'))
#на екрані зявиться напис: Доброго дня, студент!
```

В даному прикладі функція *my_fun()* виконує роль обгортки для функції *fun_hello()*. Такі функції називають декораторами, оскільки вони маскують функцію, яку обгортають. В якості функції, яку обгортають може бути використана зовнішня функція (написана користувачем або бібліотечна), а може бути функція визначена всередині функції обгортки:

```
#визначимо функцію my_fun1
def my_fun1(a):
    #визначимо функцію inner_fun
    def inner_fun(b):
        return b**2
    return inner_fun(a)
#виконаємо функцію my_fun1 з аргументом 3
my_fun1(3) # на екрані з'явиться число 9
```

Основне призначення функцій декораторів полягає в тому, що вони дають нам можливість змінити поведінку функції, не змінюючи її код. Така можливість може бути корисною, наприклад, коли модифікації потребує бібліотечна функція, яку не бажано змінювати, оскільки вона використовується в багатьох інших програмах. Наведемо приклад:

```
#функція, поведінку якої треба змінити не міняючи код функції
def lib_fun():
    print('Я - бібліотечна функція! Мій код не бажано міняти!')
#створюємо функцію декоратор
def decor_fun(fun_for_decoration):
    def inner_fun():
        print('Доповнюємо додатковими діями поведінку функції lib_fun!')
        fun_for_decoration()
        print('А ми Ваш код не змінюємо, а декоруємо!')
    return inner_fun
#підмінюємо вихідну функцію декорованою
lib_fun=decor_fun(lib_fun)
#запускаємо модифіковану функцію
lib_fun()
```

Представлений приклад можна реалізувати і в інший спосіб:

```
#створюємо функцію декоратор
def decor_fun(fun_for_decoration):
   def inner_fun():
      print('Доповнюємо додатковими діями поведінку функції lib_fun!')
      fun_for_decoration()
      print('А ми Ваш код не змінюємо, а декоруємо!')
   return inner_fun
#декоруємо функцію
@decor_fun
def lib_fun():
   print('Я - бібліотечна функція! Мій код не бажано міняти!')
#запускаємо модифіковану функцію
lib_fun()
```

Можна використовувати кілька декораторів для однієї функції. Можна також декорувати не тільки самостійні функції, а й методи об'єктів. При цьому потрібно тільки не забувати про обов'язковий для методів аргумент *self*.

Контрольні запитання

1. Яка основна ідея ООП?
2. Що таке клас?
3. Які основні принципи ООП?
4. Поясніть, що означає термін абстрагування.
5. Поясніть, що означає термін інкапсуляція.
6. Поясніть, що означає термін успадкування.
7. Поясніть, що означає термін поліморфізм.
8. Поясніть, що означає термін об’єкт.
9. Поясніть, що означає термін атрибути класу.
10. Як створити клас користувача у мові програмування Python?
11. Для чого потрібен покажчик *self*?
12. Поясніть термін композиція.
13. Чим композиція відрізняється від успадкування?

14. Для чого використовують метод __*init*__?

15. Яким чином реалізується поліморфізм у мові Python?

16. Яким чином реалізовано у мові Python перенавантаження функцій?

17. Що таке декоратори? Для чого їх використовують?

7. ГРАФІЧНІ МОЖЛИВОСТІ МОВИ PYTHON

Майже всі сучасні програми використовують GUI (graphical user interface) – графічний інтерфейс користувача. Існує багато різних бібліотек для роботи з GUI. У мові програмування Python за замовчуванням використовується бібліотека *Tkinter*, яка є складовою частиною мови Python. Тому будемо розглядати саме цю бібліотеку. Оскільки бібліотека *Tkinter* має дуже велику кількість функцій, то за браком часу ми розглянемо лише невелику їх частину, необхідну для розуміння основних принципів роботи з цією бібліотекою.

7.1. Основи графічного інтерфейсу користувача

Графічний інтерфейс складається з елементів – вікон, кнопок, меню, тощо. Елементи графічного інтерфейсу називають віджетами. Для розуміння суті роботи з GUI необхідно запам'ятати, що додатки з графічним інтерфейсом користувача подієво-орієнтовані. Тобто кожен віджет пов'язаний з певною подією, на яку він реагує виконуючи запрограмовані дії. Події можуть бути різними: натиснута клавіша клавіатури або кнопка миші, переривання від таймера, тощо.

Подієво-орієнтоване програмування базується на об'єктно-орієнтованому і структурному. Навіть якщо ми не створюємо власних класів та об'єктів, то все-одно ними користуємося. Всі віджети – це об'єкти, породжені вбудованими класами.

Щоб написати GUI-програму, треба виконати наступні дії:

- створити головне вікно програми і конфігурувати його властивості;
- створити і конфігурувати потрібні віджети (задати їх властивості, задати спосіб їх розміщення в головному вікні і т.д.);
- визначити події, на які повинна реагувати програма;
- визначити обробники подій (набір дій виконуваних програмою при виникненні тієї чи іншої події);
- запустити цикл обробки подій.

Створення і конфігурація віджетів виконується за допомогою спеціальних функцій із використовуваної бібліотеки (у нашому випадку *Tkinter*). Розглянемо деякі з цих функцій [28].

tkinter.Tk() – створює основне вікно програми з конфігурацією за замовчуванням повертає посилання на екземпляр класу. Екземпляр класу можна конфігурувати за допомогою вбудованих методів класу:

geometry(string) – задає розмір вікна у пікселях. Розмір вказують у змінній *string* у вигляді рядка. Наприклад – *'400x250'*;

title(string) – задає заголовок вікна. Заголовок вказують у змінній *string* у вигляді рядка;

mainloop() – метод створеного вікна, який відображає головне вікно програми і запускає його обробник подій.

Button(window) – створює кнопку у заданому вікні. Повертає покажчик на екземпляр класу *Button*.

Таблиця 7.1. Властивості віджета *Button*

Властивість	**Значення**
width	Ширина кнопки у пікселях. За замовчуванням дорівнює такій кількості пікселів, щоб текст умістився в кнопці впритул до її меж.
height	Висота кнопки. За замовчуванням дорівнює такій кількості пікселів, щоб текст містився в кнопці впритул до її меж.
text	Текст на кнопці. За замовчуванням текст буде відображатися по центру кнопки. Можливо зробити багаторядковий текст, використовуючи \ n.
bg	Фон кнопки, який вона матиме в той час, коли на неї не натиснули.
fg	Колір тексту, який буде мати кнопка в той час, коли на неї не натиснули.
bd	Ширина межі кнопки, яка буде до її натискання.
activebackground	Колір фону (коли кнопка натиснена).
activeforeground	Колір тексу (коли кнопка натиснена).
disabledbackground	Колір фону (коли властивість *state = DISABLED*).

Таблиця 7.1. Властивості віджета *Button* (продовження)

Властивість	**Значення**
disabledforeground	Колір тексту (коли властивість *state = DISABLED*).
state	Стан кнопки (*NORMAL, DISABLED*). *NORMAL* – звичайний стан кнопки, при якому вона може взаємодіяти з користувачем. *DISABLED* – такий стан кнопки, при якому вона не може взаємодіяти з користувачем.
compound	Розташування картинки на кнопці (*CENTER, BOTTOM, LEFT, RIGHT, TOP*). За замовчуванням картинка на кнопці буде відображатися замість тексту. *BOTTOM* – картинка буде відображатися під текстом. *LEFT* – картинка буде відображатися ліворуч від тексту. *RIGHT* – картинка відображатиметься праворуч від тексту. *TOP* – картинка буде відображатися над текстом.
relief	Рельєф кнопки (*FLAT, GROOVE, RIDGE, SUNKEN, RAISED*).
overrelief	Рельєф кнопки коли над нею знаходиться курсор (*FLAT, GROOVE, RIDGE, SUNKEN, RAISED*).
justify	Вирівнювання тексту (*CENTER, RIGHT, LEFT*). За замовчуванням текст буде відображатися з вирівнюванням по лівому краю, але це можна змінити, використовуючи властивість *justify*. *CENTER* – текст вирівнюється на кнопці по центру. *LEFT* – текст вирівнюється на кнопці по лівому краю. *RIGHT* – текст вирівнюється на кнопці по правому краю.
image	Ім'я картинки, відображеної на кнопці.
font	Вид шрифту на кнопці. Властивість має мати вигляд: "ім'я_шрифта=розмір".

Radiobutton(window) – створює радіокнопку у заданому вікні. Повертає покажчик на екземпляр класу *Radiobutton*.

Таблиця 7.2. Властивості віджета *Radiobutton*

Властивість	**Значення**
width	Ширина радіо-кнопки.
height	Висота радіо-кнопки.
text	Текст на радіо-кнопці.
bg	Фон радіо-кнопки.
fg	Колір центу.
bd	Ширина межі радіо-кнопки.
activebackground	Колір фону (коли радіо-кнопка натиснена).
activeforeground	Колір тексту (коли радіо-кнопка натиснена).
disabledbackground	Колір фону (коли властивість *state = DISABLED*).
disabledforeground	Колір тексту (коли властивість *state = DISABLED*).
state	Стан радіо-кнопки (*NORMAL, DISABLED*).
compound	Розташування картинки на радіо-кнопці (*CENTER, BOTTOM, LEFT, RIGHT, TOP*).
justify	Вирівнювання тексту.
relief	Рельєф радіо-кнопки (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
overrelief	Рельєф радіо-кнопки коли над нею знаходиться курсор (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
image	Ім'я, яке буде відображено на радіо-кнопці.
selectimage	Ім'я, яке буде відображено на радіо-кнопці (коли вона обрана).
font	Вид шрифту на радіо-кнопці.
indicatoron	Стиль відображення радіо-кнопки (якщо *true*, то буде показуватися гурток поруч з радіо-кнопкою, інакше немає).
value	Значення, яке буде присвоюватися змінної, зазначеної в параметрі *variable* при виборі радіо-кнопки.
variable	Ім'я змінної, у якій буде зміняться значення на зазначене у властивості *value* при виборі радіо-кнопки.

Checkbutton(window) – створює кнопку-прапорець у заданому вікні. Повертає покажчик на екземпляр класу *Checkbutton*.

Таблиця 7.3. Властивості віджета *Checkbutton*

Властивість	**Значення**
width	Ширина прапорця.
height	Висота прапорця.
text	Текст поряд з прапорцем.
bg	Фон прапорця.
fg	Колір прапорця.
bd	Ширина кордону прапорця.
activebackground	Колір фону (коли прапорець обрано).
activeforeground	Колір тексту (коли прапорець обрано).
disabledbackground	Колір фону (коли властивість *state = DISABLED*).
disabledforeground	Колір тексту (коли властивість *state = DISABLED*).
state	Стан прапорця (*NORMAL, DISABLED*).
compound	Розташування картинки на прапорці (*CENTER, BOTTOM, LEFT, RIGHT, TOP*).
justify	Вирівнювання тексту.
relief	Рельєф прапорця (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
overrelief	Рельєф прапорця коли над ним знаходиться курсор (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
image	Ім'я, яке буде відображено на кнопці-прапорці.
selectimage	Ім'я, яке буде відображено на кнопці-прапорці (коли він обраний).
font	Вид шрифту на кнопці-прапорці.
indicatoron	Стиль відображення прапорця (якщо *true*, то буде показуватися гурток поруч з ф, інакше немає).
onvalue	Значення, яке буде присвоюватися змінній, зазначеній у параметрі *variable* при виборі кнопки-прапорця.
offvalue	Значення, яке буде присвоюватися змінній, зазначеній в параметрі *variable* при виборі другої (не даної) кнопки-прапорця.
variable	Ім'я змінної, у якій буде змінюватись значення на зазначене у властивості *value* при виборі кнопки-прапорця.

Entry(window) – створює однорядкове текстове поле. Повертає посилання на екземпляр класу *Entry*.

Таблиця 7.4. Властивості віджета *Entry*

Властивість	**Значення**
width	Ширина поля.
bg	Фон поля.
fg	Колір поля.
bd	Ширина межі поля.
activebackground	Колір фону (коли в поле набирають текст).
activeforeground	Колір тексту (коли в поле набирають текст).
disabledbackground	Колір фону (коли властивість *state = DISABLED*).
disabledforeground	Колір тексту (коли властивість *state = DISABLED*).
state	Стан поля (*NORMAL, DISABLED*).
justify	Вирівнювання тексту.
highlightcolor	Колір другої межі (коли поле має фокус).
highlightbackground	Колір другої межі (коли поле не має фокус).
highlightthickness	Ширина другої межі.
relief	Рельєф текстового поля (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
overrelief	Рельєф текстового поля коли над ним знаходиться курсор (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
font	Вид шрифту в однорядковому текстовому полі.
textvariable	Ім'я змінної, в якій зберігається весь текст, що знаходиться в полі.
selectbackground	Колір фону виділеного фрагмента тексту.
selectforeground	Колір виділеного фрагмента тексту.
insertontime	Час, який курсор видно.
insertofftime	Час, який курсор не видно.

Таблиця 7.5. Методи віджета *Entry*

Метод	**Виконувана дія**
a.get(початок, кінець)	Отримує фрагмент тексту від символу, позиція якого визначається першим параметром, до символу, позиція якого визначена другим параметром.
a.insert(позиція,текст)	Вставляє текст у поле перед символом, індекс якого вказаний як параметр позиції. (Нумерація йде з 0.)
a.delete(початок, кінець)	Видаляє текст від символу, позиція якого визначається першим параметром, до символу, позиція якого визначена другим параметром.

Text(window) – створює багаторядкове текстове поле. Повертає покажчик на екземпляр класу *Text*.

Таблиця 7.6. Властивості віджета *Text*

Властивість	**Значення**
width	Ширина поля.
height	Висота поля.
bg	Фон поля.
fg	Колір поля.
bd	Ширина межі поля.
activebackground	Колір фону (коли у полі набирають текст).
activeforeground	Колір тексту (коли у полі набирають текст).
disabledbackground	Колір фону (коли властивість *state = DISABLED*).
disabledforeground	Колір тексту (коли властивість *state = DISABLED*).
state	Стан поля (*NORMAL, DISABLED*).
justify	Вирівнювання тексту.
highlightcolor	Колір другої межі (коли поле має фокус).
highlightbackground	Колір другої межі (коли поле не має фокусу).
highlightthickness	Ширина другої межі.
relief	Рельєф текстового поля (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).

Таблиця 7.6. Властивості віджета *Text* (продовження)

Властивість	**Значення**
overrelief	Рельєф текстового поля коли над ним знаходиться курсор (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
font	Вид шрифту в багаторядковому текстовому полі.
selectbackground	Колір фону виділеного фрагмента тексту.
selectforeground	Колір тексту виділеного фрагмента тексту.
insertontime	Час, який курсор видно.
insertofftime	Час, який курсор не видно.

Таблиця 7.7. Методи віджета *Text*

Метод	**Виконувана дія**
a.get(начало,конец)	Отримує фрагмент тексту від символу, позиція якого визначається першим параметром, до символу, позиція якого визначена другим параметром. (Нумерація рядків йде з 1, а символів в рядку з 0.). Позиція повинна мати значення виду "%d.%d " % (x,y), або вона може описуватися одним і слів: *CURRENT* (поточна позиція курсору), *END* (позиція останнього символу в полі).
a.insert(позиция,текст)	Вставляє текст в поле перед символом, індекс якого вказаний як параметр позиції. (Нумерація рядків йде з 1, а символів у рядку з 0.). Позиція при вказівці повинна мати значення виду "% d.% d " % (x,y) або вона може описуватися одним і слів: *CURRENT* (поточна позиція курсору), *END* (позиція останнього символу в полі).
a.delete(начало,конец)	Видаляє текст від символу, позиція якого визначається першим параметром, до символу, позиція якого визначена другим параметром. (Нумерація рядків йде з 1, а символів у рядку з 0.). Позиція повинна мати значення виду "% d.% d " % (x,y) або вона може описуватися одним і слів: *CURRENT* (поточна позиція курсору), *END* (позиція останнього символу в полі).

Таблиця 7.7. Методи віджета *Text* (продовження)

Метод	**Виконувана дія**
a.tag_add(имя, начало, конец)	Додає тег (об'єкт, що дозволяє форматувати текст) до поля *a*. (Нумерація рядків йде з 1, а символів в рядку з 0.). Початок або кінець повинні мати значення виду "% d.% d "% (x,y) або вони можуть описуватися одним із слів: *CURRENT* (поточна позиція курсору), *END* (позиція останнього символу в полі).
a.tag_configure(имя, параметры)	Змінює властивості тегу. Текст в діапазоні, який був зазначений при створенні тегу буде змінюватися при зміні властивостей тегу. *background* – фон тексту *foreground* – колір тексту

Label(window) – створює мітку повертає покажчик на екземпляр віджета *Label*.

Таблиця 7.8. Властивості віджета *Label*

Властивість	**Значення**
width	Ширина мітки.
height	Висота мітки.
text	Текст на мітці.
bg	Фон мітки.
fg	Колір тексту.
bd	Ширина межі мітки.
activebackground	Колір фону (коли мітка у фокусі).
activeforeground	Колір тексту (коли мітка у фокусі).
disabledbackground	Колір фону (коли властивість *state = DISABLED*).
disabledforeground	Колір тексту (коли властивість *state = DISABLED*).
state	Стан мітки (*NORMAL, DISABLED*).
compound	Розташування картинки на мітці (*CENTER, BOTTOM, LEFT, RIGHT, TOP*).
justify	Вирівнювання тексту.

Таблиця 7.8. Властивості віджета *Label* (продовження)

Властивість	**Значення**
relief	Рельєф мітки (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
overrelief	Рельєф мітки коли над нею знаходиться курсор (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
image	Ім'я, яке буде відображено на мітці.
font	Вид шрифту на мітці.
textvariable	Ім'я змінної, у якій буде зберігатися текст на мітці.

Scale(window) – створює у вікні повзунок. Повертає покажчик на екземпляр віджету *Scale*.

Таблиця 7.9. Властивості віджета *Scale*

Властивість	**Значення**
width	Ширина повзунка.
bg	Фон повзунка.
fg	Колір тексту поруч з повзунком.
bd	Ширина межі повзунка.
activebackground	Колір фону (коли повзунок рухають або клацають на нього).
activeforeground	Колір тексту (коли повзунок рухають або клацають на нього).
disabledbackground	Колір фону (коли властивість *state = DISABLED*).
disabledforeground	Колір тексту (коли властивість *state = DISABLED*).
state	Стан повзунка (*NORMAL, DISABLED*).
justify	Вирівнювання тексту.
highlightcolor	Колір другої межі (коли повзунок має фокус).
highlightbackground	Колір другий кордону (коли повзунок не має фокус).
highlightthickness	Ширина другої межі.
relief	Рельєф повзунка (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).

Таблиця 7.9. Властивості віджета *Scale* (продовження)

Властивість	**Значення**
overrelief	Рельєф повзунка коли над ним знаходиться курсор (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
font	Вид шрифту у розмітки повзунка.
*from_*	Перше число, яке відзначається рядом з повзунком.
to	останнє число, яке відзначається рядом з повзунком.
tickinterval	Відстань між діленнями (у пікселях).
resolution	Мінімальна кількість пікселів, на яку користувач може пересунути повзунок.
orient	Орієнтація смуги з повзунком (*HORIZONTAL, VERTICAL*).
length	Довжина лінії, якою рухається повзунок.

Scrollbar(window) – створює полосу прокруттки вікна. Повертає покажчик на екземпляр віджета *Scrollbar*.

Таблиця 7.10. Властивості віджета *Scrollbar*

Властивість	**Значення**
bg	Фон смуги прокрутки.
activebackground	Колір фону (коли смугу прокрутки рухають).
disabledbackground	Колір фону (коли властивість *state = DISABLED*).
state	Стан поля (*NORMAL, DISABLED*).
relief	Рельєф смуги прокрутки (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
overrelief	Рельєф смуги прокрутки коли над ним знаходиться курсор (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
command	Ім'я віджета, зміст якого буде прокручуватися. Доступні лише значення віджет.xview (віджет буде прокручуватися горизонтально) та віджет.yview (віджет буде прокручуватися вертикально).

Frame(window) – створює фрейм вікна. Повертає екземпляр віджета *Frame*.

Таблиця 7.11. Властивості віджета *Frame*

Свойство	**Значение**
width	Ширина фрейму.
height	Висота фрейму.
bg	Фон фрейму.
activebackground	Колір фону (коли властивість *state = ENABLED*).
disabledbackground	Колір фону (коли властивість *state = DISABLED*).
state	Стан фрейму (*NORMAL, DISABLED*).
relief	Рельєф межі фрейму (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
highlightcolor	Колір другої межі (коли фрейм має фокус).
highlightbackground	Колір другої межі (коли фрейм не має фокусу).
highlightthickness	Ширина другої межі.

LabelFrame(window) – створює фрейм з міткою. Повертає покажчик на екземпляр віджета *LabelFrame*.

Таблиця 7.12. Властивості віджета *LabelFrame*

Властивість	**Значення**
width	Ширина фрейму.
height	Висота фрейму.
bg	Фон фрейму.
activebackground	Колір фону (коли *state = ENSABLED*).
disabledbackground	Колір фону (коли *state = DISABLED*).
state	Стан фрейма (*NORMAL, DISABLED*).
relief	Рельєф межі фрейму (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
highlightcolor	Колір другої межі (коли фрейм має фокус).
highlightbackground	Колір другої межі (коли фрейм не має фокус).
highlightthickness	Ширина другої межі.
text	Заголовок фрейму.

Listbox(window) – створює список. Повертає екземпляр віджета *Listbox*.

Таблиця 7.13. Властивості віджета *Listbox*

Свойство	**Значение**
width	Ширина списку.
height	Висота списку.
bg	Фон списку.
disabledforeground	Колір фону (коли властивість state = DISABLED).
state	Стан списку (*NORMAL, DISABLED*).
relief	Рельєф межі списку (*FLAT, GROOVE, RIDGE, SUNKEN, RAISE*).
highlightcolor	Колір другої межі (коли список має фокус).
highlightbackground	Колір другої межі (коли список не має фокусу).
highlightthickness	Ширина другої межі.
selectbackground	Колір фону виділених елементів списку.
selectforeground	Колір тексту виділених елементів списку.
text	Заголовок списку.

Таблиця 7.14. Методи віджета *Listbox*

Метод	**Виконувана дія**
a.get(початок, кінець)	Отримує фрагмент списку з назв елементів списку. У фрагмент потраплять елементи, індекси яких, знаходяться в діапазоні, міх першим і останнім його параметрами.
a.insert(позиція, назва_елементу)	Вставляє елемент у список після даного індексу. (Нумерація йде з 0.)
a.delete(початок, кінець)	Видаляє зі списку всі елементи, індекси яких, потрапляють в діапазон індексів початок…кінець,

Menu(window) – створює меню. Повертає екземпляр віджету *Menu*.

add_command(label='Назва пункту',command=виконувана функція)- додає пункт у меню.

add_cascade(label='Назва', menu=pop_menu) – додає підменю у меню.

window.config(menu=name_menu) – відображає вказане меню.

Розглянемо приклад:

```
#імпортуємо бібліотеку tkinter
import tkinter

#створюємо обробник події вибору пункту менюexit_item (вихід з програми)
def exit_fun():
   window.destroy()
#створюємо обробник події натискання кнопки exit_about
#(закрити вікноabout_win )
def about_command():
   about_win=tkinter.Tk()
   about_win.title('Про програму')
   about_win.geometry('200x100')
   #створюємо фрейм
   about_frame = tkinter.Frame(about_win)
   #about_frame.grig(columnspan=1, rowspan=3)
   about_frame.pack()
   #створюємо і конфігуруємо текст у вікні колір шрифта чорний
   about_message=tkinter.Label(about_frame,  fg='black')
   about_message.justify='CENTER'
   about_message.config(text='Тестова програма. Версія 1.0')
   about_message.pack(side='top')
   #створюємо і конфігуруємо кнопку закриття вікна 'Про програму'
   exit_about=tkinter.Button(about_frame,                    text='Вихід',
command=about_win.destroy)
   exit_about.pack(side='right')
   #запускаємо цикл обробки подій вікна about_win
   about_win .mainloop()
#створюємо і конфігуруємо головне вікно програми
window = tkinter.Tk()
window.geometry('800x400')
window.title('Перша GUI програма!')
#створюємо головне меню
main_menu=tkinter.Menu(window)
#додаємо у головне меню підменю
pop_menu=tkinter.Menu(main_menu)
#додаємо у   підменю пункт і зв'язуємо його з командою
exit_item=pop_menu.add_command(label='Вихід',command=exit_fun)
#додаємо у  головне меню пункт з яким зв'язане підменю
main_menu.add_cascade(label='Основне', menu=pop_menu)
#додаємо у  головне меню пункт
main_menu.add_command(label='Про програму',command=about_command)
#відображаємо головне меню у вікні
window.config(menu=main_menu)
```

```
#створюємо фрейм
frame = tkinter.Frame(window)
frame.pack()
#запускаємо цикл обробки подій головного вікна
window.mainloop()
```

Рис. 7.1. Вікно простої графічної програми на мові Python

Після запуску цієї програми зявиться вікно зображение на рис. 7.1. Якщо обрати пункт меню *'Про програму'* то відкриється вікно з коротким описом програми (рис. 7.2), яке можна закрити натиснувши за допомогою маніпулятора “мишки” кнопку *'Вихід'*. Вийти з програми можна можна обравши з меню *'Основне'* пункт *'Вихід'*.

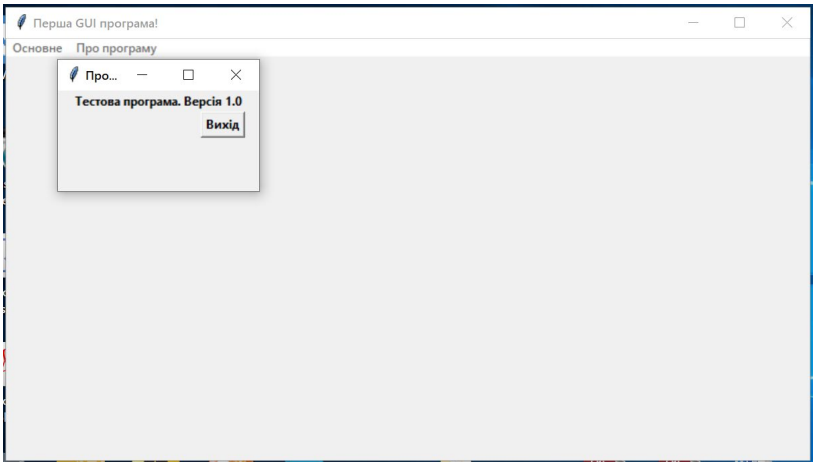

Рис. 7.2. Вікно програми після вибору пункту меню *'Про програму'*

7.2. Графічні примітиви. Побудова графіків функцій

Окрім можливості розробляти програми з графічним інтерфейсом користувача біліотека *Tkinter* надає можливість користувачеві створення простих малюнків та графіків функцій. Для цієї мети у бібліотеці *Tkinter* передбачено віджет *Canvas* (полотно).

Canvas(window) – створює полотно для малювання у заданому вікні *window*, повертає покажчик віджет *Canvas*.

Таблиця 7.15. Властивості віджета *Canvas*

Властивість	**Значення**
width	Ширина полотна для малювання (у пікселях).
height	Висота полотна для малювання (у пікселях).
bd	Ширина межі полотна для малювання (у пікселях).
state	Стан полотна для малювання (NORMAL, DISABLED).
highlightcolor	Колір межі (коли на полотні малюють).

Таблиця 7.16. Методи віджета *Canvas*

Метод	**Виконувана дія**
a.create_line(x,y,x1,y1)	Малює на полотні *a* лінію між точками (*x, y*) і (*x1, y1*).
a.create_rectangle(x,y,x1,y1)	Малює на полотні *a* прямокутник з верхньою лівою вершиною (*x, y*) і нижньою правою (*x1, y1*).
a.create_oval(x,y,x1,y1)	Малює на полотні *a* овал в межах прямокутника з верхньою лівою вершиною (*x, y*) і нижньою правою (*x1, y1*).
a.create_polygon(x,y, ... ,xn,yn)	Малює на полотні *a* ламану по точках (*x, y*) ... (*xn, yn*).
a.create_text(x,y,x1,y1)	Малює на полотні *a* текст, центруючи його по точці (*x, y*) і (*x1, y1*).

При використанні віджета *Canvas* необхідно пам’ятати, що координата *x* зростає зліва на право, а *y* – зверху до низу. Розглянемо простий приклад на малювання примітивів.

```
#імпортуємо бібліотеку
import tkinter
#створюємо головне вікно
root = tkinter.Tk()
root.geometry('600x400')
#створюємо і розміщуємо у вікні полотно
a=tkinter.Canvas(root,width=500,height=300,bg="lightblue")
a.pack()
##створюємо і розміщуємо у вікні кнопку 'Вихід'
exit_about=tkinter.Button(root, text='Вихід', command=root.destroy)
exit_about.pack()
#малюємо на полотні прямокутник
rect=a.create_rectangle(50,50,150,100,fill="white",outline="blue")
#малюємо на полотні еліпс
oval_1=a.create_oval(200,200,300,250,fill="green",outline="black")
#запускаємо обробник подій головного вікна
root.mainloop()
```

А тепер напишемо програму яка малюватиме графіки функцій однієї змінної (лише ті з функцій, які входять безпосередньо до мови Python і які не потрібно імпортувати).

```
from math import *
from tkinter import *

f = input('f(x):')
x_min=float(input('x_min='))
x_max=float(input('x_max='))
step=(x_max-x_min)/100
print('step=',step)
x=list()
y=list()
for i in range(0,101,1):
    x.append(x_min+i*step)
    fun= f.replace('x', str(x[i]))
    y.append(eval(fun))
print('x[0]=',x[0],' y[0]=',y[0])
print('x[100]=',x[100],' y[100]=',y[100])
y_min=min(y)
y_max=max(y)
y_step=(y_max-y_min)/5
```

```
mid_x_p=300
mid_y_p=200+10
max_x_p=600
max_y_p=400+10
#600x400
x_p_step=600/(x_max-x_min)
print('x_p_step=',x_p_step)
y_p_step=400/(y_max-y_min)
print('y_p_step=',y_p_step)

root = Tk()
#задаємо канву у головному вікні розмір 600x400 фон білий
canv = Canvas(root, width = 600, height = 420, bg = "white")
#задаємо вертикальну вісь
canv.create_line(mid_x_p,max_y_p,mid_x_p,10,width=2,arrow=LAST)
#задаємо горизонтальну вісь
canv.create_line(0,mid_y_p,max_x_p,mid_y_p,width=2,arrow=LAST)
#задаємо мітки по осях
for i in range(0,6,1):
    #задаємо мітки по осі x
    if((i!=0)and(i!=5)):
        canv.create_text(i*max_x_p/5, mid_y_p-10, text =
str(int(x[20*i]*100)/100), fill="purple", font=("Helvectica", "10"))
    #задаємо мітки по осі y
    canv.create_text(mid_x_p+20, max_y_p-i*(max_y_p-10)/5, text =
str(int((y_min+i*y_step)*100)/100), fill="purple", font=("Helvectica", "10"))

for i in range(0,100,1):
    #малюємо графік лініями
    canv.create_line((x[i]-x_min)*x_p_step,(y_max-
y[i])*y_p_step+10,(x[i+1]-x_min)*x_p_step,(y_max-
y[i+1])*y_p_step+10,width=1)
#пакуємо канву з графіком
canv.pack()
#запускаємо обробник подій вікна
root.mainloop()
```

Результат роботи програми наведено на рисунку 7.3.

У даному розділі було розглянуто лише базові можливості бібліотеки *Tkinter*. Більш детально можна її розглянути за посиланням [28].

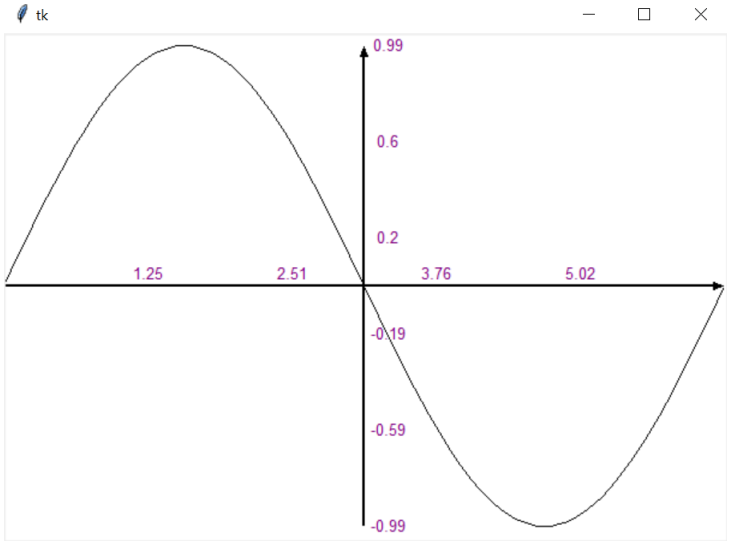

Рис. 7.3. Вікно програми з графіком функції sin(x) ($0<x<2\pi$)

Контрольні запитання

1. Що таке графічний інтерфейс користувача? Яким чином він реалізований у мові програмування Python?
2. Які дії необхідно виконати для реалізації програми з графічним інтерфейсом користувача?
3. Опишіть клас *Tk*.
4. Опишіть віджет *Button*.
5. Опишіть віджет *Radiobutton*.
6. Опишіть віджет *Checkbutton*.
7. Опишіть віджет *Entry*.
8. Опишіть віджет *Text*.
9. Опишіть віджет *Label*.
10. Опишіть віджет *Scale*.
11. Опишіть віджет *Scrollbar*.
12. Опишіть віджет *Frame*.
13. Опишіть віджет *LabelFrame*.
14. Опишіть віджет *Listbox*.
15. Опишіть віджет *Canvas*.

8. ПОТОКИ І ПРОЦЕСИ У МОВІ PYTHON

8.1. Базові поняття

З появою багатоядерних процесорів зявилась можливість розподіляти навантаження на всі доступні ядра. Натепер існує два основні підходи в розподілі навантаження: використання процесів і потоків. Використання декількох процесів фактично означає використання декількох програм, які виконуються незалежно один від одного. Такий підхід створює великі незручності в управлінні обміну даними між цими програмами. В якості альтернативи існує інший підхід – створення багатопоточних програм. Обмін даними між потоками значно простіший ніж між процесами, але програма значно ускладнюється, оскільки потоки є частиною процесу і використовують ресурси процесу [29].

Багатопотокове програмування (Multithreading) має свої переваги і недоліки [29, 30].

Переваги багатопотокового програмування:

- Multithreading може значно підвищити швидкість обчислень в багатопроцесорних або багатоядерних системах, оскільки кожен процесор або ядро обробляє окремий потік одночасно;
- всі потоки процесу мають доступ до його глобальних змінних. Якщо глобальна змінна змінюється в одному потоці, її видно і іншим потокам. Потік також може мати свої власні локальні змінні.

Недоліки багатопотокового програмування:

- у однопроцессорній системі багатопотоковість не тільки не підвищує продуктивності програми, а може навіть знизити її через накладні витрати на управління потоками;
- при багатопотоковому програмуванні необхідна синхронізація потоків, щоб уникнути конфліктів між потоками при доступі до загальних ресурсів процесу. Це веде до збільшення завантаження пам'яті і процесора, а також веде до ускладнення програми і значного ускладнення налагодження програми.

Розглянемо можливості багатопотокового програмування в мові Python. Перше, що треба врахувати – це те, потоки Python не реалізують реального паралелізму. Інтерпретатор мови Python використовує внутрішній глобальний блокувальник (GIL), який дозволяє виконуватися тільки одному потоку у даний момент часу. Це зводить нанівець переваги багатоядерної архітектури процесорів. Для багатопоточних додатків, які працюють в основному на дискові операції читання / запису, це не має особливого значення, а для додатків, які ділять процесорний час між потоками, це є серйозним обмеженням.

8.2. Клас *Thread*

Стандартна бібліотека Python надає засоби для багатопотокового програмування, і одне з таких засобів - модуль *threading*. Всі потоки, породжувані в програмі – це класи, успадковані від класу *Thread*, визначеного в модулі threading [29-33]. Розглянемо методи і властивості класу *Thread*.

Таблиця 8.1. Методи класу *Thread*

Метод	**Виконувана дія**
run():	Функція точки входу для потоку.
start():	Запускає потік при виклику методу *run*.
join([time]):	Дозволяє програмі очікувати завершення потоку.
isAlive():	Перевіряє активний потік.
getName():	Повертає ім'я потоку.
setName():	Встановлює ім'я потоку..

Властивості:

Таблиця 8.2. Властивості класу *Thread*

Властивість	**Значення**
name	Назва потоку, яка використовується лише для ідентифікації. Не має семантики. Кілька потоків можуть мати однакову назву. Початкову назву встановлює конструктор.
ident	Містить 'ідентифікатор потоку' (ненулеве ціле число) для даного потоку або *None*, якщо потік не був запущений.
native_id	Власний цілісний ідентифікатор цього потоку. Це невід'ємне ціле число або *None*, якщо потік не був запущений. Його значення може бути використане для унікальної ідентифікації конкретного потоку (поки потік не завершується).
daemon	Містить булеве значення, яке вказує у якому режимі запущено потік (*True* – потік запущено у режимі сервісу, *False* – у звичайному режимі). Це потрібно встановити до виклику методу *start()*. Його початкове значення успадковується від створюючго потоку; головний потік не може бути запущений у режимі сервісу, і тому для всіх потоків, створених в головному потоці, за замовчуванням *daemon = False*.

Розглянемо приклад паралельної роботи двох потоків.

```
#імпортуємо модуль threading
import threading
#створюємо власний потоковий клас
#екземпляр даного класу виводить на екран два рази
#повідомлення про себе, потім повідомлення про завершення
#після чого завершує роботу
class myThread(threading.Thread):
#визначення конструктора класу
   def __init__(self, name, counter):
      threading.Thread.__init__(self)
      self.threadID = counter
      self.name = name
      self.counter = counter
```

```
#визначення методу run
   def run(self):
      for i in range(1,3,1):
         print("Працює " + self.name+"!\n")
      else:
         print("Завершено " + self.name+"!\n")
#створюємо два екземпляри нашого класу
thread1 = myThread("Потік №1 ", 1)
thread2 = myThread("Потік №2", 2)
#запускаємо їх
thread1.start()
thread2.start()
#запускаємо метод join, щоб програма не завершилась раніше
#ніж відпрацюють потоки
thread1.join()
thread2.join()
#виводимо повідомлення про завершення програми
print("Програму завершено!!!")
```

В результаті виконання програми на екрані зявиться повідомлення:

```
Працює Потік №1 !
Працює Потік №2!

Працює Потік №1 !
Працює Потік №2!

Завершено Потік №1 !
Завершено Потік №2!

Програму завершено!!!
```

З результатів роботи програми видно, що процесорний час розподіляється між потоками, що і треба було продемонструвати.

8.3. Керування потоками

Разом з потоками з'являється проблема синхронізації даних. Якщо кілька потоків використовують спільні ресурси, то можлива ситуація одночасного звернення кількох потоків до якогось із ресурсів. Це може призвести до небажаних наслідків. Наприклад, якщо один потік записує дані

певну область пам'яті, а інший потік читає і обробляє ці дані, то можлива ситуація, коли перший потік ще не закінчив запис даних, а другий потік вже прочитав їх і обробляє неповний набір даних. Для запобігання подібних ситуацій у модулі *threading* є відповідні засоби. Для цього у модулі *threading* передбачено клас *Lock*, який має методи: *acquire()* і *release()*. Як зрозуміло з назви дані методи відповідно встановлюють і знімають режим блокування потоку.

Для прикладу роботи з класом *Lock()* додамо в попередню програму режим блокування потоку.

```
#імпортуємо модуль threading
import threading
#створюємо власний потоковий клас
#екземпляр даного класу виводить на екран два рази
#повідомлення про себе, потім повідомлення про завершення
#після чого завершує роботу
class myThread(threading.Thread):
#визначення конструктора класу
   def __init__(self, name, counter):
      threading.Thread.__init__(self)
      self.threadID = counter
      self.name = name
      self.counter = counter

   def run(self):
      #включаємо режим блокування потоку
       threadLock.acquire()
       for i in range(1,3,1):
          print("Працює " + self.name+"!\n")
       else:
          print("Завершено " + self.name+"!\n")
      #виключаємо режим блокування потоку
       threadLock.release()

#створюємо екземпляр класу Lock() це необхідно зробити
#до запуску потоків
threadLock = threading.Lock()
#створюємо два екземпляри нашого класу
thread1 = myThread("Потік №1 ", 1)
thread2 = myThread("Потік №2", 2)
#запускаємо їх
```

```
thread1.start()
thread2.start()
#запускаємо метод join, щоб програма не завершилась раніше
#ніж відпрацюють потоки
thread1.join()
thread2.join()
#виводимо повідомлення про завершення програми
print("Програму завершено!!!")
```

На екрані побачимо наступне:

```
Працює Потік №1 !
Працює Потік №1 !
Завершено Потік №1 !

Працює Потік №2!
Працює Потік №2!
Завершено Потік №2!

Програму завершено!!!
```

Як бачимо картина змінилася корінним чином. Тепер Потік №2 не може отримати доступ до консолі, поки не завершив роботу Потік №1.

В даному посібнику ми розглянули лише найпростіші засоби модуля *threading* більш глибоко ознайомитись з засобами модуля *threading* можна за посиланнями [33,34].

8.4. Керування процесами

Як вже було зазначено раніше у мові Python є обмеження – GIL (Global Interpreter Lock), яке ніколи не дозволяє використовувати кілька ядер CPU одночасно, і тому в Python немає справжньої багатопотоковості.

Використовуючи кілька незалежних процесів можна ефективно обійти обмеження GIL [35], оскільки, використовуючи кілька процесів ми використовуємо кілька екземплярів GIL. У зв'язку з цим немає обмежень на виконання байт-коду одного потоку одночасно. Тому є сенс розглянути можливість розподілу навантаження шляхом використання кількох незалежних процесів.

У звязку з цим виникає питання, а які є шляхи створення процесів програмою, написаною на мові Python. Цього можна досягти трьома шляхами [35]:

- використовуючи системний виклик *fork()*;
- використовуючи можливості модуля *multiprocessing*;
- використовуючи.

Одразу відзначимо, що системний виклик *fork()* і механізм *Forkserver* використовуються лише в Unix - системах, тому ми їх розглядати не будемо. Розглянемо лише використання деяких з методів модуля *multiprocessing*.

Породження процесу означає створення нового процесу батьківським процесом. Батьківський процес продовжує своє виконання асинхронно або очікує, поки дочірній процес не завершить своє виконання. Для породження процесу і запуску необхідно виконати наступні кроки:

- імпортувати модуль *multiprocessing*;
- запустити процес, викликавши метод *start()*;
- запустити метод *join()*.

Розглянемо приклад створення і запуску трьох потоків [35].

```
#імпортуємо модуль multiprocessing
import multiprocessing
#задаємо функцію, яку виконуватимуть створені процеси
def spawn_process(i):
  print ('Це процес: %s' %i)
  return

if __name__ == '__main__':
   Process_jobs = []
   for i in range(3):
       #створюємо новий процес
       p = multiprocessing.Process(target = spawn_process, args = (i,))
       #додаємо процес до списку процесів
       Process_jobs.append(p)
       #запускаємо процес
       p.start()
       p.join()
```

Результат роботи програми буде наступним:

Це процес: 0
Це процес: 1
Це процес: 2

Але, на відміну від аналогічної програми для кількох потоків (розглянутої у попередньому розділі), це повідомлення відобразиться не в консолі запущеної Вами програми, а в системній консолі Python, оскільки породжені процеси є самостійними процесами, а не потоками програми, яка їх запустила.

Модуль *multiprocessing* мови Python дозволяє також запускати процес у фоновому режимі. Щоб виконати процес у фоновому режимі, нам потрібно встановити значення *true* для прапорця фонового режиму. Фоновий процес буде продовжувати працювати до тих пір, поки виконується основний процес, і буде зупинений після завершення свого виконання або коли основна програма буде знищена.

```
import multiprocessing
import time


def daemon_spawn_process(i):
   for z in range(0,5):
       print ('Це - фоновий процес: %s' %i)
       time.sleep(1)

if __name__ == '__main__':
   p = multiprocessing.Process(target = daemon_spawn_process, args = (1,))
   p.daemon=True
   p.start()
print ('Програму завершено')
```

У системній консолі відобразиться повідомлення:

Програму завершено
Це - фоновий процес: 1
Це - фоновий процес: 1
Це - фоновий процес: 1
Це - фоновий процес: 1
Це - фоновий процес: 1

За необхідності процес можна негайно завершити за допомогою методу *terminate*():

```
import multiprocessing
import time

def daemon_spawn_process(i):
   for z in range(0,10):
       print ('Це - фоновий процес: %s' %i)
       time.sleep(1)

if __name__ == '__main__':
   p = multiprocessing.Process(target = daemon_spawn_process, args = (1,))
   p.daemon=True
   p.start()
   time.sleep(2)
   p.terminate()
```

У даному прикладі задано вивести на консоль 10 раз повідомлення про фоновий процес, але повідомлення на системній консолі буде не повне, оскільки процес буде перервано.

8.5. Обмін даними між процесами

Часто необхідно реалізувати обмін даними між процесами. Мова програмування Python дозволяє реалізувати обмін даними між процесами декількома способами [36]. Розглянемо два з них, а саме використання черги (клас *Queue*) і використання каналу (клас *Pipe*).

Queue – це структура даних, яка надає механізм взаємодії потоків між процесами за принципом FIFO, що означає першим прийшов - першим пішов (першим обслужений). Суть використання черги полягає у тому, що повідомлення записуються у чергу у тому порядку у якому поступають і зчитати їх можна лише у тому ж порядку. Записати чи зчитати повідомлення може буд-який процес, який звернеться до черги. Щоб проілюструвати цей принцип розглянемо приклад, у якому три процеси послідовно записують повідомлення у чергу, а потім три інших процеси почергово зчитують ці повідомлення:

```
from multiprocessing import Process, Queue
import queue
import random
def put_to_q(q,i):
    q.put([42, None, 'Повідомлення-'+str(i)])
    print("Я процес p"+str(i)+". Я поставив у чергу повідомлення:[42, None,'Повідомлення-'"+str(i),"]")


def get_from_q(q,i):
    a=q.get()
    print("Я процес g"+str(i)+". Я отримав із черги повідомлення:",a)

def main():
   q = Queue()
   put0 = Process(target = put_to_q, args = (q,0))
   put1 = Process(target = put_to_q, args = (q,1))
   put2 = Process(target = put_to_q, args = (q,2))
   put0.start()
   put1.start()
   put2.start()

   get0 = Process(target = get_from_q, args = (q,0))
   get1 = Process(target = get_from_q, args = (q,1))
   get2 = Process(target = get_from_q, args = (q,2))
   get0.start()
   get1.start()
   get2.start()
if __name__ == '__main__':
   main()
```

Після виконання даного коду на системній консолі з’являться повідомлення:

```
Я процес p0. Я поставив у чергу повідомлення:[42, None,'Повідомлення-'0 ]
Я процес p1. Я поставив у чергу повідомлення:[42, None,'Повідомлення-'1 ]
Я процес g0. Я отримав із черги повідомлення: [42, None, 'Повідомлення-0']
Я процес p2. Я поставив у чергу повідомлення:[42, None,'Повідомлення-'2 ]
Я процес g1. Я отримав із черги повідомлення: [42, None, 'Повідомлення-1']
Я процес g2. Я отримав із черги повідомлення: [42, None, 'Повідомлення-2']
```

Порядок у якому висвічуються повідомлення може бути будь-яким, але кожен процес отримав повідомлення згідно черги. Перше повідомлення

отримав той процесс, який першим читав з черги, а останнє – той, який читав з черги останнім.

Як уже було відзначено другий спосіб обміну даними між процесами це – використання каналу. Канал це – складна структура даних, яка поєднує два конкретних процеси. Вона складається з пари об'єктів з'єднання, які представляють два кінця каналу. Кожен з цих об'єктів має два методи - *send()* і *recv()* для взаємодії між процесами. Метод *send()* призначений для запису даних у канал, а метод *recv()* для зчитування даних з каналу. За замовчуванням канал може працювати у дуплексному режимі, тобто потік інформації може одночасно рухатись у двох напрямках. Розглянемо приклад, у якому один процес посилає повідомлення у канал, а другий зчитує це повідомлення з каналу.

```
from multiprocessing import Process, Pipe

def put(conn):
  conn.send([42, None, 'Привіт!'])
  print("Я процесс p! Я надіслав до каналу:"+"[42, None, 'Привіт!']")
  conn.close()


def get(conn):
  print("Я процесс g! Я прочитав з каналу:", conn.recv())
  conn.close()


if __name__ == '__main__':
  parent_conn, child_conn = Pipe()
  p = Process(target = put, args = (child_conn,))
  p.start()
  g = Process(target = get, args = (parent_conn,))
  g.start()
  p.join()
  g.join()
```

Після запуску програми на системній консолі зявляться наступні повідомлення:

Я процесс p! Я надіслав до каналу:[42, None, 'Привіт!']

Я процесс g! Я прочитав з каналу: [42, None, 'Привіт!']

Контрольні запитання

1. Поясніть, що таке потік у програмуванні.
2. Перерахуйте переваги і недоліки багатопотокового програмування.
3. В чому полягає основний недолік реалізації багатопоточності у Python?
4. Опишіть клас *Thread*.
5. Опишіть клас *Lock*.
6. Що таке процес?
7. Які переваги дає використання кількох процесів?
8. Який модуль містить методи для роботи з процесами?
9. Які кроки необхідно виконати для породження і запуску процесу?
10. Які режими виконання процесів підтримує мова програмування Python?
11. Що таке фоновий процес?
12. Як можна програмно завершити дочірній процес?
13. Які способи обміну даними між процесами розглянуто в нашому курсі?
14. Який принцип дії черги?
15. Яким чином працює канал?

9. МЕРЕЖЕВЕ ПРОГРАМУВАННЯ МОВОЮ PYTHON

Мережеве програмування це – подальший розвиток ідеології взаємодії між процесами. Мережеві програми також базуються на обміні інформацією між процесами. Відмінність у тому, що процеси можуть знаходитись на різних комп'ютерах, що накладає на процес обміну деякі особливості.

Натепер основною технологією взаємодії процесів у мережі є технологія клієнт-сервер. Клієнт і сервер взаємодію один з одним в мережі за допомогою різних мережевих протоколів, таких як IP протокол, HTTP протокол, FTP та інші. Функції клієнта і сервера значно відрізняються. Програма-клієнт формує запити до сервера, а програма-сервер обробляє запити і, за необхідності, повертає клієнту відповідь. Ініціатором сеансу зв'язку завжди є клієнт.

Така модель має свої переваги і недоліки. Основною перевагою моделі взаємодії клієнт-сервер є те, що програмні коди клієнтського і серверного додатків розділені, що дозволяє знизити вимоги до машин клієнтів, оскільки більша частина обчислювальних операцій проводиться на сервері. Крім того архітектура клієнт-сервер досить гнучка і дозволяє адміністратору зробити локальну мережу більш захищеною.

До недоліків моделі взаємодії клієнт-сервер можна віднести те, що вартість серверного обладнання дуже висока, сервер повинна обслуговувати спеціально навчена і підготовлена людина. Якщо в локальній мережі стає виходить з ладу сервер, то вся мережа стає непрацездатною.

Для ідентифікації клієнтських і серверних програм використовують унікальну адресу-сокет (*socket*), яка складається з IP-адреси і номера порту. IP-адреса ідентифікує комп'ютер, на якому запущено програму, а номер порту ідентифікує саму програму на комп'ютері, оскільки на одному комп'ютері може бути запущено кілька мережевих додатків одночасно.

У мові програмування є спеціальні засоби для реалізації мережевих програм. Розглянемо спрощені приклади серверної і клієнтської [4].

Програма-сервер:

```
#програма сервер запускати на різних консолях з програмою клієнт
#для цього запускаємо, наприклад два Total Comander
#у кожному Total Comander запускаємо свій інтерпретатор Python (Anaconda)
#У одному інтерпретаторі запускаємо сервер, а у іншому - клієнт

import socket # підключаємо бібліотечний модуль для взаємодії у мережі

# задаємо IP-адресу для клієнт-серверного обміну на одному ПК
HOST = '127.0.0.1'
print('IP-адреса: ',HOST)
PORT = 50007      # задаємо порт
print('ПОРТ: ',PORT)
print('\n')
# створюємо сокет
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# прив'язуємо сокет до IP-адреси (bind)   і порту
s.bind((HOST, PORT))
# сервер у циклі прослуховує сокет  (listen), чекаючи з'єднання з клієнтом
# коли поступає запрос від клієнта, то створює з'єднання accept(),
#отримує дані recv(), надсилає отримані дані назад клієнту send()
#і знову прослуховує сокет
while 1:
   print('Чекаю на запит від клієнта!')
   s.listen(1)
   conn, addr = s.accept()
   print('Є запит! Встановив з`єднання!')
   data = conn.recv(1024) # отримує дані від клієнта (не більше  1024 байт)
   print('Отримую дані!')
   print('Отримав від клієнта: ', data.decode('utf-8'))
   conn.send(data)
   print('Надіслав у відповідь: ', data.decode('utf-8'))
   print('\n')
```

Програма-клієнт:

```
#програма клієнт  запускати на різних консолях з програмою сервер
#для цього запускаємо, раприклад два Total Comander
#у кожному Total Comander запускаємо свій інтерпретатор Python (Anaconda)
#У одному інтерпретаторі запускаємо сервер, а у іншому - клієнт
import socket
HOST = '127.0.0.1' # IP-адреса сервера
```

```
PORT = 50007 # порт сервера
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# клієнт встановлює з'єднання з сервером:
s.connect((HOST, PORT))
data = 'Доброго дня!'
# обмін по мережі відбувається у форматі bytes, тому повідомлення перед
# передаванням до серверу, перетворюємо у формат utf-8 :
s.send(data.encode('utf-8'))
print('Надіслав серверу: ', data)
# отримання даних від сервера:
data = s.recv(1024)
s.close()
print('Отримав у відповідь: ', data.decode('utf-8'))
```

Для перевірки роботи програм необхідно, щоб клієнт і сервер запускалися в різних примірниках IDLE, тобто IDLE необхідно запустити два рази і в окремому вікні спочатку запустити програму-сервер, а потім в іншому вікні запустити програму-клієнта.

Першим запускаємо програму сервер. Після запуску у вікні IDLE зявиться повідомлення:

IP-адреса: 127.0.0.1
ПОРТ: 50007

Чекаю на запит від клієнта!

Сервер готовий до роботи. Тепер запускаємо програму-клієнт. Як тільки відбудеться зєд’нання, то одразу з’явиться повідомлення:

Надіслав серверу: Доброго дня!
Отримав у відповідь: Доброго дня!

Сервер також відреагує на зєднання повідомивши:

Є запит! Встановив з`єднання!
Отримую дані!
Отримав від клієнта: Доброго дня!
Надіслав у відповідь: Доброго дня!

Чекаю на запит від клієнта!

Програма клієнт виконала запит і завершила роботу, а програма сервер залишилась запущеною, чекаючи на запит від іншого клієнта.

Як бачимо все досить просто. Звичайно реальні програми більш складні, оскільки там необхідно виконувати більш складні дії, робити ряд перевірок тощо. Однак сам механізм взаємодії клієнта і сервера залишається незмінним. Більш детальну інформацію можна отримати за посиланнями [4, 37, 38].

Контрольні запитання

1. Що означає термін технологія клієнт-сервер?
2. Які переваги технології клієнт-сервер?
3. Які недоліки технології клієнт-сервер?
4. Які функції виконує програма сервер?
5. Які функції виконує програма клієнт?
6. Що таке сокет?
7. Для чого використовують IP-адресу?
8. Для чого використовують номер порту?

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. https://uk.wikipedia.org/wiki/Python
2. http://files.swaroopch.com/python/byte_of_python.pdf
3. Мусин Д. Самоучитель Python [текст] / Д. Мусин. , 2017. – 154 с.
4. Федоров, Д. Ю. Программирование на языке высокого уровня Python : учеб. пособие для прикладного бакалавриата / Д. Ю. Федоров. — 2-е изд., перераб. и доп. — М. : Издательство Юрайт, 2019. — 161 с.
5. Мельник І.В. Основи програмування на мові Python. Том 1. Базові принципи побудови мови програмування Python та головні синтаксичні конструкції: комплексний навч. Посібник з курсів «Об'єктно-орієнтоване програмування» та «обчислювальні системи та мережі» для студентів-бакалаврів, які навчаються за освітньою програмою «Електронні прилади та пристрої» - К., Кафедра, 2020. – 372 с. ISBN 978-617-7301-71-3 (заг.) ISBN 978-617-7301-72-0 (Кн. 1)
6. https://tproger.ru/translations/python-args-and-kwargs/
7. https://younglinux.info/python/modules
8. https://pythoner.name/documentation/library/functions
9. https://pythonworld.ru/moduli/modul-math.html
10. https://pythonworld.ru/moduli/modul-sys.html
11. https://pythonworld.ru/moduli/modul-time.html
12. https://pythonworld.ru/osnovy/formatirovanie-strok-metod-format.html
13. https://pythonz.net/references/named/range
14. https://pythonworld.ru/tipy-dannyx-v-python/mnozhestva-set-i-frozenset.html
15. https://tproger.ru/explain/python-dictionaries/
16. https://pythonworld.ru/tipy-dannyx-v-python/slovari-dict-funkcii-i-metody-slovarej.html
17. https://tproger.ru/translations/regular-expression-python/
18. https://pythonru.com/primery/primery-primeneniya-regulyarnyh-vyrazheniy-v-python

19. https://pyneng.readthedocs.io/ru/latest/book/15_module_re/
20. https://python-scripts.com/try-except-finally
21. https://pythonworld.ru/tipy-dannyx-v-python/isklyucheniya-v-python-konstrukciya-try-except-dlya-obrabotki-isklyuchenij.html
22. https://docs.python.org/3/library/exceptions.html
23. https://www.internet-technologies.ru/articles/funkcii-print-v-python.html
24. https://pythonz.net/references/named/input/
25. http://pythonicway.com/python-fileio
26. https://younglinux.info/oopython/polymorphism.php
27. https://pythonworld.ru/osnovy/peregruzka-operatorov.html
28. https://ru.wikibooks.org/wiki/GUI_Help/Tkinter_book
29. https://www.ibm.com/developerworks/ru/library/l-python_part_9/index.html
30. https://dev-gang.ru/article/mnogopotocznost-v-python-t2bkyunvku/
31. https://python-scripts.com/threading
32. https://habr.com/ru/post/149420/
33. https://docs.python.org/3/library/threading.html
34. https://docs.python.org/3/library/threading.html#rlock-objects
35. https://coderlessons.com/tutorials/python-technologies/izuchite-parallelizm-s-python/parallelizm-v-python-mnogoprotsessornost
36. https://coderlessons.com/tutorials/python-technologies/izuchite-parallelizm-s-python/vzaimodeistvie-protsessov
37. https://zametkinapolyah.ru/servera-i-protokoly/o-modeli-vzaimodejstviya-klient-server-prostymi-slovami-arxitektura-klient-server-s-primerami.html
38. https://echo.lviv.ua/dev/6455