

## Зміст

Лабораторна робота №1. Використання базових типів та засобів мови програмування Python. Середовища та інструментарій розроблення програм мовою Python	4
Лабораторна робота №2. Управляючі конструкції та масиви у мові Python	19
Лабораторна робота №3. Обробка послідовностей при програмуванні на мові Python. Списки	34
Лабораторна робота №4. Обробка послідовностей при програмуванні на мові Python. Рядки. Множини	52
Лабораторна робота №5. Розробка програм з використанням процедур і функцій.	68
Лабораторна робота №6. Механізми обробки винятків	82
Лабораторна робота №7. Розробка програм з використанням класів в Python	93
Лабораторна робота №8. Розробка програм з ієрархією класів. Організація класів з використанням успадкування в Python	104
Лабораторна робота №9. Робота з файлами у мові Python	120
Лабораторна робота №10. Робота з базою даних із Python-програми	130
Лабораторна робота №11. Розроблення програмного забезпечення з графічним інтерфейсом мовою Python	138
Лабораторна робота №12. Розробка додатків з графічним інтерфейсом. Програмування подій, робота з діалоговими вікнами	145
Лабораторна робота №13. Побудова графіків математичних функцій у мові Python	150
Лабораторна робота №14. Побудова 3D графіків. Робота з mplot3d Toolkit	160

# **Лабораторна робота №1**

## **Використання базових типів та засобів мови програмування Python. Середовища та інструментарій розроблення програм мовою Python**

### **1.1. Мета роботи**

Познайомитись з середовищами розробки Python і отримати головні навички розробки програмного забезпечення мовою Python.

### **1.2. Теоретичні відомості**

#### **Вступ до Python**

Python вважається однією з найпростіших мов для вивчення. У порівнянні з іншими популярними об'єктно-орієнтованими мовами, такими як C++ та Java, Python легко читати і розуміти, вона дозволяє швидко прототипувати та скорочує час розробки, має величезну кількість безкоштовних пакетів бібліотек.

Python має велику спільноту з відкритим кодом, яка спрямовує зусилля на безперервну роботу над поліпшенням Python як мови програмування. Також є спільнота Python, яка відповідає за розробку великої кількості відкритих бібліотечних пакетів, які можуть використовуватися для створення програм, які охоплюють діапазон від динамічних веб-сайтів до складного аналізу даних додатків, а також розробка простих GUI-додатків для побудови графіків із складних математичних функцій. Більшість пакетів бібліотек Python підтримують код, отриманий від спільноти з регулярними оновленнями. Де-факто репозиторій, який індексує найбільшу кількість пакетів Python - це PyPI (<https://pypi.org/>). PyPI також забезпечує простий спосіб встановлення різноманітних пакетів у вашій операційній системі.

**Python** – високорівнева мова програмування загального призначення, орієнтована на підвищення продуктивності розробника і читання коду.

Python підтримує декілька парадигм програмування, в тому числі структурну, об'єктно-орієнтовану, функціональну, імперативну та аспектно-орієнтовану.

На даний час існує дві гілки Python – 2.x та 3.x. Ми будемо працювати переважно з гілкою 3.x.

Існують різні інтерпретатори для мови Python. Офіційний інтерпретатор можна завантажити на сайті <https://www.python.org>.

### Основні принципи синтаксису мови Python:

1. Кінець рядка є кінцем інструкції (крапка з комою не потрібна).

```
x = 5  
print(2 + x)
```

2. Вкладені інструкції об'єднуються у блоки за величиною відступів. Відступ може бути будь-яким, головне, щоб в межах одного вкладеного блоку відступ був однаковий (рекомендується робити відступ 4 пробіли).

```
if x == 10:  
    print('yes')
```

3. Вкладені інструкції в Python записуються відповідно до одного і того ж шаблону, коли основна інструкція завершується двокрапкою, слідом за чим розташовується вкладений блок коду, зазвичай з відступом під рядком основної інструкції.

### Числа та операції над ними

Крім числових літералів, наведених в табл. 1.1, Python надає набір операцій для роботи з числовими об'єктами:

1. Оператори виразів +, -, \*, /, >>, \*\*, &, інші.
2. Вбудовані математичні функції *pow*, *abs*, *round*, *int*, *hex*, *bin*, інші.
3. Допоміжні модулі *random*, *math*, інші.

Для роботи з числами в основному використовуються вирази, вбудовані функції і модулі (при цьому числа мають ряд

власних, специфічних методів). Дійсні числа, наприклад, мають метод *as\_integer\_ratio*, який зручно використовувати для перетворення дійсного числа в раціональне, а також метод *is\_integer\_method*, який перевіряє – чи можна подати дійсне число як ціле значення. Цілі числа теж мають різні атрибути, включаючи метод *bit\_length*, він повертає кількість бітів, необхідних для подання значення числа. Крім того, множини підтримують власні методи і оператори виразів.

При роботі з числами часто використовують вирази: комбінації чисел (або інших об'єктів) і операторів, які повертають значення при виконанні інтерпретатором Python. Вирази в Python записуються з використанням звичайної математичної нотації і символів операторів. Наприклад, складання двох чисел  $X$  і  $Y$  записується у вигляді виразу  $X+Y$ , яке наказує інтерпретатору Python застосувати оператор «+» до значень з іменами  $X$  і  $Y$ . Результатом виразу  $X+Y$  буде інший числовий об'єкт. У табл. 1.1 наведено перелік всіх операторів, наявних в Python (математичні оператори: +, -, \*, / і т. д.; оператор % обчислює залишок від ділення, оператор «<<» виконує побітовий зсув вліво, оператор «&» виконує побітову операцію «і» і т. д.). Деякі оператори більш характерні для Python. Наприклад, оператор «is» перевіряє ідентичність об'єктів, оператор *lambda* створює неіменовані функції. Нерівність значень можна перевірити як  $X \neq Y$ . Операція ділення з округленням вниз ( $X//Y$ ) усікає дробову частину. Операція ділення  $X / Y$  виконує справжнє ділення (повертає результат з дробовою частиною).

Таблиця 1.1

## Оператори виразів в Python і правила визначення старшинства

Оператори	Опис
yield x	Підтримка протоколу send у функціях-генераторах
lambda args: expression	Створює анонімну функцію
x if y else z	Тримісний оператор вибору (значення x обчислюється, тільки якщо значення y істинно)
x or y	Логічна операція «АБО» (значення y обчислюється, тільки якщо значення x = хибність)
x and y	Логічний оператор «І» (значення y обчислюється, тільки якщо значення x = істина)
not x	логічне заперечення
x in y, x not in y	Перевірка на входження (для ітерованих множин)
x is y, x is not y	Перевірка ідентичності об'єктів
x < y, x <= y, x > y, x >= y x == y, x != y	Оператори порівняння, перевірка на підмножину і над множину Оператори перевірки на рівність
x   y	Бітова операція «АБО», об'єднання множин
x ^ y	Бітова операція «виключне АБО» (XOR), симетрична різниця множин
x & y	Бітова операція «І», перетин множин
x << y, x >> y	Зсув значення x вліво або вправо на y бітів
x, + y	Додавання, конкатенація
x - y	Віднімання, різниця множин

$x * y$	Множення, повторення
$x \% y$	Залишок, формат
$x / y, x // y$	Ділення: справжнє і з округленням вниз
$-x$	Унарний знак «мінус»
$\sim x$	Бітова операція «НЕ» (інверсія)
$x ** y$	Піднесення до степеню
$x[i]$	Індексація (в послідовності, відображеннях тощо)
$x[i:j:k]$	Витяг зрізу
$x(...)$	Виклик (функцій, класів і інших об'єктів)
$x.attr$	Звернення до атрибуту
$(...)$	Кортеж, підвираз, вираз-генератор
$[...]$	Список, генератор списків
$\{...\}$	Словник, множина, генератор словників і множин

Таблиця 1.2

### Функції в бібліотеці math

Назва функції	Призначення функції
<code>math.ceil(x)</code>	Повертає округлене $x$ як найближче ціле значення типу <code>float</code> , яке дорівнює або перевищує $x$ (округлення "вгору").
<code>math.copysign(x, y)</code>	Повертає число $x$ зі знаком числа $y$ . На платформі, яка підтримує знак нуля <code>copysign(1.0, -0.0)</code> дасть <code>-1.0</code> .
<code>math.fabs(x)</code>	Повертає абсолютне значення (модуль) числа $x$ . В Python є вбудована функція <code>abs</code> , але вона повертає модуль числа з тим же типом, що число, <code>fabs</code> же завжди повертає значення типу <code>float</code> .
<code>math.factorial(x)</code>	Повертає факторіал цілого числа $x$ , якщо $x$ не є цілим виникає помилка <code>ValueError</code> .

<code>math.floor(x)</code>	На противагу <code>ceil(x)</code> повертає округлене $x$ як найближче ціле значення типу <code>float</code> , менше або рівне $x$ (округлення "вниз").
<code>math.fmod(x, y)</code>	Аналогічна функції <code>fmod(x, y)</code> бібліотеки C. Зазначимо, що це не те ж саме, що вираз Python $x\%y$ . Бажано використовувати при роботі з об'єктами <code>float</code> , в той час як $x\%y$ більше підходить для <code>int</code> .
<code>math.frexp(x)</code>	Являє число в експоненційному записі $x = m \cdot 2^e$ і повертає мантису $m$ (дійсне число, модуль якого лежить в інтервалі від 0.5 до 1) і порядок $e$ (ціле число) як пару чисел $(m, e)$ . Якщо $x = 0$ , то повертає $(0.0, 0)$
<code>math.fsum(iterable)</code>	Повертає <code>float</code> суму від числових елементів ітерованого об'єкта.
<code>math.isinf(x)</code>	Перевіряє, чи є <code>float</code> об'єкт $x$ плюс або мінус нескінченністю, результат відповідно <code>True</code> або <code>False</code> .
<code>math.isnan(x)</code>	Перевіряє, чи є <code>float</code> об'єкт $x$ об'єктом NaN (not a number).
<code>math.ldexp(x, i)</code>	Повертає значення $x \cdot 2^i$ , тобто здійснює дію, зворотну функції <code>math.frexp(x)</code> .

### Введення та виведення даних

Введення даних з клавіатури в програму (починаючи з версії Python 3.0) здійснюється за допомогою функції `input ()`. Коли дана функція виконується, то потік виконання програми зупиняється в очікуванні даних, які користувач повинен ввести за допомогою клавіатури. Після введення даних і натискання `Enter`, функція `input ()` завершує своє виконання і повертає результат, який представляє собою рядок символів, введених користувачем. Функція `input ()` може приймати необов'язковий аргумент-запрошення рядкового типу; при виконанні функції повідомлення буде з'являтися на екрані.

```
x = input('Введіть x\n') #дані, що вводяться мають тип
рядка
y = input('Введіть y\n')
x = int(x) #здійснюємо перетворення типів
y = int(y)
print (x+y) #додаємо два числа, що були введені
```

## Перетворення типів даних

```
int(True) == 1
float(True) == 1.0
str(True) = 'True'
bool(0) == False
bool(0.0) == False
bool(1) == True
bool(10) == True
```

**В Python є особливий спосіб обміну змінних значеннями:**

```
(a, b) = (b, a)
```

Він використовується дуже часто. Даний метод працює завжди, навіть якщо змінні різних типів (в цьому випадку вони обмінюються не тільки значеннями, але і типами). Круглі дужки в цьому записі можна опустити:

```
a, b = b, a
```

## 1.3. Програма роботи

1.3.1. Ознайомитися з призначенням та принципами роботи в середовищі розробки програмного забезпечення Python IDLE.

1.3.2. Налаштувати середовище розробки Python IDLE.

1.3.3. Створити та запустити на виконання приклади 1-3.

1.3.4. Виконати завдання 1 згідно варіанту.

## 1.4. Обладнання та програмне забезпечення

1.4.1. Персональний комп'ютер.

1.4.2. Інтерпретатор Python встановлений на ПК



## 1.5. Порядок виконання роботи і опрацювання результатів

### Встановлення

Завантажте останню версію інтерпретатора для вашої операційної системи за посиланням <https://www.python.org/downloads/>. При установці на ОС Windows необхідно налаштувати змінну оточення PATH



Рис. 1.1. Інсталяція Python

Після завершення установки можна запускати програми як з командного рядка, так і за допомогою інтерфейсу з Python IDLE.



Рис. 1.2. Виконання Python команд в командному рядку

**Інтерфейс користувача IDLE.** Програма IDLE може запропонувати вам графічний інтерфейс користувача для розробки програм на мові Python (IDLE інтегроване середовище розробки – integrated development environment). IDLE - це набір

інструментальних засобів з графічним інтерфейсом, який здатний працювати на самих різних платформах, включаючи Microsoft Windows. IDLE - це програма на мові Python, яка створює графічний інтерфейс за допомогою бібліотеки *tkinter* GUI, що забезпечує її переносимість, але також означає, що для використання IDLE вам доведеться забезпечити підтримку *tkinter* в Python (версія Python для Windows має таку підтримкою за замовчуванням).

В IDLE присутні звичні пункти меню, а для виконання найбільш поширених операцій можна використовувати короткі комбінації клавіш.

Щоб створити (або відредагувати) файл з вихідним програмним кодом в середовищі IDLE, відкрийте вікно текстового редактора: в головному вікні відкрийте меню File (Файл) пункт New Window (Нове вікно) – відкрити вікно текстового редактора (або Open ... (Відкрити) – щоб відредагувати існуючий файл). Для запуску сценарію, відкритого у вікні редагування, використовують меню «Run» цього вікна (пункт «Run Module»).

IDLE забезпечує підсвічування синтаксису програмного коду, який вводиться як в головному вікні, так і у всіх вікнах текстового редактора – ключові слова виділяються одним кольором, літерали іншим кольором і т. д.

Щоб запустити файл з програмним кодом в середовищі IDLE, виберіть вікно, де редагується текст, розкрийте меню Run (Запустити) і виберіть в ньому пункт Run Module (Запустити модуль) або скористайтесь комбінацією клавіш, яка відповідає цьому пункту меню. Якщо з моменту відкриття або останнього збереження файлу його вміст змінювалося, Python запропонує зберегти його.

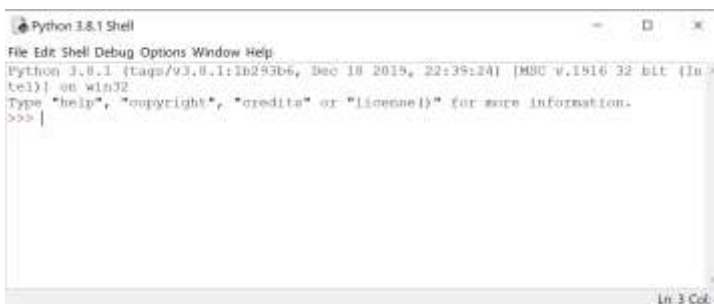
Коли сценарій запускається таким способом, весь висновок, який він генерує, а також всі повідомлення про

помилки з'являються в основному вікні інтерактивного сеансу роботи з інтерпретатором (командна оболонка Python).

Крім основних функцій редагування і запуску середовище IDLE надає цілий ряд додаткових можливостей, включаючи налагоджувач і інспектор об'єктів. Налагоджувач IDLE активується за допомогою меню Debug (Налагодження), а інспектор об'єктів – за допомогою меню File (Файл). Інспектор об'єктів дозволяє переходити, переміщаючись по шляху пошуку модулів, до файлів і об'єктів в файлах – клацання на файлі або об'єкті призводить до відкриття відповідного вихідного тексту в вікні редагування.

*Режим налагодження* в IDLE ініціюється вибором пункту меню Debug (Налагодження) → Debugger (Отладчик) головного вікна, після цього можна запустити налагоджувач сценарію вибором пункту меню Run (Запустити) → Run Module (Запустити модуль). Як тільки налагоджувач буде активований, клацанням правої кнопки миші на вибраному рядку у вікні редагування ви зможете встановлювати точки зупинки в своєму програмному коді, щоб призупиняти виконання сценарію, переглядати значення змінних тощо Ви зможете стежити за ходом виконання програм – в цьому випадку поточний виконуваний рядок програмного коду виділяється кольором.

У разі появи помилок можна натиснути правою кнопкою миші на рядку з повідомленням про помилку і швидко перейти до рядка програмного коду, який викликав цю помилку. Це дозволяє швидко з'ясувати джерело помилки і ліквідувати її. Крім цього текстовий редактор IDLE володіє великим набором можливостей, які стануть в нагоді програмістам, включаючи автоматичне оформлення відступів, розширений пошук тексту і файлів тощо.



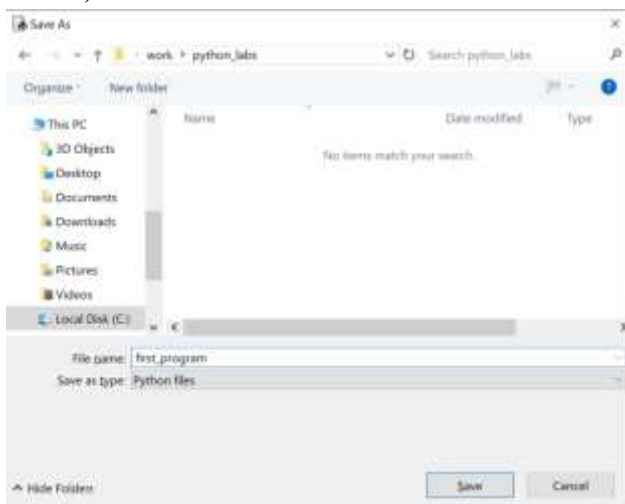
*Рис. 1.3. Вигляд Python IDLE*

### **Перша програма**

Якщо працюєте в Python IDLE. File ->New File. У вікні, що з'явилося вводите інструкції, наприклад

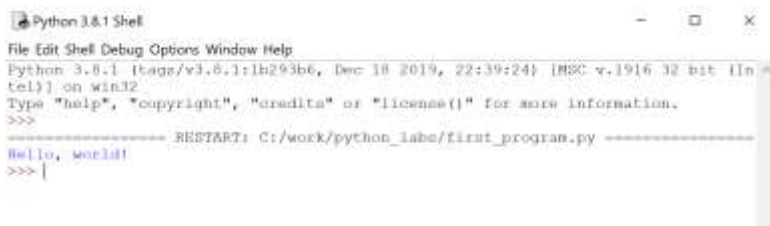
`print('Hello, world!')`

Зберігаєте файл File ->Save As (розширення файла .py Python files)



*Рис. 1.4. Збереження файлу програми*

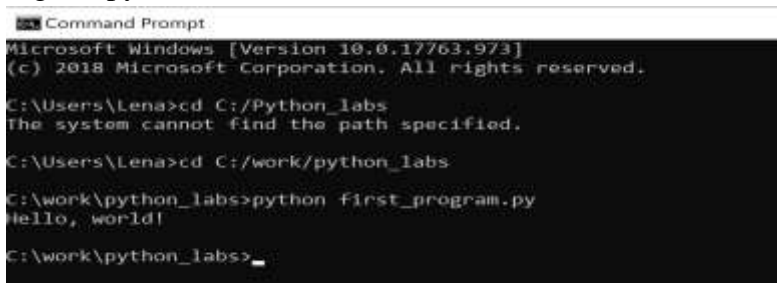
Для виконання Run ->Run Module F5



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
-----RESTART: C:/work/python_labs/first_program.py-----
Hello, world!
>>>|
```

*Рис. 1.5. Запуск на виконання програми*

Можна також запустити з командного рядка вказавши після команди python шлях до файлу, який щойно створили (first\_program.py)



```
Command Prompt
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Lena>cd C:/Python_labs
The system cannot find the path specified.

C:\Users\Lena>cd C:/work/python_labs

C:\work\python_labs>python first_program.py
Hello, world!

C:\work\python_labs>_
```

*Рис. 1.6. Запуск на виконання програми з командного рядка*

Розглянемо на прикладах основи роботи з Python.

### Приклад 1. Базові типи даних (вказуються неявно)

```
a = 5 # int
a = 5 # int
b = 7.0 # float
c = 2 > 4 # boolean
d = "World" # string
e = 1.5 + 0.5j # complex
print(type(a))
print(type(b))
print(type(c))
print(type(d))
print(type(e))
print(a, b, c, d, e.real, e.imag)
```

## Приклад 2. Основні арифметичні операції

```
a = 5 # int
b = 7.0 # float
c = 1 + 2 # 3
d = 5 - 3 # 2
e = a * b # 35.0
f = 3.0 / 2 # 1.5
g = 3 / 2 # 1
h = 5 % 3 # 2
j = 10 ** 7.3 # 19952623.1497
print(type(a))
print(type(b))
print(type(c))
print(type(d))
print(type(e))
print(type(f))
print(type(g))
print(type(h))
print(type(j))
print(a, b, c, d, e, f, g, h, j)
```

## Приклад 3. Вбудовані математичні функції (необхідне підключення бібліотеки math)

```
from math import *
a = 1
b = 2
x = sqrt(a*b)/(exp(a)*b)+a*exp((2*a)/b)
print(x)
```

### Завдання:

В завданні 1 кожного варіанту необхідно обчислити значення виразу та вивести його на екран.

## Завдання 1

Варіант	Завдання	Варіант	Завдання
1	$z = \left( \frac{e^{-x} - 12.34}{\lg x - \cos x^3} \right)^{-0.5}$	16	$y = \frac{e^{-3x} + tg(4x-1)}{ \cos x  + \sqrt{\cos 2x}}$
2	$y = \frac{\sqrt{x-1} - tg(x+1)}{\arccos x + \ln x} + 2,75$	17	$y = \frac{e^{-3x} + \ln^3(x-1)}{\ln x+1  + tg(x^2-1)}$
3	$g = \frac{\sin x^2 - \cos^4(x-1)^2}{\arctg(x+2,6) + \sqrt[3]{\ln x}}$	18	$y = \sin \left( \frac{x+2.3 \cdot \lg(x+1)}{\sqrt{2 \ln x} + \cos x} \right)$
4	$p = \frac{e^{-3x} + tg(3x-3)}{ \sin x  + \sqrt[4]{\cos x + \cos 2x}}$	19	$u = \ln 1-x  + \frac{tg x - \sin^2 x}{1 - \sqrt{\ln x}}$
5	$y = \frac{e^{-x} - 4x - \ln^3 x}{\lg x+1  + ctg(x^2-1)}$	20	$y = \frac{e^{-x} + tg(x-1)}{ \ln x  + \sqrt{\sin x + \cos 2x}}$
6	$y = \arcsin \left( \frac{x \cdot \ln x}{1 + \cos x} + 1 \right)$	21	$c = tg x  - \frac{\sqrt{\ln 2 + \ln x}}{\sqrt[3]{tg x} + \sqrt[4]{\cos x^{-1}}}$
7	$y = x + \frac{\sqrt{\arcsin x + \arctg \pi x}}{\lg(13.4x + \pi)}$	22	$y = \frac{\sqrt{x+1} - \sin(x-\pi)}{\cos(x-3.1) + \ln^2 x} + x \cdot \lg x$
8	$y = \arctg x + \frac{e^{0.6x-1} - \sqrt{(x+6,1)^3}}{\ln x + tg^2 x}$	23	$y = tg \left( \frac{x+3 \cdot \lg(x+1)}{\sqrt{\ln 4x} + \cos x} \right)$
9	$u = 0,3 \cdot \lg e^{-x} + \frac{\arctg x - \sin^2 x}{4 \cdot \sqrt{\ln x-1 }}$	24	$y = \frac{\sin^3 x - \cos(x-1)^2}{\arctg(x+1) + \sqrt[3]{\lg x}}$
10	$y = e^2 \cdot \lg x^4 \cdot \frac{(x-0,5)^2 - \cos x}{\sqrt{ x+1 } +  x }$	25	$c = \lg x  - \frac{\sqrt{\ln(x-1) + \ln x}}{\sqrt[3]{tg x} + \sqrt{\cos(x-\pi)}}$
11	$y = \frac{e^{-x} - 4 \cdot \lg x}{\ln x - \cos x+1 }$	26	$y = \frac{\sqrt{\sin(x+1) + \arctg \pi x}}{\lg(x+2\pi)} - 1$
12	$c = \sin^2 x - \frac{\sqrt{\lg 2-x  + \lg x}}{\sqrt[3]{tg x} + \sqrt{\cos^3 x}}$	27	$y = \arcsin^2(x-1) + \left( \frac{x \cdot \ln x - 2\sqrt{x}}{1 + \cos x} \right)$
13	$y = \arccos \left( \frac{x - \lg x}{1 + \cos 3x} + 1 \right)$	28	$y = \frac{e^{-x} - x \cdot \sin x - \ln^2 x}{\lg \cos x  + ctg(x^2-1)}$
14	$c = \arctg x - \frac{\sqrt{\ln 4 + \ln x}}{\sqrt[3]{\lg 2.4} + \sqrt[5]{\cos x^{-1}}}$	29	$y = \arcsin x + \frac{e^{x-1} - \sqrt{(x+1)^5}}{\lg^3 x + tg x}$

<b>15</b>	$y = \frac{\ln e^{-x} + \cos(x-1)}{\ln^3 x + \sqrt{\sin 3x + \cos 2x}}$	<b>30</b>	$y = 3 \cdot \lg x^4 \cdot \frac{(x-1)^2 - \operatorname{tg} x}{\cos x + \sqrt{ x+1  +  x }}$
-----------	---	-----------	---

### 1.6. Контрольні запитання.

- 1.6.1. Які особливості та переваги мови Python Ви знаєте?
- 1.6.2. Назвіть основні принципи синтаксису мови Python.
- 1.6.3. Як здійснюється введення/виведення даних у мові Python?
- 1.6.4. Назвіть основні тип даних в Python.
- 1.6.5. Як можна обміняти значеннями дві змінні в Python?
- 1.6.6. Яка функція приводить змінну до рядкового типу?
- 1.6.7. Що означає None?
- 1.6.8. В якій бібліотеці містяться стандартні математичні функції?
- 1.6.9. Назвіть парадигми програмування, які підтримує Python.
- 1.6.10. Яке призначення функції bool?



## Лабораторна робота №2

### Управляючі конструкції та масиви у мові Python

#### 2.1. Мета роботи

Навчитися створювати найпростіші програми на мові Python, використовуючи оператори вибору і циклів, арифметичні вирази.

#### 2.2. Теоретичні відомості

##### Умовна інструкція if

**Логіка висловлювань. Логічний тип даних.** Величини можна порівнювати. Для цього в Python є такі операції порівняння:

> більше	<= менше чи рівне
< менше	== дорівнює
>= більше чи рівне	!= не дорівнює

Наприклад:

6>5 – True

7<1 – False

7==7 – True

7 != 7 – False

Python повертає значення True (Істина == 1), коли порівняння істинне, False (Хибність == 0) – в іншому випадку. True і False відносяться до логічного (булевого) типу даних *bool*. В програмах часто використовують більш складні вирази – висловлювання (це – означає деяке твердження, яке може бути істинним або хибним). Кажуть, що істинність (True) або хибність (False) – це логічні значення висловлювання.

Логіка висловлювань вивчає способи, за допомогою яких з одних висловлювань можна утворювати інші, причому в такий спосіб, щоб істинність або хибність нових висловлювань залежала лише від істинності або хибності старих. Для цього використовуються так звані (логічні) сполучники. Python

використовує три логічних оператора *and*, *or*, *not* які відповідають сполучникам І, АБО, НЕ в логіці висловлювань.

Наприклад:

```
x = 8
y = 13
x == 8 and y < 15  # x дорівнює 8 та у менше 15
x > 8 and y < 15  # x більше 8 та у менше 15
x != 0 or y > 15  # x не дорівнює 0 або у менше 15
0 or y > 15  # x менше 0 або у менше 15
```

Для Python істинним або хибним може бути не тільки логічне висловлювання, а й об'єкт. Будь-яке число, яке не дорівнює нулю (або непорожній об'єкт) інтерпретують як «істина». Числа, які дорівнюють нулю, порожні об'єкти та спеціальний об'єкт *None* інтерпретують як «хибність». Наприклад:

```
print('' and 2)  # '' - False and True
print('' or 2)  # 2 - False or True
y = 6 > 8
print(6 > 8)  # False
print(y)  # False
print(not y)  # True
print(not None)  # True
print(not 2)  # False
print(2 > 4 and 45 > 3)  # False - False and True
```

При обчисленні оператора *and* Python обчислює операнди зліва направо і повертає перший об'єкт, який має помилкове значення.

```
print(0 and 3)  # поверне перший помилковий операнд - 0
print(5 and 4)  # поверне крайній правий операнд - 4
```

Якщо Python не знаходить помилковий об'єкт-операнд, він повертає крайній правий операнд.

Логічний оператор *or* діє схожим чином, але для об'єктів-операндів Python повертає перший об'єкт, який має значення

«істина». Python припинить подальші обчислення, як тільки буде знайдений перший об'єкт, який має значення «істина».

```
print(2 or 3)  # повертає перший об'єкт-операнд зі
               # значення «істина» 2
print(None or 5)  # повертає другий операнд, тому що
                 # перший - хибний 5
print(None or 0)  # повертає об'єкт-операнд, що
                 # залишився 0

print(1 + 3 > 7)  # пріоритет + вище, ніж > False
print((1 + 3) > 7)  # дужки сприяють наочності і
                  # позбавляють від помилок False
print(1 + (3 > 7))  # 1
```

В Python можна перевіряти належність до інтервалу:

```
x = 0
-5 < x < 10  # еквівалентно: x > -5 and x<10 True
```

Рядки в Python можна порівнювати аналогічно числам. Символи, як і все інше, подають в комп'ютері у вигляді чисел. Існує таблиця, яка ставить у відповідність кожному символу деяке число. Визначити, яке число відповідає символу можна за допомогою функції `ord()`:

```
ord('L')  # 76
ord('Ø')  # 1060
```

Тепер порівняння символів зводиться до порівняння чисел, які їм відповідають:

```
print('A' > 'L')  # False
```

При порівнянні рядків Python їх порівнює посимвольно:

```
x = 'Aa' > 'Ll'  # False
```

Оператор `in` перевіряє наявність підрядка в рядку:

```
'a' in 'abc'  # True
'A' in 'abc'  # великої літери A немає False
"" in 'abc'   # порожній рядок є в будь-якому рядку True
'' in ''      # True
```

**Інструкція if.** Розглянемо умовну інструкцію *if*, яку використовують для вибору серед альтернативних операцій на основі результатів перевірки. Інструкція *if* обирає, яку дію необхідно виконати. Вона може містити інші інструкції, в тому числі інші умовні інструкції *if*.

Спочатку записується частина *if* з умовним виразом, далі можуть слідувати одна або більше необов'язкових частин *elif* («*else if*») з умовними виразами і, нарешті, необов'язкова частина *else*. Умовні вирази і частина *else* мають асоційовані з ними блоки вкладених інструкцій, з відступом щодо основної інструкції. Під час виконання умовної інструкції *if* інтерпретатор виконує блок інструкцій, асоційований з першим умовним виразом, тільки якщо він повертає значення «істина», в іншому випадку виконується блок інструкцій *else*. Загальна форма запису умовної інструкції *if* виглядає таким чином:

```
if <test1>: # Інструкція if з умовним виразом test1
    <statements1> # асоційований блок
elif <test2>: # необов'язкові частини elif
    <statements2>
else: # необов'язковий блок else
    <statements3>
```

Розглянемо кілька прикладів. Всі частини цієї інструкції, за винятком основної частини *if* з умовним виразом і пов'язаних з нею інструкцій, є необов'язковими. У найпростішому випадку інші частини інструкції опущено:

```
if 1:
    print('true')
```

Запрошення до введення змінюється на «...» для рядків продовження в базовому інтерфейсі командного рядка, що використовується тут (в IDLE текстовий курсор переміщається

на наступний рядок вже з відступом, а натискання на клавішу *Backspace* повертає на рядок вгору). Введення порожнього рядка (подвійним натисканням клавіші Enter) завершує інструкцію і призводить до її виконання. Число «1» – це логічна істина, тому дана перевірка завжди буде успішною. Щоб обробити помилковий результат, додайте частину *else*:

```
x = int(input('Введіть x:'))
if x == 1:
    print('true')
else:
    print('false')
```

*Множинне розгалуження.* Розглянемо приклад умовної інструкції *if*, в якій присутні всі необов'язкові частини:

```
x = 'killer rabbit'
if x == 'roger':
    print("how's jessica?")
elif x == 'bugs':
    print("what's up doc?")
else:
    print("Run away! Run away!")
```

Ця багаторядкова інструкція простягається від рядка *if* до кінця блоку *else*. При виконанні цієї інструкції інтерпретатор виконає вкладені інструкції після тієї перевірки, яка дасть в результаті істину, або блок *else*, якщо всі перевірки дадуть результат «хибність». Обидві частини *elif* і *else* можуть бути опущені, і в кожній частині може бути більше однієї вкладеної інструкції. Зв'язок слів *if*, *elif* і *else* визначений тим, що вони знаходяться на одній вертикальній лінії, з одним і тим же відступом.

У Python множинне розгалуження оформляють у вигляді послідовності перевірок *if/elif*. Використання інструкції *if* є найбільш простим способом організації множинного розгалуження.

## Інструкція *while*

Алгоритм, в якому передбачено неодноразове виконання певної послідовності дій, називається алгоритмом циклічної структури або циклом. Цикл дозволяє істотно скоротити розмір запису алгоритму, зобразити його компактно шляхом відповідної організації дій. Повторювати певні дії має сенс при різних значеннях параметрів, які змінюються. Такі параметри називаються *параметрами циклу*. Блок повторюваних операторів називають *тілом циклу* (це послідовність дій, які виконується багаторазово).

Циклічний процес називається ітераційним, якщо заздалегідь невідома кількість повторень циклу, а кінець обчислення визначається при досягненні деякою величиною заздалегідь заданої точності обчислення.

При програмуванні ітераційних процесів прийнято їх розділяти на цикли з «передумовою» і з «післяумовою». Їх відмінність полягає в тому, що перевірка досягнення деякою величиною заданої точності обчислення здійснюється або на початку циклу, або наприкінці циклу відповідно. Особливість циклу з «післяумовою» полягає в тому, що повторювана ділянка алгоритму виконається хоча б один раз, у той час як в циклі з «передумовою» ця ділянка може не виконатися жодного разу. Процес ініціалізації включає в себе визначення (введення) початкових значень змінних, які використовуються в тілі циклу.

Для запису ітераційних процесів в Python використовують лише один тип операторів циклу *while* - оператор з попередньою умовою (передумовою). Для всіх операторів циклу характерні такі особливості:

1. Повторювані обчислення записуються лише один раз.
2. Вхід в цикл можливий тільки через його початок.
3. Змінні оператора циклу повинні бути визначені до входу в цикл.

4. Потрібно передбачити вихід з циклу. Якщо цього не зробити, то обчислення будуть тривати нескінченно довго.

Нескінченний цикл – це циклічна ділянка в програмі, в якій не передбачені засоби виходу з циклу при досягненні деякого умови.

Інструкція *while* організує цикл з передумовою (перевірка виконується перед початком чергової ітерації), складається з рядка заголовка з умовним виразом, тіла циклу, що містить одну або більше інструкцій з відступами, і необов'язкової частини *else*, яка виконується, коли управління передається за межі циклу без використання інструкції *break*. Інтерпретатор продовжує обчислювати умовний вираз в рядку заголовка і виконувати вкладені інструкції в тілі циклу, поки умовний вираз не поверне значення «хибність»:

```
while <test>: # Умовний вираз test  
    <statements1> # Тіло цикла  
else: # Необов'язкова частина else  
    <statements2> # Виконується, якщо вихід із цикла  
        # виконується не інструкцією break
```

Оператор *while* дозволяє багаторазово виконувати певні дії в залежності від деякої <умови> (виразу логічного типу, який приймає тільки значення *True* або *False*). Цикл виконується поки <Умова>=«істина». Як тільки <Умова> порушується, виконання циклу завершується.

Інструкція *while* продовжує виконувати блок інструкцій (зазвичай з відступами), поки умовний вираз продовжує повертати значення «істина». Вона називається «циклом», тому що управління циклічно повертається до початку інструкції, поки умовний вираз не поверне значення «хибність». Як тільки в результаті перевірки буде отримано значення «хибність»,

управління буде передано першій інструкції, яка розташована відразу ж за вкладеним блоком тіла циклу *while*.

**Break, continue, pass і else.** Розглянемо дві прості інструкції, які можна використати тільки усередині циклів – інструкції *break* і *continue*. У Python:

*Break* виконує перехід за межі циклу (всієї інструкції циклу).

*Continue* виконує перехід на початок циклу (в рядок заголовка).

*Pass* нічого не робить: це пуста інструкція.

Блок *else* виконується, тільки якщо цикл завершився звичайним чином (без використання інструкції *break*).

З урахуванням інструкцій *break* і *continue* цикл *while* має такий загальний вигляд:

```
while <test1>:  
    <statements1> # тіло циклу  
    if <test2>:  
        break # Вийти з циклу, пропустивши частину else  
    if <test3>:  
        continue #Перейти на початок циклу, до. test1  
    else:  
        <statements2> # Виконується, якщо не була  
                        # використана інструкція 'break'
```

Інструкції *break* і *continue* можуть з'являтися в будь-якому місці всередині тіла циклу *while* (або *for*), але як правило, їх використовують в умовних інструкціях *if*, щоб виконати необхідну дію у відповідь на деяку умову.

**Pass.** Інструкція *pass* не виконує ніяких дій, її використовують у випадках, коли синтаксис мови вимагає наявності інструкції, але ніяких корисних дій в цій точці програми виконати не можна. Вона часто використовується в якості порожнього тіла складної інструкції. Наприклад, створити нескінченний цикл, який нічого не робить, можна таким чином:



```
while 1:
    pass # Натисніть Ctrl-C, щоб припинити цикл!
```

Цей приклад нескінченно робить «ніщо». Ця інструкція може використовуватися, наприклад, для того, щоб ігнорувати виключення в інструкції *try*. Іноді інструкцію *pass* використовують як заповнювач, замість того, «що буде написано пізніше», і в якості тимчасового фіктивного тіла функцій.

Порожнє тіло функції викличе синтаксичну помилку, тому в подібних ситуаціях можна використати інструкцію *pass*. У Python 3.0 замість будь-якого виразу допускається використовувати три крапки «...», які самі по собі не виконують жодної дії, тому їх можна використовувати як альтернативу інструкції *pass*, зокрема замість програмного коду, який буде написано пізніше (примітка: To Be Done або TODO – слід реалізувати):

```
while 1:
    ... # Натисніть Ctrl-C, щоб припинити цикл!
```

**Continue.** Інструкція *continue* виконує перехід на початок циклу. Вона іноді дозволяє уникнути використання вкладених інструкцій. У наступному прикладі цю інструкцію використовують для пропуску непарних чисел (цей фрагмент виводить парні числа менше «10» і більше або рівні «0»). Число «0» означає «хибність», а оператор *%* обчислює залишок від ділення, тому даний цикл виводить числа в зворотному порядку, пропускаючи значення, кратні 2 (він виводить 8 6 4 2 0):

```
x = 10
while x:
    x = x - 1 # x -= 1
    if x % 2 != 0: continue # непарне, пропустіть друк
    print(x, end=' ')
```

Аргумент *end=' '* забезпечує виведення значень в один рядок через пробіл. Останній приклад виглядав би зрозуміліше, якби інструкція *print* була складовою інструкції *if*:

```

x = 10
while x:
    x = x - 1
    if x % 2 == 0: # парне? - вивести
        print(x, end=' ')

```

**Break.** Інструкція *break* виконує негайний вихід з циклу. Програмний код, розташований в циклі за цією інструкцією не виконується, якщо ця інструкція запущена. Нижче наведено інтерактивний цикл, який виконує введення даних за допомогою функції *input* і вихід з циклу, якщо у відповідь на запит імені буде введена рядок «*stop*»:

```

while 1:
    name = input('Enter name:')
    if name == 'stop':
        break
    age = input('Enter age: ')
    print('Hello', name, '=>', int(age) ** 2)

```

Цей приклад виконує перетворення рядка *age* в ціле число за допомогою функції *int*, перед тим як звести його до другого степеня (це необхідно, тому що функція *input* повертає результат введення користувача у вигляді рядка).

**Else.** При об'єднанні з частиною *else* інструкція *break* дозволяє позбутися від необхідності зберігати прапор стану пошуку. Наступний фрагмент визначає, чи є додатне ціле число простим числом, виконуючи пошук дільників більше за значення «1»:

```

y = 5
x = y // 2 # Для значень y>1
while x > 1:
    if y % x == 0: # залишок
        print(y, 'has factor', x)
        break # Переступити блок else
    else:
        x -= 1
        print(y, 'is prime')

```

Замість того щоб встановлювати прапор, який буде перевірений після закінчення циклу, досить вставити інструкцію *break* в місці, де буде знайдений дільник. При такій реалізації управління буде передано блоку *else*, тільки якщо інструкція *break* не була виконана, тобто коли з упевненістю можна сказати, що число є простим. Блок *else* циклу виконується також у разі, коли тіло циклу жодного разу не виконувалося, оскільки в цій ситуації інструкція *break* також не виконується. У циклах *while* це відбувається, коли перша ж перевірка умови в заголовку дає значення «хибність». Внаслідок цього в попередньому прикладі буде отримано повідомлення «is prime» (просте число), якщо спочатку  $x$  менше або дорівнює «1» (тобто, коли  $y=2$ ). Блок *else* в циклах дозволяє обробити «інший» спосіб виходу з циклу, без необхідності встановлювати і перевіряти прапори або умови.

Цей приклад визначає прості числа, але недостатньо точно. Числа, менші «2», не вважають простими у відповідності із суворим математичним визначенням. Якщо бути більш точним, цей код також буде терпіти невдачі при від'ємних значеннях і виконуватися успішно при використанні дійсних чисел без дробової частини. В Python замість оператора ділення  $/$  використовують оператор  $//$ , тому що тепер оператор  $/$  виконує операцію «істинного ділення» (початкове ділення необхідно, щоб відсікти залишок!).

### Цикл *for*

Цикл *for* – універсальний ітератор послідовностей: він може виконувати обхід елементів в будь-яких впорядкованих об'єктах–послідовностях. Інструкція *for* здатна обробляти рядки, списки, кортежі, інші вбудовані об'єкти, які підтримують можливість виконання ітерацій. Цикли *for* в Python розпочинають з рядка заголовка, де вказують змінну циклу для присвоювання, а також об'єкт, обхід якого буде виконано.

За заголовком розташовано блок (зазвичай з відступами) інструкцій, які потрібно виконати:

```

for <target> in <object>:
# Пов'язує елементи об'єкта зі змінною циклу
    <Statements>
# тіло циклу: використовує змінну циклу
else:
    <Statements>
# Якщо не потрапили на інструкцію 'break'

```

Коли інтерпретатор виконує цикл *for*, він по черзі присвоює змінній циклу елементи об'єкта–послідовності і виконує оператори з тіла циклу для кожного елементу. Змінну циклу можна змінити в тілі циклу, проте їй автоматично буде присвоєно наступний елемент послідовності, коли управління повернеться на початок циклу. Після виходу з циклу ця змінна зазвичай посилається на останній елемент послідовності (якщо цикл не був завершений інструкцією *break*).

Інструкція *for* підтримує необов'язкову частину *else*, яка працює так само, як і в циклах *while* – її виконують, якщо вихід з циклу реалізовано не інструкцією *break* (тобто, якщо в циклі не виконано обхід всіх елементів послідовності). Інструкції *break* і *continue* в циклах *for* працюють так само, як і у циклах *while*. Повна форма циклу *for* має такий вигляд:

```

for <target> in <object>:
# змінній циклу присвоюють елементи об'єкта
    <Statements>
    if <test>: break # Вихід з циклу, міняючи блок else
    if <test>: continue # Перехід на початок циклу
else: <Statements> # Якщо не викликано 'break'

```

## 2.3. Програма роботи

2.3.1. Ознайомитися з принципами організації програм розгалуженої структури в Python.

2.3.2. Створити та запустити на виконання приклади 1-3.

2.3.3. Виконати завдання 1 згідно варіанту.

## 2.4. Обладнання та програмне забезпечення

2.4.1. Персональний комп'ютер.

2.4.2. Інтерпретатор Python встановлений на ПК

## 2.5. Порядок виконання роботи і опрацювання результатів

### Приклад 1. Використання умовного оператора

```
name = ''
while True:
    print('Who are you?')
    name = input()
    if name != 'Joe':
        continue
    print('Hello, Joe. What is the password? (It is a fish.)')
    password = input()
    if password == 'swordfish':
        break
print('Access granted.')
```

### Приклад 2. Використання циклу while

```
name = None # спочатку ми не знаємо імені користувача
# нескінчений цикл
```

```
while True:
    print('Меню:')
    print('1. Ввести ім'я')
    print('2. Вивести привітання')
    print('3. Вийти')
    response = input('Виберіть пункт: ')
    print()
    if response == '1':
        name = input('Введіть ваше ім'я: ')
    elif response == '2':
        if name: #вітаємося з користувачем, якщо ім'я
вже введено
            print('Привіт, ', name, '!', sep='')
        else:
```

```

        print('Я не знаю вашого імені.')
    elif response == '3':
        # оператор break завершує виконання циклу
        break #якщо користувач вибрав 3, то виходимо
з циклу
    else:
        print('Неправильне введення.')
print()

```

### Завдання:

В завданні 1 кожного варіанту для його реалізації слід застосувати розгалуження та цикли.

#### Завдання 1

Варіант	Завдання
1	$y = \frac{\cos^2 x}{x^2 + 1}$ , $3,8 \leq x \leq 7,6$ , $\Delta x = 0,6$ ;
2	$y = \frac{\operatorname{tg} 0,5x}{x^3 + 7,5}$ , $0,1 \leq x \leq 1,2$ , $\Delta x = 0,1$ ;
3	$y = \frac{e^{2x} - 8}{x + 3}$ , $-1 \leq x \leq 2,3$ , $\Delta x = 0,7$ ;
4	$y = \frac{x + \cos 2x}{3x}$ , $2,3 \leq x \leq 5,4$ , $\Delta x = 0,8$ ;
5	$y = \frac{x + \cos 2x}{x + 2}$ , $0,2 \leq x \leq 10$ , $\Delta x = 0,8$ ;
6	$y = \frac{\cos^3 t^2}{1,5t + 2}$ , $2,3 \leq t \leq 7,2$ , $\Delta t = 0,8$ ;
7	$z = \frac{x^3 + 2x}{3 \cos \sqrt{x} + 1}$ , $0 \leq x \leq 2$ , $\Delta x = 0,4$ ;
8	$z = \frac{t + \sin 2t}{t^2 - 3}$ , $2,4 \leq t \leq 6,9$ , $\Delta t = 0,4$ ;
9	$y = \frac{x^3 - 2}{3 \ln x}$ , $4,5 \leq x \leq 16,4$ , $\Delta x = 2,2$ ;
10	$z = \frac{2,3t + 8}{2 \cos t + 1}$ , $0 \leq t \leq 6,5$ , $\Delta t = 1,1$ ;
11	$y = \frac{\arccos x}{2x + 1}$ , $0,1 \leq x \leq 0,9$ , $\Delta x = 0,1$ ;
12	$y = \frac{5 \operatorname{tg}(x + 7)}{(x + 3)^2}$ , $1,2 \leq x \leq 6,3$ , $\Delta x = 0,2$ ;

<b>13</b>	$y = \frac{1,5t - \ln 2t}{3t+1}, 2,5 \leq t \leq 9, \Delta t = 0,8;$
<b>14</b>	$y = \frac{2,5x^3}{e^{2x} + 2}, 0 \leq x \leq 0,5, \Delta x = 0,1;$
<b>15</b>	$y = \frac{3x-2}{2\arctg x +1}, 3,2 \leq x \leq 5,2, \Delta x = 0,4;$
<b>16</b>	$y = \frac{5\lg x}{x^2-1}, 1,2 \leq x \leq 3,8, \Delta x = 0,4;$
<b>17</b>	$z = \frac{6x+4}{\sin 3x-x}, 2,3 \leq x \leq 7,8, \Delta x = 0,9;$
<b>18</b>	$z = \frac{2\sin^2(x+2)}{x^2+1}, 7,2 \leq x \leq 12, \Delta x = 0,5;$
<b>19</b>	$y = \frac{(3x+2)^2}{\sin x+3}, 4,8 \leq x \leq 7,9, \Delta x = 0,4;$
<b>20</b>	$y = \frac{2\sin^3 x}{3x+1}, -1 \leq x \leq 1, \Delta x = 0,25;$
<b>21</b>	$y = \frac{tg 2t-3t}{t+3}, 0,2 \leq t \leq 0,8, \Delta t = 0,1;$
<b>22</b>	$y = \frac{3x+1}{\arctg x}, 0,1 \leq x \leq 1,5, \Delta x = 0,2;$
<b>23</b>	$y = \frac{2t+8}{\cos 3t+1}, 2 \leq t \leq 6,5, \Delta t = 0,8;$
<b>24</b>	$y = \frac{\arccos x}{3x+1}, 0,1 \leq x \leq 0,9, \Delta x = 0,1;$
<b>25</b>	$y = \frac{(x+2)^2}{\sqrt{x^2+1}}, 2,3 \leq x \leq 8,3, \Delta x = 0,6;$
<b>26</b>	$y = \frac{t - \ln 2t}{3t+1}, 2,1 \leq t \leq 8,5, \Delta t = 0,7;$
<b>27</b>	$y = \frac{x^2+2x}{\cos 5x+2}, -2 \leq x \leq 4,5, \Delta x = 0,5;$
<b>28</b>	$y = \frac{\ln x+1 +5}{2x+3}, 0,2 \leq x \leq 0,9, \Delta x = 0,15;$
<b>29</b>	$y = \frac{x+\cos 2x}{3x}, 2,7 \leq x \leq 8, \Delta x = 0,7;$
<b>30</b>	$z = \frac{\arcsin 2x+ x }{x^2+1}, 0 \leq x \leq 0,4, \Delta x = 0,2.$

## **2.6. Контрольні запитання.**

2.6.1. Що таке цикл, для чого його використовують.

2.6.2. Який синтаксис циклу `for` у мові Python?

2.6.3. Як описується та виконується циклічна інструкція `while`?

2.6.4. Як можна організувати нескінченні цикли? Наведіть декілька прикладів і поясніть їх.

2.6.5. Як можна вийти з нескінченних циклів?

2.6.6. Чи може оператор циклу не мати тіла? Чому?

2.6.7. Для чого служить оператор переривання `break` та `continue`?

2.6.8. Для чого служить оператор `continue`?

2.6.9. Які необов'язкові блоки містить оператор `if`?

2.6.10. Які операції порівняння визначені в Python?



## **Лабораторна робота №3**

### **Обробка послідовностей при програмуванні на мові Python**

#### **3.1. Мета роботи**

Ознайомитися з особливостями визначення та використання одновимірних та двовимірних масивів, структурною організацією масивів та способів доступу до їх елементів.

#### **3.2. Теоретичні відомості**

##### **Списки**

Список - колекція інших об'єктів, змінюваний об'єкт, вони можуть бути вкладеними, збільшуватися і зменшуватися, містити об'єкти будь-яких типів. Завдяки спискам можна створювати і обробляти в своїх сценаріях структури даних будь-якого ступеня складності. Нижче наводяться основні властивості списків, списки в мові Python - це:

1. *Впорядковані колекції об'єктів довільних типів.*

2. *Доступ до елементів за зсувом.* Можна використовувати операцію індексування для отримання окремих об'єктів зі списку за їхнім зсувом.

3. *Змінна довжина, гетерогенність і довільна кількість рівнів вкладеності.* Списки можуть збільшуватися і зменшуватися безпосередньо (їх довжина може змінюватися), вони можуть містити не тільки односимвольні рядки, а й будь-які інші об'єкти (списки гетерогенні). Списки можуть містити інші складні об'єкти, вони підтримують можливість створення довільної кількості рівнів вкладеності, тому є можливість створювати зі списків списки списків.

4. *Відносяться до категорії змінних об'єктів.*

У табл. 3.1 наведено найбільш типові операції, які застосовують до списків. Коли список визначається виразом (літерально), його записують як послідовність об'єктів в квадратних дужках, розділених комами.

Вкладені списки описують як вкладені послідовності квадратних дужок (рядок 3 у табл. 3.1), а порожні списки визначають як порожню пару квадратних дужок (рядок 1 у табл. 3. 1).

Таблиця 3.1 – Літерали списків і операції

Операція	Інтерпретація
<code>L = []</code>	Пустий список
<code>L = [0, 1, 2, 3]</code>	Чотири елемента з індексами 0...3
<code>L = ['abc', ['def', 'ghi']]</code>	Вкладені списки
<code>L = list('spam')</code> <code>L = list(range(-4, 4))</code>	Створення списку із рядка Створення списку із послідовності цілих чисел
<code>L[i]</code> <code>L[i][j]</code> <code>L[i:j]</code> <code>len(L)</code>	Індекс  зріз довжина
<code>L1 + L2</code> <code>L * 3</code>	Конкатенація дублювання
<code>for x in L: print(x)</code>  <code>3 in L</code>	Обхід у циклі, <code>x</code> - змінна для ітерації перевірка входження
<code>L.append(4)</code>  <code>L.extend([5,6,7])</code> <code>L.insert(1, X)</code>	Методи: додавання елемента «4» у список додавання списку у список додавання списку елементів у вказану позицію
<code>L.index(1)</code>  <code>L.count()</code>	Методи: Визначення зсуву елемента за заданим значенням Підрахунок кількості елементів

L.sort()	Сортування
L.reverse()	Зміна порядку слідування елементів на зворотний
del L[k] del L[i:j] L.pop() L.remove(2)  L[i:j] = []	Зменшення списку видалення елемента видалення групи елементів видалення останнього елемента й повернення його значення видалення елементів з визначеними значеннями
L[i] = 1 L[i:j] = [4,5,6]	Присвоювання за індексом Присвоювання зрізу значень
L=[x**2 for x in range(5)] list(map(ord, 'spam'))	Генератори списків відображення

Оскільки списки є послідовностями, вони, як і рядки, підтримують оператори + і \* (для списків вони так само виконують операції конкатенації і повторення), а в результаті виходить новий список:

```
len([1, 2, 3]) # Довжина
print(len([1, 2, 3])) # 3
l = [1, 2, 3] + [4, 5, 6] # Конкатенація
print(l) # [1, 2, 3, 4, 5, 6]
s = ['Ni!'] * 4 # Повторення
print(s) # ['Ni!', 'Ni!', 'Ni!', 'Ni!']
```

Не можна виконати операцію конкатенації для списку і рядка, якщо попередньо не перетворити список в рядок (використовуючи, наприклад, функцію *str* або оператор форматування %) або рядок в список (за допомогою вбудованої функцією *list*):

```
n = str([1, 2]) + '34' # те саме що '[1, 2]'+ '34'
print(n) # n = [1, 2]34
m = [1, 2] + list('34') # те саме що [1, 2] + ['3', '4']
print(m) # m = [1, 2, '3', '4']
```

*Методи списків.* Об'єкти списків в Python підтримують специфічні методи, багато з яких змінюють сам список безпосередньо:

```
L = ['eat', 'more', 'SPAM!']
L.append('please') # Виклик метода додавання елемента
у кінець списку
print(L) # ['eat', 'more', 'SPAM!', 'please']

L.sort() # Сортування елементів списку ('S' < 'e')
print(L) # ['SPAM!', 'eat', 'more', 'please']
```

*Методи* - це функції, пов'язані з певним типом об'єктів.

Метод *append* додає один елемент (посилання на об'єкт) в кінець списку. На відміну від операції конкатенації, метод *append* приймає один об'єкт-список. За своєю дією вираз *L.append (X)* схожий на вираз *L+[X]*, але в першому випадку змінюється сам список, а в другому створюється новий список. На відміну від операції конкатенації (+), метод *append* не створює новий об'єкт, тому зазвичай він виконується швидше. Існує можливість імітувати роботу методу *append* за допомогою операції присвоювання зрізу: вираз *L[len (L):]=[X]* відповідає виклику *L.append (X)*, а вираз *L [:0] = [X]* відповідає операції додавання в початок списку. В обох випадках видаляється порожній сегмент списку і вставляється елемент *X*, при цьому змінюється сам список *L*, так само швидко, як при використанні методу *append*.

Метод *sort* виконує перевпорядкування елементів в списку. За замовчуванням він використовує стандартні оператори порівняння мови Python (в даному випадку виконується порівнювання рядків) і виконує сортування в порядку зростання значень. Існує можливість змінити порядок сортування за допомогою іменованих аргументів - спеціальних синтаксичних конструкцій типу «*= name == value*», які використовують під час виклику функцій для передачі параметрів налаштування за їхніми іменами. Іменований аргумент *key* у виклику методу *sort* дозволяє визначити власну функцію порівняння, яка приймає один аргумент і повертає значення, яке буде використано в

операції порівняння, а іменований аргумент *reverse* дозволяє виконати сортування не в порядку зростання, а в порядку убутання:

```
L = ['abc', 'ABD', 'aBe']
L.sort() # Сортування з урахуванням регістру символів
print(L) # ['ABD', 'aBe', 'abc']
```

```
L = ['abc', 'ABD', 'aBe']
L.sort(key=str.lower) # Приведення символів до
нижнього регістру
print(L) # ['abc', 'ABD', 'aBe']
```

```
L = ['abc', 'ABD', 'aBe']
L.sort(key=str.lower, reverse=True) # Змінює напрямок
сортування
print(L) # ['aBe', 'ABD', 'abc']
```

Методи *append* і *sort* змінюють сам об'єкт списку і не повертають список у вигляді результату (точніше кажучи, обидва методи повертають значення None). Якщо ви написали інструкцію *L = L.append(X)*, ви не отримаєте змінене значення *L* (насправді ви взагалі втратите посилання на список) - використання атрибутів *append* і *sort*, призводить до зміни самого об'єкта, тому немає ніяких причин виконувати повторне присвоювання. Вбудована функція *sorted* здатна сортувати списки і будь-які інші послідовності, вона повертає новий список з результатом сортування (оригінальний список при цьому не змінюється):

```
L = ['abc', 'ABD', 'aBe']
L_sorted = sorted(L, key=str.lower, reverse=True)
# функція сортування
print(L) # L = ['abc', 'ABD', 'aBe'] не змінився
print(L_sorted) # L_sorted = ['aBe', 'ABD', 'abc']
```

```
L = ['abc', 'ABD', 'aBe']
L_sorted = sorted([x.lower() for x in L], reverse=True)
# елементи попередньо переводяться в нижній регістр
print(L) # L = ['abc', 'ABD', 'aBe'] не змінився
print(L_sorted) # L_sorted = ['abe', 'abd', 'abc']
```

В останньому прикладі перед сортуванням за допомогою генератора списків виконується приведення символів до нижнього регістра, і значення елементів в отриманому списку відрізняються від значень елементів в оригінальному списку. В останньому прикладі виконується сортування тимчасового списку, створеного в процесі сортування. Іноді вбудована функція *sorted* може виявитися більш зручною, ніж метод *sort*.

Метод *reverse* змінює порядок проходження елементів в списку на зворотний, а методи *extend* і *pop* вставляють кілька елементів в кінець списку і видаляють елементи з кінця списку відповідно. Крім того, існує вбудована функція *reversed*, яка нагадує вбудовану функцію *sorted*, але її необхідно обгорнути в виклик функції *list*, тому що вона повертає ітератор:

```
L = [1, 2];
L.extend([3, 4, 5]) # Додавання елементів у кінець
                    # списку
print(L) # [1, 2, 3, 4, 5]

L.pop() # видаляє і повертає останній елемент списку
print(L) # [1, 2, 3, 4]

L.reverse() # Змінює порядок слідування елементів на
            # зворотний
list(reversed(L))
# Вбудована функція сортування в зворотному порядку
print(L) # [4, 3, 2, 1]
```

Інші методи списків дозволяють видаляти елементи з певними значеннями (*remove*), вставляти елементи у визначену позицію (*insert*), визначати зсув елемента за заданим значенням (*index*) тощо:

```
L = ['spam', 'eggs', 'ham']
L.index('eggs') # індекс об'єкта
print(L) # ['spam', 'eggs', 'ham']

L.insert(1, 'toast') # Вставка у потрібну позицію
print(L) # ['spam', 'toast', 'eggs', 'ham']

L.remove('eggs') # видалення елемента із визначеним
                # значенням
```

```
print(L) # ['spam', 'toast', 'ham']

L.pop(1) # видалення елемента у вказаній позиції
print(L) # ['spam', 'ham']
```

Можна використати інструкцію *del* для видалення елемента або зрізу безпосередньо зі списку:

```
L = ['spam', 'toast', 'eggs', 'ham']

del L[0] # видалення одного елемента списку
print(L) # ['toast', 'eggs', 'ham']
del L[1:] # видалення цілого сегмента списку
# Те що і L[1:] = []
print(L) # ['toast']
```

Можна видаляти зрізи списку, привласнюючи їм порожній список ( $L[i:j]=[]$ ) – інтерпретатор спочатку видалить зріз, який визначається зліва від оператора  $=$ , а потім вставить порожній список. Присвоювання порожнього списку за індексом елемента призведе до збереження посилання на порожній список в цьому елементі, а не до його видалення:

```
L = ['Already', 'got', 'one']; L[1:] = []
print(L) # ['Already']
L[0] = []
print(L) # [[]]
```

## Масиви

Для зберігання і обробки в програмах складних видів інформації використовують структурні типи. Їх утворюють шляхом об'єднання простих елементів даних (компонентів). Компоненти можуть бути при цьому однорідними або різнорідними. Необхідність у масивах виникає щоразу, коли в пам'яті потрібно зберігати велику, але скінченну кількість однотипних впорядкованих даних.

*Масив* – це структура даних, яку можна розглядати як набір змінних однакового типу, що мають загальне ім'я.

Доступ до будь-якого елемента масиву здійснюється за його номером. У масиви можна об'єднати результати

експериментів, списки прізвищ співробітників, різні складні структури даних.

У масиві дані різняться своїм порядковим номером (індексом). Якщо кожний елемент масиву визначається за допомогою одного номера, то такий масив називається *одновимірним*, якщо за двома — то *двовимірним*.

*Двовимірний масив* — це таблиця з рядків і стовпчиків. У таблицях перший номер вказує на рядок, а другий — на положення елемента в рядку. Усі рядки таблиці мають однакову довжину.

*Одновимірний масив* (лінійна таблиця) може бути набором чисел, сукупністю символічних даних чи елементів іншої природи (навіть масив масивів). Так само, як і в послідовності, в одновимірному масиві можна вказати елемент з конкретним номером, наприклад  $a_5$ , або записати загальний вигляд елемента, використовуючи як індекс змінну  $i$ , вказуючи діапазон її зміни:  $a[i]$ ,  $i=1, 2, \dots, n$ . Для позначення окремої компоненти до імені масиву додається індекс, який і виділяє потрібну компоненту (наприклад,  $a_1, \dots, a_{50}$ ).

Найменший індекс називається *нижньою межею*, найбільший — *верхньою межею*, а число елементів — *розміром масиву*. Розмір масиву фіксується при описі і в процесі виконання програми не змінюється. Індeksi можна обчислювати. Найчастіше в якості типу індексу використовується обмежений цілий тип.

Базові типи мови Python підтримують можливість створення вкладених конструкцій довільної глибини і в будь-яких комбінаціях (наприклад, зображення матриць, або «багатовимірних масивів»). Це можна зробити за допомогою списку, що містить вкладені списки:

```
M = [[1, 2, 3],
      [4, 5, 6],
      [7, 8, 9]]
# Матриця 3x3 у вигляді вкладених списків
# Вираз в квадратних дужках може
# займати декілька рядків
print(M) # [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```



Тут реалізовано список, який містить три інших списків. В результаті отримано матрицю чисел 3х3. Звернутися до такої структури можна різними способами:

```
print(M[1]) # отримати рядок 2 [4, 5, 6]
print(M[1][2]) # отримати рядок 2, а потім елемент 3 в
               цьому рядку 6
```

## Модуль random

Модуль *random* із стандартної бібліотеки також необхідно імпортувати. Цей модуль дозволяє отримати випадкові дійсні числа в діапазоні від 0 до 1, випадкові цілі числа в заданому діапазоні, послідовність випадкових елементів, виконати випадковий вибір (в тому числі і із списку), тощо:

```
import random

random.random() # 0.844515910136422
random.randint(1, 10) # 9
random.choice(['Life of Brian', 'Holy Grail', 'Life'])
# 'Life'
random.choice([1, 2, 3, 4]) # 3
```

До складу модуля *random* входять такі функції:

1. *random.randint(a,b)* – випадкове ціле число від *a* до *b*;
2. *random.random()* – випадкове число з інтервалу [0, 1);
3. *random.choice(x)* – обирає випадковий елемент послідовності;
4. *random.shuffle(x)* – перемішує елементи послідовності;
5. *random.uniform(a,b)* – випадкове дійсне число від *a* до *b*.

## 3.3. Програма роботи

3.3.1. Запустити на виконання приклади, наведені в теоретичних відомостях та приклади 1-3.

3.3.2. Виконати завдання 1 та 2 згідно варіанту.

## 3.4. Обладнання та програмне забезпечення

3.4.1. Персональний комп'ютер.

3.4.2. Інтерпритатор Python встановлений на ПК

### 3.5. Порядок виконання роботи і опрацювання результатів

#### Приклад 1. Функція *range*

*# Параметр i приймає значення в діапазоні [0, 10)*

```
for i in range(10):  
    print('i =', i)
```

*# Параметр i приймає значення в діапазоні [5, 10)*

```
for i in range(5, 10):  
    print('i =', i)
```

*# Параметр i приймає значення в діапазоні [5, 10) з кроком 2*

```
for i in range(5, 10, 2): print('i =', i)
```

*# Цикл буде повторюватися 3 рази, якщо користувач не завершить його раніше*

```
for i in range(3):  
    response = input('Введіть stop, щоб зупинити цикл  
(інакше що завгодно): ')  
    if response == 'stop':  
        break  
    else:
```

*# цю гілку виконують тільки якщо цикл не був перерваний*

```
    print('Цикл сам був завершений')  
print('Кінець програми')
```

*# Функція reversed дозволяє обходити послідовність в зворотному напрямку*

```
for i in reversed(range(5)):  
    print(i)
```

#### Приклад 2. Зріз списку – отримання групи елементів за їхніми індексами

*# Створення списку чисел*

```
my_list = [5, 7, 9, 1, 1, 2]
```

*# Отримання передостаннього значення*

```
pre_last = my_list [-2] # pre_last == «1»  
print (pre_last)
```

*# Обчислення суми першого і останнього значень*

```
result = my_list[0] + my_list[-1];print(result)
```

```

my_list = [5, 7, 9, 1, 1, 2] # Створення списку чисел
# Отримання зрізу списку від нульового (першого) елемента
# (включаючи його) до третього (четвертого) (не включаючи)
sub_list = my_list[0:3]
print(sub_list) # Виведення отриманого списку
# Виведення елементів списку від другого до передостаннього
print(my_list[2:-2])
# Виведення ел-тів списку від 4-ого (5-ого) до 5-ого (6-ого)
print(my_list[4:5])

```

```

my_list = [5, 7, 9, 1, 1, 2]
# Вибір кожного другого ел-та списку (починаючи з першого),
# не включаючи останній елемент
sub_list = my_list[0:-1:2]
print(sub_list) # виведення отриманого списку
# Виведення елементів від 2-ого (3-ього) до передостаннього
# з кроком 2
print(my_list[2:-2:2])
# Виведення ел-тів списку, крім першого, в зворотному
порядку
print(my_list[-1:0:-1])

```

```

my_list=[5, 7, 9, 1, 1, 2]
# Виведення елементів списку від 2-ого (3-ього) значення до
кінця
print(my_list[2:])
# Виведення всіх ел-тів списку від початку до
передостаннього ел-та
print(my_list[:-2])
# Виведення всіх елементів списку в зворотному порядку
print(my_list[::-1])

```

**Приклад 3.** Нехай згенеровано список із цілих випадкових чисел та нулів  $[a_1, \dots, a_n]$ . Написати програму визначення елементів, розміщених після першого нульового. Вивести на екран початковий та отриманий списки

```
import random
```

```

arr = random.sample(range(-6, 6), 12)
print("our random list: ", arr)

```

```

flag = 0
while flag == 0:
    if 0 in arr: # перевіряємо чи є 0 в списку
        first_zero_index = arr.index(0) # індекс першого 0
        flag = 1;
    else:
        print("we have not at list one zero in list: ")
arr1 = []
if flag == 1:
    for i in arr[first_zero_index:]:
        arr1.append(i)
print(arr1)
Результат:
our random list: [-4, -6, 4, 0, 2, 5, 3, -1, -3, -5, 1, -2]
[0, 2, 5, 3, -1, -3, -5, 1, -2]

```

### Завдання 1. Одновимірні масиви (вектори)

Нехай задано список різних випадкових чисел  $[a_1, \dots, a_n]$ , значення  $n$  визначає користувач програми. Використовуючи генератор випадкових чисел, заповнити список  $[a_1, \dots, a_n]$  елементами:

- а) дійсними числами, які лежать в діапазоні від 0 до 1;
- б) цілими додатними та від'ємними числами, які лежать в діапазоні від  $-10$  до  $10$  включно;
- в) цілими додатними числами, які лежать в діапазоні від 0 до 50 включно.

Варіант:

1. Задано список (б). Написати програму формування іншого списку, в якому усі елементи, які передують найбільшому від'ємному елементу, замінити на значення їх квадратів.

2. Задано список (б). Написати програму формування іншого списку, в якому, якщо еленти заданого списку не утворюють послідовності, яка зменшується, то замінити його від'ємні елементи одиницями.

3. Задано список (б). Написати програму формування іншого списку, в якому переставити елементи таким чином, щоб спочатку були розташовані всі невід'ємні елементи, а вкінці - від'ємні елементи.

4. Задано список (а). Написати програму формування іншого списку, в якому елементи сформовані таким чином  $[a_1, a_{n+1}, a_2, a_{n+2}, \dots, a_n, a_{2n}]$ .

5. Задано список (б). Перевірити чи утворюють елементи заданого масиву послідовність, яка чітко зменшується або збільшується. Вивести відповідне повідомлення.

6. Задано список (а). За заданими дійсними числами  $a_0, a_1, \dots, a_n, t$  обчислити значення багаточлена  $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$  в точці  $t$ .

7. Задано список (б). Написати програму визначення суми всіх елементів, розміщених до останнього додатного елемента включно.

8. Задано список (в). Написати програму визначення суми лише тих елементів, які є непарними числами.

9. Задано список (б). Написати програму визначення добутку елементів, розміщених між максимальним за модулем та мінімальним за модулем елементами.

10. Задано список (б). Написати програму формування іншого списку, в якому елементи сформовані таким чином, що нульові елементи перенесено у хвіст списку.

11. Задано список (б). Написати програму визначення суми модулів елементів, розміщених після першого нульового.

12. Задано список (б). Написати програму визначення суми елементів, розміщених між першим та другим від'ємними елементами.

13. Задано список (б). Написати програму формування іншого списку, в якому елементи сформовані таким чином: якщо хоча б одне значення елементів належить проміжку  $[x, y]$ , то всі елементи, які не належать цьому проміжку, замінити на  $z$ .

14. Задано список (б). Написати програму визначення суми чисел цієї послідовності, розташованих між максимальним і мінімальним числами (до суми включити й обидва цих числа).

15. Задано різні два списки різних цілих випадкових чисел  $[a_1, a_2, \dots, a_{3n}]$  (0.6). Написати програму визначення найменшого серед тих чисел першого списку, які не входять до другого (вважаючи, що хоча б одне таке число існує).

16. Задано список (б). Написати програму формування іншого списку, в якому всі від'ємні елементи списку перенести в його початок, а всі інші – в кінець, зберігаючи початкове взаємне розміщення як серед від'ємних елементів, так і серед інших елементів.

17. Задано список (в). Написати програму формування іншого списку, в якому  $[a_1, a_1+a_2, a_1+a_2+a_3, \dots, a_1+a_2+\dots+a_n]$ .

18. Задано список (в). Написати програму формування іншого списку, в якому виконаний циклічний зсув усіх елементів на  $k$  позицій вліво, наприклад, для  $k=1$  маємо  $a_2, \dots, a_n, a_1$ .

19. Задано список (в). Написати програму формування двох списків  $[x_1, \dots, x_n]$  і  $[y_1, \dots, y_n]$ , в яких елементи сформовані таким чином  $[a_1, a_3, \dots, a_{2n-1}]$  і  $[a_2, a_4, \dots, a_{2n}]$ .

20. Задано різні два списки  $[x_1, \dots, x_n]$  і  $[y_1, \dots, y_n]$  (в). Написати програму формування списку, елементи якого дорівнюють відповідним значенням  $[x_1, y_1, \dots, x_n, y_n]$ .

21. Задано список  $[a_1, a_2, \dots, a_{3n}]$  (в). Написати програму формування іншого списку, в якому елементи дорівнюють середньому арифметичному значенню трьох наступних елементів списку.

22. Задано список випадкових чисел з нулів та одиниць  $[a_1, \dots, a_{3n}]$ . Написати програму пошуку найбільшої за довжиною ділянки заданої послідовності, яка заповнена одиницями. Вивести на екран індекси початку та кінця знайденої ділянки.

23. Задано список (в). Написати програму пошуку всіх локальних мінімумів та максимумів в списку (елемент називається локальним мінімумом/максимумом, якщо в нього немає сусідів, які менші/більші за нього).

24. Задано різні два списки цілих випадкових чисел  $[x_1, \dots, x_n]$  і  $[y_1, \dots, y_n]$  (в). Написати програму формування двох списків, елементи яких сформовані за правилами:  $x_1 = \max(x_1, y_1)$ ;  $y_1 = \min(x_1, y_1)$ .

25. Задано список (б). Написати програму визначення кількості елементів, що передують першому від'ємному елементу та їх значення належать проміжку  $[x, y]$  (значення  $x, y$

введені з клавіатури).

## **Завдання 2. Двовимірні масиви (матриці)**

Використовуючи генератор випадкових чисел, заповнити список  $[[a_{11}, \dots, a_{1n}], \dots, [a_{m1}, \dots, a_{mn}]]$ , де  $n$  – кількість літер в імені,  $m$  – кількість літер в прізвищі, елементами:

- а) дійсними числами, які лежать в діапазоні від 0 до 1;
- б) цілими додатними та від'ємними числами, які лежать в діапазоні від  $-10$  до  $10$  включно;
- в) цілими додатними числами, які лежать в діапазоні від 0 до 20 включно;

- г) дійсними числами, які лежать в діапазоні від  $-10$  до  $10$ ;

Елементи головної діагоналі розташовані з лівого верхнього кута матриці до правого нижнього; друга, зворотна діагональ матриці є побічною.

1. Задано список (б). Написати програму, яка змінить місцями два стовпчики: стовпчик, який містить максимальний від'ємний елемент, і стовпчик, який містить мінімальний додатний елемент матриці.

2. Задано список (в). Написати програму, яка змінить місцями перший рядок з рядком, що містить максимальний елемент матриці.

3. Задано список (б). Написати програму, яка змінить місцями останній рядок з рядком, який містить мінімальний додатний елемент матриці.

4. Задано список (б),  $n=m$ . Написати програму, яка визначить, чи є задана квадратна матриця симетричною відносно головної діагоналі.

5. Згенерувати список,  $n=m$ , який визначає квадратну матрицю – магічний квадрат (тобто така, в якій суми елементів у всіх рядках і стовпчиках є однаковими). Вивести на екран отриманий результат.

6. Задано список (б) (якщо треба згенерувати відповідний список). Написати програму, яка визначить добуток від'ємних елементів другого рядка та кількість елементів в другому стовпчику, які не кратні «5».

7. Задано список (б). Написати програму, яка визначить новий список, в якому кожен елемент матриці помножений на мінімальний за абсолютною величиною елемент у поточному стовпчику.

8. Задано список (з). Написати програму, яка змінить місцями останній стовпчик і стовпчик, який містить мінімальний додатний елемент матриці.

9. Задано список (б). Написати програму, яка визначити максимальний елемент третього стовпчика та суму непарних елементів першого рядка.

10. Задано список (з). Написати програму, яка визначить рядок, сума елементів якого мінімальна. Вивести на екран початкову матрицю та визначений рядок.

11. Задано список (а). Написати програму, яка визначить суму елементів кожного стовпчика (результат записати в інший список).

12. Задано список (в). Написати програму, яка визначить суму елементів тих рядків, у яких на побічній діагоналі стоять невід'ємні числа.

13. Задано список (в). Написати програму, яка сформує новий список, в якому всі елементи матриці вище побічної діагоналі є нульовими.

14. Задано список (з). Написати програму, яка змінить місцями перший стовпчик і стовпчик, який містить мінімальний за абсолютною величиною елемент матриці.

15. Задано список (з). Написати програму, яка визначить суму елементів парних рядків, записати результат у новий список.

16. Задано список (з). Написати програму, яка визначить суму елементів в кожному стовпчику, записати результат у новий список.

17. Задано список (а). Написати програму, яка змінить місцями перший рядок і рядок, який містить мінімальний елемент матриці.

18. Задано список (з). Написати програму, в якій кожен елемент матриці помножено на максимальний елемент у



поточному рядку.

19. Задано список (e). Написати програму, яка змінить матрицю таким чином, щоб всі елементи нижче головної діагоналі були нульовими.

20. Задано список (e). Написати програму, яка сформує новий список в якому змінено місцями два рядки: рядок, що містить максимальний елемент, і рядок, що містить мінімальний елемент.

21. Задано список (b). Написати програму, яка визначає добуток від'ємних парних чисел першого стовпчика.

22. Задано список (b). Написати програму, яка сформує новий список в якому змінено місцями останній рядок і рядок, що містить мінімальний за абсолютною величиною елемент.

23. Задано список (a). Написати програму, яка визначає скалярний добуток рядка, в якому знаходиться найбільший елемент матриці, на стовпчик із найменшим елементом.

24. Задано список (b). Написати програму, яка визначає суму чисел, які кратні трьом, та суму від'ємних чисел елементів третього стовпчика.

25. Задано список (b). Написати програму, яка визначає добуток парних елементів другого стовпчика та добуток непарних елементів другого рядка матриці.

### **3.6. Контрольні запитання.**

3.6.1. Що таке одновимірний масив? Для чого використовують одновимірні масиви? Як їх описують в Python?

3.6.2. Як в програмі використати значення конкретного елемента одновимірного масиву?

3.6.3. Для чого в програмах використовуються двовимірні масиви? Як їх описують в Python?

3.6.4. Скільки індексів характеризують конкретний елемент двовимірного масиву?

3.6.5. Чи може список в Python складатись із елементів різних типів?

3.6.6. Як видалити елемент зі списку за значенням?

3.6.7. Як додати елемент у вказану позицію списку?

3.6.8. Для чого призначена функція range?

3.6.9. Назвіть методи впорядкування елементів у списку.

## Лабораторна робота №4

### Обробка послідовностей при програмуванні на мові Python. Рядки. Множини

#### 4.1. Мета роботи

Познайомитися з такими об'єктами як рядки, множини, словники, кортежі мови Python.

#### 4.2. Теоретичні відомості

##### Рядки

*Рядки* – це впорядковані послідовності символів, що використовують для зберігання і подання текстової інформації (символів і слів, наприклад, ваше ім'я, змісту текстових файлів, завантажених в пам'ять, адрес в Інтернеті, програми на Python тощо). Рядки володіють потужним набором засобів для їх обробки. У Python відсутній спеціальний тип для подання одного символу, тому в разі необхідності використовуються односимвольні рядки.

Рядки відносять до категорії незмінних послідовностей, в тому сенсі, що символи, які вони містять, мають певний порядок розміщення зліва направо і самі рядки неможливо змінити. Рядки – це представник великого класу об'єктів, які називають послідовностями. Зверніть увагу на операції над послідовностями, наведені тут, тому що вони схожим чином працюють і з іншими типами послідовностей, такими як списки і кортежі, які ми будемо розглядати пізніше. У таблиці наведені найбільш типові літерали рядків і операцій

*Таблиця 4.1. Типові літерали рядків та операції над ними*

Операція	Інтерпретація
<code>S = ""</code>	Пустий рядок
<code>S = "spam's"</code>	Рядок у лапках
<code>S = 's\np\ta\x00m'</code>	Екрановані послідовності
<code>block = """..."""</code>	Блоки в потрійних лапках
<code>S = r'\temp\spam'</code>	Неформатовані рядки
<code>S = b'spam'</code>	Рядки байтів

<code>S1 + S2</code> <code>S * 3</code>	Конкатенація, повторення
<code>S[i]</code> <code>S[i:j]</code> <code>len(S)</code>	Звернення до символу за індексом Витяг підрядку (зрізу) Довжина
<code>"a %s parrot" % kind</code>	Вираз форматування рядка
<code>"a {0} parrot".format(kind)</code>	Метод форматування рядка
<code>S.find('pa')</code>	Виклик методу рядків: пошук
<code>S.rstrip()</code>	Видалення провідних й кінцевих символів пробілу
<code>S.replace('pa', 'xx')</code>	Заміна
<code>S.split(',')</code>	Разбиття за символом, який є роздільником
<code>S.isdigit()</code>	Перевірка вмісту
<code>S.lower()</code>	Перетворення регістра символів
<code>S.endswith('spam')</code>	Перевірка закінчення рядка
<code>'spam'.join(strlist)</code>	Формування рядка зі списку
<code>S.encode('latin-1')</code>	Кодування рядків Юнікоду
<code>for x in S: print(x)</code> <code>'spam' in S</code>	Обхід в циклі Перевірка на входження

Порожній рядок має вигляд пари лапок (або апострофів), між якими нічого немає. Для роботи з рядками підтримуються операції над виразами, такі як конкатенація (об'єднання рядків), виділення підрядка, вибірка символів за індексами (за зсувом від початку рядка) тощо. Python пропонує ряд методів, які реалізують різні завдання роботи з рядками.

### Множина

Множина – невпорядкований набір унікальних і незмінних об'єктів, який підтримує операції, що відповідають математичній теорії множин. За визначенням, кожний елемент може бути присутнім в множині в одному екземплярі незалежно від того, скільки разів він буде доданий. Оскільки множина є набором інших об'єктів, їй притаманні деякі властивості списків

і словників (множини підтримують ітерації, при необхідності можуть змінюватися в розмірах і містити об'єкти різних типів).

Щоб створити об'єкт множини, потрібно передати послідовність або інший об'єкт, що підтримує можливість ітерацій за його вмістом, вбудованій функції `set`:

```
x = set('abcde')
```

Функція повертає об'єкт-множину, який містить всі елементи об'єкта, переданого функції:

```
{'e', 'c', 'b', 'd', 'a'}
```

Множини, створені таким способом, підтримують звичайні математичні операції над множинами за допомогою операторів виразів (перетин, об'єднання, різниця, симетрична різниця множин, перевірка входження в множину, надмножина, підмножина):

```
x = set('abcde')
```

```
y = set('bdxyz')
```

```
'e' in x # Перевірка входження в множину. Результат True
```

```
x - y # Різниця множин. Результат set(['a', 'c', 'e'])
```

```
x | y # Об'єднання множин. Результат set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])
```

```
x & y # Перетин множин. Результат set(['b', 'd'])
```

```
x ^ y # Симетрична різниця (XOR). Результат set(['a', 'c', 'e', 'y', 'x', 'z'])
```

```
x > y, x < y # Надмножина, підмножина. Результат (False, False)
```

```
S1 = {1, 2, 3, 4}
```

```
S1 & {1, 3} # Перетин. Результат {1, 3}
```

```
{1, 5, 3, 6} | S1 # Об'єднання. Результат {1, 2, 3, 4, 5, 6}
```

```
S1 - {1, 3, 4} # Різниця. Результат {2}
```

```
S1 > {1, 3} # Надмножина. Результат True
```

Для множин визначені такі операції:

+ – об'єднання множин;

– – різниця множин;

- \* – перетин множин;
- = – перевірка еквівалентності двох множин;
- <> – перевірка нееквівалентності двох множин;
- <= – перевірка, чи є ліва множина підмножиною правої множини;
- >= – перевірка, чи є права множина підмножиною лівої множини;
- in – перевірка, чи входить елемент, який зображено зліва, в множину, вказану справа.

Результатом операції об'єднання, різниці або перетину є відповідна множина, інші операції дають результат логічного типу. Нехай змінні  $x$  і  $y$  зберегли свої значення, присвоєні в попередньому прикладі:

```
x = set('abcde')
y = set('bdxyz')
z = x.intersection(y) # x & y.Результат set(['b', 'd'])
z.add('SPAM') # додає один елемент. Результат set(['b', 'd', 'SPAM'])
z.update(set(['X', 'Y'])) # об'єднання множин. Результат set(['Y', 'X', 'b', 'd', 'SPAM'])
z.remove('b') # видалить один елемент. Результат set(['Y', 'X', 'd', 'SPAM'])
```

Будучи ітерованими контейнерами, множини можна передавати функції *len*, використовувати в циклах *for* і в генераторах списків. Однак множини є неупорядкованими колекціями, тому не підтримують операції над послідовностями, такі як індексування і витяг зрізу:

```
for item in set('abc'):
    print(item * 3)
```

```
S = set([1, 2, 3])
S | set([3, 4]) # Оператори виразу вимагають щоб обидва операнди були множинами. Результат {1, 2, 3, 4}

# S | [3, 4] # TypeError: unsupported operand type(s) for |: 'set' and 'list'
S.union([3, 4]) # Результат {1, 2, 3, 4}
```

```
S.intersection((1, 3, 5)) # Результат {1, 3}
S.issubset(range(-5, 5)) # Результат True
```

*Літерали множин.* В Python існує можливість використовувати вбудовану функцію *set* для створення множин, при цьому є форма літералів множин, в яких використовують фігурні дужки, раніше зарезервовані для літералів словників. В Python 3.0 такі інструкції є еквівалентними:

```
y = set([1, 2, 3, 4]) # Виклик функції set
z = {1, 2, 3, 4} # Літерал множини
print(y, z)
```

```
set('spam') # {'a', 'p', 's', 'm'}
```

```
S = {'s', 'p', 'a', 'm'}
S.add('alot') # {'a', 'p', 's', 'm', 'alot'}
```

Конструкція `{}` створює пустий словник (про них мова піде далі). Щоб створити пусту множину, треба викликати вбудовану функцію *set*; результат операції виведення пустої множини має інший вигляд:

```
w = set() # Пуста множина
print(type(w)) # <class 'set'>
```

```
s = {} # Літерал {} позначає пустий словник
print(type(s)) # <class 'dict'>
```

Множини можуть включати об'єкти тільки незмінних типів: списки і словники не можна додавати в множини, однак можна використати кортежі, якщо з'явиться необхідність зберігати складові значення. В операціях над множинами кортежі порівнюють за своїм повним значенням:

```
S = {1.23}
# Додаватися можуть тільки незмінні об'єкти
S.add([1, 2, 3]) # Результат TypeError: unhashable
type: 'list'
```

```
S.add({'a':1}) #TypeError: unhashable type: 'dict'
```

```

S.add((1, 2, 3)) # кортеж можна додати
print(S) # {1.23, (1, 2, 3)}
U = S | {(4, 5, 6), (1, 2, 3)}
print(U) # Об'єднання: теж, що й S.union(...) {1.23,
(4, 5, 6), (1, 2, 3)}
(1, 2, 3) in S
True
(1, 4, 3) in S
False

```

## Кортежі

Списки та рядки мають багато спільного. Наприклад, для них можна одержати елемент за індексом або застосувати операцію зрізу. Це два приклади послідовностей. Ще одним убудованим типом послідовностей є кортеж (**tuple**). Кортеж складається з набору значень, розділених комою, наприклад:

```

t = 12345, 54321, 'hello!'
print(t[0]) # 12345
print(t) # (12345, 54321, 'hello!')

u = t, (1, 2, 3, 4, 5)
print(u) # ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

```

При виведенні кортежі завжди вкладаються в круглі дужки, для того, щоб вкладені кортежі сприймалися коректно. Уводити їх можна як у круглих дужках, так і без них, хоча в деяких випадках дужки необхідні (якщо кортеж є частиною складнішого виразу).

Кортежі мають безліч застосувань: пари координат (x, y), запис у базі даних і т. д. Для кортежів, як і для рядків, неприйнятні зміни: не можна присвоїти значення елементу кортежу (проте можна імітувати таку операцію за допомогою зрізів та наступного об'єднання).

Для конструювання порожнього елемента кортежу або кортежу, що містить один елемент, доводиться йти на синтаксичні хитрування. Порожній кортеж створюється за допомогою порожньої пари дужок, кортеж з одним елементом – за допомогою значення та наступної за ним **коми** (недостатньо просто помістити значення в дужки). Наприклад:

```
empty = ()
print(len(empty)) # 0
print(empty) # ()

singleton = 'hello',
print(len(singleton)) # 1
print(singleton) # ('hello',)
```

Інструкція `t = 12345, 54321, 'hello!'` – приклад пакування в кортеж. Значення 12345, 54321 та 'hello!' пакуються разом в один кортеж. Також можлива обернена операція – розпакування кортежу:

```
x, y, z = t
```

Розпакування кортежу вимагає, щоб ліворуч стояло стільки змінних, скільки елементів у кортежі. Значимо, що багаторазове присвоювання є лише комбінацією пакування та розпакування кортежу.

Іноді є корисним розпакування списку:

```
a = ['spam', 'eggs', 100, 1234]
a1, a2, a3, a4 = a
print('a1 = ', a1, '\na2 = ', a2, '\na3 = ', a3, '\na4 = ', a4)
""" Результат
a1 = spam
a2 = eggs
a3 = 100
a4 = 1234
"""
```

Як зазначалося, у кортежах, як і в рядках, не допускаються зміни. Однак, на відміну від рядків, кортежі можуть містити об'єкти, які можна змінити за допомогою методів:

```
t = 1, ['foo', 'bar']
print(t) # (1, ['foo', 'bar'])
t[1] = []
"""
```

```
TypeError: 'tuple' object does not support item
assignment
"""
```



```
t[1].append('baz')
print(t) # (1, ['foo', 'bar', 'baz'])
```

## Словники

Ще один убудований в *Python* тип даних – словник (**dictionary**) – часто називають асоціативним масивом. На відміну від послідовностей, що індексуються діапазоном чисел, доступ до елементів словника провадиться за ключем будь-якого типу, що не дозволяє внесення змін, – рядки та числа завжди можуть бути ключами.

Кортежі використовуються як ключі, якщо вони містять рядки, числа та кортежі, що задовольняють це правило. Не можна застосовувати списки, тому що їх можна змінити (не створюючи нового об'єкта-списку) за допомогою, наприклад, методу **append()**.

Найліпше можна уявити словник як неупорядковану множину пар **key:value** (ключ:значення), причому кожен ключ є унікальним у межах одного словника. Фігурні дужки **{}** створюють порожній словник. Поміщаючи список пар **key:value**, розділених комами, у фігурні дужки, ви вказуєте початковий вміст словника.

Такий самий вигляд матиме словник при виведенні.

Основні операції над словником – збереження з указаним ключем та виведення за ним значення. Можна також видалити пари **key:value** за допомогою інструкції **del**. Під час зберігання нового значення із ключем, що вже використовується, старе значення видаляється. При спробі прочитати значення за ключем, якого немає в словнику, генерується виняткова ситуація **KeyError**.

Метод **keys()** для словника повертає список усіх використовуваних ключів у довільному порядку (якщо необхідно, щоб список був упорядкований, застосовують до нього метод **sort()**).

### Методи словників

**dict.clear()** – очищає словник.

**dict.copy()** – повертає копію словника.

**dict.fromkeys (seq [, value])** – створює словник з ключами з seq і значенням value (за замовчуванням None).

**dict.get(key [, default])** – повертає значення ключа, але якщо його немає, не викидає виняток, а повертає default (за замовчуванням None).

**dict.items()** – повертає пари (ключ, значення).

**dict.keys()** – повертає ключі в словнику.

**dict.pop(key [, default])** – видаляє ключ і повертає значення. Якщо ключа немає, повертає default (за замовчуванням кидає виняток).

**dict.popitem()** – видаляє і повертає пару (ключ, значення). Якщо словник порожній, кидає виняток KeyError. Пам'ятайте, що словники не впорядковані.

**dict.setdefault(key [, default])** – повертає значення ключа, але якщо його немає, не кидає виняток, а створює ключ із значенням default (за замовчуванням None).

**dict.update([other])** – оновлює словник, додаючи пари (ключ, значення) з other. Існуючі ключі перезаписуються. Повертає None (не новий словник!).

**dict.values()** – повертає значення в словнику.

Наведемо простий приклад використання словника як телефонного довідника:

```
price = {'apple': 40980, 'dell': 34139}
price['hp'] = 34127 # Доповнення словника
print(price) # {'apple': 40980, 'dell': 34139, 'hp':
34127}
print(price['apple']) # 4098
del price['dell'] # Видалення пари зі словника
price['asus'] = 4127
print(price) # {'apple': 40980, 'hp': 34127, 'asus':
4127}
print(price.keys()) # dict_keys(['apple', 'hp',
'asus'])
```

## Генератори

Конструкцію генератора множин розміщують у фігурні дужки. Генератор множин виконує цикл і збирає результати виразу в кожній ітерації - доступ до значення в поточній ітерації

забезпечує змінна циклу. Результатом роботи генератора є нова множина:

```
new_set = {x ** 2 for x in [1, 2, 3, 4]} # Генератор
множин
print(new_set) # {16, 1, 4, 9}
```

Генератор множин повертає «нову множину, яка містить квадрати значень X, для кожного X зі списку». В генераторах можна також використовувати інші види ітерованих об'єктів, наприклад рядки:

```
s = {x for x in 'spam'} # Те ж саме, що і set('spam')
print(s) # {'a', 'p', 's', 'm'}

w = {c * 4 for c in 'spam'} # Множина результатів
виразу
print(w) # {'ssss', 'aaaa', 'pppp', 'mmmm'}
p = {c * 4 for c in 'spamham'}
print(p) # {'ssss', 'aaaa', 'hhhh', 'pppp', 'mmmm'}
```

Оскільки елементи множин є унікальними, їх можна використовувати для фільтрації повторюваних значень в інших наборах. Для цього досить перетворити набір значень у множину, а потім виконати зворотне перетворення (множина є ітерованим об'єктом, тому її можна передавати функції *list*):

```
L = [1, 2, 1, 3, 2, 4, 5]
set(L)
L = list(set(L)) # видалення значень, які повторюються
print(L) # [1, 2, 3, 4, 5]
```

Починаючи з версії 2.0 мови *Python*, з'явилися додаткові можливості конструювання списків без використання засобів функціонального програмування. Такі визначення списків (спискові включення) записуються зі застосуванням у квадратних дужках виразу та наступних за ним блоків **for** та **if**:

```
vec = [2, 4, 6]
vec_first = [3 * x for x in vec]
print(vec_first) # [6, 12, 18]
vec_second = [[x, x ** 2] for x in vec]
print(vec_second) # [[2, 4], [4, 16], [6, 36]]
```

Елементи можна фільтрувати, указавши умову ключовим словом **if**:

```
vec = [2, 4, 6]
x = [3 * x for x in vec if x > 3] # [12, 18]
y = [3 * x for x in vec if x < 2] # []
```

Вираз, що дає в результаті кортеж, записують у круглих дужках:

```
[x, x**2 for x in vec] # SyntaxError: invalid syntax
[x, x**2 for x in vec] # SyntaxError: invalid syntax
[(x, x**2) for x in vec] # [(2, 4), (4, 16), (6, 36)]
```

Якщо в конструкторі вказано кілька блоків **for**, елементи другої послідовності перебираються для кожного елемента першої і т. д., тобто перебираються всі комбінації елементів:

```
vec1 = [2, 4, 6]
vec2 = [4, 3, -9]
[x * y for x in vec1 for y in vec2] # [8, 6, -18, 16,
12, -36, 24, 18, -54]
[x + y for x in vec1 for y in vec2] # [6, 5, -7, 8, 7,
-5, 10, 9, -3]
```

**Генераторів словників** дуже схожі на генератори списків і множин.

```
d = {a: a ** 2 for a in range(7)}
print(d) # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

### 4.3. Програма роботи

4.3.1. Створити та запустити на виконання приклади 1-2.

4.3.2. Виконати завдання 1-3 згідно варіанту.

### 4.4. Обладнання та програмне забезпечення

4.4.1. Персональний комп'ютер.

4.4.2. Інтерпритатор Python IDLE встановлений на ПК

### 4.5. Порядок виконання роботи і опрацювання результатів

**Приклад 1.** Операції з множинами (*set*; *frozenset*, яка діє так само, як функція *set*, але формує незмінну множину)

```

my_set = {4, 5, 1, 2}
# Кількість елементів множини
print('len({}) = {}'.format(my_set, len(my_set)))

# Перевірка входження елемента
print(4 in my_set)
print(3 not in my_set)
print(9 in my_set)

# Чи перетинаються множини
print({3, 4, 5}.isdisjoint({8, 1, 0}))
print({3, 4, 5}.isdisjoint({1, 2, 3}))

# Перевірка включення однієї множини в іншу
print({1, 7, 9}.issubset({1, 2, 3, 7, 9}))
print({1, 7, 9} <= {1, 2, 3, 7, 9})
print({1, 7, 9, 2, 3} <= {1, 2, 3, 7, 9})

# Перевірка чіткого включення
print({1, 7, 9} < {1, 2, 3, 7, 9})
print({1, 7, 9, 2, 3} < {1, 2, 3, 7, 9})

# Перевірка включення однієї множини в іншу
print({1, 2, 3, 4}.issuperset({1, 2}))
print({1, 2, 4, 4} >= {1, 2})
print({1, 2, 3, 4} >= {1, 2, 3, 4})

# Перевірка чіткого включення
print({1, 2, 4, 4} > {1, 2})
print({1, 2, 3, 4} > {1, 2, 3, 4})

# Об'єднання множин
print({1, 3}.union({2, 3, 4}))
print({1, 3} | {2, 3, 4})

# Перетин множин
print({1, 3}.intersection({2, 3, 4}))
print({1, 3} & {2, 3, 4})

# Різниця множин
print({1, 2, 3, 4}.difference({3, 4, 5}))
print({1, 2, 3, 4} - {3, 4, 5})

```

```
# Симетрична різниця
print({1, 2, 3, 4}.symmetric_difference({3, 4, 5, 6}))
print({1, 2, 3, 4} ^ {3, 4, 5, 6})

# Копіювання множини
my_set = set('chars')
copy = my_set.copy()
print(copy)
```

## Приклад 2. Операції зі словниками

```
phonebook = {
    'Олександр': '123-032-846',
    'Анатолій': '432-917-333',
    'Вадим': '345-120-422',
    'Андрій': '111-890-532', # остання кома ігнорується
}
# len(d) – кількість елементів
print(phonebook, '\n', len(phonebook), 'entries found')
```

**d[key]** – отримання значення з ключем *key*. Якщо ключ не існує, відображення реалізує спеціальний метод `__missing__` (*self*, *key*): якщо ключ не існує і метод `__missing__` не визначений, видається виняток `KeyError`

```
print(phonebook['Вадим']) # 345-120-422
print(phonebook['Олег']) # KeyError: 'Олег'
```

**d[key]=value** – змінити значення або створити пару ключ-значення, якщо ключ не існує

```
phonebook['Андрій'] = '222-890-532'
phonebook['Олег'] = '432-850-133'
print(phonebook, '\n', len(phonebook), 'entries found')
```

**key in d**, **key not in d** – перевірка наявності ключа в відображенні

```
for person in ('Анатолій', 'Вадим', 'Микола'):
    if person in phonebook:
        print(person, 'is in the phonebook')
    else:
        print('No entry found for', person)
```

## **Завдання 1 (обробка рядків).**

### **Варіанти:**

1. Задано речення. Скласти програму, яка визначає і виводить на екран найбільшу кількість прогалин, розташованих підряд.

2. Задано текст, в якому є дві і більше однакові літери. Скласти програму, яка визначає і виводить на екран найбільшу кількість однакових символів, розташованих підряд.

3. Задано слово. Скласти програму, яка визначає і виводить на екран кількість різних символів в ньому.

4. Задано слово, в якому є дві і більше однакові літери. Скласти програму, яка їх визначає і виводить на екран.

5. Задано три слова. Скласти програму, яка визначає і виводить на екран літери, які не повторюються в них.

6. Задано два слова. Скласти програму, яка визначає і виводить на екран ті літери слів, які є тільки в одному з них (в тому числі повторювані). Наприклад, якщо задано слова «процесор» та «інформація», то відповідь має вигляд: п е с і н ф м а і я.

7. Задано два слова. Скласти програму, яка визначає і виводить ті літери слів, які зустрічаються в обох словах тільки один раз. Наприклад, якщо задано слова «процесор» та «інформація», то відповідь має вигляд: п е ф м а я.

8. Задано два слова. Скласти програму, яка визначає, чи можна з літер першого з них здобути друге. Розглянути такі варіанти: 1) повторювані літери другого слова можуть в першому слові не повторюватися; 2) кожна літера другого слова повинна входити у перше слово стільки раз, скільки вона входить у друге.

9. Задано три слова. Скласти програму, яка визначає і виводить на екран ті літери слів, які є лише в одному зі слів. Розглянути такі варіанти: 1) повторювані літери кожного слова розглядаються; 2) повторювані літери кожного слова не розглядаються.

10. Задано три слова. Скласти програму, яка визначає і виводить на екран їх загальні літери. Повторювані літери кожного слова не розглядати.

11. Задано речення з десяти слів. Скласти програму, яка визначає і виводить на екран заповнений ними список. Розділових знаків в реченні немає.

12. Задано речення. Скласти програму, яка визначає і виводить на екран речення, в якому слова розташовані в зворотному порядку (наприклад, речення «мама мила раму» буде змінено на «раму мила мама»).

13. Задано речення. Скласти програму, яка визначає і виводить на екран речення, в якому слова змінено місцями (наприклад, замість першого слова розташовано останнє, а замість останнього – перше).

14. Задано речення. Скласти програму, яка визначає і виводить на екран всі його слова, відмінні від слова «привіт».

15. Задано речення. Скласти програму, яка визначає і виводить на екран: а) кількість слів, які розпочинаються з літери «н»; б) кількість слів, які закінчуються на літеру «р».

16. Задано речення. Скласти програму, яка визначає і виводить на екран: слова а) які розпочинаються і закінчуються на одну і ту ж літеру; б) які містять три літери «е»; в) які містять хоча б одну літеру «о».

17. Задано речення. Скласти програму, яка визначає і виводить на екран будь-яке його слово, що розпочинається на літеру «к».

18. Задано речення. Скласти програму, яка визначає і виводить на екран довжину його самого короткого слова.

19. Задано речення. Скласти програму, яка визначає і виводить на екран його найдовше слово (прийняти, що таке слово є одним).

20. Задано речення. Скласти програму, яка визначає і виводить на екран, чи правдивим є твердження, що його найдовше слово має більше 10 символів.

21. Задано речення. Скласти програму, яка визначає і виводить на екран всі слова в порядку неспадання їх довжин.



22. Задано речення. Скласти програму, яка визначає і виводить на екран всі слова, які зустрічаються в реченні один раз.

23. Задано речення. Скласти програму, яка визначає і виводить на екран всі його різні слова.

24. Задано речення, в якому є тільки два однакових слова. Скласти програму, яка визначає їх і виводить на екран.

25. Задано речення. Скласти програму, яка визначає і виводить на екран всі його слова, попередньо перетворивши кожне слово за таким правилом: а) замінити першу зустрінуту літеру «а» на «о»; б) видалити зі слова всі входження останньої літери (крім неї самої), в) залишити в слові тільки перші входження кожної літери (інші видалити); г) в самому довгому слові видалити середню (середні) літери (прийняти, що таке слово є одним 0).

### **Завдання 2 (використання множин).**

Задано текст з цифр і літер латинського алфавіту. Скласти програму, яка визначає, яких літер – голосних {a, e, i, o, u, y} або приголосних більше в цьому тексті.

### **Завдання 3 (використання множин).**

Задано текст з латинських літер. Скласти програму, яка визначає і виводить на екран в алфавітному порядку по одному разу всі голосні літери латинського алфавіту (множина {a, e, i, o, u, y}), які входять в цей текст. Текст та елементи множини задано в одному реєстрі (нижньому або верхньому).

### **4.6. Контрольні запитання.**

4.6.1. Як визначається кортеж, що містить один елемент?

4.6.2. Охарактеризуйте структуру даних "словник".

4.6.3. Які операції можна виконувати над множинами?

4.6.4. Як вивести елементи множини? Як підрахувати кількість елементів у множині?

4.6.5. Як видалити всі ключі і значення із словника?

4.6.6. Назвіть літерали створення порожнього списку та словника?

4.6.7. Як створити порожню множину?

## Лабораторна робота №5

### Розробка програм з використанням процедур і функцій

#### 5.1. Мета роботи

Познайомитися з принципами побудови функцій користувача на мові Python, з використанням локальних і глобальних змінних.

#### 5.2. Теоретичні відомості

Основна перевага використання функцій – це можливість повторного застосування програмного коду, тобто, їх можна викликати багато разів не тільки в тій програмі, де її було визначено, але, можливо, і в інших програмах, іншими користувачами та для інших цілей. Функція – це засіб, який дозволяє групувати набори інструкцій таким чином, що в програмі вони можуть запускатися неодноразово. Функції – це 1) програмні структури, які забезпечують багаторазове використання програмного коду і зменшують його надлишковість; 2) засіб проектування, який дозволяє розбити складну систему на прості і легко керовані частини; 3) засіб структурування програми; 4) можуть обчислювати деякий результат і дозволяють вказувати вхідні параметри, які різняться за своїми значеннями від виклику до виклику; 5) забезпечують можливість розбити складну систему на частини, кожна з яких грає визначену роль.

У табл. 5.1 наводяться основні інструменти, які мають відношення до функцій. В процесі розробки функції дозволяють мінімізувати надлишковість програмного коду.

Таблиця 5.1

*Інструкції та вирази, які мають відношення до функцій*

Інструкція	Приклад
Виклик	<code>myfunc('spam', 'eggs', meat=ham)</code>
<i>def, return</i>	<code>def adder(a, b=1, *c):     return a+b+c[0]</code>
<i>global</i>	<code>def changer():</code>

	global x; x = 'new'
<i>nonlocal</i>	def changer(): nonlocal x; x = 'new'
<i>yield</i>	def squares(x): for i in range(x): yield i ** 2
<i>lambda</i>	funcs = [lambda x: x**2, lambda x: x*3]

Інструкція *def* створює об'єкт функції та зв'язує його з ім'ям. У загальному вигляді інструкція має такий формат:

**def <name>(arg1, arg2,... , argN):**  
**<statements>**

Інструкція *def* складається з рядка заголовка і слідуючого за ним блоку інструкцій, зазвичай з відступами (або простої інструкції слідом за двокрапкою). Блок інструкцій утворює тіло функції, тобто програмний код, який виконується інтерпретатором щоразу, коли здійснюється виклик функції. У рядку заголовка інструкції *def* визначаються ім'я функції, з яким буде пов'язаний об'єкт функції, і список з нуля або більше аргументів (іноді їх називають параметрами) в круглих дужках. Імена аргументів в рядку заголовка будуть пов'язані з об'єктами, переданими в функцію, в точці виклику.

Тіло функції часто містить інструкцію ***return***:

**def <name>(arg1, arg2,... argN):**  
**...**  
**return <value>**

Інструкцію *return* можна розташувати в будь-якому місці в тілі функції – вона завершує роботу функції і передає результат програмі, яка її викликає. Інструкція *return* містить об'єктний вираз, який надає результат функції. Інструкція *return* є необов'язковою – якщо вона відсутня, робота функції завершується, коли потік управління досягає кінця тіла функції. Функція без інструкції *return* повертає об'єкт *None*, проте це значення зазвичай ігнорується.

Визначення функції відбувається під час виконання, тому в іменах функцій немає нічого особливого. Важливим є тільки об'єкт, на який посилається ім'я:

```
def func():  
    pass
```

```
othername = func # Зв'язування об'єкта функції з ім'ям  
othername() # Виклик функції
```

В цьому фрагменті функція зв'язана з іншим ім'ям і викликана з використанням нового імені. Функції – це звичайні об'єкти; їх явно записують в пам'ять під час виконання програми. Крім підтримки можливості виклику, функції дозволяють приєднати будь-які *атрибути*, які можуть зберігати інформацію для наступного використання:

```
value = 'some value'  
def func(): ... # Створюють об'єкт функції  
func() # Виклик об'єкту  
func.attr = value # Приєднання атрибуту до об'єкту  
print(func.__dict__)
```

Про настанову `func.__dict__` мова йтиме пізніше.

## Параметри за замовчуванням

Згадаємо, наприклад, функцію **range()**. Її можна викликати в трьох різних формах – з одним параметром, із двома та з трьома.

Для організації такої поведінки своєї функції можна описати після звичайних (позиційних) ключові параметри зі значеннями за замовчуванням:

```
from pip._vendor.distlib.compat import raw_input  
  
def ask_ok(prompt, retries=4, complaint='Yes or no,  
please!'):  
    """функція ask_ok із параметрами за  
    замовчуванням"""  
    while 1:  
        ok = raw_input(prompt) #уведення значення  
        if ok in ('т', 'так', 'yes'): return 1
```

```

if ok in ('н', 'ні', 'no', 'nope'): return 0
retries = retries - 1
if retries < 0: raise IOError('Помилка')
print(complaint)

```

Виклик `ask_ok ("Прощу ввести:", 2)` установить перший параметр – рядок, як запрошення, а другий параметр змінить кількість неправильних спроб із чотирьох на дві. Викликаючи функцію, ключові параметри можна ставити після позиційних параметрів у довільній послідовності, якщо явно вказано імена ключових параметрів.

```

ask_ok('Старт!', complaint='Так чи ні англійською, будь ласка', retries=2)

```

Розглянемо приклад функції:

```

i = 5
def f(arg=i):
    print(arg)
i = 6
f()  # виведе не 6, а 5; f(10) виведе 10

```

Механізм параметрів за замовчуванням діє так: якщо змінну проініціалізовано до виклику функції, то у функцію передається саме це значення, в іншому випадку у функцію передається значення за замовчуванням.

*Зауваження.* Тіло функції не виконується при її визначенні, а тільки компілюється. І навпаки, значення за замовчуванням обчислюються при визначенні функції та зберігаються в об'єкті-функції.

Тобто, значення за замовчуванням установлюється лише один раз. Це відіграє роль при встановленні значення за замовчуванням спискам, наприклад:

```

def f(a, L=[]):
    L.append(a)
    return L

```

```

print(f(1))
print(f(2))
print(f(3))

```

Результат роботи програми:

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

тобто елементи накопичуються в списку.

Для передачі параметрів за замовчуванням без нагромадження необхідно використовувати таку форму:

```
def f(a, L=None): # None – порожній об'єкт, не вказане
    значення
    if L is None: # якщо параметр L не вказано
        L = []
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

Результат роботи програми:

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

## Документування функцій

Вдалим стилем є документування кожної функції. Для цього в наступному рядку відразу після заголовка необхідно помістити короткий опис функції, укладений у потрібні ''' апострофи або """ лапки. Увесь вміст усередині потрібних лапок виводиться як є, наприклад, інструкцією

```
print (ім'я_функції.__doc__)
```

Такий спосіб дозволяє легко зрозуміти призначення функції, якщо прочитати початковий текст або скористатись спеціальним сервером документації *Python*.

## Передавання у функцію змінної кількості аргументів

Часто використовуваним прийомом у програмуванні є передавання у функцію змінного числа аргументів. Для цього в

*Python* можна скористатися символом `*` перед списком аргументів змінної довжини.

Попереду списку аргументів може бути (не обов'язково) один або кілька обов'язкових аргументів:

```
def fprintf(message, *args):  
    print(message, ' {}'.format(args))
```

```
fprintf('Аргументи', 1)  
fprintf('Аргументи', 1, 'sun')  
fprintf('Аргументи', 1, [1, 2, 3])
```

### Функції *lambda*

**Lambda-вирази.** В *Python* існує можливість створювати об'єкти функцій у формі виразів. Через схожість з аналогічною можливістю в мові *LISP* вона отримала назву *lambda*. Ця назва походить з мови програмування *LISP*, в якій ця назва була запозичена з лямбда-числення. Однак в *Python* це – ключове слово, яке вводить вираз синтаксично. Подібно інструкції *def* цей вираз створює функцію, яку викличуть пізніше (але на відміну від інструкції *def*, вираз повертає функцію, а не зв'язує її з ім'ям). Саме тому *lambda*-вирази іноді називають анонімними (тобто безіменними) функціями. Їх часто використовують, як спосіб отримати вбудовану функцію або відкласти виконання фрагмента програмного коду. У загальному вигляді *lambda*-вираз складається з ключового слова *lambda*, за яким слідують один або більше аргументів (як список аргументів у круглих дужках у заголовку інструкції *def*) і далі, слідом за двокрапкою, розташовано вираз:

***lambda argument1, argument2,... argumentN:***  
***вираз, який використовує аргументи***

Результатом *lambda*-виразу є такі ж об'єкти функцій, які утворюють інструкції *def*, але тут є кілька відмінностей: 1)

*lambda* – це вираз, а не інструкція; 2) тіло *lambda* – це не блок інструкцій, а один вираз. Наприклад, функцію:

```
def func(x, y, z):  
    return x + y + z  
  
func(2, 3, 4) # 9
```

можна описати за допомогою *lambda*-виразу, явно присвоївши результат імені, яке пізніше буде використано для виклику функції:

```
f = lambda x, y, z: x + y + z  
f(2, 3, 4) # 9
```

Тут імені *f* присвоєно об'єкт функції, створений *lambda*-виразом, – інструкція *def* працює так само, але присвоювання виконують автоматично.

Коли можна використовувати *lambda*-вирази? – Їх використовують для створення маленьких функцій і дозволяють вбудовувати визначення функцій в код, який їх використовує. Вони не є предметом першої необхідності (можна замість них використовувати інструкції *def*), але вони дозволяють спростити сценарії, де потрібно впроваджувати невеликі фрагменти програмного коду. Вони корисні в якості скороченого варіанту інструкції *def*, коли необхідно вставити маленькі фрагменти виконуваного коду туди, де використання інструкцій неприпустимо. Слідуючий фрагмент коду створює список з трьох функцій, *lambda*-вирази вставлено в літерал списку. Інструкція *def* не може бути вставлена в літерал, тому що це – інструкція, а не вираз.

```
L = [lambda x: x ** 2, # Вбудовані визначення функцій  
     lambda x: x ** 3,  
     lambda x: x ** 4] # Список з трьох функцій  
for f in L:  
    print(f(2)) # Виведе 4, 8, 16  
print(L[0](3)) # Виведе 9
```



Для реалізації еквівалентної таблиці переходів із застосуванням інструкцій *def* необхідно створити іменовані функції поза контекстом їх використання:

```
# визначення іменованих функцій
def f1(x): return x ** 2

def f2(x): return x ** 3

def f3(x): return x ** 4

L = [f1, f2, f3] # Посилання по імені
for f in L:
    print(f(2)) # Виведе 4, 8, 16
    print(L[0](3)) # Виведе 9
```

### 5.3. Програма роботи

5.3.1. Вивчити теоретичні основи написання алгоритмів з використанням функцій користувача. Опрацювати приклади.

5.3.2. Виконати завдання 1 та 2 згідно варіанту.

### 5.4. Обладнання та програмне забезпечення

5.4.1. Персональний комп'ютер.

5.4.2. Інтерпритатор Python IDLE встановлений на ПК

### 5.5. Порядок виконання роботи і опрацювання результатів

#### Приклад 1. Локальна змінна

```
def function():
    # визначення локальної змінної
    var = 'локальна змінна'
    # виведення значення локальної змінної на екран
    print(var)
# визначення глобальної змінної
var = 'глобальна змінна'
function()
```

```
# виведення на екран значення глобальної змінної  
print(var)
```

Результат

локальна змінна

глобальна змінна

Службове слово *global* вказує на необхідність отримати доступ до глобальної змінної *var*, а не створювати нову локальну під час спроби що-небудь їй присвоїти:

```
def function():  
    global var  
    # виведення значення глобальної змінної на екран  
    print(var)  
    # зміна глобальної змінної  
    var = 'нове значення'  
    # виведення значення глобальної змінної на екран  
    print(var)  
    var = 'глобальна змінна'  
    function(); print(var)
```

Результат

глобальна змінна

нове значення

нове значення

```
def function(c, d):  
    # a, b - глобальні змінні; c, d -- локальні  
    global a, b  
    # зміна значення глобальної змінної  
    a = 5;  
    b = 7  
    # зміна значення локальної змінної  
    c = 10  
    d = 12  
a, b, c, d = 1, 2, 3, 4 # множинне присвоєння  
print(a, b, c, d) # 1 2 3 4  
function(c, d)  
print(a, b, c, d) # 5 7 3 4
```

Результат

1 2 3 4

5 7 3 4

**Приклад 2.** Описати функцію, яка обчислює множину значень  $y = \sqrt[n]{x}$  ( $x>0, n>0$ ), для цього використати рекурентну формулу Ньютона:

$$y_{k+1} = \frac{k-1}{k} y_k + \frac{x}{k \cdot y_k^{k-1}}, \quad k = 0, 1, \dots, \quad y_0 = \frac{x+n-1}{2}, \text{ яка має}$$

місце для  $y_0 > 0$ .

Необхідну точність оцінюють співвідношенням  $|y_{n+1} - y_n| \leq \varepsilon$ .

```
import itertools
import math

def my_sqrt(a,n,Epsilon):
    term1=(a+n-1)/2
    term2=(2/3)*term1 +(a/(3*term1**2))
    k=1
    while (abs(term2- term1) > Epsilon):
        term1= term2; term2=((k-1)/k)* term1 +
a/(k*(term1**(k-1)))
        k+=1
    return term2

print(" x ", ": ", " y ", " : ", " yt", " : ", " error " )
print("-----")
Epsilon=0.0001
a=0.1;b=1
for i in itertools.count(start=a, step=0.2):
    if i>b: break
    n=3
    y= my_sqrt(i,n,Epsilon)
    y1=i**(1/n) # точне значення функції
    error=abs(y-y1)
    print('%0.2f' %i, ":", '%0.4f' %y, " : ", '%0.4f' %y1, " :
", '%0.4f' %error)
```

**Завдання 1:**

Розробіть функції для здійснення наступних

Варіант:

1. Пошук елементів в словнику з значенням ключа.
2. Пошук елемента за значенням в списку;
3. Пошук послідовності елементів в списку;
4. Пошук перших п'яти мінімальних елементів в списку;
5. Пошук перших п'яти максимальних елементів в списку;
6. Пошук середнього арифметичного в списку;
7. Пошук усіх нульових елементів в списку;
8. Повернення списку, що сформований з початкового списку, але не містить повторів (залишається лише перший з однакових елементів).
9. Пошук послідовності елементів в рядку;
10. Розбиття речення на окремі слова (повертається список слів).

## Завдання 2

Описати функцію обчислення значень функції (визначити  $m$  значень заданої функції  $f(x)$  на відрізку  $[a, b]$ ). Результат вивести на екран у вигляді табл. Значення  $a, b, \varepsilon$  ввести з клавіатури.

$x_i$	$f(x_i)$ точно значення	$f_{\text{набл}}(x_i)$ наближене значення	$\varepsilon$ Точність	Кількість ітерацій
$x_0 = a$				
$x_{m-1} = b$				

Стовпчики таблиці: 1 – значення  $x_i$ ; 2 – значення функції  $f(x_i)$ , яке обчислено з використанням функцій інтерпретатора (модуль *math*); 3 – значення функції, яке обчислено за допомогою розкладання в ряд із точністю  $\varepsilon$ ; 4 – точність обчислень; 5 – кількість ітерацій, необхідних для досягнення заданої точності.

Кількість точок на відрізку  $[a, b]$  складає не менше 10 (розбиття може бути рівномірним). Необхідну точність оцінюють співвідношенням  $|y_{n+1} - y_n| \leq \varepsilon$ .

Використати для побудови розкладання елементарних функцій в ряд Тейлора.

### Варіанти

1.  $f(x) = e^{2x}$

2.  $f(x) = e^{(1+x)}$

3.  $f(x) = \frac{\sqrt{x}}{1-x^2}$

4.  $f(x) = \frac{1}{\sqrt{1+x}}$

5.  $f(x) = e^{4x}$

6.  $f(x) = \frac{1}{1-x}$

7.  $f(x) = 1+x$

8.  $f(x) = \ln(1+x)$

9.  $f(x) = \ln(1-x)$

10.  $f(x) = e^x$

Довідкова інформація

Розкладання елементарних функцій в ряд Тейлора	Функція
$1$	$2$
$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^{n-1} \frac{x^{2n-1}}{(2n-1)!} + \dots$	$\sin(x), x \in \mathbb{R}$
$1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots + \frac{x^n}{n!} + \dots$	$e^x, x \in \mathbb{R}$
$x - \frac{x^3}{2!} + \frac{x^4}{4!} - \dots + (-1)^n \frac{x^{2n}}{(2n)!} + \dots$	$\cos(x), x \in \mathbb{R}$
$x + \frac{x^3}{3!} + \frac{x^5}{5!} + \dots + \frac{x^{2n-1}}{(2n-1)!} + \dots$	$\operatorname{sh}(x), x \in \mathbb{R}$
$x + \frac{x^3}{2!} + \frac{x^4}{4!} + \dots + \frac{x^{2n}}{(2n)!} + \dots$	$\operatorname{ch}(x), x \in \mathbb{R}$
$x - \frac{x^2}{2} + \frac{x^3}{3} + \dots + (-1)^{n-1} \frac{x^n}{n} + \dots$	$\ln(1+x), x \in (-1; 1]$
$x - \frac{x^3}{3} + \frac{x^5}{5} - \dots + (-1)^{n-1} \frac{x^{2n-1}}{(2n-1)} + \dots$	$\operatorname{arctg}(x), x \in [-1; 1]$
$-x - \frac{x^2}{2} - \frac{x^3}{3} - \dots - \frac{x^n}{n} - \dots$	$\ln(1-x), x \in (-1; 1)$
$1 - x + x^2 - \dots + (-1)^n x^n + \dots$	$\frac{1}{1+x}, x \in (-1; 1)$

1	2
$1 + x + x^2 + \dots + x^n + \dots$	$\frac{1}{1-x}, x \in (-1; 1)$
$1 + \alpha x + \frac{\alpha(\alpha-1)}{2!} x^2 + \dots + \frac{\alpha(\alpha-1)\dots(\alpha-n+1)}{n!} x^n + \dots$	$(1+x)^\alpha, x \in (-1; 1)$
$1 - \frac{1}{2}x + \frac{3}{8}x^2 - \frac{5}{16}x^3 + \dots + (-1)^{n-1} \frac{(2n-1)!! x^n}{(2n)!!} + \dots$	$\frac{1}{\sqrt{1+x}}, x \in (-1; 1)$
$e^{2x} = 1 + 2x + \frac{(2x)^2}{2!} + \frac{(2x)^3}{3!} + \dots = \sum_{n=0}^{\infty} \frac{(2x)^n}{n!}$	$e^{2x}$
$e^{x^2} = 1 + x^2 + \frac{x^4}{2!} + \frac{x^6}{3!} + \dots = \sum_{n=0}^{\infty} \frac{x^{2n}}{n!}$	$e^{x^2}$
$1 + (1+x) + \frac{(1+x)^2}{2!} + \frac{(1+x)^3}{3!} + \dots = \sum_{n=0}^{\infty} \frac{(1+x)^n}{n!}$	$e^{(1+x)}$
$x \sin x = \frac{1}{1!} x^2 - \frac{1}{3!} x^4 + \frac{1}{5!} x^6 - \frac{1}{7!} x^8 + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+2}$	$x^* \sin(x)$
$1 + 3x + 3x^2 + x^3$	$(1+x)^3$
$x^{1/2} + x^{5/2} + x^{9/2} + \dots$	$\frac{\sqrt{x}}{1-x^2}$

## **5.6. Контрольні запитання.**

5.6.1. Як створити функцію у мові Python?

5.6.2. Що таке модулі та пакети?

5.6.3. Як бувають способи підключення модулів та пакетів?

5.6.4. Що таке анонімні функції та інструкція `lambda`?

5.6.5. Який синтаксис оголошення функції?

5.6.6. Що таке позиційні та непозиційні аргументи функції?

5.6.7. Як створити функцію зі значеннями за замовчуванням?

5.6.8. Який порядок передачі аргументів у функцію, якщо вона містить позиційні та непозиційні аргументи та аргументи із значеннями за замовчуванням?

5.6.9. Для чого використовується інструкція `return`? Чи обов'язково вона присутня у функції?

5.6.10. Що повертає функція, якщо в її тілі відсутня інструкція `return`?

## Лабораторна робота №6

### Модулі в Python. Механізми обробки винятків

#### 6.1. Мета роботи

Познайомитися з принципами створення та імпорту модулів на мові Python, з механізмом обробки виняткових ситуацій.

#### 6.2. Теоретичні відомості

##### Модулі

Організація та структурування проекту здійснюється за допомогою модулів, зібраних у спеціальні каталоги, які називають пакетами.

*Python* дозволяє помістити визначення у файл та використовувати їх у програмах та в інтерактивному режимі. Такий файл називається модулем. Визначення з модуля імпортують в інші модулі та в головний модуль (набір змінних, що є доступним у програмі та в інтерактивному режимі).

Модуль – файл, що містить визначення й інші інструкції мови *Python*. Ім'я файла утворюється додавання до імені модуля суфікса (розширення) **'py'**. У межах модуля, його ім'я доступне через глобальну змінну **\_\_name\_\_**. Наприклад, створіть у поточному каталозі файл з ім'ям **'fibonacci.py'** і таким вмістом:

```
"""Генерація та виведення чисел Фібоначчі"""
```

```
def fib1(n):  
    """Виводить послідовність чисел Фібоначчі <= n"""  
    a, b = 0, 1  
    while b < n:  
        print(b)  
        a, b = b, a + b  
  
def fib2(n):  
    """Повертає список чисел Фібоначчі <= n"""  
    result = []  
    a, b = 0, 1
```



```

while b < n:
    result.append(b)
    a, b = b, a + b

return result

```

## Імпорт модулів

Перелік різноманітних варіантів інструкцій підключення модулів:

**import список\_модулів** – імпортує зазначені в списку модулі;

**from ім'я\_модуля import \*** – імпортує всі імена з модуля, за винятком таких, що починаються із символу "\_";

**from ім'я\_модуля import список\_об'єктів** – імпортує вказані об'єкти з модуля;

**import ім'я\_модуля as name** – імпортує модуль під новим іменем **name**;

**from ім'я\_модуля import identifier as name** – імпортує з модуля об'єкт **identifier** під новим іменем **name**.

Отже в іншому файлі в поточній директорії можна виконати:

```

import fibo

fibo.fib1(3)
result = fibo.fib2(5)
print(result)

```

або

```

from fibo import *

fib1(3)
result = fib2(5)
print(result)

```

## Виняткові ситуації

Винятки – це сповіщення інтерпретатора, порушені в разі виникнення помилки в програмному коді або при настанні якої-небудь події. Якщо в коді не передбачено оброблення винятків, то програма переривається і виводиться повідомлення про помилку.

Існує три типи помилок в програмі:

*Синтаксичні* – це помилки в імені оператора або функції, невідповідність закриваючих та відкриваючих лапок і т.д. Тобто помилки в синтаксисі мови. Як правило, інтерпретатор попередить про наявність помилки, а програма не виконуватиметься зовсім. Приклад синтаксичної помилки:

```
print("Невідповідність відкритих та закритих лапок!")
```

Результатом запуску даного коду буде:

```
SyntaxError: EOL while scanning string literal
```

*Семантичні* – це помилки в логіці роботи програми, які можна виявити тільки за результатами роботи скрипта. Як правило, інтерпретатор не попереджає про наявність помилки. А програма буде виконуватися, оскільки не містить синтаксичних помилок. Такі помилки досить важко виявити і виправити.

*Помилки часу виконання* – це помилки, які виникають під час роботи скрипта. Причиною є події, які не передбачені програмістом. Класичним прикладом служить ділення на нуль: Для управління помилками, що виникають у ході виконання програми, в Python використовуються спеціальні об'єкти, які називаються **винятками**.

Якщо при виникненні помилки Python не знає, що робити далі, створюється **об'єкт винятку**. Якщо у програму включений код обробки виняткової ситуації, то виконання програми продовжиться, а якщо немає - програма зупиняється і виводить **трасування** (інформацію про хід виконання програми) зі звітом про помилку.

Щось схоже можна спостерігати, коли спробувати отримати доступ до елемента, що не входить у список або кортеж, отримати значення елемента у словнику по ключу, якого не існує. Коли виконується код, який при деяких обставинах може не

спрацювати, використовують **обробники винятків**, щоб перехопити будь-які потенційні помилки.

Хорошим тоном є використання обробників винятків всюди, де може бути згенеровано виняток, щоб користувач знав, що відбувається. Ви можете бути не в змозі виправити помилку, але принаймні можете дізнатися, за яких обставин це сталося, і акуратно завершити програму.

Можна також створювати власні об'єкти винятків.

### Блок try-except

Якщо не використовувати обробники винятків, Python виведе повідомлення про помилку і деяку інформацію про те, де сталася помилка, а потім завершить програму, як показано в наступному фрагменті коду:

```
short_list = [1, 2, 3]
position = 5
print(short_list[position])
```

Виникає помилка виходу за межі діапазону списку:

```
Traceback (most recent call last):
  File "C:/work/python_labs/new.py", line 3, in <module>
    print(short_list[position])
IndexError: list index out of range
```

Розмістимо свій код у блоці **try** і використаємо блок **except**, щоб обробити помилку:

```
short_list = [1, 2, 3]
position = 5
try:
    short_list[position]
except:
    print('Need a position between 0 and',
          len(short_list)-1, 'but got', position)
```

В результаті виконання програми отримаємо повідомлення:

Need a position between 0 and 2 but got 5

Код у даному прикладі запускається всередині блока **try**. Якщо виникла помилка, генерується виняток і виконується код, розміщений всередині блока **except**. Якщо помилки не виникають в процесі виконання програми, блок **except** буде не задіяний.

В даному випадку використання блока **except** говорить лише: «Виникла помилка!». Якщо ви припускаєте, що в програмі може статися помилка і знаєте назву винятку, який при цьому згенерується, напишіть блок **try-except** для обробки відомого винятка.

Для прикладу, розглянемо виконання операції

```
print(5/0)
```

яка викликає помилку **ділення на нуль**, при якій Python ініціює виняток **ZeroDivisionError**:

```
Traceback (most recent call last):  
  File "C:/work/python_labs/new.py", line 1, in  
    <module>  
      print(5/0)  
ZeroDivisionError: division by zero
```

Помилка **ZeroDivisionError** — є **об'єктом винятку**. Повідомимо Python, як слід чинити при виникненні даного винятку, щоб програма буде підготовлена до його появи. Ось як виглядає блок **try-except** для обробки винятків **ZeroDivisionError**:

```
try:  
    print(5/0)  
except ZeroDivisionError:  
    print("You can't divide by zero!")
```

У цьому прикладі код блоку **try** породжує помилку **ZeroDivisionError**, тому Python шукає блок **except** з описом того, як слід діяти в такій ситуації. При виконанні коду блоку **except** користувач бачить зрозуміле повідомлення про помилку:

```
You can't divide by zero!
```

Іноколи необхідно отримати не лише об'єкт винятку, а зберегти його у змінну, використовуючи таку форму запису:

**try:**

**блок коду**

**except тип\_винятку as назва\_змінної:**

**блок коду, коли виникла помилка**

У наступному прикладі виконується перевірка на `IndexError`, оскільки саме цей виняток викликається, коли ви вкажете недійсну (відсутню) позицію у послідовності. Виняток `IndexError` зберігається у змінній `err`, а будь-який інший виняток `Exception` - у змінній `other`:

```
students = ["Олександр", "Вадим", "Микола", "Роман",  
            "Денис", "Віталій"]  
while True:  
    value = input('Введіть номер студента [q - вийти]?  
' )  
    if value == 'q':  
        break  
    try:  
        position = int(value)  
        print(students[position])  
    except IndexError as err:  
        print('Елемент відсутній в списку:', position)  
    except Exception as other:  
        print('Something else broke:', other)
```

У прикладі на екран виводиться все, що зберігається в змінній `other`:

```
Введіть номер студента [q - вийти]? 1  
Вадим  
Введіть номер студента [q - вийти]? 9  
Елемент відсутній в списку: 9  
Введіть номер студента [q - вийти]? ц  
Something else broke: invalid literal for int() with  
base 10: 'ц'  
Введіть номер студента [q - вийти]? q
```

Process finished **with exit** code 0

Введення позиції 9, як і очікувалося, генерує виняток `IndexError`. Введення слова 'ц' «не сподобалось» функції `int()`, яка була оброблена у другому обробнику винятку.

Є ще дві інструкції, пов'язані з блоком **try-except**, це **finally** і **else**.

Команда **finally** виконує блок інструкцій в будь-якому випадку, чи було згенерований виняток, чи ні (це корисно для випадків, коли необхідно обов'язково щось зробити, наприклад закрити файл або мережеве з'єднання).

Інструкція **else** виконується в тому випадку, якщо виняток не буде згенерований.

Наприклад, при виконанні фрагмента коду

```
def divide(x, y):  
    try:  
        result = x / y  
    except ZeroDivisionError:  
        print("Division by zero!")  
    else:  
        print("Result is", result)  
    finally:  
        print("End of the calculation.")  
divide(2, 1)  
divide(2, 0)
```

отримаємо наступні результати:

```
Result is 2.0  
End of the calculation.  
Division by zero!  
End of the calculation.
```

Можна також визначити власні типи винятків, щоб обробляти особливі ситуації, які можуть виникнути у програмах.

## Створення власного винятку

Для визначення власних винятків необхідно використовувати класи, які детально розглядатимуться далі. Будь-який виняток є класом, зокрема, нащадком класу `Exception`.

Створимо виняток, який називається `IsNotTitleException`.

Викличемо виняток, коли зустрінемо слово у списку, перша літера якого записана у нижньому регістрі:

```
class IsNotTitleException(Exception):
    pass

cities = ['Zurich', 'work', 'Frankfurt', 'Venice']
for city in cities:
    if city.title() != city:
        raise IsNotTitleException(city)
```

Поведінку для винятку `IsNotTitleException` не визначено (використано `pass`). Тому батьківський клас `Exception` самостійно виконав виведення повідомлення на екран при генерації винятку:

```
Traceback (most recent call last):
  File "C:/work/python_labs/new.py", line 7, in
<module>
    raise IsNotTitleException(city)
__main__.IsNotTitleException: work
```

Але можна вказати, що має виводити власний виняток, коли його примусово викликати за допомогою ключового слова `raise`:

```
class IsNotTitleException(Exception):
    pass

cities = ['Zurich', 'work', 'Frankfurt', 'Venice']
try:
    for city in cities:
        if city.title() != city:
            raise IsNotTitleException(city)
except IsNotTitleException as exc:
    print(exc)
```

Результат виконання коду буде таким:

```
'work'
```

На практиці слід уникати порожніх інструкцій *except*, оскільки можна перехопити виняток, яке є лише сигналом системи, а не помилкою.

Якщо в обробнику присутній блок *else*, то інструкції всередині цього блоку будуть виконані тільки при відсутності помилок. При необхідності виконати будь-які завершальні дії незалежно від того, було згенеровано виняток чи ні, слід скористатися блоком *finally*.

```
try:
    x = 10 / 2 # Немає помилки
except ZeroDivisionError:
    print("Ділення на 0")
else:
    print("Блок else")
finally:
    print("Блок finally")
```

Виконуються

Блок *else*

Блок *finally*

Якщо ж виняток стався:

```
try:
    x = 10 / 0 # Помилка: ділення на 0
except ZeroDivisionError:
    print("Ділення на 0")
else:
    print("Блок else")
finally:
    print("Блок finally")
```

Виконуються

Ділення на 0

Блок *finally*

## 6.3. Програма роботи

6.3.1. Ознайомитись із теоретичним матеріалом та опрацюйте всі приклади з теоретичних відомостей.



6.3.2. Виконати завдання 1 та 2.

## 6.4. Обладнання та програмне забезпечення

6.4.1. Персональний комп'ютер.

6.4.2. Інтерпритатор Python IDLE встановлений на ПК

## 6.5. Порядок виконання роботи і опрацювання результатів

### Завдання 1.

Файл **models.py** містить програмний код, поданий нижче, що імітує друкування 3D-моделей різних об'єктів. Перенесіть функції `print_models()` і `show_completed_models()` у окремий файл з ім'ям `printing_functions.py`. Виконайте імпорт цих функцій у файл **models.py**, змінивши файл так, щоб у ньому імпортовані функції можна було використовувати.

```
def print_models(unprinted_designs, completed_models):  
    """Імітує друк 3D-моделей, доки список не стане  
    порожнім. Кожна модель після друку переміщується у  
    completed_models."""  
  
    while unprinted_designs:  
        current_design = unprinted_designs.pop()  
        # Імітація друку моделі на 3D-принтері.  
        print("Printing model: " + current_design)  
        completed_models.append(current_design)  
  
def show_completed_models(completed_models):  
    """Виводить інформацію про усі надруковані  
    моделі."""  
    print("\nThe following models have been printed:")  
    for completed_model in completed_models:  
        print(completed_model)  
  
unprinted_designs = ['iphone case', 'robot pendant',  
    'dodecahedron']  
completed_models = []  
print_models(unprinted_designs, completed_models)  
show_completed_models(completed_models)
```

## **Завдання 2.**

Розробіть функції для здійснення наступних операцій зі списками:

1. Сортювання;
2. Пошук елемента за значенням;
3. Пошук послідовності елементів;
4. Пошук перших п'яти мінімальних елементів;
5. Пошук перших п'яти максимальних елементів;
6. Пошук середнього арифметичного;
7. Повернення списку, що сформований з початкового списку, але не містить повторів (залишається лише перший з однакових елементів).

Помістіть функції в окремий модуль. Реалізуйте програму, яка використовує всі функції зі створеного модуля. Зробити описи Doc strings для кожної реалізованої функції.

## **6.6. Контрольні запитання.**

- 6.6.1. Як можна запобігти аварійному завершенню програми?
- 6.6.2. Які блоки коду записують у блок try? А які у except?
- 6.6.3. Поняття помилки.
- 6.6.4. Що таке виняткові ситуації і яким чином здійснюється їх оброблення у Python?
- 6.6.5. Атрибути винятків, ініціювання винятків.
- 6.6.6. Для чого використовується гілка finally в інструкції try?
- 6.6.7. Чи можна гілку finally поєднувати з гілками except?
- 6.6.8. Чи можна перехоплювати різні класи помилок в одному блоці try/except?
- 6.6.9. Який клас помилок згенерується, якщо звернутись до неіснуючого елемента списку за індексом?
- 6.6.10. Чи можна створювати власні класи виняткових ситуацій?

## Лабораторна робота №7

### Розробка програм з використанням класів в Python

#### 7.1. Мета роботи

Ознайомитись з принципами реалізації об'єктно-орієнтованого програмування у мові Python та навчитись використовувати його для розроблення програмного забезпечення.

#### 7.2. Теоретичні відомості

**Клас** – це складний користувацький тип даних, що складається з полів та методів. *Поля класу* – це змінні, оголошені всередині класу для збереження даних. *Методи класу* – це функції, оголошені для обробки полів класу, взаємодії з основною програмою та іншими даними.

**Об'єкти** – це окремі екземпляри класу, по суті змінні цього типу. Оголосивши клас із полями та методами, ви можете створювати скільки завгодно об'єктів, кожен з яких міститиме оголошений в класі набір полів та методів.

Класи оголошуються з ключовим словом **class**, ім'я класу за стандартами Python записується наступним чином: всі слова разом, кожне слово з великої літери.

Для створення екземпляру клас викликається як функція з круглими дужками, повертаючи новий об'єкт, який ви можете зберегти у змінній.

Для методів класу можна вказувати атрибути.

**Атрибут** – це змінна, метод – це функція. Відмінності методу від функції в тому, що у нього є перший параметр – **self**.

Класи, як і модулі, приховують внутрішню будову, залишаючи на поверхні лише зовнішній "інтерфейс" для використання.

Це поєднання даних та функцій всередині однієї сутності, разом із прихованням внутрішньої будови, називається *інкапсуляцією* і є головним принципом ООП.

При оголошенні класу в дужках можуть бути записані (одне або декілька) імена вже існуючих класів – це називається наслідуванням. В такому випадку новий клас (який називається дочірнім) успадковує всі поля та методи класів перелічених в дужках (які називаються батьківськими).

Визначення класу:

```
class ім'я_класу:  
    інструкція 1  
    ....  
    інструкція N
```

Створення об'єкта класу:

```
об'єкт_класу = ім'я_класу()
```

Класи збирають в собі набори даних (змінних) разом з наборами функцій, що на них діють. Мета полягає в тому, щоб досягти більш модульного коду за допомогою групування змінних і функцій, в невеликі вузли, що легко модифікувати.

За згодою у Python для посилання на об'єкт використовується ім'я `self`. Змінна `self` зв'язується з об'єктом, до якого було застосовано даний метод, і через цю змінну ми отримуємо доступ до атрибутів об'єкта. Коли цей же метод застосовується до іншого об'єкта, то `self` зв'яжеться вже з саме цим іншим об'єктом, і через цю змінну будуть викликатись тільки його поля.

```
class Adder:  
    n = 1  
  
    def add(self, v):  
        return v + self.n  
  
a = Adder()  
b = Adder()  
a.n = 10
```

```
print(a.add(3))
print(b.add(4))
```

Тут від класу `Adder` створюється два об'єкта – `a` та `b`. Для об'єкта `a` заводиться власне поле `n`. Об'єкт `b`, не має такого поля, отже успадковує `n` від класу `Adder`. У методі `add()` вираз `self.n` – це звернення до поля `n`, переданого об'єкта, і не важливо, на якому рівні наслідування його буде знайдено.

## Конструктор класу¶

В ООП конструктором класу називають метод, який автоматично викликається при створенні об'єктів. Його також можна назвати конструктором об'єктів класу. Ім'я такого метода зазвичай регламентується синтаксисом конкретної мови програмування. В Python роль конструктора виконує метод `__init__()`. Об'єкт створюється в момент виклику класу по імені, і в цей момент викликається метод `__init__()`, якщо його визначено в класі.

В Python наявність пар знаків підкреслення попереду і позаду в імені метода говорить про те, що він належить до групи методів переважання операторів, які часто називають «магічні методи». Якщо подібні методи визначені у класі, то об'єкти можуть брати участь в таких операціях як додавання, віднімання, викликатись як функції та інше. При цьому методи переважання операторів не треба викликати по імені. Викликом для них є сам факт участі об'єкта в певній операції. У випадку конструктора класу — це операція створення об'єкта.

Необхідність конструкторів пов'язана з тим, що часто об'єкти повинні мати власні властивості одразу. Припустимо маємо клас `Person`, об'єкти котрого обов'язково повинні мати ім'я. Якщо клас буде описано наступним способом:

```
class Person():
    def set_name(self, name):
        self.name = name
```

то створення об'єкта можливе без полів. Для встановлення імені метод `set_name()` необхідно викликати окремо:

```
p1 = Person()
p1.name = 'Jane Doe'
print(p1.name)
'Jane Doe'
```

Наявність конструктора не дозволить створити об'єкт без полів:

```
class Person():
    def __init__(self, name):
        self.name = name

p1 = Person('Jane Doe')
print(p1.name)
```

Тут при виклику класа в круглих дужках передаються значення, котрі будуть присвоєні параметрам метода **init()**. Перший параметр – **self** – посилання на сам щойно створений об'єкт.

Тепер, якщо ми спробуємо створити об'єкт, не передавши нічого в конструктор, то буде "викинуто" виняткову ситуацію, і об'єкт не буде створено.

Однак буває, що необхідно допустити створення об'єкта, якщо деякі дані в конструктор не передаються. У такому випадку параметрам конструктора класу задаємо значення за замовчуванням:

```
class Person():
    def __init__(self, name, phone=''):
        self.name = name
        self.phone = phone
```

Якщо викликати конструктор без параметра **phone**, то буде використано значення за замовчуванням. Однак поля **name** і **phone** будуть у всіх об'єктів:

```
p1 = Person('Jane Doe', '+380971234567')
p2 = Person('John Doe')
print(p1.name, p1.phone) #('Jane Doe', '+380971234567')
print(p2.name, p2.phone) #('John Doe', '')
```

Крім того, конструктору зовсім не обов'язково приймати будь-які параметри, крім `self`. Значення полям можуть назначатись як завгодно. Також не обов'язково, щоб в конструкторі виконувалось встановлення атрибутів об'єкта. Там може бути, наприклад, код, який створює об'єкти інших класів.

### **Деструктор класу**

Окрім конструктора об'єктів в ООП є зворотній йому метод – деструктор. Він викликається, коли об'єкт знищується.

В Python об'єкт знищується, коли зникають усі пов'язані з ним змінні або їм присвоюється інше значення, в результаті чого зв'язок з старим об'єктом втрачається. Видалити змінну також можна за допомогою `del`.

В Python функцію деструктора виконує метод `__del__()`. Але в Python деструктор використовується рідко, інтерпретатор і без нього добре впорається зі "сміттям".

Згідно моделі даних, Python пропонує три види методів: статичні, класові і екземпляра класу.

- Методи екземпляра класу отримують доступ до об'єкта класу через параметр `self` і до класу через `self.__class__`.
- Методи класу не можуть отримати доступ до певного екземпляра класу, але мають доступ до самого класу через `cls`.
- Статичні методи працюють як звичайні функції, але належать до простору імен класу. Вони не мають доступу ні до самого класа, ні до його екземплярів.

### **Методи екземпляра класу**

Методи екземпляра класу приймають об'єкт класу як перший аргумент, який прийнято називати `self` і який вказує на сам екземпляр.

Використовуючи параметр `self` можна міняти стан об'єкта і звертатись до інших його методів і параметрів. Також через атрибут `self.__class__` можна отримати доступ до атрибутів класу і можливість міняти стан самого класу. Тобто методи

екземплярів класу дозволяють міняти як стан певного об'єкта, так і класу.

### Класові методи

Методи класу приймають клас в якості параметра, його прийнято позначати як `cls`. Він вказує на клас, а не на об'єкт цього класу. При декларації методів цього виду використовують декоратор `classmethod`.

Методи класу прив'язані до самого класу, а не його екземпляра. Вони можуть міняти стан класа, що відобразиться на усіх об'єктах цього класу, але не можуть міняти конкретний об'єкт.

### Статичні методи

Статичні методи декларуються за допомогою декоратора `staticmethod`. Їм не потрібно певного першого аргумента (ні `self`, ні `cls`).

Їх можна сприймати як методи, які "не знають, до якого класа відносяться".

Таким чином, статичні методи прикріплені до класа лише для зручності і не можуть міняти стан ні класа, ні його екземпляра.

Клас, де використовуються усі три види методів:

```
class MyClass:
    def instance_method(self):
        return 'instance method called', self
    @classmethod
    def class_method(cls):
        return 'class method called', cls
    @staticmethod
    def static_method():
        return 'static method called'

obj = MyClass()
```

Виклик методу екземпляра:

```
print(obj.instance_method())
```



В результаті отримаємо:

```
('instance method called', <__main__. MyClass object at 0x03370550>)
```

Приклад вище підтверджує те, що метод `instance_method` має доступ до об'єкта класу `MyClass` через аргумент `self`. Зауважте, що метод `obj.instance_method()` можна викликати і так:

```
MyClass.instance_method(obj)
```

Тепер скористаємось методом класу:

```
print(obj.class_method())
```

В результаті отримаємо:

```
('class method called', <class '__main__.MyClass'>)
```

Як видно, метод класу `class_method()` має доступ до самого класу `MyClass`, але не до його конкретного екземпляра. Клас — теж об'єкт, який ми можемо передати функції в якості аргумента.

Викликаємо статичний метод:

```
print(obj.static_method())
```

В результаті отримаємо:

```
static method called
```

Статичні методи можна викликати через об'єкт класу. Насправді статичному методу ніякі спеціальні аргументи (`self` чи `cls`) не передаються. Тобто статичні методи не можуть отримати доступ до параметрів класу чи об'єкта. Вони працюють тільки з тими даними, які їм передаються як аргументи.

Якщо викликати ті ж самі методи, але на самому класі.

```
print(MyClass.class_method())  
( 'class method called', <class '__main__.MyClass'>)
```

```
print(MyClass.instance_method())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: instance_method() missing 1 required positional
argument: 'self'
```

Виклик метода екземпляра класа видає `TypeError`. Сталося це через те, що метод очікував екземпляр класа, а було передано клас. Зі статичним методом нічого неочікуваного:

```
print(MyClass.static_method())
'static method called'
```

### **7.3. Програма роботи**

7.3.1. Ознайомитися з теоретичними відомостями.

7.3.2. Створити та запустити на виконання приклад 1. Модифікувати програму наступним чином: додати конструктор, який ініціалізує атрибут `name`; в основній програмі передбачити створення заданої користувачем кількості студентів та виведення середнього бала для кожного з них (ім'я та оцінки також вводяться користувачем).

7.3.3. Виконати завдання 1 згідно варіанту.

### **7.4. Обладнання та програмне забезпечення**

7.4.1. Персональний комп'ютер.

7.4.2. Інтерпретатор Python встановлений на ПК

### **7.5. Порядок виконання роботи і опрацювання результатів**

**Приклад 1. Програма з використанням класу. Обчислення середнього бала студента з трьох предметів.**

```
class Student:
```

```
    def set_marks(self, e1, e2, e3):
        self.e1 = e1
```

```

        self.e2 = e2
        self.e3 = e3

    def set_name(self, name):
        self.name = name

    def get_average_mark(self):
        print(self.name, ' - ', ((self.e1 + self.e2 +
self.e3) / 3))

s1 = Student()
s2 = Student()
s1.set_name('Dmytro')
s1.set_marks(5, 4, 5)
s1.get_average_mark()

s2.set_name('Mykola')
s2.set_marks(4, 4, 4)
s2.get_average_mark()

```

### Завдання:

1. Розробити клас "домашня бібліотека". Додати конструктор, який приймає ціле число – порядковий номер книги та словник, що містить інформацію про книгу наступному форматі: 1) автор; 2) назва; 3) видавництво; 4) жанр; 5) рік видання.

Реалізувати можливість роботи з довільним числом книг, пошуку по книгах за декількома параметрами (за автором, за роком видання, за жанром тощо), додавання книг у бібліотеку, видалення книг з неї, доступу до книги за номером. Написати програму, що буде демонструвати всі розроблені елементи класу.

2. Розробити клас для представлення відомостей про успішність студента. Об'єкт класу має містити поля для збереження імені студента та балів, отриманих ним за виконання лабораторних робіт та індивідуального творчого завдання.

Забезпечити наступні методи класу: конструктор, який приймає рядок ім'я студента та словник, що містить

налаштування курсу у наступному форматі: 1) максимально можлива кількість балів за здачу індивідуального творчого завдання; 2) максимально можлива кількість балів за здачу однієї лабораторної роботи; 3) кількість лабораторних робіт в курсі; метод, за допомогою якого вносяться оцінки за лабораторну роботу, який приймає параметри оцінку та номер лабораторної роботи; метод, за допомогою якого вносяться дані про оцінку за індивідуальне творче завдання; метод, який повертає дійсне число (суму балів студента за проходження курсу).

3. Розробити клас "інтернет-замовлення". Додати конструктор, який приймає ціле число – порядковий номер замовлення та словник, що містить інформацію про замовлення наступному форматі: 1) прізвище клієнта; 2) назва товару; 3) кількість; 4) вартість; 5) дата замовлення.

Реалізувати можливість роботи із замовленнями: пошук по за декількома параметрами (за прізвищем замовника, за датою замовлення, за товаром тощо), додавання нових замовлень, видалення інформації про замовлення, доступу до інформації про замовлення за номером. Написати програму, що буде демонструвати всі розроблені елементи класу.

4. Розробити клас "Працівник". Додати конструктор, який приймає ціле рядок – прізвище та ім'я працівника та словник, що містить інформацію про працівника наступному форматі: 1) назва віділу; 2) посада; 3) рік народження; 4) стаж роботи.

Реалізувати можливість роботи із замовленнями: пошук по за декількома параметрами (за прізвищем, за віком, за відділом тощо), додавання нових працівників, видалення інформації про працівника, доступу до інформації про замовлення за прізвищем та ім'я. Написати програму, що буде демонструвати всі розроблені елементи класу.

5. Розробити клас для представлення відомостей про успішність студента. Об'єкт класу має містити поля для збереження імені студента та балів, отриманих ним за виконання лабораторних робіт та лекційних занять.

Забезпечити наступні методи класу: конструктор, який приймає рядок ім'я студента та словник, що містить налаштування курсу у наступному форматі: 1) кількість лекційних занять в курсі; 2) максимально можлива кількість балів за здачу однієї лабораторної роботи; 3) кількість лабораторних робіт в курсі; метод, за допомогою якого вносяться оцінки за лабораторну роботу, який приймає параметри оцінку та номер лабораторної роботи; метод, за допомогою якого вносяться дані про кількість відвіданих лекцій; метод, який повертає дійсне число (суму балів студента за проходження курсу).

## **7.6. Контрольні запитання.**

7.6.1. Як створити клас у мові Python?

7.6.2. Що таке інкапсуляція?

7.6.3. Що таке об'єкт?

7.6.4. Що таке атрибут?

7.6.5. Для чого призначений конструктор класу?

7.6.6. Який обов'язковий аргумент приймає метод класу?

7.6.7. Що таке self?

7.6.8. Як оголосити метод класу статичним?

7.6.9. Як оголосити метод класу класовим методом?

7.6.10. Яка різниця між статичним і класовим методом?

## **Лабораторна робота №8**

### **Розробка програм з ієрархією класів. Організація класів з використанням успадкування в Python**

#### **8.1. Мета роботи**

Ознайомитися з особливостями реалізації наслідування атрибутів класу в ООП на мові Python

#### **8.2. Теоретичні відомості**

##### **Інкапсуляція**

Класи в ООП бувають великими і складними. В них може бути багато полів і методів, які не повинні використовуватись за його межами. Вони просто для цього не призначені. Вони свого роду внутрішні шестерні, які забезпечують нормальну роботу великого механізму.

Хорошою практикою вважається приховування усіх полів об'єктів, щоб запобігти прямого присвоєння значень з іншого місця програми. Їх значення можна змінювати і отримувати лише через виклики методів, спеціально для цього визначених. Наприклад, якщо необхідно перевіряти значення, яке присвоюється певному полю на коректність, то робити це кожного разу в основному коді програми буде неправильним. Перевірочний код має бути розміщено у методі, котрий отримує дані для присвоєння полю. А саме поле має бути закритим для доступу ззовні класу. У цьому випадку йому неможливо буде присвоїти недопустиме значення.

**Інкапсуляція** (encapsulation) — це механізм, який об'єднує дані і код, який маніпулює цими даними, а також захищає і те, і інше від зовнішнього втручання або неправильного використання.

В багатьох мовах програмування, які підтримують парадигму ООП, існують спеціальні модифікатори доступу атрибутів. Вони явно вказують, чи можна мати доступ до певного атрибуту класу "ззовні", чи цей атрибут доступний тільки всередині класу.

В Python механізму модифікаторів доступу не існує. Для імітації приховування атрибутів в Python використовується домовленість згідно якої якщо ідентифікатор атрибута починається з знака підкреслення, то цей атрибут призначено виключно для внутрішнього використання.

```
class Person:
    def __init__(self, age):
        self._age = age
```

Але домовленість — це не синтаксичне правило мови програмування, і при великому бажанні її можна порушити:

```
p = Person(age=35)
print(p._age)
```

Можна ще більше приховати атрибут класа. Для цього його ідентифікатор має починатись не з одного, а з двох знаків підкреслення:

```
class Person:
    def __init__(self, age):
        self.__age = age
```

```
p = Person(age=35)
print(p.__age)
```

AttributeError: 'Person' object has no attribute '\_\_age'

Насправді доступ до атрибута \_\_age можна, але вже трохи важче. Необхідно вказати атрибут класа таким чином:

1. символ підкреслення
2. ім'я класа
3. ім'я атрибута як у класі, тобто з двома підкресленнями на початку:

```
print(p._Person__age)
```

Це знову ж таки домовленість. В результаті "атрибут як він є" стає замаскованим. Ззовні класа такого атрибута просто не існує. Для програміста ж наявність двох символів підкреслення перед іменем атрибута повинно сигналізувати, що чіпати його поза класом не слід взагалі.

### Сетери, гетери і делетери

Отримати значення атрибута, в тому числі прихованого, можна за допомогою метода:

```
class Person:
    def __init__(self, age):
        self.__age = age

    def get_age(self):
        return self.__age
```

```
p = Person(age=35)
print(p.get_age())
```

Так само за допомогою методів можна реалізувати присвоєння значень прихованим атрибутам класа:

```
class Person:
    def __init__(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age
```

```
p = Person(age=35)
p.set_age(-35)
print(p.get_age())
```

В об'єктно-орієнтованому програмуванні прийнято імена методів для вилучення даних починати зі слова get(взяти), а



імена методів, в яких полям присвоюються певні значення — зі слова `set`(встановити). А самі методи часто називають відповідно **сетерами** і **гетерами**. Існують також **делетери** — методи для видалення (`delete`) полів класа.

У вищенаведеному прикладі `get_age` — це гетер, а `set_age` — сетер. Зауважте що для встановлення значення для прихованого атрибута в конструкторі скористались сетером.

## Властивості

Значення, які характеризують стан об'єкта (атрибути), доступ до яких відбувається за допомогою сетерів і гетерів, називають **властивостями** (`property`).

Для створення властивості використовують функцію:

```
property(fget, fset, fdel, doc)
```

де:

- `fget` — Функція, яка реалізує повернення значення властивості
- `fset` — Функція, яка реалізує встановлення значення властивості
- `fdel` — Функція, яка реалізує видалення значення властивості
- `doc` — рядок документації для властивості. Якщо не задано, то береться від `fget`

Усі параметри необов'язкові.

```
class Person:
    def __init__(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age
age = property(get_age, set_age, None, "Person's age")

p = Person(age=35)
```

```
p.set_age(-35)
print(p.age)
```

В Python також є ще один більш елегантний спосіб визначення властивостей за допомогою декораторів.

Для створення властивості-гетера використовуємо:

```
@property
```

Для створення властивості-сетера використовуємо:

```
@<властивість-гетер>.setter
```

Person з використанням декораторів:

```
class Person:
    def __init__(self, age):
        self.__age = 0
        self.age = age

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age > 0:
            self.__age = age
```

Зверніть увагу на наступне:

- сетер визначається після гетера
- і сетер, і гетер називаються однаково — age. І оскільки гетер називається age, то над сетером встановлюється анотація @age.setter
- і до гетера і до сетера ми звертаємось через вираз p.age

## Успадкування

Успадкування (наслідування, inheritance) — механізм утворення нових класів на основі використання властивостей і функціоналу вже існуючих класів.

Ключовими поняттями наслідування є підклас (subclass) і суперклас (super class). Суперклас ще називають базовим (base

class) або батьківським (parent class), а підклас — похідним (derived class) або дочірнім (child class). Підклас успадковує властивості, методи та інші публічні атрибути з базового класа. Він також може перевизначати (override) методи базового класа. Якщо підклас не визначає свій конструктор, він успадковує конструктор базового класа за замовчуванням.¶

В Python синтаксис для наслідування класів виглядає наступним чином:

```
class <subclass>(<superclass>):
    <subclass attributes>

class Base:
    def __init__(self):
        self.base_prop = 'base property'

    def method(self):
        print("Це метод з класа Base.")
        print("У об'єкта класа Base є атрибут base_prop,
Його значення:", self.base_prop)

class Child(Base):
    def child_method(self):
        print("Це метод з класа Child.")
        print("Об'єкт класа Child має атрибут base_prop.
Його створено в успадкованому конструкторі класа Base:",
self.base_prop)

c = Child()
c.method()
# Це метод з класа Base.
# У об'єкта класа Base є атрибут base_prop, його значення:
base property
c.child_method()
# Це метод з класа Child.
# Об'єкт класа Child має атрибут base_prop. Його створено в
успадкованому конструкторі класа Base: base property
```

Клас Child успадковує від класа Base два методи: конструктор і метод method().

Також клас Child має свій власний метод: child\_method().

При створенні об'єкта класа Child буде викликано успадкований конструктор класа Base. У конструкторі для об'єкта створюється атрибут base\_prop. У об'єктів класа Child теж буде створено цей атрибут.

Клас може бути успадкованим від класа, який у свою чергу було успадковано від іншого класа. Коли розглядають увесь ланцюжок успадкованих і базових класів, говорять про **ієрархію успадкування**.

В Python є вбудований клас який має назву object. Від цього класа явно чи неявно успадковуються усі інші класи, як вбудовані, так і ті, що створюєте ви. Якщо при створенні класа ви не вказуєте базовий клас, то неявним чином ваш клас буде успадковано від object. Отже наступні оголошення класа рівносильні:

```
class A:  
    pass
```

```
class A():  
    pass
```

```
class A(object):  
    pass
```

Розглянемо наступну ієрархію класів:

```
class A: pass
```

```
class B(A): pass
```

```
class C(B): pass
```

```
c_obj = C()
```

Дізнатись, чи є певний клас підкласом іншого класа по всій ієрархії успадкування, можна за допомогою вбудованої функції issubclass():

```
print(issubclass(C, B)) #True
print(issubclass(C, A)) #True
print(issubclass(C, object)) #True
print(issubclass(B, C)) #False
```

Також можна дізнатись чи є певний об'єкт екземпляром класа враховуючи всю ієрархію успадкування:

```
print(isinstance(c_obj, C)) #True
```

### Успадкування і приватні атрибути

Як нам вже відомо, атрибути, які починаються з двох символів підкреслення (але не закінчуються ними) є приватними атрибутами класа. Поза видимістю класа до таких атрибутів застосовується механізм *name mangling* (спотворення імені), тобто такі атрибути "поза класом" будуть мати інші імена (клас+атрибут), у тому числі і в успадкованих класах. Це дозволяє "приховати" внутрішню реалізацію класа навіть для класів, які від нього успадковуються.

### Лінеаризація

Як ми вже з'ясували, дочірній клас може не мати певного атрибута, але він може успадкувати його від базового класа. Для пошуку атрибутів в ієрархії класів використовується лінеаризація класів.

**Лінеаризація** — це черговість, при якій проводиться пошук зазначеного атрибута в ієрархії класів.

Використовуючи лінеаризацію відбувається пошук атрибутів в ієрархії класів. При простому успадкуванні алгоритм пошуку атрибутів виглядає наступним чином:

- якщо атрибут, до якого відбувається доступ, не знайдено в поточному класі, то виконується його пошук в базовому класі;
- якщо атрибут не знайдено і в базовому класі, то виконується його пошук в базовому класі базового класа;
- пошук відбувається рекурсивно аж до класа `object`;
- якщо атрибут не знайдено і в класі `object`, то отримуємо виняткову ситуацію

В Python лінеаризація ще називається MRO — Method Resolution Order, порядок вирішення методів. Назва може трошки вводити в оману, тому що таким чином відбувається пошук не тільки методів, а й будь-яких атрибутів.

Лінеаризація для певного класа знаходиться в його спеціальному атрибуті `__mro__`:

```
print(C.__mro__)
```

Результат:

```
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

Але частіше користуються атрибутом-методом класа, який повертає не кортеж, а одразу список:

```
print(C.mro())
```

### Множинне успадкування

При вказівці методів з однаковою назвою у різних класів і різним вмістом успадковується в класі-нащадку той, клас якого прописаний першим при множинному спадкуванні (в круглих дужках після назви класу-нащадка). Це просто перевірити на прикладі нижче, змінюючи послідовність розташування батьківських класів для наслідування у дочірнього класу.

```
class A(object):
    def method(self):
        print('A method')
```

```
class B(A): pass
class C(A):
    def method(self):
        print('C method')
```

```
class D(B, C): pass
```

```
obj = D()
obj.method() # 'C method'
```

Якщо в заданому класі атрибут або метод був перевизначений, то доступ до відповідного атрибуту суперкласу можна отримати двома способами.

Перший спосіб – явне звернення до атрибутів суперкласу за його ім'ям. Недолік такого підходу – ускладнюється підтримка коду.

Існує спеціальний клас *super*, екземпляри якого є спеціальними проксі-об'єктами, прив'язаними до заданої ієрархії класів і надають доступ до атрибутів наступного класу в лінеаризації того класу, в якому був створений об'єкт *super*. Отже, за допомогою *super* можна отримати доступ до атрибутів і методів суперкласу, не називаючи його імені, причому це буде давати коректні результати навіть при використанні множинного успадкування.

### Перевизначення і пошук методів

Уявімо ситуацію, що в базовому класі, від якого ми будемо успадковувати наш новий клас, вже реалізовано певний метод, котрий підходить нам по своїй функціональності, але у ньому не вистачає певних речей, або ж нам треба дещо змінити його функціонал. Звісно, що ми можемо повністю переписати цей метод у нашому новому класі, але з великою ймовірністю ми стикнемось з повторним використання коду. І якщо, припустимо, ми вносимо зміни в метод базового класа, то таоіж з зміни нам доведеться вносити і в аналогічний метод нашого нового класа, що є небажаним (підвищується ймовірність припуститись помилки, зайва робота в решті решт).

Якщо в дочірньому класі певний атрибут було перевизначено, а потрібен доступ до відповідного атрибута базового класа, в Python це можна зробити двома способами. Один з них полягає у тому, що ми явно вказуємо базовий клас, відповідний атрибут і, при необхідності, передаємо екземпляр дочірнього класа (параметр *self*).

Розглянемо приклад.

```
class Person:  
    def __init__(self, name):
```

```

        self.name = name.title()

    def say_hello(self):
        print('Hi, I am', self.name)

p = Person('john')
p.say_hello()

```

Зауважимо, що в конструкторі класа відбувається певна маніпуляція зі вхідними даними.

Тепер нам треба створити клас, який описував би не просто людину, а співробітника певної організації. Співробітник має усі атрибути, які має і людина (зокрема ім'я), власне будь-який співробітник і є людиною, тому логічно успадкуватись від класа Person. Крім того співробітник має ще й заробітню плату.

```

class Employee(Person):
    def __init__(self, name, salary):
        Person.__init__(self, name)
        self.salary = salary

    def say_hello(self):
        Person.say_hello(self)
        print('My salary is', self.salary)

e = Employee('JANE', 120)
e.say_hello()

```

В конструкторі дочірнього класа ми викликаємо конструктор базового класа при цьому передаючи йому екземпляр дочірнього класа і необхідні дані для ініціалізації атрибутів. В конструкторі базового класа відбувається певна маніпуляція над вхідними даними і відбувається ініціалізація атрибута name. І вже потім в конструкторі дочірнього класа відбувається ініціалізація атрибута salary.

Аналогічно з метода say\_hello() дочірнього класа викликається відповідний метод базового класа.



Недоліки такого підходу:

- ускладнюється підтримка коду якщо нам треба щось поміняти в ієрархії класів
- логіка коду чітко прив'язана до ієрархії успадкування класів і схильна до помилок, особливо при використанні множинного успадкування.

### 8.3. Програма роботи

8.3.1. Ознайомитися з теоретичним матеріалом.

8.3.2. Створити та запустити на виконання приклади 1-2.

8.3.3. Виконати завдання 1 згідно варіанту.

### 8.4. Обладнання та програмне забезпечення

8.4.1. Персональний комп'ютер.

8.4.2. Інтерпретатор Python встановлений на ПК

### 8.5. Порядок виконання роботи і опрацювання результатів

#### Приклад 1. Використання `super` при множинному успадкуванні

```
class Animal(object):
    def __init__(self):
        self.can_fly = False
        self.can_run = False
    def print_abilities(self):
        print(self.__class__.__name__)
        print('Can fly:', self.can_fly)
        print('Can run:', self.can_run); print()

class Bird(Animal):
    def __init__(self):
        super().__init__()
        self.can_fly = True

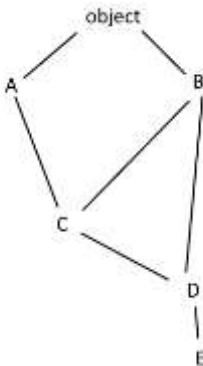
class Horse(Animal):
    def __init__(self):
        super().__init__()
        self.can_run=True
```

```
class Pegasus(Horse, Bird): pass
```

```
def main():  
    bird = Bird()  
    bird.print_abilities()  
    horse = Horse()  
    horse.print_abilities()  
    pegasus = Pegasus()  
    pegasus.print_abilities()  
if __name__ == '__main__': main()
```

**Приклад 2. Використання *super*, декоратору *gen\_init* і побудова інтерпретатором лінеаризації.** Декоратор *gen\_init* додає автоматично згенерований конструктор, це – функція, яка приймає функцію або клас і повертає інший об'єкт, що буде прив'язаний до початкового імені. Зазвичай його використовують для зміни поведінки функції (шляхом створення нової функції, яка викликає початкову) або модифікації класу (наведено в даному прикладі).

Маємо таку ієрархію класів



```
def gen_init(cls):  
    """ Декоратор gen_init:  
    :param cls: клас, який підлягає модифікації  
    :return: клас із доданим конструктором """
```

```

def init(self):
    print('Entered', cls.__name__, "constructor")
    super(cls, self).__init__()
    print('Quit', cls.__name__, "constructor")

cls.__init__ = init
return cls

```

```

@gen_init
class A(object): pass

```

```

@gen_init
class B(object): pass

```

```

@gen_init
class C(A, B): pass

```

```

@gen_init
class D(C, B): pass

```

```

@gen_init
class E(D): pass

```

```

print(E.__mro__)
obj = E()

```

**Завдання 1.** Створити ієрархію класів у відповідності з вказаною предметною областю. Передбачити можливість роботи з довільним числом записів, а також реалізувати: конструктори; функції виведення інформації на екран; функції пошуку потрібної інформації за конкретною ознакою. При розробці програми слід здійснити захищення даних для ізоляції елементів-даних класу від підпрограм, в яких цей клас використовується.

### **Варіанти:**

1. Створити базовий клас "Транспортний засіб". На його основі реалізувати класи "Літак", "Автомобіль" та "Корабель". Класи повинні мати можливість задавати та отримувати параметри засобів пересування (вартість, швидкість, рік випуску тощо) задати за допомогою полів. Для літака повинна бути визначена висота, для літака та корабля – кількість пасажирів, для корабля – порт приписки.

2. Створити базовий клас "Комп'ютер". На його основі реалізувати класи "Ноутбук", "Персональний комп'ютер" та "Сервер". Класи повинні мати можливість задавати та отримувати параметри (вартість, модель, рік випуску тощо) задати за допомогою полів. Для ноутбука повинна бути визначена діагональ екрана, для ПК та сервера – наявність оптичного приводу, для сервера – тип (шафований, підлоговий).

3. Створити базовий клас «Фігура», в якому визначено координати однієї з вершин геометричної фігури. Створити похідні класи: «Прямокутник» (додатково визначаються довжини двох сторін), «Коло» (додатково визначається радіус), «Прямокутний трикутник» (додатково визначаються довжини катетів).

4. Створити базовий клас "Товар". На його основі реалізувати класи "Телевізор", "Телефон" та "Ноутбук". Класи повинні мати можливість задавати та отримувати параметри (вартість, модель, рік випуску тощо) задати за допомогою полів. Для ноутбука повинна бути визначена діагональ екрана та тип процесора, для Телевізора – діагональ екрана та тип матриці та телефона – операційна система та кількість камер.

5. Створити базовий клас "Особа". На його основі реалізувати класи "Працівник", "Викладач" та "Студент". Класи повинні мати можливість задавати та отримувати параметри (прізвище, ідентифікаційний номер, рік народження тощо) задати за допомогою полів. Для працівника повинні бути визначені відділ та посада, для викладача – кафедра та наукове звання, для студента – кафедра та курс.

## **8.6. Контрольні запитання.**

8.6.1. Що означає множинне спадкування?

8.6.2. В чому різниця між статичними методами і методами класу.

8.6.3. Що означає лінеаризація?

8.6.4. Як визначити, чи є певний клас підкласом іншого класу?

8.6.5. Як в Python оголосити атрибут прихованим?

8.6.6. Як отримати доступ до прихованого атрибуту?

8.6.7. Що таке властивість класу?

8.6.8. Як в Python оголосити один клас нащадком іншого?

8.6.9. Що таке лінеаризація?

8.6.10. Для чого призначений клас `super`?

## Лабораторна робота №9

### Робота з файлами у мові Python

#### 9.1. Мета роботи

Навчитися здійснювати операції читання та запису для файлів у мові Python.

#### 9.2. Теоретичні відомості

##### Файл

Файл – це іменована область пам'яті в комп'ютері, якою управляє операційна система (довільна послідовність елементів одного типу, довжина цих послідовностей заздалегідь не визначається, а конкретизується в процесі виконання програми; дані, що містяться у файлі, переносять на зовнішні носії).

Текстові файли призначені для зберігання текстової інформації. Компоненти текстових файлів можуть мати змінну довжину.

Вбудована функція *open* створює об'єкт файлу, який забезпечує зв'язок з файлом, розміщеним в комп'ютері. Після виклику цієї функції можна виконувати операції зчитування і запису в зовнішній файл, використовуючи методи отриманого об'єкта. Об'єкти файлів не є ні числами, ні послідовностями або відображеннями – для роботи з файлами вони надають тільки методи. Більшість методів файлів пов'язані з виконанням операцій введення–виведення у зовнішні файли, асоційовані з об'єктом. У табл. 9.1 наведено операції над файлами, які найчастіше використовують.

Таблиця 9.1

Операція	Інтерпретація
<code>output = open(r'C:\spam', _w')</code>	Відкриває файл для запису ( 'w' означає write – запис)
<code>input = open(_data', _r')</code>	Відкриває файл для зчитування ( 'r' означає read – зчитування)
<code>input = open(_data')</code>	Відкриває файл для зчитування (режим <code>_r'</code> використовують за замовчуванням)

<code>aString = input.read()</code>	Зчитування файлу цілком в один рядок
<code>aString = input.read(N)</code>	Зчитування наступних N символів (або байтів) в рядок
<code>aString = input.readline()</code>	Зчитування наступного текстового рядка (включаючи символ кінця рядка) в рядок
<code>aList = input.readlines()</code>	Зчитування файлу цілком в список рядків (включаючи символ кінця рядка)
<code>output.write(aString)</code>	Запис рядка символів (або байтів) у файл
<code>output.writelines(aList)</code>	Запис всіх рядків зі списку в файл
<code>output.close()</code>	Закриття файлу вручну (виконується після закінчення роботи з файлом)
<code>output.flush()</code>	Виштовхує вихідні буфери на диск, файл залишається відкритим
<code>anyFile.seek(N)</code>	Змінює поточну позицію в файлі для наступної операції, зсуюючи її на N байтів від початку файлу.
<code>for line in open(__data`):</code> операції <i>над line</i>	Ітерації по файлу, зчитування по рядках
<code>open(__f.txt`, encoding='latin-1')</code>	Файли з текстом Юнікоду (рядки типу str)
<code>open(__f.bin`, __rb`)</code>	Файли з двійковими даними (рядки типу bytes)

*Відкриття файлів.* Щоб відкрити файл, програма повинна викликати функцію *open*, передавши їй ім'я зовнішнього файлу і режим роботи: як режим використовують рядок *'r'*, якщо файл відкривають для зчитування (за замовчуванням), *'w'* – якщо файл відкривають для запису або *'a'* – для запису в кінець. У рядку режиму можна також зазначати інші параметри: додавання символу в рядок режиму означає 1) *«b»* – роботу з двійковими даними (відключають інтерпретацію символів кінця рядка і кодування символів Юнікоду); 2) *«+»* – файл відкривають для

зчитування і для запису (є можливість зчитувати і записувати дані в один і той же об'єкт файлу, часто спільно з операцією позиціонування в файлі).

Обидва аргументи функції *open* повинні бути рядками. Крім того, функція може приймати третій необов'язковий аргумент, керуючий процесом буферизації виведених даних, – значення нуль означає, що вихідна інформація не буде буферизована (вона буде записуватися у зовнішній файл відразу ж, в момент виклику методу запису). Ім'я зовнішнього файлу може включати шлях до файлу, якщо шлях до файлу не вказано, передбачають, що файл розміщено в поточному робочому каталозі (тобто в каталозі, де був запущений сценарій).

*Використання файлів.* Як тільки отримано об'єкт файлу, можна викликати його методи для виконання операцій зчитування або запису.

Наведемо кілька основних зауважень щодо використання файлів:

1. *Для зчитування рядків краще використовувати ітератори файлів.*

Найкращий, мабуть, спосіб зчитування рядків з файлу на сьогоднішній день полягає в тому, щоб взагалі не використовувати операцію зчитування з файлу: файли мають ітератор, який автоматично зчитує інформацію з файлу рядок за рядком в контексті циклу *for*, в генераторах списків і в інших ітераційних контекстах.

2. *Вміст файлів знаходиться в рядках, а не в об'єктах.* Зверніть увагу: в табл. 9.1 показано, що дані, отримані з файлу, завжди потрапляють в сценарій у вигляді рядка. Якщо ця форма подання не підходить, необхідно виконати перетворення даних в інші типи об'єктів мови Python, а при виконанні операції запису даних у файл необхідно передавати методам сформовані рядки.

Тому при роботі з файлами треба згадати інструменти перетворення даних з рядка у число і навпаки (наприклад, *int*, *float*, *str*, а також вирази форматування рядків і метод *format*). Крім того, до складу Python входять додаткові стандартні бібліотечні інструменти, призначені для роботи з універсальним



об'єктом «сховище даних» (наприклад, модуль *pickle*) і обробки упакованих двійкових даних у файлах (наприклад, модуль *struct*).

3. Виклик методу *close* є необов'язковим, він розриває зв'язок із зовнішнім файлом. Інтерпретатор Python негайно звільняє пам'ять, зайняту об'єктом, як тільки в програмі буде загублена останнє посилання на цей об'єкт. Як тільки об'єкт файлу звільняють, інтерпретатор закриває асоційований з ним файл (що відбувається також в момент завершення програми). Завдяки цьому не потрібно закривати файл вручну. З іншого боку, виклик методу *close* не зашкодить, і його рекомендують використовувати у великих системах (в момент закриття файлів звільняються ресурси операційної системи і виштовхуються вихідні буфери).

4. *Файли забезпечують буферизацію введення-виведення і дозволяють виробляти позиціонування у файлі.* За замовчуванням виведення у файли завжди виконують за допомогою проміжних буферів, тобто в момент запису тексту у файл він не потрапляє відразу ж на диск – буфери виштовхуються на диск тільки в момент закриття файлу або при виклику методу *flush*. Можна відключити механізм буферизації за допомогою додаткових параметрів функції *open*, але це може призвести до зниження продуктивності операцій введення-виведення. Файли в Python підтримують і можливість позиціонування – метод *seek* дозволяє сценаріями управляти позицією зчитування і запису.

## Копіювання файлу

```
import shutil
shutil.copyfile("C:\\mydoc.doc", "C:\\My
Documents\\mydoc_2.doc")
```

## Перейменування файлу

```
import os
os.rename("C:\\mydoc.doc\\testfile.txt",
"/home/user/test.txt")
```

### Видалення файлу

```
import os
os.remove(" C:\\mydoc.doc \\testfile.txt")
```

### Читання потрібного рядка з текстового файлу

Щоб прочитати рядок під певним номером - можна скористатися як стандартним читанням файлу в список, так і використовувати модуль `linecache`:

```
line = linecache.getline("C:\\boot.ini", 2)
# or
line = open("C:\\boot.ini").readlines()
```

### Перебір файлів у каталозі

```
for filename in os.listdir("../plugins"):
    print(filename)
```

### Перебір файлів у каталозі за маскою

```
import glob
for filename in glob.glob("../plugins\\*.zip"):
    print(filename)
```

### Порівняння файлів

Порівнювати файли можна як за змістом, так і за їх властивостями, що значно швидше, за допомогою `filecmp`

```
import filecmp
similar = filecmp.cmp('C:\\file1.txt', 'C:\\file2.txt')
print(similar)
```

## 9.3. Програма роботи

### 9.3.1. Ознайомитися з принципами роботи з файлами в

Python.

9.3.2. Створити та запустити на виконання приклади 1-3.

9.3.3. Виконати завдання 1 згідно варіанту.

## 9.4. Обладнання та програмне забезпечення

9.4.1. Персональний комп'ютер.

9.4.2. Інтерпретатор Python встановлений на ПК

## 9.5. Порядок виконання роботи і опрацювання результатів

**Приклад 1. Запис даних у текстовий файл.** Створіть директорію з назвою data в середині директорії, де знаходиться даний скрипт

```
import os.path # Модуль, який містить функції для роботи з шляхом у файловій системі
```

```
text = '''Hello!  
I am a text file. And I had been written with a Python  
script  
before you opened me, so look up the docs and try to delete  
me using Python, too.'''
```

```
def write_text_to_file(filename, text):  
    """Функція для запису у файл filename рядка text"""  
    f = open(filename, "w") # відкриття файла для запису  
    f.write(text) # Запис рядка text у файл  
    f.close() # Закриття файлу
```

```
if __name__ == '__main__':  
    write_text_to_file(os.path.join('data',  
'example01.txt'), text)
```

## Приклад 2. Відкриття файла для дозапису

```
import os.path  
import datetime
```

```

text = '''This text was added in example 2!
Updated
'''

if __name__ == '__main__':
    log_file = os.path.join('data', 'example01.txt')
    with open(log_file, 'a') as log:
        print('\n', text, str(datetime.datetime.now()),
file=log)

```

### Приклад 3. Відкриття файлу для зчитування

```

import os.path

def read_file(fname):
    """Функція для зчитування файла fname
    та виведення його вмісту на екран"""
    file = open(fname, 'r') # відкриття файлу для
зчитування
    print('File ' + fname + ':') # виведення назви файлу
    # зчитування вмісту файлу по рядках
    for line in file:
        print(line, end='') # виведення рядка s

    file.close() # закриття файлу

if __name__ == '__main__':
    # функція os.path.join з'єднує частини шляху у файловій
системі
    # необхідним роздільником
    read_file(os.path.join('data', 'example01.txt'))

```

### Завдання:

**Завдання 1.** Створіть файли, у яких будуть міститися рядки з іменами студентів та їх середніми балами.

Реалізуйте читання файлів, запис та дозапис у файли, пошук файлів у каталозі та пошук даних у файлі. Також реалізуйте сортування даних у файлі за середнім балом.

**Завдання 2.** Сформувати файл (або файли) у текстовому редакторі «Блокнот». Маємо текстовий файл (або файли).

Варіанти:

1. Переписати його рядки в інший файл. Порядок розташування рядків у другому файлі повинен: а) збігатися з порядком рядків в заданому файлі; б) бути зворотним по відношенню до порядку рядків в заданому файлі.

2. Переписати його рядки в зворотному порядку (справа наліво) в інший файл. Порядок рядків у другому файлі повинен: а) збігатися з порядком рядків в заданому файлі; б) бути зворотним по відношенню до порядку рядків в заданому файлі.

3. Отримати текст, в якому в кінці кожного рядка з заданого файлу доданий знак оклику.

4. Переписати в інший файл ті його рядки, в яких є більше 30-ти символів.

5. Переписати в інший файл всі його рядки з заміною в них символу «0» на символ «1» і навпаки.

6. Всі парні рядки цього файлу записати в другий файл, а непарні в третій файл. Порядок проходження рядків зберігається.

7. Два файли з однаковою кількістю рядків. Переписати зі збереженням порядку проходження рядки першого файлу в другий, а рядки другого файлу - в перший. Використовувати допоміжний файл.

8. Два файли з однаковою кількістю рядків. З'ясувати, чи співпадають їх рядки. Якщо ні, то отримати номер першого рядка, в якому ці файли відрізняються один від одного.

9. Вивести на екран: а) всі його рядки, які розпочинаються з літери «Т»; б) всі його рядки, які містять більше 30 символів; в) всі його рядки, в яких є більше трьох прогалин; г) всі його рядки, які містять в якості фрагмента заданий текст.

10. Визначити: а) кількість рядків, що починаються з літер «А» чи «а»; б) в яких є рівно п'ять літер «і».

11. Визначити і вивести: а) довжину найдовшого рядка; б) номер найдовшого рядка (якщо таких рядків декілька, то номер

першого із них); в) сам найдовший рядок (якщо таких рядків декілька, то перший із них).

12. З'ясувати, чи є в ньому рядок, який розпочинається з літери «Т». Якщо так, то визначити номер першого з таких рядків.

13. Визначити і вивести: а) перший символ першого рядка; б) п'ятий символ першого рядка; в) перші 10 символів першого рядка; г) перший символ другого рядка; д) k-й символ n-го рядка.

14. У кожному рядку заданого файлу перші два символи є літерами. Вивести: а) слово, утворене першими літерами кожного рядка; б) слово, утворене другими літерами кожного рядка; в) послідовність символів, утворену 5-ми символами кожного рядка.

15. Підрахувати кількість рядків у ньому.

16. Підрахувати кількість символів в ньому.

17. Підрахувати кількість символів в кожному рядку.

18. Видалити з файлу третій рядок. Результат записати в інший файл.

19. Видалити з файлу його останній рядок. Результат записати в інший файл.

20. Видалити з файлу перший рядок, в кінці якого стоїть знак запитання. Результат записати в інший файл.

21. Додати у файл рядок з дванадцяти рисок (-----), розмістивши їх: а) після п'ятого рядка; б) після останнього з рядків, в яких немає прогалини. Якщо таких рядків немає, то новий рядок необхідно додати після всіх рядків наявного файлу. В обох випадках результат записати в інший файл.

22. Елементами файлу є числа. Видалити з нього п'яте число. Результат записати в інший файл.

23. Елементами файлу є цілі числа. Всі парні числа записати в інший файл.

24. 31. Елементами файлу є окремі символи (цифри та літери). Всі цифри цього файлу записати в другій файл, а решта символи - в третій файл. Порядок слідування зберігається.

25. Елементами файлу є окремі слова. Записати в інший файл слова, які розпочинаються на літеру «о» або «а».

26. Елементами файлу є цілі числа. Видалити з нього число, записане після першого нуля (нули у файлі обов'язково присутні). Результат записати в інший файл.

27. Два текстові файли однакового розміру, елементами яких є числа. Отримати третій файл, кожен елемент якого дорівнює: а) сумі відповідних елементів заданих файлів; б) більшому із відповідних елементів заданих файлів.

28. Два текстові файли однакового розміру, елементами яких є числа. Отримати третій файл, кожен елемент якого дорівнює: а) різниці відповідних елементів заданих файлів; б) меншому з відповідних елементів заданих файлів.

29. Два текстові файли однакового розміру, елементами яких є окремі літери. Отримати третій файл, кожен елемент якого є поєднанням відповідних літер першого і другого файлів.

30. Два текстові файли однакового розміру, елементами яких є окремі літери. Записати в третій файл всі співпадаючі елементи наявних файлів.

## **9.6. Контрольні запитання.**

9.6.1. Як здійснюється запис даних у файл у мові Python?

9.6.2. Як здійснюється читання даних з файлу у мові Python?

9.6.3. Як здійснюється копіювання файлу?

9.6.4. Який синтаксис функції `open()` у мові Python?

9.6.5. Які бувають режими роботи з файлами?

9.6.6. Що таке менеджер контексту `with`?

9.6.7. у якому вигляді зчитуються дані з файлу в програму?

## Лабораторна робота №10

### Робота з базою даних із Python-програми

#### 10.1. Мета роботи

Ознайомитися з організацією та розробити сховища даних з використанням парадигми ООП.

#### 10.2. Теоретичні відомості

**Реляційна база даних** – це набір таблиць із даними.

**Таблиця** – це прямокутна матриця, що складається з рядків та стовпців. Таблиця визначає відношення (*relation*).

**Рядок** – запис, що складається з полів-стовпців. У кожному полі може міститися деяке значення або спеціальне значення NULL (порожньо). У таблиці може бути довільна кількість рядків. Для реляційної моделі порядок розташування рядків не важливий і його не визначено.

Кожний **стовпець** у таблиці має власне ім'я й тип.

При програмуванні на *Python* доступ до бази даних не складніший за доступ до інших джерел даних (файлів, мережових об'єктів). Для демонстрації обрано СКБД *SQLite*, що працює як під Unix, так і під Windows. Крім установлення власне *SQLite* (сайт <http://sqlite.org>) та модуля з'єднання з *Python* (<http://pysqlite.org>), останнє необхідно для старих версій *Python*, у версії 2.6 ця СКБД йде як убудований модуль, якогось додаткового налаштування виконувати не потрібно, тому що *SQLite* зберігає дані бази в окремому файлі. Тому відразу братися за створення таблиць, занесення до них даних та виконання запитів не можна. Обрана СКБД (у силу своєї "легкості") має одну істотну особливість – за одним невеликим винятком, СКБД *SQLite* не звертає уваги на типи даних (вона зберігає всі дані у вигляді рядків), тому модуль розширення **sqlite3** для *Python* виконує додаткову роботу з перетворення типів.

Схематично робота з базою даних може виглядати приблизно так:

- під'єднання до бази даних (виклик *connect()* з отриманням об'єкта-з'єднання);
- створення одного або декількох курсорів (виклик методу об'єкта-з'єднання *cursor()* з отриманням об'єкта-курсора);
- виконання команди або запиту (виклик методу *execute()* або його варіантів);



- отримання результатів запиту (виклик методу *fetchone()* або його варіантів);
- завершення транзакції або її відкочування (виклик методу об'єкта-з'єднання *commit()* або *rollback()*);
- коли всі необхідні транзакції виконано, підключення закривається викликом методу *close()* об'єкта-з'єднання.

Спочатку нам потрібно імпортувати модуль `sqlite3` і створити зв'язок з базою даних. Ви можете передати назву файлу або просто використовувати спеціальну рядок `":memory:"` для створення бази даних в пам'яті. У нашому випадку, ми створюємо його на диску в файлі під назвою `mydatabase.db`.

```
import sqlite3
```

```
conn = sqlite3.connect("mydatabase.db") # або :memory: щоб
зберегти в RAM
cursor = conn.cursor()
```

```
# Створення таблиці
```

```
cursor.execute("""CREATE TABLE albums
                (title text, artist text, release_date
                text,
                 publisher text, media_type text)
                """)
```

Далі ми створюємо об'єкт `cursor`, який дозволяє нам взаємодіяти з базою даних і додавати записи, крім усього іншого. Тут ми використовуємо синтаксис SQL для створення таблиці під назвою альбому з п'ятьма наступними полями: `title`, `artist`, `release_date`, `publisher` і `media_type`. SQLite підтримує тільки п'ять типів даних: `null`, `integer`, `real`, `text` і `blob`. Давайте напишемо цей код і вставимо деякі дані в нашої новій таблиці. Запам'ятайте, якщо ви запускаєте команду `CREATE TABLE`, при цьому база даних вже існує, ви отримаєте повідомлення про помилку.

```
# Додаємо дані в таблицю
```

```
cursor.execute("""INSERT INTO albums
                VALUES ('Glow', 'Andy Hunter',
                '7/24/2012',
```

```

        'Xplore Records', 'MP3')""")
    )

# Зберігаємо зміни
conn.commit()

# Вставляємо список з п'яти наборів даних використовуючи
# безпечний метод "?"
albums = [('Exodus', 'Andy Hunter', '7/9/2002', 'Sparrow
Records', 'CD'),
          ('Until We Have Faces', 'Red', '2/1/2011',
'Essential Records', 'CD'),
          ('The End is Where We Begin', 'Thousand Foot
Krutch', '4/17/2012', 'TFKmusic', 'CD'),
          ('The Good Life', 'Trip Lee', '4/10/2012', 'Reach
Records', 'CD')]

cursor.executemany("INSERT INTO albums VALUES (?, ?, ?, ?, ?)",
albums)
conn.commit()

```

Тут ми використовували команду INSERT INTO SQL щоб вставити запис в нашу базу даних. Зверніть увагу на те, що кожен об'єкт знаходиться в одинарних лапках. Це може ускладнити роботу, якщо вам потрібно вставити рядки, які містять одинарні лапки. У будь-якому випадку, щоб зберегти запис у базі даних, нам потрібно створити її. Наступна частина коду показує, як додати кілька записів за раз за допомогою методу курсора executemany. Зверніть увагу на те, що ми використовуємо знаки питання (?), Замість рядків заміщення (%) щоб вставити значення. Зверніть увагу, що використання рядка заміщення небезпечно, так як може стати причиною появи атаки ін'єкцій SQL. Використання знака питання набагато краще, а використання SQLAlchemy тим більше, так як він робите все необхідне, щоб уберегти вас від правки вбудованих одинарних лапок на те, що SQLite в змозі приймати.

### **Редагування і видалення записів**

Можливість оновлювати записи у вашій базі даних це ключ до того, щоб ваші дані велися акуратно, і був повний порядок. Якщо ви не можете редагувати дані, тоді ваша база стане марною досить скоро. Іноді вам, в тому числі, потрібно

буде видаляти і рядки. Створимо новий скрипт в тій же директорії, де був створений попередній

```
import sqlite3

conn = sqlite3.connect("mydatabase.db")
cursor = conn.cursor()

sql = """
UPDATE albums
SET artist = 'John Doe'
WHERE artist = 'Andy Hunter'
"""
```

```
cursor.execute(sql)
conn.commit()
```

Тут ми використовували команду SQL UPDATE, щоб оновити таблицю альбомів. Тут ви можете використовувати команду SET, щоб змінити поле, так що в нашому випадку ми змінимо ім'я виконавця на John Doe в кожного запису, де поле виконавця зазначено для Andy Hunter. Зверніть увагу на те, що якщо ви не підтвердите зміни, то вони не будуть внесені в базу даних. Команда DELETE настільки ж проста.

```
import sqlite3

conn = sqlite3.connect("mydatabase.db")
cursor = conn.cursor()

sql = "DELETE FROM albums WHERE artist = 'John Doe'"

cursor.execute(sql)
conn.commit()
```

Видалення ще простіше, ніж оновлення. У SQL це займає всього дві строчки. В даному випадку, все, що нам потрібно зробити, це вказати SQLite, з якої таблиці видалити (albums), і яку саме запис за допомогою пункту WHERE. Таким чином, було виконано пошук запису, в якій присутня ім'я "John Doe" в поле виконавців, після чого ці дані були видалені.

## Основні запити SQLite

Запити в SQLite дуже схожі на ті, які ви використовуєте в інших базах даних, таких як MySQL або Postgres. Ми просто використовуємо звичайний синтаксис SQL для виконання запитів, після чого об'єкт cursor виконує SQL. Ось кілька прикладів:

```
import sqlite3

conn = sqlite3.connect("mydatabase.db")
# conn.row_factory = sqlite3.Row
cursor = conn.cursor()

print("Список всіх записів артиста Red:")
sql = "SELECT * FROM albums WHERE artist=?"
cursor.execute(sql, [("Red")])
print(cursor.fetchall()) # or use fetchone()

print("Список всіх записів в таблиці:")
for row in cursor.execute("SELECT rowid, * FROM albums ORDER BY artist"):
    print(row)

print("Results from a LIKE query:")
sql = "SELECT * FROM albums WHERE title LIKE 'The%'"
cursor.execute(sql)

print(cursor.fetchall())
```

Перший запит, який ми виконали, називається SELECT \*, що означає, що ми хочемо вибрати всі записи, які підходять під передане ім'я виконавця, в нашому випадку це "Red". Далі ми виконуємо SQL і використовуємо функцію fetchall () для отримання результатів. Ви також можете використовувати функцію fetchone () для отримання першого результату. Зверніть увагу на те, що тут є прокоментований розділ, пов'язаний з таємничим row\_factory. Якщо ви не прокоментуєте цей рядок, результат повернеться, так як об'єкти Row, подібні словників Python і дають вам доступ до полів рядків точнісінько, як і словник. У будь-якому випадку, ви не можете виконати

призначення пункту, використовуючи об'єкт Row. Другий запит дуже схожий на перший, але повертає кожну запис в базі даних і впорядковує результати по імені артиста в порядку зростання. Це також показує, як ми можемо зробити цикл результати видачі. Останній запит показує, як команда LIKE використовується при пошуку часткових фраз. У нашому випадку, ми шукали по всій таблиці заголовки, які починаються з артикля The. Знак відсотка (%) є підстановлювальний оператором.

### 10.3. Програма роботи

10.3.1. Ознайомитися з теоретичним матеріалом.

10.3.2. Створити та запустити на виконання приклад 1. Додати в таблицю ще 3 записи.

10.3.3. Виконати завдання 1 згідно варіанту.

### 10.4. Обладнання та програмне забезпечення

10.4.1. Персональний комп'ютер.

10.4.2. Інтерпретатор Python встановлений на ПК

### 10.5. Порядок виконання роботи і опрацювання результатів

#### Приклад 1. Збереження даних в базі даних

```
import sqlite3

# Клас для що описує працівників підприємства
class Worker():
    def __init__(self, **kwargs):
        self.name = kwargs.get('name')
        self.position = kwargs.get('position')
        self.work_from_date = kwargs.get('work_from_date')
        self.birth_date = kwargs.get('birth_date')

    def get_data(self):
        return self.name, self.position,
self.work_from_date, self.birth_date
```

*# створення екземпляра класу працівник з використанням  
непозиційних параметрів у конструкторі*

```
w1 = Worker(name='Petrov Vadym', position='manager',  
work_from_date='09-12-2019', birth_date='18-02-1990')
```

```
conn = sqlite3.connect("staff_db.db")  
cursor = conn.cursor()
```

*# Створення таблиці після першого запуску скрипта потрібно  
закоментувати*

```
cursor.execute("""CREATE TABLE staff  
                (name text, position text, work_from_date  
text,  
                birth_date text)  
                """)
```

*# Створення першого запису в таблиці "вручну"*

```
cursor.execute("""INSERT INTO staff  
                VALUES ('Ivanov Ivan', 'director', '07-10-  
2019',  
                '2-12-1980')""")
```

*# Створення першого запису в таблиці із екземпляра класу  
Worker*

```
cursor.execute("""INSERT INTO staff  
                VALUES (?, ?, ?, ?)""", w1.get_data())
```

*# Зберігаємо зміни*

```
conn.commit()
```

*# Вставляємо список з трьох працівників*

```
all_workers = [('Borysov Mykola', 'ingeneer', '23-11-1976',  
'04-12-2019'),  
                ('Pavlyik Inna', 'secretary', '12-09-1991',  
'22-01-2020'),  
                ('Kolodych Leonid', 'ingeneer', '16-08-1986',  
'13-01-2020')]
```

```
cursor.executemany("INSERT INTO staff VALUES (?, ?, ?, ?)",  
all_workers)  
conn.commit()
```

*# Виведення на екран всіх записів*

```

print("Записи в таблиці бази даних у вигляді списку:")
sql = "SELECT * FROM staff"
cursor.execute(sql)
print(cursor.fetchall())

# Редагування запису для конкретного працівника
sql = """
UPDATE staff
SET position = 'main ingeneer'
WHERE name = 'Kolodych Leonid'
"""

cursor.execute(sql)
conn.commit()

# Виводимо список всіх інженерів
print("Список всіх ingeneer:")

sql = "SELECT * FROM staff WHERE position=?"
cursor.execute(sql, [("ingeneer")])
print(cursor.fetchall())

# Виведення на екран всіх записів
print("Список всіх записів в таблиці:")
for row in cursor.execute("SELECT rowid, * FROM staff ORDER
BY name"):
    print(row)

```

### **Завдання:**

Доповнити розроблену в лабораторній роботі 7 програму збереженням даних в базі даних. Додати до створеного класу методи для запису даних у базу даних, видалення записів із бази, пошуку записів по заданому критерію.

## **10.6. Контрольні запитання.**

- 10.6.1. Дайте визначення поняттям "реляційна база даних", "таблиця", "рядок".
- 10.6.2. Які СКБД підтримує Python?
- 10.6.3. Що таке курсор (cursor)?

## **Лабораторна робота №11**

### **Розроблення програмного забезпечення з графічним інтерфейсом мовою Python**

#### **11.1. Мета роботи**

Ознайомитися з організацією графічного інтерфейсу на основі бібліотеки tkinter

#### **11.2. Теоретичні відомості**

Tkinter - це пакет для Python, призначений для роботи з бібліотекою Tk. Бібліотека Tk містить компоненти графічного інтерфейсу користувача (graphical user interface - GUI), написані на мові програмування Tcl.

Під графічним інтерфейсом користувача (GUI) маються на увазі всі ті вікна, кнопки, текстові поля для введення, скролери, списки, радіокнопки, прапорці та ін., Які ви бачите на екрані, відкриваючи ту чи іншу програму. Через них ви взаємодієте з програмою і керуєте нею. Всі ці елементи інтерфейсу разом будемо називати віджетами (widgets).

В даний час майже всі програми, які створюються для кінцевого користувача, мають GUI. Рідкісні програми, які передбачають взаємодію з людиною, залишаються консольними. У попередніх двох курсах ми писали тільки консольні програми.

Існує безліч бібліотек GUI. Tk далеко не найпопулярніша, хоча з її використанням написано чимало проектів. Однак по ряду причин вона була обрана для Python за замовчуванням. Установчий файл Пітона зазвичай вже включає пакет tkinter в складі стандартної бібліотеки поряд з іншими модулями.

Не вдаючись в подробиці, Tkinter можна охарактеризувати як перекладач з мови Python на мову Tcl. Ви пишете програму на Python, а код модуля tkinter у вас за спиною переводить ваші інструкції на мову Tcl, який розуміє бібліотека Tk.

Додатки з графічним інтерфейсом користувача подієво-орієнтовані. Ви вже повинні мати уявлення про структурний і



бажано об'єктно-орієнтованому програмуванні. Подієво-орієнтоване орієнтоване на події. Тобто та чи інша частина програмного коду починає виконуватися лише тоді, коли трапляється ту чи іншу подію.

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 - об'єкти, породжені вбудованими класами.

Події бувають різними. Спрацював часовий чинник, хтось клікнув мишкою або натиснув Enter, почав вводити текст, перемкнув радіокнопки, прокрутив сторінку вниз і т. Д. Коли трапляється щось подібне, то, якщо був створений відповідний обробник, відбувається спрацювання певної частини програми, що приводить до якого-небудь результату.

Tkinter імпортується стандартно для модуля Python будь-яким із способів: `import tkinter`, `from tkinter import *`, `import tkinter as tk`. Можна імпортувати окремі класи, що робиться рідко. В даному курсі буде в основному використовуватися `from tkinter import *`.

Щоб написати GUI-програму, треба виконати приблизно наступне:

- Створити головне вікно.
- Створити віджети і конфігурувати їх властивостей (опцій).
- Визначити події, тобто те, на що буде реагувати програма.
- Визначити обробники подій, тобто те, як буде реагувати програма.
- Розташувати віджети в головному вікні.
- Запустити цикл обробки подій.

Послідовність не обов'язково така, але перший і останній пункти завжди залишаються на своїх місцях. Подивимося все це в дії.

Існують і інші способи створення інтерфейсів програм, написаних мовою Python. Серед них зокрема і PyQt – інтерфейс

Python до бібліотеки Qt.

В сучасних операційних системах будь користувальницький додаток укладено в вікно, яке можна назвати головним, так як в ньому розташовуються всі інші віджети. Об'єкт вікна верхнього рівня створюється від класу Tk модуля tkinter. Змінну, зв'язвану з об'єктом, часто називають root (корінь):

**root = Tk ()**

Клас Tk є базовим і викликається за допомогою конструктора tkinter.Tk(screenName=None, baseName=None, className='Tk', useTk=1), що створює віджет верхнього рівня, який зазвичай є головним вікном додатку.

Tkinter є бібліотекою, орієнтованою на події, в яких є головний цикл оброблення подій. Для запуску такого циклу використовується метод mainloop, а для виходу – метод quit.

До віджетів бібліотеки tkinter належать:

- Toplevel – вікно верхнього рівня, що зазвичай використовується під час створення багатовіконних програм;
- Button – кнопка;
- Label – мітка (напис без можливості редагування);
- Entry – віджет, що дозволяє ввести один рядок тексту;
- Text – віджет для введення багаторядкового тексту;
- Listbox – перелік, з якого можна обрати один або декілька елементів;
- Frame – рамка для організації віджетів всередині вікна;
- Checkbutton – віджет, що дозволяє вибрати деякий пункт у вікні;
- Radiobutton – дозволяє обрати тільки один пункт у вікні;
- Scale – віджет, що дозволяє вибрати значення з деякого діапазону;
- Scrollbar – віджет, що дозволяє прокручувати інший віджет;
- Menu – віджет для створення меню, що спливає або випадає;
- Menubutton – кнопка з меню;

- Canvas – основа для виведення графічних примітивів;
  - Message – віджет аналогічний Label, що дозволяє розташовувати багаторядкові тексти;
  - Progressbar – віджет, який відображає статус довготривалих операцій;
  - Separator – вертикальна або горизонтальна полоса розподілу;
  - Sizegrip – віджет, який дозволяє користувачу змінити розмір вікна верхнього рівня;
  - Treeview – ієрархічна колекція пунктів.
- Усі ці класи віджетів є підкласами класу Widget.

### 11.3. Програма роботи

11.3.1. Ознайомитися з теоретичним відомостями.

11.3.2. Створити та запустити на виконання приклади 1-3.

11.3.3. Виконати завдання 1-2.

### 11.4. Обладнання та програмне забезпечення

11.4.1. Персональний комп'ютер.

11.4.2. Інтерпретатор Python встановлений на ПК

### 11.5. Порядок виконання роботи і опрацювання результатів

**Приклад 1. Демонстрація розміщення віджетів на головному вікні**

```
from tkinter import *
from tkinter import ttk

root = Tk()
root.title('Example')
content = ttk.Frame(root, padding=(3, 3, 12, 12))
frame = ttk.Frame(content, borderwidth=5,
                    relief="sunken", width=200, height=100)
namebl = ttk.Label(content, text="Name")
name = ttk.Entry(content)
onevar = BooleanVar()
twovar = BooleanVar()
```

```

threevar = BooleanVar()
one = ttk.Checkbutton(content, text="One",
                      variable=onevar, onvalue=True)
two = ttk.Checkbutton(content, text="Two",
                      variable=twovar, onvalue=True)
three = ttk.Checkbutton(content, text="Three",
                       variable=threevar, onvalue=True)
ok = ttk.Button(content, text="OK")
cancel = ttk.Button(content, text="Cancel")
content.grid(column=0, row=0, sticky=(N, S, E, W))
frame.grid(column=0, row=0, columnspan=3, rowspan=2,
           sticky=(N, S, E, W))
namelbl.grid(column=3, row=0, columnspan=2, sticky=(N,
                                                    W),
             padx=5)
name.grid(column=3, row=1, columnspan=2, sticky=(N, E,
                                                    W), pady=5,
          padx=5)
one.grid(column=0, row=3)
two.grid(column=1, row=3)
three.grid(column=2, row=3)
ok.grid(column=3, row=3)
cancel.grid(column=4, row=3)
root.columnconfigure(0, weight=1)
root.rowconfigure(0, weight=1)
content.columnconfigure(0, weight=3)
content.columnconfigure(1, weight=3)
content.columnconfigure(2, weight=3)
content.columnconfigure(3, weight=1)
content.columnconfigure(4, weight=1)
content.rowconfigure(1, weight=1)
root.mainloop()

```

## Приклад 2. Відкриття текстового файлу.

```

from tkinter import *
import tkinter.filedialog

def LoadFile(ev):
    fn = tkinter.filedialog.Open(root, filetypes=[('.txt
files', '.txt')]).show()
    if fn == '':

```

```

        return
    textbox.delete('1.0', 'end')
    textbox.insert('1.0', open(fn, 'rt').read())

root = Tk()
panelFrame = Frame(root, height=20, bg='blue')
textFrame = Frame(root, height=40, width=50)
panelFrame.pack(side='top', fill='x')
textFrame.pack(side='bottom', fill='both', expand=1)
textbox = Text(textFrame, font='Arial 12', wrap='word')
scrollbar = Scrollbar(textFrame)
scrollbar['command'] = textbox.yview
textbox['yscrollcommand'] = scrollbar.set
textbox.pack(side='left', fill='both', expand=1)
scrollbar.pack(side='right', fill='y')
loadBtn = Button(panelFrame, text='Open')
loadBtn.bind("<Button-1>", LoadFile)

loadBtn.place(x=10, y=1, width=40, height=20)
root.mainloop()

```

## Приклад 2. Додавання подій при натисненні на кнопки

```

from tkinter import *

def triangle():
    canvas.coords(r, (0, 0, 0, 0))
    canvas.itemconfig(t, fill='yellow', outline='white')
    canvas.coords(t, (50, 200, 340, 200, 110, 60))
    text.delete(1.0, END)
    text.insert(1.0, 'Зображення трикутника')
    text.tag_add('title', '1.0', '1.end')
    text.tag_config('title', font=('Times', 14),
foreground='blue')

def rectangle():
    canvas.coords(t, (0, 0, 0, 0, 0, 0))
    canvas.itemconfig(r, fill='blue', outline='white')
    canvas.coords(r, (80, 50, 320, 200))
    text.delete(1.0, END)
    text.insert(1.0, 'Зображення прямокутника')
    text.tag_add('title', '1.0', '1.end')

```

```

text.tag_config('title', font=('Times', 14),
foreground='black')

win = Tk()
b_triangle = Button(text="Трикутник", width=15,
command=triangle)
b_rectangle = Button(text="Прямокутник", width=15,
command=rectangle)

canvas = Canvas(width=400, height=300, bg='#fff')
text = Text(width=55, height=5, bg='#fff', wrap=WORD)
t = canvas.create_polygon(0, 0, 0, 0, 0, 0)

r = canvas.create_rectangle(0, 0, 0, 0)
b_triangle.grid(row=0, column=0)
b_rectangle.grid(row=1, column=0)
canvas.grid(row=0, column=1, rowspan=10)
text.grid(row=11, column=1, rowspan=3)
win.mainloop()

```

### **Завдання:**

Модифікуйте приклад 2: додайте в меню кнопку Save, яка дозволяє зберегти модифікований текстовий файл.

Модифікуйте приклад 3: додайте ще дві кнопки – одну для малювання Кола, іншу для очищення полотна.

## **11.6. Контрольні запитання.**

11.6.1. Для чого використовують ГІ?

11.6.2. Назвіть основні елементи управління (віджети) бібліотеки tkinter.

11.6.3. Який метод віджету Tk() дозволяє задати мінімальні розміри вікна

11.6.4. Який метод віджету Tk() дозволяє задати максимальні розміри вікна?

11.6.5. Який метод віджету Tk() дозволяє задати заголовок вікна?

11.6.6. Для чого використовується віджет Label ()?

11.6.7. Для чого використовується віджет Button () ?

11.6.8. Для чого використовується віджет Canvas () ?

## Лабораторна робота №12

### Розробка додатків з графічним інтерфейсом. Програмування подій, робота з діалоговими вікнами

#### 12.1. Мета роботи

Вдосконалити навички роботи з організацією графічного інтерфейсу на основі бібліотеки tkinter

#### 12.2. Теоретичні відомості

##### Події

У системі сучасного графічного інтерфейсу є можливість відслідковувати різні події, пов'язані з клавіатурою та мишею, які відбуваються на "території" того або іншого віджета. У Tk події описуються текстовим рядком – шаблоном події, що складається з трьох елементів (модифікаторів, типу події та деталізації події).

##### Тип події    Зміст події

**Activate** Активізація вікна

**ButtonPress** Натискання кнопки миші

**ButtonRelease** Звільнення кнопки миші

**Deactivate** Деактивізація вікна

**Destroy** Закриття вікна

**Enter** Входження курсору в межі віджета

**FocusOut** Втрата фокуса вікном

**KeyPress** Натискання клавіші на клавіатурі

**KeyRelease** Відпускання клавіші на клавіатурі

**Leave** Вихід курсору за межі віджета

**Motion** Рух миші в межах віджета

**MouseWheel** Прокручування коліщати миші

**Reparent** Зміна батька вікна

**Visibility** Зміна видимості вікна

**FocusIn** Одержання фокуса вікном

"<ButtonPress-3>" або просто "<3>" – натискання правою кнопкою миші (тобто, третьою, якщо рахувати на трикнопочній

миші зліва праворуч);

"<Shift-Double-Button-1>" – подвійне натискання (лівою кнопкою миші) із натиснутою клавішею **Shift**.

Як модифікатори використовують такі (список неповний):

**Control, Shift, Lock,**

**Button1-Button5 або B1-B5,**

**Meta, Alt, Double, Triple.**

Тут символ позначає подію – натискання клавіші. Наприклад, "k" – те саме, що й "<KeyPress-k>". Для неалфавітно-цифрових клавіш є спеціальні назви:

**Cancel, BackSpace, Tab, Return, Shift\_L, Control\_L, Alt\_L, Pause, Caps\_Lock, Escape, Prior, Next, End, Home, Left, Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num\_Lock, Scroll\_Lock, space, less**

Тут <space> позначає пробіл, а <less> – знак менше. <Left>, <Right>, <Up>, <Down> – стрілки. <Prior>, <Next> – це **PageUp** та **PageDown**. Інші кнопки більш-менш відповідають написам на стандартній клавіатурі.

## Менеджери розташування

Наступний приклад демонструє принципи роботи менеджерів розташування, які є у Tk. У трьох рамках можна застосувати різні менеджери: **pack**, **grid** та **place**.

```
from tkinter import *
```

```
tk = Tk()
```

```
# Створюємо три рамки
```

```
frames = {}
```

```
b = {}
```

```
for fn in 1, 2, 3:
```

```
    f = Frame(tk, width=100, height=200, bg="White")
```

```
    f.pack(side=LEFT, fill=BOTH)
```

```
    frames[fn] = f
```

```
    for bn in 1, 2, 3, 4: # Створюємо кнопки для кожної з рамок
```

```
        b[fn, bn] = Button(frames[fn], text="%s.%s" % (fn, bn))
```

```
# Перша рамка:
```



```

# Спочатку дві кнопки прикріплюємо до лівого краю
b[1, 1].pack(side=LEFT, fill=BOTH, expand=1)
b[1, 2].pack(side=LEFT, fill=BOTH, expand=1)
# Ще дві - до нижнього
b[1, 3].pack(side=BOTTOM, fill=Y)
b[1, 4].pack(side=BOTTOM, fill=BOTH)
# Друга рамка:
# Дві кнопки зверху
b[2, 1].grid(row=0, column=0, sticky=NW + SE)
b[2, 2].grid(row=0, column=1, sticky=NW + SE)
# і одна на дві колонки знизу
b[2, 3].grid(row=1, column=0, colspan=2, sticky=NW + SE)
# Третя рамка:
# Кнопки заввишки та завширшки близько 40 % розмірів рамки,
якір у лівому верхньому куті.
# Координати якоря складають 1/10 від ширини й висоти рамки
b[3, 1].place(relx=0.1, rely=0.1, relwidth=0.4,
relheight=0.4,
            anchor=NW)
# Кнопка строго по центру. Якір у центрі кнопки
b[3, 2].place(relx=0.5, rely=0.5, relwidth=0.4,
relheight=0.4,
            anchor=CENTER)
# Якір у центрі кнопки. Координати якоря складають 9/10 від
ширини й висоти рамки
b[3, 3].place(relx=0.9, rely=0.9, relwidth=0.4,
relheight=0.4,
            anchor=CENTER)
tk.mainloop()

```

Менеджер `pack` просто заповнює внутрішній простір на підставі переваги того чи іншого краю, необхідності заповнити весь простір. У деяких випадках йому доводиться змінювати розміри підпорядкованих віджетів. Цей менеджер варто використовувати лише для досить простих схем розташування віджетів.

Менеджер `grid` поміщає віджети у клітки сітки (це дуже схоже на спосіб верстання таблиць в `html`). Кожному розташовуваному віджету дають координати в одній із чарунок сітки (`row` – рядок, `column` – стовпець), а також, якщо потрібно, стільки наступних чарунок (у рядках нижче або у стовпцях

праворуч), скільки він може зайняти (властивості `rowspan` або `columnspan`). Це найгнучкіший з усіх менеджерів.

Менеджер `place` дозволяє розташовувати віджети з довільними координатами та з довільними розмірами підпорядкованих віджетів. Розміри та координати можна вказувати у відсотках від розміру віджета-хазяїна.

Безпосередньо всередині одного віджета не можна використовувати більше одного менеджера розташування: менеджери можуть накласти суперечливі обмеження на вкладені віджети, що призведе до унеможливлення розташування внутрішніх віджетів

### **12.3. Програма роботи**

12.3.1. Ознайомитися з теоретичними відомостями.

12.3.2. Створити та запустити на виконання приклади 1-3.

12.3.3. Виконати завдання 1 згідно варіанту.

### **12.4. Обладнання та програмне забезпечення**

12.4.1. Персональний комп'ютер.

12.4.2. Інтерпретатор Python встановлений на ПК

### **12.5. Порядок виконання роботи і опрацювання результатів**

#### **Приклад 1. Діалогове вікно вибору кольору**

```
from tkinter import *
from tkinter import colorchooser

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)
        self.parent = parent
        self.initUI()

    def initUI(self):
        self.parent.title("Color chooser")
        self.pack(fill=BOTH, expand=1)
```

```

self.btn = Button(self, text="Choose Color",
                  command=self.onChoose)
self.btn.place(x=30, y=30)

self.frame = Frame(self, border=1,
                   relief=SUNKEN, width=100,
height=100)
self.frame.place(x=160, y=30)

def onChoose(self):
    (rgb, hx) = colorchooser.askcolor()
    self.frame.config(bg=hx)

def main():
    root = Tk()
    ex = Example(root)
    root.geometry("300x150+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

### **Завдання:**

Модифікуйте програму побудови геометричних фігур з лабораторної роботи 11 наступним чином: додайте головне меню, в якому пункт «Налаштування» містить підпункти – «Налаштування зображень» та «Налаштування тексту». При виборі пункту «Налаштування зображень» з'являється діалогове вікно для вибору кольору та розміру кожної з тьох фігур. При натисканні на підпункт головного меню «Налаштування тексту» аналогічно з'являється діалогове вікно для налаштування розміру і кольору тексту до кожного зображення.

## **12.6. Контрольні запитання.**

12.6.1. Як викликати функції опрацювання події?

12.6.2. Призначення інструкції `root.mainloop()`.

12.6.3. Де розміщуються віджети допомогою методу `grid`?

## **Лабораторна робота №13**

### **Побудова графіків математичних функцій у мові Python**

#### **13.1. Мета роботи**

Набуття навичок роботи з бібліотекою Matplotlib для візуалізації даних

#### **13.2. Теоретичні відомості**

##### **Matplotlib**

Matplotlib – бібліотека на мові програмування Python для візуалізації даних двовимірною 2D графікою (3D графіка також підтримується). Отримувані зображення можуть бути використані як ілюстрації в публікаціях. Зображення, які генеруються в різних форматах, можуть бути використані в інтерактивній графіці, наукових публікаціях, графічному інтерфейсі користувача, веб-додатках, де потрібно будувати діаграми (англ. plotting).

Бібліотека Matplotlib побудована на принципах ООП, але має процедурний інтерфейс pylab, який надає аналоги команд MATLAB.

Пакет підтримує багато видів графіків і діаграм:

- Графіки (line plot)
- Діаграми розсіювання (scatter plot)
- Стовпчасті діаграми (bar chart) і гістограми (histogram)
- Секторні діаграми (pie chart)
- Діаграми «Стовбур-листя» (stem plot)
- Контурні графіки (contour plot)
- Поля градієнтів (quiver)
- Спектральні діаграми (spectrogram)

Набір підтримуваних форматів зображень, векторних і растрових, можна отримати з словника FigureCanvasBase.filetypes. Типові підтримувані формати: EPS, EMF, JPEG, PDF, PNG, PostScript, RGBA, SVG, SVGZ, TIFF.

## Налаштування вигляду графіків

Крім того, щоб просто побудувати криву, було б добре її назвати, позначити осі, вивести легенду (це особливо стане в нагоді, якщо будувати кілька графіків). Крім того, інколи потрібно змінити вигляд самої кривої, межі її побудови.

```
from numpy import *
import matplotlib.pyplot as plt

t = linspace(0, 3, 51)
y = t * exp(-t ** 2)
plt.plot(t, y, 'r-.', label='t*exp(-t^2)')
plt.axis([0, 1, -0.05, 0.5]) # задання [xmin, xmax, ymin, ymax]
plt.xlabel('t') # позначення вісі абсцис
plt.ylabel('y') # позначення е вісі ординат
plt.title('Графік функції') # назва графіка
plt.legend() # вставка легенди (тексту в label)
plt.show()
```

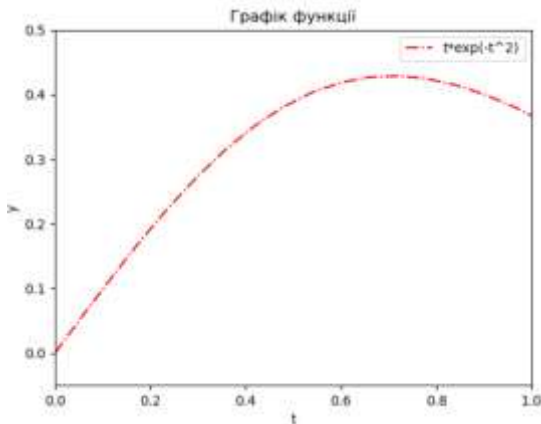


Рис 7.1.

Крім зазначених нововведень, позначених в коментарях, в аргументах функції `plot()` ми бачимо два нових. Останній задає текст легенди графіка. Строковий аргумент `r-`, відповідальний за те, що змінився вигляд кривої. За замовчуванням цей аргумент `b-` що означає синю (blue) суцільну лінію. Допустимі наступні значення:

Таблиця 7.1

b, blue	синій колір
c, cyan	блакитний колір
g, green	зелений колір
k, black	чорний колір
r, red	червоний колір
w, white	білий колір
y, yellow	жовтий колір
-	суцільна лінія
--	штрихова лінія
-. .	штрих-пунктирна лінія
:	пунктирна лінія

За замовчуванням легенда розташовується в правому верхньому куті, але можна її і перенести за рахунок аргументу loc. Цьому аргументу можна привласнювати і чисельне значення, але звичайно легше сприймається рядок. У таблиці нижче приводяться можливі варіанти.

Таблиця 7.2

Місце	String	Code
кращий варіант	best	0
вгорі справа	upper right	1
вгорі зліва	upper left	2
внизу справа	lower left	3
внизу зліва	lower right	4
справа	right	5
посередині зліва	center left	6
посередині справа	center right	7
посередині внизу	lower center	8
посередині вгорі	upper center	9
посередині	center	10

### Маркери

Тут також показано як можна об'єднувати відразу три графіка в одній інструкції. Крім того, видно, що можна не тільки використовувати маркери (y1) або лінії (y2), але і об'єднувати їх

разом (y3). Найбільш часто в наукових дослідженнях і журналах призводять графіки, що відрізняються один від одного саме маркерами, тому і в matplotlib для їх позначення є безліч способів:

- . точковий маркер;
- , точки, розміром з піксель;
- о кола;
- ∨ трикутники носом вниз;
- ∧ трикутники носом догори;
- > трикутники дивляться вправо;
- < трикутники дивляться вліво;
- s квадрати;
- p п'ятикутники;
- \* зірочки;
- h шестикутники;
- H повернені шестикутники;
- + плюси;
- x хрестики;
- D ромби;
- d вузькі ромби;
- | вертикальні зарубки.

### Додаткові аргументи plot()

Отже, в один аргумент ми можемо поставити відразу три параметра: першим вказуємо колір, другим – стиль лінії, третім – тип маркера. Однак вже така нотація може у людини незнайомої з нею, викликати подив. Крім того, вона не дозволяє розділяти параметри лінії і маркера, тому існує варіант з використанням keywords – також це дозволяє щедра функція plot():

Таблиця 7.3

Keyword argument		Що міняє
color або c		колір лінії
linestyle		стиль лінії, використовуються позначення, показані вище
linewidth		товщина лінії у вигляді float-числа

marker	вид маркера
markeredgecolor	колір краю (edge) маркера
markeredgewidth	товщина краю маркера
markerfacecolorh	колір маркера
markersize	Розмі маркера

Можна також вносити зміни у відмітки на осях координат. Робиться це за допомогою функцій `xticks()` і `yticks()`, в які передаються один або два списки значень: або просто список згаданих значень, або їх же, але спочатку ті місця, на які вони встають:

```
x = [5, 3, 7, 2, 4, 1]
plt.xticks(range(len(x)), ['a', 'b', 'c', 'd', 'e', 'f'])
plt.yticks(range(1, 8, 2))
```

Для нанесення сітки існує команда: `plt.grid(True)`

Для того, щоб одну або декілька осей виставити в логарифмічному масштабі застосовуються команди `plt.semilogx()` и `plt.semilogy()`.

## Збереження файлу

```
from numpy import *
import matplotlib.pyplot as plt

t = linspace(0, 1, 51)
y = t * exp(-t ** 2)
plt.plot(t, y)
plt.savefig('name_of_file.png', dpi=200)
```

Файл зберігається в тій же директорії з ім'ям і розширенням, зазначеним в першому аргументі. Другий необов'язковий аргумент дозволяє «на льоту» змінювати роздільну здатність картинки, що зберігається у файл.

Буває так, що дивитися на картинки в уже налаштованій програмі не потрібно і потрібно їх саме зберігати на майбутнє, щоб переглянути і порівняти їх всі разом. Тоді нам не потрібно



запускати вікно перегляду результатів. Для цього до колишніх інструкцій додаємо головному модулю повідомлення: `matplotlib.use('Agg')`

### **Текст, примітки**

Крім тексту в назвах, підписах до осей, легенди, можна безпосередньо вставляти його в графік за допомогою простої функції `text(x, y, text)`, де  $x$  і  $y$  координати, а `text` текстовий рядок. Ця функція вставляє текст відповідно до координат даних. Існує можливість вводити і в координатах графіка, в яких за  $(0, 0)$  приймається нижній лівий кут, а за  $(1, 1)$  правий верхній. Це робиться за допомогою функції `figtext(x, y, text)`.

Текстові функції, звичайно вставляють текст в графік, але часто буває потрібно саме вказати, виділити якийсь екстремум, незвичайну точку. Це легко зробити за допомогою приміток – функції `annotate('annotation', xy = (x1, y1), xytext = (x2, y2))`. Тут замість `annotation` ми пишемо текст примітки, замість  $(x1, y1)$  координати цікавої точки, замість  $(x2, y2)$  координати, де ми хочемо вставити текст.

## **13.3. Програма роботи**

13.3.1. Ознайомитися з основами візуалізації даних засобами бібліотеки `Matplotlib`.

13.3.2. Створити та запустити на виконання приклади 1-5, спробувати змінити параметри побудови графіків, модифікувати всі приклад так, щоб зберегти у файл отримані графіки.

13.3.3. Виконати завдання 1 згідно варіанту.

## **13.4. Обладнання та програмне забезпечення**

13.4.1. Персональний комп'ютер.

13.4.2. Інтерпретатор `Python` встановлений на ПК

## **13.5. Порядок виконання роботи і опрацювання результатів**

Для початку необхідно інсталиювати пакети `numpy` і `matplotlib`. Для цього зручно використовувати **pip** – це

інсталятор пакетів для мови Python.

### Установка **pip**

Завантажуємо Пітон скрипт `get-pip.py` (<https://bootstrap.pypa.io/get-pip.py>). При зберіганні переконайтесь, що зберегли файл із розширенням `.py`, а не `.txt`.

У командній стрічці (краще PowerShell) запустіть даний файл з допомогою Python інтерпретатора:

**`python \path\to\get-pip.py`**

В даному випадку файл `get-pip.py` лежить в директорії `D:/work`.

Після запуску даної команди маєте отримати повідомлення про успішну інсталяцію `pip` пакет менеджера. Інакше – потрібно розбиратись: або шлях до файла вказано не правильно, або Python не у видимих для командної стрічки шляхах.

Для остаточного тесту, що `pip` встановлено правильно просто запускаєте в командній стрічці (PowerShell) команду `pip`. Без аргументів дана команда видасть вам документацію по використанню.

Тепер маючи Python і Pip черга за `numpy` і `matplotlib`:

**`pip install numpy`**  
**`pip install matplotlib`**  
**`pip install numpy`**

### Приклад 1. Набір точок

```
import matplotlib.pyplot as plt
```

```
plt.plot([1, 3, 2, 4])  
plt.show()
```

Функція `plot()` будує графік, а функція `show()` його показує. Аргумент, що приймається функцією `plot()` – це послідовність у-значень. Інший, який ми опустили, що стоїть перед у – це послідовність х-значень. Оскільки його немає, графік генерується для чотирьох зазначених у, список з чотирьох х: [0, 1, 2, 3].

## Приклад 2. Функція

```
from numpy import * # для використання функцій exp та  
linspace  
import matplotlib.pyplot as plt
```

```
def f(t):  
    return t ** 2 * exp(-t ** 2)
```

```
t = linspace(0, 3, 51) # 51 точка між 0 та 3  
y = f(t)  
plt.plot(t, y)  
plt.show()
```

## Приклад 3. Налаштування вигляду графіків

```
import matplotlib.pyplot as plt
```

```
t = linspace(0, 3, 51)  
y = t ** 2 * exp(-t ** 2)  
plt.plot(t, y, 'g--', label='t^2*exp(-t^2)')  
plt.axis([0, 3, -0.05, 0.5]) # [xmin, xmax, ymin, ymax]  
plt.xlabel('t') # позначення вісі абсцис  
plt.ylabel('y') # позначення е вісі ординат  
plt.title('My first normal plot') # назва графіка
```

```
plt.legend() # вставка легенди (тексту в label)
plt.show()
```

#### Приклад 4. Декілька кривих на одному графіку

```
from numpy import *
import matplotlib.pyplot as plt

t = linspace(0, 3, 51)
y1 = t ** 2 * exp(-t ** 2)
y2 = t ** 4 * exp(-t ** 2)
y3 = t ** 6 * exp(-t ** 2)
plt.plot(t, y1, 'g^', # маркери із зелених трикутників
         t, y2, 'b--', # синя штрихова
         t, y3, 'ro-') # червоні круглі маркери
# з'єднані суцільною лінією
plt.xlabel('t')
plt.ylabel('y')
plt.title('Plotting with markers')
plt.legend(['t^2*exp(-t^2)',
           't^4*exp(-t^2)',
           't^6*exp(-t^2)'], # список легенди
          loc='upper left') # положення легенди
plt.show()
```

**Завдання:** зобразити 2d графік функції відповідно своєму варіанту та зберегти у .png файл.

1.  $Y(x)=x*\sin(5*x)$ ,  $x=[-2...5]$
2.  $Y(x)=1/x*\sin(5*x)$ ,  $x=[-5...5]$
3.  $Y(x)=2^x*\sin(10x)$ ,  $x=[-3...3]$
4.  $Y(x)=x^{1/2}*\sin(10*x)$ ,  $x=[0...5]$
5.  $Y(x)=15*\sin(10*x)*\cos(3*x)$ ,  $x=[-3...3]$
6.  $Y(x)=5*\sin(10*x)*\sin(3*x)$ ,  $x=[0...4]$
7.  $Y(x)=\sin(10*x)*\sin(3*x)/(x^2)$ ,  $x=[0...4]$
8.  $Y(x)=5*\sin(10*x)*\sin(3*x)/(x^{1/2})$ ,  $x=[1...7]$
9.  $Y(x)=5*\cos(10*x)*\sin(3*x)/(x^{1/2})$ ,  $x=[0...5]$
10.  $Y(x)=-5*\cos(10*x)*\sin(3*x)/(x^{1/2})$ ,  $x=[0...10]$
11.  $Y(x)=-5*\cos(10*x)*\sin(3*x)/(x^x)$ ,  $x=[0...5]$
12.  $Y(x)=5*\sin(10*x)*\sin(3*x)/(x^x)$ ,  $x=[0...8]$

13.  $Y(x)=x^{\sin(10*x)}, x=[1...10]$
14.  $Y(x)=-x^{\cos(5*x)}, x=[0...10]$
15.  $Y(x)=x^{\cos(x^2)}, x=[0...10]$
16.  $Y(x)=\cos(x^2)/x, x=[0...5]$
17.  $Y(x)=10*\cos(x^2)/x^2, x=[0...4]$
18.  $Y(x)=(1/x)*\cos(x^2+1/x), x=[1...10]$
19.  $Y(x)=\sin(x)*(1/x)*\cos(x^2+1/x), x=[-2...2]$
20.  $Y(x)=5*\sin(x)*\cos(x^2+1/x)^2, x=[1...10]$
21.  $Y(x)=5*\sin(1/x)*\cos(x^2+1/x)^2, x=[1...4]$
22.  $Y(x)=5*\sin(1/x)*\cos(x^2)^3, x=[-4...4]$
23.  $Y(x)=\cos(x^4)/x, x=[0...1]$
24.  $Y(x)=\sin(10^x)*\sin(3*x)/(x^2), x=[0...1]$
25.  $Y(x)=-2*\cos(10*x)*\sin(2*x)/(x^x), x=[0...1]$
26.  $Y(x)=8*\sin(x)*\sin(2*x)/(x^{(1/2)}), x=[1...10]$
27.  $Y(x)=2*x^{\sin(4*x)}, x=[1...4]$
28.  $Y(x)=5*\tan(x)*\exp(x^2+1/x), x=[1...10]$
29.  $Y(x)=-\cos(4*x)*\sin(2*x)/(1^x), x=[0...2]$
30.  $Y(x)=5*\sin(x)*\ln(x^2)^x, x=[1...10]$

### 13.6. Контрольні запитання.

13.6.1. Які засоби мова Python надає для роботи з 2D графікою? Які бібліотеки призначені для роботи з графікою?

13.6.2. Яким чином можна відобразити графік функції?

13.6.3. Яким чином можна зберегти зображення у файл?

16.6.4. Яким чином побудувати декілька кривих на одному графіку?

16.6.5. За допомогою якого параметру можна змінити колір графіка?

16.6.6. Як додати легенду до графіка?

16.6.7. Як зберегти графік функції у файл?

## Лабораторна робота №14

### Побудова 3D графіків. Робота з mplot3d Toolkit

#### 14.1. Мета роботи

Набуття навичок роботи з тривимірною графікою засобами мови програмування Python

#### 14.2. Теоретичні відомості

##### Гістограми

Для побудови гістограм (діаграм у вигляді набору стовпчиків) в Matplotlib використовуються функція `bar` і `barh`, які будують вертикальні або горизонтальні гістограми відповідно. Ці функції, як і інші функції малювання, імпортуються з модуля `pylab`. Функції `bar` і `barh` мають безліч необов'язкових параметрів з додатковими настройками, ми розглянемо тільки найбільш часто використовувані можливості для налаштування зовнішнього вигляду гістограм.

Функції `bar` і `barh` мають два обов'язкових параметра:

- Список координат розташування стовпчиків по осі X для `bar` або по осі Y для `barh`.
- Значення, що задають висоту (довжину) стовпчиків.

Довжини цих двох списків повинні бути рівні.

```
import matplotlib.pyplot as plt
import numpy as np

data1 = 10 * np.random.rand(5)
data2 = 10 * np.random.rand(5)
data3 = 10 * np.random.rand(5)
locs = np.arange(1, len(data1) + 1)
width = 0.27
plt.bar(locs, data1, width=width)
plt.bar(locs + width, data2, width=width, color='red')
plt.bar(locs + 2 * width, data3, width=width, color='green')
plt.xticks(locs + width * 1.5, locs)
plt.show()
```

### 3D графіка

Matplotlib дозволяє будувати 3D графіки. Імпортуємо необхідні модулі для роботи з 3D:

```
import matplotlib.pyplot as plt  
from mpl_toolkits.mplot3d import Axes3D
```

У бібліотеці доступні інструменти для побудови різних типів графіків.

Для побудови лінійного графіка використовується функція `plot()`.

`Axes3D.plot(self, xs, ys, *args, zdir='z', **kwargs)`

- `xs`: 1D масив
  - `x` координати.
- `ys`: 1D масив
  - `y` координати.
- `zs`: скалярне значення або 1D масив
  - `z` координати. Якщо переданий скаляр, то він буде присвоєний всім точкам графіка.
- `zdir`: {'x', 'y', 'z'}
  - Визначає вісь, яка буде прийнята за `z` напрямком, за замовчуванням: 'z'.
- `**kwargs`
  - Додаткові аргументи, аналогічні тим, що використовуються в функції `plot()` для побудови двовимірних графіків.

### Каркасна поверхня

Для побудови каркасної поверхні використовується функція `plot_wireframe()`.

`plot_wireframe(self, X, Y, Z, *args, **kwargs)`

- `X, Y, Z`: 2D масиви

Дані для побудови поверхні.

- `rcount, ccount`: `int` Максимальна кількість елементів каркаса, яке буде використано в кожному з напрямків. Значення за замовчуванням: 50.

- *rstride, cstride: int* Ці параметри впливають на величину кроку, з яким будуть братися елементи рядка / стовпця з переданих масивів. Параметри *rstride, cstride* і *rcount, ccount* є взаємовиключними.

- *\*\*kwargs* Додаткові параметри.

## Поверхня

Для побудови поверхні використовується функція *plot\_surface()*.

*plot\_surface(self, X, Y, Z, \*args, norm=None, vmin=None, vmax=None, lightsource=None, \*\*kwargs)*

## 14.3. Програма роботи

14.3.1. Ознайомитися з принципами побудови тривимірних та анімованих графіків.

14.3.2. Створити та запустити на виконання приклади 1-4.

14.3.3. Виконати завдання 1-2 згідно варіанту.

## 14.4. Обладнання та програмне забезпечення

14.4.1. Персональний комп'ютер.

14.4.2. Інтерпретатор Python встановлений на ПК

## 14.5. Порядок виконання роботи і опрацювання результатів

### Приклад 1. Побудова тривимірного графіка

```
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

mpl.rcParams['legend.fontsize'] = 10

fig = plt.figure()
ax = fig.gca(projection='3d')
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
```



```

x = r * np.sin(theta)
y = r * np.cos(theta)
ax.plot(x, y, z, label='параметрична крива')
ax.legend()
plt.show()

```

## Приклад 2. Побудова графіка функції $z=\sin(0.3x)*\cos(0.75y)$

```

import pylab
from mpl_toolkits.mplot3d import Axes3D
import numpy

def makeData():
    x = numpy.arange(-10, 10, 0.5)
    y = numpy.arange(-10, 10, 0.5)
    xgrid, ygrid = numpy.meshgrid(x, y)

    zgrid = numpy.sin(xgrid * 0.3) * numpy.cos(ygrid * 0.75)
    return xgrid, ygrid, zgrid

if __name__ == '__main__':
    x, y, z = makeData()

    fig = pylab.figure()
    axes = Axes3D(fig)
    axes.plot_surface(x, y, z, rstride=1, cstride=1)
    pylab.show()

```

## Приклад 3. Побудова поверхні

```

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Вхідні дані
u = np.linspace(0, 2 * np.pi, 100)

```

```

v = np.linspace(0, np.pi, 100)
x = 10 * np.outer(np.cos(u), np.sin(v))
y = 10 * np.outer(np.sin(u), np.sin(v))
z = 10 * np.outer(np.ones(np.size(u)), np.cos(v))

# Побудова поверхні
ax.plot_surface(x, y, z, color='b')
plt.show()

```

## Приклад 4. Анімована побудова графіка функції

```

from math import *
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def data_gen(t=0):
    cnt = 0
    while cnt < 1000:
        cnt += 1
        t += 0.1
        yield t, sin(2*pi*t) * exp(-t/10.)

def init():
    ax.set_ylim(-1.1, 1.1)
    ax.set_xlim(0, 10)
    del xdata[:]
    del ydata[:]
    line.set_data(xdata, ydata)
    return line,

fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
ax.grid()
xdata, ydata = [], []

def run(data):
    # update the data
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()

```

```

if t >= xmax:
    ax.set_xlim(xmin, 2*xmax)
    ax.figure.canvas.draw()
line.set_data(xdata, ydata)

return line,

ani = animation.FuncAnimation(fig, run, data_gen,
                              lit=False, interval=10,
                              repeat=False, init_func=init)

plt.show()

```

### **Завдання:**

**Завдання 1:** Зобразити гістограму частоти появи літер у певному тексті (текст зчитується із текстового файлу) та зберегти у .png файл.

**Завдання 2:** Додати анімацію до побудованого в лабораторній роботі 13 графіка.

## **14.6. Контрольні запитання.**

14.6.1. Яким чином можна відобразити гісторграму?

15.6.2. Який метод використовується для побудови поверхні? Які її параметри?

15.6.3. Який модуль бібліотеки matplotlib дозволяє будувати анімовані графіки?

## **Література:**

1. Лутц М. Программирование на Python, том I, 4-е издание. Пер. сангл. СПб. : Символ-Плюс, 2011. 992с.
2. Марк Саммерфилд "Программирование на Python 3" . Подробное руководство. Пер. с англ. СПб. : Символ-Плюс, 2013. 608 с.
3. Програмування числових методів мовою Python : підруч. / А. В. Анісімов, А. Ю. Дорошенко, С. Д. Погорілий, Я. Ю. Дорогий; за ред. А. В. Анісімова. К. : Видавничо-поліграфічний центр "Київський університет", 2014. 640 с.