МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

В.І.Попівщий, А.І.Безверхий

# ПРОГРАМУВАННЯ НА МОВІ PYTHON

Навчальний посібник
для здобувачів степеня вишої освіти бакалавра
спеціальності «Інженерія програмного забезпечення»
освітньо-професійної програми «Програмне забезпечення систем»

Затверджено
Вченою радою ЗНУ
протокол № від

Запоріжжя
2023

УДК 621.396.218

П 575

Попівщий В.І., Безверхий А.І. Програмування на мові Python: навчальний посібник для здобувачів степеня вищої освіти бакалавра спеціальності «Інженерія програмного забезпечення» освітньо-професійної програми «Програмне забезпечення систем».
Запоріжжя : ЗНУ, 2023. 180 с.

В навчальному посібнику подано в систематизованому вигляді програмний матеріал дисципліни «Програмування на мові Python». Викладено огляд екосистеми Python, розглянуто базові типи даних, прийоми роботи з функціями, модулями, пакетами, винятками, інструментальними засобами (Microsoft Visual Studio Code, IDLE), що підтримують створення програм на Python. Розглянуто основи об'єктно-орієнтованого програмування на Python, розробку програм з графічним інтерфейсом користувача, використання бібліотек NumPy, Matplotlib, Seaborn, Pygame. Для формування необхідних навичок запропоновано чотири навчальних завдання.

Наведено питання до самоконтролю, зразки завдань тестового контролю знань та список рекомендованої літератури.

Для здобувачів степеня вишої освіти бакалавра спеціальності «Інженерія програмного забезпечення» освітньо-професійної програми «Програмне забезпечення систем».

Рецензент

*В.З. Грищак***,** доктор технічних наук, професор, завідувач кафедри прикладної математики Запорізького національного університету

Відповідальний за випуск

*В.Г. Вербицький,* доктор фізико-математичних наук, професор, завідувач кафедри програмного забезпечення автоматизованих систем

ЗМІСТ

# ПЕРЕДМОВА

Екосистема мови програмування Python охоплює наукові обчислення (scientific computing), машинне навчання (machine learning), штучний інтелект (artificial intelligence), науку про дані (data science), великі дані (big data), обробку зображень (image processing), Web-розробку та інші галузі. Для кожної з названих галузей Python має як набір бібліотек (типу NumPy, TensorFlow) для підтримки відомих алгоритмів і додатків, так і фреймворки і інструменти, які сприяють швидкій розробці програм.

В наш час потреба в підготовці спеціалістів з розробки застосунків для вищеназваних галузей на мові Python зростає і тому включення в план підготовки бакалаврів спеціальності «Інженерія програмного забезпечення» курсу "Програмування на мові Python" відповідає сучасним вимогам.

**Метою** викладання навчальної дисципліни "Програмування на мові Python" є поглиблене вивчення мови Python, формування у студентів професійних компетенцій, знань, умінь і навичок володіння мовою програмування Python та ефективного її використання для розв'язку прикладних завдань.

Основними **завданнями** дисципліни є набуття навичок самостійної розробки програм на Python для різних розрахунків, обробки даних та візуалізації результатів у вигляді графіків, робота з інтегрованим середовищем розробки IDLE, вивчення принципів побудови, структури й прийомів роботи з інструментальними засобами редактора Microsoft Visual Studio Code, що підтримує створення програмного забезпечення на мові Python, формування навичок використання стандартної бібліотеки Python та зовнішніх бібліотек для підтримки інженерних і наукових застосунків.

У результаті вивчення навчальної дисципліни студент повинен

**знати:**

- основні характеристики мови програмування Python;
- парадигми програмування, які підтримує Python;
- базові конструкції мови та типи даних;
- функції, модулі, пакети Python;
- винятки, роботу з файлами в Python;
- технологію об'єктно-орієнтованого програмування в Python;
- технології розробки застосунків з графічним інтерфейсом користувача;
- структуру та основні функції бібліотек NumPy та Matplotlib;
- технологію розробки ігр за допомогою бібліотеки Pygame.

**вміти:**

- інсталювати програмне забезпечення та налагоджувати середовище розробки для мови Python;

- будувати застосунки з використанням основних типів даних мови Python;
- використовувати функції, модулі, пакети, винятки;
- використовувати можливості об'єктно-орієнтованого програмування при реалізації практичних завдань;
- розробляти програми з графічним інтерфейсом користувача;
- розробляти додатки з підтримкою бібліотек NumPy та Matplotlib.

Згідно з вимогами освітньо-професійної програми студенти повинні досягти таких **компетентностей**:

ЗК01. Здатність до абстрактного мислення, аналізу та синтезу.

ЗК05. Здатність вчитися і оволодівати сучасними знаннями.

ЗК07. Здатність працювати в команді.

ФК14. Здатність брати участь у проектуванні програмного забезпечення, включаючи проведення моделювання (формальний опис) його структури, поведінки та процесів функціонування.

ФК15. Здатність розробляти архітектури, модулі та компоненти програмних систем.

ФК17. Здатність дотримуватися специфікацій, стандартів, правил і рекомендацій в професійній галузі при реалізації процесів життєвого циклу.

Відповідно до структурно-логічної схеми освітньо-професійної програми, засвоєння навчального матеріалу курсу «Програмування на мові Python» логічно пов’язане з використанням знань, вмінь та навичок, отриманих у результаті вивчення дисциплін «Основи програмування» та «Об’єктно-орієнтоване програмування» і може використовуватись в курсі «Системи штучного інтелекту» і при написанні кваліфікаційної роботи бакалавра.

Навчальний посібник складається з шести розділів, в які включені 14 тем, що відповідають навчальним тижням семестру.

У першому розділі описана екосистема мови Python. Детально описані інструментальні засоби підтримки та налаштування середовища розробки додатків.

Другий та третій розділи присвячені викладенню основних типів таних, функцій, модулів, пакетів, винятків. Розглянуті також стандартна бібліотека Python та робота з файлами.

Четвертий розділ присвячений об'єктно-орієнтованому програмуванню на Python. Детально описані атрибути класів та об’єктів, типи методів.

В п’ятому розділі описана розробка програм з графічним інтерфейсом користувача.

Шостий розділ присвячений використанню бібліотек NumPy та Matplotlib для підтримки інженерних та наукових застосунків. Розглянута також бібліотека Pygame.

Щоб забезпечити більш ефективне сприйняття матеріалу та формування практичних навичок в кожному розділі розміщені питання для самоконтролю та навчальні завдання.

# РОЗДІЛ 1 БАЗОВІ ЕЛЕМЕНТИ МОВИ PYTHON

## Тема 1. Екосистема Python. Інсталяція програм. Базові типи даних

**Мета:** розглянути загальну характеристику мови Python; з'ясувати в чому полягає причина популярності цієї мови; ознайомитись з основними галузями екосистеми Python; навчитись встановлювати необхідне програмне забезпечення для створення робочого місця розробника; вивчити базові типи даних.

**План**

1. Загальна характеристика мови Python. Екосистема Python.
2. Інсталяція інтерпретатора Python, налаштуваня середовища розробника.
3. Базові типи даних Python.
4. Введення і виведення даних.

### Основні терміни і поняття

*Екосистема Python. Інтерпретатор. Плагін. IDLE. Ідентифікатор. Базовий тип. Ключові слова. Літерал.*

### Основні теоретичні положення

#### 1. Загальна характеристика мови Python. Екосистема Python

Python - інтерпретована, об'єктно-орієнтована мова програмування високого рівня з динамічною типізацією, автоматичним управлінням пам'яттю і зручними високорівневими структурами даних.

Мову програмування Python створив в 1991 році голландець Гвідо ван Россум (Guido van Rossum). Офіційний сайт мови: http://python.org. Python названий на честь британського серіалу 70-х "Літаючий цирк Монті Пайтона". Python впевнено входить до п'ятірки найпопулярніших комп'ютерних мов. На Python написані програми, якими ви користуєтесь щодня, такі як Google, YouTube, Instagram, Netflix. Python - найпопулярніша мова для навчальних курсів інформатики у провідних американських коледжах, офіційна мова навчання для вузів у Франції.

Екосистема мови програмування Python охоплює наукові обчислення (scientific computing), машинне навчання (machine learning), штучний інтелект (artificial intelligence), науку про дані (data science), великі дані (big data), обробку зображень (image processing), Web-розробку та інші галузі. Для кожної з названих галузей Python має як набір бібліотек (типу NumPy, TensorFlow) для підтримки відомих алгоритмів і додатків, так і фреймворки і інструменти, які сприяють швидкій розробці програм.

Останнім часом Python став надзвичайно популярним у світі штучного інтелекту, науки про дані та машинного навчання. Якщо ви хочете отримати добре оплачувану роботу з програмування в цікавій сфері, зараз Python є хорошим вибором.

Python працює майже скрізь і має “batteries included” (батареї в комплекті) - корисні програми у своїй стандартній бібліотеці [5].

Історично склалися дві версій мови: Python 2 і Python 3. Python 2 - це минуле, а Python 3 - майбутнє. Підтримка мови Python 2 припинилася у січні 2020 року. Ми будемо вивчати Python 3.

## 2. Встановлення Python.

Для завантаження останньої версії Python 3 відвідайте сторінку сайту за адресою www.python.org/downloads. Розглянемо інсталяцію на операційну систему Windows. Після того, як ви завантажили інсталятор Python, його слід виконати. Настійно рекомендується на першій сторінці вибрати обидва пункти внизу екрана: *Install launcher for all users* та *Add Python 3.x to PATH*. Потім слід натиснути кнопку *Customize installation*, а в наступному діалоговому вікні, де пропонується вибрати компоненти для установки, треба залишити всі вибрані прапорці і натиснути *Next*.

На наступному кроці потрібно задати деякі додаткові налаштування і вибрати шлях установки. Перевірте, чи встановлені прапорці *Associate files with Python, Create shortcuts for installed applications, Add Python to environment variables, Precompile standard library*.

Тепер уточнимо шлях, по якому буде встановлено Python. Рекомендуємо задати *C:\Python39*, тобто безпосередньо в корінь диску. В цьому випадку ми уникнемо проблем при установці додаткових бібліотек.

Задавши всі необхідні параметри, натискаємо кнопку *Install* і позитивно відповідаємо на попередження UAC що з'явилося на екрані. Після завершення установки відкриється вікно, в якому слід натиснути кнопку *Close* для виходу з програми установки. Після успішної установки інтерпретатора можна приступати до написання коду на Python.

### Встановлення редактора Visual Studio Code

Інсталюйте редактор Visual Studio Code (https://code.visualstudio.com/). Він нам знадобиться для виконання лабораторних робіт і вправ даного навчального курсу. Після завершення установки запустіть редактор і встановіть плагін від фірми Microsoft для кращої підтримки мови Python. Для цього на панелі зліва виберіть піктограму з підказкою Extensions, в рядку пошуку введіть “Python”, в випадаючому списку, що з’явиться, виберіть самий верхній пункт “Python extension for Visual Studio Code” від Microsoft і натисніть Install.

### Запуск Python.

З встановленим інтерпретатором Python можна працювати в двох режимах: в *інтерактивному режимі*, в якому кожна інструкція Python подається і

зразу виконується, і в *режимі виконання скриптів*, тобто програм на мові Python, які знаходяться в файлах з розширенням *.py*. Ми будемо в основному використовувати другий режим. Для цього нам буде потрібен або текстовий редактор, або IDLE (Integrated Development and Learning Environment).

IDLE - це інтегроване середовище розробки, яке об'єднує кілька інструментів розробки в одну програму, включаючи наступне:

- Оболонка Python (Python shell), яка працює в інтерактивному режимі. Ви можете ввести оператори Python у командному рядку оболонки і негайно їх виконати. Ви також можете запускати повні програми Python.
- Текстовий редактор, колір якого кодує ключові слова Python та інші частини програм.
- Інструмент “check module”, який перевіряє програму Python на наявність синтаксичних помилок без запуску програми.
- Інструменти пошуку, які дозволяють знаходити текст в одному або кількох файлах.
- Інструменти форматування тексту, які допомагають підтримувати послідовні рівні відступу в програмі Python.
- Налагоджувач, який дозволяє проходити покроково програму Python і спостерігати за змінами змінних під час виконання кожного оператора.
- Кілька інших додаткових інструментів для розробників.

Програмне забезпечення IDLE поставляється в комплекті з Python. Коли ви встановлюєте інтерпретатор Python, автоматично встановлюється і IDLE. Після встановлення Python у вашій системі група програм Python з’явиться у списку програм меню Пуск. Один із пунктів у групі програм матиме назву IDLE (Python GUI). Натисніть цей пункт, щоб запустити IDLE, і ви побачите вікно оболонки Python. Підказка >>> вказує на те, що інтерпретатор чекає, поки ви введете оператор Python.

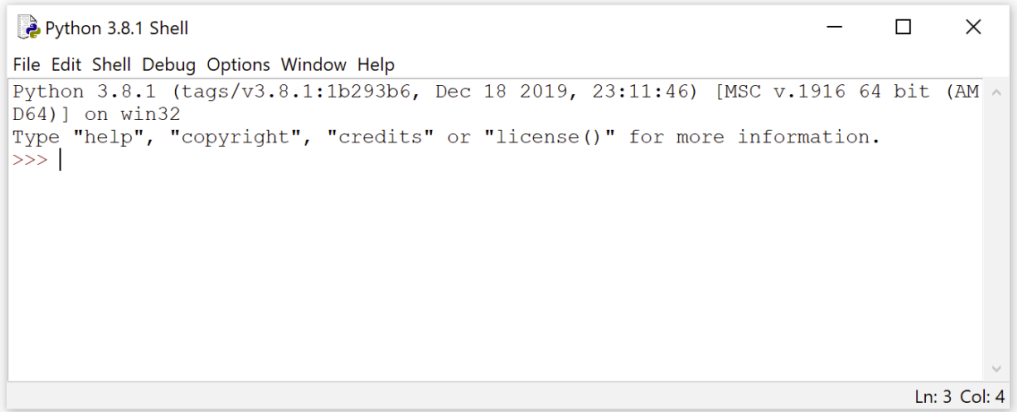


Рис. 1 Так виглядає Python shell (оболонка Python)

Оболонка Python - дуже зручний інструмент для тестування команд, особливо на перших етапах роботи. Але якщо ви вийдете з оболонки і ввійдете в неї знову, то всі команди, що були введені, зникнуть. Крім того, оболонку не можна використовувати для створення реальної програми. Щоб створити спра-

вжню програму, потрібно написати код в текстовому файлі і зберегти його з розширенням .py. Такий файл називається сценарієм Python. Щоб створити сценарій Python треба натиснути File => New File у верхньому меню оболонки Python. З'явиться текстовий редактор, тощо.

Але в данному навчальному курсі рекомендованим способом створення програм (сценаріїв) Python є використання текстового редактора Visual Studio Code.

### 3. Базові типи даних Python

Деякі мови збирігають значення змінних в пам’яті, відстежуючи їхні розміри та типи. Замість того, щоб безпосередньо обробляти такі необроблені значення даних, Python обертає кожне значення даних — логічні значення, цілі числа, числа з плаваючою точкою, рядки, навіть великі структури даних і функції — у пам’яті як об’єкт певного класу [9]. “Мантра” Python стверджує, що все є об’єкт.

Таблиця 1 показує основні типи даних у Python. Другий стовпець (Type) містить ім'я Python цього типу. Третій стовпець (Mutable?) вказує, чи можна змінити значення після створення. Examples показують один або кілька прикладів літералів такого типу.

Таблиця 1. Основні типи даних Python

| Name | Type | Mutable? | Examples |
|---|---|---|---|
| Boolean | `bool` | no | `True, False` |
| Integer | `int` | no | `47, 25000, 25_000` |
| Floating point | `float` | no | `3.14, 2.7e5` |
| Complex | `complex` | no | `3j, 5 + 9j` |
| Text string | `str` | no | `'alas', "alack", '''a verse attack'''` |
| List | `list` | yes | `['Winken', 'Blinken', 'Nod']` |
| Tuple | `tuple` | no | `(2, 4, 8)` |
| Bytes | `bytes` | no | `b'ab\xff'` |
| ByteArray | `bytearray` | yes | `bytearray(...)` |
| Set | `set` | yes | `set([3, 5, 7])` |
| Frozen set | `frozenset` | no | `frozenset(['Elsa', 'Otto'])` |
| Dictionary | `dict` | yes | `{'game': 'bingo', 'dog': 'dingo', 'drummer': 'Ringo'}` |

У Python, якщо ви хочете знати тип чого-небудь (змінна або значення літерала), ви можете використовувати type(thing), де type() – це одна з вбудованих функцій Python. Якщо ви хочете перевірити, чи вказує змінна на об’єкт певного типу, використовуйте функцію isinstance (type), наприклад:

```
>>> type(7)
<class 'int'>
>>> type(7) == int
True
>>> isinstance(7, int)
```

```
True
```

**Ідентифікатори Python**

Ідентифікатор Python використовується для позначення змінної, класу, функції, модуля або будь-якого іншого об'єкта. Python чутливий до регістру, тому великі та малі літери вважаються різними.

Правила декларування ідентифікатора:

- Єдиними дозволеними символами є літери, цифри та символ підкреслення (_).
- Ідентифікатор не повинен починатися з цифри.
- Ідентифікатори за своєю природою чутливі до регістру.
- Якщо ідентифікатор починається з підкреслення (_), то це приватний ідентифікатор.
- Символ долара ($) заборонений у Python.

Імена, які починаються і закінчуються двома підкресленнями (__), зарезервовані для використання в Python, тому не варто використовувати їх зі своїми власними змінними. Цей шаблон іменування був обраний тому, що розробники додатків навряд чи вибрали б його для власних змінних.

Наприклад, ім'я функції зберігається в системній змінній function.__ name__, а її рядок документації – в змінній function.__ doc__:

```
>>> def amazing():
        '''This is the amazing function.
        Want to see it again?'''
        print('This function is named:', amazing.__name__)
        print('And its docstring is:', amazing.__doc__)

>>> amazing()
This function is named: amazing
And its docstring is: This is the amazing function.
    Want to see it again?
```

Головній програмі присвоєно спеціальну назву __main__.

**Зарезервовані ключові слова**

Це ключові слова, зарезервовані мовою програмування, які не можна використовувати як ідентифікатор у програмі. Існує 33 зарезервовані ключові слова (залежить від версії), як показано в Таблиці 2. Отримати список ключових слів можна подавши команду

```
>>> help("keywords")
```

Таблиця 2. Зарезервовані ключові слова Python

| Reserved keywords | | | | |
|---|---|---|---|---|
| False | class | finally | is | return |
| None | continue | for | lambda | try |
| True | def | from | nonlocal | while |
| And | del | global | not | with |
| As | elif | if | or | yield |
| Assert | else | import | pass | |
| Break | except | in | raise | |

**Літерали**

Літерали - це числа, рядки або символи, які з'являються безпосередньо в програмі. Мова програмування містить дані з точки зору введення та виведення, а будь-який вид даних може бути представлений з точки зору значення. Значення може бути будь-якого типу, наприклад літерали, що містять цифри, літерали, що містять символи або рядки, тощо. Щоб знати тип значення в Python, існує вбудована функція під назвою type.

`Syntax: type(value)`

Приклади літералів:

20

25.5

'A'

"PYTHON"

Рис. 2 демонструє використання функції `type` для літералів.

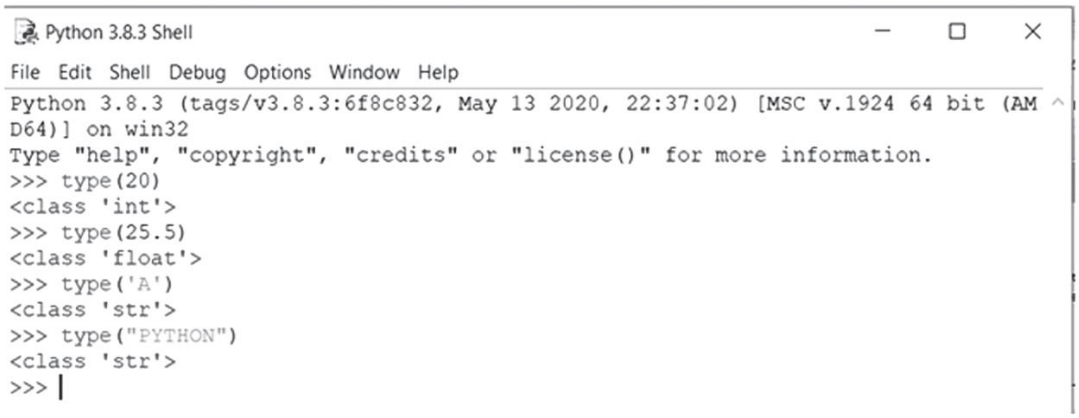


Рис. 2 Застосування функції `type` для літералів

## 4. Введення і виведення даних

**Введення**

Програми Python можуть читати введення з клавіатури, викликаючи функцію *input*. Ця функція змушує програму зупинятися і чекати, поки користувач щось набере. Коли користувач натискає клавішу Enter, символи, введені корис-

тувачем, повертаються функцією введення. Потім програма продовжує виконання. Вхідні значення зазвичай зберігаються у змінній за допомогою оператора присвоєння, щоб їх можна було використовувати пізніше в програмі. Функція input() може приймати необов'язковий аргумент-запрошення строкового типу; при виконанні функції повідомлення буде з'являтися на екрані і інформувати користувача про запитувані дані. Наприклад, наступний вираз читає значення, введене користувачем, і зберігає його у змінній з ім'ям `a`.

```
a = input("Введіть значення змінної a ")
```

Функція *input* завжди повертає рядок, тобто послідовність символів. Якщо значення, яке читається, - це ім'я людини, назва книги чи назва вулиці, то доцільно зберігати значення як рядок. Але якщо значення є числовим, наприклад, вік, температура або вартість страви в ресторані, то рядок, введений користувачем, зазвичай перетворюється на число. Перетворення в ціле число виконується шляхом виклику функції *int*, тоді як перетворення в число з плаваючою точкою здійснюється шляхом виклику функції *float*.

```
n1 = int(input("Enter n1 : "))
n2 = int(input("Enter n2 : "))
sum = n1 + n2
print("n1 + n2 = ", sum)
```

**Виведення даних**

Вивести значення на екран можна за допомогою функції `print`.

```
print("When x is", x, "the value of y is", y)
```

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

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

де *value* – перераховуються через кому об'єкти, які необхідно вивести на екран; *sep* – рядок-розділювач між об'єктами (за замовчуванням стоїть пробіл); *end* – рядок, що розміщений після останнього об'єкту (за замовчуванням – перехід на новий рядок).

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

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

**Основні принципи синтаксису мови Python**

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

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

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).

```
x = 1
if x:
    y = 2
    if y:
        print('block2')
    print('block1')
print('block0')
```

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.

**Коментарі**

Коментарі - це короткі примітки, розміщені в різних частинах програми, які пояснюють, як ці частини програми працюють. Хоча коментарі є важливою частиною програми, інтерпретатор Python їх ігнорує. Коментарі призначені для особи, яка читає код програми, а не для комп'ютера.

У Python ви починаєте коментар із символу #. Коли інтерпретатор Python бачить символ #, він ігнорує все від цього символу до кінця рядка.

Приклад.

```
# 60 sec/min * 60 min/hr * 24 hr/day
seconds_per_day = 86400
```

або

```
seconds_per_day = 86400 # 60 sec/min * 60 min/hr * 24 hr/day
```

**Числові типи даних**

***Цілі числа (Integer Numbers)***

Ціле число - це комбінація додатніх і від'ємних чисел, включаючи (нуль) 0. У програмі цілочисельні літерали записуються без коми та з провідним знаком мінус для позначення від'ємного значення.

Приклад:

```
a = 10
print(a)
print(type(a))
10
<class 'int'>
```

Цілі значення відображаються такими способами:

A) Десяткова форма (основа 10)

Система числення за замовчуванням; дозволені значення 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

B) Двійкова форма (Base-2)

Дозволені значення - 0 і 1; літеральне значення має префікс 0b або 0B

C) вісімкова форма (Base-8)

Дозволені значення 0, 1, 2, 3, 4, 5, 6, 7; літеральні значення мають префікс 0o або 0O.

D) Шістнадцяткова форма (Base-16)

Дозволені цифри 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a – f; літеральні значення повинні мати префікс 0x або 0X

Приклад:

```
p = 100
q = 0o100
r = 0X10c
s = 0B1010
print(p)
print(q)
print(r)
print(s)
100
64
268
10
```

Цілочисельні операції

Таблиця 3. Цілочисельні операції

| Operator | Description | Example | Result |
|---|---|---|---|
| + | Addition | 5 + 8 | 13 |
| - | Subtraction | 90 - 10 | 80 |
| * | Multiplication | 4 * 7 | 28 |
| / | Floating-point division | 7 / 2 | 3.5 |
| // | Integer (truncating) division | 7 // 2 | 3 |
| % | Modulus (remainder) | 7 % 3 | 1 |
| ** | Exponentiation | 3 ** 4 | 81 |

***Числа з плаваючою точкою***

Числа з плаваючою точкою складаються з цілого числа, десяткової точки і дробової частини. Довжина дійсних чисел може бути нескінченною, тобто кількість цифр у дробовій частині не обмежена. Таким чином, Python використовує числа з плаваючою точкою для представлення дійсних чисел.

Приклад:

```
f = 13.22
print(type(f))
f1 = 13.22e4
print(f1)
<class 'float'>
132200.0
```

***Комплексні числа***

Комплексні числа виражаються у вигляді (a + bj), де a і b - дійсні числа, а j - уявна частина.

Приклад:

```
c = 30+2.6j
print(c)
```

```
30+2.6j
print(type(c))
<class 'complex'>
```

***Boolean Type***

У Python булевий тип даних представлений як *bool*. Це примітивний тип даних зі значеннями True або False. Значення True представлено як 1, а False - як 0. Крім того, якщо додати два True, воно видасть 2 як результат (True + True = 2), тоді як коли False віднято від True, воно видасть 1 як вихід (True - False = 1).

Приклад:

```
b = True
print(type(b))
<class 'bool'>
```

**Питання для самоконтролю**

1. Які ви знаєте правила декларування ідентифікаторів в мові Python?
2. Перелічте змінні (mutable) і незмінні типи даних.
3. Що таке літерал?
4. Як виконується введення і виведення даних в мові Python?

**Завдання для самостійного виконання**

**Завдання 1.** Встановіть на домашньому комп'ютері інтерпретатор мови програмування Python. В стандартному середовищі розробки IDLE протестуйте приклади, розглянуті в лекції.

**Завдання 2.** Встановіть на домашньому комп'ютері редактор Visual Studio Code і налаштуйте його для роботи з мовою програмування Python.

## Тема 2. Базові конструкції мови Python

**Мета:** розглянути базові конструкції мови Python; з'ясувати як здійснюється управління потоком виконання; вивчити умовні, циклічні вирази, вирази передачі управління.

**План**

1. Огляд засобів управління потоком виконання.
2. Умовні вирази.
3. Циклічні вирази.
4. Вирази передачі управління.

## 🖉Основні терміни і поняття

*Управляючі інструкції. If statement. If-Else Statement. If-Elif-Else Statement. Цикл For. Цикл While. Break Statement. Continue Statement. Pass Statement. Цикл з Else блоком.*

## 🖘 Основні теоретичні положення

**Управління потоком виконання у Python (Control Flow in Python)**
Управляючі інструкції Python діляться на наступні категорії (Рис. 3).

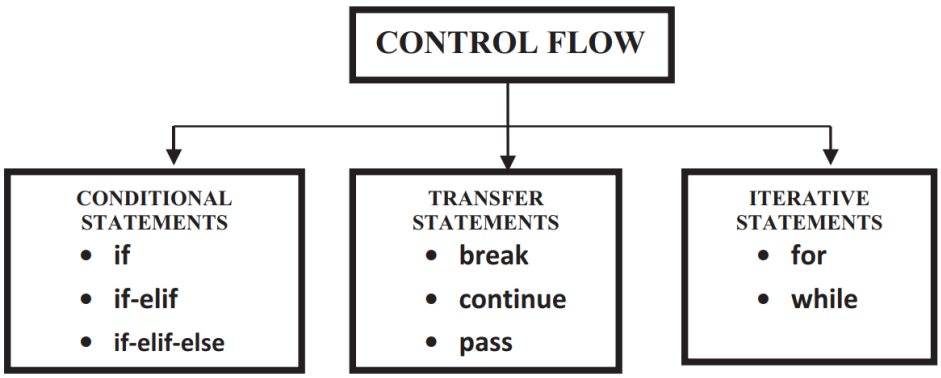


Рис. 3 Управляючі інструкції Python

**Умовні вирази**

***If statement***

Оператор if виконує оператор, якщо дана умова відповідає True.
Синтаксис

```
if condition: statement
```

Або

```
if condition:
    statement1
    statement2
    statement3
```

Приклад.

```
n = input("Enter Name: ")
if n == "Vijay" :
    print("Correct Information")
print("Done")
```

***If-Else Statement***

Оператор if-else враховує умови True та False. Якщо умова, вказана у if, є True, то Action1 буде оброблено, інакше Action2 буде оброблено.

```
if condition:
    Action1
else:
    Action2
```

Приклад.

```
n = input("Enter Name: ")
if n == "Vijay":
    print("Correct Information")
else:
    print("Not Correct Information")
```

***If-Elif-Else Statement***

Оператор if-elif-else враховує кілька умов. Якщо умова, вказана в if, є True, то дія Action1 буде оброблена, якщо умова, задана як condition2, є істинною, то дія Action2 буде оброблена, якщо всі умови elif - хибні, то дія за замовчуванням, вказана у else, буде оброблена.

Синтаксис:

```
if condition:
    Action1
elif condition2:
    Action2
elif condition3:
    Action3
elif condition4:
    Action4
else:
    default action
```

Приклад:

```
n = input("Enter Input: ")
if n == "Man":
    print("This is Male")
elif n == "Women":
    print("This is Female")
elif n == "Kids":
    print("This is child")
else :
    print("No Information")
```

**Циклічні вирази (Iterative Statements)**

Якщо ви хочете кілька разів обробити групу виразів, вам слід віддати перевагу ітераційним операторам. Python пропонує два види ітераційних виразів.

***Цикл For***

Цикл for в Python дещо відрізняється від інших мов програмування. У Python цикл for повторює послідовність об'єктів, тобто повторює кожне значення в послідовності. Якщо ви хочете щось обробити на кожному елементі, доступному в деякій серії, вам слід вибрати цикл for.

Синтаксис:

```
for p in sequence:
    body
```

Де, `sequence` може бути будь-яким рядком або колекцією, body виконується для всіх елементів, доступних в послідовності.

```
n = "Python"
```

```
for i in n:
    print(i)
```
Результат:
```
P
y
t
o
n
```

***Цикл While***

Цикл while є оператором управління циклом у Python і часто використовується в програмуванні для повторного виконання операторів циклу. Він обробляє послідовність тверджень, поки умова є істинною. Якщо ви хочете послідовно обробляти набір операторів, поки якась умова не є False, вам слід вибрати цикл while.

Синтаксис:
```
while condition:
   body
```
Приклад:
```
a = 1
while a <= 4:
   print(a)
   a = a + 1
```
Результат:
```
1
2
3
4
```

**Вирази передачі управління (Transfer Statements)**

***Break Statement***

Ключове слово break дозволяє програмісту завершити цикл. Цикл негайно припиняється, коли зустрічається оператор break у циклі, і управління програмою переходить до першого оператора, що слідує за циклом. Ви можете вибрати оператор break всередині циклу, щоб розірвати виконання циклу на основі умови.

Приклад:
```
list = [15, 18, 25, 45]
for i in list:
    if i>18:
        print("This is the location")
        break
    print(i)
```
Результат:
```
15
18
This is the location
```

***Continue Statement***

Оператор continue є прямою протилежністю оператору break. Натрапивши на оператор continue всередині циклу, інші оператори всередині тіла будуть пропущені, а стан циклу буде перевірено, чи повинен цикл продовжуватися чи припинятися. Інструкція continue може бути використана для пропуску поточної ітерації та продовження наступної ітерації.

Приклад:

```
list = [15, 17, 25, 45, 65, 9, 12]
for i in list:
    if i>18:
        continue
    print(i)
```

Результат:

```
15
17
9
12
```

***Pass Statement***

Залежно від використовуваної програми, якщо потрібен блок, який нічого не виконуватиме, ви можете оголосити порожній блок, використовуючи ключове слово pass.

Приклад:

```
for i in range(8):
    if i%2==0:
        print(i)
    else:
        pass
```

Результат:

```
0
2
4
6
```

***Цикл з Else блоком***

Блок else відразу після for/while циклу виконується лише тоді, коли цикл НЕ припиняється через break.

Такий тип else є корисним, тільки якщо в циклі присутня умова if, яка так чи інакше залежить від змінної циклу.

У наведеному нижче прикладі оператор else буде виконуватися лише в тому випадку, якщо жоден елемент масиву не є парним, тобто якщо оператор if не був виконаний для жодної ітерації. Тому для масиву [1, 9, 8] if виконується у третій ітерації циклу, а отже, else, що є після циклу for, ігнорується. У випадку масиву [1, 3, 5] if не виконується для жодної ітерації, а отже, else після циклу виконується.

```
# Python 3.x program to check if an array consists
# of even number
def contains_even_number(l):
     for ele in l:
```

```
            if ele % 2 == 0:
                 print ("list contains an even number")
                 break

        # This else executes only if break is NEVER
        # reached and loop terminated after all iterations.
        else:
             print ("list does not contain an even number")

# Driver code
print ("For List 1:")
contains_even_number([1, 9, 8])
print (" \nFor List 2:")
contains_even_number([1, 3, 5])

Output:

For List 1:
list contains an even number

For List 2:
list does not contain an even number
```

**Питання для самоконтролю**

1. Які ви знаєте категорії управляючих конструкцій Python?
2. Наведіть синтаксис if-elif-else виразу.
3. Чим відрізняється цикл for в мові Python від інших мов програмування?
4. Як виконується в Python цикл з else блоком?

## РОЗДІЛ 2 ТИПИ ДАНИХ В МОВІ PYTHON

### Тема 3 Рядки. Кортежі

**Мета:** розглянути рядки мови Python, операції з рядками та методи рядків; з'ясувати як здійснюється форматування рядків; вивчити кортежі та базові операції для кортежів; закріпити пройдений матеріал за допомогою вирішення навчального завдання.

**План**

1. Рядки. Базові операції з рядками.
2. Методи рядків.
3. Форматування рядків.
4. Кортежі. Створення та базові операції.

## 🖉Основні терміни і поняття

*Рядок. Зріз. Форматування. F-рядок. Кортеж (Tuple). Immutable. Mutable.*

## Основні теоретичні положення

**Рядки**

У Python 3 рядок (тип str) - це незмінна послідовність, що зберігає символи Unicode. Рядки можуть міститися як в одиночних, так і в подвійних лапках. Однак, початок і кінець рядка повинен відкриватися та закриватися однаковим типом лапок. Метою використання двох видів лапок є можливість створювати рядки, що містять лапки чи апострофи. Отже, усередині одинарних лапок можна розташувати подвійні та навпаки. Можна також використовувати три одинарні або три подвійні лапки, наприклад, щоб створити багаторядкові рядки.

У мові Python немає окремого символьного типу. Символ – це просто рядок довжини 1.

**Базові операції**

- Конкатенація (додавання)

```
>>>
>>> S1 = 'spam'
>>> S2 = 'eggs'
>>> print(S1 + S2)
'spameggs'
```

- Дублювання рядка

```
>>>
>>> print('spam' * 3)
spamspamspam
```

- Довжина рядка

```
>>>
>>> len('spam')
4
```

- Доступ за індексом

```
>>>
>>> S = 'spam'
>>> S[0]
's'
>>> S[2]
'a'
>>> S[-2]
'a'
```

Як видно з прикладу, в Python є можливість доступу по негативного індексу, при цьому відлік йде від кінця рядка.

- Отримання зрізу

Оператор вилучення зрізу: [X: Y]. X - початок зрізу, а Y - закінчення; символ з номером Y в зріз не входить. За умовчанням перший індекс дорівнює 0, а другий - довжині рядка.

```
>>>
>>> s = 'spameggs'
>>> s[3:5]
'me'
>>> s[2:-2]
'ameg'
>>> s[:6]
'spameg'
>>> s[1:]
'pameggs'
>>> s[:]
'spameggs'
```

Крім того, можна задати крок, з яким потрібно витягувати зріз.

```
>>>
>>> s[::-1]
'sggemaps'
>>> s[3:5:-1]
''
>>> s[2::2]
'aeg'
```

Приклад. Визначити, чи має ім’я файлу розширення .py

```
if s[len(s)-3:len(s)] == '.py':
    print("This is a Python program.")
```

**f-рядок**

Новим для Python 3.6 функціоналом є форматоване рядкове значення, або f-рядок, який забезпечує простий спосіб підстановки значень в послідовності символів.

```
>>> name = 'Johnny'
>>> print(f'Hello {name}.')
Hello Johnny.
```

**Методи рядків**

S.find(str, [start],[end]) - Пошук підрядка в рядку. Повертає номер першого входження або -1.

S.replace(шаблон, заміна) - Заміна шаблону.

S.split(символ) - Розбиття рядка по роздільнику.

S.join(список) - Збірка рядка зі списку з роздільником S.

S.isdigit() - Чи складається рядок з цифр.

S.islower() - Чи складається рядок із символів в нижньому регістрі.

S.upper() - Перетворення рядка до верхнього регістру.

S.startswith(str) - Чи починається рядок S з шаблону str.

S.count(str, [start],[end]) - Повертає кількість неперетинаючихся входжень підрядка в діапазоні [початок, кінець] (0 і довжина рядка за замовчуванням).

S.lstrip([chars]) - Видалення символів пробілів на початку рядка.

S.rstrip([chars]) - Видалення символів пробілів в кінці рядка.

S.strip([chars]) - Видалення символів пробілів на початку і в кінці рядка.

S.format(*args, **kwargs) - Форматування рядка.

Більш детальну інформацію про ці та інші методи рядків дивись в документації (https://docs.python.org/3/library/stdtypes.html#string-methods ).

*Форматування*

Ви бачили, що можна об'єднати рядки за допомогою +. Давайте розглянемо, як інтерполювати значення даних у рядки за допомогою різних форматів. Ви можете використовувати це для створення звітів, форм та інших результатів, де вигляд тексту має значення.

Крім функцій у попередньому розділі, Python має три способи форматування рядків:

- Old style (старий стиль, підтримується в Python 2 і 3);
- New style (новий стиль, - Python 2.6 і вище);
- f-рядки (Python 3.6 і вище).

***Old style: %***

Старий стиль форматування рядків має форму `format_string % data`. Усередині рядка форматування є послідовності інтерполяції. Таблиця 4 ілюструє, що найпростіша послідовність - це %, за яким слідує буква, що вказує тип даних для форматування.

Таблиця 4. Типи перетворення

| | |
|---|---|
| %s | string |
| %d | decimal integer |
| %x | hex integer |
| %o | octal integer |
| %f | decimal float |
| %e | exponential float |
| %g | decimal or exponential float |
| %% | a literal % |

Ви можете використовувати %s для будь-якого типу даних, і Python відформатує його як рядок без зайвих пробілів. Нижче наведено кілька простих прикладів. Спочатку ціле число:

```
>>> '%s' % 42
'42'
>>> '%d' % 42
'42'
>>> '%x' % 42
'2a'
>>> '%o' % 42
'52'
```

Число з плаваючою точкою:

```
>>> '%s' % 7.03
```

```
'7.03'
>>> '%f' % 7.03
'7.030000'
>>> '%e' % 7.03
'7.030000e+00'
>>> '%g' % 7.03
'7.03'
```

Ціле число і літерал %:

```
>>> '%d%%' % 100
'100%'
```

Давайте спробуємо деяку рядкову та цілочисельну інтерполяцію:

```
>>> actor = 'Richard Gere'
>>> cat = 'Chester'
>>> weight = 28
>>> "My wife's favorite actor is %s" % actor
"My wife's favorite actor is Richard Gere"
>>> "Our cat %s weighs %s pounds" % (cat, weight)
'Our cat Chester weighs 28 pounds'
```

Ви можете додати інші значення у рядку форматування між % та специфікатором типу для позначення мінімальної та максимальної ширини, вирівнювання та заповнення символів. Давайте коротко розглянемо ці значення:

- Початковий символ "%".
- Необов'язковий символ вирівнювання: нічого або '+' означає вирівнювання по правому краю, а '-' означає вирівнювання по лівому краю.
- Необов'язкову ширину поля minwidth для використання.
- Необов'язковий "." символ для розділення minwidth і maxchars.
- Необов’язковий параметр maxchars (якщо тип конвертації s), який вказує, скільки символів надрукувати зі значення даних. Якщо тип перетворення f, це вказує точність (скільки цифр надрукувати після десяткової коми).
- Символ типу перетворення з попередньої таблиці.

Це все дещо заплутано, тому ось кілька прикладів для рядка форматування:

```
>>> thing = 'woodchuck'
>>> '%s' % thing
'woodchuck'
>>> '%12s' % thing
'   woodchuck'
>>> '%+12s' % thing
'   woodchuck'
>>> '%-12s' % thing
'woodchuck   '
>>> '%.3s' % thing
'woo'
>>> '%12.3s' % thing
'         woo'
>>> '%-12.3s' % thing
'woo         '
```

Варіанти форматування для дійсних чисел:

```
>>> thing = 98.6
>>> '%f' % thing
'98.600000'
>>> '%12f' % thing
'   98.600000'
>>> '%+12f' % thing
'  +98.600000'
>>> '%-12f' % thing
'98.600000   '
>>> '%.3f' % thing
'98.600'
>>> '%12.3f' % thing
'      98.600'
>>> '%-12.3f' % thing
'98.600      '
```

Варіанти форматування для цілих чисел:

```
>>> thing = 9876
>>> '%d' % thing
'9876'
>>> '%12d' % thing
'        9876'
>>> '%+12d' % thing
'       +9876'
>>> '%-12d' % thing
'9876        '
>>> '%.3d' % thing
'9876'
>>> '%12.3d' % thing
'        9876'
>>> '%-12.3d' % thing
'9876        '
```

Для цілого числа %+12d просто змушує друкувати знак, а рядки форматування з .3 не мають для цілих жодного ефекту, вони працюють для float.

***New style: {} and format()***

Форматування старого стилю все ще підтримується у Python 2. Для Python 3 використовуйте форматування “нового стилю”, описане в цьому розділі. Якщо у вас Python 3.6 або новіша версія, f-рядки ще кращі.

“New style” форматування має форму **format_string.format(data)**.

Рядок формату не зовсім такий, як у попередньому розділі. Найпростіше використання показано тут:

```
>>> thing = 'woodchuck'
>>> '{}'.format(thing)
'woodchuck'
```

Аргументи функції format () мають бути в порядку як заповнювачі {} у рядку формату:

```
>>> thing = 'woodchuck'
>>> place = 'lake'
>>> 'The {} is in the {}.'.format(thing, place)
'The woodchuck is in the lake.'
```

За допомогою форматування в новому стилі ви також можете вказати аргументи за позицією, наприклад:

```
>>> 'The {1} is in the {0}.'.format(place, thing)
'The woodchuck is in the lake.'
```

Значення 0 відноситься до першого аргументу, place, а 1 - до thing.

Аргументи до format () також можуть бути іменованими

```
>>> 'The {thing} is in the {place}'.format(thing='duck',
place='bathtub')
'The duck is in the bathtub'
```

або словником:

```
>>> d = {'thing': 'duck', 'place': 'bathtub'}
```

У наступному прикладі {0} є першим аргументом format () (словник d):

```
>>> 'The {0[thing]} is in the {0[place]}.'.format(d)
'The duck is in the bathtub.'
```

Усі ці приклади друкували свої аргументи у форматах за замовчуванням. Форматування нового стилю має дещо інше визначення формату рядка від старого стилю (наведені приклади):

• Початкова двокрапка (':').

• Необов’язковий символ заповнення (за замовчуванням ' ') для вставки в рядок, якщо він коротший за minwidth.

• Додатковий символ вирівнювання. Вирівнювання вліво - за замовчуванням. '<' також означає ліворуч, '>' означає праворуч, а '^' означає центр.

• Додатковий знак для цифр. Знак мінус для від'ємних чисел і пробіл (' ') для додатних.

• Необов'язкова minwidth. Необов’язковий період ('.') Для розділення minwidth та maxchars.

• Необов'язкова maxchars.

• Тип перетворення.

```
>>> thing = 'wraith'
>>> place = 'window'
>>> 'The {} is at the {}'.format(thing, place)
'The wraith is at the window'
>>> 'The {:10s} is at the {:10s}'.format(thing, place)
'The wraith     is at the window    '
>>> 'The {:<10s} is at the {:<10s}'.format(thing, place)
'The wraith     is at the window    '
>>> 'The {:^10s} is at the {:^10s}'.format(thing, place)
'The   wraith   is at the   window  '
>>> 'The {:>10s} is at the {:>10s}'.format(thing, place)
'The     wraith is at the     window'
>>> 'The {:!^10s} is at the {:!^10s}'.format(thing, place)
'The !!wraith!! is at the !!window!!'
```

***Newest Style: f-strings***

f-рядки з'явилися в Python 3.6 і тепер є рекомендованим способом форматування рядків.

Щоб створити f-рядок:

• Введіть букву f або F безпосередньо перед початковим лапком.

• Додайте імена змінних або вирази до фігурних дужок ({}), щоб отримати їх значення у рядку.

Це схоже на форматування "нового стилю" попереднього розділу, але без функції format () і без порожніх дужок ({}) або позиційних ({1}) у рядку форматування.

```
>>> thing = 'wereduck'
>>> place = 'werepond'
>>> f'The {thing} is in the {place}'
'The wereduck is in the werepond'
```

Вирази також допускаються всередині фігурних дужок:

```
>>> f'The {thing.capitalize()} is in the {place.rjust(20)}'
'The Wereduck is in the              werepond'
```

f-рядки використовують таку саму мову форматування (ширину, відступ, вирівнювання), що й форматування нового стилю, після ':'.

```
>>> f'The {thing:>20} is in the {place:.^20}'
'The              wereduck is in the ......werepond......'
```

Починаючи з Python 3.8, f-рядки отримують нову можливість, яка стане в нагоді, коли потрібно надрукувати імена змінних, а також їх значення. Це зручно під час налагодження.

```
>>> f'{thing =}, {place =}'
thing = 'wereduck', place = 'werepond'
```

**Кортежі (Tuples)**

Більшість комп'ютерних мов може представляти послідовність елементів, індексованих їх цілочисельним положенням: перший, другий і так далі до останнього. Ви вже бачили рядки Python, які є послідовністю символів.

Python має ще дві структури послідовностей: *кортежі* та *списки*. Вони містять нуль або більше елементів. На відміну від рядків, елементи можуть бути різних типів. Фактично, кожен елемент може бути будь-яким об'єктом Python. Це дозволяє створювати складні структури даних.

Чому Python містить і списки, і кортежі? *Кортежі незмінні (immutable);* коли ви призначаєте значення елементам кортежу, вони фіксуються і не можуть бути змінені. *Списки є змінними (mutable),* тобто ви можете вставляти та видаляти елементи.

***Створення кортежа за допомогою ком та* ()**

Синтаксис створення кортежів трохи суперечливий, що демонструють наступні приклади. Почнемо зі створення порожнього кортежа за допомогою ():

```
>>> empty_tuple = ()
>>> empty_tuple
()
```

Щоб створити кортеж з одним або кількома елементами, поставте після кожного елемента кому. Це працює для одноелементних кортежів:

```
>>> one_marx = 'Groucho',
>>> one_marx
('Groucho',)
```

Ви можете укласти їх у дужки і при цьому отримати той самий кортеж:

```
>>> one_marx = ('Groucho',)
>>> one_marx
('Groucho',)
```

Але якщо у вас є один елемент в дужках і пропущена кома, ви не отримаєте кортеж, а лише елемент (у цьому прикладі рядок "Groucho"):

```
>>> one_marx = ('Groucho')
>>> one_marx
'Groucho'
>>> type(one_marx)
<class 'str'>
```

Якщо у вас більше одного елемента, поставте всі коми, крім останньої:

```
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo'
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

Python включає дужки при відображенні кортежу. Вони часто вам не потрібні, коли ви визначаєте кортеж, але використання дужок трохи безпечніше, і це допомагає зробити кортеж більш помітним:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

Кортежі дозволяють призначати кілька змінних одночасно:

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> a, b, c = marx_tuple
>>> a
'Groucho'
>>> b
'Chico'
>>> c
'Harpo'
```

Іноді це називають *розпаковуванням* кортежів (tuple unpacking).

***Створення кортежа за допомогою tuple()***

Функція перетворення tuple() робить кортежі з інших типів, наприклад зі списка:

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> tuple(marx_list)
('Groucho', 'Chico', 'Harpo')
```

***Об'єднання кортежів за допомогою +***

```
>>> ('Groucho',) + ('Chico', 'Harpo')
('Groucho', 'Chico', 'Harpo')
```

***Дублювання елементів за допомогою ****

```
>>> ('yada',) * 3
('yada', 'yada', 'yada')
```

***Порівняння кортежів***

```
>>> a = (7, 2)
>>> b = (7, 2, 9)
>>> a == b
```

```
False
>>> a <= b
True
>>> a < b
True
```

***Ітерація за допомогою for - in***

```
>>> words = ('fresh','out', 'of', 'ideas')
>>> for word in words:
...     print(word)
...
fresh
out
of
ideas
```

***Змінення кортежу***

Ви це не можете! Як і рядки, кортежі є незмінними, тому ви не можете змінити існуючий кортеж. Як ви бачили нещодавно, ви можете об'єднати кортежі, щоб створити новий:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop,')
>>> t1 + t2
('Fee', 'Fie', 'Foe', 'Flop')
```

Здається це означає, що ви можете змінити кортеж таким чином:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop,')
>>> t1 += t2
>>> t1
('Fee', 'Fie', 'Foe', 'Flop')
```

Але це не той самий t1. Python створив новий кортеж з оригінальних кортежів, на які вказують t1 і t2, і присвоїв цьому новому кортежу назву t1. Ви можете побачити за допомогою функції id (), що назва змінної t1 вже вказує на нове значення:

```
>>> t1 = ('Fee', 'Fie', 'Foe')
>>> t2 = ('Flop',)
>>> id(t1)
4365405712
>>> t1 += t2
>>> id(t1)
4364770744
```

**Питання для самоконтролю**

1. Які послідовності є незмінними (immutable) а які змінними (mutable)?
2. Які базові операції з рядками ви знаєте?
3. Які методи рядків ви знаєте?
4. Що таке f-рядок?
5. Які методи форматування рядків ви знаєте?

6. Що таке кортеж і як його можна створити?

 **Тестові завдання для перевірки знань**

1. Перший індекс у рядку це:
A. -1
B. 1
C. 0
D. Розмір рядка мінус одиниця

2. Цей метод рядка повертає найменший індекс у рядку, де знайдено вказаний підрядок.
A. first_index_of
B. locate
C. find
D. index_of

3. Цей метод рядка повертає True, якщо рядок містить лише алфавітні символи та має довжину принаймні один символ.
A. метод isalpha
B. метод alpha
C. метод alphabetic
D. метод isletters

## Тема 4 Списки

**Мета:** розглянути списки як один з основних типів послідовностей (колекцій) у мові Python; з'ясувати сутність управління елементами списку; ознайомитись із основними методами списків.

### План

1. Знайомство зі списками. Створення списку. Отримання елементів.
2. Додавання елементів. Змінювання елементів.
3. Методи списків remove, pop, index, count.
4. Сортування списків.
5. Копіювання списків.
6. Генератори списків (List comprehensions).
7. Список списків.

### Основні терміни і поняття

*Послідовність. Список. Ітерабельний тип даних. Методи списків. Генератор списку.* List comprehensions.

## ⛃ Основні теоретичні положення

### Знайомство зі списками. Створення списку. Отримання елементів

Послідовність — це об’єкт, який містить кілька елементів даних, які зберігаються один за одним. Ви можете виконувати операції над послідовністю, щоб досліджувати збережені в ній елементи та маніпулювати ними.

У Python існує кілька різних типів об’єктів послідовності. У цьому розділі ми розглянемо один з основних типів послідовностей (колекцій) – списки. Списки використовують для відстеження елементів колекцій за їхнім порядком, особливо коли порядок і вміст можуть змінитися. На відміну від рядків, списки змінюються. Ви можете змінити наявний список, додати нові елементи та видалити або замінити наявні елементи. Одне і те ж значення може зустрічатися у списку кілька разів.

У Python замість масивів використовуються списки, словники, кортежі. Багатомірний масив реалізується за допомогою списку списків (вкладеного списку). За необхідності використовувати саме масиви, наприклад, при обробці великого об’єму даних, слід застосовувати бібліотеку NumPy, яка дозволяє з ними працювати [2].

Список в Python - це упорядкований набір об'єктів, в список можуть одночасно входити об'єкти різних типів. Список складається з нуля або більше елементів, розділених комами та оточених квадратними дужками:

```
lst = [12, ’b’, 34.6, ’derevo’]
empty_list = [ ]
weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday']
first_names = ['Graham', 'John', 'Terry', 'Terry', 'Michael']
```

Функція list() перетворює інші ітерабельні типи даних (такі як кортежі, рядки, множини та словники) у списки. Наступний приклад перетворює рядок у список односимвольних рядків:

```
>>> list('cat')
['c', 'a', 't']
```

*Список створює метод split() рядка:*

```
>>> day = '9/19/2019'
>>> day.split('/')
['9', '19', '2019']
```

*Отримання елементів списку по індексу:*

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[0]
'Groucho'
>>> marxes[1]
'Chico'
>>> marxes[2]
'Harpo'
```

Знову ж таки, як і у рядках, від’ємні індекси відлічуються назад від кінця:

```
>>> marxes[-1]
'Harpo'
>>> marxes[-2]
'Chico'
```

*Отримання елементів за допомогою зрізів.*

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[0:2]
['Groucho', 'Chico']
```

Як і у випадку рядків, крок для зрізів може бути відмінним від одиниці. В наступному прикладі зріз починається з початку і йде з кроком 2:

```
>>> marxes[::2]
['Groucho', 'Harpo']
```

Тут ми починаємо з кінця і йдемо ліворуч через 2:

```
>>> marxes[::-2]
['Harpo', 'Groucho']
```

І, нарешті, спосіб реверсувати список:

```
>>> marxes[::-1]
['Harpo', 'Chico', 'Groucho']
```

Жоден із цих фрагментів не змінив сам список `marxes`, тому що ми не присвоїли їх `marxes`. Щоб змінити наявний список, скористайтеся list.reverse ():

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.reverse()
>>> marxes
['Harpo', 'Chico', 'Groucho']
```

**Додавання елементів. Змінювання елементів**

*Додавання елемента в кінець списку за допаванняомогою методу append().*

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.append('Zeppo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

*Додавання елемента за зміщенням за допомогою insert().*

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.insert(2, 'Gummo')
>>> marxes
['Groucho', 'Chico', 'Gummo', 'Harpo']
```

*Об'єднання списків за допомогою extend() або +*

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.extend(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

Крім того, ви можете використовувати + або + =:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes += others
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

*Зміна елемента за допомогою [offset]*

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[2] = 'Wanda'
```

```
>>> marxes
['Groucho', 'Chico', 'Wanda']
```

*Змінювання елементів за допомогою зрізів*

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [8, 9]
>>> numbers
[1, 8, 9, 4]
```

Права частина навіть не повинна мати таку саму кількість елементів, як зріз ліворуч:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = [7, 8, 9]
>>> numbers
[1, 7, 8, 9, 4]
```

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = []
>>> numbers
[1, 4]
```

Насправді, справа не обов'язково має бути лише список, а може бути будь-який ітерабельний об'єкт Python:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = (98, 99, 100)
>>> numbers
[1, 98, 99, 100, 4]
```

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1:3] = 'wat?'
>>> numbers
[1, 'w', 'a', 't', '?', 4]
```

**Методи списків remove, pop, index, count**

*Видалення елемента через Offset за допомогою del*

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo', 'Karl']
>>> marxes[-1]
'Karl'
>>> del marxes[-1]
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo']
```

*Видалення елемента за значенням за допомогою remove ()*

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.remove('Groucho')
>>> marxes
['Chico', 'Harpo']
```

*Отримання елемента за допомогою Offset та видалення його за допомогою pop ()*

Ви можете отримати елемент зі списку та видалити його зі списку одночасно за допомогою pop(). Якщо ви викликаєте метод pop() зі зміщенням, він поверне елемент із цим зміщенням; при виклику без аргументів метод pop()

використовує за замовчуванням індекс -1. Отже, pop (0) повертає заголовок (початок) списку, а pop() або pop (-1) повертає хвіст (кінець), як показано тут:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.pop()
'Zeppo'
>>> marxes
['Groucho', 'Chico', 'Harpo']
>>> marxes.pop(1)
'Chico'
>>> marxes
['Groucho', 'Harpo']
```

*Знайдення зміщення елемента за значенням за допомогою index ()*

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.index('Chico')
1
```

*Перевірка наявності значення з in*

Pythonic спосіб перевірити наявність значення у списку є використання in:

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> 'Groucho' in marxes
True
>>> 'Bob' in marxes
False
```

*Порахування кількості значень за допомогою count()*

Щоб підрахувати, скільки разів певне значення зустрічається у списку, скористайтеся count ():

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.count('Harpo')
1
>>> marxes.count('Bob')
0
>>> skit = ['cheeseburger', 'cheeseburger', 'cheeseburger']
>>> skit.count('cheeseburger')
3
```

*Перетворення списку в рядок за допомогою join()*

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> ', '.join(marxes)
'Groucho, Chico, Harpo'
```

**Сортування списків**

*Змінення порядку елементів за допомогою sort() або sorted()*

Вам часто доведеться сортувати елементи у списку за їх значеннями. Python пропонує дві функції:

• Метод списку sort(), що сортує сам список на місці.

• Загальну функцію sorted(), яка повертає відсортовану копію списку.

Якщо елементи у списку є числовими, вони за замовчуванням сортуються у порядку зростання числа. Якщо це рядки, їх сортують за алфавітом:

```
>>> marxes = ['Vika', 'Nika', 'Mary', 'Anna', 'Alex']
>>> sorted_marxes = sorted(marxes)
>>> sorted_marxes
['Alex', 'Anna', 'Mary', 'Nika', 'Vika']
```

sort_marxes - це новий список, і його створення не змінило вихідний список:

```
>>> marxes
['Vika', 'Nika', 'Mary', 'Anna', 'Alex']
```

Але виклик метода списку sort() у списку marxes дійсно змінює marxes:

```
>>> marxes.sort()
>>> marxes
['Alex', 'Anna', 'Mary', 'Nika', 'Vika']
```

Якщо всі елементи вашого списку мають один і той же тип (наприклад, рядки в `marxes`), sort () буде працювати коректно. Іноді можна навіть змішувати типи - наприклад, цілі числа і плаваючі:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort()
>>> numbers
[1, 2, 3, 4.0]
```

За замовчуванням порядок сортування за зростанням, але ви можете додати аргумент reverse = True, щоб встановити його за спаданням:

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort(reverse=True)
>>> numbers
[4.0, 3, 2, 1]
```

*Key функція*

І метод list.sort(), і функція sorted() мають необов'язковий параметр *key* для указання функції (або lambda), яка буде викликана для кожного елемента списку перед проведенням порівнянь. Ця функція приймає один аргумент і повертає ключ для використання в цілях сортування. Наприклад:

```
def takeSecond(el):
    return el[1]
my_list = [(2, 'Mary'), (3, 'Anna'), (4, 'Alex'), (1, 'Vika')]
my_list.sort(key=takeSecond)
print('Sorted list:', my_list)
```

На виході отримаємо:

```
Sorted list: [(4, 'Alex'), (3, 'Anna'), (2, 'Mary'), (1, 'Vika')]
```

Розглянемо ще один приклад:

```
students = [('john','A',15),('jane','B',12),('dave','B',10)]
# sort by age
sorted_list=sorted(students, key=lambda student: student[2])
print(sorted_list)
```

На виході отримаємо:

```
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

*Отримання довжини за допомогою len ()*

len () повертає кількість елементів у списку:

```
>>> marxes = ['Vika', 'Nika', 'Mary', 'Anna', 'Alex']
>>> len(marxes)
5
```

*Присвоєння за допомогою =*

Коли ви призначаєте один список для кількох змінних, зміна списку в одному місці також змінює його в іншому, як показано тут:

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = 'surprise'
>>> a
['surprise', 2, 3]
```

То що зараз у b? Це все ще [1, 2, 3] або ['surprise', 2, 3]? Подивимось:

```
>>> b
['surprise', 2, 3]
```

**Копіювання списків**

*Копіювання за допомогою copy(), list() або Slice*

Ви можете скопіювати значення списку в незалежний, свіжий список, використовуючи будь-який із цих методів:

- Метод списку copy ()
- Функцію перетворення list ()
- Зріз списку [:]

Наш оригінальний список буде a. Копії b, c, d:

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
```

*Копіювати все за допомогою deepcopy ()*

Функція copy() добре працює, якщо значення списку незмінні. Як ви бачили раніше, змінні значення (наприклад, списки) - це посилання. Зміна оригіналу або копії відображатиметься в обох. Давайте скористаємося попереднім прикладом, але зробимо останній елемент у списку списком [8, 9] замість цілого числа 3:

```
>>> a = [1, 2, [8, 9]]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
>>> a
[1, 2, [8, 9]]
>>> b
```

```
[1, 2, [8, 9]]
>>> c
[1, 2, [8, 9]]
>>> d
[1, 2, [8, 9]]
```

Все йде нормально. Тепер змініть елемент у цьому підсписку a:

```
>>> a[2][1] = 10
>>> a
[1, 2, [8, 10]]
>>> b
[1, 2, [8, 10]]
>>> c
[1, 2, [8, 10]]
>>> d
[1, 2, [8, 10]]
```

Значення [2] тепер є списком, і його елементи можна змінювати. Усі методи копіювання списку, які ми використовували, були неглибокими (поверхневими, shallow) .

Щоб виправити це, нам потрібно використовувати функцію deepcopy ():

```
>>> import copy
>>> a = [1, 2, [8, 9]]
>>> b = copy.deepcopy(a)
>>> a
[1, 2, [8, 9]]
>>> b
[1, 2, [8, 9]]
>>> a[2][1] = 10
>>> a
[1, 2, [8, 10]]
>>> b
[1, 2, [8, 9]]
```

deepcopy () може обробляти глибоко вкладені списки, словники та інші об’єкти.

*Ітерація за допомогою for – in*

```
>>> cheeses = ['brie', 'gjetost', 'havarti']
>>> for cheese in cheeses:
... if cheese.startswith('g'):
... print("I won't eat anything that starts with 'g'")
... break
... else:
... print(cheese)
...
brie
I won't eat anything that starts with 'g'
```

**Генератори списків**

***Створення списку за допомогою генератора списку (List comprehensions)***

Python підтримує концепцію під назвою "list comprehensions" ("включення списку"), яка може допомогти вам створити списки дуже легко. Ось кілька прикладів:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [[x,x**2] for x in vec if x > 3] #x**2 in Python is x*x
[[4, 16], [6, 36]]
>>> [[0]*3 for i in range(3)]
[[0,0,0],[0,0,0],[0,0,0]]
```

Таку техніку створення списку також називають генератором списку. Розглянемо ще один приклад.

```
>>> number_list = [number for number in range(1,6)]
>>> number_list
[1, 2, 3, 4, 5]
```

У першому рядку перша змінна number потрібна для створення значень для списку: тобто для розміщення результату циклу у number_list. Друге number є частиною циклу for. Щоб показати, що перше number є виразом, спробуйте такий варіант:

```
>>> number_list = [number-1 for number in range(1,6)]
>>> number_list
[0, 1, 2, 3, 4]
```

Генератор списку може включати умовний вираз, який виглядає приблизно так:

```
[expression for item in iterable if condition]
```

Давайте зробимо новий генератор, який будує список лише непарних чисел між 1 і 5 (пам’ятайте, що число % 2 є істинним для непарних чисел та хибним для парних чисел):

```
>>> a_list = [number for number in range(1,6) if number % 2
== 1]
>>> a_list
[1, 3, 5]
```

**Список списків**

Для роботи з багатомірними масивами вам знадобляться списки списків.

```
>>> grid = [[1,2,3], [4,5,6], [7,8,9]]
>>> grid
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> grid[0][1]
2
>>> grid[2]
[7,8,9] # row 2 is itself a list within the 2D grid list
```

**Питання для самоконтролю**

1. Як додати новий елемент до списку?
2. Чи може одне і те ж значення зустрічатися у списку кілька разів?
3. Чи можна індексувати список, використовуючи від'ємні індекси?
4. Які ви знаєте способи реверсу списку?
5. Як об'єднати списки?
6. Як працює метод pop() для списку?
7. Як перевірити наявність значення в списку?
8. Що таке list comprehensions для списку?

**Тестові завдання для перевірки знань**

1. Як отримати кількість входжень значення в список a?
A. find()
B. a.count(…)
C. len(a)
D. a.len

2. Як видалити елемент з початку списку a?
A. del a(1)
B. a.pop(0)
C. a.remove()
D. remove(0)

## Тема 5 Словники. Множини

**Мета:** розглянути типи даних словники (Dictionaries) і множини (Sets) в мові Python; з'ясувати сутність внутрішнього устрою цих типів; ознайомитись із особливостями застосування основних методів і глобальних функцій Python для словників і множин.

**План**

1. Словники (Dictionaries). Створення словника.
2. Отримання значення зі словника. Додавання або зміна елемента.
3. Поєднання словників. Видалення елементів.
4. Перевірка наявності ключа. Копіювання словника.
5. Генератор словника.
6. Множини (Sets). Створення множини.
7. Додавання та видалення елементів.
8. Перетин, об'єднання, різниця множин. Генератор множин.

## Основні терміни і поняття

*Словник. Зсув (offset). Ключ (key). Значення (value). Пара ключ-значення. Виняток. Генератор словника. Множина. Перетин, об'єднання, різниця множин. Генератор множин.*

## Основні теоретичні положення

### Словники (Dictionaries). Створення словника

Словник схожий на список, але порядок елементів не має значення, і вони не вибираються зсувом (offset), таким як 0 або 1. Натомість ви вказуєте унікальний ключ, який потрібно пов'язати з кожним значенням. Цей ключ часто є рядком, але насправді це може бути будь-який із незмінних типів Python: boolean, integer, float, tuple, string та інші. Словники змінюються, тому ви можете додавати, видаляти та змінювати їх елементи ключ-значення.

***Створення за допомогою {}***

Щоб створити словник, потрібно поставити фігурні дужки ({}) навколо пар розділених комами ключ: значення. Найпростіший словник - це порожній, який взагалі не містить ключів або значень:

```
>>> empty_dict = {}
>>> empty_dict
{}
```

А ось приклад словника телефонних номерів:

```
phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
```

***Створення за допомогою dict()***

```
>>> customer = dict(first="Wile", middle="E", last="Coyote")
>>> customer
{'first': 'Wile', 'middle': 'E', 'last': 'Coyote'}
```

***Перетворення за допомогою dict()***

Ви також можете використовувати функцію dict() для перетворення двозначних послідовностей у словник. Ви можете інколи стикатися з такими послідовностями "ключ-значення", як-от "Strontium, 90, Carbon, 14." Перший елемент у кожній послідовності використовується як ключ, а другий як значення.

Ось невеликий приклад використання lol (список списків із двох пунктів):

```
>>> lol = [ ['a', 'b'], ['c', 'd'], ['e', 'f'] ]
>>> dict(lol)
{'a': 'b', 'c': 'd', 'e': 'f'}
```

### Отримання значення зі словника. Додавання або зміна елемента

***Додавання або зміна елемента за допомогою [key]***

Додати елемент до словника легко. Просто зверніться до елемента за його ключем і призначте значення. Якщо ключ вже був у словнику, існуюче значення замінюється новим. Якщо ключ новий, його значення додається до словника.

На відміну від списків, вам не потрібно турбуватися про те, що Python під час призначення призначатиме виняток, вказавши індекс, який виходить за межі діапазону.

```
>>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
>>> phonebook['Joe'] = '555-0123'
>>> phonebook['Chris'] = '555-4444'
>>> phonebook
{'Chris': '555-4444', 'Joanne': '555-3333', 'Joe': '555-0123', 'Katie': '555-2222'}
>>>
```

***Отримання елемента за допомогою [key] або за допомогою get()***

Це найпоширеніше використання словника. Ви вказуєте словник і ключ, щоб отримати відповідне значення: Використаємо `phonebook` з попереднього прикладу:

>>>phonebook[‘Chris’]

‘555-4444’

Можна також використати метод get() словника.

>>> phonebook.get([‘Chris’])

‘555-4444’

***Отримання елемента за ключем і видалення його за допомогою pop ()***

Це поєднання get () та del. Якщо ви надаєте pop () ключ і він існує у словнику, він повертає відповідне значення і видаляє пару ключ-значення. Якщо його немає, він створює виняток:

```
>>> first = {'a': 'agony', 'b': 'bliss', ‘c’: ‘candy’}
>>> len(first)
3
>>> first.pop('b')
'bliss'
>>> len(first)
2
```

***Отримання всіх ключів за допомогою keys()***

Ви можете використовувати keys(), щоб отримати всі ключі у словнику. Для наступних кількох прикладів ми будемо використовувати такий зразок словника:

```
>>> signals = {'green': 'go', 'yellow': 'go faster', 'red': 'smile for the camera'}
>>> signals.keys()
dict_keys(['green', 'yellow', 'red'])
```

У Python 2 key() просто повертає список. Python 3 повертає dict_keys(), який є ітерабельним переглядом ключів.

Але часто вам справді потрібен список. У Python 3 вам потрібно викликати list(), щоб перетворити об’єкт dict_keys у список.

```
>>> list( signals.keys() )
['green', 'yellow', 'red']
```

***Отримання всіх значень за допомогою values()***

Щоб отримати всі значення у словнику, використовуйте values ():

```
>>> list( signals.values() )
['go', 'go faster', 'smile for the camera']
```

***Отримання всіх пар ключ-значення за допомогою items()***

Якщо ви хочете отримати всі пари ключ-значення зі словника, використовуйте метод items ():

```
>>> list( signals.items() )
[('green', 'go'), ('yellow', 'go faster'), ('red', 'smile for the camera')]
```

Кожен ключ і значення повертаються у вигляді кортежа, наприклад ('green', 'go').

***Отримання довжини за допомогою len ()***

Порахуйте свої пари ключ-значення:

```
>>> len(signals)
3
```

**Поєднання словників. Видалення елементів.**

***Поєднання словників за допомогою {** a, ** b}***

Починаючи з Python 3.5, є новий спосіб об'єднання словників за допомогою **

```
>>> first = {'a': 'agony', 'b': 'bliss'}
>>> second = {'b': 'bagels', 'c': 'candy'}
>>> {**first, **second}
{'a': 'agony', 'b': 'bagels', 'c': 'candy'}
```

***Поєднання словників за допомогою update()***

Ви можете скористатися функцією update(), щоб скопіювати ключі та значення одного словника в інший.

```
>>> first = {'a': 'agony', 'b': 'bliss'}
>>> second = {'b': 'bagels', 'c': 'candy'}
>>> first.update(second)
>>> first
{'a': 'agony', 'b': 'bagels', 'c': 'candy'}
```

***Видалити елемент з заданим ключем за допомогою del***

```
>>> del first['b']
>>> first
{'a': 'agony', 'c': 'candy'}
```

***Видалити всі елементи з clear ()***

Щоб видалити всі ключі та значення зі словника, використовуйте clear () або просто призначте порожній словник ({}) імені:

```
>>> first.clear()
>>> first
{}
>>> first = {}
>>> first
{}
```

**Перевірка наявності ключа. Копіювання словника.**

***Перевірка наявності ключа з in***

Якщо ви хочете дізнатися, чи існує ключ у словнику, скористайтеся in.

```
>>> first = {'a': 'agony', 'b': 'bliss', 'c': 'candy'}
>>> 'a' in first
True
```

***Призначення за допомогою =***

Як і у випадку зі списками, якщо ви внесете зміни до словника, це відобразиться у всіх іменах, які посилаються на нього:

```
>>> signals = {'green': 'go',
... 'yellow': 'go faster',
... 'red': 'smile for the camera'}
>>> save_signals = signals
>>> signals['blue'] = 'confuse everyone'
>>> save_signals
{'green': 'go',
'yellow': 'go faster',
'red': 'smile for the camera',
'blue': 'confuse everyone'}
```

***Копіювання за допомогою copy ()***

Щоб насправді скопіювати ключі та значення зі словника в інший словник, ви можете скористатися copy ():

```
>>> signals = {'green': 'go',
... 'yellow': 'go faster',
... 'red': 'smile for the camera'}
>>> original_signals = signals.copy()
>>> signals['blue'] = 'confuse everyone'
>>> signals
{'green': 'go',
'yellow': 'go faster',
'red': 'smile for the camera',
'blue': 'confuse everyone'}
>>> original_signals
{'green': 'go',
'yellow': 'go faster',
'red': 'smile for the camera'}
>>>
```

***Ітерація за допомогою for – in***

Ітерація над словником (або його функцією keys ()) повертає ключі. У цьому прикладі ключі - це типи карт у настільній грі Clue (Cluedo outside of North America):

```
>>> accusation = {'room': 'ballroom', 'weapon': 'lead pipe',
... 'person': 'Col. Mustard'}
>>>  for card in accusation:  # or, for card in
accusation.keys():
...     print(card)
...
room
weapon
person
```

Щоб перебирати значення, а не ключі, використовуйте метод values() словника:

```
>>> for value in accusation.values():
```

```
...     print(value)
...
ballroom
lead pipe
Col. Mustard
```

Щоб повернути ключ і значення як кортеж, можна скористатися функцією items ():

```
>>> for item in accusation.items():
...     print(item)
...
('room', 'ballroom')
('weapon', 'lead pipe')
('person', 'Col. Mustard')
```

**Генератор словника**

Найпростіша форма генератора словника наступна:

```
{key_expression : value_expression for expression in iterable}
```

Наприклад

```
>>> word = 'letters'
>>> letter_counts = {letter: word.count(letter) for letter in word}
>>> letter_counts
{'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1}
```

Інша форма генератора словника:

```
{key_expression : value_expression for expression in iterable if condition}
```

Наприклад:

```
>>> vowels = 'aeiou'
>>> word = 'onomatopoeia'
>>> vowel_counts = {letter: word.count(letter) for letter in set(word)
if letter in vowels}
>>> vowel_counts
{'e': 1, 'i': 1, 'o': 4, 'a': 2}
```

**Множини (Sets). Створення множини**

Тип даних set використовується, коли вам потрібно позначити комбінацію значень як єдину сутність. Він не дозволяє вставляти повторювані значення, але дозволяє змінювати дані, неоднорідні об’єкти та множини, що мають можливість розширення, тобто ви можете збільшити або зменшити розмір на основі ваших вимог.

Оскільки множини є невпорядкованими, операції індексування та зрізу для множин не надаються. Python підтримує membership (is), розмір (len ()) і looping on membership (for i in set).

***Створення за допомогою set ()***

Щоб створити множину, можна використати функцію set () або вкласти одне або кілька значень, розділених комами, у фігурні дужки, як показано тут:

```
>>> empty_set = set()
>>> empty_set
```

```
set()
>>> even_numbers = {0, 2, 4, 6, 8}
>>> even_numbers
{0, 2, 4, 6, 8}
>>> odd_numbers = {1, 3, 5, 7, 9}
>>> odd_numbers
{1, 3, 5, 7, 9}
```

***Конвертування за допомогою set ()***

Ви можете створити множину зі списку, рядка, кортежу або словника, відкинувши будь-які повторювані значення.

По-перше, давайте поглянемо на рядок з більш ніж одним входженням деяких букв:

```
>>> set( 'letters' )
{'l', 'r', 's', 't', 'e'}
```

Зверніть увагу, що множина містить лише одне "e" або "t", хоча "letters" містять по два екземпляра кожного.

Тепер давайте зробимо множину зі списку:

```
>>> set( ['Dasher', 'Dancer', 'Prancer', 'Mason-Dixon'] )
{'Dancer', 'Dasher', 'Mason-Dixon', 'Prancer'}
```

Давайте кортеж перетворимо в множину:

```
>>> set( ('Ummagumma', 'Echoes', 'Atom Heart Mother') )
{'Ummagumma', 'Atom Heart Mother', 'Echoes'}
```

Коли ви застосовуєте set () до словника, то відбирають лише ключі:

```
>>> set( {'apple': 'red', 'orange': 'orange', 'cherry': 'red'} )
{'cherry', 'orange', 'apple'}
```

***Отримання довжини за допомогою len ()***

```
>>> reindeer = set(['Dasher', 'Dancer', 'Prancer', 'Mason-Dixon'])
>>> len(reindeer)
4
```

**Додавання та видалення елементів.**

***Додавання елемента за допомогою add ()***

```
>>> s = set((1,2,3))
>>> s
{1, 2, 3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
```

***Видалиння елемента за допомогою remove ()***

```
>>> s = set((1,2,3))
>>> s.remove(3)
>>> s
{1, 2}
```

***Ітерація за допомогою for - in***

```
>>> furniture = set(('sofa', 'ottoman', 'table'))
>>> for piece in furniture:
...     print(piece)
...
```

```
ottoman
table
sofa
```

### Перетин, об'єднання, різниця множин. Генератор множин

***Перетин множин***

```
>>> a = {1, 2}
>>> b = {2, 3}
>>> a & b
{2}
>>> a.intersection(b)
{2}
```

***Об'єднання множин***

```
>>> a | b
{1, 2, 3}
>>> a.union(b)
{1, 2, 3}
```

***Різниця множин***

Різниця (члени першого набору, але не другого) отримується за допомогою символу - або функції difference ():

```
>>> a - b
{1}
>>> a.difference(b)
{1}
```

***Генератор множин (Set Comprehensions)***

```
>>> a_set = {number for number in range(1,6) if number % 3 == 1}
>>> a_set
{1, 4}
```

### Питання для самоконтролю

1. Сформулюйте визначення словника в мові Python.
2. Які ви знаєте способи створення словника?
3. Як додати значення в словник?
4. Як працює метод pop для словника?
5. Як отримати всі ключі зі словника?
6. Як працює метод items для словника?
7. Як видаляти елементи зі словника?
8. Як виконувати ітерацію по словнику?
9. Що таке множина в Python? Як її створити?
10. Як знаходити перетин, об’єднання, різницю множин?

## Навчальне завдання №1

**Обробка послідовностей на мові Python. Рядки. Списки. Кортежі**

Вивчити особливості обробки послідовностей в мові Python, отримати практичні навички розробки програм обробки рядків списків і кортежів.

**Варіант 1.**

**Вправа 1.**

Напишіть програму, яка читає цілі числа у користувача та зберігає їх у списку. Ваша програма повинна продовжувати читати значення, поки користувач не введе 0. Потім вона повинна відображати всі значення, введені користувачем (крім 0), у порядку зростання, з одним значенням, що відображається в кожному рядку. Для сортування списку використовуйте або метод sort, або функцію sorted.

**Вправа 2.**

Напишіть програму Python, щоб перетворити рядок s у нижній регістр, відсортувати символи кожного слова в алфавітному порядку та надрукувати отриманий рядок. Ви можете припустити, що рядок містить лише літери та пробіли для розділення слів. Пробіли повинні бути збережені в кінцевому рядку.

Таблиця 5. Приклад введення-виведення

| String | Output |
|---|---|
| "Hello World" | "ehllo dlorw" |
| "Wonderful World" | "deflnoruw dlorw" |

Підказка.

У мові Python великі літери йдуть перед малими літерами в алфавітному порядку. Можна використати методи split, lower, join, rstrip для рядка і функцію sorted.

**Вправа 3.**

Відсортуйте кортеж кортежів за 2-м елементом:
tuple1 = (('a', 23), ('b', 37), ('c', 11), ('d',29))
Очікуваний результат:
(('c', 11), ('a', 23), ('d', 29), ('b', 37))

Підказка.

Можна використати функцію sorted.

**Варіант 2.**

**Вправа 1.**

Напишіть програму, яка читає слова від користувача, поки користувач не введе порожній рядок. Після того, як користувач введе порожній рядок, ваша програма повинна відображати кожне слово, введене користувачем, рівно один раз. Слова повинні відображатись у тому самому порядку, в якому вони були введені вперше. Наприклад, якщо користувач вводить:

first
second
first
third
second

тоді ваша програма повинна відображати:

first
second
third

**Вправа 2.**

Напишіть програму Python для друку зазначеного списку після видалення 0-го, 4-го та 5-го елементів.
Зразок: ['Red', 'Green', 'White', 'Black', 'Pink', 'Yellow']
Очікуваний результат: ['Green', 'White', 'Black']

Підказка.
Можна використати генератор списків і функцію enumerate.

**Вправа 3.**

Перевірте, чи всі елементи в заданому кортежі однакові.

Підказка.

Можна використати функцію all.

**Варіант 3.**

**Вправа 1.**

Напишіть програму, яка буде запитувати у користувача числа, поки він не пропустить введення. Спочатку на екран слід вивести середнє значення введеного ряду чисел, після цього один за одним необхідно вивести список чисел

нижче середнього, рівних йому (якщо такі знайдуться) і вище середнього. Кожному списку повинен передувати відповідний заголовок.

**Вправа 2.**

Напишіть програму на Python для підрахунку кількості повторюваних символів у рядку s. Програма повинна надрукувати загальну кількість повторюваних символів і в наступному рядку - повторювані символи, розділені пробілом і відсортовані за алфавітом. Якщо в рядку немає повторюваних символів, надрукуйте 0 як загальну кількість і в наступному рядку - None.

Таблиця 6. Приклад введення-виведення

| String | Output |
|---|---|
| "Hello" | 1 |
| | "l" |
| "Corporation" | 2 |
| | o r |
| "Python" | 0 |
| | None |

Підказка.

Можна використати метод count для рядка і функцію sorted. Повторювані символи можна зберігати у списку. Але будьте обережні, щоб не додавати повторювані символи більше одного разу.

**Вправа 3.**

Скопіюйте елементи 44 і 55 з наступного кортежу в новий кортеж:
tuple1 = (11, 22, 33, 44, 55, 66)

**Методичні вказівки до виконання Завдання №1**

Приклади розв’язування задач

**Приклад 1.** Обчислити податок (5%) і суму чайових (18%) в ресторані на суму рахунку.

```
TAX_RATE = 0.05
TIP_RATE = 0.18

cost = float(input("Введіть суму рахунку "))

tax = cost * TAX_RATE
tip = cost * TIP_RATE
total = cost + tax + tip

print("Податок склав %.2f, чайові – %.2f, загальна сума", \
"замовлення: %.2f" % (tax, tip, total))
```

**Приклад 2.** Напишіть програму, яка запитує у користувача цілі числа, поки він не залишить рядок введення порожній. Після закінчення введення на екран повинні бути виведені спочатку всі від'ємні числа, які були введені, потім нульові і тільки після цього додатні. Усередині кожної групи числа повинні відображатися в тій послідовності, в якій були введені користувачем. Наприклад, якщо він ввів такі числа: 3, -4, 1, 0, -1, 0 і -2, виведення повинне бути таким: -4, -1, -2, 0, 0, 3 і 1. Кожне значення має відображатися на новому рядку.

```
negatives = []
zeros = []
positives = []
line = input("Enter an integer (blank to quit): ")
while line != "":
    num = int(line)
    if num < 0:
        negatives.append(num)
    elif num > 0:
        positives.append(num)
    else:
        zeros.append(num)
    line = input("Enter an integer (blank to quit): ")
print("The numbers were: ")
for n in negatives:
    print(n)
for n in zeros:
    print(n)
for n in positives:
    print(n)
```

**Приклад 3.** Введення списку з клавіатури.

```
# Блок введення списку з клавіатури
N = int(input("Кількість елементів списку "))
l = []            # Створюємо порожній список
for i in range(N):
    x = float(input("Задайте %d-й елемент списку " % (i)))
    l.append(x)   # Додаємо елемент x у список
# ================================
# Виведення списку на екран
print(l)
```

**Приклад 4.** Перевірити, чи є рядок симетричним.

```
S = input("Введіть рядок ")

if S == S[::-1]:
    print ("Заданий рядок є симетричним")
else:
    print ("Заданий рядок НЕ є симетричним")
```

**Приклад 5.** Перевірити, чи правильно в заданому виразі розставлені круглі дужки (тобто, чи стоїть справа від кожної відкритої дужки відповідна до неї закрита дужка, а зліва від кожної закритої – відповідна до неї відкрита).

```
expresion = input("Введіть вираз ")
k = 0    # різниця кількості відкритих і закритих дужок
r = True # закритій дужці відповідає відкрита
for c in expresion:
    if c == '(':
        k += 1
    if c == ')':
        k -= 1
    if k < 0: # закритій дужці не відповідає відкрита
        r = False
        break

if (k == 0 and r):
    print("Дужки розставлені правильно")
else:
    print("Дужки розставлені неправильно")
```

**Приклад 6.** Сортувати заданий список за спаданням.

```
vowels = ['e', 'a', 'u', 'o', 'i']
vowels.sort(reverse=True)
print('Sorted list (in Descending):', vowels)
```

**Приклад 7.** Сортувати список кортежів по другому параметру.

```
# take second element for sort
def takeSecond(elem):
    return elem[1]

random = [(2, 2), (3, 4), (4, 1), (1, 3)]  # random list

random.sort(key=takeSecond)  # sort list with key
print('Sorted list:', random)

Output
Sorted list: [(4, 1), (2, 2), (1, 3), (3, 4)]
```

**Приклад 8.** Вилучити елемент із зазначеним індексом зі списку.

```
languages = ['Python', 'Java', 'C++', 'French', 'C']

# remove and return the 4th item
return_value = languages.pop(3)

print('Return Value:', return_value)
print('Updated List:', languages)

Output
Return Value: French
Updated List: ['Python', 'Java', 'C++', 'C']
```

 **Тестові завдання для перевірки знань**

1. Цей метод списку додає елемент у кінець існуючого списку.
A. add
B. add_to
C. increase
D. append

2. Яке з наведених нижче тверджень створює кортеж?
A. values = [1, 2, 3, 4]
B. values = {1, 2, 3, 4}
C. values = (1)
D. values = (1,)

3. Ви можете використовувати оператор _____, щоб визначити, чи існує ключ у словнику.
A. &
B. in
C. ^
D. ?

4. Ви використовуєте _____, щоб видалити елемент зі словника.
A. remove метод
B. erase метод
C. delete метод
D. del твердження

## РОЗДІЛ 3 ФУНКЦІЇ, МОДУЛІ, ПАКЕТИ. ВИНЯТКИ. РОБОТА З ФАЙЛАМИ

### Тема 6. Функції. Ітератори. Генератори. Декоратори

**Мета:** розглянути основи роботи з функціями як одною з базових програмних структур Python; з’ясувати різницю між позиційними аргументами та аргументами ключових слів; ознайомитись з внутрішніми функціями, закриттями (closures), анонімними функціями (lambda); вивчити деякі стандартні функції Python.

**План**

1. Функції. Визначення функції. Аргументи та параметри.
2. Внутрішні функції. Закриття (Closures). Анонімні функції: lambda.

3. Ітератори та ітераційні об’єкти (iterables). Генератори. Декоратори.
4. Простір імен та область застосування.
5. Деякі стандартні функції Python.

## Основні терміни і поняття

*Code Reuse. Функція. Аргументи та параметри. Позиційні аргументи. Аргументи ключових слів. Закриття (Closures). Лямбда-функція. Ітератори. Функції-генератори. Декоратори. Простір імен.*

## Основні теоретичні положення

### Функції. Визначення функції. Аргументи та параметри

Першим кроком для повторного використання коду (code reuse) є функція: іменований фрагмент коду, відокремлений від усіх інших. Функція може приймати будь-яке число та тип вхідних параметрів та повертати будь-яке число та тип результатів виводу.

Ви можете зробити дві речі з функцією:

• Визначити її з нулем або більше параметрів.

• Викликати її і отримати нуль або більше результатів.

#### *Визначення функції за допомогою def*

Щоб визначити функцію Python, ви вводите def, ім'я функції, дужки, що містять будь-які *параметри* введення до функції, а потім, нарешті, двокрапку (:). Імена функцій мають ті ж правила, що і імена змінних (вони повинні починатися з літери або _ і містити лише букви, цифри або _).

Ось найпростіша функція Python:

```
>>> def do_nothing():
...     pass
```

#### *Виклик функції*

Ви викликаєте цю функцію, просто ввівши її ім’я та дужки.

```
>>> do_nothing()
>>>
```

#### *Аргументи та параметри*

Значення, які ви передаєте функції під час її виклику, відомі як *аргументи*. Коли ви викликаєте функцію з аргументами, значення цих аргументів копіюються у відповідні параметри всередині функції.

```
def total(a, b)  # function accepting prameters
    result = a + b
    print("Sum of ", a, " and ", b, " = ", result)

a = int(input("Enter the first number : ")
b = int(input("Enter the second number : ")
total(a, b)  # function call with two arguments

OUTPUT
```

```
Enter the first number : 10
Enter the second number : 20
Sum of 10 and 20 = 30
```

***Позиційні аргументи***

Python обробляє аргументи функцій дуже гнучко, якщо порівнювати з багатьма мовами. Найбільш відомі типи аргументів - позиційні аргументи, значення яких копіюються у відповідні параметри за порядком.

Ця функція будує словник з його позиційних вхідних аргументів і повертає його:

```
>>> def menu(wine, entree, dessert):
... return {'wine': wine, 'entree': entree, 'dessert': dessert}
...
>>> menu('chardonnay', 'chicken', 'cake')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'cake'}
```

Хоча це дуже поширене явище, мінус позиційних аргументів полягає в тому, що вам потрібно пам’ятати значення кожної позиції. Якби ми забули і викликали menu() з вином як останнім аргументом замість першого, страва була б зовсім іншою:

```
>>> menu('beef', 'bagel', 'bordeaux')
{'wine': 'beef', 'entree': 'bagel', 'dessert': 'bordeaux'}
```

***Аргументи ключових слів***

Щоб уникнути плутанини позиційних аргументів, можна вказати аргументи за іменами відповідних параметрів навіть у порядку, відмінному від їх визначення у функції:

```
>>> menu(entree='beef', dessert='bagel', wine='bordeaux')
{'wine': 'bordeaux', 'entree': 'beef', 'dessert': 'bagel'}
```

Ви можете змішувати позиційні та ключові аргументи. Давайте спочатку вкажемо вино, але використовуємо ключові аргументи для страви та десерту:

```
>>> menu('frontenac', dessert='flan', entree='fish')
{'wine': 'frontenac', 'entree': 'fish', 'dessert': 'flan'}
```

Якщо ви викликаєте функцію з позиційними та ключовими аргументами, позиційні аргументи мають бути на першому місці.

***Вказування значення параметрів за замовчуванням***

Ви можете вказати значення за замовчуванням для параметрів. Це значення використовується, якщо абонент не надає відповідний аргумент.

```
>>> def menu(wine, entree, dessert='pudding'):
...       return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

Цього разу спробуйте викликати menu() без аргументу десерту:

```
>>> menu('chardonnay', 'chicken')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'pudding'}
```

Якщо ви надаєте аргумент, він використовується замість типового:

```
>>> menu('dunkelfelder', 'duck', 'doughnut')
```

```
{'wine': 'dunkelfelder', 'entree': 'duck', 'dessert': 'dough-
nut'}
```

***Розгорнути/зібрати позиційні аргументи за допомогою ****

Якщо ви програмували на C або C ++, можна припустити, що зірочка (*) у програмі Python має щось спільне з покажчиком. Ні, у Python немає вказівників. При використанні всередині функції з параметром зірочка *групує змінну кількість позиційних аргументів в єдиний кортеж* значень параметрів. У наступному прикладі args - це кортеж параметрів, який є результатом нульових або більше аргументів, які були передані функції print_args ():

```
>>> def print_args(*args):
...     print('Positional tuple:', args)
...
```

Якщо ви викликаєте функцію без аргументів, ви не отримаєте нічого в *args:

```
>>> print_args()
Positional tuple: ()
```

Все, що ви задасте, буде надруковано як кортеж args:

```
>>> print_args(3, 2, 1, 'wait!', 'uh...')
Positional tuple: (3, 2, 1, 'wait!', 'uh...')
```

Це корисно для написання таких функцій, як print (), які приймають змінну кількість аргументів. Якщо ваша функція також потребує позиційних аргументів, поставте їх на перше місце; *args йде в кінці і захоплює все інше.

***Розгорнути/зібрати аргументи ключових слів за допомогою *****

Ви можете використовувати дві зірочки (**) для групування аргументів ключових слів у словнику, де назви аргументів - це ключі, а їх значення - відповідні значення словника. Наступний приклад визначає функцію print_kwargs () для друку аргументів ключових слів:

```
>>> def print_kwargs(**kwargs):
... print('Keyword arguments:', kwargs)
...
```

Тепер спробуйте викликати її з деякими ключовими аргументами:

```
>>> print_kwargs()
Keyword arguments: {}
>>>    print_kwargs(wine='merlot',    entree='mutton',    des-
sert='macaroon')
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot',
'entree': 'mutton'}
```

Усередині функції kwargs - це параметр словник.

***Keyword-Only аргументи***

Можна передати аргумент ключового слова з такою ж назвою, що і позиційний параметр, ймовірно, це не призведе до того, що вам потрібно. Python 3 дозволяє вказувати лише ключові аргументи. Як випливає з назви, вони повинні надаватися як name=value, а не позиційно як значення. Єдиний * у визначенні функції означає, що наступні параметри start та end повинні бути надані як іменовані аргументи, якщо ми не хочемо їх значень за замовчуванням:

```
>>> def print_data(data, *, start=0, end=100):
```

```
...     for value in (data[start:end]):
...         print(value)
...
>>> data = ['a', 'b', 'c', 'd', 'e', 'f']
>>> print_data(data)
a
b
c
d
e
f
>>> print_data(data, start=4)
e
f
>>> print_data(data, end=2)
a
b
```

***Змінні та незмінні аргументи***

Пам'ятайте, що якщо ви призначили один і той самий список двом змінним, ви могли б змінити його за допомогою будь-якої з них. І що ви це зробити не можете, якщо обидві змінні посилаються на щось на кшталт цілого чи рядка. Це є тому, що список був змінним, а ціле число та рядок незмінними.

Ви повинні стежити за такою ж поведінкою при передачі аргументів функціям. Якщо аргумент змінюється, його значення можна змінити всередині функції за допомогою відповідного параметра:

```
>>> outside = ['one', 'fine', 'day']
>>> def mangle(arg):
...     arg[1] = 'terrible!'
...
>>> outside
['one', 'fine', 'day']
>>> mangle(outside)
>>> outside
['one', 'terrible!', 'day']
```

***Docstrings***

Чіткість має значення, справді каже дзен Python. До визначення функції можна додати документацію, включивши рядок на початку тіла функції. Це рядок docstring:

```
def print_if_true(thing, check):
    '''
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    '''
    if check:
        print(thing)
```

***Всі функції в Python є First-Class Citizens***

Згадаймо мантру Python: «все є об’єктом». Це включає в себе числа, рядки, кортежі, списки, словники та функції. Функції є first-class citizens у Python. Ви можете призначити їх змінним, використовувати їх як аргументи для інших функцій та повертати їх із функцій. Це дає вам можливість робити деякі речі на Python, які важко виконати багатьма іншими мовами.

Щоб перевірити це, давайте визначимо просту функцію під назвою answer (), яка не має жодних аргументів; вона просто друкує номер 42:

```
>>> def answer():
...     print(42)
```

Давайте визначимо іншу функцію під назвою run_something. У неї є один аргумент, який називається func, функція для запуску:

```
>>> def run_something(func):
...     func()
```

Якщо ми передаємо answer в run_something (), ми використовуємо функцію як дані, так само як і з будь-яким іншим типом даних:

```
>>> run_something(answer)
42
```

Зверніть увагу, що ми передали answer, а не answer(). У Python ці дужки означають виклик цієї функції. Без дужок, Python просто розглядає функцію як будь-який інший об’єкт. Це тому, що, як і все інше в Python, функція - це об’єкт:

```
>>> type(run_something)
<class 'function'>
```

**Внутрішні функції. Закриття (Closures). Анонімні функції: lambda**

***Внутрішні функції***

Ви можете визначити функцію в іншій функції:

```
>>> def outer(a, b):
...     def inner(c, d):
...         return c + d
...     return inner(a, b)
...
>>>
>>> outer(4, 7)
11
```

Внутрішня функція може бути корисною під час виконання складного завдання кілька разів у межах іншої функції, щоб уникнути циклів або дублювання коду. Для прикладу рядка ця внутрішня функція додає текст до свого аргументу:

```
>>> def knights(saying):
...     def inner(quote):
...         return "We are the knights who say: '%s'" % quote
...     return inner(saying)
...
>>> knights('Ni!')
"We are the knights who say: 'Ni!'"
```

***Закриття (Closures)***

Внутрішня функція може виконувати роль закриття. Це функція, яка динамічно генерується іншою функцією і може як змінювати, так і запам'ятовувати значення змінних, створених поза функцією.

Наступний приклад ґрунтується на попередньому прикладі knight(). Давайте назвемо нову функцію knight2 (), і перетворимо функцію inner() на закриття під назвою inner2(). Ось відмінності:

• inner2 () використовує зовнішній параметр saying безпосередньо, а не отримує його як аргумент.

• knights2 () повертає ім'я функції inner2 замість того, щоб викликати її:

```
>>> def knights2(saying):
...     def inner2():
...         return "We are the knights who say: '%s'" % saying
...     return inner2
...
```

Функція inner2() знає значення переданого saying і запам'ятовує його. Рядок return inner2 повертає цю спеціалізовану копію функції inner2 (але не викликає її). Це своєрідне закриття: динамічно створена функція, яка запам'ятовує, звідки вона прийшла.

Давайте двічі викличемо knights2() з різними аргументами:

```
>>> a = knights2('Duck')
>>> b = knights2('Hasenpfeffer')
```

Гаразд, а які типи у a і b?

```
>>> type(a)
<class 'function'>
>>> type(b)
<class 'function'>
```

Вони є функціями, але вони також є закриттями:

```
>>> a
<function knights2.<locals>.inner2 at 0x10193e158>
>>> b
<function knights2.<locals>.inner2 at 0x10193e1e0>
```

Якщо ми їх викликаємо, вони пам’ятають saying, який використовувався, коли їх створювали knights2:

```
>>> a()
"We are the knights who say: 'Duck'"
>>> b()
"We are the knights who say: 'Hasenpfeffer'"
```

***Анонімні функції: lambda***

Лямбда-функція Python - це анонімна функція, виражена як єдиний оператор. Ви можете використовувати його замість звичайної крихітної функції.

Щоб проілюструвати це, давайте спочатку зробимо приклад, який використовує звичайні функції. Для початку давайте визначимо функцію edit_story(). ЇЇ аргументи такі:

• words — список слів

• func — функція, що застосовується до кожного слова в словах.

```
>>> def edit_story(words, func):
...     for word in words:
...         print(func(word))
```

Тепер нам потрібен список слів і функція для застосування до кожного слова:

```
>>> stairs = ['thud', 'meow', 'thud', 'hiss']
```

Нехай функція кожне слово буде писати з великої літери та додаватиме знак оклику:

```
>>> def enliven(word):
...     return word.capitalize() + '!'
```

Тепер викличемо функцію edit_story():

```
>>> edit_story(stairs, enliven)
Thud!
Meow!
Thud!
Hiss!
```

Нарешті ми доходимо до лямбди. Функція enliven () була настільки короткою, що ми могли б замінити її на лямбду:

```
>>> edit_story(stairs, lambda word: word.capitalize() + '!')
Thud!
Meow!
Thud!
Hiss!
```

Lambda містить нуль або більше аргументів, розділених комами, за якими йде двокрапка (:), а потім-визначення функції. Ви не використовуєте дужки з lambda, як це було б під час виклику функції, створеної за допомогою def.

Часто використання реальних функцій, таких як enliven(), набагато зрозуміліше, ніж використання лямбд. Лямбди переважно корисні для випадків, коли в іншому випадку вам потрібно було б визначити багато крихітних функцій і запам'ятати, як ви їх усі назвали. Зокрема, ви можете використовувати лямбди в графічних інтерфейсах користувача для визначення функцій зворотного виклику.

## Ітератори та ітераційні об'єкти (iterables). Генератори. Декоратори

### *Ітератори та ітераційні об'єкти (iterables)*

Згідно з документацією Python (https://docs.python.org/3/glossary.html ), iterable є:

Об'єкт, здатний повертати своїх членів по одному. Приклади iterables включають усі типи послідовностей (наприклад, список, рядок і кортеж) і деякі типи не послідовності, як-от dict, об'єкти файлів та об'єкти будь-яких класів, які ви визначаєте за допомогою методу __iter__() або методу __getitem__(), який реалізує семантику послідовності.

Iterables можна використовувати в циклі for і в багатьох інших місцях, де потрібна послідовність (zip(), map(), ...). Коли ітераційний об'єкт передається як аргумент вбудованій функції iter(), вона повертає ітератор для об'єкта. Цей ітератор підходить для одного проходу через набір значень. При використанні ітераторів зазвичай немає необхідності викликати iter() або самостійно працювати з об'єктами ітератора. Оператор for робить це автоматично за вас, створюючи тимчасову змінну без імені, щоб утримувати ітератор протягом циклу.

Деякі структури даних під час ітерації перебирають свої елементи в певному порядку, наприклад списки, кортежи, словники та рядки, тоді як інші, наприклад множини, цього не роблять. Python дає нам можливість виконувати ітерацію над iterables, використовуючи тип об’єкта, який називається ітератором.

Згідно з офіційною документацією, ітератором є:

Об'єкт, що представляє потік даних. Повторювані виклики методу __next__() ітератора (або передача його вбудованій функції next()) повертають послідовні елементи в потоці. Якщо дані більше не доступні, замість цього створюється виняток StopIteration. На цьому етапі об’єкт ітератора вичерпано, і будь-які подальші виклики його методу __next__() просто знову викликають StopIteration. Ітератори повинні мати метод __iter__(), який повертає сам об’єкт ітератора, тому кожен ітератор також є ітераторним (iterable) і може використовуватися в більшості місць, де приймаються інші iterables.

На практиці весь механізм iterable/iterator дещо прихований за кодом. Якщо з якоїсь причини вам не потрібно запрограмувати свій власний iterable або ітератор, вам не доведеться занадто турбуватися про це. Але дуже важливо розуміти, як Python справляється з цим ключовим аспектом потоку керування, оскільки це вплине на спосіб написання коду [18].

***Генератори***

Генератор - це об’єкт Python для створення послідовності. З його допомогою можна перебирати потенційно величезні послідовності, не створюючи та не зберігаючи всю послідовність у пам’яті одночасно. Генератори часто є джерелом даних для ітераторів. Якщо ви пам'ятаєте, ми вже використовували один з них, range(), у попередніх прикладах коду для створення ряду цілих чисел. Цей приклад додає всі цілі числа від 1 до 100:

```
>>> sum(range(1, 101))
5050
```

Кожного разу, коли ви проходите ітерацію через генератор, він відстежує, де він востаннє викликався, і повертає наступне значення. Це відрізняється від звичайної функції, яка не має пам’яті про попередні виклики і завжди починається з першого рядка з тим самим станом.

***Функції-генератори (Generator Functions)***

Якщо ви хочете створити потенційно велику послідовність, ви можете написати функцію генератора. Це нормальна функція, але вона повертає своє значення через yield, а не return. Давайте напишемо власну версію range ():

```
>>> def my_range(first=0, last=10, step=1):
...     number = first
...     while number < last:
...         yield number
...         number += step
...
```

Це нормальна функція:

```
>>> my_range
<function my_range at 0x10193e268>
```

І вона повертає об’єкт генератора:

```
>>> ranger = my_range(1, 5)
>>> ranger
<generator object my_range at 0x101a0a168>
```

Ми можемо виконувати ітерацію через цей об’єкт генератора:

```
>>> for x in ranger:
...     print(x)
...
1 2 3 4
```

ПРИМІТКА. Генератор можна запускати лише один раз. Списки, множини, рядки та словники існують у пам’яті, але генератор створює свої значення на льоту і роздає їх по одному за допомогою ітератора. Він не пам’ятає їх, тому ви не можете перезапустити або створити резервну копію генератора.

***Генератор включень (Generator Comprehensions)***

Ви бачили генератори списків, словників та наборів. Генератор включень виглядає так само, але він оточений круглими дужками замість квадратних або фігурних дужок. Це як скорочена версія генераторної функції, яка робить невидиму віддачу, а також повертає об’єкт генератора:

```
>>> genobj = (pair for pair in zip(['a', 'b'], ['1', '2']))
>>> genobj
<generator object <genexpr> at 0x10308fde0>
>>> for thing in genobj:
...     print(thing)
...
('a', '1')
('b', '2')
```

***Декоратори (Decorators)***

Іноді вам потрібно змінити існуючу функцію без зміни її вихідного коду. Поширеним прикладом є додавання оператора налагодження, щоб побачити, в яких аргументах було передано.

Декоратор - це функція, яка бере одну функцію як вхідну і повертає іншу. Давайте заглибимося у вже розглянуті засоби Python і скористаємося наступним:

• *args та ** kwargs.
• Внутрішні функції.
• Функції як аргументи.

Функція document_it () визначає декоратор, який буде виконувати наступне:

• Роздрукує назву функції та значення її аргументів.
• Запустить функцію з аргументами.
• Роздрукує результат.
• Поверне змінену функцію для використання.

Ось як виглядає код:

```
>>> def document_it(func):
...     def new_function(*args, **kwargs):
```

```
...         print('Running function:', func.__name__)
...         print('Positional arguments:', args)
...         print('Keyword arguments:', kwargs)
...         result = func(*args, **kwargs)
...         print('Result:', result)
...         return result
...     return new_function
```

Яку б функцію ви не передали до document_it (), ви отримаєте нову функцію, яка включає додаткові оператори, які додає document_it (). Декоратор насправді не повинен запускати код з func, але document_it () викликає функцію посередині, щоб ви могли отримати результати функції func, а також усі додаткові функції.

Отже, як ви цим користуєтесь? Ви можете застосувати декоратор вручну:

```
>>> def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
8
>>> cooler_add_ints = document_it(add_ints) # manual decorator assignment
>>> cooler_add_ints(3, 5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

Як альтернативу ручному призначенню декоратора, який ми щойно розглянули, ви можете додати @decorator_name перед функцією, яку потрібно декорувати:

```
>>> @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Start function add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

**Простір імен та область застосування (Namespaces and Scope)**

Ім’я може позначати різні речі, залежно від того, де воно використовується. Програми Python мають різні *простори імен* - розділи, в яких певне ім’я є унікальним і не має відношення до того самого імені в інших просторах імен.

Кожна функція визначає свій власний простір імен. Якщо ви визначаєте змінну під назвою `x` у головній програмі, а іншу змінну під назвою `x` у функції, вони стосуються різних речей. Але стіни можна зламати: за потреби ви можете отримати доступ до імен в інших просторах імен різними способами.

Основна частина програми визначає *глобальний (global)* простір імен; таким чином, змінні в цьому просторі імен є глобальними (*global variables*).

Ви можете отримати значення глобальної змінної зсередини функції:

```
>>> animal = 'fruitbat'
>>> def print_global():
...     print('inside print_global:', animal)
...
>>> print('at the top level:', animal)
at the top level: fruitbat
>>> print_global()
inside print_global: fruitbat
```

Але якщо ви спробуєте отримати значення глобальної змінної та змінити її в межах функції, ви отримаєте помилку:

```
>>> def change_and_print_global():
...     print('inside change_and_print_global:', animal)
...     animal = 'wombat'
...     print('after the change:', animal)
...
>>> change_and_print_global()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in change_and_print_global
UnboundLocalError: local variable 'animal' referenced before
assignment
```

Якщо ви просто зміните значення, це змінить іншу змінну, яка також називається animal, але ця змінна знаходиться всередині функції:

```
>>> animal = 'fruitbat'
>>> def change_local():
...     animal = 'wombat'
...     print('inside change_local:', animal, id(animal))
...
>>> change_local()
inside change_local: wombat 4330406160
>>> animal
'fruitbat'
>>> id(animal)
4330390832
```

Що тут сталося? Перший рядок призначає рядок 'fruitbat' глобальній змінній з назвою animal. У функції change_local () також є змінна з назвою animal, але це в її локальному просторі імен.

Ми використали тут функцію Python id (), щоб надрукувати унікальне значення для кожного об'єкта і довести, що змінна animal всередині change_local () не те саме, що animal на основному рівні програми.

Щоб отримати доступ до глобальної змінної, а не до локальної в межах функції, вам потрібно використовувати ключове слово global:

```
>>> animal = 'fruitbat'
>>> def change_and_print_global():
...     global animal
...     animal = 'wombat'
...     print('inside change_and_print_global:', animal)
...
>>> animal
'fruitbat'
```

```
>>> change_and_print_global()
inside change_and_print_global: wombat
>>> animal
'wombat'
```

Якщо ви не використовуєте global у межах функції, Python використовує локальний простір імен, а змінна є локальною. Вона зникає після завершення функції.

Python надає дві функції для доступу до вмісту ваших просторів імен:

• local () повертає словник вмісту локального простору імен.

• globals () повертає словник вмісту глобального простору імен.

І ось як вони використовуються:

```
>>> animal = 'fruitbat'
>>> def change_local():
...     animal = 'wombat' # local variable
...     print('locals:', locals())
...
>>> animal
'fruitbat'
>>> change_local()
locals: {'animal': 'wombat'}
>>> print('globals:', globals()) # reformatted a little for
presentation
globals: {'animal': 'fruitbat',
'__doc__': None,
'change_local': <function change_local at 0x1006c0170>,
'__package__': None,
'__name__': '__main__',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__builtins__': <module 'builtins'>}
>>> animal
'fruitbat'
```

Локальний простір імен у change_local () містив лише локальну змінну animal. Глобальний простір імен містив окрему глобальну змінну animal та ряд інших речей.

**Деякі стандартні функції Python**

Інтерпретатор Python має ряд вбудованих функцій і типів, які завжди доступні. Вони перераховані тут (Таблиця 7) в алфавітному порядку (https://docs.python.org/3/library/functions.html ).

Таблиця 7. Вбудовані функції Python

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

*Функція enumerate()*

```
enumerate(iterable, start=0)
```

Повертає об'єкт перерахування. iterable має бути послідовністю, ітератором або іншим об’єктом, що підтримує ітерацію. Метод __next __ () ітератора, який повертає enumerate (), повертає кортеж, що містить count (від початку, який за замовчуванням до 0) та значення, отримані в результаті ітерації.

Приклад:

```
>>>seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>>list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>>list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

***Функція range()***

class range(stop)

class range(start, stop[, step])

Замість того, щоб бути функцією, range насправді є незмінним типом послідовності. Дивись https://docs.python.org/3/library/stdtypes.html#typesseq-range .

Приклади:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
```

***Функція any()***

Синтаксис: `any(iterable)`, де `iterable` - ітерабельний об'єкт (list, tuple, dictionary).

Функція any() повертає True, якщо будь-який елемент в ітерабельному об’єкті є істинним, інакше повертає False. Якщо ітерабельний об'єкт порожній, функція any() поверне False.

Приклад:

```
>>> mytuple = (0, 1, False)
>>> x = any(mytuple)
>>> print(x)
True
```

***Функція map()***

Синтаксис:

`map(function, iterable, [iterable 2, iterable 3, ...])`,

де `function` - функція, `iterable` - ітерабельний об'єкт (list, tuple, dictionary).

Повертає ітератор, який застосовує функцію до кожного елемента iterable, повертаючи результат. Якщо передаються додаткові аргументи iterable, функція повинна приймати стільки ж аргументів і застосовуватись до елементів з усіх iterable паралельно. Із кількома iterable ітератор зупиняється, коли вичерпано найкоротший iterable.

Приклад 1:

```
>>> def square(number):
     return number ** 2

>>> numbers = [1, 2, 3, 4, 5]
>>> squared = map(square, numbers)
>>> list(squared)
[1, 4, 9, 16, 25]
>>>
```

Приклад 2:

```
>>> t1 = (1, 2, 3)
>>> t2 = (5.0, 6.0, 7.0)
>>> list(map(lambda x, y : x/y, t1, t2))
[0.2, 0.3333333333333333, 0.42857142857142855]
>>>
```

***Функція zip()***

Синтаксис:

`zip(*iterables)`

Створює ітератор, який об’єднуватиме елементи з кожного з `iterables`.

Повертає ітератор кортежів, де і-й кортеж містить і-й елемент з кожної з послідовностей аргументів (`iterables`). Ітератор зупиняється, коли вичерпується найкоротший вхідний `iterables`. З одним ітерабельним аргументом він повертає ітератор 1-кортежу. Без аргументів він повертає порожній ітератор.

Приклад 1.

```
>>> countries = ["Ecuador", "Peru", "Colombia", "USA"]
>>> capitals = ["Quito", "Lima", "Bogota", "Washimgton DC"]
```

```
>>> countries_and_capitals = zip(countries, capitals)
>>> countries_and_capitals
<zip object at 0x0000021255B30F40>
>>> countries_and_capitals.__next__()
('Ecuador', 'Quito')
>>> countries_and_capitals.__next__()
('Peru', 'Lima')
>>> countries_and_capitals.__next__()
('Colombia', 'Bogota')
>>>
```

```
>>> countries_and_capitals = zip(countries, capitals)
>>> for s in countries_and_capitals:
        print(s)

('Ecuador', 'Quito')
('Peru', 'Lima')
('Colombia', 'Bogota')
('USA', 'Washimgton DC')
>>>
```

Приклад 2.

```
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = ['tri', 'dva', 'raz']
>>> lst = list(zip(lst1, lst2))
>>> lst
[(1, 'tri'), (2, 'dva'), (3, 'raz')]
>>>
```

**Питання для самоконтролю**

1. Які ви знаєте способи визначення функції в мові Python?
2. Як функції допомагають повторно використовувати код у програмі?
3. Що таке аргументи та параметри функції?
4. Яка різниця між позиційними аргументами функції і аргументами ключових слів?
5. Чи може функція в мові Python мати змінну кількість параметрів?
6. Що означає твердження «Функції є first-class citizens у Python»?
7. Що таке закриття (Closures)?
8. Яка роль анонімних функцій (lambda)?
9. Що таке ітераційні об’єкти (iterables) та ітератори?
10. Що таке генератор і як від пов’язаний з ітератором?
11. Що таке декоратори і для чого вони призначені?
12. Що таке Namespaces і Scope?

 **Тестові завдання для перевірки знань**

1. Техніка проектування, яка допомагає зменшити дублювання коду в програмі та є перевагою використання функцій, це _____.

A. code reuse
B. divide and conquer
C. debugging
D. facilitation of teamwork

2. Перший рядок визначення функції відомий як _____.

A. body
B. introduction
C. initialization
D. header

3. Ключове слово _____ ігнорується інтерпретатором Python і може використовуватися як заповнювач для коду, який буде написаний пізніше.

A. placeholder
B. pass
C. pause
D. skip

4. _____ — це змінна, яка створюється всередині функції.

A. global variable
B. local variable
C. hidden variable
D. жоден з перерахованих вище; Ви не можете створити змінну всередині функції

5. _____ — це частина програми, у якій можна отримати доступ до змінної.

A. declaration space
B. area of visibility
C. scope
D. mode

6. _____ — це частина даних, яка надсилається у функцію..

A. argument
B. parameter
C. header
D. packet

7. _____ — це спеціальна змінна, яка отримує частину даних під час виклику функції.

A. argument
B. parameter
C. header
D. packet

## Тема 7. Модулі. Стандартна бібліотека Python. Пакети

**Мета:** розглянути модулі як засоби повторного використання коду; з'ясувати суттєві характеристики стандартної бібліотеки, укомплектованої інструментами та функціями практично для будь-яких цілей; вивчити роль пакетів та інструментів управління пакетами; розглянути віртуальні середовища та методи їх створення.

**План**

1. Модулі та конструкція import.
2. Стандартна бібліотека Python.
3. Пакети (Packages).
4. Створення віртуальних середовищ

### Основні терміни і поняття

*Модуль. Імпорт модуля. Псевдонім (alias). Стандартна бібліотека. Зовнішні бібліотеки. Пакети (Packages). Віртуальне середовище.*

### Основні теоретичні положення

**Модулі та конструкція import**

Модулі виконують дві важливих функції:

- Повторне використання коду.
- Управління адресним простором імен.

Python дозволяє помістити класи, функції або дані в окремий файл і використовувати їх в інших програмах. Такий файл називається модулем.

Підключити модуль можна за допомогою інструкції import:

```
>>> import os
>>> os.getcwd()
'C:\\Python39'
>>> import math
>>> math.e
2.718281828459045
>>>
```

Ми будемо створювати та використовувати код Python у кількох файлах, створювати власні модулі. Модуль - це просто файл будь-якого коду Python. Вам не потрібно робити нічого особливого, будь-який код Python може бути використаний як модуль іншими.

Ми посилаємося на код інших модулів за допомогою оператора import Python. Це робить код та змінні в імпортованому модулі доступними для вашої програми.

***Імпортування модуля***

Найпростіший спосіб використання оператора import –

```
import module,
```

де module - це ім’я іншого файлу Python без розширення .py.

Скажімо, ви та кілька інших бажаєте чогось швидкого на обід, але не хочете довгих обговорень, і ви завжди вибираєте те, що хоче найгучніша людина. Нехай комп'ютер вирішує! Давайте напишемо модуль з однією функцією, яка повертає випадковий вибір фаст-фуду, та основну програму, яка викликає його та друкує вибір.

Модуль (fast.py) показаний у Лістингу 1.

Лістинг 1. Файл fast.py

```
from random import choice
places = ['McDonalds", "KFC", "Burger King", "Taco Bell",
"Wendys", "Arbys", "Pizza Hut"]
def pick():
    """Return random fast food place"""
    return choice(places)
```

Лістинг 2 показує основну програму, яка його імпортує (назвемо її lunch.py).

Лістинг 2. Файл lunch.py

```
import fast
place = fast.pick()
print("Let's go to", place)
```

Розмістимо ці два файли в одному каталозі та запустимо lunch.py як основну програму, вона отримає доступ до модуля `fast` та запустить функцію pick(), яка поверне випадковий результат зі списку рядків:

```
$ python lunch.py
Let's go to Burger King
$ python lunch.py
Let's go to Pizza Hut
$ python lunch.py
Let's go to Arbys
```

Ми використовували імпорт у двох різних місцях:

• Основна програма lunch.py імпортувала наш новий модуль `fast`.

• Файл модуля fast.py імпортував функцію `choice` зі стандартного бібліотечного модуля Python під назвою `random`.

Ми також використовували імпорт двома різними способами в нашій головній програмі та нашому модулі:

• У першому випадку ми імпортували весь модуль `fast`, але його потрібно було використовувати як префікс для pick(). Після цього оператора імпорту все у файлі fast.py доступне для головної програми. Ставлячи назву модуля перед функцією, ми уникаємо будь-яких неприємних конфліктів імен. У іншому модулі може бути функція pick(), і ми б не викликали її помилково.

• У другому випадку ми знаходимось у модулі і знаємо, що тут немає нічого іншого з назвою choice, тому ми імпортували функцію choice() безпосередньо з модуля random.

Ми могли б написати fast.py, як показано в Лістингу 3, імпортуючи random у функції pick() замість верхньої частини файлу.

Лістинг 3. Файл fast2.py

```
    places = ['McDonalds", "KFC", "Burger King", "Taco Bell",
"Wendys", "Arbys", "Pizza Hut"]
    def pick():
        import random
        return random.choice(places)
```

***Імпортування модуля з іншою назвою***

У нашій головній програмі lunch.py ми називали імпорт `fast`. Але що, якщо:

• Десь є інший модуль з назвою `fast`?

• Хочете використовувати більш мнемонічну назву?

У цих випадках можна імпортувати за допомогою *псевдоніма* (alias), як показано в Лістингу 4. Скористаємося псевдонімом f.

Лістинг 4. Файл fast3.py

```
import fast as f
place = f.pick()
print("Let's go to", place)
```

***Імпортуйте з модуля лише те, що вам потрібно***

Ви можете імпортувати цілий модуль або лише його частини. Ви щойно побачили останнє: нам потрібна була лише функція choice() з модуля `random`.

Як і сам модуль, ви можете використовувати псевдонім для кожного імпортованого елементу.

Повторимо lunch.py ще кілька разів. Спочатку імпортуйте pick() з модуля `fast` з його оригінальною назвою (Лістинг 5).

Лістинг 5. Файл fast4.py

```
from fast import pick
place = pick()
print("Let's go to", place)
```

Тепер імпортуйте його як who_cares (Лістинг 6).

Лістинг 6. Файл fast5.py

```
from fast import pick as who_cares
place = who_cares()
print("Let's go to", place)
```

**Стандартна бібліотека Python**

Python поставляється з величезною стандартною бібліотекою, укомплектованою інструментами та функціями практично для будь-яких цілей, які ви можете собі уявити. Новачки в Python, які звикли писати власні функції для виконання основних завдань, часто шоковані тим, що сама мова поставляється з такою кількістю функцій, вбудованих і готових до використання [10].

Коли виникає спокуса написати власну функцію для виконання простого завдання, спочатку зупиніться та перегляньте стандартну бібліотеку - може ця функція вже існує у стандартній бібліотеці.

Ми поговоримо про деякі з цих модулів, такі як functools та itertools, трохи далі, а ось декілька стандартних модулів, які вам неодмінно стануть у пригоді:

- atexit дозволяє реєструвати функції для виклику вашої програми під час її виходу.
- argparse надає функції для аналізу аргументів командного рядка.
- bisect надає алгоритми бісекції для сортування списків.
- calendar забезпечує ряд функцій, пов'язаних із датою.
- codecs надає функції для кодування та декодування даних.
- collections надає різноманітні корисні структури даних.
- copy надає функції для копіювання даних.
- csv надає функції для читання та запису файлів CSV.
- datetime надає класи для обробки дат і часу.
- fnmatch надає функції для відповідності шаблонів імен файлів Unixstyle.
- concurrent забезпечує асинхронні обчислення.
- glob надає функції для узгодження шаблонів шляхів Unixstyle.
- io надає функції для обробки потоків вводу -виводу.
- json надає функції для читання та запису даних у форматі JSON.
- logging надає доступ до власних вбудованих функцій журналювання Python.
- multiprocessing дозволяє запускати декілька підпроцесів із вашої програми, надаючи при цьому API, що робить їх схожими на потоки.
- operator надає функції, що реалізують основні оператори Python, які можна використовувати замість того, щоб писати власні лямбда - вирази.
- os надає доступ до основних функцій ОС.
- random надає функції для створення псевдовипадкових чисел.
- re забезпечує функціональність регулярних виразів.
- sched надає планувальник подій без використання багатопоточності.
- select надає доступ до функцій select () та poll () для створення циклів подій.
- shutil надає доступ до функцій файлів високого рівня.
- signal забезпечує функції для обробки сигналів POSIX.

- tempfile надає функції для створення тимчасових файлів і каталогів.
- threading надає доступ до функцій потокової роботи високого рівня.
- urllib надає функції для обробки та аналізу URL-адрес.
- uuid дозволяє створювати універсальні унікальні ідентифікатори (UUID).

Використовуйте цей список як короткий довідник щодо того, що роблять ці модулі. Якщо ви можете запам'ятати навіть частину цього списку, тим менше часу вам доведеться витрачати на пошук бібліотечних модулів, тим більше часу ви можете витратити на написання коду, який вам дійсно потрібен.

Більшість стандартної бібліотеки написано на Python, тому ніщо не заважає вам переглянути вихідний код модулів та функцій. Якщо ви сумніваєтесь, відкрийте код і подивіться, що він робить. Навіть якщо в документації є все, що вам потрібно знати, завжди є шанс дізнатися з коду щось корисне.

***Зовнішні бібліотеки***

Філософія Python з "включеними батареями" ("batteries included") полягає в тому, що після того, як ви встановите Python, у вас повинно бути все необхідне, щоб створити все, що завгодно. Це робиться для того, щоб запобігти програмуванню, еквівалентному розпакуванню приголомшливого подарунка, щоб дізнатися, що той, хто вам його подарував, забув купити для нього батарейки [10].

На жаль, люди, які стоять за Python, не можуть передбачити все, що ви хочете зробити. І навіть якби вони могли, більшість людей не хотіли б мати справу з багатогігабайтним завантаженням, особливо якщо вони просто хотіли написати швидкий сценарій для перейменування файлів. Тому навіть з її широкими функціональними можливостями стандартна бібліотека Python не охоплює всього. На щастя, члени спільноти Python створили зовнішні бібліотеки.

***Модуль sys***

Модуль sys забезпечує доступ до деяких змінних і функцій, які взаємодіють з інтерпретатором Python.

*sys.argv* - список аргументів командного рядка, що передаються сценарію Python. sys.argv [0] є іменем скрипта.

Наступна програма демонструє доступ до аргументів командного рядка. Вона починається з повідомлення про кількість аргументів командного рядка, наданих програмі, та ім'я вихідного файлу, який виконується. Потім вона продовжує роботу і відображає аргументи, які з'являються після назви вихідного файлу, якщо такі значення були надані. В іншому випадку відображається повідомлення, яке вказує на те, що не було аргументів командного рядка, окрім виконуваного файлу .py.

```
import sys

print("The program has", len(sys.argv), \
      "command line argument(s).")
print("The name of the .py file is", sys.argv[0])
```

```
if len(sys.argv) > 1:
    print("The remaining arguments are:")
    for i in range(1, len(sys.argv)):
        print(" ", sys.argv[i])
else:
    print("No additional arguments were provided.")
```

**Модуль datetime**

Стандартний модуль datetime обробляє дати та час. Він визначає чотири основні класи об'єктів, кожен з яких має безліч методів:

• *date* для років, місяців та днів
• *time* для годин, хвилин, секунд
• *datetime* для спільних дат і часу
• *timedelta* для інтервалів дати та/або часу

Ви можете створити об'єкт дати, вказавши рік, місяць і день. Тоді ці значення доступні у вигляді атрибутів:

```
>>> from datetime import date
>>> halloween = date(2019, 10, 31)
>>> halloween
datetime.date(2019, 10, 31)
>>> halloween.day
31
>>> halloween.month
10
>>> halloween.year
2019
```

Ви можете надрукувати дату за допомогою методу isoformat ():

```
>>> halloween.isoformat()
'2019-10-31'
```

**Пакети (Packages)**

Пакети в мові Python забезпечують спосіб розподілу набора модулів по каталогам файлової системи та використання точкової нотації для доступу до модулів підпакетів.

Ми перейшли від поодиноких рядків коду, до багаторядкових функцій, до окремих програм, до кількох модулів в одному каталозі. Якщо у вас немає багато модулів, один каталог працює добре.

Щоб дозволити додаткам Python ще більше масштабуватися, ви можете організувати модулі у файли та ієрархії модулів, які називаються пакетами. Пакет - це лише підкаталог, що містить файли .py. І ви можете заглибитися на кілька рівнів з каталогами всередині них.

Ми щойно написали модуль, який вибирає місце для швидкого харчування. Давайте додамо подібний модуль, щоб роздавати життєві поради. Ми зробимо одну нову основну програму під назвою question.py у нашому поточному каталозі. Тепер зробіть підкаталог з назвою choices і помістіть у нього два модулі —fast.py та advice.py. Кожен модуль має функцію, яка повертає рядок.

Основна програма (questions.py) має імпорт та додатковий рядок (Лістинг 7).

Лістинг 7. Файл questions.py

```
from choices import fast, advice

print("Let's go to", fast.pick())
print("Should we take out?", advice.give())
```

Інструкція `from choices` змушує Python шукати каталог з назвою `choices`, починаючи з вашого поточного каталогу. Усередині `choices` він шукає файли fast.py та advice.py.

Перший модуль (choices/fast.py) - це той самий код, що і раніше, щойно переміщений у каталог choices (Лістинг 8).

Лістинг 8. choices/fast.py

```
from random import choice

places = ["McDonalds", "KFC", "Burger King", "Taco Bell",
"Wendys", "Arbys", "Pizza Hut"]
def pick():
    """Return random fast food place"""
    return choice(places)
```

Другий модуль (choices/advice.py) є новим, але він багато в чому працює як його родич fast-food (Лістинг 9).

Лістинг 9. choices/advice.py

```
from random import choice

answers = ["Yes!", "No!", "Reply hazy", "Sorry, what?"]
def give():
    """Return random advice"""
    return choice(answers)
```

ПРИМІТКА

Якщо ваша версія Python раніша 3.3, вам знадобиться ще одна річ у підкаталозі choices, щоб зробити його пакетом Python: файл з назвою __init__.py. Це може бути порожній файл, але Python до 3.3 має потребу в ньому, щоб розглядати каталог, що містить його, як пакет.

Запустіть основну програму questions.py (з вашого поточного каталогу, а не в `choices`), щоб побачити, що відбувається:

```
$ python questions.py
Let's go to KFC
Should we take out? Yes!
$ python questions.py
Let's go to Wendys
Should we take out? Reply hazy
$ python questions.py
Let's go to McDonalds
Should we take out? Reply hazy
```

***Інструменти управління пакетами***

Пакетами Python можуть бути інструменти, бібліотеки, фреймворки, застосунки. Існують десятки тисяч доступних пакетів, які можна використовувати для створення власних проектів. Багато пакетів можна знайти в Python Package Index (PyPI, https://pypi.org/ ).

Найбільш часто використовувані менеджери пакетів Python – це **pip** і easy_install.

Дані інструменти допомагають виконати наступні завдання:

- скачування, установка, видалення пакетів;
- зборка пакетів;
- управління пакетами Python і багато іншого.

Менеджер пакетів pip входить в поставку Python починаючи з версії 3.4.

Одна з головних переваг pip - це простота інтерфейсу командного рядка, яка дозволяє встановити пакети Python простою командою

```
pip install some-package-name
```

Так само просто і видаляти пакети:

```
pip uninstall some-package-name
```

Щоб побачити список усіх доступних команд та опцій менеджера пакетів pip нам потрібно буде лише виконати в командному рядку термінала:

```
pip
```

Можна дізнатися більше про команду (наприклад, `install`):

```
D:\Demo\Python>pip install --help
Usage:
  pip install [options] <requirement specifier> [package-index-options] ..
  pip install [options] -r <requirements file> [package-index-options] ...
  pip install [options] [-e] <vcs project url> ...
  pip install [options] [-e] <local project path> ...
  pip install [options] <archive url/path> ...
```

**Створення віртуальних середовищ**

Перш ніж встановлювати будь-який пакет Python, рекомендується створити віртуальне середовище. Віртуальні середовища Python дозволяють вам встановлювати пакети Python в ізольованому місці, а не глобально.

Як випливає з назви, віртуальне середовище – це програмний інструмент для створення ізольованих або незалежних віртуальних середовищ Python в комп'ютерній системі. Цей інструмент можна використовувати для керування пакетами Python для проектів з різними (і часто несумісними) потребами з точки зору версій і взаємозалежностей між пакетами. Він створює середовище, яке має власні каталоги встановлення, яке не ділиться бібліотеками з іншими середовищами (і за бажанням не має доступу до глобально встановлених бібліотек).

Існує кілька різних методів створення віртуального середовища у вашій системі. Починаючи з Python 3.5, рекомендованим способом створення віртуального середовища є використання модуля venv. Ви можете переглянути його на офіційній сторінці документації (https://docs.python.org/3/library/venv.html ) для отримання додаткової інформації.

Іншим поширеним способом створення віртуальних середовищ є використання стороннього пакета Python *virtualenv*. Ви можете знайти його на його офіційному веб-сайті: https://virtualenv.pypa.io.

Розглянемо рекомендовану техніку, яка використовує модуль venv зі стандартної бібліотеки Python.

Використання віртуального середовища включає етапи створення, активації, роботи з ним і деактивації. Активація віртуального середовища створює певний шлях за лаштунками, так що коли ви викликаєте інтерпретатор Python з цієї оболонки, ви фактично викликаєте активне віртуальне середовище, а не системне.

Далі наведено повний приклад створення віртуального середовиша для ОС Windows [18]:

1. Відкрийте термінал і перейдіть до папки (каталогу), яку ви використовуєте як root для наших проектів (нехай це папка D:\Demo\Python). Створіть нову папку під назвою my-project і перейдіть до неї.

```
D:\Demo\Python>mkdir my-project
D:\Demo\Python>cd my-project
```

Перевірте шлях до інтерпретатора Python

```
D:\Demo\Python\my-project>where python
C:\Python39\python.exe
C:\Users\vavia\AppData\Local\Microsoft\WindowsApps\python.exe
```

2. Створіть віртуальне середовище під назвою lpp3ed.

```
D:\Demo\Python\my-project>python -m venv lpp3ed
```

3. Після створення віртуального середовища активуйте його.

```
D:\Demo\Python\my-project>lpp3ed\Scripts\activate
(lpp3ed) D:\Demo\Python\my-project>
```

Перевірте шлях до інтерпретатора Python ще раз, тепер віртуальний env python вказано першим:

```
(lpp3ed) D:\Demo\Python\my-project>where python
D:\Demo\Python\my-project\lpp3ed\Scripts\python.exe
C:\Python39\python.exe
C:\Users\vavia\AppData\Local\Microsoft\WindowsApps\python.exe
```

4. Потім переконайтеся, що ви використовуєте потрібну версію Python (3.9.X), запустивши інтерактивну оболонку Python.

```
(lpp3ed) D:\Demo\Python\my-project>python
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct 5 2020, 15:34:40)
[MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more in-
formation.
>>> exit()
(lpp3ed) D:\Demo\Python\my-project>
```

5. Деактивуйте віртуальне середовище.

```
(lpp3ed) D:\Demo\Python\my-project>deactivate
D:\Demo\Python\my-project>
```

Ці кроки – це все, що вам потрібно для початку роботи з віртуальним середовищем

## Питання для самоконтролю

1. Які функції виконують модулі Python?
2. Які є варіанти імпортування функцій з модуля?
3. Дайте характеристику стандартній бібліотеці Python.
4. Чим пакети відрізняються від модулів?
5. Які завдання вирішують інструменти управління пакетами?
6. Для яких цілей створюють віртуальні середовища Python? Які ви знаєте методи створення віртуального середовища?

## Навчальне завдання №2

**Розробка програм на Python з використанням функцій. Словники. Обробка виняткових ситуацій, робота з файлами**

Вивчити особливості використання функцій, обробки виняткових ситуацій, роботи з файлами в мові Python, отримати практичні навички розробки програм.

**Варіант 1.**

**Вправа 1.**

На вході маємо список рядків різної довжини. Необхідно написати функцію all_eq (lst), яка поверне новий список з рядків однакової довжини. Довжину підсумкового рядка визначаємо виходячи з найдовшого з них. Якщо конкретний рядок коротше найдовшого, доповнити його нижніми підкресленнями з правого краю до необхідної кількості символів. Розташування елементів початкового списку не змінювати.

Підказка.

Використати функції max(iterable, *[, key]), len, генератор списку і метод рядка ljust.

**Вправа 2.**

Напишіть програму Python, яка показує перші 10 рядків файлу, ім'я якого передане в якості аргумента командного рядка. Вивести відповідне повідомлення про помилку, якщо файл, запитуваний користувачем, не існує або аргумент командного рядка пропущено.

Підказка.

Імпортуйте модуль sys. Перевірте, що програмі було передано тільки один аргумент командного рядка. В блоці try відкрийте файл для читання. Організуйте читання і виведення 10-ти рядків файлу. Перед виведенням можна

видалити символ кінця рядка методом rstrip. В блоці except відобразити повідомлення про помилку.

**Варіант 2.**

**Вправа 1.**

Напишіть функцію change (lst), яка приймає список і змінює місцями його перший і останній елемент. У вихідному списку мінімум 2 елементи.

Підказка.

Можна використати методи pop, append, insert для списка.

**Вправа 2.**

Python використовує символ # для позначення початку коментаря. Коментар продовжується від символу # до кінця рядка, що його містить. У цій вправі ви створите програму, яка видаляє всі коментарі з вихідного файлу Python. Перевірте кожен рядок у файлі, щоб визначити, чи присутній символ #. Якщо це так, тоді ваша програма повинна видалити всі символи з символу # до кінця рядка (ми будемо ігнорувати ситуацію, коли символ коментаря зустрічається всередині рядка). Збережіть змінений файл, використовуючи нову назву. І ім'я вхідного файлу, і ім'я вихідного файлу слід читати від користувача. Переконайтеся, що у разі виникнення проблеми під час доступу до будь-якого з файлів відображається відповідне повідомлення про помилку.

Підказка.

Використайте три блоки try – except. Використайте цикл for in для читання рядків файлу, метод find для рядка, рядковий зріз для виділення частини рядка до коментаря.

**Варіант 3.**

**Вправа 1.**

Дано рядок у вигляді випадкової послідовності чисел від 0 до 9. Потрібно створити словник, який в якості ключів буде приймати дані числа (тобто ключі будуть типу int), а в якості значень - кількість цих чисел в наявній послідовності. Для побудови словника створіть функцію count_it (sequence), яка приймає рядок з цифр. Функція повинна повернути словник з 3-х найбільш часто повторюваних чисел.

Підказка.

Можна використати генератор для створення словника, метод count для рядка, метод items для словника, функцію sorted для сортування словника і зрізи.

**Вправа 2.**

Напишіть програму Python, яка виводить 10 останніх рядків файлу, ім'я якого передане в якості аргумента командного рядка. Вивести відповідне повідомлення про помилку, якщо файл, запитуваний користувачем, не існує або аргумент командного рядка пропущено. Для вирішення цієї проблеми можна застосувати кілька різних підходів. Одним із варіантів є завантаження всього вмісту файлу в список, а потім відображення його останніх 10 елементів. Інший варіант - прочитати вміст файлу двічі, один раз для підрахунку рядків і другий раз для відображення його останніх 10 рядків. Однак обидва ці рішення небажані при роботі з великими файлами. Існує ще одне рішення, яке вимагає лише одного прочитання файлу та одночасне збереження 10 рядків із файлу. Для цього рядки файлу додають до списку. При накопиченні більше 10-ти елементів самий старий рядок видаляють. Розробіть таке рішення.

Підказка.

Імпортуйте модуль sys. Перевірте, що програмі було передано тільки один аргумент командного рядка. В блоці try відкрийте файл для читання. Для додавання рядка в список використайте метод append, для видалення – метод pop. В блоці except відобразити повідомлення про помилку. Виведіть елементи списку.

**Методичні вказівки до виконання завдання №2**

**Приклади розв’язування задач**

**Приклад 1.** Відсортувати словник за значеннями.

```
scores = {
    'Foo' : 10,
    'Bar' : 34,
    'Miu' : 88,
}
# sort using a lambda expression
sorted_names = sorted(scores, key=lambda x: scores[x])
for k in sorted_names:
    print("{} : {}".format(k, scores[k]))

# Foo : 10
# Bar : 34
# Miu : 88

print('')
```

**Приклад 1.** Відсортувати словник за ключами.

```
scores = {
    "Jane" : 30,
    "Joe" : 20,
```

```
    "George" : 30,
    "Hellena" : 90,
}
for name in sorted(scores.keys()):
    print(f"{name:8} {scores[name]}")
```

Output

```
George  30
Hellena 90
Jane    30
Joe     20
```

**Приклад 3.** Кожен веб-сервер реєструє відвідувачів та їх запити у файлі журналу. Веб-сервер Apache має файл журналу, подібний до наведеного нижче. Кожен рядок - це «хіт», запит від браузера відвідувача. Кожен рядок починається з IP -адреси відвідувача. наприклад 217.0.22.3. Враховуючи такий файл журналу від Apache, повідомте, скільки звернень (рядків) відбулось з кожної IP-адреси.

Лістинг 6. Приклад файла журналу `apache_access.log`

```
127.0.0.1 - - [10/Apr/2007:10:39:11] "GET / HTTP/1.1" 500 606 "-"
127.0.0.1 - - [10/Apr/2007:10:39:11] "GET /favicon.ico HTTP/1.1" 200 766
139.12.0.2 - - [10/Apr/2007:10:40:54] "GET / HTTP/1.1" 500 612 "-"
139.12.0.2 - - [10/Apr/2007:10:40:54] "GET /favicon.ico HTTP/1.1" 200 766
127.0.0.1 - - [10/Apr/2007:10:53:10] "GET / HTTP/1.1" 500 612 "-"
127.0.0.1 - - [10/Apr/2007:10:54:08] "GET / HTTP/1.0" 200 3700 "-"
127.0.0.1 - - [10/Apr/2007:10:54:08] "GET /style.css HTTP/1.1" 200 614
127.0.0.1 - - [10/Apr/2007:10:54:08] "GET /img/pti-round.jpg HTTP/1.1" 200
127.0.0.1 - - [10/Apr/2007:10:54:21] "GET /unix_sysadmin.html HTTP/1.1"
217.0.22.3 - - [10/Apr/2007:10:54:51] "GET / HTTP/1.1" 200 34 "-"
217.0.22.3 - - [10/Apr/2007:10:54:51] "GET /favicon.ico HTTP/1.1" 200
217.0.22.3 - - [10/Apr/2007:10:54:53] "GET /cgi/pti.pl HTTP/1.1" 500 617
127.0.0.1 - - [10/Apr/2007:10:54:08] "GET / HTTP/0.9" 200 3700 "-"
```

Очікуваний результат:

```
127.0.0.1       7
139.12.0.2      2
217.0.22.3      3
```

Рішення.

```
filename = 'apache_access.log'

count = {}

with open(filename) as fh:
    for line in fh:
        space = line.index(' ')
        ip = line[0:space]
        if ip in count:
            count[ip] += 1
        else:
            count[ip] = 1

for ip in count:
    print("{:16} {:>3}".format(ip, count[ip]))
```

**Приклад 4.** Задано два файли – a.txt:

```
Tomato=78
Avocado=23
Pumpkin=100
```

і b.txt:

```
Cucumber=17
Avocado=10
Cucumber=10
```

Напишіть скрипт, який бере два файли та об'єднує їх, додаючи значення для кожного овоча. Очікуваний результат такий:

```
Avocado=33
Cucumber=27
Pumpkin=100
Tomato=78
```

Рішення:

```
c = {}
with open('a.txt') as fh:
    for line in fh:
        k, v = line.rstrip("\n").split("=")
        if k in c:
            c[k] += int(v)
        else:
            c[k] = int(v)

with open('b.txt') as fh:
    for line in fh:
        k, v = line.rstrip("\n").split("=")
        if k in c:
            c[k] += int(v)
        else:
            c[k] = int(v)

with open('out.txt', 'w') as fh:
    for k in sorted(c.keys()):
        fh.write("{}={}\n".format(k, c[k]))
```

**Приклад 5.** Написати функцію, яка імітує кидання двох гральних кубиків.

```
import random
def main():
    min=1
    max=6
    roll_again ="yes"
    while roll_again == "yes" or roll_again == "y":
        print("Rolling the dices...")
        print("The values are..")
        print(random.randint(min, max))
        print(random.randint(min, max))
        roll_again = input("Roll the dices again?")
```

```
if __name__ == '__main__':
    main()
```

**Приклад 6.** Реалізувати ділення цілих чисел, введених з клавіатури. Передбачити обробку різних типів винятків.

```
try:
    n = input("Enter 1st number:")
    m = input("Enter 2nd number:")
    result = int(num1)/int(num2)
except ValueError as ve:
    print(ve)
    exit()
except ZeroDivisionError as zde:
    print(zde)
    exit()
except TypeError as te:
    print(te)
    exit()
except:
    print('Unexpected Error!')
    exit()
print(result)
```

## Тема 8. Винятки. Робота з файлами

**Мета:** розглянути обробку винятків, які виникають при помилках часу виконання; з'ясувати режими відкриття файлів та типи файлів; вивчити особливості читання та запису для файлів різних типів.

**План**

1. Винятки (Exceptions).
2. Робота з файлами.

### Основні терміни і поняття

*Винятки (Exceptions). Обробка винятків. Блок try. Блок except. Файл. Об'єкт файлу. Режим відкриття файлу. Файли CSV. Файли JSON.*

### Основні теоретичні положення

**Винятки (Exceptions)**

Винятки – це помилки часу виконання (ділення на нуль, не той тип введених даних, тощо). У деяких мовах помилки позначаються спеціальними повернеными значеннями функцій. Коли справи йдуть в програмі не так, Python генерує виняток і запускає код, пов'язаний з обробкою помилки.

Деякі з винятків ви вже бачили, наприклад, зверталися до списку або кортежу з положенням поза діапазоном або до словника з неіснуючим ключем. Коли ви запускаєте код, який може вийти з ладу за певних обставин, вам також потрібні відповідні обробники винятків, щоб перехопити будь-які потенційні помилки.

Радимо додати обробку винятків будь-де, де може виникнути виняток, щоб користувач знав, що відбувається. Можливо, вам не вдасться виправити проблему, але принаймні ви можете відзначити обставини та витончено закрити програму. Якщо виняток трапляється в якійсь функції і не фіксується там, він як бульбашка спливає наверх, поки він не буде спійманий відповідним обробником у якійсь викликовій функції. Якщо ви не надаєте власний обробник винятків, Python надрукує повідомлення про помилку та деяку інформацію про те, де сталася помилка, а потім припинить роботу програми, як показано в наступному фрагменті:

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> short_list[position]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

***Інструкція try…except…else…finally***

Для обробки винятків призначено інструкцію

try:
  <Блок, в якому можливі винятки>
except [<Exception1>] :
  <Блок, що виконується при виникненні винятку [<Exception1>]>
. . . . .
[except [<ExceptionN>] :
  <Блок, що виконується при виникненні винятку [<ExceptionN>]>]
[else :
  <Блок, що виконується якщо виняток не виник>]
[finally:
  <Блок, який виконується в будь-якому випадку>]

Тут інструкції в квадратних дужках є необов'язковими, а Exception1, …, ExceptionN – класи (типи) винятків, які можуть виникнути.

*Деякі типи винятків*

ArithmeticError - генерується, коли в числових обчисленнях виникає помилка.

IndexError - генерується, коли індекс послідовності не існує.

ValueError - викликається, коли в указаному типі даних є неправильне значення.

ZeroDivisionError - генерується, коли другий оператор у діленні дорівнює нулю.

FileExistsError - викликається під час спроби створити файл або каталог, який уже існує.

Більш детальну інформацію дивись на офіційному сайті (https://docs.python.org/3/library/exceptions.html ).

Тепер розглянемо деякі приклади.

***Обробка помилок за допомогою try і except***

Замість того, щоб залишати речі на волю випадку, скористайтеся try щоб обернути код з можливими помилками і except щоб забезпечити обробку помилок:

```
>>> 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)
...
Need a position between 0 and 2 but got 5
```

Запускається код всередині блоку try. Якщо є помилка, виникає виняток, і код всередині блоку except запускається. Якщо помилок немає, блок except пропускається.

Використання протого except без аргументів, як ми зробили тут, є загальним для будь-якого типу винятків. Якщо може виникнути кілька типів винятків, краще надати окремий обробник винятків для кожного.

Іноді вам потрібні відомості про винятки поза типом. Ви отримуєте повний об'єкт виключення в змінній `name`, якщо використовуєте форму:

```
except exceptiontype as name
```

У наведеному нижче прикладі спочатку шукається IndexError, оскільки це тип винятку, який виникає, коли ви задаєте для послідовності неправильний індекс. Програма зберігає виняток IndexError у змінній err та будь-який інший виняток у змінній other. У прикладі друкується все, що зберігається в other, щоб показати, що ви отримуєте в цьому об’єкті:

```
>>> short_list = [1, 2, 3]
>>> while True:
...     value = input('Position [q to quit]? ')
...     if value == 'q':
...         break
...     try:
...         position = int(value)
...         print(short_list[position])
...     except IndexError as err:
...         print('Bad index:', position)
...     except Exception as other:
...         print('Something else broke:', other)
...
Position [q to quit]? 1
2
Position [q to quit]? 0
1
Position [q to quit]? 2
3
Position [q to quit]? 3
```

```
Bad index: 3
Position [q to quit]? 2
3
Position [q to quit]? two
Something else broke: invalid literal for int() with base 10: 'two'
Position [q to quit]? Q
```

Введення позиції 3 викликало IndexError, як і очікувалося. Введення two не влаштовує функцію int (), і тут спрацьовує другий except.

**Робота з файлами**

***Відкриття файлу***

Ви використовуєте функцію open у Python, щоб відкрити файл. Функція open створює об'єкт файлу та пов'язує його з файлом на диску. Ось загальний формат використання функції open:

```
file_variable = open(filename, mode)
```

У загальному форматі:

• file_variable - це ім’я змінної, яка посилатиметься на об’єкт файлу.

• filename - це рядок, що визначає ім’я файлу.

• mode - це рядок, що визначає режим (читання, запис тощо), у якому буде відкрито файл.

Перша буква режиму позначає операцію:

• **r** означає читати.

• **w** означає писати. Якщо файл не існує, його створюють. Якщо файл існує, його переписують.

• **x** означає запис, але тільки якщо файл ще не існує.

• **a** означає додавання (запис після закінчення), якщо файл існує.

Друга буква режиму - це тип файлу:

• **t** (або нічого) означає текст.

• **b** означає двійковий.

Після відкриття файлу ви викликаєте функції для читання або запису даних; це буде показано у наступних прикладах.

Нарешті, вам потрібно закрити файл, щоб переконатися, що будь-які записи завершені, а пам'ять звільнена. Пізніше ви побачите, як використовувати with для автоматизації закриття.

Ця програма відкриває файл під назвою oops.txt і закриває його, нічого не записуючи. Це створить порожній файл:

```
>>> fout = open('oops.txt', 'wt')
>>> fout.close()
```

Розглянемо наступний фрагмент коду:

```
# Read the file name from the user
fname = input("Enter the file name: ")
# Attempt to open the file
try:
    inf = open(fname, "r")
except FileNotFoundError:
    # Display an error message and quit if the file was not
    # opened successfully
```

```
        print("'%s' could not be opened. Quitting...")
        quit()
. . .
```

***Запис в текстовий файл за допомогою print()***

Давайте відтворимо oops.text, але тепер запишемо рядок і закриємо його:

```
>>> fout = open('oops.txt', 'wt')
>>> print('Oops, I created a file.', file=fout)
>>> fout.close()
```

Для print ми використовували аргумент file. Без цього print записує на стандартний вивід, який є вашим терміналом.

***Запис в текстовий файл за допомогою write()***

```
>>> poem = '''There was a young lady named Bright,
... Whose speed was far faster than light;
... She started one day
... In a relative way,
... And returned on the previous night.'''
>>> len(poem)
150
>>> fout = open('relativity', 'wt')
>>> fout.write(poem)
150
>>> fout.close()
```

Функція write повертає кількість записаних байтів. Вона не додає пробілів або нових рядків, як це робить print. Як і раніше, ви також можете використати print щоб записати багаторядковий рядок у текстовий файл:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout)
>>> fout.close()
```

Якщо файл relativity є для нас цінним, подивимось, чи дійсно використання режиму x захищає нас від перезапису:

```
>>> fout = open('relativity', 'xt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'relativity'
```

Ви можете застосувати обробник винятків:

```
>>> try:
...     fout = open('relativity', 'xt')
...     fout.write('stomp stomp stomp')
... except FileExistsError:
...     print('relativity already exists!. That was a close
one.')
...
relativity already exists!. That was a close one.
```

***Читання текстового файлу за допомогою read(), readline() або readlines()***

Ви можете викликати read() без аргументів, щоб прочитати весь файл одночасно, як показано в наведеному нижче прикладі (будьте обережні, роблячи це з великими файлами; гігабайтний файл споживає гігабайт пам'яті):

```
>>> fin = open('relativity', 'rt' )
>>> poem = fin.read()
>>> fin.close()
>>> len(poem)
150
```

Ви можете вказати максимальну кількість символів, щоб обмежити, скільки read() повертає одночасно. Давайте прочитаємо 100 символів одночасно і додамо кожен шматок до рядка вірша, щоб відновити оригінал:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> chunk = 100
>>> while True:
...     fragment = fin.read(chunk)
...     if not fragment:
...         break
...     poem += fragment
...
>>> fin.close()
>>> len(poem)
150
```

Після того, як ви прочитаєте до кінця, подальші виклики read () повернуть порожній рядок (''), який вважається False в `if not fragment`. Це вириває нас з циклу while True.

Ви також можете прочитати файл по рядкам за допомогою readline(). У цьому наступному прикладі ми додаємо кожен рядок до рядка вірша, щоб відновити оригінал:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> while True:
... line = fin.readline()
... if not line:
... break
... poem += line
...
>>> fin.close()
>>> len(poem)
150
```

Для текстового файлу навіть порожній рядок має довжину одиниця (символ нового рядка) і оцінюється як True. Коли файл буде прочитано, readline() (як і read()) також повертає порожній рядок, який також оцінюється як False.

Найпростіший спосіб читання текстового файлу - це використання ітератора. Він повертає один рядок за раз. Це схоже на попередній приклад, але з меншим кодом:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> for line in fin:
...     poem += line
...
>>> fin.close()
>>> len(poem)
```

```
150
```

Усі попередні приклади врешті-решт побудували вірш по одному рядку. Виклик readlines () читає рядок за раз і повертає список однорядкових рядків:

```
>>> fin = open('relativity', 'rt' )
>>> lines = fin.readlines()
>>> fin.close()
>>> print(len(lines), 'lines read')
5 lines read
>>> for line in lines:
... print(line, end='')
...
There was a young lady named Bright,
Whose speed was far faster than light;
She started one day
In a relative way,
And returned on the previous night.>>>
```

***Автоматичне закриття файлів за допомогою with***

Якщо ви забули закрити відкритий файл, Python закриє його після того, як на нього більше не буде посилань. Це означає, що якщо ви відкриваєте файл у функції та не закриваєте його явно, він закриється автоматично після завершення функції. Але ви могли відкрити файл у довгостроковій функції або в головному розділі програми. Файл слід закрити, щоб змусити завершити всі залишкові записи.

У Python є контекстні менеджери для очищення таких речей, як відкриті файли. Ви використовуєте форму `with expression as variable:`

```
>>> with open('relativity', 'wt') as fout:
...     fout.write(poem)
...
```

Після завершення блоку коду в контекстному менеджері (в даному випадку один рядок) (зазвичай або через виняток) файл автоматично закривається.

***Робота з файлами CSV***

Формат CSV (Comma Separated Values) є найпоширенішим форматом імпорту та експорту для електронних таблиць і баз даних. Однак CSV не є точним стандартом – є різні додатки, які мають різні конвенції стандарту [16].

Модуль csv Python реалізує класи для читання та запису табличних даних у форматі CSV. У рамках цього він підтримує різні діалекти формата CSV, в тому числі діалект, який використовує програма Excel.

Модуль csv надає ряд функцій, включаючи:

• csv.reader (csvfile, dialect='excel', **fmtparams) Повертає об'єкт reader, який перебирає рядки у даному файлі csv. Можна вказати додатковий параметр діалекту. Це може бути екземпляр підкласу класу Dialect або один із рядків, які повертає функція list_dialects().

• csv.writer (csvfile, dialect='excel', **fmtparams) Повертає об'єкт writer, який відповідає за перетворення даних користувача в рядки з роздільниками у даному файлі csv. Надано необов’язковий параметр діалекту. Аргументи клю-

чового слова fmtparams можна надати, щоб замінити окремі параметри форматування в поточному діалекті.

• csv.list_dialects() Повертає назви всіх зареєстрованих діалектів. Наприклад, у Mac OS X стандартним списком діалектів є ['excel', 'excel tab', 'unix'].

*Клас Writer*

Об'єкт класу Writer повертає функція csv.writer(). Клас Writer підтримує два методи, які використовуються для запису даних у файл CSV:

• writerow(row) Записує рядок у файл, відформатований відповідно до поточного діалекту.

• writerows(rows) Записати декілька рядків у файл, відформатований відповідно до поточного діалекту.

Наступна програма, в якій створюється файл з назвою sample.csv, ілюструє просте використання модуля csv.

Оскільки ми не вказали діалект, буде використовуватися діалект «excel» за замовчуванням. Метод writerow() використовується для запису кожного окремого списку рядків, розділених комами, у файл CSV.

```
print('Crearting CSV file')
with open('sample.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['She Loves You', 'Sept 1963'])
    writer.writerow(['I Want to Hold Your Hand', 'Dec 1963'])
    writer.writerow(['Cant Buy Me Love', 'Apr 1964'])
    writer.writerow(['A Hard Days Night', 'July 1964'])
```

Отриманий файл можна переглянути, як показано нижче:


Рис. 4 Вміст файлу CSV

Більше того, оскільки це файл CSV, ми також можемо відкрити його в Excel:

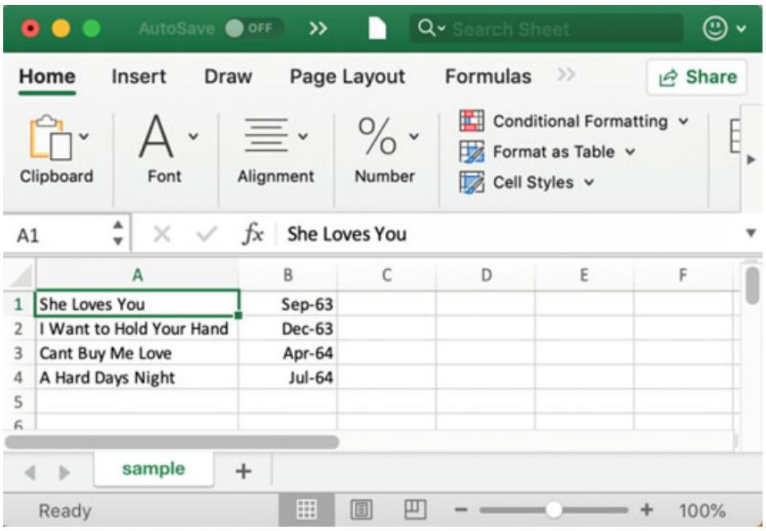


Рис. 5 Файл CSV, відкритий в Excel

Нижче наведено дуже простий приклад читання файлу CSV за допомогою об’єкта reader csv:

```
print('Starting to read csv file')
with open('sample.csv', newline='') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        print(*row, sep=', ')
print('Done Reading')
```

Виведення цієї програми на основі файлу sample.csv, створеного раніше, має вигляд:

```
Starting to read csv file
She Loves You, Sept 1963
I Want to Hold Your Hand, Dec 1963
Cant Buy Me Love, Apr 1964
A Hard Days Night, July 1964
Done Reading
```

Більш детальну інформацію про роботу з CSV файлами можна знайти за посиланням https://docs.python.org/3/library/csv.html .

***Робота з JSON файлами***

JSON — це абревіатура від JavaScript Object Notation, і це підмножина мови JavaScript. Він існує вже майже два десятиліття, тому він добре відомий і широко поширений у більшості мов, хоча насправді він незалежний від мови. Ви можете прочитати про це все детально на веб-сайті (https://www.json.org/ ), а тут ми дамо короткий вступ [18].

JSON заснований на двох структурах: наборі пар ім’я/значення та впорядкованому списку значень. Досить просто усвідомити, що ці два об’єкти відображаються відповідно до типів даних dict і list в Python. Як типи даних JSON пропонує рядки, числа, об’єкти та значення, що складаються з true, false та null. Давайте розглянемо короткий приклад:

```
# json_basic.py
import sys
import json
data = {
    'big_number': 2 ** 3141,
    'max_float': sys.float_info.max,
    'a_list': [2, 3, 5, 7],
}
json_data = json.dumps(data)
data_out = json.loads(json_data)
assert data == data_out # json and back, data matches
```

Програма починається з імпорту модулів sys та json. Потім ми створюємо простий словник із деякими числовими даними та списком. Ми хочемо протестувати серіалізацію та десеріалізацію за допомогою дуже великих чисел, як int, так і float, тому ми поставили 2**3141 і будь-яке найбільше число з плаваючою точкою, яке може обробити наша система.

Ми серіалізуємо за допомогою метода json.dumps(), який приймає дані та перетворює їх у рядок у форматі JSON. Потім ці дані подаються в метод json.loads(), який робить навпаки: із рядка, відформатованого JSON, він відновлює дані в Python. В останньому рядку ми переконуємося, що вихідні дані та результат серіалізації/десеріалізації через JSON збігаються.

Давайте подивимося, як виглядали б дані JSON, якби ми їх надрукували:

```
import json
info = {
    'full_name': 'Sherlock Holmes',
    'address': {
        'street': '221B Baker St',
        'zip': 'NW1 6XE',
        'city': 'London',
        'country': 'UK',
    }
}
print(json.dumps(info, indent=2, sort_keys=True))
```

У цьому прикладі ми створюємо словник із даними Шерлока Холмса. Але зверніть увагу, як ми викликаємо json.dumps. Ми заказали робити відступ з двома пробілами та сортувати ключі за алфавітом. Результат буде такий:

```
{
  "address": {
    "city": "London",
    "country": "UK",
    "street": "221B Baker St",
    "zip": "NW1 6XE"
  },
  "full_name": "Sherlock Holmes"
}
```

**Питання для самоконтролю**

1. Що таке виняток (exception)? Які бувають винятки?
2. Які існують засоби Python для обробки винятків?
3. Як відкрити файл у Python?
4. Які варіанти запису в текстовий файл ви знаєте?
5. Які варіанти читання з текстового файлу ви знаєте?
6. Що таке автоматичне закриття файлів?
7. Що таке файл CSV? Як реалізувати в Python читання та запис в такий файл?
8. Що таке файл JSON? Як реалізувати в Python читання та запис в такий файл?

# РОЗДІЛ 4 ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ НА МОВІ PYTHON

## Тема 9. Класи і об’єкти. Успадкування

**Мета:** розглянути основні принципи об’єктно-орієнтованого програмування, яке зосереджене на створенні об’єктів; з’ясувати як втілюються ці принципи в мові Python; ознайомитись з особливостями одиночного та множинного успадкування; закріпити пройдений матеріал за допомогою вирішення навчального завдання.

**План**

1. Класи і об’єкти.
2. Атрибути.
3. Методи.
4. Ініціалізація об’єктів.
5. Успадкування від батьківського класу.
6. Перевизначення (override, перекриття) методів.
7. Виклик ініціалізатора базового класу за допомогою super().
8. Множинне успадкування.

### Основні терміни і поняття

*Об’єктно-орієнтованого програмування (ООП). Клас. Об’єкт. Атрибути. Методи. Успадкування. Ініціалізація. Перевизначення (override) метода.*

### Основні теоретичні положення

**Класи і об’єкти**

Об'єктно-орієнтоване програмування (ООП) – це важлива, потужна та виразна парадигма програмування. Програміст мислить з точки зору класів та об’єктів. Клас - це план об’єкта. Об'єкт містить певні дані та забезпечує функціональні можливості, зазначені в класі.

Скажімо, ви програмуєте гру для побудови, моделювання та розвитку міст. В об'єктно-орієнтованому програмуванні ви б представляли всі речі (будівлі, людей або автомобілі) як об'єкти. Наприклад, кожен будівельний об’єкт зберігає такі дані, як назва, розмір та ціна. Крім того, кожна будівля забезпечує певну функціональність, таку як calculate_monthly_earnings (розрахунок місячних доходів). Це спрощує читання та розуміння вашого коду для інших програмістів. Що ще важливіше, тепер ви можете легко розподілити обов’язки між програмістами. Наприклад, ви кодуєте будівлі, а ваш колега кодує автомобілі, що рухаються.

Коротше кажучи, об’єктно-орієнтоване програмування допомагає писати читабельний код. Вивчаючи його, ваша здатність співпрацювати з іншими над складними проблемами покращується.

*Classes*. Клас інкапсулює дані і функціональність: дані як атрибути і функціональність як методи. Це план для створення конкретних екземплярів (instances) в пам’яті.

*Objects*. Об’єкт є конкретним втіленням (екземпляром) класу. Кожен об’єкт має власні атрибути, незалежні від інших об’єктів.

Як ми вже згадували раніше, все в Python, від чисел до функцій, є об’єктом. Однак Python приховує більшість деталей об'єктів за допомогою спеціального синтаксису. Ви можете ввести num = 7, щоб створити об'єкт типу integer зі значенням 7 і призначити посилання на об'єкт імені num. Єдиний раз, коли вам потрібно зазирнути всередину об’єктів, це коли ви хочете створити власні або змінити поведінку існуючих об’єктів. У цьому розділі ви побачите, як це зробити.

Що таке об’єкти?

Об'єкт - це власна структура даних, що містить як дані (змінні, які називаються атрибутами), так і код (функції, що називаються методами). Він являє собою унікальний приклад якоїсь конкретної речі. Подумайте про предмети як про іменники, а про їхні методи - як про дієслова. Об'єкт являє собою окрему річ, а його методи визначають, як він взаємодіє з іншими речами.

Наприклад, цілочисельний об’єкт зі значенням 7 є об’єктом, який має такі методи, як додавання та множення. Цілочисельний об’єкт зі значенням 8 - це інший об’єкт. Це означає, що в Python є вбудований клас integer, до якого належать і 7, і 8. Рядки "cat" та "duck" також є об'єктами в Python і мають рядкові методи, такі як capitalize() та replace().

### *Визначення класу за допомогою ключового слова class*

Щоб створити новий об'єкт, який ніхто ніколи раніше не створював, спочатку визначте клас, який вказує, що він містить.

Об’єкт можна порівняти із пластиковою коробкою. Клас - це як форма, яка робить цю коробку. Наприклад, у Python є вбудований клас, який створює рядкові об'єкти, такі як "cat" та "duck", та інші стандартні типи даних - списки, словники тощо. Щоб створити власний користувацький об’єкт у Python, спочатку потрібно визначити клас за допомогою ключового слова class. Давайте розглянемо кілька простих прикладів.

Припустимо, що ви хочете визначити об’єкти для представлення інформації про кішок. Кожен об'єкт буде представляти одну тварину. Спочатку вам потрібно визначити клас під назвою Cat як форму. У наведених нижче прикладах ми пробуємо декілька версій цього класу, будуючи від найпростішого класу до тих, які насправді роблять щось корисне.

Наша перша спроба - найпростіший можливий клас, порожній:

```
>>> class Cat:
...     pass
```

Тепер створимо об’єкт з класу, викликаючи ім’я класу так, ніби це функція:

```
>>> a_cat = Cat()
>>> another_cat = Cat()
```

У цьому випадку виклик Cat () створює два окремих об’єкта з класу Cat, і ми призначаємо їх іменам a_cat та another_cat. Але наш клас Cat не мав іншого коду, тому об’єкти, які ми створили з нього, просто сидять там і більше нічого робити не можуть.

**Атрибути**

Атрибут - це змінна всередині класу або об'єкта. Під час та після створення об’єкта чи класу ви можете призначити йому атрибути. Атрибутом може бути будь-який інший об’єкт. Давайте знову створимо два об’єкта класу Cat:

```
>>> class Cat:
...     pass
...
>>> a_cat = Cat()
>>> a_cat
<__main__.Cat object at 0x100cd1da0>
>>> another_cat = Cat()
>>> another_cat
<__main__.Cat object at 0x100cd1e48>
```

Коли ми визначали клас Cat, ми не вказували, як друкувати об’єкт із цього класу. Python друкує щось на зразок `<__ main__.Cat object 0x100cd1da0>`. Далі ми розглянемо, як змінити цю поведінку за замовчуванням.

Тепер призначимо нашому першому об’єкту кілька атрибутів:

```
>>> a_cat.age = 3
>>> a_cat.name = "Mr. Fuzzybuttons"
>>> a_cat.nemesis = another_cat
```

Чи можемо ми отримати до них доступ? Ми на це сподіваємось:

```
>>> a_cat.age
3
>>> a_cat.name
'Mr. Fuzzybuttons'
>>> a_cat.nemesis
<__main__.Cat object at 0x100cd1e48>
```

Оскільки nemesis був атрибутом, що посилається на інший об’єкт Cat, ми можемо використовувати a_cat.nemesis для доступу до нього, але цей інший об’єкт ще не має атрибута name:

```
>>> a_cat.nemesis.name
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Cat' object has no attribute 'name'
>>> a_cat.nemesis.name = "Mr. Bigglesworth"
>>> a_cat.nemesis.name
'Mr. Bigglesworth'
```

Навіть найпростіший об’єкт, такий як цей, можна використовувати для зберігання кількох атрибутів. Отже, ви можете використовувати кілька об’єктів для зберігання різних значень, замість того, щоб використовувати щось на зразок списку чи словника.

Коли ми еоворимо атрибути, це зазвичай означає атрибути об’єкта. Існують також атрибути класів, і ми побачимо відмінності трохи пізніше.

**Методи**

Метод - це функція в класі або об’єкті. Метод виглядає як будь-яка інша функція, але його можна використовувати особливими способами, які ви побачите пізніше.

**Ініціалізація об’єктів**

Якщо ви хочете призначити атрибути об’єктів під час створення, вам потрібен спеціальний метод ініціалізації об’єктів Python __init __ ():

```
>>> class Cat:
...     def __init__(self):
...         pass
```

Це те, що ви побачите у реальних визначеннях класу Python. Можна визнати, що __init __ () і self виглядають дивно. __init __ () - це спеціальне ім'я Python для методу, який ініціалізує окремий об'єкт з його визначення класу. Аргумент self вказує, що він відноситься до окремого об'єкта.

Коли ви визначаєте __init __ () у визначенні класу, його перший параметр має бути self.

Але навіть це визначення другого класу Cat не створило об’єкт, який справді щось зробив. Третя спроба дійсно показує, як створити простий об’єкт у Python та призначити один із його атрибутів. Цього разу ми додаємо назву параметра до методу ініціалізації:

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

Тепер ми можемо створити об'єкт з класу Cat, передавши рядок для параметра name:

```
>>> furball = Cat('Grumpy')
```

Ось що робить цей рядок коду:

• Виявляє визначення класу Cat.

• Визначає (створює) новий об’єкт у пам’яті.

• Викликає метод __init __ () об'єкта, передаючи цей новостворений об'єкт як self, а інший аргумент ('Grumpy') як ім'я.

• Зберігає значення імені в об’єкті.

• Повертає новий об’єкт.

• Прикріплює змінну `furball` до об’єкта.

Цей новий об’єкт схожий на будь-який інший об’єкт у Python. Ви можете використовувати його як елемент списку, кортежу, словника або набору. Ви можете передати його функції як аргумент або повернути як результат.

Як щодо значення імені, яке ми передали? Воно було збережене з об'єктом як атрибут. Ви можете читати та писати його безпосередньо:

```
>>> print('Our latest addition: ', furball.name)
Our latest addition: Grumpy
```

Не обов'язково мати метод __init __ () у кожному визначенні класу; використовується для того, щоб зробити все, що потрібно, щоб відрізнити цей об’єкт від інших, створених з того ж класу. Це не те, що деякі інші мови називають "конструктором". Python вже створив об’єкт для вас. Подумайте про __init __ () як про ініціалізатор.

ПРИМІТКА

З одного класу можна створити багато окремих об’єктів. Але пам’ятайте, що Python реалізує дані як об’єкти, тому сам клас є об’єктом. Однак у вашій програмі є лише один об’єкт класу.

**Успадкування. Успадкування від батьківського класу**

Коли ви намагаєтесь вирішити якусь проблему кодування, ви часто знаходите існуючий клас, який створює об’єкти, які роблять майже те, що вам потрібно. Що ми можемо зробити?

Одне з рішень - успадкування: створення нового класу з існуючого класу, але з деякими доповненнями або змінами. Це хороший спосіб повторного використання коду. Коли ви використовуєте успадкування, новий клас може автоматично використовувати весь код зі старого класу, але без необхідності копіювати його.

***Успадкування від батьківського класу***

Ви визначаєте лише те, що вам потрібно додати або змінити у новому класі, і це замінює поведінку старого класу. Початковий клас називається батьківським (parent), надкласом (superclass) або базовим (base) класом; новий клас називається дочірнім (child), підкласом (subclass) або похідним (derived) класом. Ці терміни взаємозамінні в об'єктно-орієнтованому програмуванні.

Отже, давайте щось успадкуємо. У наступному прикладі ми визначаємо порожній клас під назвою Car. Далі ми визначаємо підклас автомобіля під назвою Yugo. Ви визначаєте підклас, використовуючи те саме ключове слово класу, але з ім’ям батьківського класу всередині дужок (тут клас Yugo (Car)):

```
>>> class Car():
...     pass
...
>>> class Yugo(Car):
...     pass
...
```

Ви можете перевірити, чи походить клас з іншого класу, використовуючи issubclass ():

```
>>> issubclass(Yugo, Car)
True
```

Далі створіть об’єкт з кожного класу:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Дочірній клас - це спеціалізація батьківського класу; в об'єктно-орієнтованому мовленні, Yugo - це автомобіль (is-a Car). Об'єкт з назвою give_me_a_yugo є екземпляром класу Yugo, але він також успадковує все, що

може зробити Car. У цьому випадку Car і Yugo не дуже корисні, тому давайте спробуємо нові визначення класів, які насправді щось роблять:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     pass
...
```

Нарешті, створіть по одному об’єкту з кожного класу та викличте метод exclaim:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Car!
```

Не роблячи нічого особливого, Yugo успадкував метод exclaim () від Car. Насправді, Yugo каже, що це Car, і це може призвести до кризи ідентичності. Давайте подивимось, що ми можемо з цим зробити.

ПРИМІТКА

Спадковість приваблива, але її можна надмірно використовувати. Багаторічний досвід програмування показав, що надмірне використання спадкування може ускладнити управління програмами. Натомість часто рекомендується використовувати інші прийоми, такі як агрегація та композиція. Ми дійдемо до цих альтернатив у цьому розділі.

### Перевизначення (override, перекриття) методів

Як ви тільки що бачили, новий клас спочатку успадковує все від свого батьківського класу. Продовжуючи, ви побачите, як замінити або перевизначити батьківський метод. Напевно, Yugo в чомусь має відрізнятися від Car; інакше, який сенс визначати новий клас? Давайте змінимо, як метод exclaim() працює для Yugo:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car")
...
```

Тепер зробіть два об’єкти з цих класів:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Що вони кажуть?

```
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Yugo! Much like a Car
```

У цих прикладах ми замінили метод exclaim (). Ми можемо змінити будь-які методи, включаючи __init __ (). Ось ще один приклад, який використовує клас Person. Давайте зробимо підкласи, які представляють лікарів (MDPerson) та юристів (JDPerson):

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>> class MDPerson(Person):
...     def __init__(self, name):
...         self.name = "Doctor " + name
...
>>> class JDPerson(Person):
...     def __init__(self, name):
...         self.name = name + ", Esquire"
...
```

У цих випадках метод ініціалізації __init __ () бере ті ж аргументи, що і батьківський клас Person, але зберігає значення імені по-різному в екземплярі об'єкта:

```
>>> person = Person('Fudd')
>>> doctor = MDPerson('Fudd')
>>> lawyer = JDPerson('Fudd')
>>> print(person.name)
Fudd
>>> print(doctor.name)
Doctor Fudd
>>> print(lawyer.name)
Fudd, Esquire
```

***Додавання метода***

Дочірній клас також може додати метод, якого не було в його батьківському класі. Повертаючись до класів Car і Yugo, ми визначимо новий метод need_a_push () лише для класу Yugo:

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car.")
...     def need_a_push(self):
...         print("A little help here?")
...
```

**Виклик ініціалізатора базового класу за допомогою super()**

Ми побачили, як дочірній клас може додати або замінити метод від батьків. Що якби він хотів викликати цей батьківський метод? "Я радий, що ви за-

питали", - каже super (). Тут ми визначаємо новий клас під назвою EmailPerson, який представляє особу з адресою електронної пошти. По-перше, знайоме нам визначення Person:

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

Зверніть увагу, що виклик __init __ () у наступному підкласі має додатковий параметр електронної пошти:

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         super().__init__(name)
...         self.email = email
```

Коли ви визначаєте метод __init __ () для свого класу, ви замінюєте метод __init __ () його батьківського класу, і останній більше не викликається автоматично. Як наслідок, нам потрібно це явно назвати. Ось що відбувається:

• super () отримує визначення батьківського класу, Person.

• Метод __init __ () викликає метод Person .__ init __ (). Він піклується про передачу аргумента self до суперкласу, тому вам просто потрібно надати йому інші аргументи. У нашому випадку єдиний інший аргумент, який приймає Person (), - це ім'я.

• Рядок self.email = email - це новий код, який відрізняє цю EmailPerson від Person.

Продовжуючи, давайте створимо об’єкт EmailPerson:

```
>>> bob = EmailPerson('Bob Frapples', 'bob@frapples.com')
```

Ми маємо доступ до обох атрибутів name та email:

```
>>> bob.name
'Bob Frapples'
>>> bob.email
'bob@frapples.com'
```

Чому ми просто не визначили наш новий клас наступним чином?

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

Ми могли б це зробити, але це б відкинуло наше використання успадкування. Ми використовували super(), щоб змусити Person виконувати свою роботу, так само як звичайний об'єкт Person. Існує ще одна перевага: якщо визначення Person зміниться в майбутньому, використання super() забезпечить, щоб атрибути та методи, які EmailPerson успадкував від Person, відображали зміни.

Використовуйте super (), коли похідний клас (child) робить щось по-своєму, але все ще потребує чогось від батьків (як у реальному житті).

**Множинне успадкування**

Ви щойно бачили деякі приклади класів без батьківського класу, а деякі з ним. Насправді об’єкти можуть успадковуватися від кількох батьківських класів.

Якщо ваш клас посилається на метод або атрибут, якого у нього немає, Python буде шукати всіх батьків. Що робити, якщо у кількох з них є щось з такою назвою? Хто виграє?

На відміну від успадкування у людей, де домінантний ген перемагає незалежно від того, від кого він походить, успадкування в Python залежить від *method resolution order* (порядку розрішення методу). Кожен клас Python має спеціальний метод під назвою mro (), який повертає список класів, які слід відвідати, щоб знайти метод або атрибут для об'єкта цього класу. Подібний атрибут, який називається __mro__, є кортежем цих класів. Перемагає перший в списку.

Давайте визначимо клас Animal, два дочірні класи (Horse та Donkey), а потім два, отримані з них:

```
>>> class Animal:
...     def says(self):
            return 'I speak!'
...
>>> class Horse(Animal):
...     def says(self):
...         return 'Neigh!'
...
>>> class Donkey(Animal):
...     def says(self):
...         return 'Hee-haw!'
...
>>> class Mule(Donkey, Horse):
...     pass
...
>>> class Hinny(Horse, Donkey):
...     pass
...
```

Якщо ми шукаємо метод або атрибут Mule, Python розгляне такі речі в такому порядку:

1. Сам об'єкт (типу Mule)
2. Клас об'єкта (Mule)
3. Перший батьківський клас (Donkey)
4. Другий батьківський клас (Horse)
5. Клас бабусь і дідусів (Animal)

Це приблизно те саме для Hinny, але з Horse перед Donkey:

```
>>> Mule.mro()
[<class '__main__.Mule'>, <class '__main__.Donkey'>,
<class '__main__.Horse'>, <class '__main__.Animal'>,
<class 'object'>]
>>> Hinny.mro()
[<class '__main__.Hinny'>, <class '__main__.Horse'>,
<class '__main__.Donkey'>, <class '__main__.Animal'>,
class 'object'>]
```

То що ж говорять ці прекрасні звірі?

```
>>> mule = Mule()
>>> hinny = Hinny()
```

```
>>> mule.says()
'hee-haw'
>>> hinny.says()
'neigh'
```

Ми перерахували батьківські класи у порядку (батько, мати), тому вони розмовляють, як їхні тата.

Якби Horse та Donkey не мали методу says(), mule або hinny скористалися би методом says() класу бабусь і дідусів Animal і повернули б 'I speak!'.

**Питання для самоконтролю**

1 Що таке об'єктно-орієнтоване програмування (ООП)?
2 Дайте визначення класа та об’єкта.
3 Чи можна в Python після створення об’єкта чи класу призначити йому атрибути?
4 Як називають метод ініціалізації об’єктів Python? Яким повинен бути його перший параметр?
5 Чи може похідний клас викликати ініціалізатор батьківського класу?
6 Сформулюйте порядок розрішення методу при множинному успадкуванні.

**Навчальне завдання**

1. Напишіть визначення класу Book. Клас Book має атрибути даних для назви книги, імені автора та кількості сторінок. Клас також має такі методи:

a. Метод _ _init_ _ для класу. Метод повинен приймати аргументи для кожного з атрибутів даних.

b. Спеціальний метод _ _len_ _ для повернення кількості сторінок у книзі.

в. Метод _ _str_ _, який повертає рядок, що показує стан об’єкта.

2. Перегляньте наступний опис предметної області:

Банк пропонує своїм клієнтам такі типи рахунків: ощадні рахунки, чекові рахунки та рахунки грошового ринку. Клієнтам дозволяється вносити гроші на рахунок (збільшуючи таким чином його баланс), знімати гроші з рахунку (зменшуючи таким чином його баланс) і отримувати відсотки на рахунку. Кожен рахунок має процентну ставку.

Припустімо, що ви пишете програму, яка розраховуватиме суму відсотків, зароблених на банківський рахунок.

a. Визначте потенційні класи в цій предметній області.

b. Уточніть список, щоб включити лише необхідний клас або класи для цієї проблеми.

в. Визначте обов’язки класу чи класів.

 **Тестові завдання для перевірки знань**

1. Практика програмування _____ зосереджена на створенні функцій, відокремлених від даних, з якими вони працюють.

A. modular
B. procedural
C. functional
D. object-oriented

2. Практика програмування _____ зосереджена на створенні об’єктів.

A. object-centric
B. objective
C. procedural
D. object-oriented

3. Метод _____ автоматично викликається під час створення об’єкта.

A. _ _init_ _
B. init
C. _ _str_ _
D. _ _object_ _

## Тема 10. Атрибути класів та об’єктів. Властивості. Типи методів

**Мета:** розглянути атрибути класів та об’єктів; з’ясувати роль та характеристики властивостей; вивчити типи методів; закріпити пройдений матеріал за допомогою вирішення навчального завдання.

### План

1. Доступ до атрибутів.
2. Властивості для доступу до атрибутів.
3. Властивості обчислюваних значень.
4. Зміна імен для конфіденційності.
5. Атрибути класів та об’єктів.
6. Типи методів.
7. Агрегація та композиція.

## Основні терміни і поняття

*Доступ до атрибутів. Прямий доступ. Геттери та сеттери. Властивості для доступу до атрибутів. Декоратори. Методи екземплярів. Методи класів. Статичні методи.*

## Основні теоретичні положення

### Доступ до атрибутів

У Python атрибути та методи об'єктів зазвичай є загальнодоступними (public), тобто до них є прямий доступ за межами класу. Давайте порівняємо прямий доступ з деякими альтернативами.

#### *Прямий доступ*

Як ви бачили, ви можете отримувати та встановлювати значення атрибутів безпосередньо:

```
>>> class Duck:
...     def __init__(self, input_name):
...         self.name = input_name
...
>>> fowl = Duck('Daffy')
>>> fowl.name
'Daffy'
```

Але це може призвести до неконтрольованої класом зміни атрибутів.

```
>>> fowl.name = 'Daphne'
>>> fowl.name
'Daphne'
```

Наступні два розділи показують способи отримати певну конфіденційність для атрибутів, які ви не хочете, щоб хтось випадково змінив.

#### *Геттери та сеттери*

Деякі об'єктно-орієнтовані мови підтримують приватні атрибути об'єктів, до яких неможливо отримати доступ безпосередньо ззовні. Тоді програмістам може знадобитися написати методи getter та setter для читання та запису значень таких приватних атрибутів.

Python не має приватних атрибутів, але ви можете писати геттери та сеттери з заплутаними іменами атрибутів, щоб отримати трохи конфіденційності. (Найкраще рішення - використовувати властивості, описані в наступному розділі.)

У наступному прикладі ми визначаємо клас Duck з одним атрибутом екземпляра під назвою hidden_name. Ми не хочемо, щоб люди отримували прямий доступ до нього, тому ми визначаємо два методи: getter (get_name ()) та setter (set_name ()). Доступ до кожного здійснюється за допомогою властивості під назвою name. Я додав оператор print () до кожного методу, щоб показати, коли він викликається:

```
>>> class Duck():
...     def __init__(self, input_name):
```

```
...         self.hidden_name = input_name
...     def get_name(self):
...         print('inside the getter')
...         return self.hidden_name
...     def set_name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
>>> don = Duck('Donald')
>>> don.get_name()
inside the getter
'Donald'
>>> don.set_name('Donna')
inside the setter
>>> don.get_name()
inside the getter
'Donna'
```

**Властивості для доступу до атрибутів**

Рішенням Pythonic для конфіденційності атрибутів є використання властивостей.

Є два способи зробити це. Перший спосіб - додати name = property (get_name, set_name) як останній рядок нашого попереднього визначення класу Duck:

```
>>> class Duck():
>>>     def __init__(self, input_name):
>>>         self.hidden_name = input_name
>>>     def get_name(self):
>>>         print('inside the getter')
>>>         return self.hidden_name
>>>     def set_name(self, input_name):
>>>         print('inside the setter')
>>>         self.hidden_name = input_name
>>>     name = property(get_name, set_name)
```

Старий геттер і сеттер все ще працюють:

```
>>> don = Duck('Donald')
>>> don.get_name()
inside the getter
'Donald'
>>> don.set_name('Donna')
inside the setter
>>> don.get_name()
inside the getter
'Donna'
```

Але тепер ви також можете використовувати ім'я властивості, щоб отримати та встановити `hidden_name`:

```
>>> don = Duck('Donald')
>>> don.name
inside the getter
'Donald'
>>> don.name = 'Donna'
inside the setter
>>> don.name
```

```
inside the getter
'Donna'
```

У другому методі ви додаєте кілька декораторів і замінюєте імена методів get_name та set_name на ім'я:

• @property, що передує методу getter

• @name.setter, що передує методу setter

Ось як вони виглядають у коді:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.hidden_name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
```

Ви все ще можете отримати доступ до name так, ніби це атрибут:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

**Властивості обчислюваних значень**

У попередніх прикладах ми використовували властивість name для посилання на єдиний атрибут (hidden_name), що зберігається в об'єкті.

Властивість також може повертати обчислене значення. Давайте визначимо клас Circle, який має атрибут radius та обчислювану властивість diameter:

```
>>> class Circle():
...     def __init__(self, radius):
...         self.radius = radius
...     @property
...     def diameter(self):
...         return 2 * self.radius
...
```

Створіть об’єкт Circle з початковим значенням для його радіуса:

```
>>> c = Circle(5)
>>> c.radius
5
```

Ми можемо посилатись на diameter так, ніби це атрибут, такий як radius:

```
>>> c.diameter
10
```

Ось найцікавіша частина: ми можемо будь-коли змінити атрибут radius, а властивість diameter буде обчислюватися з поточного значення радіуса:

```
>>> c.radius = 7
```

```
>>> c.diameter
14
```

Якщо ви не вказуєте властивість setter для атрибута, ви не можете встановити його ззовні. Це зручно для атрибутів лише для читання:

```
>>> c.diameter = 20
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Існує ще одна перевага використання властивості перед прямим доступом до атрибутів: якщо ви коли-небудь змінюєте визначення атрибута, вам потрібно виправити лише код у визначенні класу, а не у всіх абонентах.

**Зміна імен для конфіденційності**

У прикладі класу Duck трохи раніше ми називали наш (не повністю) прихований атрибут hidden_name. Python має умову іменування атрибутів, які не повинні бути видимими поза визначенням їх класу: почніть з двох знаків підкреслення (__).

Давайте перейменуємо hidden_name на __name, як показано тут:

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.__name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.__name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.__name = input_name
...
```

Перевіримо, чи все ще працює:

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

Виглядає добре. І ви не можете отримати доступ до атрибута __name:

```
>>> fowl.__name
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Duck' object has no attribute '__name'
```

Ця угода про іменування не робить атрибути повністю конфіденційними, незважаючи на те, що Python змінює назву атрибута. Зовнішній код при певних зусиллях може мати доступ до атрибуту:

```
>>> fowl._Duck__name
'Donald'
```

**Атрибути класів та об’єктів**

Ви можете призначати атрибути класам, і вони успадковуватимуться їхніми дочірніми об'єктами:

```
>>> class Fruit:
...     color = 'red'
...
>>> blueberry = Fruit()
>>> Fruit.color
'red'
>>> blueberry.color
'red
```

Але якщо ви зміните значення атрибута в дочірньому об'єкті, це не вплине на атрибут класу:

```
>>> blueberry.color = 'blue'
>>> blueberry.color
'blue'
>>> Fruit.color
'red'
```

Якщо ви пізніше змініте атрибут класу, це не вплине на існуючі дочірні об’єкти:

```
>>> Fruit.color = 'orange'
>>> Fruit.color
'orange'
>>> blueberry.color
'blue'
```

Але це вплине на нові:

```
>>> new_fruit = Fruit()
>>> new_fruit.color
'orange'
```

**Типи методів**

Деякі методи є частиною самого класу, деякі - частиною об’єктів, створених з цього класу, а деякі - жодним із перерахованих вище:

• Якщо попереднього декоратора немає, це метод екземпляра, і його перший аргумент має бути "self" для посилання на сам окремий об'єкт.

• Якщо існує попередній декоратор @classmethod, це метод класу, і його перший аргумент має бути cls (або що завгодно, тільки не зарезервоване слово class), посилаючись на сам клас.

• Якщо існує попередній декоратор @staticmethod, це статичний метод, і його перший аргумент не є об’єктом або класом.

У наступних розділах є деякі деталі.

***Методи екземплярів***

Коли ви бачите початковий аргумент self у методах у визначенні класу, це метод екземпляра. Це типи методів, які ви зазвичай пишете при створенні власних класів. Першим параметром методу екземпляра є self, і Python передає об'єкт методу під час його виклику. Це ті методи, які ви бачили досі.

***Методи класів***

Навпаки, метод класу впливає на клас в цілому. Будь-яка зміна класу впливає на всі його об’єкти. У визначенні класу попередній декоратор @classmethod вказує, що наступна функція є методом класу. Крім того, першим параметром методу є сам клас. Традиція Python - називати параметр cls, оскільки class є зарезервованим словом і не може використовуватися тут. Давайте визначимо метод класу для A, який підраховує, скільки екземплярів об’єктів було зроблено з нього:

```
>>> class A():
...     count = 0
...     def __init__(self):
...         A.count += 1
...     def exclaim(self):
...         print("I'm an A!")
...     @classmethod
...     def kids(cls):
...         print("A has", cls.count, "little objects.")
...
>>>
>>> easy_a = A()
>>> breezy_a = A()
>>> wheezy_a = A()
>>> A.kids()
A has 3 little objects.
```

Зверніть увагу, що ми посилалися на A.count (атрибут класу) у __init __ (), а не на self.count (який був би атрибутом екземпляра об'єкта). У методі kids () ми використовували cls.count, але ми могли б так само використовувати A.count.

***Статичні методи***

Третій тип методу у визначенні класу не впливає ні на клас, ні на його об’єкти; він просто для зручності. Це статичний метод, якому передує декоратор @staticmethod, без початкових параметрів self або cls. Ось приклад, який служить рекламним роликом для класу CoyoteWeapon:

```
>>> class CoyoteWeapon():
...     @staticmethod
...     def commercial():
...          print('This CoyoteWeapon has been brought to you
by Acme')
...
>>>
>>> CoyoteWeapon.commercial()
This CoyoteWeapon has been brought to you by Acme
```

Зауважте, що нам не потрібно було створювати об’єкт із класу CoyoteWeapon, щоб отримати доступ до цього методу.

***Магічні методи***

Магічні методи - це спеціальні методи Python назви яких починаються і закінчуються подвійним підкресленням (__). Ви вже бачили такий метод:

__init __ () ініціалізує новостворений об’єкт. Є ще багато інших магічних методів.

Досить часто нам потрібно відобразити повідомлення, яке вказує на стан об’єкта. Стан об’єкта – це просто значення атрибутів об’єкта в будь-який момент. Відображення стану об'єкта є звичайним завданням. Це настільки поширено, що багато програмістів оснащують свої класи методом, який повертає рядок, що містить стан об’єкта. У Python ви даєте цьому методу спеціальну назву _ _str_ _.

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...     def __str__(self):
...         return self.text
...     def __repr__(self):
...         return 'Word("' + self.text + '")'
...
>>> first = Word('ha')
>>> first        # uses __repr__
Word("ha")
>>> print(first) # uses __str__
ha
```

При порівнянні об’єктів можна використовувати магічні методи `__eq__`, `__ne__`, `__lt__`, `__gt__`, `__le__`, `__ge__`. Інтерактивний інтерпретатор використовує функцію __repr__() для відтворення змінних для виведення.

**Агрегація та композиція**

Спадкування – це хороший прийом, який потрібно використовувати, коли ви хочете, щоб дочірній клас міг діяти як його батьківський клас (коли дитина є (is-a) батьком). Буде спокуса побудувати складні ієрархії спадкування, але іноді композиція або агрегація мають більший сенс. Яка різниця? При композиції одне є частиною іншого. Качка - це птах (спадкування), але має (has-a) хвіст (композиція). Хвіст (tail) - це не вид качки, а частина качки. У наступному прикладі давайте зробимо bill (дзьоб) та tail (хвіст) об’єктами та надамо їх новому об’єкту качки:

```
>>> class Bill():
...     def __init__(self, description):
...         self.description = description
...
>>> class Tail():
...     def __init__(self, length):
...         self.length = length
...
>>> class Duck():
...     def __init__(self, bill, tail):
...         self.bill = bill
...         self.tail = tail
```

```
...     def about(self):
...         print('This duck has a', self.bill.description,
...             'bill and a', self.tail.length, 'tail')
...
>>> a_tail = Tail('long')
>>> a_bill = Bill('wide orange')
>>> duck = Duck(a_bill, a_tail)
>>> duck.about()
This duck has a wide orange bill and a long tail
```

Агрегування теж виражає відносини «ціле – частина цілого», але трохи вільніше: одне використовує (uses) інше, але обидва існують незалежно. Качка використовує озеро, але не є його частиною.

## Питання для самоконтролю

1. Які ви знаєте способи доступу до атрибутів?
2. Що таке властивості і як їх використовують?
3. Яке призначення геттерів та сеттерів?
4. Що таке властивості обчислюваних значень?
5. Як досягти конфіденційності для імен атрибутів?
6. Чим відрізняються атрибути класів від атрибутів об’єктів?
7. Які типи методів ви знаєте?
8. Дайте характеристику методам екземплярів.
9. Чим відрізняються методи класів?
10. Дайте характеристику магічним методам Python.
11. Які типи зв’язків між класами ви знаєте?

## Навчальне завдання №3

**Розробка програм з використанням класів**

Мета: ознайомитись з реалізацією об’єктно-орієнтованого програмування у мові Python та навчитись використовувати його для розроблення програмного забезпечення.

**Варіант 1.**

**Вправа 1.**

Розробити клас 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.

**Вправа 2.**

Удосконалити додаток, розроблений в Завданні 1, додавши збереження інформації в файлі та зчитування із файлу.

**Варіант 2.**

**Вправа 1.**

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 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.

**Вправа 2.**

Удосконалити додаток, розроблений в Вправі 1, додавши збереження інформації в файлі та зчитування із файлу.

**Варіант 3.**

**Вправа 1.**

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 працівниками: пошук за декількома параметрами (за прізвищем, за віком, за відділом тощо), додавання нових працівників, видалення інформації про працівника, доступу до інформації про працівника за прізвищем та ім’ям. Написати програму, що буде демонструвати всі розроблені елементи класу.

**Вправа 2.**

Удосконалити додаток, розроблений в Вправі 1, додавши збереження інформації в файлі та зчитування із файлу.

 **Тестові завдання для перевірки знань**

1. Якщо клас має метод з назвою _ _str_ _, який із цих способів є способом виклику методу?

A. ви викликаєте його, як і будь-який інший метод: object._ _str_ _()

B. шляхом передачі екземпляра класу вбудованій функції str

C. метод автоматично викликається під час створення об'єкта

D. шляхом передачі екземпляра класу вбудованій функції роздруківки стану

2. Які ознаки метода екземпляра?

A. відсутність декоратора і наявність першого параметра self

B. наявність декоратора @instancemethod

C. наявність першого параметра self

D. відсутність декоратора і наявність першого параметра cls

## РОЗДІЛ 5 РОЗРОБКА ДОДАТКІВ З ГРАФІЧНИМ ІНТЕРФЕЙСОМ КОРИСТУВАЧА НА МОВІ PYTHON

### Тема 11. Засоби створення програм з графічним інтерфейсом користувача

**Мета:** розглянути основні вимоги до інтерфейсу користувача комп’ютера; з’ясувати основні характеристики програм з графічним інтерфейсом користувача; ознайомитись з основними інструментами побудови додатків з графічним інтерфейсом користувача.

**План**

1. Інтерфейс користувача комп’ютера.
2. Програми GUI керуються подіями.
3. Деякі Python GUI інструменти для розробників.

**Основні терміни і поняття**

*Інтерфейс користувача. Графічний інтерфейс користувача (GUI). Інтерфейс керований подіями. Фреймворк. Міжплатформенний застосунок.*

## Основні теоретичні положення

### Інтерфейс користувача комп'ютера

Інтерфейс користувача комп'ютера – це частина комп'ютера, з якою взаємодіє користувач. Частина інтерфейсу користувача складається з апаратних пристроїв, таких як клавіатура та відеодисплей. Інша частина інтерфейсу користувача залежить від того, як операційна система комп'ютера приймає команди від користувача. Протягом багатьох років єдиним способом взаємодії користувача з операційною системою був інтерфейс командного рядка, який зазвичай відображає підказку, і користувач вводить команду, яка потім виконується..

Графічний інтерфейс користувача (GUI), дозволяє користувачеві взаємодіяти з операційною системою та іншими програмами за допомогою графічних елементів на екрані. Графічні інтерфейси також популяризували використання миші як пристрою введення. Замість того, щоб вимагати від користувача вводити команди на клавіатурі, GUI дозволяє користувачеві вказувати на графічні елементи та натискати кнопку миші, щоб активувати їх.

### Програми GUI керуються подіями

У текстовому середовищі, такому як інтерфейс командного рядка, програми визначають порядок, у якому відбуваються події. Для прикладу розглянемо програму, яка обчислює площу прямокутника. Спочатку програма пропонує користувачеві ввести ширину прямокутника. Користувач вводить ширину, а потім програма пропонує користувачеві ввести довжину прямокутника. Користувач вводить довжину, потім програма обчислює площу. У користувача немає іншого вибору, окрім як вводити дані в тому порядку, у якому вони запитані [8].

Однак у середовищі GUI користувач визначає порядок, у якому відбуваються події. Наприклад, на рис. 6 показано програму графічного інтерфейсу користувача (написану мовою Python), яка обчислює площу прямокутника.

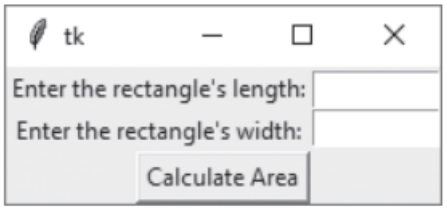


Рис. 6 Програма GUI для обчислення площі прямокутника

Користувач може ввести довжину та ширину в будь-якому бажаному порядку. Якщо допущено помилку, користувач може видалити введені дані та ввести їх повторно. Коли користувач готовий обчислити площу, він натискає кнопку Calculate Area, і програма виконує обчислення. Оскільки програми GUI повинні реагувати на дії користувача, кажуть, що вони керуються подіями. Користувач викликає події, такі як натискання кнопки, і програма повинна реагувати на події.

**Деякі Python GUI інструменти для розробників**

Python моє багато фреймворків для розробки GUI, і ми наведемо деякі з найпопулярніших:

1. **Tkinter**. Це модуль Python (поставляється з Python), який є інтерфейсом до інструментарію (бібліотеки) Tk GUI.
2. **PyQt**. Розроблений Riverbank Computing (https://riverbankcomputing.com/software/pyqt/intro). Пакет PyQt побудовано навколо фреймворку Qt, який є міжплатформенним фреймворком, який використовується для створення програм GUI для різних платформ. Пакет PyQt6 містить детальний набір прив'язок для Python на основі останньої версії v6 середовища розробки Qt.
3. **wxPython**. Розробник Robin Dunn. Сайт https://www.wxpython.org/ . wxPython, по суті, є модулем розширення Python, який діє як оболонка для wxWidgets API. wxPython дозволяє розробникам Python створювати власні інтерфейси користувача, які можна розгортати на таких платформах, як Windows, Mac OS, Linux і системи на базі Unix.
4. **Kivy**. Розроблений Kivy Organization. Сайт https://kivy.org/ . Це платформа з відкритим кодом для розробки GUI програм Python. Дозволяє створювати та розгортати міжплатформенні програми Python із графічним інтерфейсом. Kivy працює на Android, iOS, Linux, macOS і Windows.

   ПРИМІТКА. Фреймворк (англ. framework) – це програмне середовище (каркас, платформа), яке спрощує і прискорює розробку програм. Це готовий до використання комплекс програмних рішень, включаючи дизайн, логіку та базову функціональність системи.

## Тема 12. Модуль Tkinter. Огляд віджетів. Обробка подій

**Мета:** розглянути основні можливості модуля Tkinter; з'ясувати три компоненти програмування графічного інтерфейсу; вивчити варіанти розміщення віджетів та прив'язку подій; закріпити пройдений матеріал за допомогою вирішення навчального завдання.

**План**

1. Огляд можливостей модуля Tkinter.
2. Відображення тексту за допомогою віджетів Label.
3. Організація віджетів за допомогою Frames.
4. Віджети та діалогові вікна інформації
5. Отримання вводу за допомогою віджета Entry.
6. Використання Labels як полів виводу.

## Основні терміни і поняття

*Tkinter. Віджет. Управлінням геометрією (geometry management). Прив'язка подій. Зворотній виклик. Контейнер. Обробник подій (event handler).*

## Основні теоретичні положення

### Огляд можливостей модуля Tkinter

У Python ви можете використовувати модуль Tkinter для створення простих програм з графічним інтерфейсом.

Python не має функцій програмування графічного інтерфейсу, вбудованих у саму мову. Однак він поставляється з модулем під назвою Tkinter, який дозволяє створювати прості програми з графічним інтерфейсом. Назва "Tkinter" скорочено від "Tk interface". Це названо так, тому що це дає можливість програмістам Python використовувати бібліотеку графічного інтерфейсу під назвою Tk. Багато інших мов програмування також використовують бібліотеку Tk.

ПРИМІТКА. Для Python існує безліч бібліотек графічного інтерфейсу. Оскільки модуль tkinter поставляється з Python, ми будемо його використовувати.

Програма графічного інтерфейсу представляє вікно з різними графічними віджетами, з якими користувач може взаємодіяти або переглядати. Модуль tkinter містить 15 віджетів, описаних у Таблиці 8. Ми не розглянемо всі віджети tkinter у цьому розділі, але ми покажемо, як створювати прості програми з графічним інтерфейсом, які збирають вхідні дані та відображають ці дані.

Таблиця 8. Віджети модуля Tkinter

| **Widget** | **Опис** |
|---|---|
| Button | Кнопка, яка може викликати дію при її натисканні. |
| Canvas | Прямокутна область, яку можна використовувати для відображення графіки. |
| Checkbutton | Кнопка, яка може бути в положенні "on" або "off". |
| Entry | Область, у якій користувач може ввести один рядок введення з клавіатури. |
| Frame | Контейнер, який може містити інші віджети. |
| Label | Область, яка відображає один рядок тексту або зображення. |
| Listbox | Список, з якого користувач може вибрати елемент. |
| Menu | Список варіантів меню, які відображаються, коли користувач натискає на віджет Menubutton. |
| Menubutton | Меню, яке відображається на екрані та може бути натиснене користувачем. |
| Message | Відображає кілька рядків тексту. |
| Radiobutton | Віджет, який можна вибрати або скасувати. Віджети радіокнопок зазвичай з'являються в групах і дозволяють користувачеві вибрати один з кількох варіантів. |
| Scale | Віджет, що дозволяє користувачеві вибрати значення, перемістивши повзу- |

| | |
|---|---|
| | нок уздовж доріжки. |
| Scrollbar | Може використовуватися з деякими іншими типами віджетів для забезпечення можливості прокрутки. |
| Text | Віджет, що дозволяє користувачеві вводити кілька рядків введення тексту. |
| Toplevel | Контейнер, як і фрейм, але відображається у власному вікні. |

Найпростіша програма з графічним інтерфейсом, яку ми можемо продемонструвати, - це програма, яка відображає порожнє вікно. Лістинг 10 показує, як ми можемо це зробити за допомогою модуля tkinter. Під час запуску програми відображається вікно, зображене на Рис. 7. Щоб вийти з програми, просто натисніть стандартну кнопку закриття Windows (×) у верхньому правому куті вікна.

Лістинг 10. empty_window1.py

```
# This program displays an empty window.

import tkinter

def main():
    # Create the main window widget.
    main_window = tkinter.Tk()

   # Enter the tkinter main loop.
    tkinter.mainloop()

    # Call the main function.
if __name__ == '__main__':
    main()
```

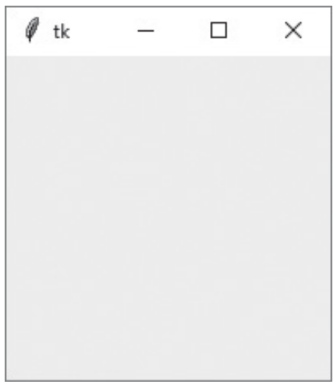


Рис. 7 Програма відображає вікно

Більшість програмістів вважають за краще використовувати об'єктно-орієнтований підхід під час написання програми з графічним інтерфейсом. Замість того, щоб писати функцію для створення екранних елементів програми, звичайною практикою є написання класу методом __init__, який створює графічний інтерфейс. Коли створюється екземпляр класу, на екрані з'являється графічний інтерфейс. Для демонстрації Лістинг 11 показує об’єктно-орієнтовану версію нашої програми, яка відображає порожнє вікно. Коли ця програма запускається, вона відображає вікно, зображене на Рис. 4.

Лістинг 11. Об’єктно-орієнтована версія empty_window2.py

```
# This program displays an empty window.
```

```
import tkinter

class MyGUI:
    def __init__(self):
        # Create the main window widget.
        self.main_window = tkinter.Tk()

        # Enter the tkinter main loop.
        tkinter.mainloop()

# Create an instance of the MyGUI class.
if __name__ == '__main__':
    my_gui = MyGUI()
```

Наступна діаграма показує три компоненти програмування графічного інтерфейсу:

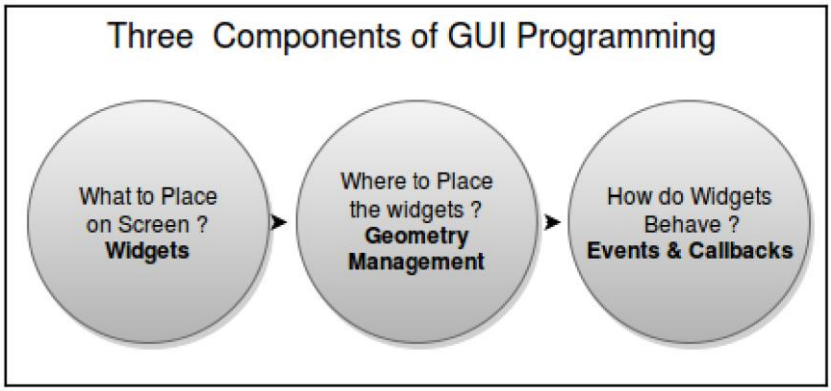


Рис. 8. Три аспекти GUI

Перш за все треба вирішити які компоненти (виджети) повинні з'являтися на екрані.

Потім необхідно розмістити віджети на екрані. Це включає вирішення положення та структурного планування різних компонентів. У Tkinter це називається управлінням геометрією (geometry management).

Після цього треба вирішити як компоненти взаємодіють і поводяться. Це передбачає додавання функціональності кожному компоненту. Кожен компонент або віджет щось робить. Наприклад, при натисканні на кнопку щось відбувається у відповідь. Смуга прокручування обробляє прокрутку, а прапорці та перемикачі дозволяють користувачам зробити певний вибір. У Tkinter функціональні можливості різних віджетів управляються прив'язкою команд або прив'язкою подій за допомогою зворотного виклику.

**Відображення тексту за допомогою віджетів Label**

Ви можете використовувати віджет Label для відображення одного рядка тексту у вікні. Щоб створити віджет Label, потрібно створити екземпляр класу Label модуля tkinter. Програма на Лістингу 12 створює вікно, що містить віджет Label, який відображає текст “Hello World!”. Вікно показано на Рис. 9.

Лістинг 12. hello_world.py

```
import tkinter

class MyGUI:
    def __init__(self):
    # Create the main window widget.
        self.main_window = tkinter.Tk()
        self.label = tkinter.Label(self.main_window,
                                   text='Hello World!')

        # Call the Label widget's pack method.
        self.label.pack()

        # Enter the tkinter main loop.
        tkinter.mainloop()

# Create an instance of the MyGUI class.
if _ _name_ _ == '_ _main_ _':
    my_gui = MyGUI()
```


Рис. 9 Вікно програми

Метод pack визначає, де має розміщуватися віджет, і робить його видимим під час відображення головного вікна. (Ви викликаєте метод pack для кожного віджета у вікні.)

**Організація віджетів за допомогою Frames**

Frame - це контейнер, який може містити інші віджети. Ви можете використовувати Frames для організації віджетів у вікні.

Наприклад, ви можете розмістити набір віджетів в одному фреймі та упорядкувати їх певним чином, а потім розмістити набір віджетів в іншому фреймі та розташувати їх по-іншому. Програма на Лістингу 13 це демонструє. Коли програма запускається, відображається вікно, зображене на Рис. 10.

Лістинг 13. frame_demo.py

```
import tkinter

class MyGUI:
    def __init__(self):
        self.main_window = tkinter.Tk()
        self.top_frame = tkinter.Frame(self.main_window)
        self.bottom_frame = tkinter.Frame(self.main_window)

        self.label1 = tkinter.Label(self.top_frame,
                                    text='Winken')
        self.label2 = tkinter.Label(self.top_frame,
                                    text='Blinken')
        self.label3 = tkinter.Label(self.top_frame,
                                    text='Nod')
```

```
        self.label1.pack(side='top')
        self.label2.pack(side='top')
        self.label3.pack(side='top')

        self.label4 = tkinter.Label(self.bottom_frame,
                                    text='Winken')
        self.label5 = tkinter.Label(self.bottom_frame,
                                    text='Blinken')
        self.label6 = tkinter.Label(self.bottom_frame,
                                    text='Nod')
        self.label4.pack(side='left')
        self.label5.pack(side='left')
        self.label6.pack(side='left')

        self.top_frame.pack()
        self.bottom_frame.pack()

        tkinter.mainloop()

if __name__ == '__main__':
    my_gui = MyGUI()
```

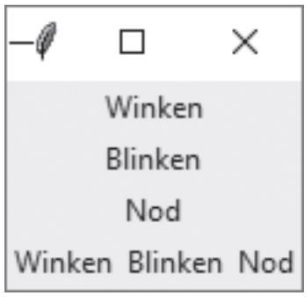


Рис. 10 Два фрейми

**Віджети та діалогові вікна інформації**

КОНЦЕПЦІЯ: Ви використовуєте віджет Button для створення стандартної кнопки у вікні. Коли користувач натискає кнопку, викликається певна функція або метод.

Інформаційне діалогове вікно - це просте вікно, яке відображає повідомлення користувачеві, і має кнопку OK, яка закриває діалогове вікно. Ви можете використовувати функцію showinfo модуля tkinter.messagebox для відображення діалогового вікна інформації.

Кнопка - це віджет, який користувач може натиснути, щоб викликати дію. Під час створення віджета кнопки можна вказати текст, який має відображатися на лицьовій стороні кнопки, та назву функції зворотного виклику. Функція зворотного виклику - це функція або метод, який виконується, коли користувач натискає кнопку.

ПРИМІТКА. Функція зворотного виклику також відома як обробник подій (event handler), оскільки вона обробляє подію, яка відбувається, коли користувач натискає кнопку.

Для демонстрації ми розглянемо програму на Лістингу 14. Ця програма відображає вікно, зображене на Рис. 11. Коли користувач натискає кнопку, про-

грама відображає окреме інформаційне діалогове вікно, показане на Рис. 12. Для відображення діалогового вікна інформації ми використовуємо функцію showinfo, яка знаходиться в модулі tkinter.messagebox. (Щоб використовувати функцію showinfo, вам потрібно імпортувати модуль tkinter.messagebox.) Це загальний формат виклику функції showinfo:

```
tkinter.messagebox.showinfo(title, message)
```

У загальному форматі `title` - це рядок, який відображається у рядку заголовка діалогового вікна, а `message` - інформаційний рядок, який відображається у головній частині діалогового вікна.

Лістинг 14. button_demo.py

```
import tkinter
import tkinter.messagebox

class MyGUI:
    def __init__(self):
        self.main_window = tkinter.Tk()
        self.my_button = tkinter.Button(self.main_window,
             text='Click Me!', command=self.do_something)
        self.my_button.pack()
        tkinter.mainloop()

    def do_something(self):
        tkinter.messagebox.showinfo('Response',
                    'Thanks for clicking the button.')

if __name__ == '__main__':
    my_gui = MyGUI()
```


Рис. 11 Вікно програми

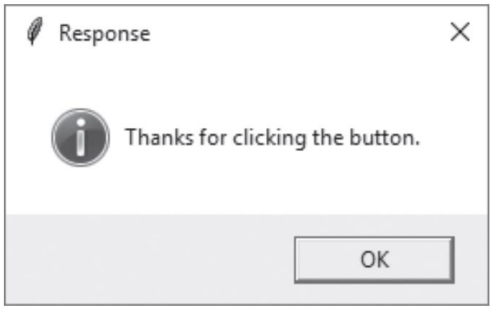


Рис. 12 Інформаційне діалогове вікно

**Отримання вводу за допомогою віджета Entry**

КОНЦЕПЦІЯ: Віджет Entry - це прямокутна область, у яку користувач може вводити дані. Ви використовуєте метод get віджета Entry для отримання даних, які були набрані у віджеті.

Віджет Entry - це прямокутна область, в яку користувач може вводити текст. Віджети входу використовуються для збору вхідних даних у програмі графічного інтерфейсу. Як правило, програма матиме один або кілька віджетів

Entry у вікні разом із кнопкою, яку користувач натискає, щоб надіслати дані, які він ввів у віджети Entry. Функція зворотного виклику кнопки витягує дані з віджетів входу вікна та обробляє їх.

Ви використовуєте метод get віджета Entry для отримання даних, які користувач ввів у віджет. Метод get повертає рядок, тому його доведеться конвертувати у відповідний тип даних, якщо віджет Entry використовується для числового введення.

Для демонстрації ми розглянемо програму, яка дозволяє користувачеві ввести відстань у кілометрах у віджет Entry, а потім натиснути кнопку, щоб побачити цю відстань, перетворену в милі. Формула перетворення кілометрів у милі така:

```
Miles = Kilometers * 0.6214
```

На Рис. 13 показано вікно, яке відображає програма. Щоб розташувати віджети у положеннях, показаних на малюнку, ми організуємо їх у два фрейми, як показано на Рис. 14. Мітка, яка відображає запит та віджет Entry, зберігатиметься у top_frame, а їх методи pack будуть викликані з аргументом side = 'left'. Це призведе до їх горизонтального відображення в фреймі. Кнопки Convert та Quit зберігатимуться у bottom_frame, а також їх методи pack також будуть викликатися з аргументом side = 'left'.

Програма на Лістингу 15 показує код програми. На Рис. 15 показано, що відбувається, коли користувач вводить 1000 у віджет Entry і натискає кнопку Convert.

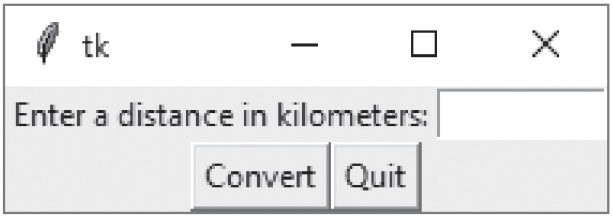


Рис. 13 Вікно програми

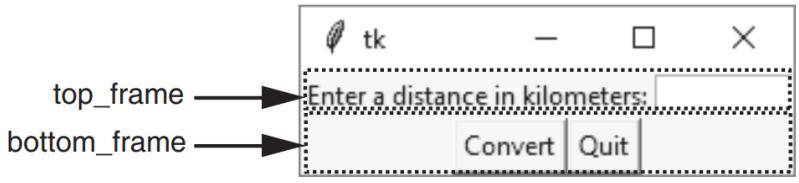


Рис. 14. Розташування віджетів

Лістинг 15. kilo_converter.py

```
import tkinter
import tkinter.messagebox

class KiloConverterGUI:
    def __init__(self):
        self.main_window = tkinter.Tk()
        self.top_frame = tkinter.Frame(self.main_window)
        self.bottom_frame = tkinter.Frame(self.main_window)

        self.prompt_label = tkinter.Label(self.top_frame,
                    text='Enter a distance in kilometers:')
```

```
        self.kilo_entry = tkinter.Entry(self.top_frame,
                                        width=10)
        self.prompt_label.pack(side='left')
        self.kilo_entry.pack(side='left')

        self.calc_button = tkinter.Button(self.bottom_frame,
                     text='Convert', command=self.convert)
        self.quit_button = tkinter.Button(self.bottom_frame
            , text='Quit', command=self.main_window.destroy)
        self.calc_button.pack(side='left')
        self.quit_button.pack(side='left')

        self.top_frame.pack()
        self.bottom_frame.pack()

        tkinter.mainloop()

    def convert(self):
        kilo = float(self.kilo_entry.get())
        miles = kilo * 0.6214
        tkinter.messagebox.showinfo('Results', str(kilo) +
         ' kilometers is equal to ' + str(miles) + ' miles.')

if __name__ == '__main__':
    kilo_conv = KiloConverterGUI()
```

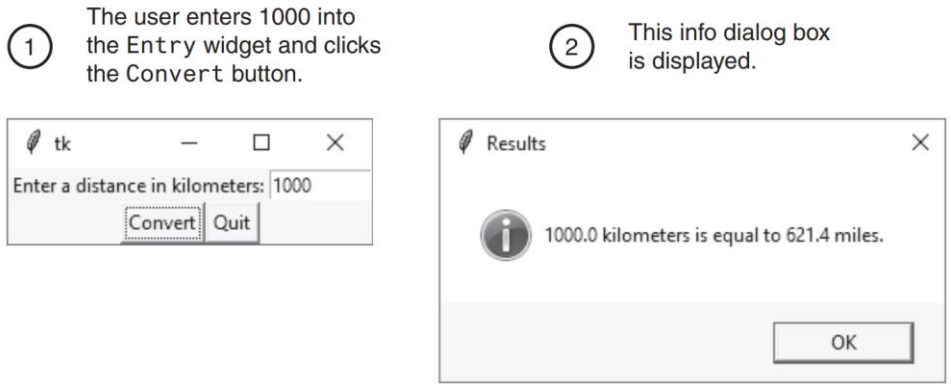


Рис. 15 Результат в інформаційній панелі

**Використання Labels як полів виводу**

КОНЦЕПЦІЯ: Коли об’єкт StringVar асоціюється з віджетом Label, віджет Label відображає будь-які дані, що зберігаються в об’єкті StringVar.

Раніше ви бачили, як використовувати інформаційне діалогове вікно для відображення результатів. Якщо ви не хочете відображати окреме діалогове вікно для виведення програми, ви можете використовувати віджети Label у головному вікні програми для динамічного відображення результатів. Ви просто створюєте порожні віджети Label у своєму головному вікні, а потім вводите код, який відображає потрібні дані у цих мітках при натисканні кнопки.

Модуль tkinter надає клас з назвою StringVar, який можна використовувати разом з віджетом Label для відображення даних. По-перше, ви створюєте

об'єкт StringVar. Потім ви створюєте віджет Label і пов'язуєте його з об'єктом StringVar. З цього моменту будь-яке значення, яке потім зберігається в об’єкті StringVar, автоматично відображатиметься у віджеті Label.

Програма на Лістингу 16 демонструє, як це зробити. Це модифікована версія програми kilo_converter, яку ви бачили на Лістигу 15. Замість того, щоб спливати інформаційне діалогове вікно, ця версія програми відображає кількість миль на мітці у головному вікні.

Лістинг 16. kilo_converter2.py

```
import tkinter

class KiloConverterGUI:
    def __init__(self):
        self.main_window = tkinter.Tk()
        self.top_frame = tkinter.Frame()
        self.mid_frame = tkinter.Frame()
        self.bottom_frame = tkinter.Frame()

        self.prompt_label = tkinter.Label(self.top_frame,
                  text='Enter a distance in kilometers:')
        self.kilo_entry = tkinter.Entry(self.top_frame,
                                        width=10)
        self.prompt_label.pack(side='left')
        self.kilo_entry.pack(side='left')

        self.descr_label = tkinter.Label(self.mid_frame,
                                text='Converted to miles:')
        self.value = tkinter.StringVar()
        self.miles_label = tkinter.Label(self.mid_frame,
                                   textvariable=self.value)
        self.descr_label.pack(side='left')
        self.miles_label.pack(side='left')

        self.calc_button = tkinter.Button(self.bottom_frame,
                        text='Convert', command=self.convert)
        self.quit_button = tkinter.Button(self.bottom_frame,
              text='Quit', command=self.main_window.destroy)
        self.calc_button.pack(side='left')
        self.quit_button.pack(side='left')

        self.top_frame.pack()
        self.mid_frame.pack()
        self.bottom_frame.pack()

        tkinter.mainloop()

    def convert(self):
        kilo = float(self.kilo_entry.get())
        miles = kilo * 0.6214
        self.value.set(miles)

if _ _name_ _ == '_ _main_ _':
```

```
kilo_conv = KiloConverterGUI()
```

Коли ця програма працює, вона відображає вікно, зображене на Рис. 16. На Рис. 17 показано, що відбувається, коли користувач вводить 1000 за кілометри і натискає кнопку Convert. Кількість миль відображається на мітці у головному вікні.

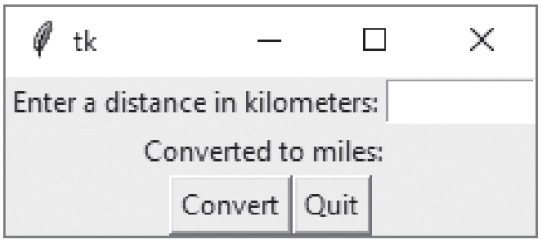


Рис. 16 Вікно модифікованої програми

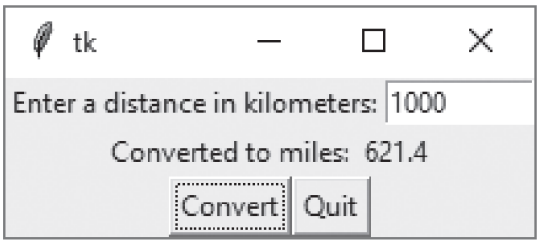


Рис. 17 Після введення 1000 і натискання кнопки Convert

**Питання для самоконтролю**

1. Поясніть назву модуля Tkinter.
2. Що таке віджет?
3. Які дії виконує найпростіша програма з графічним інтерфейсом?
4. Назвіть три компоненти програмування графічного інтерфейсу.
5. Що таке прив'язка подій за допомогою зворотного виклику?
6. Як ввести данні в програму графічного інтерфейсу?
7. Які є варіанти виведення інформації в програмі з GUI?

**Тестові завдання для перевірки знань**

1. Ці типи програм керуються подіями.

A. command line
B. text-based
C. GUI
D. procedural

2. Елемент, який відображається в графічному інтерфейсі користувача програми, називається _____.

A. gadget
B. widget
C. tool
D. iconified object

3. Цей віджет являє собою область, яка відображає один рядок тексту.
A. Label
B. Entry
C. TextLine
D. Canvas

4. Цей метод розташовує віджет у належному положенні, і він робить його видимим, коли відображається головне вікно.
A. arrange
B. pack
C. position
D. show

5. _____ — це функція або метод, який викликається, коли відбувається певна подія.
A. auto function
B. startup function
C. callback function
D. exception

# РОЗДІЛ 6 БІБЛІОТЕКИ NUMPY, MATPLOTLIB ДЛЯ НАУКОВИХ ОБЧИСЛЕНЬ. БІБЛІОТЕКА PYGAME

## Тема 13. Бібліотеки NumPy, Matplotlib, Seaborn

**Мета:** розглянути основні можливості бібліотеки NumPy для наукових обчислень; з’ясувати в чому полягають переваги спеціальних бібліотек в порівнянні зі стандартним Python; ознайомитись з основами роботи з бібліотеками Seaborn та Matplotlib; виконати навчальне завдання.

**План**

1. Бібліотека NumPy.
2. Об'єкт масиву NumPy.
3. Створення та ініціалізація масиву.
4. Читання та запис масиву у файл.
5. Робота з матрицями.
6. Бібліотека Matplotlib.
7. Робота з Seaborn.

## Основні терміни і поняття

*Векторизовані обчислення. Бібліотека NumPy. Об'єкт масиву NumPy. Атрибути класу ndarray. Універсальні функції (ufuncs). Пакет linalg. Бібліотека SciPy. Бібліотека Matplotlib. Бібліотека Seaborn.*

## Основні теоретичні положення

### Бібліотека NumPy

Вектори, матриці та багатовимірні масиви є важливими інструментами чисельних обчислень. Коли обчислення необхідно повторити для набору вхідних значень, природно і вигідно представляти дані у вигляді масивів, а обчислення - з точки зору операцій з масивами. Обчислення, сформульовані таким чином, називаються *векторизованими*. Векторизоване обчислення усуває необхідність у багатьох явних циклах над елементами масиву шляхом застосування пакетних операцій до даних масиву. Результатом є стислий і більш підтримуваний код, який дозволяє делегувати реалізацію (наприклад, поелементних операцій з масивами) до більш ефективних бібліотек низького рівня.

Наприклад, порівняйте множення двох одновимірних списків з n чисел, a і b, в ядрі мови Python:

```
c = []
for i in range(n):
    c.append(a[i] * b[i])
```

та за допомогою масивів NumPy:

```
c = a * b
```

Тому векторизовані обчислення можуть бути значно швидшими, ніж послідовні обчислення по елементах. Це особливо важливо в інтерпретованій мові, такій як Python, де цикл по елементах тягне за собою значні накладні витрати на продуктивність [12].

У науковому обчислювальному середовищі Python ефективні структури даних для роботи з масивами надаються бібліотекою NumPy. Ядро NumPy реалізовано на C і забезпечує ефективні функції для маніпулювання та обробки масивів. На перший погляд, масиви NumPy мають деяку схожість зі структурою даних списку Python. Але важливою відмінністю є те, що хоча списки Python є загальними контейнерами об’єктів, масиви NumPy є однорідними та типізованими масивами фіксованого розміру. Однорідний означає, що всі елементи в масиві мають однаковий тип даних. Фіксований розмір означає, що розмір масиву неможливо змінити (без створення нового масиву). З цих та інших причин операції та функції, що діють на масивах NumPy, можуть бути набагато ефективнішими, ніж ті, що використовують списки Python.

NumPy - це основна бібліотека для наукових обчислень на Python. Вона надає високопродуктивний багатовимірний об’єкт масиву та інструменти для роботи з цими масивами. У Python ми маємо списки, які служать цілям масивів, але вони повільно обробляються. NumPy має на меті надати об’єкт масиву, який до 50 разів швидше традиційних списків Python. Об'єкт масиву в NumPy називається ndarray, він надає багато допоміжних функцій, які роблять роботу з ndarray дуже легкою. Масиви дуже часто використовуються в науці про дані, де швидкість та ресурси дуже важливі. Більше інформації про NumPy можна знайти на веб-сайті www.numpy.org.

Ви можете встановити NumPy через командний рядок за допомогою такої команди:

```
pip install numpy
```

***Імпорт модулів***

Для того, щоб використовувати бібліотеку NumPy, нам потрібно імпортувати її до нашої програми. За домовленістю, модуль numpy імпортують під псевдонімом np:

```
import numpy as np
```

Зокрема, ця угода використовується в документації NumPy і в ширшій науковій екосистемі Python (SciPy, Pandas і так далі). Після цього ми можемо отримати доступ до функцій і класів у модулі numpy за допомогою простору імен np. У цьому розділі ми припускаємо, що модуль NumPy імпортується таким чином.

**Об'єкт масиву NumPy**

Ядром бібліотеки NumPy є структури даних для представлення багатовимірних масивів однорідних даних (з однаковим типом даних). Основною структурою даних для багатовимірних масивів у NumPy є клас ***ndarray***. Окрім даних, що зберігаються в масиві, ця структура даних також містить важливі метадані про масив, такі як його форма, розмір, тип даних та інші атрибути. Більш детальний опис цих атрибутів дивись у Таблиці 9. Повний список атрибутів з описами доступний у рядку docstring ndarray, доступ до якого можна отримати, викликавши довідку (np.ndarray) у інтерпретаторі Python.

Таблиця 9. Основні атрибути класу ndarray

| Атрибут | Опис |
|---|---|
| shape | Кортеж, який містить кількість елементів (тобто довжину) для кожного виміру (осі) масиву. |
| size | Загальна кількість елементів у масиві. |
| ndim | Кількість розмірів (осей). |
| nbytes | Кількість байтів, використаних для зберігання даних. |
| data | "Буфер" в пам'яті, що містить фактичні елементи масиву |
| dtype | Тип даних елементів у масиві. |

**Створення та ініціалізація масиву**

***Базовий метод створення масиву***

Найпростіший спосіб створити невеликий масив NumPy - викликати функцію np.array зі списком або кортежем значень:

```
>>> array1d = np.array([1, 2, 3, 4])
>>> print(array1d)
[1 2 3 4]
>>> print(type(array1d))
<class 'numpy.ndarray'>
>>> print(array1d.shape)
(4,)
>>> b = np.array( [[1.,2.], [3.,4.]] )
>>> b
array([[1., 2.],
       [3., 4.]])
>>>
```

Індексування багатовимірного масиву NumPy дещо відрізняється від індексування звичайного списку списків Python: замість b [i][j] звертайтесь до індексу необхідного елемента як кортежа цілих чисел, b [i, j]:

```
>>> b[0,1]   # same as b[(0,1)]
2.0
>>> b[1,1] = 0.  # also for assignment
>>> b
array([[1., 2.],
       [3., 0.]])
>>>
```

Якщо ваш масив великий або ви не знаєте значення елементів під час створення, існує кілька способів оголосити масив певної форми, заповнений стандартними або довільними значеннями. Найпростіший і найшвидший, np.empty, приймає кортеж форми масиву і створює масив без ініціалізації його елементів: значення початкових елементів невизначені (як правило, випадкове сміття, взяте з будь-якого вмісту пам'яті, виділеного Python для масиву).

```
>>> np.empty((3,3))
array([[0.00e+000, 0.00e+000, 0.00e+000],
       [0.00e+000, 0.00e+000, 3.22e-321],
       [0.00e+000, 0.00e+000, 0.00e+000]])
>>>
```

Існують також допоміжні методи np.zeros та np.ones, які створюють масив зазначеної форми з елементами, попередньо заповненими 0 та 1 відповідно. np.empty, np.zeros і np.ones також беруть необов'язковий аргумент dtype.

```
>>> np.zeros((3,2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> np.ones((3,3), dtype=int)
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
>>>
```

***Ініціалізація масиву з послідовності***

Для створення масиву, що містить послідовність чисел, є два методи: np.arange та np.linspace. np.arange є еквівалентом range NumPy, за винятком того, що він може генерувати послідовності з плаваючою точкою. Він також фактично виділяє пам'ять для елементів у ndarray замість повернення об'єкта, схожого на генератор.

```
>>> np.arange(7)
array([0, 1, 2, 3, 4, 5, 6])
>>> np.arange(1.5, 3., 0.5)
array([1.5, 2. , 2.5])
>>>
```

Як і range, масив, створений у цих прикладах, не включає останні елементи, 7 та 3. Однак у arange є проблема: через скінченну точність арифметики з плаваючою точкою не завжди можна дізнатися, скільки елементів буде створено. З цієї причини, а також тому, що часто потрібен останній елемент заданої послідовності, функція np.linspace може бути більш корисним способом створення послідовності. Наприклад, для створення рівномірно розподіленого масиву з п’яти чисел від 1 до 20 включно:

```
>>> np.linspace(1, 20, 5)
array([ 1., 5.75, 10.5, 15.25, 20.])
>>>
```

np.linspace містить пару необов'язкових boolean аргументів. По-перше, якщо для параметра *retstep* встановлено значення True, повертається інтервал між значеннями (розмір кроку):

```
>>> x, dx = np.linspace(0., 2*np.pi, 100, retstep=True)
>>> dx
0.06346651825433926
>>>
```

Це позбавить вас від окремого обчислення dx = (end-start)/(num-1); у цьому прикладі 100 точок між 0 і 2π включно рознесені на 2π /99 = 0.0634665… Нарешті, встановлення *endpoint* на False опускає кінцеву точку в послідовності, як для np.arange:

```
>>> x = np.linspace(0, 5, 5, endpoint=False)
>>> x
array([0., 1., 2., 3., 4.])
```

Зауважте, що масив, створений np.linspace, має dtype для чисел з плаваючою точкою, навіть якщо послідовність генерує цілі числа.

***Ініціалізація масиву з функції***

Для створення масиву, ініціалізованого значеннями, обчисленими за допомогою функції, використовуйте NumPy метод np.fromfunction, який бере в якості аргументів функцію та кортеж, що представляє форму потрібного масиву. Сама функція повинна приймати таку саму кількість аргументів, що і розміри в масиві: ці аргументи індексують кожен елемент, для якого функція повертає значення. Приклад зробить це більш зрозумілим:

```
>>> def f(i, j):
        return 2 * i * j

>>> np.fromfunction(f,(4,3))
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  2.,  4.],
       [ 0.,  4.,  8.],
       [ 0.,  6., 12.]])
>>>
```

Функція f викликається для кожного індексу у зазначеній формі, і значення, які вона повертає, використовуються для ініціалізації відповідних елементів. В попередньому прикладі при бажанні можна використати анонімну лямбда-функцію:

```
np.fromfunction(lambda i,j: 2*i*j, (4,3))
```

***Універсальні функції (ufuncs)***

На додаток до основних арифметичних операцій додавання, ділення тощо, NumPy надає багато знайомих математичних функцій, які виконує math модуль, реалізованих у вигляді так званих універсальних функцій, які діють на кожен елемент масиву, створюючи масив без необхідність явного циклу. Універсальні функції - це спосіб векторизації NumPy, що сприяє чистому, ефективному та простому у обслуговуванні коду. Наприклад,

```
>>> x = np.linspace(1, 5, 5)
>>> x**2
array([ 1.,  4.,  9., 16., 25.])
>>> x - 1
array([0., 1., 2., 3., 4.])
>>> np.sqrt(x - 1)
array([0. , 1. , 1.41421356, 1.73205081, 2. ])
>>> y = np.exp(-np.linspace(0., 2., 5))
>>> np.sin(x - y)
array([0., 0.98431873, 0.48771645, -0.59340065, -0.98842844])
>>>
```

Множення масивів відбувається поелементно (elementwise): множення матриць реалізується оператором @ або функцією dot NumPy:

```
>>> a = np.array( ((1, 2), (3, 4)) )
>>> b = a
>>> a * b  # elementwise multiplication
array([[ 1,  4],
       [ 9, 16]])
>>> a @ b  # matrix multiplication; also a.dot(b) or np.dot(a, b)
array([[ 7, 10],
       [15, 22]])
```

Оператори порівняння та логіки (~, & та | для not, and та or, відповідно) також векторизовані і призводять до масивів булевих значень:

```
>>> a = np.linspace(1, 6, 6)**3
>>> print(a)
[  1.   8.  27.  64. 125. 216.]
>>> print(a > 100)
[False False False False  True  True]
>>> print((a < 10) | (a > 100))
[ True  True False False  True  True]
```

***Спеціальні значення NumPy, nan та inf***

NumPy визначає два особливих значення для представлення результатів обчислень, які не визначені математично або не кінцеві. Значення np.nan ("Не число", NaN) представляє результат обчислення, яке не є чітко визначеною математичною операцією (наприклад, 0/0); np.inf представляє нескінченність. Наприклад,

```
>>> a = np.arange(4, dtype='f8')
>>> a /= 0
... RuntimeWarning: invalid value encountered in true_divide ...
... RuntimeWarning: divide by zero encountered in true_divide ...
>>> a
array([nan, inf, inf, inf])
```

**Приклад.** Магічний квадрат - це сітка чисел N × N, в якій сума чисел в кожному рядку, стовпці та основній діагоналі складають однакову величину (рівну $N (N^2 + 1) /2$).

Спосіб побудови магічного квадрата для непарного N виглядає наступним чином:

Крок 1. Почніть із середини верхнього ряду і нехай n = 1.

Крок 2. Вставте n у поточне положення сітки.

Крок 3. Якщо $n = N^2$ сітка завершена, зупиніться. В іншому випадку приріст n.

Крок 4. Переміщення по діагоналі вгору і вправо, перенесення до першого стовпця або останнього рядка, якщо переміщення веде поза сітки. Якщо ця клітинка вже заповнена, перемістіться вертикально вниз на один пробіл.

Крок 5. Поверніться до кроку 2.

Наступна програма створює та відображає магічний квадрат [13].

```
# Create an N x N magic square. N must be odd.
import numpy as np
N = 5
magic_square = np.zeros((N, N), dtype=int)
n = 1
i, j = 0, N//2
while n <= N**2:
    magic_square[i, j] = n
    n += 1
    newi , newj = (i - 1) % N, (j + 1)% N
    if magic_square[newi , newj]:
        i += 1
    else:
        i, j = newi , newj
print(magic_square)
```

На виході отримаємо магічний квадрат 5 × 5:

```
[[17 24  1  8 15]
 [23  5  7 14 16]
 [ 4  6 13 20 22]
 [10 12 19 21  3]
 [11 18 25  2  9]]
```

**Читання та запис масиву у файл**

Наукові дані часто зчитуються з текстового файлу, який може містити коментарі, відсутні значення та порожні рядки. Стовпці значень можуть бути вирівняні у форматі фіксованої ширини або розділені одним або кількома символами-розмежувачами (наприклад, пробілами, табуляціями чи комами). Крім того, у файлі може бути описовий заголовок і навіть виноски, що ускладнює аналіз безпосередньо за допомогою рядкових методів Python.

NumPy надає кілька функцій для зчитування даних з текстового файлу. Найпростіша, np.loadtxt, обробляє багато поширених випадків; більш складна np.genfromtxt дозволяє краще обробляти відсутні значення та колонтитули.

***np.save та np.load***

Існує незалежний від платформи двійковий формат для збереження масиву NumPy. Команда

```
np.save('my-array.npy', a)
```

збереже масив `a` у двійковому файлі my-array.npy (розширення .npy додається, якщо воно не надається). Потім масив можна завантажити за допомогою NumPy на будь-якій іншій операційній системі

```
a = np.load('my-array.npy')
```

***np.loadtxt***

Прототип методу np.loadtxt має вигляд:

```
np.loadtxt(fname , dtype=<class 'float'>, comments='#',
           delimiter=None , converters=None , skiprows=0,
           usecols=None , unpack=False , ndmin=0)
```

Аргументи такі:

• fname: Єдиний необхідний аргумент, який може бути іменем файлу, відкритим файлом або генератором, що повертає рядки даних для аналізу.

• dtype: Тип даних масиву (за замовчуванням плаваючий).

• comments: Коментарі у файлі зазвичай починаються з якогось символу, наприклад # (як у Python) або %. За замовчуванням він встановлений на #.

• delimiter: рядок, що використовується для розділення стовпців даних у файлі; за замовчуванням це значення None, тобто будь-яка кількість пробілів (пробілів, табуляцій) розмежовує дані. Щоб прочитати файл, розділений комами (csv), встановіть delimiter= ’,’.

• converters: необов’язковий словник, що відображає індекс стовпця у функцію, що перетворює значення рядків у цьому стовпці в дані (наприклад, з плаваючою точкою).

• skiprows: ціле число, що дає кількість рядків на початку файлу для пропуску перед читанням даних (наприклад, для проходження рядків заголовка). За замовчуванням 0 (без заголовка).

• usecols: послідовність індексів стовпців, що визначає, які стовпці файлу повертати як дані; за замовчуванням це значення None, тобто всі стовпці будуть проаналізовані та повернуті.

• unpack: за замовчуванням таблиця даних повертається в одному масиві рядків і стовпців, що відображає структуру прочитаного файлу. Установка un-

pack = True буде транспонувати цей масив так, що окремі стовпці можна вибирати та призначати різним змінним.

• ndmin: Мінімальна кількість розмірів, які має мати повернутий масив. За замовчуванням 0 (тому файл, що містить один номер, читається як скаляр); його також можна встановити на 1 або 2.

**Приклад**. Збережіть масив [[1.20, 2.20, 3.00], [4.14, 5.65, 6.42]] у файлі з назвою my_array.txt і прочитайте його назад до змінної my_arr.

```
import numpy as np
arr = np.array([[1.20, 2.20, 3.00], [4.14, 5.65, 6.42]])
np.savetxt("my_arr.txt", arr, fmt="%.2f",\
           header = "Col1 Col2 Col3")
my_arr = np.loadtxt("my_arr.txt")
print(my_arr)
```

**Робота з матрицями**

Масиви NumPy також служать матрицями, які є фундаментальними в математиці та обчислювальному програмуванні. Матриця - це просто двовимірний масив.

Одинична матриця (розміру n) — це матриця n × n, де (i, i)-й елемент дорівнює 1, а (i, j)-ий елемент дорівнює нулю для i ≠ j. Існує функція створення масиву, яка дає одиничну матрицю n × n для заданого значення n:

```
np.eye(3)
# array([[1., 0., 0.],
#        [0., 1., 0.],
#        [0., 0., 1.]])
```

***Транспонування матриці***

Масиви NumPy можна легко транспонувати, викликавши метод transpose для об’єкта масиву. Насправді, оскільки це така поширена операція, масиви мають зручну властивість T, яка повертає транспоновану матрицю. Транспонування змінює порядок форми матриці (масиву), так що рядки стають стовпцями, а стовпці — рядками.

```
mat = np.array([[1, 2], [3, 4]])
mat.transpose()
# array([[1, 3],
#        [2, 4]])
mat.T
# array([[1, 3],
#        [2, 4]])
```

***Матричне множення***

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[-1, 1],[0, 1]])
A @ B
# array([[-1, 3],
#        [-3, 7]])
A * B # different from A @ B
# array([[-1, 2],
#        [ 0, 4]])
```

***Детермінант і обернена матриця***

Підпрограма NumPy для обчислення визначника матриці міститься в окремому модулі, який називається linalg. Цей модуль містить багато загальних процедур для лінійної алгебри, яка є розділом математики, що охоплює векторну та матричну алгебру. Підпрограмою для обчислення визначника квадратної матриці є програма det:

```
from numpy import linalg
linalg.det(A) # -2.0000000000000004
```

Оберненою до n × n матриці A є (обов'язково єдина) n × n матриця B, така що AB = BA = I, де I позначає одиничну n × n матрицю, а множення, що виконується тут, є множенням матриць.

Підпрограма inv з модуля linalg обчислює обернену матрицю, якщо вона існує:

```
linalg.inv(A)
# array([[-2. , 1. ],
#        [ 1.5, -0.5]])
```

Ми можемо перевірити чи справді матриця, обчислена підпрограмою inv, є матрицею, оберненою до A, помноживши матрицю A (з будь-якого боку) на обернену і перевіривши, що ми отримуємо ідентичну матрицю 2 × 2:

```
Ainv = linalg.inv(A)
Ainv @ A
# Approximately
# array([[1., 0.],
#        [0., 1.]])
A @ Ainv
# Approximately
# array([[1., 0.],
#        [0., 1.]])
```

Пакет linalg також містить ряд інших методів, таких як norm, який обчислює різні норми матриці. Він також містить функції для розкладання матриць різними способами та розв'язування систем рівнянь.

***Системи рівнянь***

Для ілюстрації методики розв'яжемо наприклад таку систему рівнянь:

$$\begin{aligned} 3x_1 - 2x_2 + x_3 &= 7 \\ x_1 + x_2 - 2x_3 &= -4 \\ -3x_1 - 2x_2 + x_3 &= 1 \end{aligned}$$

Ця система рівнянь має три невідомі значення: $x_1$, $x_2$ і $x_3$. Спочатку створюємо матрицю коефіцієнтів A і вектор b. Оскільки ми використовуємо NumPy як наш засіб роботи з матрицями та векторами, ми створюємо двовимірний масив NumPy для матриці A та одновимірний масив для b:

```
import numpy as np
from numpy import linalg
A = np.array([[3, -2, 1], [1, 1, -2], [-3, -2, 1]])
b = np.array([7, -4, 1])
```

Тепер рішення системи рівнянь можна знайти за допомогою процедури solve:

```
linalg.solve(A, b) # array([ 1., -1., 2.])
```

***Чисельне розв'язування простих диференціальних рівнянь***

Диференціальні рівняння виникають у ситуаціях, коли величина розвивається, як правило, з часом відповідно до заданого співвідношення. Вони надзвичайно поширені в інженерії та фізиці і з'являються цілком природно. Одним із класичних прикладів (дуже простого) диференціального рівняння є закон охолодження, розроблений Ньютоном. Температура тіла охолоджується зі швидкістю, пропорційною поточній температурі. Математично це означає, що ми можемо записати похідну температури T тіла в момент t > 0 за допомогою диференціального рівняння

$$\frac{dT}{dt} = -kT$$

де k – додатна константа, що визначає швидкість охолодження. Це диференціальне рівняння можна розв'язати аналітично, спочатку «розділивши змінні», а потім інтегрувавши та перегрупувавши. Після виконання цієї процедури отримуємо загальне рішення

$$T(t) = T_0 e^{-kt}$$

де $T_0$ - початкова температура.

У цьому підрозділі ми розв'яжемо просте звичайне диференціальне рівняння чисельно, використовуючи підпрограму solve_ivp від SciPy.

Ми продемонструємо техніку чисельного розв'язування диференціального рівняння в Python за допомогою рівняння охолодження, описаного раніше, оскільки в цьому випадку ми можемо обчислити істинне рішення. Приймемо початкову температуру $T_0 = 50$ і $k = 0{,}2$. Давайте також знайдемо рішення для значень t від 0 до 5.

Загальне (першого порядку) диференціальне рівняння має вигляд

$$\frac{dy}{dt} = f(t, y)$$

де f деяка функція від t (незалежна змінна) і y (залежна змінна). У цій формулі T є залежною змінною, а f(t, T) = -kt. Підпрограми для розв'язування диференціальних рівнянь у пакеті SciPy вимагають функції f і початкового значення $y_0$ і діапазону значень t, де нам потрібно обчислити рішення. Для початку нам потрібно визначити нашу функцію f у Python і створити діапазон змінних $y_0$ і t, готовий для надання в підпрограму SciPy:

```
def f(t, y):
    return -0.2*y
t_range = (0, 5)
```

Далі нам потрібно визначити початкову умову, з якої слід знайти рішення. З технічних причин початкові значення у повинні бути вказані як одновимірний масив NumPy:

```
T0 = np.array([50.])
```

Оскільки в цьому випадку ми вже знаємо точне рішення, ми також можемо визначити його в Python, готове до порівняння з чисельним рішенням, яке ми будемо обчислювати:

```
def true_solution(t):
    return 50.*np.exp(-0.2*t)
```

Ми використовуємо процедуру solve_ivp з модуля integrate в SciPy для чисельного розв'язання диференціального рівняння. Ми додаємо параметр для максимального розміру кроку зі значенням 0,1, тому розв'язок обчислюється з розумною кількістю точок.

Далі ми витягуємо значення рішення з об'єкта sol, поверненого методом solve_ivp.

Оскільки ми також збираємося побудувати помилку апроксимації на тому самому малюнку, ми створюємо два підграфіки, використовуючи процедуру subplots.

В результаті отримаємо таку програму:

```
import numpy as np
from scipy import integrate
import matplotlib.pyplot as plt

def f(t, y):
    return -0.2*y

t_range = (0, 5)

T0 = np.array([50.])

def true_solution(t):
    return 50.*np.exp(-0.2*t)

sol = integrate.solve_ivp(f, t_range, T0, max_step=0.1)

t_vals = sol.t
T_vals = sol.y[0, :]

fig, (ax1, ax2) = plt.subplots(1, 2, tight_layout=True)

ax1.plot(t_vals, T_vals)
ax1.set_xlabel("$t$")
ax1.set_ylabel("$T$")
ax1.set_title("Solution of the cooling equation")

err = np.abs(T_vals - true_solution(t_vals))
ax2.semilogy(t_vals, err)
ax2.set_xlabel("$t$")
ax2.set_ylabel("Error")
ax2.set_title("Error in approximation")

plt.show()
```

Виконавши програму, отримаємо наступні графіки:

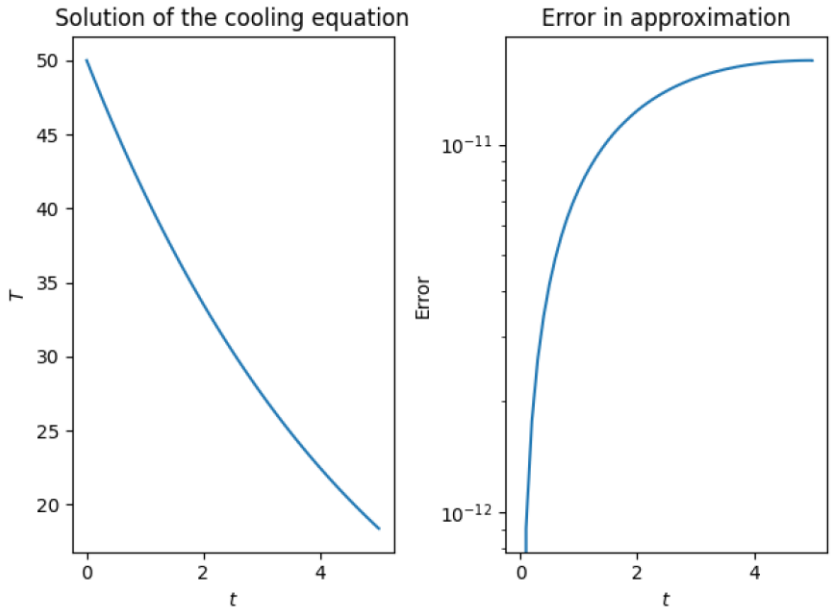


Рис. 18 Графік чисельного рішення

Лівосторонній графік на Рис. 18 показує зниження температури з часом, тоді як правий графік показує, що помилка збільшується, коли ми віддаляємося від відомого значення, заданого початковою умовою.

**Бібліотека Matplotlib**

Matplotlib - це найпростіша бібліотека для графічної візуалізації даних у Python. Вона включає майже всі види графіків, які ви можете собі уявити. Те, що вона базова, не означає, що вона не є потужною, багато інших бібліотек візуалізації даних, про які ми будемо говорити, базуються на ній.

Matplotlib - родоначальник бібліотек візуалізації даних Python. Незважаючи на те, що їй більше десяти років, вона все ще є найбільш широко використовуваною бібліотекою для створення графіків у спільноті Python. Вона була розроблена, щоб дуже нагадувати MATLAB, фірмову мову програмування, розроблену у 1980-х роках.

Оскільки Matplotlib була першою бібліотекою візуалізації даних Python, багато інших бібліотек побудовані на її основі або призначені для роботи в парі з нею під час аналізу. Деякі бібліотеки, такі як pandas та Seaborn, є "обгортками" над Matplotlib. Вони дозволяють отримати доступ до кількох методів Matplotlib з меншим кодом.

Matplotlib спочатку був написаний Джоном Д. Хантером (John D. Hunter), з тих пір він має активну спільноту розробників і розповсюджується за ліцензією BSD.

Matplotlib - це бібліотека для створення 2D графіків масивів у Python. Хоча вона походить від імітації графічних команд MATLAB, вона не залежить від MATLAB і може використовуватися в Pythonic, об'єктно-орієнтованому стилі. Хоча Matplotlib написаний переважно на чистому Python, він широко використовує NumPy та інші коди розширень, щоб забезпечити хорошу продуктивність навіть для великих масивів. Matplotlib розроблений з філософією того, що ви повинні мати можливість створювати прості сюжети за допомогою кіль-

кох команд або лише однієї! Якщо ви хочете побачити гістограму своїх даних, вам не потрібно створювати екземпляри об’єктів, викликати методи, задавати властивості тощо; це має просто працювати.

Код Matplotlib концептуально поділений на три частини:

- Інтерфейс pylab - це набір функцій, наданих matplotlib.pylab, що дозволяє користувачеві створювати графіки з кодом, дуже подібним до коду, що генерує фігури MATLAB.
- Інтерфейс Matplotlib або API Matplotlib - це набір класів, які виконують важку роботу, створюють та керують фігурами, текстом, рядками, графіками, тощо. Це абстрактний інтерфейс, який нічого не знає про виведення.
- Бекенди - це пристрої для малювання, залежні від пристрою, також відомі як візуалізатори, які виводять frontend зображення на друк або на пристрій відображення.

***Як користуватися Matplotlib?***

Matplotlib зображує ваші дані на фігурах (Figures), кожна з яких може містити одну або кілька осей (Axes, тобто областей, де точки можна вказати з точки зору координат x-y (або тета-r у полярній сюжет, або x-y-z у тривимірному графіку, тощо). Найпростіший спосіб створення фігури з осями - це використання pyplot.subplots. Потім ми можемо використовувати Axes.plot для нанесення деяких даних по осях:

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots() #Create a figure containing a single axes
```

Наведений вище рядок коду створює два об’єкти і призначає їх змінним "fig" та "ax" відповідно. Спробуємо викликати перший об’єкт:

```
fig
plt.show()
```

На виході отримаємо:

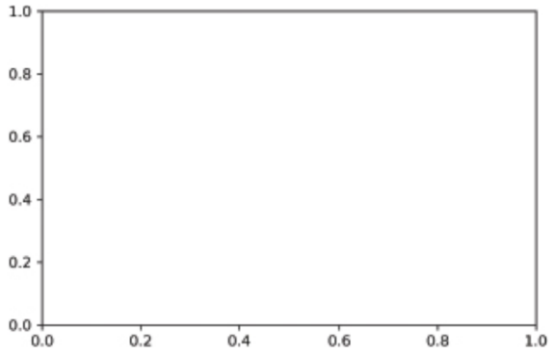


Рис. 19 Фігура Matplotlib, що містить тільки осі

Як бачите, це просто чисте полотно, на якому ви можете намалювати свій графік. Тож почнемо малювати. Інший об’єкт - це об’єкт осей, на якому будуть побудовані ділянки.

```
fig
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
plt.show()
```

Результат буде таким:

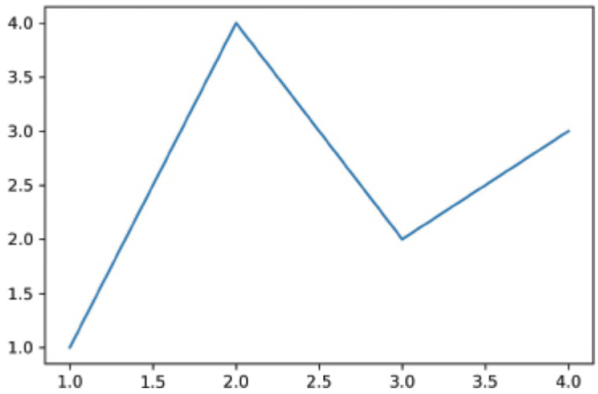


Рис. 20 Малювання ліній на об’єкті осей

Багато інших бібліотек або мов побудови графіків не вимагають явного створення осей. Ви можете зробити те ж саме в Matplotlib: для кожного методу осей для графічного відображення існує відповідна функція в модулі matplotlib.pyplot, яка виконує цей графік на "поточних" осях, створюючи ці осі (і їх батьківську фігуру), якщо вони ще не існують. Тому попередній приклад можна записати коротше як

```
plt.plot([1, 2, 3, 4], [1, 4, 2, 3])
plt.show()
```

***Інтерфейси Matplotlib***

Існує два способи використання Matplotlib:

- Об'єктно-орієнтований інтерфейс.
- Інтерфейс pyplot.

***Об'єктно-орієнтований інтерфейс***

Якщо ви підете саме цим способом використання Matplotlib, то ви повинні явно створити фігури та осі та викликати на них методи ("об’єктно-орієнтований (ОО) стиль").

Приклад інтерфейсу OO-Style:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2, 100)
fig, ax = plt.subplots()
ax.plot(x, x, label = "linear")
ax.plot(x, x**2, label = "quardratic")
ax.plot(x, x**3, label = "cubic")
ax.set_xlabel('x label')
ax.set_ylabel('y label')
ax.set_title("Simple Plot")
ax.legend()
plt.show()
```

Отримаємо графіки, показані на Рис. 21.

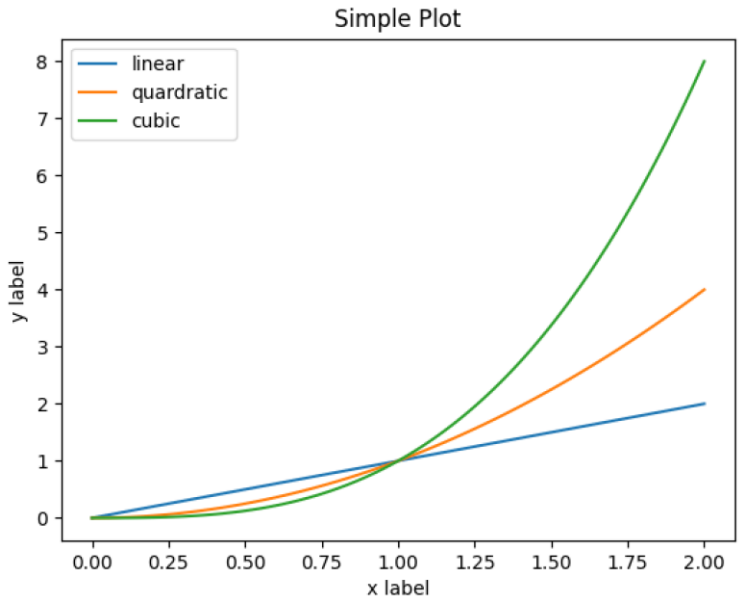


Рис. 21 Графіки функцій y=x, y=x**2  y=x**3

***Інтерфейс pyplot***

Якщо ви йдете саме цим способом використання Matplotlib, то замість того, щоб створювати фігури та осі вручну, ви покладаєтесь на pyplot для автоматичного створення та керування фігурами та осями та використовуєте функції pyplot для побудови графіків.

Той самий приклад, який ми бачили вище, тепер ми побачимо з інтерфейсом pyplot для порівняння.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2, 100)
plt.plot(x, x, label = "linear")
plt.plot(x, x**2, label = "quardratic")
plt.plot(x, x**3, label = "cubic")
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
plt.show()
```

Результат буде таким самим, як на Рис. 21.

Для відображення сітки на графіку можна додати рядок

```
plt.grid()
```

***Основні елементи графіка Matplotlib***

Основні елементи графіка Matplotlib показані на Рис. 22 (https://matplotlib.org/stable/gallery/showcase/anatomy.html ).

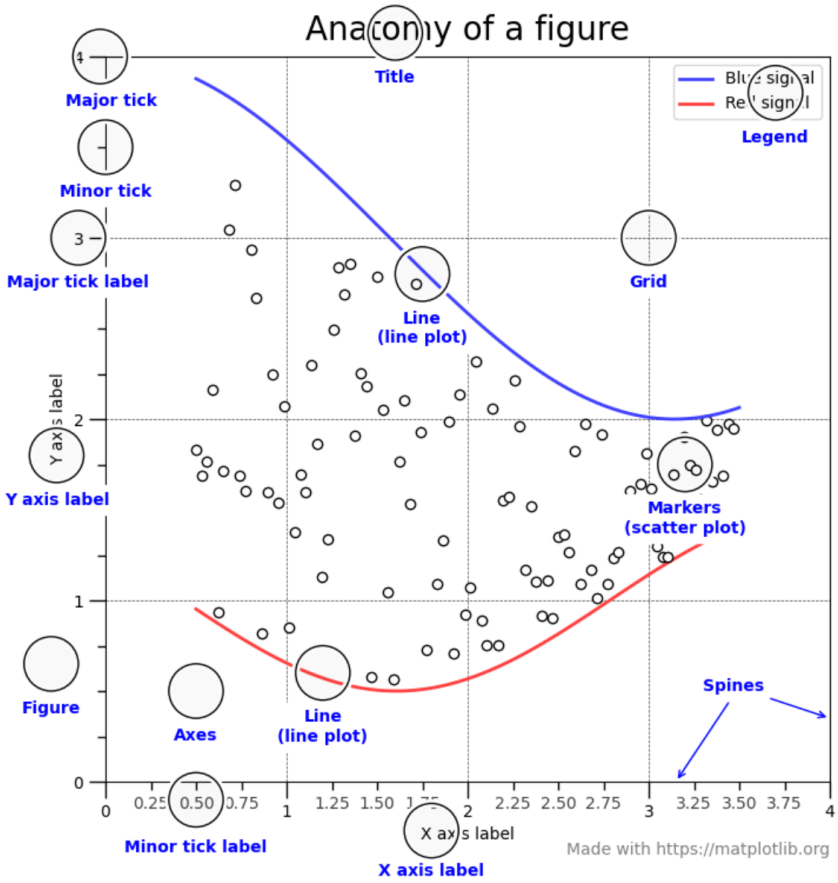


Рис. 22 Основні елементи графіка Matplotlib

Кореневим елементом при побудові графіків в системі Matplotlib є Фігура (Figure). Все, що намальовано на рисунку вище є елементами фігури. Розглянемо її складові більш докладно.

Графік

На рисунку представлені три графіка – два лінійних та точковий. Matplotlib надає величезну кількість різних налаштувань, які можна використовувати для того, щоб надати графікам необхідний вид: задати колір, товщину, тип, стиль лінії і багато іншого.

Осі

Другим, після безпосередньо самого графіка, за важливістю елементом фігури є осі. Для кожної осі можна задати мітку (підпис), основні (major) і додаткові (minor) елементи шкали, їх підписи, розмір, товщину і діапазони.

Сітка і легенда

Сітка і легенда є елементами фігури, які значно підвищують інформативність графіка. Сітка може бути основною (major) і додатковою (minor). Кожному типу сітки можна задавати колір, товщину лінії і тип. Для відображення сітки і легенди використовуються відповідні команди.

Код, за допомогою якого була побудована фігура, зображена на Рис. 22 представлений за посиланням на початку цього підрозділу.

***Поверхні і лінії рівня***

Matplotlib також може побудувати тривимірні дані різними способами. Два поширених варіанти для відображення тривимірних даних – це використання поверхневих або контурних графіків (як контурні лінії на карті). У цьому підрозділі ми розглянемо метод для побудови поверхонь із тривимірних даних і побудуємо контури тривимірних даних.

Щоб побудувати тривимірні дані, їх потрібно впорядкувати у двовимірні масиви для компонентів x, y та z, де обидва компоненти x та y повинні мати ту саму форму, що й компонент z. Для цієї демонстрації ми побудуємо поверхню, що відповідає функції $z=f(x, y) = x^2y^3$ в діапазоні $-2 \leq x \leq 2$ і $-1 \leq y \leq 1$.

Перше завдання полягає в тому, щоб створити відповідну сітку з пар (x, y), на якій можна оцінити цю функцію:

1. Спочатку ми використовуємо np.linspace для створення розумної кількості точок у цих діапазонах:
   ```
   X = np.linspace(-2, 2)
   Y = np.linspace(-1, 1)
   ```
2. Тепер нам потрібно створити сітку, на якій будемо обчислювати значення z. Для цього ми використовуємо метод np.meshgrid:
   ```
   x, y = np.meshgrid(X, Y)
   ```
3. Тепер ми можемо обчислити значення z в кожній з точок сітки:
   ```
   z = x**2 * y**3
   ```
4. Щоб побудувати тривимірні поверхні, нам потрібно завантажити панель інструментів Matplotlib, mplot3d, яка постачається з пакетом Matplotlib. Це не буде використовуватися явно в коді, але за лаштунками це робить утиліти для тривимірного графіка доступними для Matplotlib:
   ```
   from mpl_toolkits import mplot3d
   ```
5. Далі створюємо нову фігуру і набір тривимірних осей для фігури:
   ```
   fig = plt.figure()
   ax = fig.add_subplot(projection="3d")
   ```
6. Тепер ми можемо викликати метод plot_surface на цих осях, щоб побудувати дані:
   ```
   ax.plot_surface(x, y, z)
   ```
7. Надзвичайно важливо додавати мітки осей до тривимірних графіків, оскільки може бути не зрозуміло, яка вісь яка на відображеній діаграмі:
   ```
   ax.set_xlabel("$x$")
   ax.set_ylabel("$y$")
   ax.set_zlabel("$z$")
   ```
8. На цьому етапі ми також повинні встановити назву:
   ```
   ax.set_title("Graph of the function $f(x) = x^2y^3$)
   ```
   Ви можете використовувати метод plt.show для відображення фігури в новому вікні або plt.savefig, щоб зберегти фігуру у файлі. Результат попередньої послідовності показано тут:

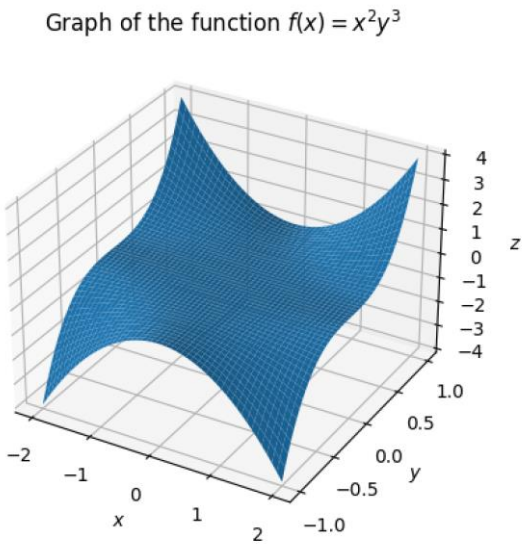


Рис. 23 Графік функції $z = x^2y^3$

9. Для контурних графіків не потрібен набір інструментів mplot3d, а в інтерфейсі pyplot є процедура contour, яка створює контурні графіки. Однак, на відміну від звичайних (двовимірних) програм побудови графіків, підпрограма contour вимагає тих самих аргументів, що й метод plot_surface. Ми використовуємо таку послідовність для створення сюжету:

```
fig = plt.figure() # Force a new figure
plt.contour(x, y, z)
plt.title("Contours of $f(x) = x^2y^3$")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.show()
```

Результат показаний на наступному графіку:

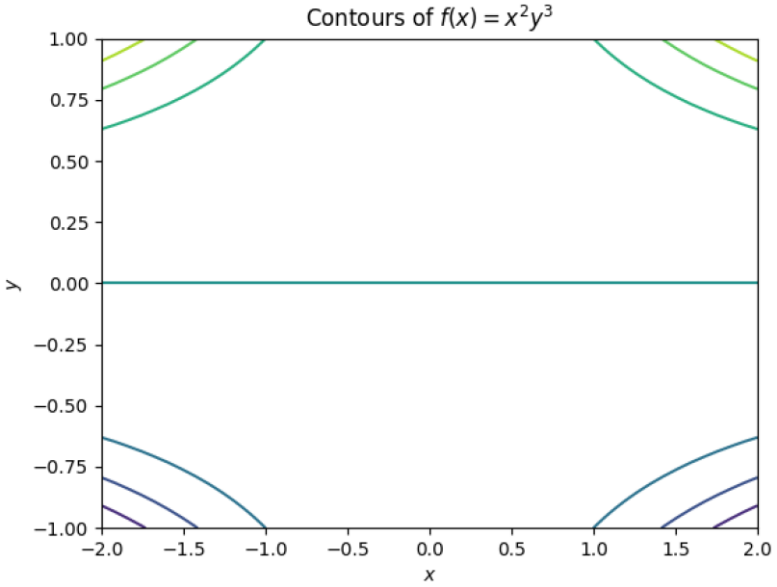


Рис. 24 Графік ліній рівня

**Робота з Seaborn**

Seaborn — це пакет Python для візуалізації даних, який також забезпечує високорівневий інтерфейс для Matplotlib. З Seaborn легше працювати, ніж Matplotlib, і насправді він розширює Matplotlib, але майте на увазі, що Seaborn не такий потужний, як Matplotlib.

Seaborn вирішує дві проблеми Matplotlib. Перша включає параметри Matplotlib за замовчуванням. Seaborn працює з різними параметрами, що забезпечує більшу гнучкість, ніж відтворення графіків Matplotlib за замовчуванням. Seaborn усуває обмеження значень за замовчуванням Matplotlib для таких функцій, як кольори, галочки на верхній і правій осях і стиль (серед інших).

Крім того, Seaborn полегшує побудову цілих фреймів даних (подібно до Pandas), робити це в Matplotlib важче. Тим не менш, оскільки Seaborn розширює Matplotlib, знання Matplotlib є вигідним і спростить вашу криву навчання.

Інсталяцію пакета Seaborn можна виконати командою

```
pip install seaborn
```

***Можливості Seaborn***

Деякі з можливостей Seaborn включають:

• масштабування графіка Seaborn
• задання стилю графіка
• встановлення розміру фігури
• обертання тексту напису
• встановлення xlim або ylim
• встановлення логарифмічного масштабу
• додавання заголовку

Деякі корисні методи:

• plt.xlabel()
• plt.ylabel()
• plt.annotate()
• plt.legend()
• plt.ylim()
• plt.savefig()

Seaborn підтримує різні вбудовані набори даних, як і NumPy і Pandas, включаючи набір даних Iris і Titanic, обидва з яких ви побачите в наступних розділах. Як відправну точку, трирядковий зразок коду в наступному розділі показує, як відобразити рядки у вбудованому наборі даних «tips».

***Вбудовані набори даних Seaborn***

У наступному листингі показано вміст файлу seaborn_tips.py, який ілюструє, як читати набір даних tips у фрейм даних і відображати перші п’ять рядків набору даних.

```
import seaborn as sns
df = sns.load_dataset("tips")
print(df.head())
```

Цей листинг дуже простий: після імпорту seaborn змінна df ініціалізується даними з вбудованого набору даних tips, а оператор print() відображає перші

п’ять рядків df. Зверніть увагу, що API load_dataset() шукає онлайн або вбудовані набори даних. На виході отримаємо:

```
   total_bill   tip     sex smoker  day    time   size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

***Набір даних Iris в Seaborn***

У наступному листингі показано вміст файлу seaborn_iris.py, який ілюструє, як відобразити набір даних Iris.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load iris data
iris = sns.load_dataset("iris")
# Construct iris plot
sns.swarmplot(x="species", y="petal_length", data=iris)
# Show plot
plt.show()
```

Програма спочатку імпортує seaborn і matplotlib.pyplot, а потім ініціалізує змінну iris вмістом вбудованого набору даних Iris. Далі API swamplot() відображає графік з горизонтальною віссю, позначеною species, і вертикальною віссю з позначкою petal_length, а відображені точки є з набору даних Iris.

На Рис. 25 показано зображення набору даних Iris на основі коду програми seaborn_iris.py.

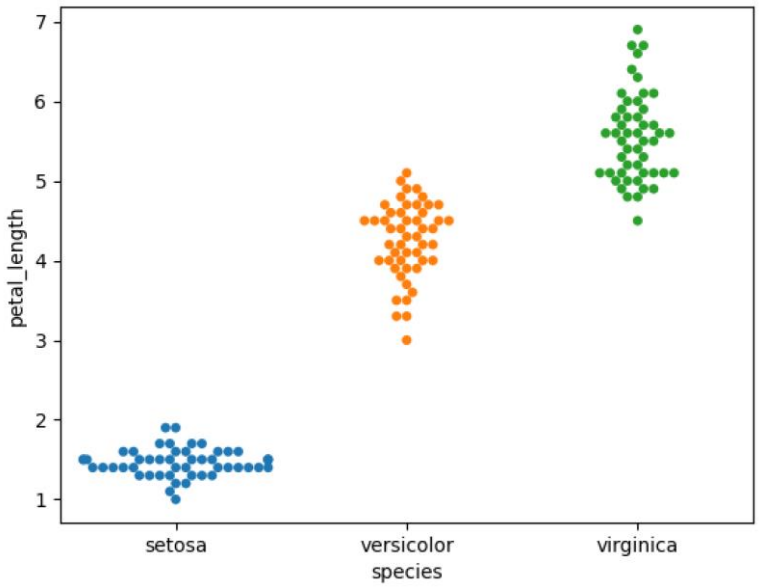


Рис. 25 Набір даних Iris

***Набір даних Titanic в Seaborn***

Наступний листинг відображає вміст файлу seaborn_titanic_plot.py, який ілюструє, як відобразити дані з вбудованого набіру даних Titanic.

```
import matplotlib.pyplot as plt
```

```
import seaborn as sns
titanic = sns.load_dataset("titanic")
g = sns.factorplot("class", "survived", "sex",
data=titanic, kind="bar", palette="muted", legend=False)
plt.show()
```

Листинг містить ті самі оператори імпорту, що й попередня програма, а потім ініціалізує змінну titanic вмістом вбудованого набору даних Titanic. Далі API factorplot() відображає графік з атрибутами набору даних, які вказані у виклику API.

На Рис. 26 показано графік даних з набору даних Titanic на основі коду програми.

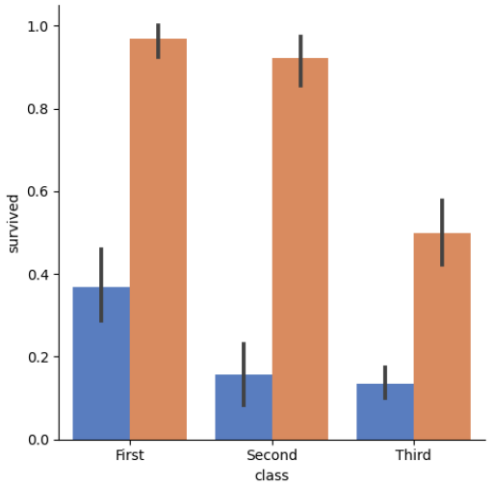


Рис. 26 Гістограма для набору даних Titanic

***Витягування даних з набору даних Titanic в Seaborn***

У наступному листингі показано вміст файлу seaborn_titanic.py, який ілюструє, як витягувати підмножини даних із набору даних Titanic.

```
import matplotlib.pyplot as plt
import seaborn as sns
titanic = sns.load_dataset("titanic")
print("titanic info:")
titanic.info()
print("first five rows of titanic:")
print(titanic.head())
print("first four ages:")
print(titanic.loc[0:3,'age'])
print("fifth passenger:")
print(titanic.iloc[4])
#print("first five ages:")
#print(titanic['age'].head())
#print("first five ages and gender:")
#print(titanic[['age','sex']].head())
#print("descending ages:")
#print(titanic.sort_values('age', ascending = False).head())
#print("older than 50:")
#print(titanic[titanic['age'] > 50])
```

```
#print("embarked (unique):")
#print(titanic['embarked'].unique())
#print("survivor counts:")
#print(titanic['survived'].value_counts())
#print("counts per class:")
#print(titanic['pclass'].value_counts())
#print("max/min/mean/median ages:")
#print(titanic['age'].max())
#print(titanic['age'].min())
#print(titanic['age'].mean())
#print(titanic['age'].median())
```

Листинг містить ті самі оператори імпорту, що й попередні листинги, а потім ініціалізує змінну titanic вмістом вбудованого набору даних Titanic. Наступна частина лістингу відображає різні аспекти набору даних Titanic, такі як його структура, перші п’ять рядків, перші чотири віки (ages) та деталі п’ятого пасажира.

Як бачите, існує великий блок «прокоментованого» коду, який ви можете розкоментувати, щоб побачити пов’язані результати, такі як вік, стать, особи старше 50 років, унікальні рядки тощо. Вихід програми буде таким:

```
titanic info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    category
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object
 13  alive        891 non-null    object
 14  alone        891 non-null    bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
first five rows of titanic:
   survived  pclass     sex   age  sibsp  parch     fare embarked  class
who  adult_male deck  embark_town alive  alone
0         0       3    male  22.0      1      0   7.2500        S  Third
man        True  NaN  Southampton    no  False
1         1       1  female  38.0      1      0  71.2833        C  First
woman       False    C    Cherbourg   yes  False
2         1       3  female  26.0      0      0   7.9250        S  Third
woman       False  NaN  Southampton   yes   True
3         1       1  female  35.0      1      0  53.1000        S  First
woman       False    C  Southampton   yes  False
```

```
4         0       3    male  35.0      0      0   8.0500        S  Third
man        True  NaN  Southampton    no   True
first four ages:
0    22.0
1    38.0
2    26.0
3    35.0
Name: age, dtype: float64
fifth passenger:
survived                0
pclass                  3
sex                  male
age                  35.0
sibsp                   0
parch                   0
fare                 8.05
embarked                S
class               Third
who                   man
adult_male           True
deck                  NaN
embark_town    Southampton
alive                  no
alone                True
Name: 4, dtype: object
```

**Питання для самоконтролю**

1. В чому полягають переваги спеціальних бібліотек для наукових обчислень в порівнянні зі стандартним Python?
2. Що таке векторизовані обчислення?
3. Як встановити бібліотеку NumPy?
4. Охарактеризуйте об'єкт масиву NumPy.
5. Як створити та ініціалізувати масив NumPy?
6. Як оголосити масив певної форми, заповнений стандартними або довільними значеннями?
7. Що таке універсальні функції (ufuncs)?
8. Як читати та записувати масиву у файл?
9. Як працювати з матрицями в NumPy?
10. Які можливості має бібліотека Matplotlib?
11. Чим відрізняється бібліотека Seaborn?
12. Як працювати з вбудованими наборами даних Seaborn?

 **Тестові завдання для перевірки знань**

1. Який правильний синтаксис для створення масиву NumPy?
A. np.object([1,2,3,4,5])
B. np.array([1,2,3,4,5])
C. np.createArray([1,2,3,4,5])

2. Який правильний синтаксис для перевірки кількості вимірів у масиві?
A. arr.dim()
B. arr.dim
C. arr.ndim
D. arr.ndim()

3. Який правильний синтаксис для друку числа 8 із наведеного нижче масиву: arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])?
A. print(arr[3,0])
B. print(arr[7,2])
C. print(arr[1,2])

 **Завдання для самостійного виконання**

**Завдання 1.** Напишіть програму NumPy для створення матриці 3x3 зі значеннями в діапазоні від 2 до 10.

**Завдання 2.** Напишіть програму NumPy для обчислення координат x і y для точок на синусоїді та побудуйте точки за допомогою Matplotlib.

## Тема 14. Бібліотека Pygame

**Мета:** розглянути використання бібліотеки Pygame для створення комп'ютерних ігр; з'ясувати ключові поняття Pygame; на прикладі конкретної гри показати використання об'єктно-орієнтованого підходу при розробці програми.

**План**

1. Створення ігр за допомогою Pygame.
2. Інсталяція Pygame.
3. Ключові поняття Pygame. Поверхня відображення.

4. Події. Типи подій. Черга подій.
5. Перша програма Pygame.
6. Гра StarshipMeteors. Діаграма класів.
7. Клас GameObject. Відображення Starship.
8. Переміщення космічного корабля. Обробка подій.
9. Додавання класу Meteor.
10. Переміщення метеорів.
11. Виявлення зіткнення.
12. Визначення виграшу.
13. Призупинення гри. Відображення повідомлення про закінчення гри.
14. Гра StarshipMeteors (повний листинг).

## Основні терміни і поняття

*Бібліотека Pygame. Поверхня відображення. Події та їх типи. Константи клавіатури. Черга подій. Класи та об'єкти гри. Переміщення об'єктів. Виявлення зіткнення.*

## Основні теоретичні положення

### Створення ігр за допомогою Pygame

Pygame — це кросплатформна безкоштовна бібліотека Python з відкритим вихідним кодом, призначена для полегшення створення мультимедійних програм, таких як ігри. Розробка Pygame почалася ще в жовтні 2000 року, а версія Pygame 1.0 була випущена через півроку [16].

Pygame побудовано на основі бібліотеки SDL. SDL (або Simple Directmedia Layer) — це міжплатформна бібліотека розробки, призначена для забезпечення доступу до аудіо, клавіатури, миші, джойстика та графічного обладнання через OpenGL і Direct3D.

SDL офіційно підтримує Windows, Mac OS X, Linux, iOS і Android. Сама SDL написана на C, а Pygame надає обгортку навколо SDL. Однак Pygame додає функціональні можливості, яких немає в SDL, щоб полегшити створення графічних або відеоігор. Ці функції включають векторну математику, виявлення зіткнень, керування графіком 2D-спрайтів, підтримку MIDI, камеру, маніпуляції з масивом пікселів, трансформації, фільтрацію, розширену підтримку шрифтів та малювання.

#### *Інсталяція Pygame*

Щоб інсталювати Pygame, введіть таку команду в командному рядку термінала:

```
pip install pygame
```

Ця команда запускає менеджер пакетів pip і той встановлює пакет pygame до інсталяції Python.

#### *Ключові поняття Pygame*

Далі в цьому розділі ми ознайомимось з ключовими поняттями, модулі, класами та функціями Pygame та створимо дуже простий перший додаток Pygame. У наступному розділі відбудеться розробка простої відеоігри в аркадному стилі, яка ілюструє, як можна створити гру за допомогою Pygame.

*Поверхня відображення*

Дисплейна поверхня (Display Surface) є найважливішою частиною гри Pygame. Це головне вікно вашої гри, яке може бути будь-якого розміру, однак ви можете мати лише одну дисплейну поверхню.

Багато в чому дисплейна поверхня схожа на чистий аркуш паперу, на якому можна малювати. Сама поверхня складається з пікселів, які пронумеровані від 0,0 у верхньому лівому куті, а розташування пікселів проіндексовано по осі x та осі y. Це показано нижче:

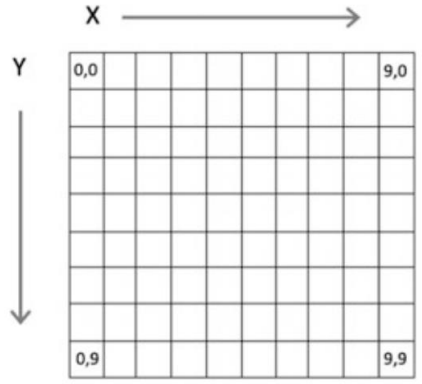


Рис. 27 Display Surface

Дисплейна поверхня створюється за допомогою функції pygame.display.set_mode(). Ця функція приймає кортеж, який визначає розміри поверхні. Наприклад:

```
display_surface = pygame.display.set_mode((400, 300))
```

Це створить дисплейну поверхню (вікно) розміром 400 на 300 пікселів.

Після того, як у вас є дисплейна поверхня, ви можете заповнити її відповідним кольором фону (за замовчуванням чорний), однак, якщо ви хочете інший колір фону або хочете очистити все, що раніше було намальовано на поверхні, ви можете використовувати поверхні метод fill():

```
WHITE = (255, 255, 255)
display_surface.fill(WHITE)
```

Щоб підвищити продуктивність, будь-які зміни, які ви вносите до дисплейної поверхні, насправді відбуваються у фоновому режимі і не відображатимуться на фактичному дисплеї, який бачить користувач, доки ви не викличете методи update() або flip() на поверхні. Наприклад:

- `pygame.display.update()`
- `pygame.display.flip()`

Метод update() перемалює дисплей з усіма змінами, внесеними в дисплей у фоновому режимі. Він має додатковий параметр, який дозволяє вказати лише область дисплея для оновлення (це визначається за допомогою Rect, який представляє прямокутну область на екрані). Метод flip() завжди оновлює весь дисплей (і таким чином робить те саме, що й метод update() без параметрів).

Інший метод, який не є спеціально методом поверхні відображення, але який часто використовується під час створення поверхні відображення, забезпечує заголовок для вікна верхнього рівня. Це функція pygame.display.set_caption(). Наприклад:

```
pygame.display.set_caption('Hello World')
```

*Події*

Подібно до того, як системи графічного інтерфейсу користувача, описані в попередніх розділах, мають цикл подій, який дозволяє програмісту визначити, що робить користувач (у цих випадках це, як правило, вибір пункту меню, натискання кнопки або введення даних, тощо); Pygame має цикл подій, який дозволяє грі визначити, що робить гравець. Наприклад, користувач може натиснути клавішу зі стрілкою вліво або вправо. Це представлено подією.

*Типи подій*

Кожна подія, що відбувається, має пов’язану інформацію, наприклад тип цієї події. Наприклад:

• Натискання клавіші призведе до події типу KEYDOWN, а відпускання клавіші призведе до типу події KEYUP.

• Вибір кнопки закриття вікна призведе до створення типу події QUIT, тощо.

• Використання миші може генерувати події MOUSEMOTION, а також типи подій MOUSEBUTTONDOWN і MOUSEBUTTONUP.

• За допомогою джойстика можна генерувати кілька різних типів подій, включаючи JOYAXISMOTION, JOYBALLMOTION, JOYBUTTONDOWN і JOYBU TTONUP.

Ці типи подій повідомляють вам, що сталося. Це означає, що ви можете вибрати типи подій, з якими ви хочете мати справу, і ігнорувати інші події.

*Інформація про подію*

Кожен тип об’єкта події надає інформацію, пов’язану з цією подією. Наприклад, об’єкт події, орієнтований на клавішу, надасть фактично натиснуту клавішу, тоді як об’єкт події, орієнтований на мишу, надасть інформацію про положення миші, яку кнопку було натиснуто тощо. Якщо ви спробуєте отримати доступ до атрибута для події, яка цей атрибут не підтримує, то буде згенеровано помилку.

Нижче наведено деякі атрибути, доступні для різних типів подій:

• KEYDOWN та KEYUP події мають атрибут key та атрибут mod (що вказує, чи натискаються інші клавіші модифікації, наприклад Shift).

• MOUSEBUTTONUP і MOUSEBUTTONDOWN мають атрибут pos, який містить кортеж, що вказує розташування миші за координатами x і y на нижній поверхні. Вони також мають атрибут button, що вказує, яка кнопка миші була натиснута.

• MOUSEMOTION має атрибути pos, rel і buttons. Pos — це кортеж, що вказує розташування курсору миші по x і y. Атрибут rel вказує на кількість переміщень миші, a buttons вказують на стан кнопок миші.

Нариклад, якщо ми хочемо перевірити тип події клавіатури, а потім перевірити, чи натиснута клавіша була пробілом, ми можемо написати:

```
if event.type == pygame.KEYDOWN:
    # Check to see which key is pressed
    if event.key == pygame.K_SPACE:
        print('space')
```

Існує багато констант клавіатури, які використовуються для представлення клавіш на клавіатурі в Pygame. Константа K_SPACE, використана вище, є лише однією з них.

Усі константи клавіатури мають префікс «K_», за яким слідує клавіша або назва клавіші, наприклад:

• K_TAB, K_SPACE, K_PLUS, K_0, K_1, K_AT, K_a, K_b, K_z, K_DELTE, K_DOWN, K_LEFT, K_RIGHT, K_LEFT тощо.

Додаткові константи клавіатури надаються для станів модифікаторів, які можна об'єднати з наведеними вище, наприклад KMOD_SHIFT, KMOD_CAPS, KMOD_CTRL та KMOD_ALT.

*Черга подій*

Події надходять до програми Pygame через чергу подій.

Черга подій використовується для збору подій, коли вони відбуваються. Наприклад, припустимо, що користувач двічі клацає мишею та двічі клавішею, перш ніж програма зможе їх обробити; тоді в черзі подій буде чотири події, як показано нижче:

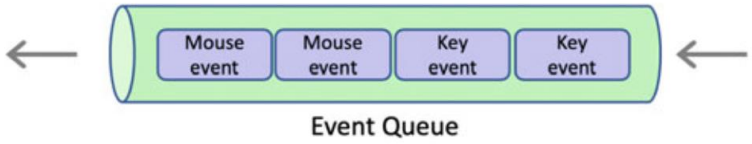


Рис. 28 Черга подій

Потім програма може отримати ітератор із черги подій і обробляти події по черзі. Поки програма обробляє ці події, можуть відбуватися інші події, які будуть додані до черги подій. Коли програма завершила обробку початкової колекції подій, вона може отримати наступний набір подій для обробки.

Однією з істотних переваг цього підходу є те, що жодна подія ніколи не втрачається; тобто якщо користувач двічі клацне мишею, поки програма обробляє попередній набір подій; вони будуть записані та додані до черги подій. Ще одна перевага полягає в тому, що події будуть представлені програмі в тому порядку, в якому вони відбувалися.

Функція pygame.event.get() прочитає всі події, які наразі знаходяться в черзі подій (видаляючи їх із черги подій). Метод повертає EventList, який є ітераційним списком прочитаних подій. Потім кожну подію можна обробляти по черзі. Наприклад:

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        print('Received Quit Event:')
    elif event.type == pygame.MOUSEBUTTONDOWN:
        print('Received Mouse Event')
    elif event.type == pygame.KEYDOWN:
```

```
            print('Received KeyDown Event')
```

*Перша програма Pygame*

Програма, яку ми створимо, відобразить вікно Pygame з назвою «Hello World». Тоді ми зможемо вийти з гри. Хоча технічно це не гра, але вона має базову архітектуру програми Pygame.

```
import pygame
def main():
    print('Starting Game')
    print('Initialising Pygame')
    pygame.init() # Required by every Pygame application
    print('Initialising HelloWorldGame')
    pygame.display.set_mode((200, 100))
    pygame.display.set_caption('Hello World')
    print('Update display')
    pygame.display.update()
    print('Starting main Game Playing Loop')
    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                print('Received Quit Event:', event)
                running = False
    print('Game Over')
    pygame.quit()
if __name__ == '__main__':
    main()
```

*Більш інтерактивна програма Pygame*

Перша програма Pygame, яку ми розглянули раніше, просто показувала вікно з написом «Hello World». Тепер ми можемо трохи розширити її, погравши з деякими функціями, які ми розглянули вище.

Нова програма додасть деяку обробку подій миші. Це дозволить нам визначити розташування миші, коли користувач клацнув на вікні, і намалювати в цій точці невелике синє поле.

Якщо користувач клацне мишею кілька разів, ми отримаємо кілька синіх прямокутників. Це показано нижче.


Рис. 29 Вікно другої програми

Це все ще не справжня гра, але цей застосунок Pygame є більш інтерактивним.

Вихідний код програми представлений нижче:

```
import pygame
FRAME_REFRESH_RATE = 30
BLUE = (0, 0, 255)
BACKGROUND = (255, 255, 255) # White
WIDTH = 10
HEIGHT = 10
def main():
    print('Initialising PyGame')
    pygame.init() # Required by every PyGame application
    print('Initialising Box Game')
    display_surface = pygame.display.set_mode((400, 300))
    pygame.display.set_caption('Box Game')
    print('Update display')
    pygame.display.update()
    print('Setup the Clock')
    clock = pygame.time.Clock()
    # Clear the screen of current contents
    display_surface.fill(BACKGROUND)
    print('Starting main Game Playing Loop')
    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                print('Received Quit Event:', event)
                running = False
            elif event.type == pygame.MOUSEBUTTONDOWN:
                print('Received Mouse Event', event)
                x, y = event.pos
                pygame.draw.rect(display_surface,  BLUE,  [x,
y, WIDTH, HEIGHT])
        # Update the display
        pygame.display.update()
        # Defines the frame rate - the number of frames per
    second
        # Should be called once per frame (but only once)
        clock.tick(FRAME_REFRESH_RATE)
    print('Game Over')
    # Now tidy up and quit Python
    pygame.quit()
if __name__ == '__main__':
    main()
```

***Гра StarshipMeteors***

У цьому підрозділі ми створимо гру, в якій ви керуєте зоряним кораблем через поле метеорів [16]. Чим довше ви граєте в гру, тим більшу кількість метеорів ви зустрінете. Типовий екран гри показаний нижче:

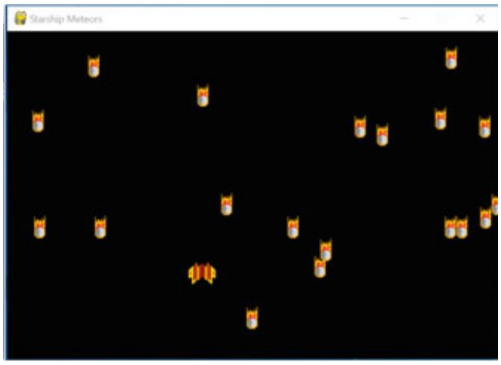

Рис. 30 Типовий екран гри

Ми реалізуємо кілька класів для представлення сутностей у грі. Використання класів не є обов'язковим способом реалізації гри, і слід зазначити, що багато розробників уникають використання класів. Однак використання класу дозволяє зберігати дані, пов'язані з об'єктом у грі, в одному місці; це також спрощує створення кількох екземплярів одного і того ж об'єкта (наприклад, метеорів) у грі.

Нижче показано класи та їх взаємозв'язки:

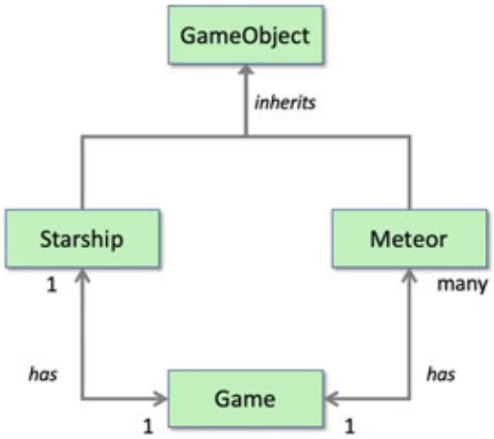


Рис. 31 Діаграма класів

Ця діаграма показує, що класи Starship і Meteor розширюють клас під назвою GameObject.

У свою чергу, це також показує, що Game має співвідношення 1:1 з класом Starship. Тобто Game містить посилання на один Starship, а Starship, у свою чергу, містить одне посилання на Game.

На противагу цьому Game має відношення 1 до багатьох із класом Meteor. Тобто об'єкт Game містить посилання на багато Meteors, і кожен Meteor містить посилання на один об'єкт Game.

*Клас GameObject*

Клас GameObject визначає три методи:

Метод load_image() можна використовувати для завантаження зображення, яке буде використано для візуального представлення конкретного типу ігрового об'єкта. Потім метод використовує ширину та висоту зображення для визначення ширини та висоти ігрового об'єкта.

Метод rect() повертає прямокутник, що представляє поточну область, яку використовує ігровий об'єкт на базовій поверхні малювання. Прямокутники дуже корисні для порівняння розташування одного об'єкта з іншим (наприклад, коли визначають, чи відбулося зіткнення).

Метод draw() малює зображення GameObjects на display_surface, використовуючи поточні координати x і y. Метод можна замінити в підкласах, якщо їх потрібно намалювати іншим способом.

Код для класу GameObject представлено нижче:

```
class GameObject:
    def load_image(self, filename):
        self.image = pygame.image.load(filename).convert()
        self.width = self.image.get_width()
        self.height = self.image.get_height()
    def rect(self):
        """ Generates a rectangle representing the objects
            location and dimensions """
        return   pygame.Rect(self.x,   self.y,   self.width,
self.height)
    def draw(self):
        """ draw the game object at the
            current x, y coordinates """
        self.game.display_surface.blit(self.image,   (self.x,
self.y))
```

Клас GameObject безпосередньо розширюється класами Starship і Meteor.

На даний момент існує лише два типи ігрових елементів: зоряний корабель і метеори; але в майбутньому це може бути поширено на планети, комети, падаючі зірки тощо.

*Відображення Starship*

Гравець-людина в цій грі керуватиме зоряним кораблем, який можна переміщати по дисплею.

Зоряний корабель буде представлений екземпляром класу Starship. Цей клас розширить клас GameObject, який містить загальну поведінку для будь-якого типу елементів, представлених у грі.

Клас Starship визначає власний метод __init__(), який бере посилання на game, частиною якої є зореліт. Цей метод ініціалізації встановлює початкове розташування Starship як половину ширини дисплея для координати x і висоти відображення мінус 40 для координати y (це дає трохи буфера перед кінцем екрана). Потім він використовує метод load_image() з батьківського класу GameObject, щоб завантажити зображення, яке буде використовуватися для представлення Starship. Це зображення зберігається у файлі під назвою starship.png. На даний момент ми залишимо клас Starship як він є (однак ми повернемося до цього класу, щоб перетворити його на рухомий об'єкт у наступному розділі).

Нижче наведена поточна версія класу Starship:

```
class Starship(GameObject):
    """ Represents a starship"""
    def __init__(self, game):
```

```
        self.game = game
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')
```

У класі Game ми тепер додамо рядок до методу __init__() для ініціалізації об’єкта Starship. Ось цей рядок:

```
# Set up the starship
self.starship = Starship(self)
```

Ми також додамо рядок до основного циклу while у методі play() безпосередньо перед тим, як оновити дисплей. Цей рядок викличе метод draw() для об’єкта зоряного корабля:

```
# Draw the starship
self.starship.draw()
```

Це матиме ефект малювання зоряного корабля на поверхні малювання вікна у фоновому режимі до того, як дисплей буде оновлено.

Коли ми зараз запустимо цю версію гри StarshipMeteor, ми бачимо Starship на дисплеї:


Рис. 32 Відображення зоряного корабля

Звичайно, зараз зоряний корабель не рухається; але ми розглянемо це в наступному розділі.

*Переміщення космічного корабля*

Ми хочемо мати можливість переміщати Starship в межах екрана дисплея. Для цього нам потрібно змінити його координати x і y у відповідь на натискання користувачами різних клавіш.

Ми будемо використовувати клавіші зі стрілками для переміщення вгору і вниз по екрану або ліворуч або праворуч від екрана. Для цього ми визначимо чотири методи в класі Starship; ці методи будуть рухати зореліт вгору, вниз, вліво і вправо тощо.

Оновлений клас Starship показано нижче:

```
class Starship(GameObject):
    """ Represents a starship"""
    def __init__(self, game):
        super().__init__()
        self.game = game
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
```

```
            self.load_image('starship.png')
        def move_right(self):
            """ moves the starship right across the screen """
            self.x = self.x + STARSHIP_SPEED
            if self.x + self.width > DISPLAY_WIDTH:
                self.x = DISPLAY_WIDTH - self.width
        def move_left(self):
            """ Move the starship left across the screen """
            self.x = self.x - STARSHIP_SPEED
            if self.x < 0:
                self.x = 0
        def move_up(self):
            """ Move the starship up the screen """
            self.y = self.y - STARSHIP_SPEED
            if self.y < 0:
                self.y = 0
        def move_down(self):
            """ Move the starship down the screen """
            self.y = self.y + STARSHIP_SPEED
            if self.y + self.height > DISPLAY_HEIGHT:
                self.y = DISPLAY_HEIGHT - self.height
        def __str__(self):
            return 'Starship(' + str(self.x) + ', ' + str(self.y)
+ ')'
```

Ця версія класу Starship визначає різні методи переміщення. Ці методи використовують нове глобальне значення STARSHIP_SPEED, щоб визначити, як далеко і як швидко рухається Starship. Якщо ви хочете змінити швидкість, з якою рухається Starship, ви можете змінити це глобальне значення.

Залежно від наміченого напрямку нам потрібно буде змінити координати x або y зоряного корабля.

• Якщо зореліт рухається ліворуч, то координата x зменшується на STARSHIP_SPEED,

• якщо він рухається вправо, то координата x збільшується на STARSHIP_SPEED,

• у свою чергу, якщо зоряний корабель рухається вгору по екрану, то координата y зменшується на STARSHIP_SPEED,

• але якщо він рухається вниз по екрану, то координата y збільшується на STARSHIP_SPEED.

Звичайно, ми не хочемо, щоб наш Зоряний корабель відлетів від краю екрана, тому потрібно перевірити, чи досяг він меж екрана. Таким чином проводяться тести, щоб побачити, чи опустилися значення x або y нижче нуля чи вище значень DISPLAY_WIDTH або DISPLAY_HEIGHT. Якщо будь-яка з цих умов виконується, значення x або y скидаються до відповідного за замовчуванням.

Тепер ми можемо пов’язати ці методи з введенням гравця. Оскільки для цього ми використовуємо клавіші зі стрілками вліво, вправо, вгору та вниз (K_LEFT, K_RIGHT, K_UP і K_DOWN), ми можемо розширити цикл обробки подій, який ми вже визначили для основного циклу гри. Коли натиснуто одну з

цих клавіш, ми викличемо відповідний метод переміщення на об'єкті зоряного корабля, що вже утримується об'єктом Game.

Основна обробка подій для циклу тепер наступна:

```
# Work out what the user wants to do
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        is_running = False
    elif event.type == pygame.KEYDOWN:
        # Check to see which key is pressed
        if event.key == pygame.K_RIGHT:
            # Right arrow key has been pressed
            # move the player right
            self.starship.move_right()
        elif event.key == pygame.K_LEFT:
            # Left arrow has been pressed
            # move the player left
            self.starship.move_left()
        elif event.key == pygame.K_UP:
            self.starship.move_up()
        elif event.key == pygame.K_DOWN:
            self.starship.move_down()
        elif event.key == pygame.K_p:
            self._pause()
        elif event.key == pygame.K_q:
            is_running = False
```

Проте ми не зовсім закінчили. Якщо ми спробуємо запустити цю версію програми, ми отримаємо слід зоряних кораблів, намальований на екрані; наприклад:

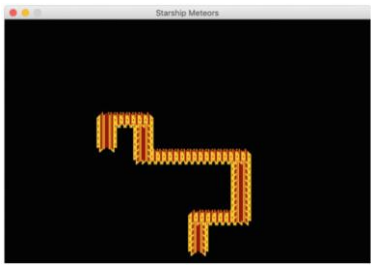


Рис. 33 Слід від зоряних кораблів

Проблема в тому, що ми перемальовуємо зоряний корабель в іншому положеннів той час як попереднє зображення все ще присутнє.

Тепер у нас є два варіанти, один – просто заповнити весь екран чорним; ефективно приховуючи все, що було намальовано до цього часу; або, як альтернативу, ми можемо просто перемалювати область, яка використовувалась для попередньої позиції зображення. Який підхід буде прийнятий, залежить від конкретного сценарію, представленого вашою грою. Оскільки після додавання на екрані буде багато метеорів; найпростіший варіант — перезаписати все на екрані перед тим, як перемалювати зореліт. Тому ми додамо наступний рядок:

```
# Clear the screen of current contents
self.display_surface.fill(BACKGROUND)
```

Цей рядок додається безпосередньо перед тим, як ми намалюємо Starship в головній грі в циклі while.

Тепер, коли ми переміщаємо Starship, старе зображення видаляється, перш ніж ми намалюємо нове зображення:

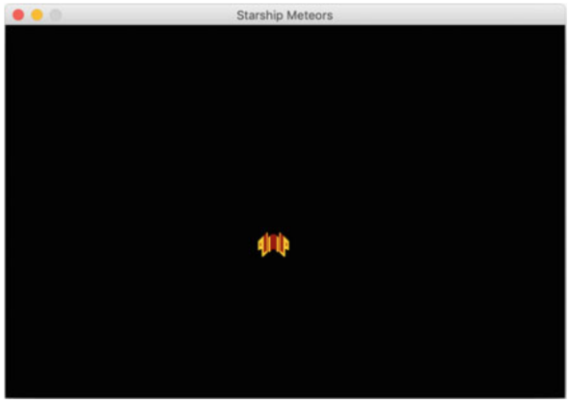


Рис. 34 Вікно гри після видалення старого зображення

Слід зазначити, що ми також визначили інше глобальне значення - BACKGROUND, яке використовується для збереження кольору фону ігрової поверхні. Воно встановлено в чорний колір, як показано нижче:

```
# Define default RGB colours
BACKGROUND = (0, 0, 0)
```

Якщо ви хочете використовувати інший колір фону, змініть це глобальне значення.

*Додавання класу Meteor*

Клас Meteor також буде підкласом класу GameObject. Однак він надасть лише метод move_down(), а не різноманітні методи переміщення Starship.

Він також повинен мати випадкову початкову координату x, щоб, коли до гри додається метеор, його початкова позиція змінювалася. Цю випадкову позицію можна створити за допомогою функції random.randint(), використовуючи значення від 0 до ширини поверхні малювання. Метеор також почнеться у верхній частині екрана, тому матиме іншу початкову координату y від зоряного корабля. Нарешті, ми також хочемо, щоб наші метеори мали різну швидкість; це може бути інше випадкове число від 1 до певної заданої максимальної швидкості метеора.

Щоб підтримати все це, нам потрібно додати до модулів, що імпортуються, random та визначити кілька нових глобальних значень, наприклад:

```
import pygame, random
INITIAL_METEOR_Y_LOCATION = 10
MAX_METEOR_SPEED = 5
```

Тепер ми можемо визначити клас Meteor:

```
class Meteor(GameObject):
    """ represents a meteor in the game """
    def __init__(self, game):
        super().__init__()
        self.game = game
        self.x = random.randint(0, DISPLAY_WIDTH)
        self.y = INITIAL_METEOR_Y_LOCATION
```

```
            self.speed = random.randint(1, MAX_METEOR_SPEED)
            self.load_image('meteor.png')
        def move_down(self):
            """ Move the meteor down the screen """
            self.y = self.y + self.speed
            if self.y > DISPLAY_HEIGHT:
                self.y = 5
        def __str__(self):
            return 'Meteor(' + str(self.x) + ', ' + str(self.y) +
')'
```

Метод __init__() для класу Meteor має ті самі кроки, що і в Starship; різниця в тому, що координата x і швидкість генеруються випадковим чином. Зображення, яке використовується для Meteor, також відрізняється, оскільки це «meteor.png».

Ми також реалізували метод move_down(). Це по суті те саме, що і в Starships move_down().

Зауважте, що на цьому етапі ми можемо створити підклас GameObject під назвою MoveableGameObject (який розширює GameObject) і перенести операції переміщення вгору до цього класу, а класи Meteor і Starship розширять цей клас. Однак ми насправді не хочемо дозволяти метеорам рухатися в будь-якому місці на екрані.

Тепер ми можемо додати метеори до класу Game. Ми додамо нове глобальне значення, щоб вказати кількість початкових метеорів у грі:

```
INITIAL_NUMBER_OF_METEORS = 8
```

Далі ми ініціалізуємо новий атрибут для класу Game, який міститиме список Meteors. Ми будемо використовувати тут список, оскільки хочемо збільшити кількість метеорів у міру просування гри.

Щоб полегшити цей процес, ми будемо використовувати list comprehension:

```
# Set up meteors
self.meteors = [Meteor(self) for _ in range(0,
INITIAL_NUMBER_OF_METEORS)]
```

Тепер у нас є список метеорів, які потрібно відобразити. Таким чином, нам потрібно оновити цикл while методу play(), щоб намалювати не тільки зореліт, але й усі метеори:

```
# Draw the meteors and the starship
self.starship.draw()
for meteor in self.meteors:
    meteor.draw()
```

Кінцевим результатом є те, що набір метеорних об’єктів створюється у випадкових початкових місцях у верхній частині екрана:


Рис. 35 Поява метеорів

*Переміщення метеорів*

Тепер ми хочемо мати можливість переміщати метеори вниз по екрану, щоб у Starship було кілька об'єктів, яких слід уникати.

Ми можемо зробити це дуже легко, оскільки ми вже реалізували метод move_down() у класі Meteor. Тому нам потрібно лише додати цикл for до основної гри в цикл while, який переміщуватиме всі метеори. Наприклад:

```
# Move the Meteors
for meteor in self.meteors:
    meteor.move_down()
```

Це можна додати після обробки події для циклу і до оновлення/перемальовування або оновлення екрана.

Тепер, коли ми запускаємо гру, метеори рухаються, і гравець може переміщатися зоряним кораблем між падаючими метеорами.

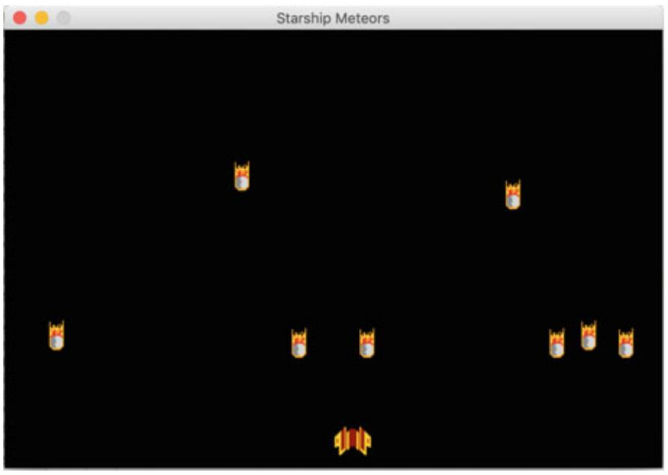


Рис. 36 Рух метеорів

*Виявлення зіткнення*

На даний момент гра буде грати вічно, оскільки немає кінцевого стану і немає спроб визначити, чи зіткнувся зоряний корабель з метеором.

Ми можемо додати виявлення зіткнення Meteor/Starship за допомогою PyGame Rects. Rect — це клас PyGame, який використовується для представлення прямокутних координат. Це особливо корисно, оскільки клас pygame.Rect надає кілька методів виявлення зіткнень, які можна використовувати для перевірки, чи знаходиться один прямокутник (або точка) всередині

іншого прямокутника. Тому ми можемо використовувати один із методів, щоб перевірити, чи перетинається прямокутник навколо Starship з будь-яким із прямокутників навколо метеорів.

Клас GameObject вже надає метод rect(), який повертає об'єкт Rect, що представляє поточний прямокутник об'єктів відносно поверхні малювання (по суті, поле навколо об'єкта, що відображає його розташування на екрані).

Таким чином, ми можемо написати метод виявлення зіткнень для класу Game, використовуючи rects, згенерований GameObject, і метод colliderect() класу Rect:

```
def _check_for_collision(self):
    """ Checks to see if any of the meteors have collided
        with the starship """
    result = False
    for meteor in self.meteors:
        if self.starship.rect().colliderect(meteor.rect()):
            result = True
            break
    return result
```

Зауважте, що тут ми дотримувались угоди щодо передування імені метода нижнього підкреслення, яка вказує, що цей метод слід вважати приватним для класу. Тому він ніколи не повинен викликатися нічим за межами класу Game. Ця угода визначена в PEP 8.

Тепер ми можемо використовувати цей метод в основному циклі while гри, щоб перевірити наявність зіткнення:

```
# Check to see if a meteor has hit the ship
if self._check_for_collision():
    starship_collided = True
```

Цей фрагмент коду також представляє нову локальну змінну starship_collided. Спочатку ми встановимо для цього значення False, і це ще одна умова, за якої основна гра в циклі while завершиться:

```
is_running = True
starship_collided = False
# Main game playing Loop
while is_running and not starship_collided:
```

Таким чином, цикл гри завершиться, якщо користувач вирішить вийти або зіткнеться зоряний корабель з метеором.

*Визначення виграшу*

Наразі у нас є спосіб програти гру, але у нас немає способу виграти гру! Проте ми хочемо, щоб гравець міг виграти гру, вижившипротягом певного періоду часу. Ми могли б представити це за допомогою якогось таймера. Однак у нашому випадку ми представимо виграш у вигляді певної кількості циклів основного ігрового циклу. Якщо гравець вижив протягом такої кількості циклів, то він виграв. Наприклад:

```
# See if the player has won
if cycle_count == MAX_NUMBER_OF_CYCLES:
    print('WINNER!')
    break
```

У цьому випадку роздруковується повідомлення про те, що гравець виграв, а потім основний ігровий цикл припиняється (за допомогою оператора break).

Глобальне значення MAX_NUMBER_OF_CYCLES можна встановити відповідним чином, наприклад:

```
MAX_NUMBER_OF_CYCLES = 1000
```

*Збільшення кількості метеорів*

Ми могли б залишити гру як є на даний момент, оскільки тепер можна виграти або програти гру. Однак є кілька речей, які можна легко додати, які покращать враження від гри. Одним з них є збільшення кількості метеорів на екрані, що ускладнює процес у ході гри.

Ми можемо зробити це за допомогою NEW_METEOR_CYCLE_INTERVAL.

```
NEW_METEOR_CYCLE_INTERVAL = 40
```

Коли цей інтервал буде досягнутий, ми можемо додати новий метеор до списку поточних метеорів; потім він буде автоматично відображений класом Game. Наприклад:

```
# Determine if new meteors should be added
if cycle_count % NEW_METEOR_CYCLE_INTERVAL == 0:
    self.meteors.append(Meteor(self))
```

Тепер кожен NEW_METEOR_CYCLE_INTERVAL ще один метеор буде додаватися до гри за випадковою координатою x.

*Призупинення гри*

Ще одна особливість багатьох ігор - це можливість призупинити гру. Це можна легко додати, відстежуючи клавішу паузи (це може бути буква p, представлена event_key pygame.K_p). Якщо натиснути цю кнопку, гру можна призупинити до повторного натискання клавіші.

Операцію паузи можна реалізувати як метод _pause(), який буде поглинати всі події, доки не буде натиснута відповідна клавіша. Наприклад:

```
def _pause(self):
    paused = True
    while paused:
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_p:
                    paused = False
                    break
```

У цьому методі зовнішній цикл while буде зациклюватися до тих пір, поки локальна змінна paused не стане False. Це відбувається лише при натисканні клавіші «p». Оператор break після установки змінної paused в False гарантує, що внутрішній цикл for закінчиться, дозволяючи зовнішньому циклу while перевірити значення paused та завершитись.

Метод _pause() можна викликати під час ігрового циклу, відстежуючи клавішу «p» у циклі для події та викликаючи звідти метод _pause():

```
elif event.key == pygame.K_p:
    self._pause()
```

*Відображення повідомлення про закінчення гри*

PyGame не має простого способу створення спливаючого діалогового вікна для відображення таких повідомлень, як «Ви виграли»; або «Ви програли», тому досі ми використовували оператори print. Однак ми можемо використовувати для цього бібліотеку графічного інтерфейсу, наприклад wxPython, або відобразити повідомлення на поверхні дисплея, щоб вказати, виграв гравець чи програв.

Ми можемо відобразити повідомлення на поверхні дисплея за допомогою класу pygame.font.Font. Тому додамо метод _display_message() до класу Game, який можна використовувати для відображення відповідних повідомлень:

```
def _display_message(self, message):
    """ Displays a message to the user on the screen """
    text_font = pygame.font.Font('freesansbold.ttf', 48)
    text_surface=text_font.render(message, True, BLUE, WHITE)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center=(DISPLAY_WIDTH/2, DISPLAY_HEIGHT/2)
    self.display_surface.fill(WHITE)
    self.display_surface.blit(text_surface, text_rectangle)
```

Тут провідне нижнє підкреслення в назві методу вказує, що його не слід викликати з-за меж класу Game.

Тепер ми можемо змінити основний цикл так, щоб користувачеві відображалися відповідні повідомлення, наприклад:

```
# Check to see if a meteor has hit the ship
if self._check_for_collision():
    starship_collided = True
    self._display_message('Collision: Game Over')
```

Результат виконання наведеного вище коду під час зіткнення показаний нижче:


Рис. 37 Повідомлення про закінчення гри

***Гра StarshipMeteors (повний листинг)***

Нижче наведено повний листинг остаточної версії гри StarshipMeteors [16]:

```
import pygame, random, time

FRAME_REFRESH_RATE = 30
DISPLAY_WIDTH = 600
DISPLAY_HEIGHT = 400
BLUE = (0, 0, 255)
```

```
WHITE = (255, 255, 255)
BACKGROUND = (0, 0, 0)
INITIAL_METEOR_Y_LOCATION = 10
INITIAL_NUMBER_OF_METEORS = 8
MAX_METEOR_SPEED = 5
STARSHIP_SPEED = 10
MAX_NUMBER_OF_CYCLES = 1000
NEW_METEOR_CYCLE_INTERVAL = 40

class GameObject:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.image = None
        self.width = 0
        self.height = 0
    def load_image(self, filename):
        self.image = pygame.image.load(filename).convert()
        self.width = self.image.get_width()
        self.height = self.image.get_height()
    def rect(self):
        """ Generates a rectangle representing the objects location
        and dimensions """
        return pygame.Rect(self.x, self.y, self.width, self.height)
    def draw(self):
        """ draw the game object at the
            current x, y coordinates """
        self.game.display_surface.blit(self.image,          (self.x,
self.y))

class Starship(GameObject):
    """ Represents a starship"""
    def __init__(self, game):
        super().__init__()
        self.game = game
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')
    def move_right(self):
        """ moves the starship right across the screen """
        self.x = self.x + STARSHIP_SPEED
        if self.x + self.width > DISPLAY_WIDTH:
            self.x = DISPLAY_WIDTH - self.width
    def move_left(self):
        """ Move the starship left across the screen """
        self.x = self.x - STARSHIP_SPEED
        if self.x < 0:
            self.x = 0

    def move_up(self):
        """ Move the starship up the screen """
        self.y = self.y - STARSHIP_SPEED
        if self.y < 0:
            self.y = 0
    def move_down(self):
        """ Move the starship down the screen """
        self.y = self.y + STARSHIP_SPEED
```

```
            if self.y + self.height > DISPLAY_HEIGHT:
                self.y = DISPLAY_HEIGHT - self.height
        def __str__(self):
            return 'Starship(' + str(self.x) + ', ' + str(self.y) + ')'

    class Meteor(GameObject):
        """ represents a meteor in the game """
        def __init__(self, game):
            super().__init__()
            self.game = game
            self.x = random.randint(0, DISPLAY_WIDTH)
            self.y = INITIAL_METEOR_Y_LOCATION
            self.speed = random.randint(1, MAX_METEOR_SPEED)
            self.load_image('meteor.png')
        def move_down(self):
            """ Move the meteor down the screen """
            self.y = self.y + self.speed
            if self.y > DISPLAY_HEIGHT:
                self.y = 5
        def __str__(self):
            return 'Meteor(' + str(self.x) + ', ' + str(self.y) + ')'

    class Game:
        """ Represents the game itself, holds the main game playing
            loop """
        def __init__(self):
            pygame.init()
            # Set up the display
            self.display_surface =
pygame.display.set_mode((DISPLAY_WIDTH, DISPLAY_HEIGHT))
            pygame.display.set_caption('Starship Meteors')
            # Used for timing within the program.
            self.clock = pygame.time.Clock()
            # Set up the starship
            self.starship = Starship(self)
            # Set up meteors
            self.meteors = [Meteor(self) for _ in range(0,
INITIAL_NUMBER_OF_METEORS)]
        def _check_for_collision(self):
            """ Checks to see if any of the meteors have collided with
                the starship """
            result = False
            for meteor in self.meteors:
                if self.starship.rect().colliderect(meteor.rect()):
                    result = True
                    break
            return result
        def _display_message(self, message):
            """ Displays a message to the user on the screen """
            text_font = pygame.font.Font('freesansbold.ttf', 48)
            text_surface = text_font.render(message, True, BLUE, WHITE)
            text_rectangle = text_surface.get_rect()
            text_rectangle.center = (DISPLAY_WIDTH/2, DISPLAY_HEIGHT/2)
            self.display_surface.fill(WHITE)
            self.display_surface.blit(text_surface, text_rectangle)
        def _pause(self):
            paused = True
```

```
        while paused:
            for event in pygame.event.get():
                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_p:
                        paused = False
                        break
    def play(self):
        is_running = True
        starship_collided = False
        cycle_count = 0

        # Main game playing Loop
        while is_running and not starship_collided:
         # Indicates how many times the main game loop has been run
            cycle_count += 1

            # See if the player has won
            if cycle_count == MAX_NUMBER_OF_CYCLES:
                self._display_message('WINNER!')
                break

            # Work out what the user wants to do
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    is_running = False
                elif event.type == pygame.KEYDOWN:
                    # Check to see which key is pressed
                    if event.key == pygame.K_RIGHT:
                        # Right arrow key has been pressed
                        # move the player right
                        self.starship.move_right()
                    elif event.key == pygame.K_LEFT:
                        # Left arrow has been pressed
                        # move the player left
                        self.starship.move_left()
                    elif event.key == pygame.K_UP:
                        self.starship.move_up()
                    elif event.key == pygame.K_DOWN:
                        self.starship.move_down()
                    elif event.key == pygame.K_p:
                        self._pause()
                    elif event.key == pygame.K_q:
                        is_running = False
            # Move the Meteors
            for meteor in self.meteors:
                meteor.move_down()
            # Clear the screen of current contents
            self.display_surface.fill(BACKGROUND)

            # Draw the meteors and the starship
            self.starship.draw()
            for meteor in self.meteors:
                meteor.draw()

            # Check to see if a meteor has hit the ship
            if self._check_for_collision():
                starship_collided = True
```

```
                self._display_message('Collision: Game Over')

            # Determine if new mateors should be added
            if cycle_count % NEW_METEOR_CYCLE_INTERVAL == 0:
                self.meteors.append(Meteor(self))

            # Update the display
            pygame.display.update()

 #Defines the frame rate. The number is number of frames per second
 # Should be called once per frame (but only once)
            self.clock.tick(FRAME_REFRESH_RATE)

        time.sleep(1)
        # Let pygame shutdown gracefully
        pygame.quit()
def main():
    print('Starting Game')
    game = Game()
    game.play()
    print('Game Over')
if __name__ == '__main__':
    main()
```

**Питання для самоконтролю**

1. Як інсталювати Pygame?
2. Дайте характеристику моделі подій Pygame.
3. Яка функція створює дисплейну поверхню?
4. Як виявити зіткнення об’єктів гри?

**Навчальне завдання №4**

**Розробка додатків на Python з графічним інтерфейсом користувача і використання бібліотек Matplotlib, NumPy**

Розробити застосунок з GUI згідно варіанту завдання. Використати бібліотеки NumPy і Matplotlib.

**Завдання**:

**Вправа 1** (для всіх варіантів).

Встановіть matplotlib. Накресліть діаграму розсіювання (scatter diagram) цих пар (x, y): ( (0, 0), (3, 5), (6, 2), (9, 8), (14, 10) ). Намалюйте лінійний графік тих самих даних. Намалюйте графік (лінійний графік з маркерами) тих самих даних.

Рішення.

Запустіть VS Code і перейдіть в директорію лабораторних робіт нашого курсу (наприклад, Python_Labs).

Активізуйте вікно терміналу.

В вікні терміналу наберіть:

```
pip install matplotlib
```

Після встановлення бібліотеки створіть новий файл (lab4_Ex1.py) з таким вмістом:

```
import matplotlib.pyplot as plt

x = (0, 3, 6, 9, 14)
y = (0, 5, 2, 8, 10)

fig, plots = plt.subplots(nrows=1, ncols=3)
plots[0].scatter(x, y)
plots[1].plot(x, y)
plots[2].plot(x, y, 'o-')
plt.show()
```

Запустіть файл на виконання. Ви отримаєте наступне:

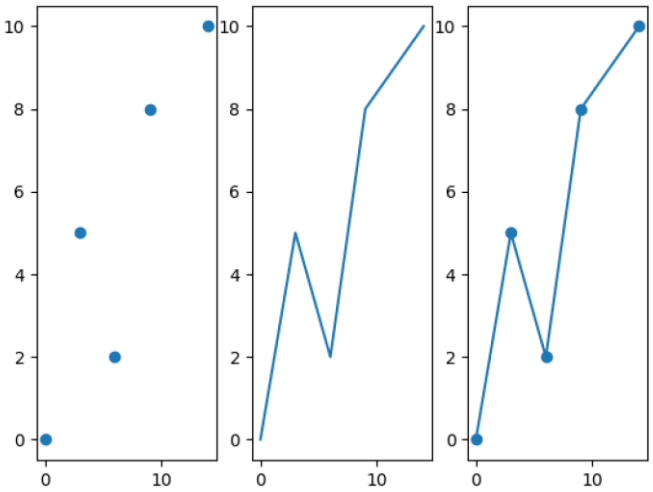


Рис. 38 Результат виконання lab4_Ex1.py

**Вправа 2** (для всіх варіантів).

Ми розробимо класи для чисельного інтегрування [2].

Так само, як чисельне диференціювання, чисельне інтегрування є основою обчислювальної математики. Існує безліч методів на вибір, і всі вони можуть бути записані у формі:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i).$$

На основі цієї загальної формули різні методи реалізуються шляхом вибору точок інтегрування $x_i$ та відповідних ваг $w_i$. Наприклад, ***формула трапецій*** має вигляд:

$$x_i = a + ih, \quad w_0 = w_{n-1} = \frac{h}{2}, \; w_i = h \;\; (i \neq 0, n-1),$$

де h = (b − a)/(n − 1), серединний метод (***midpoint rule***) має вигляд:

$$x_i = a + \frac{h}{2} + ih, \quad w_i = h,$$

де h = (b−a)/n, а ***формула Сімпсона*** (Simpson) має вигляд:

$$x_i = a + ih, \quad h = \frac{b-a}{n-1},$$

$$w_0 = w_{n-1} = \frac{h}{6},$$

$$w_i = \frac{h}{3} \text{ for } i \text{ even}, \quad w_i = \frac{2h}{3} \text{ for } i \text{ odd}.$$

Інші методи мають більш складні формули для $w_i$ та $x_i$, а деякі методи вибирають точки випадковим чином (наприклад, інтегрування Монте-Карло).

Формула чисельного інтегрування може бути реалізована як клас з a, b і n як атрибутами та методом інтегрування для обчислення інтегралу.

Має сенс розмістити весь загальний код у базовому класі та код, специфічний для різних методів, у похідних класах. Тут ми можемо поставити $\sum_j w_j f(x_j)$

в базовий клас (метод integrate) і нехай підкласи доповнюють цей клас кодом, специфічним для певної формули, тобто вибором $w_i$ та $x_i$. Код цього методу можна помістити всередину методу, наприклад, з назвою construct_rule. Суперклас (базовий клас) для ієрархії чисельного інтегрування може виглядати так:

```
import numpy as np
class Integrator:
    def __init__(self, a, b, n):
        self.a, self.b, self.n = a, b, n
        self.points, self.weights = self.construct_method()

    def construct_method(self):
        raise NotImplementedError('no rule in class %s' % \
                                  self.__class__.__name__)
    def integrate(self, f):
        s = 0
        for i in range(len(self.weights)):
            s += self.weights[i]*f(self.points[i])
        return s

    def vectorized_integrate(self, f):
        # f must be vectorized for this to work
        return np.dot(self.weights, f(self.points))
```

Зверніть увагу на реалізацію construct_method, яка викликатиме помилку, якщо її викликати, вказуючи, що єдина мета інтегратора - це суперклас, і його не слід використовувати безпосередньо. З іншого боку, ми могли б, звичайно, просто не включати метод construct_method до суперкласу взагалі. Однак підхід, що використовується тут, робить ще більш очевидним, що клас - це просто суперклас і що цей метод потрібно реалізувати у підкласах.

Надклас забезпечує загальну основу для реалізації різних методів, які потім можуть бути реалізовані як підкласи. Метод трапецій та серединний метод можна реалізувати наступним чином:

```
class Trapezoidal(Integrator):
    def construct_method(self):
        h = (self.b - self.a)/float(self.n - 1)
        x = np.linspace(self.a, self.b, self.n)
        w = np.zeros(len(x))
        w[1:-1] += h
        w[0] = h/2; w[-1] = h/2
        return x, w

class Midpoint(Integrator):
    def construct_method(self):
        a, b, n = self.a, self.b, self.n # quick forms
        h = (b-a)/float(n)
        x = np.linspace(a + 0.5*h, b - 0.5*h, n)
        w = np.zeros(len(x)) + h
        return x, w
```

Більш складне правило Сімпсона можна додати до такого підкласу:

```
class Simpson(Integrator):
    def construct_method(self):
        if self.n % 2 != 1:
            print ('n=%d must be odd, 1 is added' % self.n)
            self.n += 1
        x = np.linspace(self.a, self.b, self.n)
        h = (self.b - self.a)/float(self.n - 1)*2
        w = np.zeros(len(x))
        w[0:self.n:2] = h*1.0/3
        w[1:self.n-1:2] = h*2.0/3
        w[0] /= 2
        w[-1] /= 2
        return x, w
```

Правило Сімпсона є більш складним, оскільки воно використовує різні ваги для непарних і парних точок. Тут ми наводимо всі деталі для повноти, але насправді не потрібно вивчати деталі всіх формул. Важливими частинами тут є дизайн класів та використання ієрархії класів.

Щоб продемонструвати, як можна використовувати класи, обчислимо інтеграл

$$\int_0^2 x^2 dx$$

Використовуючи 101 точку:

```
def f(x):
    return x*x

simpson = Simpson(0, 2, 101)
print(simpson.integrate(f))
trapez = Trapezoidal(0, 2, 101)
print(trapez.integrate(f))
```

Потік виконання програми в цьому випадку може бути не зовсім очевидним. Коли ми створюємо екземпляр за допомогою `simpson = Simpson (0, 2, 101)`, викликається конструктор суперкласу, але цей метод потім викликає construct_method у класі Simpson. Метод `simpson.integrate(f)` потім викликає метод інтегрування, успадкований від суперкласу. Однак, як для користувачів класу, жодна з цих деталей насправді не має значення. Ми використовуємо клас Simpson так, ніби всі методи були реалізовані безпосередньо в цьому класі, незалежно від того, що насправді вони успадковані від іншого класу.

**Завдання вправи**:

Сформувати файл integral.py, помістивши в нього наведені вище фрагменти коду. Виконати програму і проаналізувати результат. Чи зміниться результат, якщо задати 11 точок? Обчисліть точне значення інтегралу. Поясніть використані в програмі засоби бібліотеки NumPy.

**Варіант 1.**

**Вправа 1** (GUI).

В полі введення задана вага в кілограмах. При натисканні кнопки перевести вагу в грами, в фунти і в унції.

**Вправа 2** (matplotlib).

Зобразити 2d графік функції відповідно своєму варіанту та зберегти у .png файл.

Y(x)=x*sin(5*x), x=[-2...5]

**Варіант 2.**

**Вправа 1** (GUI).

Розробіть графічний інтерфейс до однієї з виконаних раніше лабораторних робіт.

**Вправа 2** (matplotlib).

Зобразити 2d графік функції відповідно своєму варіанту та зберегти у .png файл.

Y(x)=1/x*sin(5*x), x=[-5...5]

**Варіант 3.**

**Вправа 1** (GUI).

Розробіть графічний інтерфейс до однієї з виконаних раніше лабораторних робіт.

**Вправа 2** (matplotlib).

Зобразити 2d графік функції відповідно своєму варіанту та зберегти у .png файл.

Y(x)=2^x*sin(10x), x=[-3...3]

## ПИТАННЯ І ЗАВДАННЯ ДО ЗАЛІКУ

1. Python — мова, що інтерпретується чи компілюється?
2. Що таке рядки документації (Docstrings) в Python?
3. Як прокоментувати кілька рядків в Python?
4. Що таке локальні та глобальні змінні в Python?
5. Як обробляти вхідні дані в Python?
6. У чому різниця між тим, коли функція range () приймає один аргумент, два і три?
7. Напишіть кращий код для перестановки двох чисел місцями.
8. Як можна оголосити кілька призначень в одному рядку коду?
9. Що робить оператор with в Python?
10. Коли виконується except, в блоці try-except?
11. Які типи даних підтримуються в Python?
12. Що таке негативні індекси і чому вони використовуються?
13. Які є змінні та незмінні типи даних?
14. Що таке область видимості змінних?
15. Як отримати доступ до значень в словнику?
16. Що таке модулі Python?
17. Що таке PYTHONPATH?
18. Назвіть режими обробки файлів за допомогою Python?
19. Які є види імпорту?
20. Що таке ітератори в Python?
21. Що робить [:: -1}?
22. Який сенс оператора pass в Python?
23. Який сенс // в Python?
24. Що таке функція map в Python?
25. Що за функції help () і dir () в Python?
26. Що робить функція zip ()?
27. Що таке функція enumerate в Python?
28. Як додати нове значення в об'єкт списку?
29. Що таке поверхнева копія?
30. Що таке глибока копія?
31. Як створити порожній клас в Python?
32. Що означає ключове слово self в Python?

33. Як перетворити список рядків в рядок?
34. Що таке оператор членства (приналежності)?
35. Що таке оператори тотожності в Python?
36. Що таке клас, ітератор, генератор?
37. У чому різниця між ітераторами та генераторами?
38. У чому різниця між staticmethod та classmethod?
39. Як працюють декоратори?
40. Як працюють dict comprehension, list comprehension і set comprehension?
41. Що таке lambda-функції?
42. Що таке зріз?
43. Які є основні популярні пакети?
44. Що означає *args, **kwargs та як вони використовуються?
45. Що таке exceptions?
46. Що таке PEP, які з них знаєте?
47. Які є типи даних і яка різниця між list і tuple, навіщо вони?
48. Як прибрати зі списку дублікат елементу?
49. Як вставити об'єкт в список, щоб він опинився під певним індексом?
50. Як створюється об’єкт в Python, для чого __new__, навіщо __init__?
51. Що відбувається, коли створюється віртуальне середовище?
52. Чи є масиви Python NumPy краще списків?
53. Як отримати список з усіх ключів словника (dictionary keys)?
54. Як отримати поточний робочий каталог за допомогою Python?
55. Як можна видалити змінні в Python?
56. У чому різниця між методами append() та extend()?
57. У чому полягає складність доступу до елементів dict?
58. Як передаються аргументи функцій у Python (by value or reference)?
59. Що таке @property?
60. Що таке розпакування кортежу?
61. Напишіть програму на Python, щоб перевірити, чи є послідовність, яку ви вводите, паліндромом.

Що знаходиться в кожній із змінних?

```
A0 = dict(zip((‘a’,’b’,’c’,’d’,’e’),(1,2,3,4,5)))
A1 = range(10)
A2 = sorted([i for i in A1 if i in A0])
A3 = sorted([A0[s] for s in A0])
A4 = [i for i in A1 if i in A3]
A5 = {i:i*i for i in A1}
A6 = [[i,i*i] for i in A1]
```

## РЕКОМЕНДОВАНА ЛІТЕРАТУРА

**Основна:**


1. Васильєв О.М. Програмування мовою Python : Видавництво «Навчальна книга — Богдан», 2018. 503 с.
2. Програмування числових методів мовою Python : підруч. / А. В. Анісімов, А. Ю. Дорошенко, С. Д. Погорілий, Я. Ю. Дорогий; за ред. А. В. Анісімова. Київ : Видавничо-поліграфічний центр "Київський університет", 2014. 640 с.
3. Яковенко А.В. Основи програмування. Python. Частина 1. : Київ, КПІ ім. Ігоря Сікорського, 2018. 195 с.
4. Копей В.Б. Мова програмування Python для інженерів і науковців. Івано-Франківськ : ІФНТУНГ, 2019. 272 с.
5. Lubanovic B. Introducing Python. Modern Computing in Simple Packages: O'Reilly, 2020. 597 p.


**Додаткова:**


6. Stephenson B. The Python Workbook A Brief Introduction with Exercises and Solutions Second Edition. : Springer, 2019. 219 p.
7. Heisler F., Amos D., Bader D., Jablonski J. Python Basics. : Real Python (realpython.com), 2020. 641 p.
8. Gaddis T. Starting Out with Python, Fifth Edition: Pearson Education Limited, 2022. 892 p.
9. Костюченко А.О. Основи програмування мовою Python: навчальний посібник. – Чернігів: ФОП Баликіна С.М., 2020. 180 с.
10. Креневич А.П. Python у прикладах і задачах. Частина 1. Структурне програмування. Навчальний посібник із дисципліни "Інформатика та програмування" – К.: ВПЦ "Київський Університет", 2017. 206 с.
11. Лутц М. Изучаем Python, том 1, 5-е изд. — СПб.: ООО "Диалектика", 2019. 832 с.
12. Лутц М. Изучаем Python, том 2, 5-е изд. — СПб.: ООО "Диалектика", 2020. 720 с.
13. Sundnes J. Introduction to Scientific Programming with Python. –Springer, 2020. 148 p.
14. Danjou J. Serious Python: black-belt advice on deployment, scalability, testing, and more. – San Francisco, CA : No Starch Press, Inc., 2019. 258 p.
15. The Python Standard Library – [URL] https://docs.python.org/3.9/library/
16. Hunt J. Advanced Guide to Python 3 Programming. – Springer, 2019. 497 p.
17. Johansson R. Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib. – Apress, 2019. 700 p.

18. Hill C. Learning Scientific Programming with Python – Cambridge University Press, 2020. 557 p.
19. Campesato O. Python 3 and Data Analytics Pocket Primer. – Mercury Learning and information, 2021. 238 p.
20. Morley S. Applying Math with Python. – Packt Publishing, 2020. 336 p.
21. Matthes E. Python Crash Course. – No Starch Press, 2019. 506 p.
22. Romano F., Kruger H. Learn Python Programming Third Edition. – Packt Publishing, 2021. 527 p.

Навчально-методичне видання
(українською мовою)

Попівщий Василь Іванович
Безверхий Анатолій Ігорович

# ПРОГРАМУВАННЯ НА МОВІ PYTHON

Навчальний посібник
для здобувачів степеня вищої освіти бакалавра
спеціальності «Інженерія програмного забезпечення»
освітньо-професійної програми «Програмне забезпечення систем»

Рецензент *В.З. Грищак*
Відповідальний за випуск
Коректор *А.І. Безверхий*