МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

Серія: Електронні технології

В. В. Палагін, О. А. Палагіна, О. С. Зорін

Основи Python та програмування електронних систем

Черкаси
ЧДТУ
2024

УДК 004.43(075.8)
П54

Рекомендовано вченою радою Черкаського державного технологічного університету, протокол № 10 від 27.05.2024.

Рецензенти: *С. А. Положаєнко,* д-р техн. наук, професор, завідувач кафедри комп'ютеризованих систем і програмних технологій Національного університету «Одеська політехніка»;
Р. С. Одарченко, д-р техн. наук, професор, в.о. декана факультету аеронавігації, електроніки та телекомунікацій Національного авіаційного університету;
В. А. Федорчук, д-р техн. наук, професор, професор кафедри комп'ютерних наук Кам'янець-Подільського національного університету імені Івана Огієнка

Палагін В. В. Основи Python та програмування електронних систем : [навч. посіб.] [Електронний ресурс] / В. В. Палагін, О. А. Палагіна, О. С. Зорін ; М-во освіти і науки України, Черкас. держ. технол. ун-т. – Черкаси : ЧДТУ, 2024. – 216 с. – (Серія : Електронні технології). – Назва з титульного екрана.
П54
ISBN 978-966-402-135-4

Практикум з основ Python та програмування електронних систем містить теоретичні відомості про основи програмування цією мовою, основні синтаксичні конструкції та бібліотеки, приклади реалізації різноманітних програмних кодів. Подано інформацію про середовище розробки та його налаштування. Практичне застосування мови Python продемонстровано на прикладі програмування мікрокомп'ютера Raspberry Pi.

Для здобувачів вищої освіти, які навчаються за галузями знань «Електроніка та телекомунікації», «Інформаційні технології», «Математика та статистика» за відповідними програмами бакалаврської та магістерської підготовки.

УДК 004.43(075.8)

Навчальне електронне видання

Палагін Володимир Васильович
Палагіна Олена Анатоліївна
Зорін Олександр Сергійович

Основи Python та програмування електронних систем

Коректор *Тамара Костенко*
Технічний редактор *Тетяна Манжура*

Гарнітура Times New Roman. Обл.-вид. арк. 15,65. Зам. № 24-70.

Черкаський державний технологічний університет
Свідоцтво про державну реєстрацію ДК № 896 від 16.04.2002.
бульвар Шевченка, 460, м. Черкаси, 18006.
Редакційно-видавничий відділ ЧДТУ
red_vidav@chdtu.edu.ua

ЗМІСТ

ПЕРЕДМОВА

Програмування електронних систем відіграє ключову роль у функціонуванні сучасних пристроїв та систем – від простих мікроконтролерів, які керують побутовою технікою, до складних обчислювальних систем, що лежать в основі Інтернету речей (IoT), автомобілів з автопілотом, смартфонів, медичного обладнання та багатьох інших.

Програмне забезпечення слугує мостом між апаратним забезпеченням і користувачем, дозволяючи ефективно керувати ресурсами та функціями електронного пристрою. Наприклад, драйвери пристроїв забезпечують комунікацію між операційною системою та апаратними компонентами. Від базових функцій, таких як обробка тексту в додатках для смартфонів, до складних алгоритмів машинного навчання, що керують автономними транспортними засобами, – все це забезпечується за допомогою програмування.

Програмне забезпечення можна легко оновлювати, надаючи можливість вносити покращення та виправляти помилки без зміни апаратної частини. Це робить електронні системи гнучкішими та здатними адаптуватися до нових вимог і технологій.

У сучасному світі електронні системи часто потрібно інтегрувати та змушувати взаємодіяти одна з одною. Програмування дозволяє створювати стандартизовані протоколи та інтерфейси для обміну даними, що є критично важливим для роботи таких систем, як IoT.

Python став популярним вибором для програмування електронних систем, особливо у сферах, де не вимагається висока продуктивність апаратного забезпечення або можливе використання високорівневих абстракцій і простота розробки є переважними. Його застосування включає, але не обмежується такими сферами, як програмування мікроконтролерів та одноплатних комп’ютерів. Python використовується в одноплатних комп’ютерах, таких як Raspberry Pi, через бібліотеки, такі як RPi.GPIO для керування GPIO пінами, а також на платформах, що підтримують MicroPython або CircuitPython, спрощені версії Python для мікроконтролерів, таких як ESP8266, ESP32, і мікроконтролери серії STM32.

Python використовується для розробки IoT-пристроїв і систем, що вимагають інтеграції з різними протоколами зв’язку. Його використання спрощує реалізацію складних мережевих операцій і взаємодії між пристроями.

Python дозволяє швидко створювати прототипи електронних систем і пристроїв, забезпечуючи велику швидкість розробки завдяки зручності

синтаксису та великій кількості готових бібліотек для різноманітних застосувань.

Python є чудовим інструментом для освітніх проєктів у сфері електроніки та робототехніки, допомагаючи студентам засвоїти основи програмування і взаємодії з апаратним забезпеченням. Завдяки своїй універсальності, великій спільноті та розвиненій екосистемі бібліотек Python продовжує залишатися важливим інструментом для розробників електронних систем, які прагнуть швидко та ефективно реалізовувати свої ідеї.

Навчальний посібник «Основи Python та програмування електронних систем» знайомить студентів з інтегрованим середовищем розробки IDE (Integrated Development Environment), що надає комплексні можливості для розробників програмного забезпечення на прикладі застосування платформи Jupyter.

Видання містить відомості про теоретичні засади програмування мовою Python, основні синтаксичні конструкції та бібліотеки, приклади реалізації різноманітних програмних кодів. Практичне застосування мови Python продемонстровано на прикладі програмування мікрокомп’ютера Raspberry Pi, наведено інформацію про інсталяцію операційної системи, налаштування віддаленого доступу та реалізацію практичних задач на базі Raspberry Pi.

1 ПОЧАТОК РОБОТИ В СЕРЕДОВИЩІ IDE JUPYTER. ОСНОВНІ СИНТАКСИЧНІ СТРУКТУРИ МОВИ PYTHON

Мета практичної роботи № 1: ознайомитися з інсталяцією та налаштуванням середовища розробки IDE Jupyter Notebook для програмування мовою Python, основними синтаксичними конструкціями та реалізацією програмних кодів

ТЕОРЕТИЧНІ ПОЛОЖЕННЯ

1.1 Початок роботи та встановлення Python

Сучасне застосування Python для програмування електронних систем стає все більш поширеним і зростає з кожним роком. Відомий своєю простотою, зручністю та потужністю, Python знаходить широке застосування у великому спектрі галузей програмування електроніки.

Python – високорівнева мова програмування загального призначення, орієнтована на підвищення продуктивності розробника і читання коду. Python підтримує декілька парадигм програмування, зокрема структурну, об’єктно-орієнтовану, функціональну, аспектно-орієнтовану.

Нині існують дві версії реалізації Python – 2.x та 3.x. Ми надалі будемо працювати з версією 3.x.

Існують різні інтерпретатори для мови Python. Офіційний інтерпретатор можна завантажити з офіційного сайту за посиланням https://www.python.org, де можна обрати необхідну версію, наприклад **Python 3.10.x** або вище (рисунок 1.1).

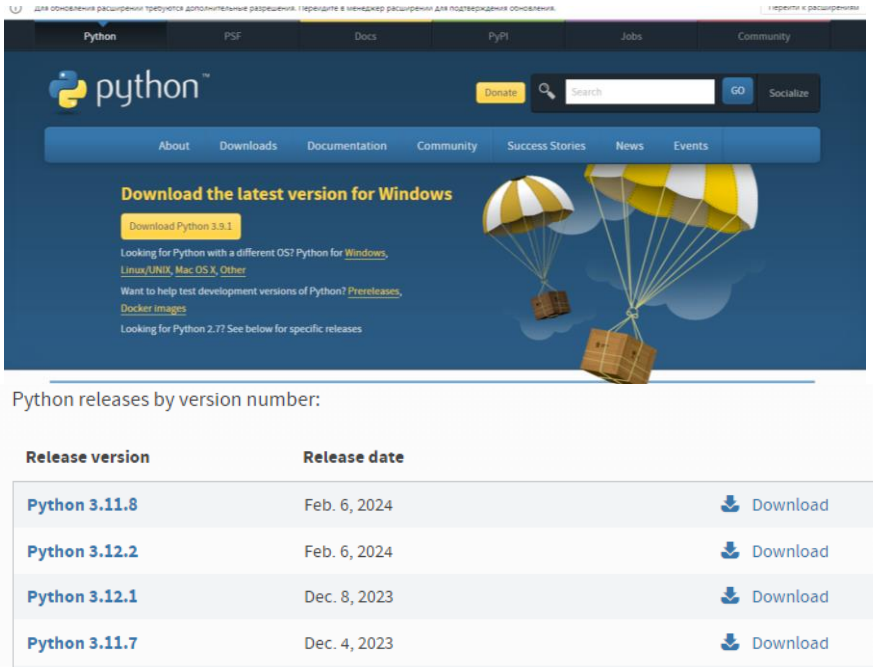


Рисунок 1.1 – Завантаження інтерпретатора мови Python з офіційного сайту

З чого потрібно розпочати? З встановлення дистрибутиву Python на комп'ютер після скачування з офіційного сайту. Не забудьте встановити прапорець «**Add Python 3.x to PATH**» – це полегшить правильне налаштування системи (рисунок 1.2).

Рисунок 1.2 – Встановлення прапорця «**Add Python 3.x to PATH**»

Для того щоб перевірити встановлення Python на комп'ютер і його версію, потрібно з командного рядка (термінала) **cmd** ввести команду ***python*** (рисунок 1.2), в результаті чого має запуститися інтерактивна версія Python і з'явитися наступна інформація (рисунок 1.3).

```
C:\Users\Admin>python
Python 3.10.10 (main, Feb 10 2023, 19:20:41)  [GCC 12.2.0 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 3+7
10
>>> for i in range(5):
...     print ("i= ", i)
...
i=  0
i=  1
i=  2
i=  3
i=  4
>>>
```

Рисунок 1.3 – Запуск Python з термінала

Навіть у такому варіанті запуску Python його можна вже використовувати для елементарних обрахунків, наприклад, як калькулятор при додаванні чисел, написанні циклічної структури програми тощо. Для більш зручного написання програми та її налагодження можна

скористатися вбудованим редактором Python, який запускається через головне меню комп'ютера. На рисунку 1.4 наведено фрагмент програми в такому редакторі і результат її запуску.

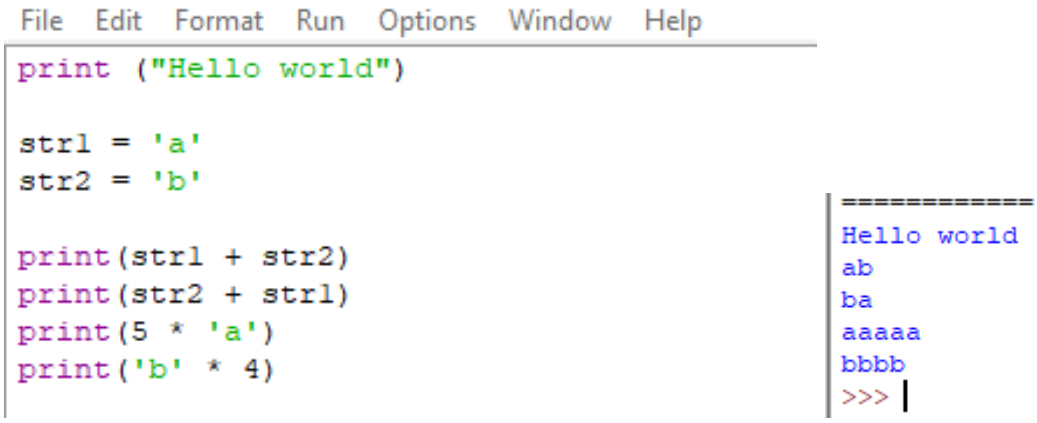


Рисунок 1.4 – Демонстрація написання програми
у вбудованому редакторі Python

Окрім того, існують так звані середовища розробки IDE (Integrated Development Environment), що спрощує взаємодію розробника з програмним кодом і візуалізацією його роботи. Серед цих середовищ можна виділити такі:

PyCharm – одне з найпопулярніших IDE для Python, розроблене компанією JetBrains. PyCharm пропонує потужні можливості для професійної розробки на Python, включаючи підтримку веб-розробки, наукових обчислень, роботу з базами даних тощо;

Jupyter Notebook – інтерактивне середовище, особливо популярне серед науковців, інженерів даних та аналітиків для експлоративного аналізу даних (EDA), візуалізації та машинного навчання;

Visual Studio Code (VS Code) – легкий, але потужний редактор коду від Microsoft, який підтримує Python через розширення та пропонує широкі можливості для розробки, включаючи інтеграцію з Git, різні інструменти для відлагодження та багато іншого;

Spyder – Spyder є IDE, популярний серед науковців, інженерів та аналітиків даних. Включає вбудовані інструменти для наукових обчислень, як-от IPython (Interactive Python), NumPy, SciPy, Matplotlib і Pandas;

Anaconda – це потужна дистрибуція Python, яка спрямована на спрощення роботи з науковими обчисленнями, аналізом даних та машинним навчанням. Вона забезпечує легкий доступ до великої кількості пакетів та інструментів для наукових досліджень, а також управління залежностями та проєктами (рисунок 1.5).

Ось деякі з ключових особливостей та компонентів Anaconda:

- управління пакетами з Conda – це менеджер пакетів, розроблений спеціально для наукових пакетів Python (хоча він також може управляти пакетами для інших мов програмування, таких як R, Scala

тощо). Він дозволяє легко встановлювати, запускати й оновлювати наукові бібліотеки та програми без конфліктів залежностей;

- велика екосистема – Anaconda включає понад 1500 пакетів для наукових досліджень, обробки даних, аналізу даних, машинного навчання тощо. Серед них – відомі інструменти, такі як NumPy, Pandas, SciPy, Matplotlib, Jupyter, а також багато інших;
- інтеграція з хмарними сервісами – Anaconda може інтегруватися з різними хмарними платформами та сервісами для обробки даних, що робить її відмінним вибором для проєктів, що вимагають значних обчислювальних ресурсів.

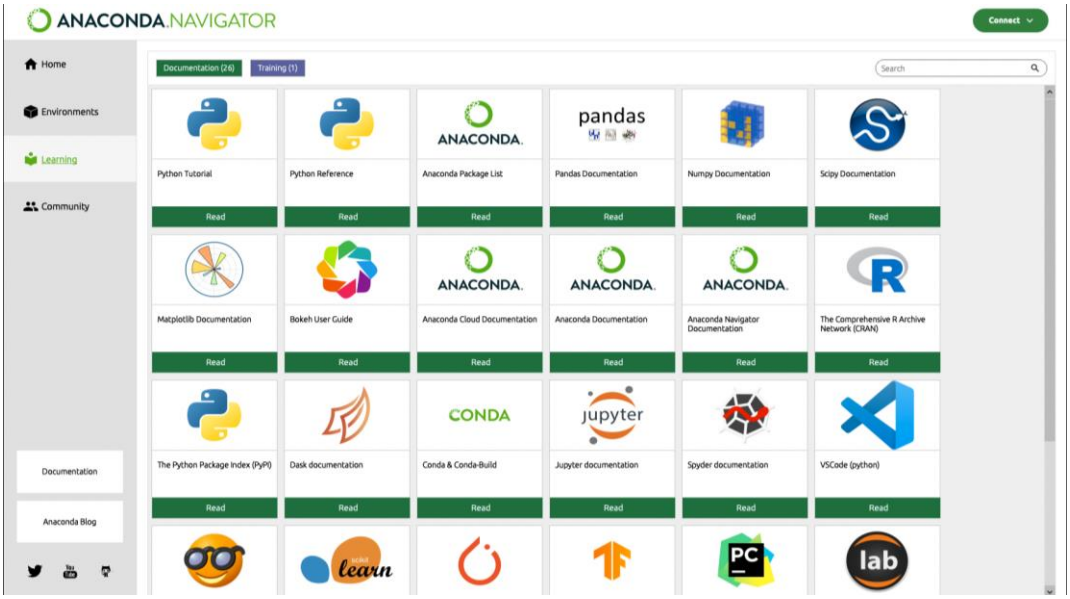


Рисунок 1.5 – Зовнішній вигляд додатка Anaconda Navigator

1.2 Встановлення та налаштування IDE Jupyter Notebook

Для проведення лабораторних робіт і дослідження синтаксису та різноманітних структур мови програмування Python у цьому практикумі пропонується встановити і використовувати середовище **Jupyter.**

Jupyter є відкритим проєктом, який підтримує інтерактивне програмування в браузері, найчастіше використовується для наукових обчислень, візуалізації даних, машинного навчання та інших областей аналізу даних. Його найбільш відомим компонентом є Jupyter Notebook, інструмент, що дозволяє створювати та ділитися документами, які містять живий код, рівняння, візуалізації та пояснювальний текст.

Jupyter Notebook дозволяє виконувати код по частинах (в комірках), що сприяє експериментуванню та ітеративному аналізу.

Jupyter Notebook підтримує різноманітні мови програмування через розширення, відомі як «ядра». До них відносяться R, Julia, Scala, Ruby та багато інших. Інтеграція з бібліотеками візуалізації, такими як Matplotlib, Seaborn, Plotly, дозволяє легко створювати інтерактивні графіки та візуалізації прямо в Notebook.

Notebook можна експортувати у різні формати, включаючи HTML, PDF, Markdown, що робить їх зручними для спільного використання та публікації.

Через велику кількість плагінів і розширень Jupyter можна адаптувати під конкретні потреби користувача. Це включає інструменти для кодування, візуалізації, інтеграції з базами даних тощо. Jupyter Notebook часто використовується для навчальних матеріалів, оскільки дозволяє комбінувати пояснення з виконуваним кодом та результатами.

Як почати з Jupyter Notebook

Для початку роботи з Jupyter Notebook його потрібно інсталювати на комп'ютер. Одним із можливих варіантів встановлення є інсталяція через середовище Anaconda, популярний дистрибутив Python, який вже містить Jupyter і багато інших корисних наукових бібліотек. Однак такий шлях буде потребувати додаткових системних ресурсів, оскільки сама по собі Anaconda вжє є досить «важкою» платформою.

Пропонується альтернативний варіант, який прискорює встановлення IDE Jupyter Notebook при збереженні всього набору його функціональних можливостей через застосування менеджера пакетів Python **«pip install»** з термінала. Для цього потрібно в терміналі виконати команду **pip install jupyter**:

```
C:\Users\Admin>pip install jupyter
```

після чого Jupyter Notebook буде вже встановлений на комп'ютері і готовий до використання.

Для запуску Jupyter Notebook потрібно в терміналі виконати команду «**jupyter-notebook»:**

```
C:\Users\Admin>jupyter-notebook
```

після чого запуститься середовище розробки в браузері, зображене на рисунку 1.6.

Головне меню в IDE Jupyter Notebook є інтуїтивно зрозумілим і дозволяє швидко виконувати такі операції, як завантаження файлу, зберігання тощо через меню «File». Запуск програми відбувається при натисканні кнопки «Run», а керування процесами запуску і зупинки знаходиться в меню «Kernel». Для створення нового файлу використовують «New Notebook» в меню «File».

Особливістю роботи в цьому середовищі є те, що розробник може створювати незалежні програмні фрагменти коду в різних полях і запускати їх окремо або разом. Додавання нових полів здійснюється через піктограму «+» головного меню (рисунок 1.7), виділену червоним колом #1. В результаті цього з'являється новий рядок, готовий для введення програми або її фрагменту і незалежного запуску через «Run». Наприклад, фрагменти програми в рядках 1 і 2 (рисунок 1.7) були

запущені окремо, що дозволяє проводити налагодження коду або його фрагментів незалежно один від одного в одному робочому файлі і середовищі без необхідності коментування чи створення інших файлів. Необхідний коментар до відповідних полів можна створювати за допомогою піктограми #2, де обирається відповідний розмір (від найбільшого #1 до найменшого #6).

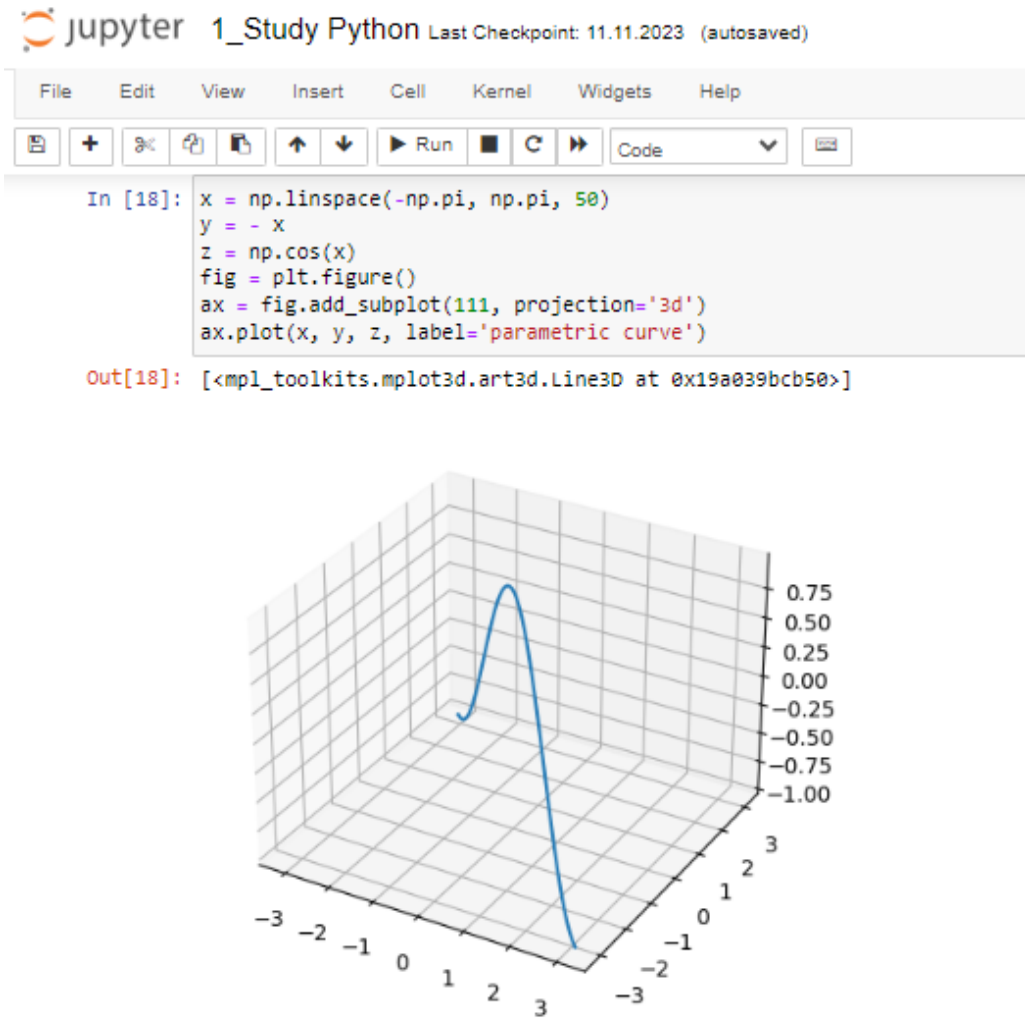


Рисунок 1.6 – Приклад програми в IDE Jupyter Notebook

При запуску Jupyter Notebook з термінала за замовчуванням середовище розробки відриває системну папку на диску C:\, що не завжди зручно. Для роботи з конкретною папкою, наприклад на диску D:\, необхідно запустити Jupyter Notebook з термінала таким чином:

jupyter notebook --notebook-dir=‘D:\Python’,

```
C:\Users\Admin>jupyter notebook --notebook-dir=‘D:\Python’
```

де ‘D:\Python’ – шлях до робочої папки, де будуть зберігатися файли для роботи jupyter notebook.

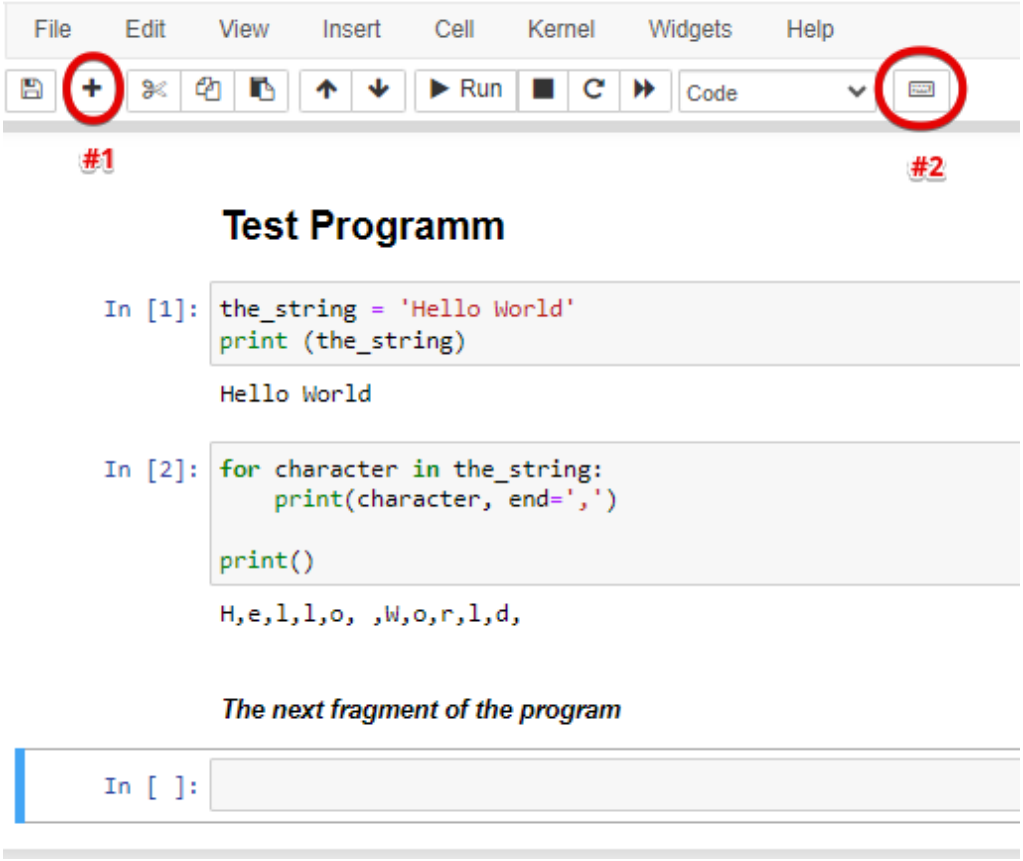


Рисунок 1.7 – Демонстрація додавання робочих полів в Jupyter Notebook

Відзначимо, що файли jupyter notebook мають своє унікальне розширення ***.ipynb** і можуть бути зконвертовані в стандартний формат мови програмування Python ***.py** при застосуванні команд головного меню «File» → «Download as» → «Python» (рисунок 1.8) або в інший формат для зручного перегляду, наприклад ***.html.**

Для постійного налаштування робочого каталогу в середовищі jupyter-notebook, щоб не вводити з термінала кожного разу шлях до робочої папки, можна виконати такі налаштування.

1) З командного рядка (cmd) запускаємо команду створення файлу конфігурації:

jupyter notebook --generate-config.

2) Знаходимо файл **jupyter_notebook_config.py** (який розташований, наприклад в *C:\Users\Admin\.jupyter*), відкриваємо довільним редактором, наприклад NotePad, шукаємо рядок

c.NotebookApp.notebook_dir

і розкоментовуємо його (видаляємо службовий знак #), вносимо зміни, наприклад

c.NotebookApp.notebook_dir = 'D:\Python'

та записуємо файл.

3) Запуск ноутбука може відбуватися за одним із варіантів:
3.1) з командного рядка **jupyter-notebook,**
3.2) або можемо створити ярлик для його запуску на робочому столі.

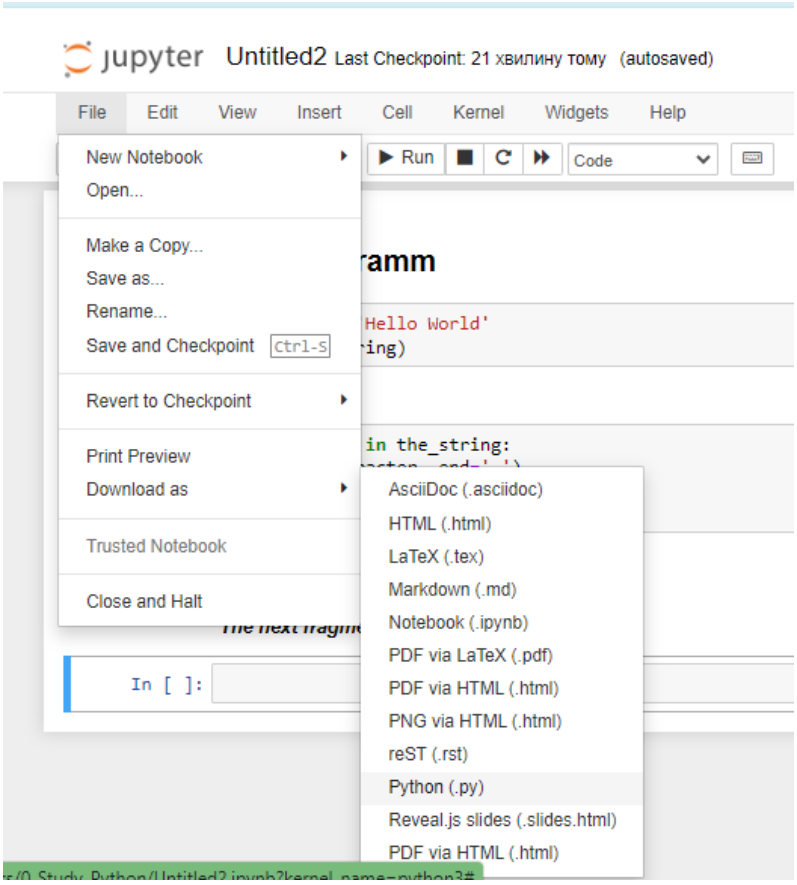


Рисунок 1.8 – Конвертування файлів Jupyter Notebook ***.ipynb** в інші формати

Для цього знаходимо відповідний файл на диску, наприклад за таким шляхом:

C:\Users\Admin\AppData\Local\Programs\Python\Python38\Scripts.

Знаходимо файл

jupyter_notebook.exe

та створюємо для нього ярлик на робочому столі.

В результаті такого налаштування середовища з'являється зручна можливість запускати **jupyter-notebook** з робочого столу і з вказаної робочої папки, де будуть розміщені проєкти та програми на Python.

1.3 Основні властивості та бібліотеки мови програмування Python

Мова програмування Python була створена **Гвідо ван Россумом** в кінці 1980-х – на початку 1990-х років з метою створення високорівневої мови, яка б комбінувала читабельність коду з можливістю використання для повсякденних завдань програмування. Перша версія Python (0.9.0) була

опублікована у лютому 1991 року. Ван Россум хотів, щоб Python був «другою найкращою мовою» для всього, поступаючись лише спеціалізованим мовам у конкретних галузях.

Популярність і значне поширення Python обумовлені його властивостями, серед яких можна виділити такі:

Читабельність і простота – Python має простий і читабельний синтаксис, що сприяє легкому вивченню та використанню мови. Він використовує відступи для визначення блоків коду замість фігурних дужок або ключових слів, як у багатьох інших мовах.

Мультипарадигмальність – Python підтримує різні парадигми програмування, включаючи об'єктно-орієнтоване, процедурне та деякою мірою функціональне програмування.

Велика стандартна бібліотека – Python поставляється з великою стандартною бібліотекою, яка включає інструменти для різноманітних завдань – від роботи з вебом до наукових обчислень.

Інтерпретованість – Python є інтерпретованою мовою, що означає, що код виконується безпосередньо, без попередньої компіляції, що сприяє швидкій розробці та тестуванню.

Динамічна типізація – у Python типи даних визначаються автоматично в момент виконання програми, а не в момент компіляції. Це робить мову гнучкою, але вимагає від розробників бути уважними до управління типами.

Розширюваність – Python дозволяє інтегрувати інші мови програмування, що може бути корисним для виконання оптимізованих частин коду, наприклад на C або C++.

Python є розширюваним – це означає, що код, написаний на C або C++, можна інтегрувати в Python. Є різні варіанти Python: Java+Python – **Jython, Cpython, Ironpython**.

Разом з тим можна відзначити і деякі відмінності від інших мов програмування. Порівнюючи з Java або C#, Python часто вважається більш продуктивним завдяки своїй простоті та читабельності, що дозволяє розробникам з меншими зусиллями досягати аналогічних результатів.

Python, порівняно із JavaScript, широко використовується для серверного програмування та скриптів, але це не обмежується лише веб-розробкою.

На відміну від C або C++, Python пропонує вищий рівень абстракції, що може призводити до *меншої продуктивності* в деяких обчислювально-інтенсивних задачах, але значно *спрощує процес розробки*.

Порівняно з R, який також широко використовується в аналітиці даних і статистиці, Python вважається більш універсальним і легшим для інтеграції з веб-додатками та іншими системами.

Python встановлюється за замовчуванням на всі Linux-сервери і використовується в автоматизації роботи системного адміністратора.

Вбудовані системи. Дуже часто Python використовується для програмування вбудованих систем. Найвідоміший проєкт, який використовує Python, – це Raspberry Pi. Саме цією мовою програмування і буде продемонстровано низку лабораторних робіт для закріплення його практичного застосування.

Бібліотеки Python є однією з ключових причин широкої популярності цієї мови програмування. Вони надають готові до використання модулі та функції, які значно прискорюють розробку програмного забезпечення, зменшуючи кількість коду, який потрібно написати вручну. Деякі з найбільш важливих і широко використовуваних бібліотек Python наведені на рисунку 1.9.

Рисунок 1.9 – Найпоширеніші бібліотеки Python

Дамо коротку характеристику деяких поширених бібліотек та їх застосування.

Для наукових обчислень та аналізу даних:

NumPy – бібліотека для ефективної роботи з великими масивами та матрицями чисел. Є основою для багатьох інших бібліотек аналізу даних.

Pandas – надає структури даних та інструменти для аналізу і маніпулювання табличними даними. Широко використовується в обробці та аналізі даних.

SciPy – колекція математичних алгоритмів і функцій, побудована на основі NumPy. Використовується для наукових та інженерних обчислень.

Для машинного навчання та штучного інтелекту:

Scikit-learn – надає простий доступ до великої кількості алгоритмів машинного навчання для класифікації, регресії, кластеризації та інших завдань.

TensorFlow та **PyTorch** – популярні бібліотеки для глибокого навчання, які надають потужні інструменти для створення та тренування нейронних мереж.

Для розробки веб-додатків:

Django та **Flask** – два популярні веб-фреймворки для розробки веб-додатків. Django надає багато готових рішень «з коробки», тоді як Flask є більш легковісним і гнучким.

Для роботи з базами даних:

SQLAlchemy – надає могутній і гнучкий інструментарій для роботи з реляційними базами даних за допомогою об'єктно-реляційного відображення (ORM).

SQLite3 – вбудована в Python бібліотека для роботи з базами даних SQLite, яка дозволяє здійснювати операції з базами даних без необхідності встановлення додаткових серверів або систем.

Для розробки графічного інтерфейсу користувача (GUI):

Tkinter – стандартний інтерфейс Python до Tk GUI тулкіту. Використовується для створення простих графічних інтерфейсів.

PyQt або **PyGTK** – більш потужні інструменти для створення складних графічних інтерфейсів користувача.

Візуалізація даних:

Matplotlib – основна бібліотека для створення статичних, анімованих та інтерактивних візуалізацій у Python.

Seaborn – будується на Matplotlib і надає більш високорівневий інтерфейс для створення привабливих статистичних графіків.

Ці та багато інших бібліотек роблять Python надзвичайно універсальною мовою, яка знаходить застосування в широкому спектрі галузей: від веб-розробки, аналізу даних та машинного навчання до наукових досліджень, автоматизації та багато іншого.

При написанні коду мовою Python, окрім синтаксису, потрібно дотримуватися певних правил, які уніфікують сам код і дозволяють іншим розробникам писати програми приблизно в одному стилі. Такі правила розміщено в інструкціях «**PEP 8 – Style Guide for Python Code»,** з якими можна ознайомитися на сайті https://peps.python.org/pep-0008 /.

PEP 8 є офіційним керівництвом зі стилю написання коду на Python. Він містить набір рекомендацій щодо того, як форматувати код Python, щоб зробити його більш читабельним і зрозумілим. PEP 8 був написаний Гвідо ван Россумом, Баррі Варсоу та Ніком Когланом і вперше опублікований у 2001 році. Цей документ є важливим для розробників Python, оскільки він сприяє забезпеченню єдності та консистентності стилю кодування в Python-спільноті.

Ось деякі ключові аспекти, які охоплює PEP 8:

- відступи – використання чотирьох пробілів на рівень відступу замість табуляції або змішування табуляцій та пробілів;

- продовження ліній має бути вирівняно з візуальним відступом або на один рівень відступу вправо, якщо це допомагає підвищити читабельність;
- рекомендована максимальна довжина рядка – 79 символів для коду і 72 для коментарів та docstrings;
- імпорти мають бути на початку файлу, після будь-яких коментарів до модуля та docstrings, і вони мають бути згруповані в такому порядку:
    - стандартна бібліотека імпортів;
    - імпорти сторонніх бібліотек;
    - локальні імпорти програми/бібліотеки.

Кожна група імпортів повинна бути розділена порожньою лінією;

- пробіли у виразах та інструкціях.

Уникайте зайвих пробілів у таких ситуаціях:

- відразу всередині круглих, квадратних або фігурних дужок;
- між комою та наступним словом;
- перед відкриваючою дужкою, за якою йде список аргументів функції або індексування/зрізу.

Це далеко не повний перелік правил, з якими кожний розробник мовою Python повинен ознайомитися і дотримуватися в подальшій роботі, щоб код був стандартизованим і читабельним.

1.4 Розробка програмного забезпечення та основи синтаксису Python

Процес розробки програмного забезпечення може варіюватися залежно від обраної методології (наприклад водоспад, гнучка розробка, DevOps тощо), але існує кілька основних етапів, які є спільними для більшості підходів (рисунок 1.10).

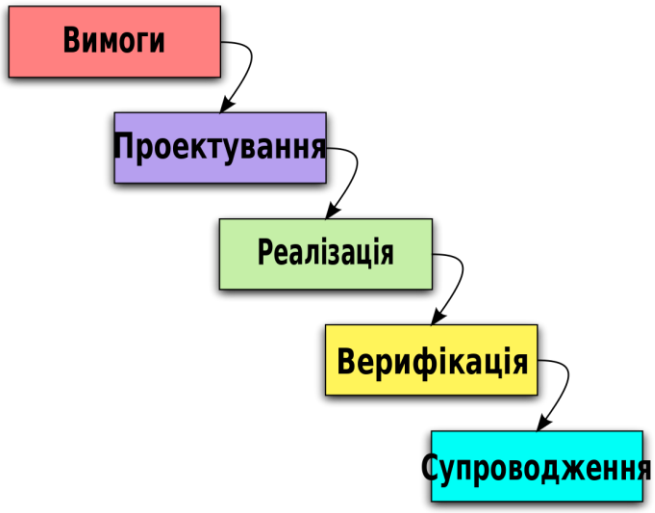


Рисунок 1.10 – Водоспадна (каскадна) модель життєвого циклу програмного забезпечення

Джерело: https://uk.wikipedia.org/wiki/Водоспадна_модель

Наведемо загальний огляд основних етапів процесу розробки програмного забезпечення.

Визначення вимог – на цьому етапі замовник та розробники визначають цілі проєкту та його вимоги. Це включає збір вимог, аналіз можливостей системи, визначення обмежень та встановлення очікувань.

Планування – планування включає розробку плану проєкту, розподіл ресурсів, визначення таймлайну та бюджету. На цьому етапі також визначаються методологія розробки, інструменти й технології, що будуть використані.

Проєктування архітектури – етап проєктування передбачає створення високорівневої архітектури програми та детальне проєктування компонентів системи. Це включає створення діаграм класів, баз даних, інтерфейсів користувача та іншої документації проєкту.

Розробка – на цьому етапі команда програмістів реалізує визначену архітектуру, пише код згідно з проєктними специфікаціями. Розробка зазвичай ведеться ітераційно, з постійними перевірками якості коду та його відповідності вимогам.

Тестування – на цьому етапі виконується ретельне тестування програми для виявлення та усунення помилок. Тестування може включати *модульне тестування, інтеграційне тестування, системне тестування, тестування продуктивності та приймальне тестування*.

Розгортання – після завершення розробки та тестування програма готова до розгортання. Це може включати встановлення програми на серверах клієнта, налаштування середовища та міграцію даних.

Технічна підтримка та обслуговування – після розгортання програма потребує постійної технічної підтримки для вирішення проблем користувачів, виправлення помилок та оновлення функціоналу згідно з новими вимогами і технологічними змінами.

Етапи процесу реалізації програмного коду можна також представити у такому у вигляді:

Етап 1 – вводимо текст розробленої програми, яку називають *початковим кодом*, у комп'ютер і зберігаємо в пам'яті. Для цього середовище програмування/розробки має **редактор тексту**, який забезпечує введення й редагування початкового коду.

Етап 2 – після введення програми та виправлення помилок, які могли статися під час введення, здійснюється перетворення програми з мови програмування високого рівня у двійковий код. Таке перетворення здійснюється за допомогою **транслятора програм**. Розрізняють два типи трансляторів: ***компілятори*** та ***інтерпретатори***.

У процесі ***інтерпретації*** з початкового програмного коду послідовно кожна команда (інструкція) перетворюється у двійковий код і відразу виконується – на екрані висвітлюється результат її виконання. Після завершення виконання однієї команди виконується наступна і так далі до останньої команди.

У процесі ***компіляції*** здійснюється перетворення всього тексту програмного коду у двійковий код. Отриману після компіляції програму називають ***об'єктним модулем.*** Така програма ще не готова до виконання. Початковий код зазвичай містить посилання на інші модулі (підпрограми), які містяться **в бібліотеці підпрограм.** Таким чином, до програмного модуля потрібно додати коди необхідних підпрограм, щоб підготувати програму для виконання.

Етап 3 – після компіляції **редактор зв'язків** «склеює» окремі двійкові модулі в єдину програму, яка називається програмою, що виконується, і яка вже призначена для виконання (рисунок 1.11).

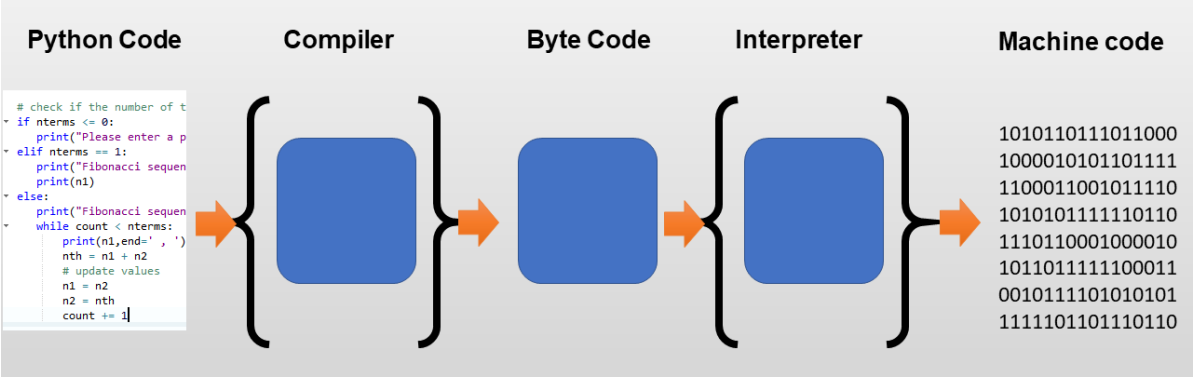


Рисунок 1.11 – Процес отримання машинного коду програми для виконання

Етап 4 – ***тестування.*** Це досить відповідальний етап, над яким у великих IT-компаніях працюють десятки і навіть сотні програмістів різних напрямів. Одним із видів діяльності є тестування готового коду, яке може бути як ручним, так і автоматизованим і призначене для виявлення помилок, так званих «багів», невідповідностей тощо.

Традиційно першою програмою при вивченні будь-якої мови програмування є програма «**Hello, World**». Ця програма виводить на екран (або в консоль) привітання «Hello, World!». Щоб написати її мовою Python, скористаємося середовищем розробки **Jupyter Notebook.** Для цього відриємо **Jupyter Notebook** у спосіб, який був описаний раніше, і в меню «File» натиснемо на «New Notebook» і оберемо Python3, після чого утвориться робоче поле для введення інструкцій (рисунок 1.12).

Для виведення на екран «**Hello, World**» достатньо всього однієї інструкції, представленої на рисунку 1.13. Наступні інструкції можуть вводитися в цьому самому робочому рядку або створюються нові за допомогою піктограми «+» (див. рисунок 1.7). Для виконання цієї інструкції натискають «**Run**» у головному меню.

Синтаксис мови програмування – це набір правил, що описує комбінації символів алфавіту, які вважаються правильно структурованою програмою або її фрагментом. Отже, синтаксис визначає, як буде виглядати програма цією мовою, зокрема, як пишуться оператори, оголошення та інші мовні конструкції.

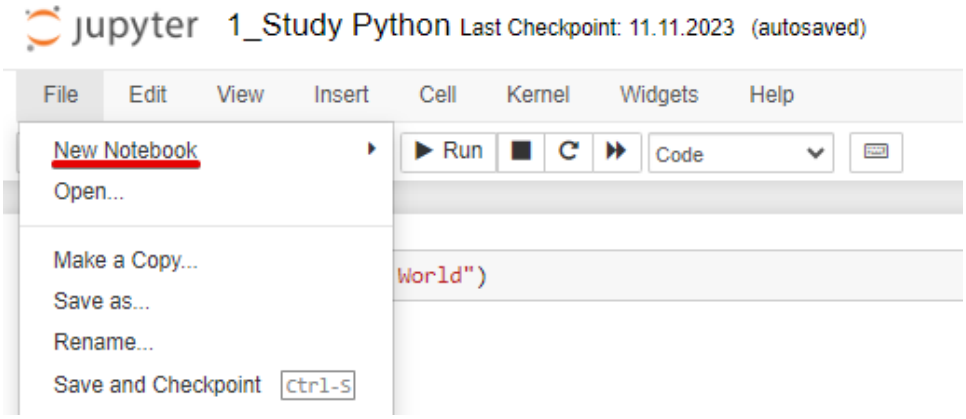


Рисунок 1.12 – Створення файлу в середовищі Jupyter Notebook

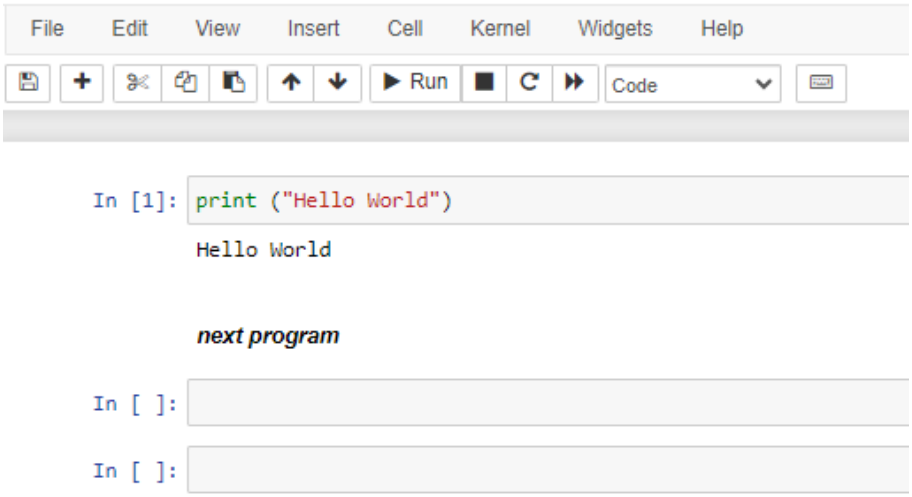


Рисунок 1.13 – Демонстрація виконання інструкції «Hello, World»

Основи синтаксису мови програмування Python можна відобразити таким чином:

Кінець рядка є кінцем інструкції (крапка з комою не потрібні):

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

```
7
```

Допускається записувати кілька інструкцій в одному рядку. При цьому їх розділяють крапкою з комою

```
a = 5; b = 2; print ('a =', a, ';', 'b =', b)
```

```
a = 5 ; b = 2
```

.

Допускається записувати одну інструкцію у кількох рядках. Для цього її необхідно обмежити круглими, квадратними або фігурними дужками:

```
y = (x**5 + x**4 +
     x**3 + x**2 + x + 1)
```

.

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

```
x = float(input("Введіть значення x: "))
if x == 10:
    print('Yes')
print('No')
```

```
Введіть значення x: 7
No
```

Вкладені інструкції в Python записуються відповідно до того самого шаблону, коли основна інструкція завершується двокрапкою, після чого розташовується вкладений блок коду, зазвичай з відступом під рядком основної інструкції.

Розглянемо на прикладах основи роботи з Python. Для цього введемо поняття ідентифікатора.

Ідентифікатор – це послідовність символів, що може використовуватися як ім’я елемента (об’єкта) у мові програмування. У Python у ролі ідентифікаторів може використовуватися послідовність символів, що складається з літер латинського алфавіту (при цьому розрізняються великі та маленькі літери), цифр та знака нижнього підкреслення «_», за умови, що першим символом не є цифра:

```
y
Y
y123
variable_111
```

Базові типи даних (вказуються неявно) можуть бути отримані через команду

```
a = 5
print(type(a))
```

```
<class 'int'>
```

У Python типи даних визначають класифікацію або категоризацію даних, які програма може обробляти. Тип даних визначає вид операцій, які можуть бути виконані над даними, а також формат зберігання. Основні типи даних у Python включають:

1. *Числові типи:*
   - **int** (цілі числа), наприклад 5, -3, 42;
   - **float** (дійсні числа), наприклад 3.14, -0.001, 2.0;
   - **complex** (комплексні числа), наприклад 3+4j.
2. *Булевий тип (Boolean):*
   - **bool** представляє логічні значення True або False.
3. *Послідовності:*
   - **str** (рядки), наприклад 'hello', "Python";
   - **list** (списки), наприклад [1, 2, 3], ['a', 'b', 'c'];
   - **tuple** (кортежі), наприклад (1, 2, 3), ('a', 'b', 'c').
4. *Множини:*
   - **set** представляє невпорядковану колекцію унікальних елементів, наприклад {1, 2, 3};
   - **frozenset** є незмінною версією set.
5. *Словники:*
   - **dict** представляє пари ключ-значення, наприклад {'name': 'John', 'age': 30}.

У Python **константи** – це змінні, значення яких не призначені для зміни протягом виконання програми. Python не має вбудованої підтримки для декларації констант на мовному рівні (на відміну від деяких інших мов програмування), але за домовленістю, константи зазвичай позначаються великими літерами з підкресленнями між словами:

```
PI = 3.14159
GRAVITY = 9.81
MAX_CONNECTIONS = 100
```

Змінна – іменована область пам’яті, ім’я якої (ідентифікатор) використовується для здійснення доступу до даних (значення змінної), що містяться у цій області пам’яті. Значення змінної є літералом. Тип змінної визначається типом літералу, що є значенням цієї змінної:

```
x = 23      # x – змінна зі значенням 23
y = x + 3   # y – змінна зі значенням 26
```

При роботі зі змінними у Python потрібно пам’ятати, що змінна є посиланням на область пам’яті, де зберігаються її дані. Наприклад, якщо

```
a = 3
s = a
```

,

то це фактично означає, що використовуються не дві різні змінні **a** та **s**, а одна змінна зі значенням 3, що має два різні імені **a** та **s.**

В Python є особливий спосіб обміну змінних значеннями:

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

.

Такий спосіб використовується дуже часто. Цей метод працює завжди, навіть якщо змінні різних типів (у цьому випадку вони обмінюються не тільки значеннями, але й типами). Круглі дужки у цьому записі можна опустити:

Ключові слова мови програмування – це зарезервовані ідентифікатори, що наділені певним сенсом. Їх можна використовувати виключно відповідно до значення, яке закріплене за ними у мові.

При оголошенні нового ідентифікатора як імені змінної, функції або іншого об'єкта потрібно переконатися, що цей ідентифікатор не є ключовим словом мови програмування. Інакше програма або не виконається, або виконається з помилками.

Сучасні інтегровані середовища програмування, як правило, ключові слова **виділяють іншим кольором**. Тому при написанні програми, наімовірніше, будемо розуміти, що цей ідентифікатор не можна використовувати як змінну. Наприклад, у коді нижче ідентифікатори **while** і **if** є ключовими словами:

```
while i <= N - 1:
    if N % i == 0:
        prime = False
        i = i + 1
```

Коментарі – зрозуміла для програміста анотація, що знаходиться безпосередньо у вихідному коді комп'ютерної програми. Коментарі пишуть для того, щоб зробити код зрозумілішим. Вони ігноруються при компіляції й інтерпретації. Коментарі також обробляють різними способами для створення окремої документації на базі вихідного коду за допомогою генераторів документації. Коментарі поділяють на однорядкові та багаторядкові. Однорядковий коментар може розташовуватися лише в одному фізичному рядку програми, починається **символом** # і закінчується кінцем рядка. Багаторядковий коментар може розташовуватися у кількох фізичних рядках програми. Він виділяється з двох боків потрійними апострофами або потрійними подвійними лапками:

```
# a - вхідне число
a = int(input("введіть число:"))
if a%2 == 0:
    print("Число є парним")

'''
Багарядковий коментар - перший рядок
Багарядковий коментар - другий рядок
xxx

'''
```

Присвоєння – механізм у програмуванні, що дозволяє динамічно змінювати зв'язки об'єктів даних (наприклад змінних) із їхніми значеннями. Після інструкції присвоєння попереднє значення змінної **a** стає недосяжним:

```
a = 7
b = 8.0
a = 10
```

Інструкції введення та виведення використовуються для взаємодії користувача та програми, а також для обміну інформацією між зовнішніми носіями та оперативною пам'яттю комп'ютера (об'єктами програми). Існують різноманітні способи введення і виведення інформації з використанням клавіатури, екрану, файлів, звукової карти комп'ютера, принтера та інших периферійних пристроїв. Проте наразі під введенням будемо розуміти введення користувачем певної інформації з клавіатури під час виконання програми, а під виведенням – друкування даних у консоль під час виконання програми. Інструкція введення має такий синтаксис:

```
x = input(S)
```

,

де **x** – змінна, в яку записується результат введення з клавіатури, **S** – рядок підказки. Наприклад, введення інформації та її виведення може мати такий вигляд:

```
name = input("Як тебе звати? ")
print ("Привіт, мене звати: ", name)
```

```
Як тебе звати? Петро
Привіт, мене звати:  Петро
```

f-рядки (форматовані рядкові літерали) в Python – це зручний спосіб вбудовувати вирази всередину рядкових літералів для форматування. Вони були введені в Python 3.6 і дозволяють включати вирази Python безпосередньо всередині рядків, використовуючи фігурні дужки {}. Результат виразу у фігурних дужках буде виведений у вигляді рядка. Це робить код більш читабельним і ефективним при форматуванні тексту. Ось приклад використання f-рядка для виведення повідомлення з числовим значенням:

```
name = "Петро"
age = 30
print(f"Ім'я: {name}, Вік: {age}")
```

```
Ім'я: Петро, Вік: 30
```

У цьому прикладі **name** та **age** є змінними, що містять рядок і число відповідно. f-рядок f"Ім'я: {name}, Вік: {age}" включає ці змінні безпосередньо всередині рядка. Фігурні дужки **{}** використовуються для зазначення місця в рядку, де має бути вставлено значення змінних. Python автоматично перетворить числове значення змінної age на рядок при його вставці в f-рядок.

Лінійна програма – це програма, що складається лише з інструкцій введення, виведення, присвоєння та тотожної команди.

Числа у Python нічим не відрізняються від звичайних чисел та підтримують математичні операції, відомі вам з математики. Числові типи у Python представлені трьома типами: цілими, дійсними та комплексними. Розглянемо детальніше кожен із них.

Цілі числа. Змінні та літерали цього типу набувають значень з (обмеженої) множини цілих чисел. Цілий тип у Python, при необхідності, позначають int. Арифметичні операції над цілими числами представлені в таблиці 1.1.

Таблиця 1.1 – Арифметичні операції над цілими числами

Операція над числами	Опис операції
x + y	сума x та y
x - y	різниця x та y
x * y	добуток x та y
x / y	частка від ділення x на y
x // y	ділення націло x на y
x % y	остача від ділення x на y
-x	унарний мінус
x ** y	піднесення x до степеня y

Приклади основних арифметичних операцій:

a = 5 # int
b = 7.0 # float
c = 1 + 2 # 3
d = 5 - 3 # 2
e = **a** * **b** # 35.0
f = 3.0 / 2 # 1.5
g = 3 / 2 # 1
h = 5 % 3 # 2
j = 7**5.5 # 44467.1422

Для застосування спеціальних математичних функцій необхідно підключити бібліотеку **math.** Наступний приклад ілюструє обрахунок математичного виразу і виведення його на екран у заданому форматі кількості знаків після коми:

```
from math import *
a = 1; b = 2
x = sqrt(a*b)/(exp(a)*b)+a*exp((2*a)/b)
print(f"x = {x:.2f}")
```

```
x = 2.98
```

Основні математичні функції бібліотеки **math** наведено в таблиці 1.2.

Виведення даних за форматом

Для виведення числа з конкретною кількістю знаків після коми через **f**-рядки в Python ви можете вказати формат безпосередньо всередині фігурних дужок:

```
number = 123.4567
print(f"{number:.2f}")
```

```
123.46
```

Ще одним варіантом форматованого виведення є **format**, але він вважається застарілим, порівнюючи з **f** форматом. Метод **format** у Python використовується для форматування рядків. Він дозволяє вставляти значення змінних в рядок на місце плейсхолдерів, що представлені фігурними дужками {}. Це забезпечує створення рядків, які містять дані зі змінних або виразів.

```
name = "Дмитро"
age = 20
message = "Привіт, моє ім'я - {}, мені {} років."
print(message.format(name, age))
```

```
Привіт, моє ім'я - Дмитро, мені 20 років.
```

Метод **format** також підтримує більш складне форматування, таке як зазначення формату чисел, вирівнювання тексту, використання ключових словників для підстановки значень та багато іншого. Наприклад

```
price = 19.82
quantity = 3
message = "Ціна одного товару: {price:.2f} грн, кількість: {quantity},\
загальна сума: {total:.2f} грн."
print(message.format(price=price, quantity=quantity, total=price*quantity))
```

```
Ціна одного товару: 19.82 грн, кількість: 3,загальна сума: 59.46 грн.
```

Таблиця 1.2 – Функції в бібліотеці **math**

Назва функції	Призначення функції
math.ceil(x)	Повертає округлене x як найближче ціле значення типу float, яке дорівнює або перевищує x (округлення «вгору»).
math.copysign(x, y)	Повертає число x зі знаком числа y. На платформі, яка підтримує знак нуля, copysign (1.0, -0.0) дасть -1.0.
math.fabs(x)	Повертає абсолютне значення (модуль) числа x. В Python є вбудована функція abs, але вона повертає модуль числа з тим же типом, що число, fabs же завжди повертає значення типу float.

Продовження таблиці 1.2

Назва функції	Призначення функції
math.factorial(x)	Повертає факторіал цілого числа x, якщо x не є цілим, виникає помилка ValueError.
math.floor(x)	На противагу ceil(x), повертає округлене x як найближче ціле значення типу float, менше або рівне x (округлення «вниз»).
math.fmod(x, y)	Аналогічна функції fmod(x, y) бібліотеки C. Зазначимо, що це не те саме, що вираз Python x%y. Бажано використовувати при роботі з об’єктами float, у той час як x% y більше підходить для int.
math.frexp(x)	Являє число в експоненційному записі x = m*2e і повертає мантису m (дійсне число, модуль якого лежить в інтервалі від 0.5 до 1) і порядок e (ціле число) як пару чисел (m, e). Якщо x = 0, то повертає (0.0, 0).
math.fsum(iterable)	Повертає float суму від числових елементів об’єкта, що ітерується.
math.isinf(x)	Перевіряє, чи є float об'єкт x плюс або мінус нескінченністю, результат відповідно True або False.
math.isnan(x)	Перевіряє, чи є float об’єкт x об’єктом NaN (not anumber).
math.ldexp(x, i)	Повертає значення x*2i, тобто здійснює дію, зворотнуфункції math.frexp(x).
math.modf(x)	Повертає частину, що йде після коми, і цілу частину від float числа. Обидва результати зберігають знак початкового числа x і представлені типом float.
math.trunc(x)	Повертає цілу частину числа x у вигляді int об’єкта.
math.exp(x)	Повертає exp.
math.log(x [, base])	При передачі функції одного аргументу x повертає натуральний логарифм x. При передачі двох аргументів другий береться як основа логарифма.
math.log1p(x)	Повертає натуральний логарифм від x + 1.
math.log10(x)	Повертає десятковий логарифм x.
math.pow (x, y)	Повертає x в степені y.
math.sqrt(x)	Квадратний корінь (square root) з x.
Тригонометричні функції	
math.acos(x)	Повертає арккосинус x, в радіанах.

Закінчення таблиці 1.2

Назва функції	Призначення функції
math.asin(x)	Повертає арксинус x, в радіанах.
math.atan(x)	Повертає арктангенс x, в радіанах.
math.atan2(y, x)	Повертає atan(y / x), в радіанах. Результат лежить в інтервалі [-π, π]. Вектор, кінець якого задається точкою (x, y), утворює кут з додатним напрямком осі x. Тому ця функція має більш загальне призначення, ніж попередня. Наприклад й atan(1), й atan2(1, 1) дадуть в результаті pi/4, але atan2(-1, -1) це вже 3*pi/4.
math.cos(x)	Повертає косинус x, де x виражений в радіанах.
math.hyp(x, y)	Повертає евклідову норму, тобто sqrt(x**2+y**2). Зручно для обчислення гіпотенузи (hyp) і довжини вектора.
math.sin(x)	Повертає синус x, де x виражений в радіанах.
math.tan(x)	Повертає тангенс x, де x виражений в радіанах.
Радіани в градуси і навпаки	
math.degrees(x)	Конвертує значення кута x з радіан в градуси.
math.radians(x)	Конвертує значення кута x з градусів в радіани.

Приклад 1.1. Знайти суму цифр двозначного числа.

Розв'язок. Нехай x – задане двозначне число, наприклад 25. Сума цифр цього числа є сумою першої та другої цифр цього числа: 2 + 5 = 7. Першу цифру цього числа можна знайти як ділення числа x націло на 10. Друга ж цифра цього числа є остачею від ділення числа x на 10. Тоді програма буде мати вигляд

```
x = int(input("Введіть двозначне число "))
first = x // 10 # first - перша цифра числа
second = x % 10 # second - друга цифра числа
suma = first + second
print("Сума цифр числа %d = %d" % (x, suma))
```

```
Введіть двозначне число 25
Сума цифр числа 25 = 7
```

Дійсні числа. Дійсні числа у Python записуються у вигляді десяткового дробу, для розділення цілої і дробової частини в якому застосовується символ «крапка». Дійсний тип у Python, позначають **float.**

Приклад 1.2. Обчислити значення багаточлена

$$y = x^6 - 4x^4 + 3x - 7$$

для заданого числа x.

Розв'язок. Програма буде мати вигляд

```
x = float(input('x=? '))
y = x ** 6 - 4 * x ** 4 + 3 * x - 7
print('y=', y)
```

```
x=? 10
y= 960023.0
```

Комплексні числа. Окрім дій над цілими та дійсними числами, у мові програмування Python передбачені операції з комплексними числами:

```
x = complex(1,3)
print(x)
```

```
(1+3j)
```

Послідовність виконання арифметичних дій. Порядок виконання дій із числами має наступний вигляд. Найбільший пріоритет має піднесення до степеня, далі множення, ділення, цілочисельне ділення, остача від ділення. Найменший пріоритет мають дії додавання та віднімання. Якщо у рядку, який описує арифметичний вираз, стоять дії з однаковим пріоритетом, то операції виконуються зліва направо послідовно. Пріоритетність дій завжди можна змінити за допомогою дужок.

Логічні операції та операції порівняння. Для порівняння значень змінних використовуються вже знайомі з математики символи:

> – знак «більше»,

< – знак «менше»,

>= – більше або дорівнює,

<= – менше або дорівнює.

Результатом виконання таких операцій буде два значення булевого типу (True або False):

```
5 > 4
```

```
True
```

```
print(1 > 2)
print(7 == 7)
print(1 != 2)
```

```
False
True
True
```

ПРАКТИЧНА ЧАСТИНА

1.5 Підготовка до виконання завдання

1. Ознайомтеся з теоретичними відомостями щодо встановлення Python, середовища розробки та основними синтаксичними структурами.

2. Якщо на вашому робочому комп'ютері не встановлений Python, встановіть з офіційного сайту https://www.python.org та виконайте необхідні інструкції щодо його налаштування.

3. Встановіть і налаштуйте середовище розробки *Jupyter Notebook* згідно з інструкціями, наведеними в цій роботі. За бажанням можна обрати інше середовище розробки.

1.6 Практичне завдання

Згідно зі своїм варіантом напишіть програму для виконання наступних завдань.

У завданні № 1 обчисліть значення функції **z** або **y** та виведіть його на екран. Параметри правої частини рівняння вводяться з клавіатури при запиті. При необхідності продемонструйте декларування констант. Застосуйте **f**-формат виведення з обмеженою кількістю розрядів після коми.

Варіанти завдання № 1

Варіант	Завдання
1	$z = \cos\alpha + \sin\alpha + \cos 3\alpha + \sin 3\alpha$
2	$z = \frac{1}{4} - \frac{1}{4}\sin(\frac{5}{2}\pi - 8\alpha)$
3	$y = \frac{e^{-x} - x\cdot\sin x - \ln^2 x}{\lg\lvert\cos x\rvert + ctg(x^2 - 1)}$
4	$y = \frac{e^{-x} + tg(x-1)}{\lvert\ln x\rvert + \sqrt{\sin x + \cos 2x}}$
5	$z = \cos\alpha + \cos 2\alpha + \cos 6\alpha + \cos 7\alpha$
6	$y = \frac{e^{-3x} + \ln^3(x-1)}{\ln\lvert x+1\rvert + tg(x^2 - 1)}$
7	$z = \cos^2\alpha + \cos^4\alpha$
8	$z = (\cos\alpha - \cos\beta)^2 - (\sin\alpha - \sin\beta)^2$
9	$z = \sin(\alpha + \beta)\cdot\sin(\alpha - \beta)$
10	$z = \sqrt{\frac{m+3}{m-3}}$
11	$z = \cos^2 x + \sin^2 y$
12	$z = \left(\frac{e^{-x} - 12.34}{\lg x - \cos x^3}\right)^{-0.5}$
13	$y = \frac{\sqrt{x-1} - tg(x+1)}{\arccos x + \ln x} + 2,75$

14	$y=\frac{e^{-3x}+tg(4x-1)}{\lvert\cos x\rvert+\sqrt{\cos 2x}}$
15	$y=\frac{\sqrt{x+1}-\sin(x-\pi)}{\cos(x-3.1)+\ln^2 x}+x\cdot\lg x$

Завдання № 2

Розробіть програму для обчислення площі трикутника за введеними з клавіатури довжинами трьох його сторін *a, b, c* за формулою Герона:

$S=\sqrt{p(p-a)(p-b)(p-c)}$, де $p=(a+b+c)/2$.

На вході подаються цілі числа, на виході – дійсне число, яке потрібно подати у вигляді **f**-формату.

Приклад:

Введіть *a* : 3.
Введіть *b* : 4.
Введіть *c* : 5.
Площа трикутника дорівнює 6.12.

Завдання № 3

Розробіть програму «Щасливий квиток». Припустимо, користувач придбав квиток з шестизначним номером. Будемо вважати квиток «щасливим», якщо сума перших трьох цифр дорівнює сумі останніх трьох цифр.

На вході маємо шестизначне число, на виході – повідомлення у вигляді True/False.

Приклад:

Input	Input
Введіть чотиризначний номер:	Введіть чотиризначний номер:
12345678	341530
Output	Output
Щасливий квиток 12345678? *False*	Щасливий квиток 341530? *True*

Завдання № 4

Сашко кожного дня лягає спати рівно опівночі і нещодавно дізнався, що оптимальний час для його сну становить X хвилин. Сашко хоче встановити собі будильник так, щоб він дав сигнал через X хвилин після півночі, але для цього необхідно вказати час сигналу в форматі години, хвилини. Необхідно визначити, на який час встановити будильник. Значення X вводиться з клавіатури у хвилинах.

Приклад:

Input	Input
480	512
Output	Output
8 годин 0 хвилин	8 годин 32 хвилини

Завдання № 5

Напишіть програму, яка обчислює вік користувача на основі введеної дати народження.

Вхідні дані: Рядок *birthd* у форматі «день-місяць-рік».

Вихідні дані: Число *age* – повних років користувача.

1.7 Зміст протоколу роботи

Протокол до виконаної роботи оформлюється у вигляді файлу і містить титульний аркуш із назвою навчального закладу, факультету та випускової кафедри, дисципліни та лабораторної роботи, яка виконується, прізвище та ім'я студента і викладача. Наступна сторінка містить назву роботи та її мету, короткі теоретичні відомості про основні оператори і команди, які використовуються для розв'язання поставлених у роботі задач.

Звіт має містити постановку задачі і програмний код її розв'язання з наданням необхідних коментарів. Наприкінці роботи наводяться висновки щодо отриманих результатів по розв'язанню поставлених задач.

Робота захищається індивідуально.

1.8 Контрольні питання для самоперевірки

1. Які особливості та переваги мови Python ви знаєте?
2. Що таке PEP8, назвіть основні принципи програмування за цими правилами. Які основні правила PEP8 стосуються імен змінних, функцій, відступів тощо?
3. Назвіть основні принципи синтаксису мови Python (змінні, пріоритети, типи даних тощо).
4. Як здійснюється введення/виведення даних у мові Python?
5. Для чого використовується метод format() у Python і як він працює для форматування рядків? Що таке f-формат виводу (виведення)? Продемонструйте вивід (виведення) для різних типів даних.
6. Назвіть основні типи даних в Python.
7. Як можна обміняти значення двох змінних в Python?
8. Що означає None?
9. В якій бібліотеці містяться стандартні математичні функції?
10. Які основні математичні операції підтримуються в Python безпосередньо і як виконати операцію піднесення до степеня?
11. Назвіть парадигми програмування, які підтримує Python

1.9 Задачі до практичної роботи № 1

1. Напишіть програму, яка конвертує температуру з градусів Цельсія в градуси Фаренгейта і навпаки. Формули перетворення є такими:

$$°C = (°F - 32) \times 5 / 9$$

$$°F = (°C \times 9 / 5) + 32$$

Значення температур вводяться з клавіатури.

2. Напишіть програму, що перетворює швидкість з кілометрів за годину (км/год) в метри за секунду (м/с) і навпаки. Значення швидкості вводиться з клавіатури.

3. Напишіть програму, що обчислює відстань між двома містами в кілометрах, використовуючи координати цих міст (широту та довготу). Для обрахунків використайте *формулу гаверсинуса*.

Вхідні дані: Числа *lat1, lon1* – широта та довгота першого міста, *lat2, lon2* – широта та довгота другого міста.

Вихідні дані: Число *distance* – відстань між містами в кілометрах.

4. Напишіть програму, яка приймає на вхід верхню межу діапазону (позитивне ціле число) та виводить всі прості числа у цьому діапазоні. Просте число – це натуральне число, більше за 1, яке має тільки два дільники: 1 і саме число

Приклад введення:

```
Введіть верхню межу діапазону: 20
```

Приклад виведення:

```
Прості числа у діапазоні від 1 до 20: 2, 3, 5, 7, 11, 13, 17, 19
```

5. Користувач купує яблука в магазині. Якщо відомо, що в одній упаковці може міститися рівно **6** яблук, напишіть програму, яка приймає кількість яблук, яку купив користувач, і виводить кількість повних упаковок, які можна заповнити цими яблуками, та кількість яблук, які залишаться поза упаковками.

1.10 Завдання до самостійної роботи

1. PEP 8 – конвенція для написання коду на Python.
2. Різновиди середовищ розробки та їх встановлення. Особливості застосування.
3. Поняття виключення на Python. Призначення, їх види та реалізація. Як обробляти виключення за допомогою конструкції try-except? Як використовувати блок finally в обробці виключень?
4. Як створити власний клас виключення у Python, і в яких ситуаціях це може бути особливо корисно для розробки додатків?
5. Імпорти бібліотек на Python. Імпорт конкретної функції або класу з модуля у Python для оптимізації використання пам’яті та часу завантаження програми.
6. Найпоширеніші бібліотеки та їх призначення.
7. Інтерпретовані та компільовані мови програмування. Особливості їх використання.
8. Що таке віртуальне оточення Python і навіщо воно потрібне?
9. Моделі життєвого циклу програмного забезпечення: водоспадна (каскадна) модель; спіральна модель, ітераційна модель, гнучка модель (Agile).
10. Особливості та порівняння реалізації мови Python: Cpython, Jython.

2 УМОВНІ ТА ЦИКЛІЧНІ ОПЕРАТОРИ МОВИ PYTHON

Мета практичної роботи № 2: ознайомитися з умовними та циклічними операторами мови Python, їх синтаксичними конструкціями та програмною реалізацією.

ТЕОРЕТИЧНІ ПОЛОЖЕННЯ

2.1 Поняття про алгебру висловлювань

Розгалуження та умовні оператори є однією з основних контрольних структур у програмуванні, поряд з циклами та послідовним виконанням команд. Вони дозволяють програмі динамічно реагувати на різні умови та вводи, виконуючи різні блоки коду залежно від того, які умови виявляються істинними. Це дає програмам значну гнучкість і можливість обробки різноманітних ситуацій без необхідності зміни коду.

Реалізація розгалужень безпосередньо пов'язана з таким поняттям, як *алгебра висловлювань*. Алгебра висловлювань, яка також відома як булева алгебра або логіка висловлювань, є фундаментальною областю математики та логіки, що займається аналізом і маніпуляцією висловлюваннями, які можуть бути істинними або хибними. Вона лежить в основі багатьох областей комп'ютерних наук, особливо в областях, що стосуються логічного програмування, проєктування цифрових схем і, як ми обговорювали, реалізації розгалужень у програмуванні.

Основні поняття алгебри висловлювань можна охарактеризувати наступним чином.

Висловлювання – в алгебрі висловлювань висловлюванням є будь-яке твердження, яке може бути однозначно ідентифіковане як істинне (**true**) або хибне (**false**). Наприклад, висловлювання «сьогодні дощить» може бути істинним або хибним залежно від погоди.

Змінні – у булевій алгебрі змінні використовуються для представлення висловлювань. Кожна змінна може приймати значення істинності: 0 (false) або 1 (true).

Операції – алгебра висловлювань оперує з декількома базовими логічними операторами, які дозволяють конструювати складніші висловлювання з простіших. До основних логічних операторів відносять такі:

кон'юнкція (I, AND), позначається символом ∧ або **&**. Результат істинний, тільки якщо обидва висловлювання істинні;

диз'юнкція (АБО, OR), позначається символом ∨ або |. Результат істинний, якщо хоча б одне з висловлювань істинне;

інверсія (НЕ, NOT), позначається символом ¬ або **!**. Змінює значення висловлювання на протилежне;

імплікація (ЯКЩО... ТО), позначається символом →. Результат хибний тільки тоді, коли перше висловлювання істинне, а друге хибне;

еквівалентність (ТОДІ І ТІЛЬКИ ТОДІ), позначається символом ↔. Результат істинний, коли обидва висловлювання мають однакове значення істинності.

Булева алгебра має набір законів, які використовуються для спрощення логічних виразів та виведення висновків. До основних законів належать закони ідемпотентності, домінування, подвійного заперечення, комутативності, асоціативності, дистрибутивності, поглинання та де Моргана.

Алгебра висловлювань є критично важливою для розуміння і проєктування логічних схем, аналізу програмного коду, особливо коли йдеться про умовні конструкції та розгалуження. Вона дозволяє формалізувати та систематизувати логічне мислення, що є ключовим у програмуванні та комп'ютерних науках.

Розглянемо операції відношення для числових типів даних. Нехай вирази **a** і **b** належать до одного з числових типів – цілого, дійсного або комплексного. Тоді:

- **==** (дорівнює) – бінарна операція, що діє за правилом:

висловлювання **a == b** істинне тоді і тільки тоді, коли значення виразів **a** і **b** є однаковими;

- **!=** (не дорівнює) – бінарна операція, що діє за правилом:

висловлювання **a != b** істинне тоді і тільки тоді, коли значення виразів **a** і **b** є різними.

Наприклад, відношення нижче буде завжди набувати значення False:

```
5==7
```

```
False
```

Нехай тепер вирази **a** і **b** належать до цілого або дійсного типу (тобто впорядкованого). Тоді:

- < (менше) – бінарна операція, що діє за правилом:

висловлювання **a < b** істинне тоді і тільки тоді, коли значення виразу **a** є меншим за значення виразу **b**;

- <= (менше або дорівнює) – бінарна операція, що діє за правилом: висловлювання **a <= b** істинне тоді і тільки тоді, коли значення виразу **a** є меншим або дорівнює виразу **b**;
- (більше) – бінарна операція, що діє за правилом:

висловлювання **a > b** істинне тоді і тільки тоді, коли значення виразу **a** є більшим за значення виразу **b**;

- >= (більше або дорівнює) – бінарна операція, що діє за правилом: висловлювання **a >= b** істинне тоді і тільки тоді, коли значення виразу **a** є більшим або дорівнює виразу **b**.

Приклади відношень:

```
5 < 7
```

```
True
```

```
a = 5
b = 7
a <= b
```

```
True
```

Пріоритет операторів в Python визначає порядок, в якому виконуються операції у виразах, що містять кілька операторів. Наприклад, множення та ділення мають вищий пріоритет, ніж додавання та віднімання. Наведемо простий приклад, який демонструє різний результат обчислення при врахуванні пріоритету операцій:

```
result = 10 + 2 * 3
print(result)
```

```
16
```

```
result = (10 + 2) * 3
print(result)
```

```
36
```

Приклад 2.1. Обчислити математичний вираз:

```
result = (10 + 2) * 3 ** 2 - 5 / (2 + 3)
print(result)
```

```
107.0
```

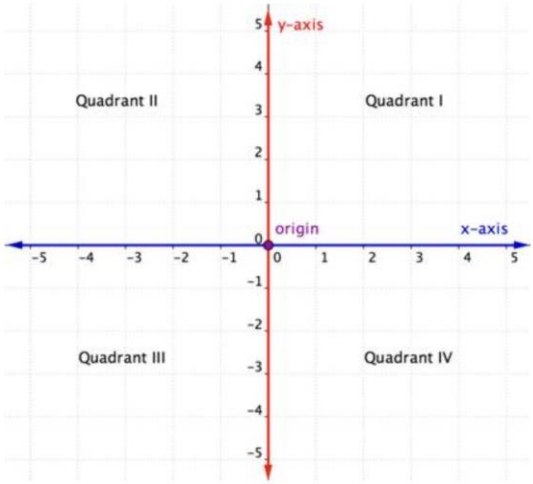
Розв'язок. Виконання операцій відбувається таким чином:
Спочатку виконуються операції в дужках: 10 + 2 дає 12, і 2 + 3 дає 5.
Далі виконується піднесення до степеня: 3 ** 2 дає 9.
Потім виконується множення: (10 + 2) * 3 ** 2 дає 12 * 9 або 108.
Далі виконується ділення: 5 / (2 + 3) дає 1.
Нарешті, виконується віднімання: 108 - 5 / (2 + 3) або 108 - 1, що дає 107.

Пріоритет логічних операцій та операцій відношення від найвищого до найнижчого відображено у таблиці 2.1.

Таблиця 2.1 – Пріоритет виконання операцій

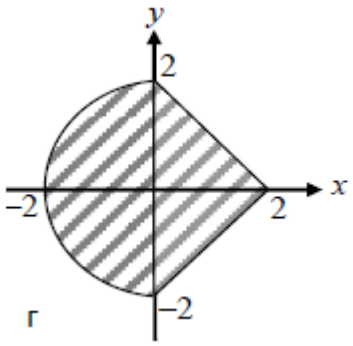
`**`
`*, /, //, %`
`+, -`
`==, !=, >, <, >=, <=`
`not`
`and`
`or`

Приклад 2.2. Умова знаходження точки з координатами (x, y) в першому координатному квадранті буде виглядати таким чином:




```
x >= 0 and y >= 0
```

Приклад 2.3. Записати умову належності точки площини з координатами (x, y) до зафарбованої області.



Розв'язок. Визначимо заштриховану область через операції з множинами. Заштрихована область буде об'єднанням заштрихованих підобластей з лівої ($x \leq 0$) та правої ($x > 0$) півплощини. Отже,

$$(\{x \leq 0\} \cap \{x2 + y2 \leq 4\}) \cup (\{x \geq 0\} \cap \{|x| + |y| \leq 2\}).$$

Замінивши операції над множинами логічними операціями, отримуємо умову належності точки до заштрихованої області:

```
# Перевірка першої області: {x ≤ 0} ∩ {x^2 + y^2 ≤ 4}
    condition1 = (x <= 0) and ((x**2 + y**2) <= 4)

    # Перевірка другої області: {x ≥ 0} ∩ {|x| + |y| ≤ 2}
    condition2 = (x >= 0) and (abs(x) + abs(y) <= 2)

    # Об'єднання умов
    result = condition1 or condition2
```

2.2 Розгалуження та умовні оператори

Розгалуження – це структура у мовах програмування, яка дозволяє виконати певну інструкцію або набір інструкцій, коли задовольняється певна умова, або вибрати для виконання одну з декількох можливих інструкцій або груп інструкцій на основі результату перевірки цієї умови.

Розгалуження дозволяють втілити умовну логіку в програмах, що є фундаментальним принципом програмування. Це означає, що програма може виконувати різні дії залежно від даних або стану додатка.

Використання розгалужень допомагає структурувати код, робить його більш читабельним і легшим для розуміння. Логічне розділення коду на блоки сприяє кращій організації програми.

Розгалуження можуть бути використані для перевірки даних на валідність або для обробки виняткових ситуацій, тим самим забезпечуючи стабільність та надійність програми.

Розгалуження роблять програму адаптивною, оскільки вона може виконувати різні дії залежно від змін у вхідних даних або зовнішньому середовищі.

Особливості розгалужень у програмуванні:

логічні умови – розгалуження базуються на логічних умовах, які можуть включати порівняння, логічні оператори (AND, OR, NOT) та інші умови, що повертають булеві значення (True або False);

вкладеність – розгалуження можуть бути вкладеними, тобто одне розгалуження може міститися всередині іншого. Це дозволяє реалізувати складніші умовні конструкції;

ефективність – на виконання програми розгалуження мають незначний вплив, але необхідно уважно ставитися до складності умов та кількості вкладених розгалужень, щоб уникнути зниження продуктивності.

Конструкція if-else у мові програмування Python дозволяє виконати розгалуження логіки програми на основі перевірки деякої умови. Це одна з найбільш базових форм контролю потоку в програмуванні, яка дозволяє програмі вибирати між двома шляхами виконання залежно від того, чи є зазначена умова істинною, чи хибною:

```
if condition:
    state1
else:
    state2
```

Ця конструкція працює таким чином:

if condition – ця частина вказує умову, яка буде перевірена. Якщо умова виявляється істинною (тобто вона оцінюється як **True**), то виконується блок коду, який іде безпосередньо після цього виразу (вказаний як **state1** у прикладі);

state1 – блок коду, який буде виконано, якщо умова, вказана після **if**, є істинною. Цей блок коду має бути відділений відповідним відступом (чотири пробіли) від **if** для зазначення належності до блоку **if**;

else – ця частина коду є необов'язковою і вказує на альтернативний блок коду, який буде виконано, якщо умова в **if** не є істинною (тобто оцінюється як **False**);

state2 – це блок коду, який буде виконано, якщо умова в **if** є хибною. Так само, як і блок для **state1**, він має бути відділений відповідним відступом (чотири пробіли).

Приклад 2.4. Навести приклад використання конструкції **if-else** для запуску тієї частини коду, яка буде відповідати умові:

```
temperature = int(input("temperature today:"))

if temperature > 30:
    print("It's a hot day.")
else:
    print("It's not a hot day.")
```

```
temperature today:25
It's not a hot day.
```

В умовному операторі блок **else** є необов'язковим і конструкція **if condition:** застосовується в Python для виконання умовного розгалуження коду. Це означає, що якщо задана умова (condition) виявляється істинною (**True**), то виконується певний блок коду (**state**):

```
x = 10
if x > 5:
    print("x більше ніж 5")
```

```
x більше ніж 5
```

```
x = 5
if x > 5:
    print("x більше ніж 5")
```

Якщо ж умова не виконується (тобто є хибною або False), цей блок коду пропускається.

Каскадне розгалуження в програмуванні – це метод структурування коду, який дозволяє виконати серію умовних перевірок одна за одною. В Python це зазвичай реалізується за допомогою послідовності конструкцій **if, elif** (яке є скороченням від "**else if**"), і необов'язково **else** на кінці. Така структура дозволяє програмі вибрати один із багатьох можливих шляхів виконання на основі декількох умов.

Синтаксис каскадного розгалуження виглядає таким чином:

```
if condition1:
    # Виконується, якщо condition1 істинна
    state1
elif condition2:
    # Виконується, якщо condition1 хибна, але condition2 істинна
    state2
elif condition3:
    # Виконується, якщо і condition1, і condition2 хибні, але condition3 істинна
    state3
...
else:
    # Виконується, якщо всі вищезазначені умови хибні
    stateN
```

Приклад 2.5. Для введеної з клавіатури точки x знайти значення функції

$$f(x) = \begin{cases} -x^2 + 1, & x < -1, \\ 0, & |x| \le 1, \\ x^2 - 1, & x > 1. \end{cases}$$

Програмна реалізація на Python набуде вигляду:

```
x = float(input("x = "))
if x < -1:
    f = -x ** 2 + 1
elif abs(x) <= 1:
    f = 0
else:
    f = x ** 2 - 1
print("f(", x, ")= ", f)
```

```
x = 10
f( 10.0 )=  99.0
```

Тернарний умовний оператор в Python – це компактний спосіб виконання умовного вибору між двома значеннями на основі деякої умови. Він є аналогом конструкції **if-else**, але використовується у виразах для прямого визначення значення. Основна форма запису тернарного оператора виглядає таким чином:

```
<вираз1> if <умова> else <вираз2>
```

Тут **<умова>** перевіряється першою. Якщо умова істинна (**True**), результатом виразу буде **<вираз1>.** Якщо умова хибна (**False**), результатом буде <**вираз2**>.

Приклад 2.6. Знайти модуль дійсного числа x, введеного з клавіатури.

Програмна реалізація на Python набуде вигляду:

```
# Введення числа з клавіатури
x = float(input("Введіть дійсне число: "))

# Використання тернарного умовного оператора для знаходження модуля числа
abs_x = x if x >= 0 else -x

print(f"Модуль числа {x} дорівнює {abs_x}")
```

```
Введіть дійсне число: -7
Модуль числа -7.0 дорівнює 7.0
```

У цьому коді спочатку користувачу пропонується ввести дійсне число, яке зчитується з клавіатури та конвертується у тип **float**. Потім за допомогою тернарного умовного оператора визначається модуль цього числа: якщо x є невід'ємним (x >= 0), то модуль дорівнює самому x, інакше модуль дорівнює -x. Результат виводиться на екран.

2.3 Реалізація циклічних структур

Циклічні структури в програмуванні використовуються для повторення певного блоку коду певну кількість разів або доки не буде виконана певна умова. Це дозволяє ефективно обробляти колекції даних, виконувати задачі, що вимагають повторень (наприклад обчислення, що базуються на ітераціях), а також створювати програми, що можуть працювати до виконання певної умови (наприклад сервери, що обробляють запити).

Циклічні структури знаходять застосування у широкому спектрі задач програмування, таких як:

- *обробка колекцій даних* – цикли дозволяють обробляти кожний елемент списку, кортежу або іншої колекції даних, виконуючи над ними певні операції (наприклад фільтрація, трансформація, агрегація);
- *обчислення, що вимагають повторень* – задачі, що потребують ітеративного наближення до результату (наприклад алгоритми пошуку, сортування);

- *інтерактивні програми та сервери* – цикли можуть використовуватися для створення програм, які працюють до отримання команди про завершення від користувача, або серверів, що безперервно обслуговують запити;
- *автоматизація та скриптинг* – цикли широко використовуються в скриптах для автоматизації повторюваних задач, таких як обробка файлів, збір даних з веб-сторінок тощо.

Циклічні структури є фундаментальними інструментами в програмуванні, що дозволяють розробникам писати ефективний, компактний код, що є легко читабельним для різноманітних алгоритмічних задач.

У Python цикли поділяються на два типи:

- цикли з умовою продовження (**while**);
- цикли-ітератори по колекції (**for**).

Цикл з умовою продовження

Цикл з умовою продовження, відомий також як цикл **while**, виконує блок коду доти, доки задана умова залишається істинною. Це означає, що перед кожною ітерацією циклу перевіряється певна умова, і якщо ця умова істинна, то цикл продовжує виконуватися. Як тільки умова стає хибною, виконання циклу припиняється, і програма переходить до наступного рядка коду після циклу. Основна структура такого циклу має вигляд:

```
while умова:
    # Блок коду, який виконуватиметься, поки умова істинна
```

Правило роботи виконання циклу з умовою:

1. Python обчислює значення «умови».
2. Якщо «умова» == True, то інтерпретатор виконує інструкцію, яка розміщена в блоці конструкції while, після чого все починається спочатку з пункту 1.
3. Якщо «умова» == False, то інтерпретатор не виконує інструкцію в блоці конструкції while і завершує виконання циклу.

Приклад 2.7. Застосувати циклічну структуру для виконання повторюваної задачі виведення на екран.

Програмна реалізація на Python набуде вигляду:

```
# Цикл виконуватиметься, поки змінна і менша за 5
i = 0
while i < 5:
    print(i, end = " ")
    i += 1  # Важливо змінювати змінну циклу,
            # щоб уникнути нескінченного циклу
```

```
0 1 2 3 4
```

У цьому прикладі, цикл **while** виводить числа від 0 до 4. Перевірка умови відбувається перед кожною ітерацією, і цикл продовжує виконуватися доти, поки змінна **i** менша за 5. З кожною ітерацією змінна **i** збільшується на 1, що забезпечує зміну умови **i**, і врешті-решт, припинення виконання циклу.

Зверніть увагу, що в Python аргумент **end** використовується в функції print() для зазначення того, що буде виведено після всіх значень, переданих у функцію. За замовчуванням **end='\n'**, що означає, що після виконання функції print() курсор переходить на новий рядок. Якщо ви замініте значення end на пробіл, наприклад **end=" "**, то після виведення тексту курсор залишиться на тому самому рядку, і наступний виклик print() продовжить виведення тексту на цьому ж рядку.

Приклад 2.8. Реалізувати програму для перевірки числа, чи є воно простим.

Для довідки: просте число – це натуральне число, більше за один, яке має лише два дільники: одиницю і саме себе. Інакше кажучи, просте число не має інших дільників, окрім одиниці і самого себе. Наприклад, числа 2, 3, 5, 7, 11 є простими числами, оскільки їх можна розділити лише на одиницю і на саме число.

Програмна реалізація на Python набуде вигляду:

```
n = int(input("Введіть число для перевірки на простоту: "))
is_prime = True  # Припускаємо, що число просте

if n > 1:
    i = 2
    while i*i <= n:
        if n % i == 0:
            is_prime = False
            break
        i += 1
else:
    is_prime = False

if is_prime:
    print(f"Число {n} є простим.")
else:
    print(f"Число {n} не є простим.")
```

```
Введіть число для перевірки на простоту: 113
Число 113 є простим.
```

У цьому прикладі спочатку встановлюємо змінну **is_prime** в **True**, що вказує на те, що число вважається простим до проведення перевірки. Потім ми перевіряємо, чи число не є меншим або дорівнює 1, оскільки числа 0 і 1 не є простими за визначенням.

Цикл **while** перебирає числа від 2 до квадратного кореня з n (для ефективності перевірки, оскільки дільники числа, якщо вони існують, будуть знайдені в діапазоні від 2 до sqrt(n)). Якщо будь-яке число ділить n без залишку, змінна is_prime встановлюється в False, і цикл переривається за допомогою **break**. Якщо цикл завершується без переривання, це означає, що жодний дільник так і не був знайдений, і n залишається простим числом. Наприкінці програма виводить, чи є введене число простим, чи ні.

Приклад 2.9. Знайти суму цифр заданого натурального числа при застосуванні циклу **while**.

Розв'язок. Щоб знайти суму цифр заданого натурального числа за допомогою циклу **while**, можна використовувати такий алгоритм:

- встановимо початкову суму рівною 0;
- використаємо цикл while, який працюватиме, поки число більше 0;
- на кожній ітерації циклу відокремимо останню цифру числа за допомогою операції взяття залишку від ділення на 10;
- додамо останню цифру до суми;
- відкинемо останню цифру з числа за допомогою цілочисельного ділення на 10;
- повторимо кроки 3–5, доки число не стане рівним 0;
- виведемо результат суми цифр числа.

Програмна реалізація на Python набуде вигляду:

```
# Введіть число
n = int(input("Введіть натуральне число: "))

# Ініціалізація суми
sum_of_digits = 0

# Цикл для визначення суми цифр
while n > 0:
    digit = n % 10  # Отримання останньої цифри
    sum_of_digits += digit  # Додавання цифри до суми
    n = n // 10  # Відкидання останньої цифри

# Виведення результату
print("Сума цифр заданого числа:", sum_of_digits)
```

```
Введіть натуральне число: 1234
Сума цифр заданого числа: 10
```

Цикли-ітератори по колекції

Цикл по колекції у Python зазвичай реалізується за допомогою циклу **for**. Цей тип циклу дозволяє ітерувати по елементах будь-якої колекції (наприклад списку, кортежу, множини, словника) або будь-якого іншого ітерованого об'єкта. Під час кожної ітерації циклу **for** можна отримати доступ до поточного елемента колекції та виконати з ним певні дії.

В Python *колекції – це структури даних*, які дозволяють зберігати набір елементів у впорядкованому або неупорядкованому вигляді. До колекцій в Python належать списки, кортежі, словники, множини та інші.

Звичайні колекції в Python включають:

Список (List): впорядкована колекція елементів, яка дозволяє дублювання. Створюється за допомогою квадратних дужок [].

Наприклад: my_list = [1, 2, 3, 4, 5].

Кортеж (Tuple)*:* впорядкована колекція елементів, яка не може бути змінена (імутабельна). Створюється за допомогою круглих дужок ().

Наприклад: my_tuple = (1, 2, 3, 4, 5).

Словник (Dictionary): невпорядкована колекція пар ключ-значення. Створюється за допомогою фігурних дужок {}.

Наприклад: my_dict = {'apple': 5, 'banana': 3, 'orange': 2}.

Множина (Set): невпорядкована колекція унікальних елементів без дублювання. Створюється за допомогою фігурних дужок {} або функції set().

Наприклад: my_set = {1, 2, 3, 4, 5}.

Рядок (String): невпорядкована колекція символів. Створюється за допомогою лапок " або "".

Наприклад: my_string = "Hello, world!"

Ці колекції можуть містити будь-які типи даних: числа, рядки, списки, кортежі, інші словники, множини тощо.

Синтаксична структура циклу для ітерування по колекції має вигляд:

```
for iterator in collection:
    process_iteration
```

,

де **for** – ключове слово, яке починає цикл в Python;

iterator – змінна, яка використовується для зберігання кожного елемента колекції під час ітерації. Назву змінної iterator можна вибрати довільно, але вона повинна бути зрозумілою для тих, хто читає код;

in – ключове слово, що використовується для вказівки на те, що ітерація відбувається по елементах праворуч від нього;

collection – колекція або інший ітерований об’єкт (наприклад список, кортеж, словник, множина, рядок), по елементах якого відбувається ітерація;

process_iteration – блок коду всередині циклу, який виконується для кожного елемента колекції. Тут може розміщуватися будь-яка логіка обробки: виведення на екран, обчислення, додавання у нову колекцію тощо.

Приклад 2.10. Вивести на екран елементи колекції при застосуванні циклу **for.**

Програмна реалізація на Python набуде вигляду:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

```
apple
banana
cherry
```

Приклад 2.11. Знайти суму цифр заданого натурального числа при застосуванні циклу **for**.

Програмна реалізація на Python набуде вигляду:

```
# Вхідні дані: задане число
number = input("Введіть число: ")  # Зауважте, тут number є рядком

# Ініціалізація змінної для суми цифр
sum_of_digits = 0

# Використання циклу for для обчислення суми цифр
for digit in number:
    sum_of_digits += int(digit)  # Конвертація кожного символу
                                 # назад у число і додавання до суми

print("Сума цифр:", sum_of_digits)
```

```
Введіть число: 1234
Сума цифр: 10
```

Цей метод передбачає конвертацію числа у рядок, щоб потім ітерувати по кожному символу (цифрі) у цьому рядку.

При порівнянні цієї програмної реалізації з прикладом 2.7 можна побачити, що вона буде простішою.

У мові програмування Python часто використовується оператор **range,** який є вбудованою функцією і призначений для генерації послідовності чисел. Він часто використовується в циклах **for** для ітерації по послідовності числових значень. Функція **range** може приймати від одного до трьох аргументів, які визначають початок, кінець і крок послідовності.

Синтаксис функції **range**:

range(stop): створює послідовність чисел від 0 до stop – 1;

range(start, stop): створює послідовність чисел від start до stop – 1;

range(start, stop, step): створює послідовність чисел від start до stop – 1, змінюючись на step за кожну ітерацію.

Приклади використання range:

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

```
0
1
2
3
4
```

```
for i in range(1, 5):
    print(i)
```

```
1
2
3
4
```

```
for i in range(2, 11, 2):
    print(i)
```

```
2
4
6
8
10
```

Приклад 2.12. Обчислити суму чисел від 1 до n.

Реалізацію цього прикладу проілюструємо двома способами.

```
n = int(input("Введіть число n: "))
total_sum = 0

for i in range(1, n + 1):
    total_sum += i
print(f"Сума чисел від 1 до {n} = {total_sum}")
```

```
Введіть число n: 5
Сума чисел від 1 до 5 = 15
```

Ініціалізується змінна **total_sum** з початковим значенням 0.

Цикл **for** проходить через всі числа від 1 до n (включно), додаючи кожне число до **total_sum**.

Після завершення циклу **total_sum** міститиме суму всіх чисел від 1 до n, і це значення виводиться на екран.

Наступний спосіб реалізації вбудованої функції **sum** для цієї задачі дає такий самий результат.

```
n = int(input("Введіть число n: "))
total_sum = sum(range(1, n + 1))
print(f"Сума чисел від 1 до {n} = {total_sum}")
```

```
Введіть число n: 5
Сума чисел від 1 до 5 = 15
```

У цьому прикладі **range(1, n + 1)** генерує послідовність чисел від 1 до n (включно), оскільки верхня межа в **range** виключається, тому ми використовуємо n + 1. Функція **sum** обчислює суму всіх чисел у цій послідовності.

Цей метод ефективний і зручний для обчислення суми послідовності чисел без необхідності використання явних циклів.

Приклад 2.13. Обчислити факторіал числа n (n!).

Програмна реалізація на Python набуде вигляду:

```
n = int(input("Введіть невід'ємне ціле число: "))
factorial = 1

if n < 0:
    print("Факторіал визначений лише для невід'ємних цілих чисел.")
else:
    for i in range(1, n + 1):
        factorial *= i
    print(f"Факторіал числа {n} = {factorial}")
```

```
Введіть невід'ємне ціле число: 6
Факторіал числа 6 = 720
```

В цьому коді спочатку користувачу пропонується ввести число n. Перед розрахунком перевіряється, чи n невід'ємне, оскільки факторіал визначений тільки для невід'ємних цілих чисел.

Якщо число n відповідає умовам, ініціалізується змінна factorial значенням 1, а потім за допомогою циклу for та range обчислюється добуток усіх чисел від 1 до n. На завершення, виводиться результат обчислення факторіала.

Приклад 2.14. Обчислити подвійний факторіал числа n (n!!).

Подвійний факторіал числа n, позначається як n!!, – це добуток усіх чисел від 1 до n, що мають ту саму парність, що й n. Інакше кажучи, для парного числа n подвійний факторіал включатиме всі парні числа від 2 до n, а для непарного – всі непарні числа від 1 до n.

Формально подвійний факторіал визначається так:

якщо n парне,

$$n!!=n\times(n-2)\times(n-4)\times\ldots\times 2.$$

якщо n непарне,

$$n!!=n\times(n-2)\times(n-4)\times\ldots\times 1,$$

за умови: 0!!=1.

Подвійний факторіал часто зустрічається в комбінаториці та деяких задачах математичного аналізу.

Програмна реалізація на Python набуде вигляду:

```
n = int(input("Введіть число для обчислення подвійного факторіала: "))
result = 1

if n == 0 or n == -1:
    result = 1
else:
    while n > 0:
        result *= n
        n -= 2

print(f"Подвійний факторіал числа = {result}")
```

```
Введіть число для обчислення подвійного факторіала: 6
Подвійний факторіал числа = 48
```

*# Для непарного числа 5!! = 5 * 3 * 1 = 15*
*# Для парного числа 6!! = 6 * 4 * 2 = 48*

2.4 Переривання та продовження циклів

В Python, як і в багатьох інших мовах програмування, існують спеціальні оператори для управління поведінкою циклів. Це оператори **break** та **continue**, які дозволяють, відповідно, негайно припинити виконання циклу або пропустити поточну ітерацію циклу та перейти до наступної. Ці оператори можуть бути використані в циклах for та while.

Оператор break призводить до негайного припинення виконання циклу. Контроль над програмою передається інструкції, яка йде за тілом циклу:

```
# Приклад використання оператора break
for i in range(1, 10):
    if i == 5:
        break
    print(i, end = " ")
# Виведе числа від 1 до 4
```

```
1 2 3 4
```

У цьому прикладі цикл припиняє своє виконання, коли змінна і досягає значення 5.

Оператор continue призводить до пропуску поточної ітерації циклу та переходу до наступної ітерації. Це означає, що код, який йде після continue у поточній ітерації, ігнорується, але цикл продовжує своє виконання з наступної ітерації:

```
# Приклад використання оператора continue
for i in range(1, 10):
    if i % 2 == 0:
        continue
    print(i, end = " ")
# Виведе всі непарні числа від 1 до 9
```

```
1 3 5 7 9
```

Приклад 2.15. Знайти перше просте число в послідовності Фібоначчі, що перевищує задане користувачем число N. Використати цикли та оператори переривання.

Опис задачі: послідовність Фібоначчі – це ряд чисел, в якому кожне наступне число є сумою двох попередніх:

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377…]

Починається з 0 та 1. Просте число – це число, яке ділиться без остачі тільки на 1 та саме себе.

Алгоритм розв'язку:

- згенерувати числа Фібоначчі;
- перевіряти, чи є поточне число Фібоначчі простим;
- якщо число просте і більше за N, завершити пошук.

Програмна реалізація на Python набуде вигляду:

```python
N = int(input("Введіть число N: "))

a, b = 0, 1  # Початкові значення для послідовності Фібоначчі

while True:
    a, b = b, a + b  # Генерація чисел Фібоначчі
    if b <= N:
        continue  # Шукаємо число, що більше за N

    # Перевіряємо, чи є b простим числом
    prime = True  # Припускаємо, що число просте
    for i in range(2, int(b**0.5) + 1):
        if b % i == 0:
            prime = False
            break  # Якщо число не просте, припиняємо перевірку

    if prime:
        print(f"Перше просте число в послідовності Фібоначчі, що більше за {N}, є {b}.")
        break  # Завершуємо цикл, знайшовши відповідне число
```

```
Введіть число N: 15
Перше просте число в послідовності Фібоначчі, що більше за 15, є 89.
```

Приклад 2.16. Обчислити значення виразу для n-го члена послідовності:

$$S_n = \sum_{i=0}^{n} \frac{x^i}{i!}, \ , \ n \geq 0.$$

Алгоритм розв'язку:

1. Ініціалізувати суму (sum) зі значенням 0.
2. Для кожного i від 0 до n включно:
   - обчислити x^i;
   - обчислити $i!$ (факторіал i);
   - додати до sum результат ділення x^i на $i!$.
3. Вивести обчислену суму.

Програмна реалізація на Python набуде вигляду:

```python
# Вводимо значення x та n від користувача
x = float(input("Введіть значення x: "))
n = int(input("Введіть кількість членів ряду n: "))

# Ініціалізуємо змінні для суми, поточного члена ряду та факторіалу
sum = 1.0  # оскільки перший член ряду (i=0) завжди дорівнює 1
current_term = 1.0  # поточний член ряду, також ініціалізуємо як 1 для i=0
factorial = 1  # факторіал для i=0 дорівнює 1

# Обчислюємо суму ряду
for i in range(1, n + 1):
    factorial *= i  # обчислюємо факторіал i
    current_term *= x / i  # обчислюємо поточний член ряду використовуючи попередній член
    sum += current_term  # додаємо поточний член до суми

# Виводимо результат
print("Сума ряду дорівнює:", sum)
```

```
Введіть значення x: 1
Введіть кількість членів ряду n: 2
Сума ряду дорівнює: 2.5
```

ПРАКТИЧНА ЧАСТИНА

2.5 Підготовка до виконання завдання

1. Ознайомтеся з теоретичними відомостями щодо умовних та циклічних операторів мовою Python, перериваннями і продовженнями циклів.

2. Опрацюйте приклади, наведені в теоретичній частині.

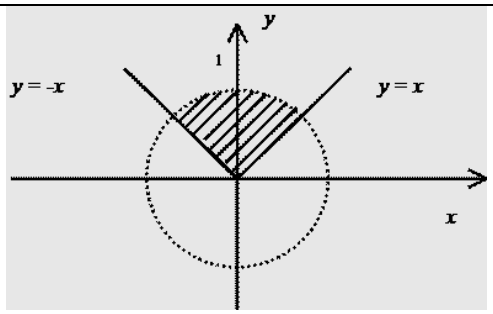
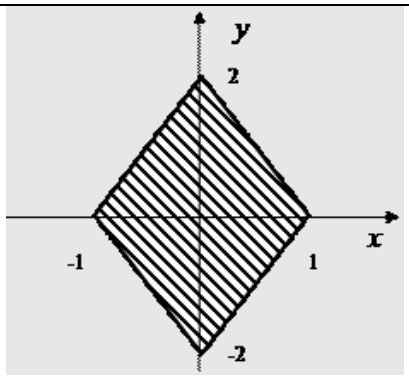
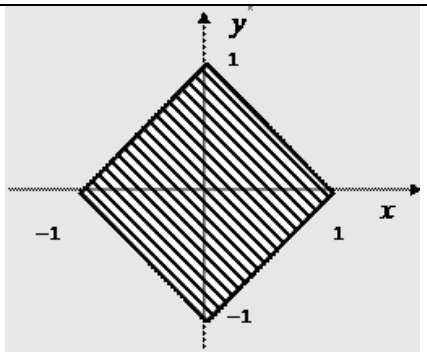
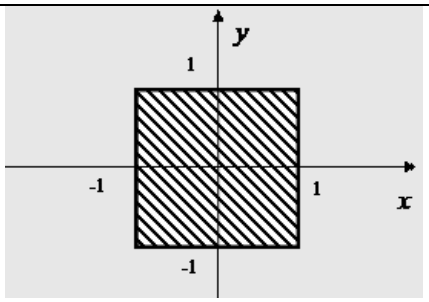
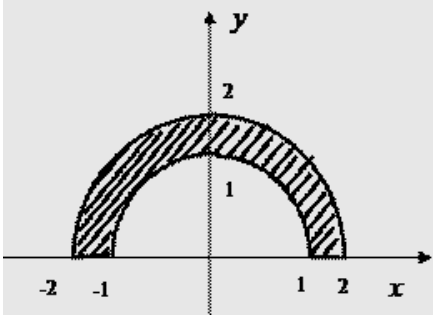
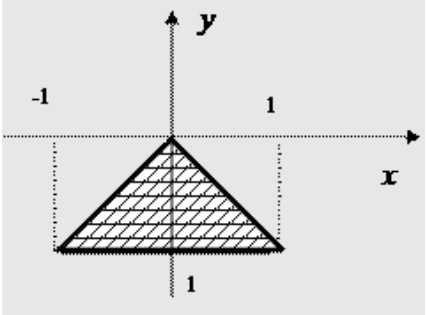
2.6 Практичне завдання

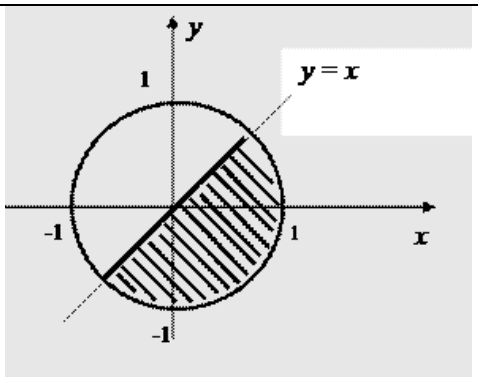
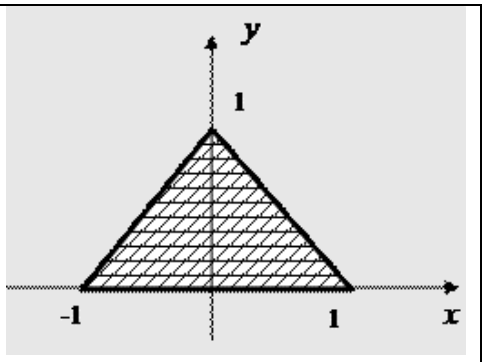
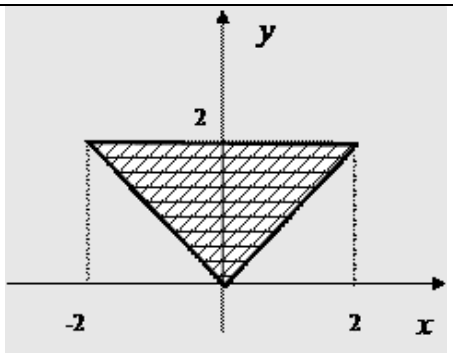
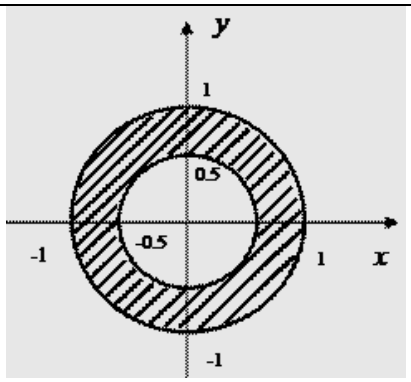
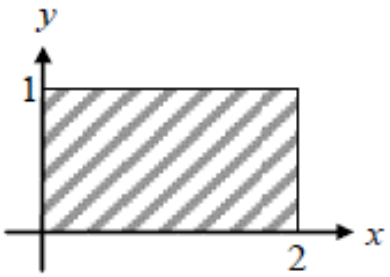
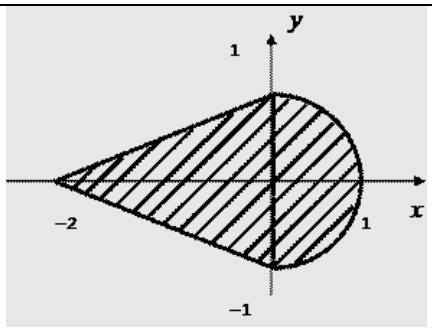
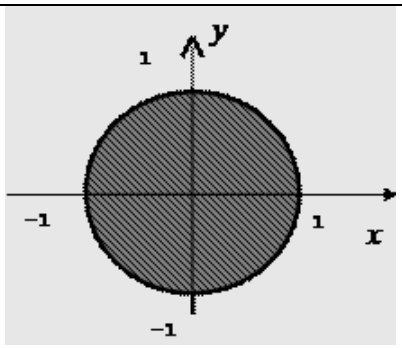
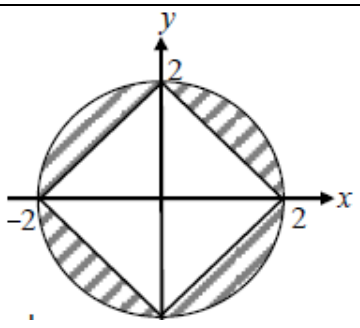
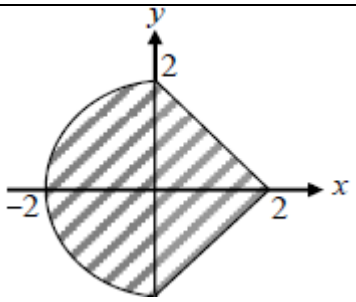
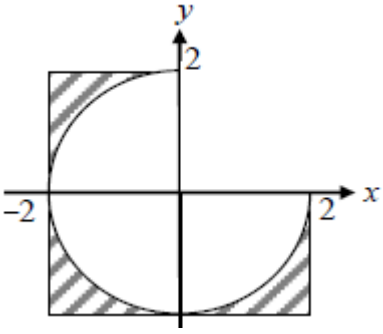
Згідно зі своїм варіантом напишіть програму для виконання наступних завдань і надайте пояснення.

Завдання № 1

Реалізуйте програму мовою Python для перевірки, чи належать точки площини з координатами (x, y), які введені з клавіатури, до зафарбованої області:

Варіанти завдання № 1

№	Завдання	№	Завдання
1		9	
2		10	
3		11	

4		**12**	
5		**13**	
6		**14**	
7		**15**	
8		**16**	

Завдання № 2

Дано чотиризначне число. Перевірте, чи:

а) містить воно цифру 5;
б) складається лише з непарних чисел;
в) сума його цифр більша за число 10;
г) сума його перших двох цифр менша за суму наступних двох цифр.

Завдання № 3

Дано три сторони трикутника a, b, c. Визначте тип трикутника із заданими сторонами a, b, c. Виведіть одну з чотирьох відповідей:

- прямокутний трикутник;
- гострокутний трикутник;
- тупокутний трикутник;
- трикутника з такими сторонами не існує.

Завдання № 4

Шахова тура ходить по горизонталі або вертикалі. Дані дві різні клітинки шахової дошки. Визначте, чи може тура перейти з першої клітинки на другу за один хід.

Вхідні дані. Програма отримає на вхід чотири числа від 1 до 8 кожне, яке задає номер стовпчика і номер рядка спочатку для першої клітинки, потім – для другої.

Вихідні дані. Програма повинна вивести YES, якщо тура може перейти з першої клітинки на другу за один хід, або NO в іншому випадку.

Приклад вхідних і вихідних даних:

Input	Input
4 4 5 5	**4 4 5 4**
Output	Output
NO	**YES**

Завдання № 5

Обчисліть суму ряду. Створіть дві різні програми з використанням різних типів циклів (один типу **for,** другий типу **while**) для x = 0,N, де N – номер варіанта за академічним журналом, наприклад x = 0,5 (N = 5):

$$\sum_{j=1}^{10}\sum_{i=1}^{5}\frac{i^2x^i+1}{j!+i^2}$$

2.7 Зміст протоколу роботи

Протокол до виконаної роботи оформлюється у вигляді файлу і містить титульний аркуш із назвою навчального закладу, факультету та

випускової кафедри, дисципліни і лабораторної роботи, яка виконується, прізвище та ім’я студента і викладача. Наступна сторінка містить назву роботи та її мету, короткі теоретичні відомості про основні оператори і команди, які використовуються для розв’язання поставлених у роботі задач.

Звіт має містити постановку задачі і програмний код її розв’язання з наданням необхідних коментарів. Наприкінці роботи наводяться висновки щодо отриманих результатів по розв’язанню поставлених задач.

Робота захищається індивідуально.

2.8 Контрольні питання для самоперевірки

1. Як в Python використовується конструкція *if* для реалізації розгалуження? Наведіть приклад коду, який перевіряє, чи є число парним.
2. Як використовувати *elif* у Python для створення множинного розгалуження? Напишіть фрагмент коду, де на основі віку визначається категорія: дитина, підліток, дорослий.
3. Що таке вкладені розгалуження в Python і як вони працюють? Наведіть приклад коду з вкладеним розгалуженням, що визначає, чи потрапляє точка в задану область координатної площини.
4. Як в Python використовуються логічні оператори (*and, or, not*) у розгалуженнях? Наведіть приклад коду, де на основі двох умов вирішується, чи виконувати певну дію.
5. Що таке тернарний умовний оператор у Python і як його використовувати? Напишіть приклад коду, де з допомогою тернарного оператора визначається максимальне з двох чисел.
6. Що таке цикл *for* у мові програмування Python, і як він використовується для ітерації по об’єктах?
7. Яка різниця між циклом *for* і циклом *while* у Python, і коли краще використовувати кожен із них?
8. Як ви можете уникнути нескінченного виконання циклу *while* і визначити умову виходу з циклу?
9. Для чого використовується оператор *break* у циклах Python і як він впливає на виконання програми?
10. Як працює оператор *continue* у циклах в Python, і яка його основна відмінність від оператора *break*?
11. Наведіть приклад використання *break* у циклі *while*. В яких ситуаціях це може бути корисно?
12. Наведіть приклад, де оператор *continue* використовується в циклі *for* для пропускання деяких ітерацій. Як це впливає на поведінку циклу?
13. Чи можна використовувати оператори *break* та *continue* поза циклами? Якщо ні, то яка помилка виникне при спробі це зробити?

2.9 Задачі до практичної роботи № 2

1. Є дві коробки: перша розміром A1×B1×C1, друга розміром A2×B2×C2. Визначте, чи можна розмістити одну з цих коробок всередині другої за умови, що розвертати коробки можна тільки на 90 градусів навколо ребер.

Вхідні дані. Програма отримує на вхід числа A1, B1, C1, A2, B2, C2.

Вихідні дані. Програма виводить одну з відповідей:

- коробки однакові;
- першу коробку можна покласти в другу;
- другу коробку можна покласти в першу;
- коробки не розміщуються одна в одну.

Приклад вхідних і вихідних даних:

Input	Input
12 33 21	**34 52 46**
Output	Output
Коробки однакові.	**Коробки не розміщуються одна в одну**.

1. Дано чотири числа, які визначають довжини відрізків a, b, c, d. Визначте, чи можна з цих відрізків утворити прямокутник.

2. Задано тризначне натуральне число. Напишіть програмну реалізацію, яка б переводила цифрове представлення числа в опис у вигляді слів. Наприклад, число 123 виведеться на екран як «сто двадцять три».

3. Обчисліть добуток усіх непарних чисел від 1 до n:

1*3*5*7*….*n.

4. Напишіть програму друку таблиці значень функції $y = \sin(x)$ на відрізку [0; 1] з кроком, що дорівнює 0.1.

5. Задані натуральне число n та дійсні числа $a1, a2, \ldots, an$. Напищіть програми для знаходження:

- $\min(a1, a2, \ldots, an)$;
- $\max(a1, a2, a4, a8, \ldots)$;

Визначте моду, медіану послідовності, чи є аномальні елементи, які виходять за межі правила 3 сігм (правило 3-sigma rule).

2.10 Завдання до самостійної роботи

1. Вкладені цикли в Python: Аналіз використання циклів всередині інших циклів, розгляд типових задач, де це може бути корисним.
2. Ефективне використання спискових включень (*list comprehensions*): Заміна циклів for на більш лаконічні та ефективні спискові включення для створення або трансформації списків.
3. Оптимізація циклів і розгалужень: Розгляд методів оптимізації коду для підвищення ефективності виконання циклів і умовних конструкцій.

4. Генератори та ітератори: Дослідження механізмів Python для створення і використання генераторів та ітераторів, включаючи ефективну роботу з потоками даних.
5. Рекурсія, основні принципи та патерни рекурсивного програмування, включаючи рекурсивні функції, рекурсивне розв’язання задач.
6. Функціональне програмування: Застосування концепцій функціонального програмування в Python, включаючи використання функцій вищого порядку, таких як *map, filter, reduce, і лямбда-функцій*.
7. Модульність та рефакторинг коду. Основи побудови модульного коду, що включає розгалуження та цикли, принципи рефакторингу для покращення читабельності та підтримки коду.
8. Обробка винятків у циклах та розгалуженнях. Розгляд механізмів обробки помилок та винятків для забезпечення стійкості програм при виконанні умовних операцій та циклів.

3 РОБОТА ЗІ СТРУКТУРАМИ ДАНИХ

Мета практичної роботи № 3: ознайомитися з вбудованими структурами даних на прикладі списків, кортежів, словників, множин для ефективної обробки даних.

ТЕОРЕТИЧНІ ПОЛОЖЕННЯ

3.1 Поняття про структури даних у Python

Структури даних у Python – це способи зберігання та організації даних у програмі таким чином, що можна ефективно виконувати операції з цими даними. Python має вбудовані структури даних, такі як **списки, кортежі, словники** та **множини**, кожна з яких підходить для різних типів задач. Крім того, в Python можна створювати складніші структури даних, такі як дерева, графи, черги, стеки тощо, за допомогою класів та модулів. Ось основні структури даних у Python:

- **списки (Lists)** – *упорядковані (ordered)* колекції об'єктів (у структурі даних зберігається порядок додавання елементів), які можуть бути різного типу. Списки мутабельні, тобто їх можна змінювати після створення;
- **кортежі (Tuples)** – схожі на списки, але є незмінними (імутабельними). Це означає, що після створення кортежу його не можна змінити;
- **словники (Dictionaries)** – *неупорядковані (unordered)* колекції пар *ключ-значення*. Словники дозволяють швидко отримувати доступ до значення за ключем та змінювати дані;
- **множини (Sets)** – *неупорядковані* колекції унікальних елементів. Множини корисні для видалення дублікатів з послідовності та виконання математичних операцій над множинами, таких як об'єднання, перетин, різниця;
- **List Comprehensions** – забезпечують компактний спосіб створення списків. Це вирази, які дозволяють створювати нові списки на основі існуючих послідовностей чи ітерабельних об'єктів з додатковою можливістю фільтрації та застосування функцій;
- **генератори (Generators)** – це простий спосіб створювати ітератори. Генератор виробляє елементи «на льоту», що може бути більш ефективним за пам'яттю, ніж створення списку з усіма елементами.

Крім вбудованих структур, Python дозволяє розробникам створювати користувацькі структури даних за допомогою класів, що робить мову дуже гнучкою та потужною для розв'язання широкого спектру задач програмування. Наведемо приклади застосування наведених структур.

3.2 Створення та робота зі списками (Lists)

Список у Python – це упорядкована колекція об'єктів, які можуть бути різних типів. Списки є одними з найбільш універсальних структур

даних у Python, оскільки вони дозволяють зберігати послідовність елементів, які можуть бути змінені. Основні операції зі списками включають додавання, видалення та зміну елементів.

Створення списків. Один зі способів створення списків – за допомогою літерала. Списки створюються шляхом розміщення елементів всередині квадратних дужок **[ ]**, розділяючи їх комами, наприклад:

```
my_list = [1, 2, 3]
print(my_list)  # Виведе: [1, 2, 3]

mixed_list = [1, "Hello", 3.14, [4, 5, 6]]
print(mixed_list)  # Виведе: [1, 'Hello', 3.14, [4, 5, 6]]
```

```
[1, 2, 3]
[1, 'Hello', 3.14, [4, 5, 6]]
```

В Python можна перетворювати інші колекції або ітерабельні об’єкти в списки за допомогою функції **list()**. Це дозволяє легко перетворювати рядки, кортежі, множини та інші ітерабельні об’єкти в списки:

```
my_string = "hello"
my_list_num = list(range(5))

my_list = list(my_string)
print(my_list)
print(my_list_num)
```

```
['h', 'e', 'l', 'l', 'o']
[0, 1, 2, 3, 4]
```

Також списки можна утворювати за допомогою **List Comprehensions:**

```
new_lst = [expr(i) for i in collection if condition]
```

.

Ця конструкція дозволяє створювати нові списки шляхом ефективної обробки інших ітерабельних об’єктів (колекцій) і застосування виразу до кожного елемента цієї колекції.

Наведемо пояснення коду програми:

new_lst – новий список, який буде створено;

[expr(i) for i in collection if condition] – конструкція спискового включення;

expr(i) – вираз, який застосовується до кожного елемента **i** з колекції. Вираз може бути будь-якою функцією або операцією, яка може бути застосована до елементів;

for i in collection – цикл, що проходить через кожний елемент **i** в колекції collection;

if condition – умова, яка визначає, чи повинен елемент **i** з колекції бути включеним у новий список. Тільки елементи, для яких умова видає True, будуть оброблені і додані до **new_lst**.

Приклад 3.1. Припустимо, що необхідно створити список *квадратів* усіх *парних чисел* зі списку вхідних чисел. Використовуючи списокове включення, це можна зробити так:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares_of_even = [x**2 for x in numbers if x % 2 == 0]
print(squares_of_even)
```

```
[4, 16, 36, 64, 100]
```

Додавання елементів. Для додавання елементів до списку використовують метод **append()** для додавання одного елемента або **extend()** для додавання декількох елементів:

```
my_list = [1, 2, 3]
print(my_list)  # Виведе: [1, 2, 3]

my_list.append(4)
print(my_list)  # Виведе: [1, 2, 3, 4]

my_list.extend([5, 6])
print(my_list)  # Виведе: [1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4, 5, 6]
```

Видалення елементів. Елементи можна видаляти за значенням за допомогою методу **remove()** або за індексом за допомогою оператора **del** або методу **pop()**.

Метод **remove()** використовується для видалення першого елемента зі списку, який відповідає заданому значенню. Якщо елемент не знайдено, Python видасть помилку ValueError:

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
print(fruits)  # Виведе: ['apple', 'cherry']
```

```
['apple', 'cherry']
```

Оператор **del** може бути використаний для видалення елемента за індексом або для видалення зрізу списку. Він не тільки видаляє елементи зі списку, але й може видалити саму змінну, роблячи її недоступною для подальшого використання:

```python
fruits = ["apple", "banana", "cherry"]
del fruits[1]
print(fruits)  # Виведе: ['apple', 'cherry']

# Видалення зрізу
fruits = ["apple", "banana", "cherry", "date"]
del fruits[1:3]
print(fruits)  # Виведе: ['apple', 'date']
```

```
['apple', 'cherry']
['apple', 'date']
```

Метод **pop()** видаляє та повертає елемент за вказаним індексом. Якщо індекс не вказаний, **pop()** видаляє та повертає останній елемент списку. Це корисно, коли потрібно працювати з елементом після його видалення:

```python
fruits = ["apple", "banana", "cherry"]
removed_element = fruits.pop(1)
print(fruits)  # Виведе: ['apple', 'cherry']
print(removed_element)  # Виведе: banana

# Видалення останнього елемента
last_element = fruits.pop()
print(fruits)  # Виведе: ['apple']
print(last_element)  # Виведе: cherry
```

```
['apple', 'cherry']
banana
['apple']
cherry
```

Індексація у списках в Python дозволяє отримати доступ до елементів списку за їх позиціями. Індекси в списках починаються з 0 для першого елемента, 1 для другого елемента і так далі. Також Python підтримує від'ємну індексацію, де -1 відповідає останньому елементу, -2 передостанньому і так далі:

```python
my_list = ['Python', 'Java', 'C++', 'JavaScript']

# Отримання першого елемента
print(my_list[0])  # Виведе: Python

# Отримання останнього елемента за допомогою від'ємної індексації
print(my_list[-1])  # Виведе: JavaScript

# Отримання другого елемента (індекс 1, оскільки індексація починається з 0)
print(my_list[1])  # Виведе: Java

# Зміна елемента за індексом
my_list[2] = 'C#'
print(my_list)  # Виведе: ['Python', 'Java', 'C#', 'JavaScript']
```

```
Python
JavaScript
Java
['Python', 'Java', 'C#', 'JavaScript']
```

Індексація також використовується для отримання **зрізів (slices)** списку, дозволяючи отримати підсписок з визначеного діапазону індексів:

```
# Отримання зрізу з другого по третій елемент включно
print(my_list[1:3])  # Виведе: ['Java', 'C#']

# Отримання всіх елементів до третього індексу (не включаючи його)
print(my_list[:3])  # Виведе: ['Python', 'Java', 'C#']

# Отримання всіх елементів, починаючи з другого індексу
print(my_list[2:])  # Виведе: ['C#', 'JavaScript']
```

```
['Java', 'C#']
['Python', 'Java', 'C#']
['C#', 'JavaScript']
```

Зрізи (Slices) – це дуже гнучкий інструмент для роботи з послідовностями в Python, що дозволяє легко отримувати нові послідовності на основі існуючих списків, що проілюстровано в наступному прикладі на основі базового списку

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

, який може бути перетворений на інші списки згідно з наведеними інструкціями при застосуванні зрізів:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(numbers)
# Вибірка елементів з 2-го по 5-й
slice1 = numbers[2:6]
print(slice1)  # Виведе: [2, 3, 4, 5]

# Вибірка елементів з кроком 2, починаючи з початку до кінця списку
slice2 = numbers[::2]
print(slice2)  # Виведе: [0, 2, 4, 6, 8]

# Вибірка елементів з кінця до початку з кроком -1 (реверсування списку)
slice3 = numbers[::-1]
print(slice3)  # Виведе: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

# Вибірка останніх трьох елементів
slice4 = numbers[-3:]
print(slice4)  # Виведе: [7, 8, 9]

# Вибірка всіх елементів до 5-го індексу (не включно)
slice5 = numbers[:5]
print(slice5)  # Виведе: [0, 1, 2, 3, 4]

# Вибірка елементів з 3-го по 7-й з кроком 2
slice6 = numbers[3:8:2]
print(slice6)  # Виведе: [3, 5, 7]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5]
[0, 2, 4, 6, 8]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[7, 8, 9]
[0, 1, 2, 3, 4]
[3, 5, 7]
```

Елемент списку може також брати участь у різних операціях згідно з існуючими правилами, які можна використовувати для об'єктів його типу. Наступний приклад ілюструє знаходження суми чисел у списку різними способами, зокрема і при застосуванні вбудованої функції **sum** з бібліотеки **numpy**:

```
numbers = [1, 2, 3, 4]
total_sum = 0  # Початкове значення суми

for number in numbers:
    total_sum += number
print(total_sum)  # Виводимо суму
```

```
10
```

```
import numpy as np
numbers = [1, 2, 3, 4]

# Використання функції sum
total = np.sum(numbers)

# Виведення результату
print("Сума чисел:", total)
```

```
Сума чисел: 10
```

Конкатенація (від лат. concatenatio – з'єднання) і **зчеплення** списків у Python – це процес об'єднання двох або більше списків в один. Це одна з основних операцій над списками, яка дозволяє ефективно маніпулювати даними.

Конкатенація списків виконується за допомогою оператора «+». Результатом цієї операції є новий список, який містить всі елементи першого списку, за якими йдуть всі елементи другого списку, і так далі, якщо списків більше двох:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
concatenated_list = list1 + list2
print(concatenated_list)  # Виведе: [1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

Термін **«зчеплення»** зазвичай використовується менш формально і може означати те саме, що й конкатенація. В контексті програмування обидва терміни використовуються для опису об'єднання списків. Однак у деяких випадках «зчеплення» може також використовуватися для опису об'єднання не тільки списків, але й інших типів колекцій або навіть рядків:

```
list1.extend(list2)
print(list1)  # Виведе: [1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

Python надає набір вбудованих функцій та методів, які дозволяють виконувати різноманітні операції зі списками. Ось декілька з них.

Функція **len()** повертає кількість елементів у списку:

```
my_list = [1, 2, 3, 4, 5]
print(len(my_list))  # Виведе: 5
```

```
5
```

Функції **min()** та **max()** визначають відповідно мінімальне та максимальне значення серед елементів списку:

```
print(my_list)
print(min(my_list))  # Виведе: 1
print(max(my_list))  # Виведе: 5
```

```
[1, 2, 3, 4, 5]
1
5
```

Метод **index()** повертає індекс першого входження заданого елемента у списку:

```
print(my_list)
print(my_list.index(3))  # Виведе: 2
```

```
[1, 2, 3, 4, 5]
2
```

Метод **count()** повертає кількість входжень заданого елемента у списку:

```
my_list = [1, 2, 3, 4, 5]
print(my_list)
my_list.append(2)
my_list.append(2)
print(my_list)
print(my_list.count(2))  # Виведе: 3
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 2, 2]
3
```

Метод **clear()** видаляє всі елементи зі списку, залишаючи його порожнім:

```
lst = [1, 2, 3, 4, 5]
lst.clear()
print(lst)  # Виведе []
```

```
[]
```

Метод **copy()** створює поверхневу копію списку. Зміни, внесені до копії, не впливатимуть на оригінальний список, і навпаки. Для порівняння зі звичайним присвоєнням наведемо два такі приклади:

```
original_list = [1, 2, 3]
copied_list = original_list
copied_list.append(4)
print(original_list)  # Виведе [1, 2, 3, 4]
print(copied_list)  # Виведе [1, 2, 3, 4]
```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
```

```
original_list = [1, 2, 3]
copied_list = original_list.copy()
copied_list.append(4)
print(original_list)  # Виведе [1, 2, 3]
print(copied_list)  # Виведе [1, 2, 3, 4]
```

```
[1, 2, 3]
[1, 2, 3, 4]
```

Метод **sort()** сортує елементи списку в порядку зростання. Також може приймати параметри **key і reverse**, які дозволяють вказати функцію для визначення порядку сортування та здійснити сортування в оберненому порядку відповідно:

```
lst = [3, 1, 4, 1, 5, 9, 2]
lst.sort()
print(lst)  # Виведе [1, 1, 2, 3, 4, 5, 9]

lst.sort(reverse=True)
print(lst)  # Виведе [9, 5, 4, 3, 2, 1, 1]
```

```
[1, 1, 2, 3, 4, 5, 9]
[9, 5, 4, 3, 2, 1, 1]
```

Приклад 3.2. Відсортувати елементи списку за їх довжиною.

```
words = ['banana', 'pie', 'Washington', 'book']
words.sort(key=len)
print(words)  # Виведе ['pie', 'book', 'banana', 'Washington']
```

```
['pie', 'book', 'banana', 'Washington']
```

Приклад 3.3. Знайти індекс найбільшого елемента в списку.

Для цього можна скористатися функцією max для знаходження найбільшого елемента, а потім методом **index** для отримання індексу цього елемента у списку:

```
lst = [10, 65, 22, 11, 3, 85, 72]
max_value = max(lst)  # Знаходження найбільшого елемента
max_index = lst.index(max_value)  # Знаходження індексу
print(f"Індекс найбільшого елемента: {max_index}")
```

```
Індекс найбільшого елемента: 5
```

Функція **enumerate** в Python є дуже корисним інструментом для ітерації по колекціях (наприклад списках, кортежах) й одночасного отримання індексу та значення кожного елемента в колекції. Це особливо зручно, коли вам потрібно не просто пройти по елементах колекції, але й знати їх індекси.

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

iterable – колекція, яку потрібно перебирати (наприклад список, кортеж, рядок тощо);
start – значення, з якого почнеться відлік індексів (за замовчуванням 0):

```
fruits = ['apple', 'banana', 'cherry', 'date']

for index, fruit in enumerate(fruits, start = 1):
    print(index, fruit)
```

```
1 apple
2 banana
3 cherry
4 date
```

Робота з двовимірними матрицями в Python часто здійснюється за допомогою списків. Кожний підсписок представляє рядок матриці, і ви можете виконувати різні операції, такі як доступ до елементів, зміну елементів, обчислення тощо.

Наведений приклад створення матриці, визначення її розмірності та виведення кожного елемента матриці:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print("Початкова матриця:")
for row in matrix:
    print(row)
```

```
Початкова матриця:
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

```
rows = len(matrix)
cols = len(matrix[0]) if matrix else 0
print("\nРозміри матриці:", rows, "x", cols)


print("\nОбхід матриці:")
for i in range(rows):
    for j in range(cols):
        print(matrix[i][j], end=' ')
    print()  # Перехід на новий рядок
```

```
Розміри матриці: 3 x 3

Обхід матриці:
1 2 3
4 5 6
7 8 9
```

Приклад 3.4. Реалізувати програму введення матриці з клавіатури і визначення найбільшого значення та індексу елемента в матриці.

Основні аспекти, на які варто звернути увагу при аналізі програми.

Ініціалізація матриці: Користувачеві пропонується ввести кількість рядків і стовпців для матриці, що дозволяє створити матрицю потрібної розмірності. Матриця представляється як список списків (або двовимірний список), де кожний внутрішній список є рядком матриці.

Введення даних: Програма використовує вкладені цикли for для заповнення матриці. Для кожного рядка (зовнішній цикл) вона запитує елементи стовпців (внутрішній цикл), формуючи таким чином повну матрицю.

Пошук найбільшого елемента та його індексу: Після заповнення матриці програма використовує ще один набір вкладених циклів для проходження по всіх елементах матриці та знаходження найбільшого

елемента. При цьому зберігається не тільки значення найбільшого елемента, а й його індекс у формі кортежу (i, j), де i – номер рядка, а j – номер стовпця.

Виведення результатів: На завершення програма виводить весь вміст матриці, найбільше знайдене значення та індекс цього значення. Це дозволяє користувачеві не тільки побачити результат, але й переконатися в правильності введених даних.

```
# Введення розмірностей матриці
rows = int(input("Введіть кількість рядків: "))
cols = int(input("Введіть кількість стовпців: "))

# Ініціалізація матриці
matrix = []

print("Введіть елементи матриці по одному в кожному рядку:")
for i in range(rows):
    row = []
    for j in range(cols):
        # Додавання елементу в рядок
        row.append(int(input(f"Елемент [{i}][{j}]: ")))
    # Додавання рядка до матриці
    matrix.append(row)

# Пошук найбільшого значення та його індексу
max_value = matrix[0][0]
max_index = (0, 0)
for i in range(rows):
    for j in range(cols):
        if matrix[i][j] > max_value:
            max_value = matrix[i][j]
            max_index = (i, j)

print("\nВаша матриця:")
for row in matrix:
    print(row)

print(f"Найбільше значення в матриці: {max_value}, індекс: {max_index}")
```

Результат роботи програми:

```
Введіть кількість рядків: 2
Введіть кількість стовпців: 2
Введіть елементи матриці по одному в кожному рядку:
Елемент [0][0]: 2
Елемент [0][1]: 7
Елемент [1][0]: 11
Елемент [1][1]: 3

Ваша матриця:
[2, 7]
[11, 3]
Найбільше значення в матриці: 11, індекс: (1, 0)
```

Наведені та додаткові функції роботи зі списками подано в таблицях 3.1 та 3.2.

Таблиця 3.1 – Пошук та заміна в рядках

Метод	Опис методу
s.count(p)	Повертає кількість входжень рядка p до рядка s
s.find(p, i, j)	Індекс першого входження рядка p у рядок s. Пошук здійснюється у зрізі s[i:j]. Параметри i і j мають типові значення – 0 і кількість символів у рядку відповідно. Повертає значення -1 якщо p не знайдено.
s.index(p, i, j)	Індекс першого входження рядка p у рядок s. Пошук здійснюється у зрізі s[i:j]. Параметри i і j мають типові значення – 0 і кількість символів у рядку відповідно. Породжує помилку, якщо p не знайдено.
s.replace(old, new, count)	Повертає рядок s, в якому всі входження рядка old замінено рядком new. Параметр count можна опустити. Якщо задано count, то замінюється не більше count перших входжень.

Таблиця 3.2 – Методи аналізу в рядках

Метод	Опис методу
s.isalnum()	Повертає **True**, якщо всі символи рядка s є літерами або цифрами
s.isalpha()	Повертає **True**, якщо всі символи рядка s є літерами
s.isdigit()	Повертає **True**, якщо всі символи рядка s є цифрами.
s.isidentifier()	Повертає **True**, якщо рядок s є ідентифікатором.
s.islower()	Повертає **True**, якщо всі літери рядка s у нижньому регістрі.
s.isnumeric()	Повертає **True**, якщо всі символи рядка s є числовими.
s.isprintable()	Повертає **True**, якщо всі символи рядка s є друкованими.
s.isspace()	Повертає **True**, якщо всі символи рядка s є пропусками.
s.istitle()	Повертає **True**, якщо рядок s є заголовком (усі слова починаються з великої літери).
s.isupper()	Повертає **True**, якщо всі літери рядка s у верхньому регістрі.

3.3 Робота з кортежами (Tuples)

Кортежі в Python – це незмінні (*immutable*) послідовності, які використовуються для зберігання декількох об'єктів разом. Від списків вони відрізняються тим, що їх не можна змінювати після створення. Це означає, що елементи кортежу не можна змінювати, додавати або видаляти.

Створення кортежів в Python можливо кількома способами.

1) Використання круглих дужок для набору елементів:

```
my_tuple = (1, 2, 3)
print(my_tuple)  # Вивід: (1, 2, 3)
```

```
(1, 2, 3)
```

2) Одноелементний кортеж. Для створення кортежу, що містить лише один елемент, необхідно поставити кому після цього елемента. Це важливо, оскільки без коми Python не розпізнає вираз як кортеж:

```
single_element_tuple = (4,)
print(single_element_tuple)  # Вивід: (4,)
```

```
(4,)
```

3) Використання вбудованої функції **tuple()**

```
list_to_tuple = tuple([1, 2, 3])
print(list_to_tuple)  # Вивід: (1, 2, 3)

string_to_tuple = tuple('hello')
print(string_to_tuple)  # Вивід: ('h', 'e', 'l', 'l', 'o')
```

```
(1, 2, 3)
('h', 'e', 'l', 'l', 'o')
```

Пакування кортежу (tuple packing) в Python – це процес, за допомогою якого декілька значень об'єднуються в один кортеж без використання явних дужок. Це одна з основних особливостей кортежів, що дозволяє зручно групувати дані.

```
# Пакування значень 1, 2, 3 в кортеж
my_tuple = 1, 2, 3

print(my_tuple)  # Виведе: (1, 2, 3)
```

```
(1, 2, 3)
```

Розпакування кортежу в Python – це процес присвоєння значень з кортежу до змінних. Це дозволяє легко розділити дані, які зберігаються в кортежі, на окремі змінні:

```python
# Визначаємо кортеж із більшою кількістю елементів
my_tuple = (1, 2, 3, 4, 5)

# Використовуємо розширене розпакування
a, b, *rest = my_tuple

# Виводимо значення
print(a)  # Виведе: 1
print(b)  # Виведе: 2
print(rest)  # Виведе: [3, 4, 5]

# Альтернативно, можна розпакувати останній елемент окремо
a, *middle, c = my_tuple

# Виводимо значення
print(a)  # Виведе: 1
print(middle)  # Виведе: [2, 3, 4]
print(c)  # Виведе: 5
```

```
1
2
[3, 4, 5]
1
[2, 3, 4]
5
```

Розширене розпакування особливо корисне, коли ви працюєте з кортежами невідомої заздалегідь довжини і хочете ігнорувати частину елементів або зберегти їх окремо.

В Python **обмін значеннями** між двома змінними можна здійснити в один рядок без використання додаткової тимчасової змінної:

```python
# Початкові значення змінних
a = 5
b = 10

# Обмін значеннями
a, b = b, a

# Виведення результату
print("a:", a)  # a: 10
print("b:", b)  # b: 5
```

```
a: 10
b: 5
```

Цей прийом використовує розпакування кортежів для одночасного присвоєння нових значень змінним a і b. Коли ви виконуєте a, b = b, a, Python спочатку створює кортеж зі значеннями b і a (в цьому випадку (10, 5)), а потім одразу розпаковує цей кортеж у змінні a і b, тим самим обмінюючи їх значеннями.

Приклад 3.5. Обчислити задане число Фібоначчі при застосуванні кортежів.

Для довідки: числа Фібоначчі – це послідовність чисел, в якій кожне наступне число є сумою двох попередніх. Послідовність починається з 0 і 1, і кожне наступне число визначається як сума двох попередніх. Отже, послідовність чисел Фібоначчі виглядає так: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 і так далі.

```
n = int(input("Введіть порядковий номер числа Фібоначчі: "))

a, b = 0, 1
for _ in range(n):
    a, b = b, a + b

print(f"{n}-те число Фібоначчі: {a}")
```

```
Введіть порядковий номер числа Фібоначчі: 10
10-те число Фібоначчі: 55
```

У цьому коді:

a і **b** ініціалізуються першими двома числами послідовності Фібоначчі: **0** і **1** відповідно.

Цикл **for** використовується для ітерації через послідовність чисел від **0** до **n-1**. Зверніть увагу, що ми використовуємо **_ як змінну циклу**, що є звичайною практикою, коли змінна циклу не використовується в тілі циклу.

Під час кожної ітерації **a** і **b** переприсвоюються новими значеннями: **a** отримує значення **b**, а **b** отримує суму попередніх значень **a** і **b**, що відповідає наступному числу в послідовності Фібоначчі.

Після завершення циклу **a** містить значення n-го числа Фібоначчі, яке виводиться на екран.

Отриманий результат можна порівняти з *прикладом 2.13.*

3.4 Генератор-вирази для послідовностей

Генератор-вирази (generator expressions) дозволяють створювати генератори за допомогою синтаксису, схожого на включення списку (list comprehension), але замість квадратних дужок використовуються круглі. Генератори – це ітератори, що виробляють елементи послідовностей лише за потребою, замість створення всієї послідовності відразу. Це робить їх більш ефективними з погляду використання пам’яті для великих послідовностей, оскільки вони не потребують збереження всієї послідовності в пам’яті одразу.

Генератор-вираз має такий синтаксис:

```
(element for element in iterable if condition)
```

де **element** – це вираз, що визначає, яким буде кожний елемент генерованої послідовності, **iterable** – ітерабельний об’єкт, який ми проходимо, а **condition** – необов’язкова умова фільтрації для елементів.

Приклад 3.6. Створити генератор для знаходження квадратів непарних чисел у заданому діапазоні:

```
n = 20
gen_expr = (x ** 2 for x in range(n) if x % 2 != 0)

for num in gen_expr:
    print(num, end = " ")
```

```
1 9 25 49 81 121 169 225 289 361
```

У цьому прикладі **gen_expr** створює генератор, який виробляє квадрати всіх непарних чисел від 0 до 9. Цикл **for** ітерує через генератор, виводячи кожне значення. Важливо зазначити, що значення виробляються по одному, і весь діапазон не зберігається в пам'яті одразу.

Приклад 3.7. Створити генератор коренів квадратних цілих чисел від 0 до 4:

```
import math

sqrt_generator = (math.sqrt(x) for x in range(0, 5))
for num in sqrt_generator:
    print(f"sqrt_x = {num:.2f}") # Виведе квадратні корені
                                 # для цілих чисел від 0 до 4
```

```
sqrt_x = 0.00
sqrt_x = 1.00
sqrt_x = 1.41
sqrt_x = 1.73
sqrt_x = 2.00
```

Приклад 3.8. Утворити пари чисел у заданому діапазоні, які не дорівнюють один одному:

```
nested = ((x, y) for x in range(3) for y in range(3) if x != y)
print(list(nested))
# Виведе пари чисел, де x не дорівнює y
```

```
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

Приклад 3.9. Визначити наближено значення числа π при застосуванні формули Грегорі–Лейбніца.

Формула Грегорі–Лейбніца для обчислення числа π є однією з найпростіших і водночас найвідоміших формул, яка має вигляд:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} \cdots$$

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

Ця формула збігається дуже повільно, тому для отримання хоча б кількох знаків після коми потрібно виконати велику кількість ітерацій. Нижче наведено приклад програми на Python, яка використовує цю формулу для наближеного обчислення числа π.

```
n_terms = 100000  # Визначаємо кількість ітерацій

pi_estimate = 0  # Початкове значення для наближення числа π

for n in range(n_terms):
    term = (-1)**n / (2*n + 1)  # Обчислюємо кожен терм ряду
    pi_estimate += term  # Додаємо терм до наближення числа π

pi_estimate *= 4  # Множимо на 4, як вимагає формула

print(f"Наближене значення числа π (після {n_terms} ітерацій): {pi_estimate}")
```

```
Наближене значення числа π (після 100000 ітерацій): 3.1415826535897198
```

3.5 Робота з символами

Поняття символу в комп'ютерній техніці. У комп'ютерній техніці «символ» – це найменша неділима одиниця тексту. Це може бути буква, цифра, пробіл, пунктуаційний або інший знак, який можна відобразити за допомогою комп'ютера або ввести за допомогою клавіатури. Символи використовуються для створення слів, речень та інших форм текстової інформації.

Комп'ютери працюють лише з числами, тому для представлення символів використовуються спеціальні кодування – системи присвоєння числових значень різним символам. Найпростіші кодування мапують символи до цілих чисел. Наведемо найвідоміші приклади таких систем кодування.

ASCII (American Standard Code for Information Interchange). Це одне з найбільш ранніх кодувань, яке використовується для представлення англійських літер, цифр, керуючих та деяких інших символів. Воно використовує 7 бітів для кожного символу, що дозволяє кодувати 128 різних символів (таблиця 3.3).

Розширене ASCII (Extended ASCII). Це набори кодувань, які використовують 8 бітів для символу і розширюють базовий набір ASCII до 256 символів, додаючи додаткові символи, такі як латинські букви з діакритичними знаками, графічні символи тощо.

Unicode. Сучасна система кодування, розроблена для представлення тексту майже всіма писемними мовами світу. Unicode може використовувати різну кількість бітів для кодування символів (найчастіше використовуються формати UTF-8, UTF-16, UTF-32), і вона включає понад 143 тисячі символів.

Таблиця 3.3 – ASCII-таблиця кодування

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

У Python, починаючи з версії 3, всі рядки (str тип) за замовчуванням представлені у вигляді тексту в кодуванні Unicode, дозволяючи безпроблемно працювати з текстом, написаним різними мовами і символами. Це означає, що Python автоматично обробляє і зберігає текст у форматі Unicode внутрішньо, а не конкретно в UTF-8. Однак, коли йдеться про збереження тексту в файлі або передачу його по мережі, часто використовується кодування UTF-8, яке є одним із найпопулярніших способів кодування тексту Unicode завдяки своїй ефективності та широкій підтримці.

UTF-8 – це змінна довжина кодування, яка означає, що різні символи можуть займати від 1 до 4 байтів. Це кодування забезпечує повну сумісність з ASCII для символів з ASCII-діапазону, роблячи текст, який складається лише з символів ASCII, ідентичним в обох кодуваннях.

Приклад 3.10. Приклад кодування та декодування в Python

```
# Оригінальний Unicode рядок
original_string = "Привіт, світ!"

# Кодування рядка в UTF-8
encoded_string = original_string.encode('utf-8')
print(encoded_string)  # Виведе байтовий рядок

# Декодування назад у Unicode (str в Python 3)
decoded_string = encoded_string.decode('utf-8')
print(decoded_string)  # Виведе оригінальний рядок "Привіт, світ!"
```

```
b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd1\x96\xd1\x82, \xd1\x81\xd0\xb2\xd1\x96\xd1\x82!'
Привіт, світ!
```

Цей приклад демонструє, як рядок може бути закодований у байтовий рядок за допомогою UTF-8, а потім декодований назад у рядок Unicode. Важливо розуміти різницю між типом str в Python 3, який

представляє текст у форматі Unicode, і типом bytes, який представляє байтові послідовності, включаючи закодований текст.

В Python **спеціальні (екрановані) символи** являють собою символи, які мають особливе значення в рядкових літералах і передаються з використанням зворотного слеша (). Ці символи дозволяють вставляти в рядки символи, які або важко ввести з клавіатури, або виконують спеціальні функції, як-от новий рядок чи табуляція. Ось деякі з найбільш використовуваних екранованих символів у Python:

\n – новий рядок, використовується для переходу на новий рядок;

\t – горизонтальна табуляція, використовується для вставки табуляції у текст;

\r – повернення каретки, використовується для повернення курсору на початок поточного рядка без переходу на новий рядок;

\\ – зворотний слеш, використовується для вставки самого зворотного слеша в рядок;

\' – одинарна лапка, використовується в рядках, обмежених одинарними лапками, для представлення самої одинарної лапки або апострофа;

\" – подвійна лапка, використовується в рядках, обмежених подвійними лапками, для представлення самої подвійної лапки;

\b – крок назад, використовується для видалення попереднього символу в тексті;

\f – подача форми, використовується для переходу на нову сторінку.

Деякі елементи застосування спеціальних символів продемонстровані в коді:

```
print("Це приклад утворення двох нових рядків:\n\ni ось ми на новому рядку.")
print("Це приклад табуляції:\tпісля табуляції.")
print("Це приклад подвійної лапки: \" і одинарної лапки: \'.")
```

```
Це приклад утворення двох нових рядків:

і ось ми на новому рядку.
Це приклад табуляції:   після табуляції.
Це приклад подвійної лапки: " і одинарної лапки: '.
```

В Python робота з символами та їх перетворення згідно з ASCII (American Standard Code for Information Interchange) здійснюється завдяки вбудованим функціям ord() та chr().

Функція ord(). Функція ord() приймає символ (рядок довжиною в один символ) та повертає його числове представлення в кодуванні Unicode, яке для стандартних символів ASCII збігається з кодами ASCII (таблиця 3.3):

```
# Отримати ASCII код символу 'A'
code = ord('A')
print(code)  # Виведе: 65

# Отримати ASCII код символу 'a'
code = ord('a')
print(code)  # Виведе: 97
```

```
65
97
```

Функція chr(). На відміну від ord(), функція chr() приймає ціле число (код символу) та повертає відповідний йому символ у кодуванні Unicode. Для значень у діапазоні від 0 до 127 результати будуть відповідати символам ASCII (таблиця 3.3):

```
# Отримати символ з ASCII коду 65
symbol = chr(65)
print(symbol)  # Виведе: 'A'

# Отримати символ з ASCII коду 97
symbol = chr(97)
print(symbol)  # Виведе: 'a'
```

```
A
a
```

Приклад 3.11. Визначити належність символу до латинського алфавіту (велика чи маленька літера), до цифри або до символу, який не є ні літерою, ні цифрою.

Цю задачу розв'яжемо двома способами.

1-й спосіб. Для визначення, чи є певний символ латинською літерою (великою або маленькою), цифрою, або ні тим, ні іншим, можна використати стандартні функції мови Python: *str.isalpha()*, *str.isupper()*, *str.islower()* та *str.isdigit()* (див. таблицю 3.2). Нижче наведено приклад реалізації задачі на Python.

```
character = input("Введіть символ: ") # Вкажіть символ для перевірки тут

if len(character) != 1:
    result = "Будь ласка, введіть один символ."
else:
    if character.isalpha():  # Перевірка на латинську літеру
        if character.isupper():  # Велика латинська літера
            result = f"Символ '{character}' є великою латинською літерою."
        else:  # Маленька латинська літера
            result = f"Символ '{character}' є маленькою латинською літерою."
    elif character.isdigit():  # Цифра
        result = f"Символ '{character}' є цифрою."
    else:  # Ні тим ні іншим
        result = f"Символ '{character}' не є ні латинською літерою, ні цифрою."

print(result)
```

```
Введіть символ: A
Символ 'A' є великою латинською літерою.
```

```
Введіть символ: #
Символ '#' не є ні латинською літерою, ні цифрою.
```

2-й спосіб. У таблиці ASCII (таблиця 3.3) коди символів визначаються таким чином:

- великі латинські літери: від 65 ('A') до 90 ('Z');
- маленькі латинські літери: від 97 ('a') до 122 ('z');
- цифри: від 48 ('0') до 57 ('9').

```python
character = input("Введіть символ: ") # Вкажіть символ для перевірки тут

if len(character) != 1:
    result = "Будь ласка, введіть один символ."
else:
    char_code = ord(character)  # Перетворює символ у його код ASCII

    if 65 <= char_code <= 90:  # Велика латинська літера
        result = f"Символ '{character}' є великою латинською літерою."
    elif 97 <= char_code <= 122:  # Маленька латинська літера
        result = f"Символ '{character}' є маленькою латинською літерою."
    elif 48 <= char_code <= 57:  # Цифра
        result = f"Символ '{character}' є цифрою."
    else:  # Ні тим ні іншим
        result = f"Символ '{character}' не є ні латинською літерою, ні цифрою."

print(result)
```

```
Введіть символ: A
Символ 'A' є великою латинською літерою.
```

```
Введіть символ: 1
Символ '1' є цифрою.
```

3.6 Робота зі словниками та множинами

Представлені раніше колекції (списки, кортежі, рядки) були впорядкованими, тобто всі елементи у колекції займали свою чітко визначену позицію. Таким чином, до елементів цих типів можна здійснювати доступ за індексами.

Окрім впорядкованих типів даних, у Python існують невпорядковані колекції, у яких не визначено порядок доступу до елементів. До таких типів відносяться словники та множини.

У Python словники (**dict**) – це вбудований тип даних, що використовується для зберігання даних у парах **ключ-значення**. Словники є надзвичайно гнучкими структурами даних, які дозволяють швидко отримувати доступ до значень, знаючи відповідні ключі. Вони мутабельні, тобто можуть бути змінені після створення. Ключі в словнику повинні бути унікальними і незмінними (**immutable**), тому як ключі можна використовувати рядки, числа, кортежі (якщо вони містять тільки незмінні об’єкти) та інші незмінні типи.

Основні властивості і призначення словників:

швидкий доступ до даних – доступ до значень у словнику здійснюється за допомогою ключів, що дозволяє швидко знаходити необхідну інформацію;

зберігання даних у вигляді пар ключ-значення – це корисно для представлення зв’язаних між собою даних, наприклад інформації про співробітника, де ключами можуть бути «ім’я», «посада», «відділ» і так далі;

гнучкість у роботі з даними – можливість додавання, видалення та зміни пар ключ-значення;

використання як **«швидкісні таблиці»** – застосовуються для швидкого відображення одних значень на інші, оптимізації пошуку та видалення елементів.

Структура словника my_dict в Python є досить простою та інтуїтивно зрозумілою. Словник складається з пар ключ-значення, де кожний ключ унікальний у межах словника, і кожному ключу відповідає певне значення. У цьому прикладі наявні ключі та значення:

```
# Створення словника
my_dict = {"name": "Petrenko", "age": 20, "city": "Cherkasy"}
```

Ключі – у цьому словнику використовуються три ключі, які є рядками (стрічками) – "name", "age" та "city". Ці ключі служать ідентифікаторами для значень, що зберігаються в словнику.

Значення – кожному ключу присвоєно певне значення. Для ключа "name" значенням є рядок "Perrenko", для "age" – це число 20, і для "city" – рядок "Cherkasy". Значення можуть бути різного типу: рядки, числа, списки, інші словники тощо.

Словник зберігає дані у вигляді пар **ключ-значення**, розділених комами. Ключ і відповідне йому значення розділені двокрапкою. У цьому прикладі є три такі пари.

Фігурні дужки – словник визначається за допомогою фігурних дужок {}. Всередині цих дужок розташовуються пари ключ-значення.

Важливо зазначити, що ключі в словнику повинні бути унікальними, тобто ви не можете мати два однакові ключі в одному словнику, але значення можуть повторюватися. Якщо спробувати додати до словника нову пару ключ-значення, де ключ вже існує, старе значення буде перезаписано новим.

В наступному прикладі продемонстровано доступ до елемента словника, зміна значення, видалення пари ключ-значення, перебір ключів і значень словника.

```
# Створення словника
my_dict = {"name": "Petrenko", "age": 20, "city": "Cherkasy"}

# Доступ до елемента словника
print(my_dict["name"])  # Виведе: Petrenko

# Додавання нової пари ключ-значення
my_dict["email"] = "petrenko@example.com"

# Зміна значення за ключем
my_dict["age"] = 25

# Видалення пари ключ-значення
del my_dict["city"]
# або за допомогою методу pop
# my_dict.pop("city")

# Перевірка наявності ключа в словнику
if "name" in my_dict:
    print("Ім'я присутнє в словнику.")

# Перебір ключів і значень словника
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

```
Petrenko
Ім'я присутнє в словнику.
name: Petrenko
age: 25
email: petrenko@example.com
```

У Python існує кілька способів **створення словників**. Ось деякі з найпоширеніших методів з прикладами:

1. Використання літеральної нотації – словник можна створити безпосередньо, використовуючи **літеральну нотацію** з фігурними дужками {}:

```
# Створення словника
my_dict = {"name": "Petrenko", "age": 20, "city": "Cherkasy"}
print(person)
```

```
{'name': 'Petrenko', 'age': 30, 'city': 'Cherkasy'}
```

2. Використання конструктора **dict()** для створення словника. Це може бути корисно, коли ключі словника є простими рядками:

```
person = dict(name="Petrenko", age=30, city="Cherkasy")
print(person)
```

```
{'name': 'Petrenko', 'age': 30, 'city': 'Cherkasy'}
```

3. Ініціалізація за допомогою **пар ключ-значення**. Можна створити словник, передавши dict() ітерабельний об’єкт, що складається з пар ключ-значення:

```
person = dict([("name", "Petrenko"), ("age", 30), ("city", "Cherkasy")])
print(person)
```

```
{'name': 'Petrenko', 'age': 30, 'city': 'Cherkasy'}
```

4. Ініціалізація за допомогою ключів та значень. Метод **fromkeys()** дозволяє створити новий словник із заданими ключами, кожному з яких присвоєно однакове значення (або None, якщо значення не задано):

```
keys = ["name", "age", "city"]
value = "unknown"
person = dict.fromkeys(keys, value)
print(person)
```

```
{'name': 'unknown', 'age': 'unknown', 'city': 'unknown'}
```

5. Ініціалізація за допомогою генератора словників. Генератор словників (**dict comprehension**) дозволяє створювати словники з виразів, що генерують пари ключ-значення, подібно до генераторів списків.

```
squares = {x: x*x for x in range(6)}
print(squares)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Доступ до елементів словника в Python можна отримати за допомогою ключа. Якщо вам потрібно отримати значення певного ключа, ви використовуєте синтаксис з квадратними дужками **[ ]** або метод **get().** Ось приклади обох способів:

```
# Створення словника
person = {"name": "Petrenko", "age": 30, "city": "Cherkasy"}

# Доступ до значення за ключем
print(person["name"])  # Виводить "Petrenko"
print(person["age"])   # Виводить 30
```

Метод **get()** є безпечним способом отримання значення за ключем, оскільки він дозволяє уникнути виникнення помилки KeyError, якщо ключ не знайдений. Це особливо корисно в ситуаціях, коли ви не впевнені у наявності ключа в словнику:

```
# Створення словника
person = {"name": "Petrenko", "age": 30, "city": "Cherkasy"}

# Доступ до значення за ключем за допомогою методу get()
print(person.get("name"))  # Виводить "Petrenko"
print(person.get("age"))   # Виводить 30

# Доступ до неіснуючого ключа без помилки, поверне None
print(person.get("job"))   # Виводить None

# Можна задати значення за замовчуванням, яке повернеться, якщо ключ не знайдено
print(person.get("job", "Unemployed"))  # Виводить "Unemployed"
```

```
Petrenko
30
None
Unemployed
```

Обхід словника в Python можна виконати кількома способами залежно від того, яку інформацію потрібно отримати: ключі, значення чи пари ключ-значення. Нижче наведено приклади кожного з цих способів:

```
# Створення словника
person = {"name": "Petrenko", "age": 30, "city": "Cherkasy"}

# Виведення всіх ключів
for key in person:
    print(key)
```

```
name
age
city
```

Для отримання всіх значень із словника:

```python
# Виведення всіх значень
for value in person.values():
    print(value)
```

```
Petrenko
30
Cherkasy
```

Якщо потрібні і ключі, і значення:

```python
# Виведення пар ключ-значення
for key, value in person.items():
    print(f"Key: {key}, Value: {value}")
```

```
Key: name, Value: Petrenko
Key: age, Value: 30
Key: city, Value: Cherkasy
```

Приклад 3.12. Визначити кількість входжень слів у реченні і представити результат у вигляді словника

```python
# Вхідне речення
sentence = "Це речення, для якого деякі слова зустрічаються більше одного разу,\
і це нормально для цього речення."
# Перетворення речення на нижній регістр і видалення пунктуації
# Для спрощення припустимо, що пунктуацією є стандартні знаки
import string
sentence = sentence.lower().translate(str.maketrans('', '', string.punctuation))
# Розділення речення на слова
words = sentence.split()

# Створення словника для зберігання кількості входжень кожного слова
word_counts = {}
# Підрахунок входжень кожного слова
for word in words:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
# Виведення результату
for word, count in word_counts.items():
    print(f"{word}: {count}")
```

```
це: 2
речення: 2
для: 2
якого: 1
деякі: 1
слова: 1
зустрічаються: 1
більше: 1
одного: 1
разуі: 1
нормально: 1
цього: 1
```

Спочатку речення перетворюється на нижній регістр і очищається від пунктуаційних знаків. Це дозволяє уникнути дублікацій через різницю в регістрах або наявність знаків пунктуації.

Потім речення розділяється на слова за допомогою методу split(). Далі йде проходження по списку слів із підрахунком їх входжень, результати зберігаються в словнику word_counts.

На останньому кроці виводиться словник з кількістю входжень кожного слова.

Цей код ефективно підраховує кількість входжень кожного слова в реченні і представляє результат у форматі словника, де ключами є слова, а значеннями – кількість їх входжень.

Множини. Множини в Python – це колекції, що містять неповторні елементи в невпорядкованому порядку. Вони є дуже корисними для виконання математичних операцій множин, таких як об'єднання, перетин, різниця та симетрична різниця.

Призначення множин:

уникнення дублікатів: Множини автоматично видаляють повторювані елементи, тому вони корисні для видалення дублікатів з послідовності;

операції над множинами: Виконання математичних операцій множин, таких як об'єднання, перетин, різниця;

перевірка належності: Множини надають ефективний спосіб перевірки належності елемента до колекції.

Множини в Python можна створити за допомогою фігурних дужок {} або вбудованої функції set(). Порожню множину можна створити тільки за допомогою set(), оскільки порожні фігурні дужки {} створюють порожній словник:

```
# Створення множини за допомогою фігурних дужок
my_set = {1, 2, 3, 4, 5}
print(my_set)

# Створення множини за допомогою функції set()
my_set2 = set([1, 2, 2, 3, 4])
print(my_set2)  # Повторювані елементи будуть видалені

# Створення порожньої множини
my_set3 = set()
print(my_set3)  # Виведе set()
```

```
{1, 2, 3, 4, 5}
{1, 2, 3, 4}
set()
```

Операції над множинами:

```python
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

# Об'єднання множин
print(a | b)  # {1, 2, 3, 4, 5, 6}

# Перетин множин
print(a & b)  # {3, 4}

# Різниця множин
print(a - b)  # {1, 2}
print(b - a)  # {5, 6}

# Симетрична різниця (елементи, що є в одній множині,
# але не є в обох)
print(a ^ b)  # {1, 2, 5, 6}
```

```
{1, 2, 3, 4, 5, 6}
{3, 4}
{1, 2}
{5, 6}
{1, 2, 5, 6}
```

Можна **додати елемент** до множини, використовуючи метод add(). Якщо елемент вже існує у множині, то він не буде доданий знову, оскільки всі елементи в множині є унікальними:

```python
# Створення множини
my_set = {1, 2, 3}
print("Початкова множина:", my_set)

# Додавання елементу до множини
my_set.add(4)
print("Множина після додавання елементу:", my_set)
```

```
Початкова множина: {1, 2, 3}
Множина після додавання елементу: {1, 2, 3, 4}
```

Видалення елементів. Є кілька методів для видалення елементів з множини:

remove(): видаляє заданий елемент з множини, якщо елемента не існує, Python повідомить помилку;

discard(): також видаляє заданий елемент з множини, але якщо елемента не існує, то помилка не повідомляється;

pop(): видаляє та повертає елемент з множини. Оскільки множини є невпорядкованими, не можна контролювати, який саме елемент буде видалено;

clear(): видаляє всі елементи з множини, залишаючи її порожньою.

```python
# Видалення елемента методом remove()
my_set.remove(4)
print("Множина після видалення елемента методом remove():", my_set)

# Видалення елемента методом discard()
my_set.discard(3)
print("Множина після видалення елемента методом discard():", my_set)

# Видалення елемента методом pop()
popped_element = my_set.pop()
print("Видалений елемент методом pop():", popped_element)
print("Множина після видалення елемента методом pop():", my_set)

# Очищення множини методом clear()
my_set.clear()
print("Множина після очищення методом clear():", my_set)
```

```
Множина після видалення елемента методом remove(): {1, 2, 3}
Множина після видалення елемента методом discard(): {1, 2}
Видалений елемент методом pop(): 1
Множина після видалення елемента методом pop(): {2}
Множина після очищення методом clear(): set()
```

Приклад 3.13. Маємо набір даних про використання телефонної мережі в місті. Кожний запис у наборі даних являє собою дзвінок між двома номерами телефонів у певний час. Знайти кількість унікальних номерів телефонів, з яких було здійснено дзвінки, і визначити загальну кількість здійснених дзвінків.

```python
# Припустимо, у нас є набір даних з записами про дзвінки
calls_data = [
    {"from": "123456789", "to": "987654321", "duration": 60},
    {"from": "987654321", "to": "555555555", "duration": 120},
    {"from": "123456789", "to": "987654321", "duration": 10},
    {"from": "123456789", "to": "987654321", "duration": 20},
    # Інші записи про дзвінки...
]

# Створимо множину для зберігання унікальних номерів телефонів
unique_phone_numbers = set()

# Додамо номери телефонів з кожного запису у множину
for call in calls_data:
    unique_phone_numbers.add(call["from"])
    unique_phone_numbers.add(call["to"])

# Обчислимо кількість унікальних номерів телефонів
unique_phone_numbers_count = len(unique_phone_numbers)

# Обчислимо загальну кількість здійснених дзвінків
total_calls_count = len(calls_data)

print("Кількість унікальних номерів телефонів:", unique_phone_numbers_count)
print("Загальна кількість здійснених дзвінків:", total_calls_count)
```

```
Кількість унікальних номерів телефонів: 3
Загальна кількість здійснених дзвінків: 4
```

3.7 Генерація випадкових чисел

Генерація випадкових чисел має критичне значення в багатьох областях, зокрема в симуляціях, тестуванні програмного забезпечення, статистиці та ін. Основні підходи до генерації випадкових чисел можна класифікувати на дві великі групи: використання апаратних генераторів випадкових чисел (АГВЧ) та алгоритмічне генерування (або псевдовипадкові числа).

АГВЧ використовують фізичні процеси для генерації випадковості, як-от радіоактивний розпад, тепловий шум або фотонний шум. Вони надають «істинні» випадкові числа, оскільки базуються на непередбачуваних фізичних явищах.

На відміну від АГВЧ, псевдовипадкові генератори чисел (ПВГЧ) використовують математичні алгоритми для генерації послідовностей чисел, які виглядають випадковими, але насправді є повністю визначеними попереднім станом генератора (наприклад початковим значенням або «зерном»).

Призначення ПВГЧ – широко використовуються в симуляціях, статистичних моделях, комп'ютерних іграх, тестуванні програмного забезпечення, де є потреба у великій кількості «випадкових» даних, але справжня випадковість не є критичною.

У Python для генерації псевдовипадкових чисел використовується модуль **random**. Він надає різноманітні функції для створення випадкових чисел, вибору випадкових елементів зі списків, перемішування даних тощо.

Приклад 3.14. Сформувати випадкове число і випадковий вибір із заданої послідовності.

```
import random

# Генерація випадкового числа від 0 до 1
print(random.random())

# Генерація випадкового цілого числа від 1 до 10
print(random.randint(1, 10))

# Вибір випадкового елементу зі списку
print(random.choice(['яблуко', 'банан', 'вишня']))

# Перемішування списку
list_example = [1, 2, 3, 4, 5]
random.shuffle(list_example)
print(list_example)
```

```
0.5423511421422376
10
вишня
[3, 4, 5, 1, 2]
```

Метод **seed()** використовується у модулі random мови програмування Python для ініціалізації початкового числа (seed) генератора псевдовипадкових чисел. Це дозволяє відтворювати результати генерації випадкових чисел. Якщо seed() викликаний з однаковим аргументом, послідовність випадкових чисел, які генеруються після цього, буде однаковою при кожному виконанні програми:

```
import random
# Ініціалізація генератора
# псевдовипадкових чисел з конкретним початковим числом
random.seed(42)

# Генерація випадкового числа
print(random.random())  # Виведе псевдовипадкове число між 0 і 1
```

```
0.6394267984578837
```

Приклад 3.15. Сгенерувати випадковий пароль заданої довжини при використанні літер, цифр і спеціальних символів.

```
import random
import string

random.seed(42)
length = 22  # Встановлюємо довжину пароля

# Символи для генерації пароля
characters = string.ascii_letters + string.digits + string.punctuation
# Генерація пароля
password = ''.join(random.choice(characters) for i in range(length))

print(password)  # Виводимо пароль
```

```
>odJFCrn](l.2edlBD#:d*
```

Приклад 3.16. Заповнити матрицю розмірності 3x3 випадковими числами в діапазоні від 1 до 100.

Для заповнення матриці 3x3 випадковими числами з використанням обходу в циклі без застосування бібліотеки NumPy можна використати стандартний модуль random для генерації випадкових чисел:

```
import random

random.seed(42)
# Ініціалізація матриці 3x3 нулями
matrix = [[0 for _ in range(3)] for _ in range(3)]

# Заповнення матриці випадковими числами від 1 до 100
for i in range(3):
    for j in range(3):
        # Випадкове число від 1 до 100
        matrix[i][j] = random.randint(1, 100)

# Виведення матриці
for row in matrix:
    print(row)
```

```
[82, 15, 4]
[95, 36, 32]
[29, 18, 95]
```

У цьому прикладі спочатку створюється матриця matrix розміром 3x3, яка ініціалізується нулями. Потім за допомогою вкладених циклів for матриця заповнюється випадковими цілими числами від 1 до 100, генерованими функцією **random.randint(1, 100)**. В кінці виконується виведення кожного рядка матриці на екран.

Цю задачу можна розв'язати й інакше при застосуванні бібліотеки NumPy:

```
import numpy as np

# Ініціалізація генератора випадкових чисел
np.random.seed(42)

# Створення матриці 3x3 з
# випадковими цілими числами від 1 до 100
matrix = np.random.randint(1, 100, size=(3, 3))

print(matrix)
```

```
[[52 93 15]
 [72 61 21]
 [83 87 75]]
```

У цьому прикладі, після ініціалізації випадкового генератора чисел з використанням seed(42), створюється матриця 3x3 з випадковими числами, які генеруються функцією np.random.rand. Завдяки використанню seed ця матриця буде мати однакові значення кожного разу при виконанні коду.

ПРАКТИЧНА ЧАСТИНА

3.8 Підготовка до виконання завдання:

1. Ознайомтеся з теоретичними відомостями щодо організації і роботи зі структурами даних мовою Python: списки, кортежі, словники, множини.
2. Опрацюйте приклади, наведені в теоретичній частині.

3.9 Практичне завдання

Згідно зі своїм варіантом напишіть програму для виконання наступних завдань.

Варіанти завдання № 1

Варіант	***Завдання***
1	Дано список натуральних чисел розмірності N. Створіть новий список з наданого, який містить: a) всі парні значення з заданого списку; b) всі значення списку, що є його квадратами; c) квадрати всіх непарних значень заданого списку.
2	Заданий список дійсних чисел $a1, a2, \ldots, an$. Складіть програми для знаходження: - $\min(a1, a2, \ldots, an)$; - $\min(a2, a4, \ldots) + \max(a1, a3, \ldots)$; - $\max(a1, 2a2, \ldots, nan)$.

3	Дано список натуральних чисел розмірності N. Перевірте, чи впорядкований числовий список за спаданням.
4	Дано двовимірну матрицю розміром m*n. Напишіть програми для обчислення: a) суми всіх елементів матриці, що належать головній діагоналі і є більшими за число 2; b) кількості нульових елементів матриці; c) суми елементів першого і останнього рядка.
5	Дано двовимірну матрицю розміром m*n . Напишіть програму пошуку заданого елемента в матриці та виведення його індексів.
6	Напишіть програму, яка знаходить і виводить всі дублікати в списку. Наприклад, для списку [1, 2, 3, 2, 1, 5, 6, 5, 5, 5] має вивести 1, 2, 5, оскільки ці значення зустрічаються у списку більше одного разу.
7	Напишіть програму, яка ротує елементи списку на N позицій вправо або вліво. Наприклад, при ротації списку [1, 2, 3, 4, 5] на дві позиції вправо результат буде [4, 5, 1, 2, 3].
8	Напишіть програму, яка підраховує кількість входжень кожного елемента в списку і повертає результат у вигляді словника. Наприклад, для списку [1, 2, 2, 3, 3, 3] результатом буде {1: 1, 2: 2, 3: 3}.
9	Напишіть програму, яка приймає від користувача два списки слів, об’єднує їх в один список слів та сортує їх в алфавітному порядку.
10	Напишіть програму, яка приймає словник з оцінками учнів та обчислює середню оцінку кожного учня. Також потрібно вивести прізвище найкращого учня.
11	Напишіть програму, яка приймає кортеж з радіусами кругів та обчислює їх площу та об'єм.
12	Напишіть програму, яка аналізує речення і повертає словник, де ключами є символи речення, а значеннями – кількість їх входжень у речення.
13	Визначте, яка з двох заданих літер у наведеному тексті/реченні трапляється частіше.
14	Задані n точок площини. Складіть програму підрахунку кількості рівнобічних трикутників з вершинами у цих точках.
15	Заданий вектор розмірності n, компоненти якого містять інформацію про студентів деякого ВНЗ. Відомості про кожного студента складаються з зазначення його прізвища, імені, по батькові, статі, віку, курсу. Складіть програму пошуку: а) найбільш поширених чоловічих та жіночих імен; б) прізвищ та ініціалів усіх студентів, вік яких є найбільш поширеним.

Завдання № 2

Створіть список, що містить 100 значень, згенерованих через генератор випадкових цілих чисел у діапазоні від 1 до 9 включно. Необхідно з'ясувати, скільки значень потрапило в діапазон [1, 3], [4, 6], [7, 9].

Завдання № 3

Створіть список, що містить 50 значень, згенерованих через генератор випадкових цілих чисел у діапазоні від 1 до 100.

1. В списку знайдіть максимальний елемент.
2. Вилучіть елемент на третій позиції.
3. Упорядкуйте елементи в порядку збільшення значень списку зі збереженням попереднього списку.
4. Вставте на четверту позицію число 11111.
5. Знайдіть числа, менші за 10, збільшіть їх утричі.

Завдання № 4

Заданий вектор розмірності *n*, компоненти якого містять інформацію про студентів деякого ВНЗ. Відомості про кожного студента складаються з зазначення його прізвища, імені, по батькові, статі, віку, курсу. Складіть програму пошуку:

а) найбільш поширених чоловічих та жіночих імен;

б) прізвищ та ініціалів усіх студентів, вік яких є найбільш поширеним.

Завдання № 5

Заданий список, заповнений випадково нулями і одиницями. Знайдіть найдовшу неперервну послідовність одиниць та індекси початкового і останнього елемента в ній.

3.10 Зміст протоколу роботи

Протокол до виконаної роботи оформлюється у вигляді файлу і містить титульний аркуш із назвою навчального закладу, факультету та випускової кафедри, дисципліни та лабораторної роботи, яка виконується, прізвище та ім'я студента і викладача. Наступна сторінка містить назву роботи та її мету, короткі теоретичні відомості про основні оператори і команди, які використовуються для розв'язання поставлених у роботі задач.

Звіт має містити постановку задачі і програмний код її розв'язання з наданням необхідних коментарів. Наприкінці роботи наводяться висновки щодо отриманих результатів по розв'язанню поставлених задач.

Робота захищається індивідуально.

3.11 Контрольні питання для самоперевірки

1. Як створити новий список, що містить квадрати чисел від 1 до 10?
2. Як перевірити, чи ключ "name" існує в словнику user?
3. Як згенерувати список з п'яти випадкових чисел у діапазоні від 1 до 50?
4. Як перетворити задане число у рядок, де всі знаки розділені комами?
5. Як знайти довжину найбільшого слова в реченні за допомогою Python?
6. Як видалити зі списку всі входження заданого елемента?

7. Що робить метод dict.get() і як він відрізняється від звертання до словника за ключем через dict[key]?
8. Як перевірити, чи всі елементи в списку унікальні?
9. Як можна видалити дублікати зі списку, зберігши порядок елементів?
10. Як об’єднати два словники, де значення з першого словника мають пріоритет перед значеннями з другого словника?
11. Як знайти перетин (спільні елементи) двох списків без використання множин?
12. Як створити кортеж з елементів списку, що відповідають заданій умові (наприклад парні числа зі списку)?
13. Як використовувати генератор списків для створення матриці (списку списків) розміром *m на n* з випадковими числами?
14. Як можна впорядкувати словник за значенням, а не за ключем?
15. Як можна обчислити різницю між двома списками, тобто знайти елементи, що є в першому списку, але відсутні в другому?
16. Як видалити зі списку всі елементи, що відповідають певній умові (наприклад всі негативні числа), не використовуючи додатковий список для зберігання результату?
17. Як знайти всі унікальні комбінації з *n* елементів списку, де порядок елементів не має значення?
18. Як можна перевірити, чи є словник підмножиною іншого словника (тобто чи всі ключі та значення першого словника наявні в другому)?
19. Як зібрати інформацію з декількох списків у словник, де ключі отримані з одного списку, а значення – з інших, за певною логікою?
20. Як знайти елемент, який найчастіше трапляється у списку, з огляду на те, що список може бути дуже великим і містити мільйони елементів?

3.12 Задачі до практичної роботи № 3

1. Задані *n* точок площини. Проведіть коло, на якому лежить найбільша кількість точок цієї множини.
2. Задане речення, що містить послідовність слів. Знайдіть:
   а) найкоротше слово речення, його довжину та позицію у реченні;
   б) найдовше слово речення, його довжину та позицію у реченні;
   в) кількість входжень заданої літери, починаючи з заданої позиції;
   г) слово, що містить найбільшу кількість голосних літер,
3. Задано вектор *C* розміру *n*, компонентами якого є відомості про мешканців деяких міст. Інформація про кожного мешканця містить його прізвище, назву міста мешкання; адресу мешкання у вигляді: вулиця, будинок, квартира. Складіть програму пошуку двох будь-яких жителів зі списку *C*, що мешкають у різних містах за однаковою адресою.
4. Заданий вектор розмірності *n*, компонентами якого є відомості про складання іспитів студентами деякого ВНЗ. Інформація про кожного студента задана в такому вигляді: прізвище, номер групи, оцінка_1, оцінка_2, оцінка_3. Складіть програму пошуку:

а) студентів, що мають заборгованості хоча б з одного з предметів;
б) предмета, який було здано краще за усі інші;
в) студентів, що склали всі іспити на 5 і 4.

5. Задана послідовність чисел у діапазоні [1, N]. Визначте ті числа, які є числами Фібоначчі. Окрім того:
   а) визначте загальну кількість таких чисел;
   б) вкажіть номери чисел у послідовності чисел Фібоначчі, які відсортовані за зростанням.

3.13 Завдання до самостійної роботи

1. Ефективне використання словників у Python. Реалізація вкладених словників і робота з ними. Використання defaultdict і OrderedDict з модуля collections.
2. Алгоритми обходу множин. Оптимізація операцій з множинами (об'єднання, перетин, різниця, симетрична різниця). Використання множин для видалення дублікатів зі списків. Розгляд множин як засіб для ефективного фільтрування даних.
3. Розширені можливості списків у Python. Вивчення list comprehensions для створення складних списків. Робота зі зрізами списків і використання методів списків для обробки даних. Реалізація стеків і черг за допомогою списків.
4. Кортежі та їх застосування. Незмінність кортежів і сценарії їх ефективного використання. Розпакування кортежів і обмін значеннями без тимчасової змінної. Використання кортежів як ключів у словниках.
5. Генерація випадкових чисел і їх застосування. Модуль random і його функції для генерації випадкових даних. Реалізація алгоритмів на основі випадкових вибірок. Використання seed для відтворюваності результатів.
6. Робота з великими даними за допомогою структур даних Python. Оптимізація використання пам'яті та часу обчислень при роботі з великими наборами даних. Використання генераторів для ефективної ітерації по великих даних.
7. Використання lambda-функцій зі списками, словниками та множинами. Розуміння та використання lambda-функцій для анонімних обчислень. Застосування filter, map, і reduce з lambda-функціями.
8. Серіалізація даних з використанням JSON та робота з ними. Зберігання та відновлення складних структур даних за допомогою модуля json. Перетворення словників, списків та інших структур даних в JSON-формат і навпаки.
9. Мультитредінг (багатопотоковість) та асинхронне програмування з використанням структур даних. Використання множин, списків та словників у багатопоточних і асинхронних програмах.
10. Забезпечення безпеки даних при роботі в мультипоточному середовищі.

4 РЕАЛІЗАЦІЯ ФУНКЦІЙ МОВОЮ PYTHON ТА РОБОТА З ФАЙЛАМИ

Мета практичної роботи № 4: ознайомитися з поняттям функцій, їх різновидами та практичною реалізацією мовою Python, організацією роботи з файлами.

ТЕОРЕТИЧНІ ПОЛОЖЕННЯ

4.1 Поняття про застосування функцій в програмуванні

Побудова і застосування функцій в програмуванні мають критичне значення з декількох причин, які разом покращують якість коду, його читабельність, масштабованість та легкість утримання.

Функція в програмуванні – це блок коду, який виконує певну задачу та може бути викликаний з різних місць програми, часто з можливістю приймання параметрів і повернення результату. Функції дозволяють структурувати код, зробити його більш читабельним, уникнути дублювання коду та полегшити процес тестування і відлагодження. Деякі з ключових аспектів важливості застосування функцій полягають в наступному.

Модулярність – функції дозволяють розділити програму на менші частини, кожна з яких виконує окрему задачу. Це робить код більш організованим і модульним, полегшуючи розуміння, редагування та відладку.

Повторне використання коду – замість копіювання та вставки того самого коду в різних місцях, функції дозволяють централізовано визначити логіку, яку потім можна викликати з будь-якої точки програми. Це зменшує кількість коду, спрощує його утримання та зменшує ймовірність помилок.

Абстракція – функції дозволяють абстрагувати складність, приховуючи деталі реалізації від користувача функції. Це дозволяє фокусуватися на тому, що робить функція, а не на тому, як вона це робить.

Параметризація – функції можуть приймати параметри, що робить їх гнучкими та здатними обробляти різні вхідні дані без необхідності зміни їх коду. Це розширює можливості повторного використання коду та адаптації до різних сценаріїв.

Тестування та відладка – функції можна тестувати і відлагоджувати незалежно від решти програми, що спрощує пошук і виправлення помилок. Модульні тести можуть бути написані для кожної функції, забезпечуючи, що кожна частина програми працює правильно.

Легкість розширення – коли програма структурована навколо функцій, додавання нової функціональності стає простішим. Можна легко додати нові функції або змінити існуючі без ризику порушення інших частин програми.

Сприяння командній роботі – функції дозволяють різним розробникам працювати над окремими частинами програми одночасно, знижуючи ризик конфліктів у коді та спрощуючи інтегрування. Функції можна прокласифікувати за різними ознаками та способом їх застосування. Наведемо таку класифікацію:

Класифікація функцій за типом визначення

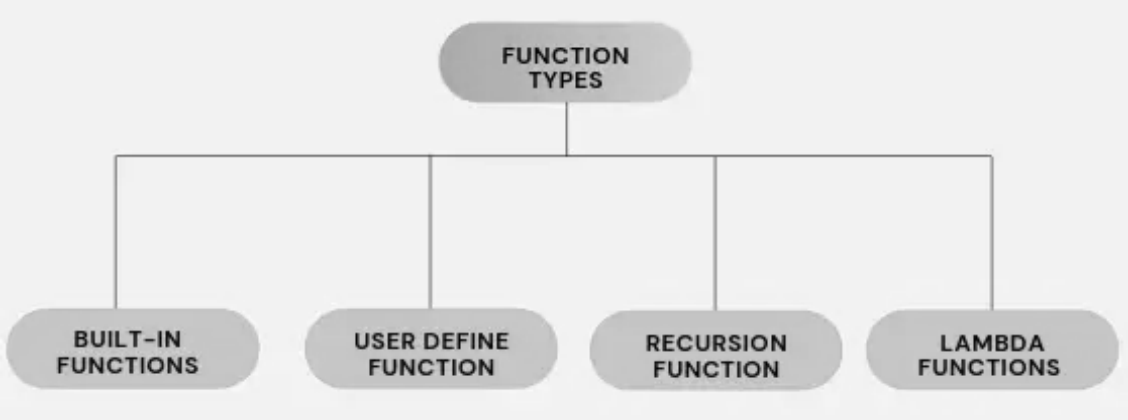


Рисунок 4.1 – Класифікація функцій за типом визначення

Вбудовані функції (Built-in Functions) – попередньо визначені у мові програмування або стандартних бібліотеках. Python має багато вбудованих функцій, які готові до використання, наприклад **print(), len(), max(), input()** тощо. Вони завжди доступні і спрощують виконання базових операцій.

Користувацькі функції (User-Defined Functions) – функції, які створюються розробниками для вирішення конкретних завдань у їх програмах. Це функції, які розробники створюють самостійно для виконання специфічних задач. Створення відбувається за допомогою ключового слова **def**, за яким йде назва функції і круглі дужки.

Рекурсивні функції (Recursive Functions) – це функції, які викликають самих себе. Рекурсія може бути дуже потужним інструментом у програмуванні для розв'язання завдань, які можуть бути поділені на схожі підзадачі меншого масштабу. Рекурсія часто використовується для роботи з даними, що мають вкладену або ієрархічну структуру, наприклад при обході деревоподібних структур або при реалізації алгоритмів сортування та пошуку.

Анонімні функції (Lambda Functions) – функції, які не мають імені і зазвичай використовуються для виконання коротких операцій. Визначаються за допомогою ключового слова **lambda**. Використовуються для створення невеликих функцій, які складаються з одного виразу. Зазвичай використовуються в комбінації з функціями вищого порядку, такими як **map(), filter().**

Класифікація функцій за способом повернення результату

Функції, які повертають значення – після виконання своєї роботи ці функції повертають результат за допомогою оператора **return**. Можуть повертати будь-який тип даних: числа, рядки, списки, словники тощо.

Функції, які не повертають значення – виконують певні дії, але не повертають значення. У деяких мовах програмування (наприклад Python) такі функції технічно повертають спеціальне значення **None**.

Класифікація функцій за способом передачі аргументів

Функції з фіксованою кількістю аргументів – кількість і тип параметрів визначено при визначенні функції.

Функції зі змінною кількістю аргументів – дозволяють передати довільну кількість аргументів. У Python для цього використовуються оператори * для списків і ** для словників.

Використання функцій є фундаментальним аспектом багатьох мов програмування, зокрема Python, дозволяючи розробникам писати більш модульний, читабельний і ефективний код. Розглянемо приклади реалізації функцій та їх застосування.

4.2 Реалізація функцій за типом визначення

Вбудовані функції (Built-in Functions). Розглянемо застосування вбудованих функцій на деяких практичних прикладах.

Приклад 4.1. Застосування вбудованих функцій

```
my_string = "Hello, world!"
print(len(my_string))
```

```
13
```

```
list = [1, 2, 10, 20, 5, 7]
print(max(list))
```

```
20
```

В цьому прикладі проілюстровано застосування декількох вбудованих функцій, зокрема функції **print(),** для виведення довжини стрінгового повідомлення, для чого було використано ще одну вбудовану функцію **len()** з відповідним аргументом. Окрім цього, показано вивід (виведення) максимального значення списку елементів при застосування вбудованої функції **max().**

Нижче наводиться результат застосування вбудованої функції сортування набору даних в прямому (функція **sort()**) та інверсному (функція **sort(reverse=True)**) напрямках.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]
numbers.sort()
print(numbers)  # Виведе: [1, 1, 2, 3, 4, 5, 6, 9]

# Сортування у зворотньому порядку
numbers.sort(reverse=True)
print(numbers)  # Виведе: [9, 6, 5, 4, 3, 2, 1, 1]
```

```
[1, 1, 2, 3, 4, 5, 6, 9]
[9, 6, 5, 4, 3, 2, 1, 1]
```

Мова програмування Python користується великою популярністю насамперед за широкий набір вже готових до використання вбудованих функцій у найрізноманітніших біблотеках (рисунок 1.9). Деякі приклади вбудованих функцій наведені в таблицях 3.1 і 3.2.

Користувацькі функції (User-Defined Functions). Незважаючи на великий набір різноманітних вбудованих функцій, вони не можуть замінити користувацькі функції, які розв'язують певні специфічні задачі або уніфіковують код. Користувацькі функції у Python використовуються для групування коду, що виконує певну задачу, з метою його повторного використання та підвищення читабельності програми.

Шаблон оголошення користувацької функції в Python наведений нижче:

```
def назва_функції(параметр1, параметр2, ..., параметрN):
    """
    Документаційний рядок (docstring), який описує функцію.
    """
    # Тіло функції
    ...
    return значення_повернення
```

- **def** – це ключове слово, що використовується для оголошення функції в Python. Воно сигналізує інтерпретатору, що наступний блок коду буде визначенням функції;
- **назва_функції** – це ідентифікатор функції, який використовується для її виклику. Назви функцій зазвичай описують, що саме робить функція. В Python прийнято використовувати нижнє підкреслення між словами для покращення читабельності (наприклад calculate_average);
- **(параметр1, параметр2, ..., параметр N)** – це список параметрів, які функція приймає. Коли функція описується (декларується) – це **формальні параметри.** Коли функція викликається, ці параметри передаються у функцію. Такі конкретні параметри (цифрові, стрінгові або булеві значення тощо) стають **фактичними** і використовуються в тілі функції для виконання її задач. Параметри є необов'язковими; функція може не приймати жодних параметрів;
- **документаційний рядок (docstring)** – це рядок, що йде безпосередньо за визначенням функції. Вона використовується для опису того, що робить функція, які параметри приймає, що вона повертає та іншої важливої інформації. Docstring оформлюється у вигляді рядка з використанням потрійних подвійних лапок;
- **тіло функції** – це основна частина функції, де знаходиться код, що виконує потрібні дії. Тіло функції може містити різні оператори, виклики інших функцій та ключове слово return;
- **return значення_повернення** – ключове слово return використовується для завершення виконання функції та повернення результату роботи функції. Значення_повернення може бути будь-яким типом даних, включаючи числа, рядки, об'єкти або навіть інші функції. Повернення значення є необов'язковим; якщо return не використовується, функція поверне спеціальне значення None.

Приклад 4.2. Реалізувати у вигляді функції операцію факторіала заданого числа **n**.

Факторіал числа **n**, позначається як **n!**, є добутком усіх натуральних чисел від 1 до n. Наприклад, 5! = 5 * 4 * 3 * 2 * 1 = 120.

```
def factorial(n):
    """
    Обчислює факторіал числа n.
    Параметри:
    n (int): Ціле число, для якого обчислюється факторіал.
    Повертає:
    int: Факторіал числа n.
    Виключення:
    ValueError: Якщо n від'ємне.
    """
    if n < 0:
        raise ValueError("n не може бути від'ємним")
    if n == 0:
        return 1
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Оголошена користувацька функція **factorial(n)** призначена для обчислення факторіала переданого їй цілого числа n. Факторіал числа n, позначений як **n!**, визначається як добуток усіх цілих чисел від 1 до n включно. За визначенням факторіал 0 дорівнює 1.

Параметри функції – **n (int)**: це параметр, який вказує ціле число, для якого буде обчислено факторіал. Функція повертає ціле число, яке є факторіалом введеного числа n.

Виключення – *ValueError:* Це виключення генерується, якщо передане значення n є від'ємним, оскільки факторіал від'ємного числа не визначено.

Реалізація. Функція спочатку перевіряє, чи число n не є від'ємним. Якщо n від'ємне, генерується виключення *ValueError* з повідомленням, що n не може бути від'ємним. Далі, якщо n дорівнює 0, функція негайно повертає 1, оскільки факторіал 0 за визначенням дорівнює 1. У випадку, коли n більше 0, функція ініціалізує змінну result значенням 1, а потім використовує цикл for для обчислення добутку всіх цілих чисел від 1 до n включно. Цей добуток зберігається в змінній result. Після завершення циклу значення змінної result, яке є факторіалом числа n, повертається як результат роботи функції.

Результат роботи функції буде мати вигляд:

```
try:
    print(factorial(5))  # Виведе 120
    print(factorial(-1))  # Викличе ValueError
except ValueError as e:
    print(e)
```

```
120
n не може бути від'ємним
```

Ця функція є прикладом базової математичної операції в Python, демонструє використання циклів, умовних операторів, а також генерацію виключень.

Приклад 4.3. Написати функцію для додавання двох чисел.

```
def add_numbers(a, b):
    """
    Функція для додавання двох чисел.
    Параметри:
    a (float or int): Перше число для додавання.
    b (float or int): Друге число для додавання.
    Повертає:
    float or int: Сума двох чисел.
    """
    return  a + b

# Використання функції
result = add_numbers(10, 5)
print("Сума чисел: ", result)

result = add_numbers(-10, 5)
print("Сума чисел: ", result)
```

```
Сума чисел:  15
Сума чисел:  -5
```

Користувацька функція **add_numbers()**, визначена в цьому прикладі, призначена для додавання двох чисел (**a, b)** та ілюструє базові принципи роботи з функціями в Python.

Параметри:

a *(float or int)* – це перше число, яке буде додано. Функція може приймати як цілі числа (int), так і числа з плаваючою крапкою (*float*);

b *(float or int)* – це друге число, яке буде додано до першого. Як і **a**, цей параметр може бути як цілим числом, так і числом з плаваючою крапкою.

Повертає за командою ***return****:* функція повертає суму двох чисел *(a+b)* типу *float or int***.** Тип повернутого значення залежить від типів вхідних параметрів. Якщо обидва параметри є цілими числами, результат також буде цілим числом. Якщо хоча б один із параметрів є числом з плаваючою крапкою, результат буде числом з плаваючою крапкою.

Реалізація: функція використовує простий оператор додавання (+) для обчислення суми *(a+b)*, а потім повертає результат.

Приклад демонструє використання функції **add_numbers()** з різними наборами вхідних параметрів. У першому випадку до функції передаються числа 10 і 5, і вона повертає їхню суму 15. У другому випадку до функції передаються числа -10 і 5 і функція повертає їхню суму -5.

Незалежний виклик функції **add_numbers()** буде мати вигляд:

```
x = float(input("Введіть перше число: ")) # Вкажіть число 1
y = float(input("Введіть друге число: ")) # Вкажіть число 2
result = add_numbers(x, y)
print("Сума чисел: ", result)
```

```
Введіть перше число: 2
Введіть друге число: -3
Сума чисел:  -1.0
```

Приклад 4.4. Написати функцію для визначення максимального значення з набору даних.

Функція **find_max**, представлена у прикладі, призначена для пошуку максимального числа у переданому списку чисел. Ця функція демонструє базовий приклад алгоритму пошуку максимуму, який є фундаментальним концептом у програмуванні та обробці даних.

Параметри: **numbers (list)** – список чисел, серед яких потрібно знайти максимальне. Цей список може містити як цілі числа (*int*), так і числа з плаваючою крапкою (*float*).

```
def find_max(numbers):
    """
    Повертає максимальне значення зі списку чисел.
    Параметри:
    numbers (list): Список чисел.
    Повертає:
    int or float: Максимальне число у списку.
    """
    # Початково призначаємо максимальним значенням перше число у списку
    max_value = numbers[0]

    # Перебираємо всі числа у списку, починаючи з другого
    for number in numbers[1:]:
        # Якщо поточне число більше за збережене максимальне, оновлюємо максимальне
        if number > max_value:
            max_value = number
    return max_value

# Використання користувацької функції
numbers = [5, 10, 25, 20, 15]
max_value_custom = find_max(numbers)
print("Максимальне значення (користувацька функція): ", max_value_custom)
```

```
Максимальне значення (користувацька функція):  25
```

Повертає: функція повертає максимальне значення зі списку типу *int or float*. Тип повернутого значення відповідатиме типу елементів у вхідному списку. Якщо всі числа у списку цілі, результат буде типу int. Якщо у списку наявні числа з плаваючою крапкою, результат також буде представлено у форматі з плаваючою крапкою.

Реалізація: спочатку функція призначає перше число зі списку як тимчасове максимальне значення (max_value). Далі функція використовує

цикл *for* для перебору всіх чисел у списку, починаючи з другого елемента (тому що перший елемент уже призначений як max_value).

Для кожного числа в циклі проводиться порівняння з поточним максимальним значенням. Якщо поточне число виявляється більшим, то воно стає новим максимальним значенням.

Після завершення циклу функція повертає знайдене максимальне значення.

Використання: у прикладі функція **find_max** використовується для знаходження максимального числа у списку [5, 10, 25, 20, 15]. Результатом її роботи є число 25, яке і є максимальним значенням у цьому списку. Цей результат виводиться на екран.

Незалежний виклик функції **find_max()** буде мати вигляд:

```
numbers_2 = [5, 4, 3, 20, 1]
find_max(numbers_2)
```

```
20
```

Рекурсивні функції (Recursive Functions). Це функції, які викликають самих себе. Рекурсія може бути дуже потужним інструментом у програмуванні для розв'язання завдань, які можуть бути поділені на схожі підзадачі меншого масштабу. Рекурсія часто використовується для роботи з даними, що мають вкладену або ієрархічну структуру, наприклад, при обході деревоподібних структур або при реалізації алгоритмів сортування та пошуку.

Основні компоненти рекурсивної функції:

базовий випадок (Base Case) – умова, при якій рекурсія припиняється, і це запобігає нескінченному виклику функції;

рекурсивний випадок (Recursive Case) – виклик функцією самої себе з новим набором параметрів, які наближають розв'язок до базового випадку.

Приклад 4.5. Реалізувати у вигляді рекурсивної функції операцію *факторіал заданого числа* і порівняти реалізацію з *прикладом 4.2*.

Факторіал числа може бути реалізований за допомогою рекурсії таким чином:

```
def factorial(n):
    # Базовий випадок
    if n == 1:
        return 1
    # Рекурсивний випадок
    else:
        return n * factorial(n-1)

print(factorial(5))  # Виведе 120
```

```
120
```

У цьому прикладі, коли n досягає 1, функція повертає **1**, що є базовим випадком і зупиняє рекурсію. Для більших значень n вона викликає сама себе з параметром **n-1.**

Застереження при використанні рекурсії:

глибина рекурсії – Python має обмеження на максимальну глибину рекурсії, що може призвести до помилки RecursionError, якщо рекурсивний ланцюжок стає занадто довгим. Це обмеження може бути змінено за допомогою функції sys.setrecursionlimit();

ефективність – у деяких випадках рекурсивні рішення можуть бути менш ефективними та використовувати більше пам'яті, ніж їх ітеративні аналоги, особливо якщо в рекурсії відсутня оптимізація хвостової рекурсії (tail call optimization), яка не підтримується в Python.

Приклад 4.6. Обчислити задане число Фібоначчі при застосуванні рекурсивної функції.

```
def fibonacci(n):
    """
    Обчислює n-те число Фібоначчі.

    Параметри:
    n (int): Індекс числа у послідовності Фібоначчі.

    Повертає:
    int: n-те число Фібоначчі.
    """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

# Приклад виклику функції
print(fibonacci(10))  # Виведе 55, що є 10-м числом Фібоначчі
```

```
55
```

Отриманий результат 55 є десятим числом у послідовності Фібоначчі, що збігається з результатом *прикладу 3.5.*

Анонімні функції (Lambda Functions). Анонімні функції, відомі також як **lambda-функції**, у Python – це спосіб створення невеликих функцій, не присвоюючи їм імені. Це особливо корисно, коли потрібна функція для короткого використання, і немає сенсу визначати її за допомогою стандартного синтаксису def.

Lambda-функції можуть приймати будь-яку кількість аргументів, але можуть виконувати тільки один вираз. Результат цього виразу автоматично стає значенням, яке повертає lambda-функція.

Синтаксис Lambda-функції наступний:

```
lambda arguments: expression
```

arguments – це список аргументів, які приймає функція. Аргументи вказуються через кому;

expression – це вираз, що обчислюється і повертається функцією.

Приклад 4.7. Обчислити суму двох чисел при застосуванні **lambda-функції.**

```
# Визначаємо lambda-функцію, яка повертає суму двох чисел
add = lambda x, y: x + y

# Використовуємо цю функцію
print(add(5, 3))  # Виведе: 8
```

```
8
```

Створення **lambda-функції**:

add = lambda x, y: x + y створює нову lambda-функцію, яка приймає два аргументи *(x і y)* і повертає їхню суму *(x + y)*;

lambda – ключове слово, яке вказує на початок визначення lambda-функції;

x, y – параметри, які функція приймає.

x + y – вираз, що виконується і його результат повертається як результат функції.

Присвоєння функції змінній:

Lambda-функція присвоюється змінній *add*. Це означає, що тепер *add* можна використовувати як ім'я для виклику цієї функції.

Виклик lambda-функції:

add(5, 3) викликає lambda-функцію, передаючи їй аргументи 5 та 3. Функція виконує вираз *x + y*, де x дорівнює 5, а y – 3, і повертає їх суму.

Виведення результату:

print(add(5, 3)) виводить результат виклику функції, який є 8.

Цей приклад демонструє як лаконічність, так і потужність lambda-функцій у Python. Вони чудово підходять для визначення невеликих функцій, які можуть бути виражені одним виразом, особливо коли такі функції використовуються одноразово

```
add(5, 3)
```

```
8
```

Приклад 4.8. Написати функцію по виведенню всіх непарних елементів заданого списку при застосуванні *lambda-функції.*

Lambda-функції часто використовуються разом з функціями вищого порядку, які приймають одну або кілька функцій як аргументи, наприклад **map(), filter(), і sorted()**.

Для розв'язку цієї задачі використаємо *lambda-функцію,* яка приймає як аргумент іншу функцію ***filter():***

```python
# Вихідний список
numbers = [1, 2, 3, 4, 5, 6]

# Використовуємо lambda-функцію для фільтрації непарних чисел
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))

print(odd_numbers)  # Виведе: [1, 3, 5]
```

```
[1, 3, 5]
```

Цей код демонструє використання *lambda-функції* разом з функцією **filter()** для відбору непарних чисел зі списку.

Вихідний список numbers: numbers = [1, 2, 3, 4, 5, 6] визначає список з послідовністю чисел від 1 до 6.

Фільтрація з використанням filter() і lambda-функції:

filter(lambda x: x % 2 != 0, numbers) застосовує lambda-функцію до кожного елемента списку numbers.

Lambda-функція *lambda x: x % 2 != 0* перевіряє, чи елемент x є непарним числом (x % 2 != 0). Якщо умова виконується, елемент проходить через фільтр.

filter() повертає ітератор, тому виклик **list()** перетворює результати, що пройшли фільтрацію, назад у список.

Присвоєння результату змінній odd_numbers: результат виклику list(filter(...)) присвоюється змінній odd_numbers. Цей новий список містить лише ті елементи з вихідного списку numbers, які є непарними.

Виведення результату: print(odd_numbers) виводить фінальний список непарних чисел: [1, 3, 5].

Цей приклад показує, як ефективно використовувати lambda-функції для створення невеликих анонімних функцій, які можуть бути передані як аргументи в інші функції, такі як filter(). Використання filter() у поєднанні з lambda-функціями дозволяє писати виразний і лаконічний код для виконання операцій фільтрації на колекціях.

Приклад 4.9. Написати функцію по сортуванню кортежу за значенням другого елемента при застосуванні *lambda-функції*.

```python
# Список кортежів
pairs = [(1, 'one'), (3, 'three'), (2, 'two'), (4, 'four')]

# Сортуємо за другим елементом у кортежі
sorted_pairs = sorted(pairs, key=lambda pair: pair[1])

print(sorted_pairs)  # Виведе: [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

```
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Цей приклад демонструє використання lambda-функції у поєднанні з функцією sorted() для сортування списку кортежів за певним критерієм.

Список кортежів pairs:

pairs = [(1, 'one'), (3, 'three'), (2, 'two'), (4, 'four')] визначає список, кожний елемент якого є кортежем. Перший елемент у кортежі – це число, а другий – його текстове представлення англійською мовою.

Сортування за другим елементом у кортежі: використання *sorted(pairs, key=lambda pair: pair[1])* дозволяє відсортувати список pairs за значенням другого елемента кожного кортежа *(pair[1])*.

Lambda-функція *lambda pair: pair[1]* вказує sorted(), що критерієм сортування є другий елемент кортежа.

sorted() повертає новий список відсортованих елементів, не змінюючи вихідний список pairs.

Присвоєння результату змінній sorted_pairs: відсортований список присвоюється змінній *sorted_pairs*.

Відповідно до сортування за другим елементом список виглядає так: *[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]*, оскільки слова "four", "one", "three", "two" відсортовані в алфавітному порядку.

Виведення результату: print(sorted_pairs) виводить відсортований список кортежів. Зауважимо, що елементи розташовані за алфавітним порядком других компонентів кортежів: "four", "one", "three", "two".

Приклад 4.10. Написати функцію по перетворенню чисел списку у їх квадрати при застосуванні *lambda-функції.*

Для роботи зі списками і застосування функції до кожного з елементів такого списку використовується функція **map().** Така функція в Python призначена для застосування функції до кожного елемента ітерованого об’єкта (наприклад списку) і повертає ітератор з результатами. Загальний синтаксис виглядає таким чином:

```
map(function, iterable, ...)
```

function – функція, яку потрібно застосувати до кожного елемента ітерованого об’єкта;

iterable – ітерований об’єкт, елементи якого будуть оброблені заданою функцією (можуть бути вказані декілька ітерованих об’єктів).

Ось як буде виглядати рішення цієї задачі:

```
# Визначаємо список чисел
numbers = [1, 2, 3, 4, 5]

# Використовуємо map() для обчислення квадратів кожного числа
squares = map(lambda x: x ** 2, numbers)

# Перетворюємо результат з ітератора в список
squares_list = list(squares)

print(squares_list)  # Виведе: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

У цьому прикладі, *lambda x: x ** 2* – це функція, яка обчислює квадрат числа, а *numbers* – це список чисел, до якого буде застосована ця функція. **map()** застосовує функцію до кожного елемента *numbers* і повертає ітератор. За допомогою *list(squares)* ми перетворюємо ітератор у список для виведення результату.

Приклад 4.11. Написати функцію для додавання елементів двох списків при застосуванні *lambda-функції*.

```
# Визначаємо два списки чисел
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

# Використовуємо map() для додавання відповідних елементів
result = map(lambda x, y: x + y, numbers1, numbers2)

# Перетворюємо результат з ітератора в список
result_list = list(result)

print(result_list)  # Виведе: [5, 7, 9]
```

```
[5, 7, 9]
```

Тут **map()** використовується для додавання відповідних елементів з двох списків, повертаючи суму кожної пари елементів. Цей приклад показує, як **map()** може застосовуватися не тільки до одного, але й до декількох ітерованих об’єктів одночасно.

Для порівняння нижче наводяться ще два приклади реалізації цієї задачі. Розробник програмного коду сам обирає зручний для себе спосіб розв’язку поставленої задачі, але разом з тим повинен пам’ятати про лаконічність і зрозумілість коду для читання і, що досить важливо, його швидкодію, яка буде залежати від застосування відповідних функцій:

- *приклад реалізації задачі при застосуванні функції* ***append()***

```
# Визначаємо два списки чисел
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

# Створюємо пустий список для зберігання результатів
result_list = []

# Проходимося по елементах першого списку
for i in range(len(numbers1)):
    # Додаємо відповідні елементи з обох списків і
    # додаємо результат у новий список
    result_list.append(numbers1[i] + numbers2[i])

print(result_list)  # Виведе: [5, 7, 9]
```

```
[5, 7, 9]
```

- *приклад реалізації задачі при додаванні елементів*

```
# Визначаємо два списки чисел
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

# Створюємо список потрібного розміру з нулів
result_list = [0] * len(numbers1)

# Проходимося по елементах списків
for i in range(len(numbers1)):
    # Додаємо відповідні елементи з обох списків і
    # зберігаємо результат у новий список
    result_list[i] = numbers1[i] + numbers2[i]

print(result_list)  # Виведе: [5, 7, 9]
```

```
[5, 7, 9]
```

Для великих обсягів даних найбільш ефективним з погляду швидкодії, як правило, буде варіант з використанням **map().** Це пояснюється тим, що Python оптимізує виклики функцій, виконаних через map(), на рівні C, що робить цей метод дуже швидким для ітерації по великих даних.

Функція **map()** повертає ітератор, що означає, що елементи обробляються послідовно і не потребують алокації пам'яті для всього списку результатів відразу. Це робить **map()** більш ефективним з погляду використання пам'яті при роботі з великими даними.

Цикл з append(): – цей метод ефективний і читабельний, але кожний виклик append() потребує додаткового виклику методу, що може сповільнити виконання при роботі з дуже великими списками.

Цикл зі заздалегідь визначеним розміром списку – хоча цей метод усуває необхідність виклику **append()** і може бути швидшим за варіант з **append()** завдяки відсутності необхідності динамічно збільшувати розмір списку, він все одно може бути повільнішим за **map(),** оскільки не оптимізований на такому низькому рівні.

4.3 Реалізація функцій за способом повернення результату

Функції в Python можуть повертати результати різними способами.

Повернення одного значення – функція може повертати одне значення за допомогою оператора **return:**

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

print(square(4))  # Виведе: 16
```

```
16
```

Повернення декількох значень – функція може повертати декілька значень, які будуть упаковані в кортеж:

```python
def arithmetic_operations(a, b):
    return a+b, a-b, a*b, a/b

sum, difference, product, division = arithmetic_operations(10, 5)
print(f"Сума: {sum}, Різниця: {difference},\
      Добуток: {product}, Ділення: {division}")
```

```
Сума: 15, Різниця: 5,      Добуток: 50, Ділення: 2.0
```

Повернення булевого значення – функції часто використовуються для перевірки умов і повертають True або False:

```python
def is_even(number):
    return number % 2 == 0

print(is_even(4))  # Виведе: True
print(is_even(5))  # Виведе: False
```

```
True
False
```

Повернення списку – функція може повертати список значень:

```python
def get_even_numbers(numbers):
    return [num for num in numbers if num % 2 == 0]

print(get_even_numbers([1, 2, 3, 4, 5, 6]))  # Виведе: [2, 4, 6]
```

```
[2, 4, 6]
```

Повернення словника – функція може повертати словник, що є дуже зручним для повернення складних даних:

```python
def create_user(name, age):
    return {"name": name, "age": age}

user = create_user("Анна", 30)
print(user)  # Виведе: {'name': 'Анна', 'age': 30}
```

```
{'name': 'Анна', 'age': 30}
```

Повернення None – якщо функція не має явного оператора return або якщо return викликається без значення, функція повертає None.

```python
def print_message(message):
    print(message)

result = print_message("Привіт!")
print(result)  # Виведе: None
```

```
Привіт!
None
```

4.4 Реалізація функцій за способом передачі аргументів

Функції в Python можуть приймати аргументи різними способами, що надає гнучкість при передачі даних у функції. Наведемо основні способи передачі аргументів.

Позиційні аргументи – аргументи, значення яких передаються у функцію відповідно до їхнього положення. Зміна положення принципово впливає на результат:

```
def multiply(a, b, c):
    return (a * b)/c

result_1 = multiply(3, 4, 2)  # a=3, b=4, c=2
print(result_1)  # Виведе: 6.0

result_2 = multiply(2, 4, 3)  # a=2, b=4, c=3
print(f'{result_2:.4f}')  # Виведе: 2.(6)
```

```
6.0
2.6667
```

Ключові аргументи (Named or Keyword Arguments) – аргументи передаються у функцію за допомогою їхнього імені, що робить код більш читабельним і дозволяє задавати аргументи в довільному порядку:

```
def greeting(name, age):
    print(f"Привіт, мене звати {name}, мені {age} років.")

greeting(age=25, name="Олексій")  # Порядок аргументів змінено
```

```
Привіт, мене звати Олексій, мені 25 років.
```

Значення аргументів за замовчуванням – функції можуть мати аргументи із попередньо визначеними значеннями:

```
def greeting(name, greeting="Привіт"):
    print(f"{greeting}, {name}!")

greeting("Анна")  # Використовується значення за замовчуванням для greeting
greeting("Олексій", "Добрий день")  # Перевизначення значення за замовчуванням
```

```
Привіт, Анна!
Добрий день, Олексій!
```

Передача довільної кількості позиційних аргументів – використовується ***args,** щоб функція могла приймати довільну кількість позиційних аргументів:

```
def sum_numbers(*args):
    return sum(args)

print(sum_numbers(1, 2, 3, 4))  # Виведе: 10
```

```
10
```

Передача довільної кількості ключових аргументів – використовується ****kwargs** для прийняття довільної кількості ключових аргументів у вигляді словника:

```
def user_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

user_info(name="Віктор", age=30, city="Черкаси")
```

```
name: Віктор
age: 30
city: Черкаси
```

Приклад 4.12. За допомогою розкладу в ряд Тейлора функції e^x

$$y(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots$$

обчислити її значення з точністю $\varepsilon > 0$ для заданого значення x.

```
def exp_taylor(x, epsilon):
    """
    Обчислює e^x з використанням ряду Тейлора з точністю до epsilon.

    Параметри:
    x (float): значення степеня для e^x.
    epsilon (float): точність обчислення.

    Повертає:
    float: наближене значення e^x.
    """
    term = 1  # Перший член ряду, x^0/0! = 1
    sum = term  # Початкова сума ряду
    n = 1  # Індекс наступного члена ряду

    while abs(term) > epsilon:
        term *= x / n  # Обчислення наступного члена ряду
        sum += term  # Додавання цього члена до суми
        n += 1  # Збільшення індексу на 1

    return sum

# Приклад використання функції
x = 1  # e^1 = e
epsilon = 0.0001
print(f"e^{x} ≈ {exp_taylor(x, epsilon):.3f}")
```

```
e^1 ≈ 2.718
```

4.5 Локальні та глобальні змінні

Локальні та глобальні змінні – це два ключові поняття у програмуванні, які відносяться до області видимості змінних.

Локальні змінні визначаються всередині функції або блоку коду і *доступні тільки в межах цієї функції або блоку*. Їх область видимості обмежена місцем їх визначення, тому змінні з однаковими іменами в різних функціях не конфліктують одна з одною:

```
def function():
    local_var = 5  # Локальна змінна,
#     доступна тільки всередині цієї функції
    print(local_var)

function()
# print(local_var)
```

```
5
```

Разом з тим спроба виклику змінної поза межами її доступності викличе помилку

```
# Це викличе помилку, оскільки local_var тут не визначена
print(local_var)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[9], line 2
      1 # Це викличе помилку, оскільки local_var тут не визначена
----> 2 print(local_var)

NameError: name 'local_var' is not defined
```

Глобальні змінні визначаються поза будь-якими функціями і доступні з будь-якої точки програми, включаючи всі функції та модулі. Використання глобальних змінних може спростити обмін даними між різними частинами програми, але також може призвести до зменшення читабельності та підтримуваності коду, а також до помилок, пов’язаних з неочікуваною зміною їх значення.

Приклад 4.13. Визначення та модифікація глобальної змінної.

У цьому прикладі ми визначили глобальну змінну *global_var* зі значенням 10. У функції **modify_global_var** ми використовуємо *global* для вказівки, що *global_var* всередині функції відноситься до глобальної змінної, а не до локальної змінної з тим же ім’ям.

Ми змінюємо значення *global_var* всередині функції, збільшуючи його на 5.

Після виклику функції **modify_global_var** ми бачимо, що значення global_var змінилося на глобальному рівні.

```
global_var = 10  # Глобальна змінна, доступна з будь-якого місця програми

def modify_global_var():
    global global_var  # Вказуємо, що global_var всередині функції
#                        - це та сама змінна, що і зовні
    global_var += 5  # Змінюємо значення глобальної змінної
    print("Значення всередині функції:", global_var)

modify_global_var()  # Змінюємо значення глобальної змінної та друкуємо його

# Перевіряємо, що значення змінної змінилося глобально
print("Значення поза функцією:", global_var)
```

```
Значення всередині функції: 15
Значення поза функцією: 15
```

Цей приклад показує, як можна змінювати глобальні змінні всередині функцій, а також важливість явного вказування працювати з глобальною змінною за допомогою ключового слова global, щоб уникнути непорозумінь та помилок.

Наступний приклад ілюструє існування змінних з однаковим ім’ям незалежно одна від одної в різних областях видимості, і зміни в одній змінній не впливають на змінну в іншій області видимості:

```
x = 5  # Глобальна змінна 'x'

def func():
    x = 10  # Локальна змінна 'x',
#     яка є в області видимості тільки всередині функції
    print("Локальне значення 'x':", x)

func()
print("Глобальне значення 'x':", x)
```

```
Локальне значення 'x': 10
Глобальне значення 'x': 5
```

Ми визначили глобальну змінну x зі значенням 5.

У функції func, ми визначили локальну змінну x зі значенням 10. Ця змінна доступна тільки всередині func.

Коли ми викликаємо func(), вона виводить локальне значення x (10), але це не впливає на глобальну змінну x.

Після виклику func(), коли ми друкуємо глобальне значення x, воно все ще дорівнює 5.

Нерозуміння різниці між локальними і глобальними змінними може призвести до неправильної роботи всього коду. Така ситуація може трапитися, коли одна змінна з однаковим ім'ям використовується і в глобальному, і в локальному контексті, особливо коли намагаємося змінити глобальну змінну всередині функції без явного оголошення про це:

```
: x = 5  # Глобальна змінна 'x'

def func():
    x = x + 10  # Спроба змінити глобальну змінну 'x' всередині функції
    print("Значення 'x' всередині func:", x)

func()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[15], line 7
```

Приклад 4.14. Ілюстрація різної видимості для локальних і глобальних змінних

```
y = 1  # y - глобальна змінна зі значенням 1

def func():
    y = 111  # Це локальна змінна y, яка існує тільки всередині func()
    print(y)  # Виводимо локальну змінну y, що має значення 111

func()  # Виклик функції, що виводить локальну змінну y = 111
print(y)  # Виводимо глобальну змінну y, яка все ще має значення 1
```

```
111
1
```

Коли ми викликаємо **func(),** функція створює нову локальну змінну **y** зі значенням 111 і відразу ж виводить її. Ця локальна змінна **y** не впливає на глобальну змінну **y**, яка була оголошена за межами функції зі значенням 1. Після виходу з функції локальна змінна **y** перестає існувати, і коли ми виводимо **y** поза функцією, ми виводимо глобальну змінну **y** з її початковим значенням 1.

Приклад 4.15. Створити таблицю залежності функції sin(x) від її аргументу x, який рівномірно змінюється на проміжку x=[0-2pi] і складається з 10 значень.

```
import math
def tabl(n, f): # n - значення аргумента,
#                 f - значення функції
    dict = {} # Словник dict - таблиця значень функції
    for i in n: # Проходимо по всіх i
        dict[i] = f(i) # Додаємо пару (i, f(i)) у словник dict
    return dict # Повертаємо словник у вигляді таблиці
```

Цей код демонструє спосіб створення таблиці значень тригонометричної функції синус за допомогою словника в Python. Функція **tabl** приймає два параметри: список значень аргументів **n** і функцію **f**, для якої потрібно обчислити значення. В цьому випадку використовується стандартна функція **math.sin** з модуля **math**.

```
# Створюємо набір аргументів функції
# як ділення проміжку [0, 2*pi] на 10 частин
x = [i * math.pi / 10 for i in range(11)]
# Будуємо таблицю значень функції sin
Table = tabl(x, math.sin)
# Виводимо результат на екран у вигляді sin(0) = 0
for i in x:
    print(f"sin({i:.3f}) = {Table[i]:.3f}")
```

Спочатку створюється список x, який містить 11 значень аргументів, рівномірно розподілених у проміжку від 0 до 2π (включно). Це робиться за допомогою генератора списку, який використовує формулу i * math.pi / 10, де i змінюється від 0 до 10.

Функція **tabl** створює порожній словник **dict**, де ключами будуть значення аргументів (елементи списку x), а значеннями – результати виклику функції **f(i),** тобто значення синуса для кожного аргументу.

Після того як словник заповнений парами ключ-значення, він повертається як результат роботи функції **tabl**.

В кінці програми за допомогою циклу **for** виводиться на екран таблиця значень функції синус для кожного аргументу зі списку **x**. Вивід форматується так, що і аргумент, і значення синуса відображаються з трьома знаками після коми:

```
sin(0.000) = 0.000
sin(0.314) = 0.309
sin(0.628) = 0.588
sin(0.942) = 0.809
sin(1.257) = 0.951
sin(1.571) = 1.000
sin(1.885) = 0.951
sin(2.199) = 0.809
sin(2.513) = 0.588
sin(2.827) = 0.309
sin(3.142) = 0.000
```

Цей підхід дозволяє ефективно створити та використати таблицю значень будь-якої математичної функції, переданої як аргумент, для набору значень. Він демонструє гнучкість Python у роботі з функціями вищого порядку.

4.6 Функції-генератори

Генератори в Python – це спеціальний тип ітераторів, реалізований за допомогою функцій, які використовують ключове слово **yield**. Генератори дозволяють функціям повертати значення одне за одним, замість одного повернення всієї колекції значень одразу. Це може бути корисно для оптимізації використання пам'яті та збільшення ефективності виконання програм, особливо при роботі з великими обсягами даних.

Приклад 4.16. Генератор, який створює послідовність чисел

```
def simple_number_generator(max):
    number = 1
    while number <= max:
        yield number
        number += 1

# Використання генератора
for number in simple_number_generator(5):
    print(number, end = " ")
```

```
1 2 3 4 5
```

Цей код виведе числа від 1 до 5. Кожного разу, коли yield передає число, виконання функції призупиняється, зберігаючи свій поточний стан.

Приклад 4.17. Генератор, який створює нескінченну послідовність Фібоначчі

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Використання генератора
fib_gen = fibonacci()
for _ in range(11):  # Виведемо перші 10 чисел Фібоначчі
    print(next(fib_gen), end = " ")
```

```
0 1 1 2 3 5 8 13 21 34 55
```

Цей приклад демонструє створення нескінченної послідовності чисел Фібоначчі. За допомогою **next()** можна отримати наступне значення з генератора.

Приклад 4.18. Генератор, що фільтрує елементи і повертає тільки парні елементи

```
def even_numbers(numbers):
    for number in numbers:
        if number % 2 == 0:
            yield number

# Використання генератора
for number in even_numbers([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]):
    print(number, end = " ")
```

```
2 4 6 8 10
```

Ця функція **even_numbers(numbers)** приймає на вхід список чисел **numbers**. Вона ітерує по кожному елементу цього списку за допомогою циклу **for**.

Для кожного числа **number** у списку **numbers** виконується перевірка: якщо залишок від ділення number на 2 дорівнює 0,

(тобто number % 2 == 0),

це означає, що число є парним.

Коли умова виконується (число парне), то використовується оператор **yield**, який «повертає» це число з функції. Важливо зазначити, що **yield** тимчасово призупиняє виконання функції, зберігаючи її стан до наступного виклику. Це означає, що цикл не завершується, а виконання функції буде продовжено з місця, де було зупинено, при наступному виклику **next()** або ітерації по генератору.

У фрагменті коду «*Використання генератора*» ми використовуємо функцію-генератор **even_numbers** для ітерації по списку чисел від 1 до 10. Команда **for** автоматично викликає генератор для отримання наступного парного числа зі списку, починаючи з першого елемента. Кожного разу, коли генератор досягає **yield**, він передає число до циклу **for**, який використовується у зовнішньому коді для виведення числа. Після кожного виведення виконання повертається до генератора, щоб продовжити пошук наступного парного числа, поки не будуть перевірені всі елементи вхідного списку.

Ця модель дозволяє ефективно працювати з великими даними або виконувати обчислення, які вимагають частинного виведення результатів, не завантажуючи всю послідовність чисел в пам’ять одразу.

4.7 Декоратори для функцій

Декоратори в Python – це функції, які модифікують функціонал інших функцій. Вони дозволяють обгортати іншу функцію для розширення її поведінки без безпосереднього змінення її коду. Це потужний інструмент, який дозволяє додавати нові можливості до існуючих функцій або методів класів.

Декоратор – це, по суті, функція, яка приймає іншу функцію як аргумент і повертає нову функцію, додаючи до неї додатковий функціонал. Декоратор, інакше кажучи, – це функція, яка дозволяє змінити поведінку функції (отриманої як аргумент), не змінюючи при цьому коду самої функції.

Призначення декораторів:

логування – декоратори можуть використовуватися для логування аргументів функції та результатів її роботи;

перевірка доступу – перевіряти, чи має користувач права доступу для виконання певної операції;

кешування – зберігати результати функцій, щоб при повторному виклику з тими самими аргументами повернути результат з кеша.

вимірювання часу виконання – обчислювати, скільки часу займає виконання функції;

модифікація аргументів або результату – декоратори можуть змінювати аргументи, передані у функцію, або модифікувати результат перед його поверненням та ін.

Застосування декоратора. Щоб застосувати декоратор до функції, використовують синтаксис **@** перед оголошенням функції, яку потрібно декорувати.

Розглянемо приклад проміжного варіанта обгортання функції без використання синтаксису декоратора **@**, а потім перепишемо його з використанням декоратора.

Проміжний варіант без декоратора @.

Припустимо, ми маємо функцію **hello**, яка виводить привітання:

```
def hello():
    print("Привіт, світ!")
```

Тепер ми хочемо додати функціональність логування, яка буде виводити повідомлення перед і після виклику функції **hello**. Ми можемо зробити це шляхом створення обгортки, яка викликає функцію **hello** і виводить логи:

```
def log_hello():
    print("Викликається функція hello")
    hello()
    print("Функція hello викликана")
```

Результат запуску двох функцій буде мати вигляд:

```
log_hello()
```

```
Викликається функція hello
Привіт, світ!
Функція hello викликана
```

Цей підхід працює, але він не дуже зручний і породжує дублювання коду. Тепер ми можемо викликати **log_hello()** замість **hello()**, але це потребує модифікації кожного виклику *hello* в нашому коді.

Застосування декоратора @

Тепер давайте застосуємо декоратор, щоб спростити цю задачу.

Приклад 4.19. Реалізація декоратора

```
def log_decorator(func):
    def wrapper():
        print(f"Викликається функція {func.__name__}")
        func()
        print(f"Функція {func.__name__} викликана")
    return wrapper

@log_decorator
def hello():
    print("Привіт, світ!")

hello()
```

```
Викликається функція hello
Привіт, світ!
Функція hello викликана
```

У цьому прикладі **@log_decorator** перед функцією **hello** вказує Python, щоб вона автоматично обгорталася за допомогою функції **log_decorator**. Після цього ми можемо викликати hello() як звичайно, і вона автоматично буде обгорнута в функцію wrapper, яка забезпечує логування перед і після виклику функції hello.

Застосування декораторів у цьому випадку значно спрощує код і робить його більш зрозумілим, адже функціональність логування відділена від основної логіки програми.

Приклад 4.20. Реалізація декоратора для вимірювання часу виконання функції на прикладі знаходження факторіала числа.

Для вимірювання часу виконання функції можна створити декоратор, який використовує модуль **time** для фіксування часу перед викликом функції та після завершення її виконання. Різниця між цими двома показниками дасть час виконання функції.

```
import time

def timeit(func):
    """Декоратор для вимірювання часу виконання функції в мікросекундах."""
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()

        # Переводимо секунди в мікросекунди
        elapsed_time_us = (end_time - start_time) * 1_000_000
        print(f"Функція {func.__name__} \
        виконувалася {elapsed_time_us:.3f} мкс.")

        return result
    return wrapper

@timeit
def factorial(n):
    """Функція для обчислення факторіалу числа n."""
    if n == 0:
        return 1
    return n * factorial(n-1)

# Викликаємо функцію, вимірюючи час її виконання
print(factorial(10))
```

Результат роботи програми:

```
Функція factorial виконувалася 0.000 мкс.
Функція factorial виконувалася 998.974 мкс.
Функція factorial виконувалася 998.974 мкс.
Функція factorial виконувалася 998.974 мкс.
Функція factorial виконувалася 998.974 мкс.
Функція factorial виконувалася 998.974 мкс.
Функція factorial виконувалася 998.974 мкс.
Функція factorial виконувалася 998.974 мкс.
Функція factorial виконувалася 998.974 мкс.
Функція factorial виконувалася 998.974 мкс.
Функція factorial виконувалася 998.974 мкс.
3628800
```

Декоратор **timeit** бере будь-яку функцію, виконує її, фіксуючи час перед її викликом та після завершення. Він виводить різницю цих двох часових показників, яка є часом виконання функції.

Функція **factorial** є рекурсивною функцією для обчислення факторіала числа n. Після декорування функції **factorial** за допомогою **@timeit** кожний виклик factorial буде автоматично вимірювати і виводити час її виконання.

Таким чином, ми отримаємо не тільки результат виконання функції, але й інформацію про те, скільки часу було витрачено на її виконання.

Приклад 4.21. Реалізувати кешування результатів обчислення функції.

Кеш – це технологія, що використовується для тимчасового зберігання копій даних у швидкодіючому сховищі з метою швидкого доступу до них при повторних запитах. Кеш може бути реалізований на різних рівнях системи – від апаратного забезпечення (наприклад кеш процесора) до програмного забезпечення (наприклад кеш веб-сторінок у веб-браузері).

Основне призначення кешування – зменшення часу доступу до даних та зниження навантаження на основне джерело даних. Це досягається шляхом тимчасового зберігання копій часто запитуваних або недавно доступних даних у швидкодіючому сховищі. При повторних запитах дані можуть бути швидко отримані з кеша, що значно скорочує час їх обробки порівняно з завантаженням з основного джерела.

Приклади застосування кеша:

веб-кешування – веб-браузери кешують веб-сторінки, зображення та інші медіафайли, які користувач вже відвідав, щоб при повторному відвідуванні цих ресурсів зменшити час завантаження;

кешування запитів до баз даних – прикладні програми можуть кешувати результати часто виконуваних запитів до бази даних, щоб зменшити кількість звернень до бази та підвищити швидкість відповіді на запити користувачів;

кешування на рівні процесора – процесори використовують кеш для тимчасового зберігання інструкцій і даних, які використовуються найчастіше, з метою зменшення часу доступу до пам’яті.

Продемонструємо реалізацію кешування на програмному рівні при застосуванні декоратора. Для функцій, виконання яких є часозатратним, декоратори можуть використовуватися для кешування результатів. Це дозволяє зберігати результат виклику функції і повторно використовувати його замість повторного виконання функції.

```python
def cache(func):
    memo = {}
    def wrapper(*args):
        if args in memo:
            return memo[args]
        result = func(*args)
        memo[args] = result
        return result
    return wrapper

@cache
def long_running_func(num):
    time.sleep(2)  # Припустимо, що ця операція триває довго
    return num * num

# Перший виклик займе близько 2 секунд,
# але повторні виклики з тим самим аргументом будуть миттєвими.
```

```python
long_running_func(2)
```

```
4
```

В цьому прикладі перший раз виклик простої функції **long_running_func(num)** перемноження двох чисел **num** займе близько двох секунд, як це записано в програмі для імітації тривалого обчислення (функція time.sleep(2) примусово призупинить виконання коду на 2 с для імітації затримки). Однак повторний виклик цієї самої функції з таким же параметром дасть миттєвий результат.

4.8 Робота з файлами

Файл (англ. file) – це об’єкт, який зберігає в собі дані чи програмне забезпечення, розташований на конкретній ділянці носія даних з унікальним іменем. Це основна одиниця зберігання інформації (media unit), яка надає доступ до специфічних ресурсів обчислювальної системи, характеризуючись сталим ім’ям (яке є унікальною послідовністю символів, числом або іншим ідентифікатором, що однозначно вказує на файл) та визначеною логічною структурою, з можливістю виконання над ним операцій читання та запису. Файл являє собою іменований сегмент даних, що міститься на носії інформації.

У Python робота з файлами не обмежується якимось конкретним типом файлів, адже мова здатна читати, записувати та обробляти різноманітні формати. Проте можна виділити декілька основних різновидів файлів, з якими часто працюють розробники на Python.

Текстові файли. Текстові файли містять дані у вигляді тексту, який можна читати і редагувати за допомогою стандартних текстових редакторів. Python дозволяє легко читати з текстових файлів та записувати в них, використовуючи кодування, як-от UTF-8:

.txt – звичайний текстовий файл без форматування;

.csv – значення, розділені комами, часто використовуються для таблиць даних;

.json – текстовий формат обміну даними, що використовується для серіалізації структурованих даних.

Бінарні файли. Бінарні файли містять дані у форматі, призначеному для обробки програмним забезпеченням, а не для читання людиною. Python може працювати з бінарними файлами, такими як зображення або виконувані файли, але для цього може знадобитися спеціалізована бібліотека або модуль:

.bin – загальний формат бінарних файлів;

.dat – зазвичай використовується для файлів, що містять конкретні дані програми або системи;

.png, .jpg, .gif та інші формати зображень.

Файли даних. Ці файли містять дані у структурованому форматі і вимагають спеціалізованих програм або модулів для їх читання та обробки:

.xlsx – файли електронних таблиць Excel, для роботи з якими в Python можна використовувати бібліотеку openpyxl або pandas;

.xml – файли, що містять марковані дані. Python може читати та модифікувати XML за допомогою модулів xml.etree.ElementTree або lxml;

.html – файли гіпертекстової розмітки, для обробки яких можна використовувати бібліотеки, як-от BeautifulSoup.

Скриптові та виконувані файли. Це файли, що містять код на Python або інших мовах програмування, які можуть бути виконані на комп’ютері:

.py – стандартний файл скрипта Python;

.ipynb – файли Jupyter Notebook, які містять як виконуваний код, так і елементи форматування Markdown.

Це лише декілька прикладів різновидів файлів, з якими Python може працювати. Завдяки великій кількості бібліотек та модулів можливості Python у роботі з файлами практично не обмежені.

Обробка файлів у Python дозволяє зчитувати з файлів, записувати в них дані, а також виконувати інші операції з файловою системою, наприклад перейменування або видалення файлів. Ось основні концепції і приклади роботи з файлами в Python.

Відкриття файлу. Щоб працювати з файлом, спершу потрібно його відкрити за допомогою функції **open()**. Ця функція повертає об’єкт файлу, який далі використовується для читання або запису.

```
# Відкриття файлу для читання ('r' - read)
file = open('example.txt', 'r')
```

У функції Open() багато параметрів:

```
open(file, mode='r', buffering=-1,
     encoding=None, errors=None,
     newline=None, closefd=True, opener=None)
```

Нам поки важливі три аргументи: перший – це ім’я файлу. Шлях до файлу може бути відносним або абсолютним. Другий аргумент – це режим, в якому буде відкриватися файл:

Режим	**Призначення**
'r'	відкриття на читання (є значенням за замовчуванням)
'w'	відкриття на запис, вміст файлу видаляється; якщо файлу не існує, створюється новий
'x'	ексклюзивне створення (збуджується виключення FileExistsError, якщо файл вже існує)
'a'	відкриття на дозапис, інформація додається в кінець файлу
'b'	відкриття в двійковому режимі
't'	відкриття в текстовому режимі (є значенням за замовчуванням)
'+'	відкриття на читання і запис

Приклади використання декількох режимів одночасно

Режим	**Призначення**
r+b	Відкриває бінарний файл на читання і запис. Покажчик стоїть на початку файлу.
w+b	Відкриває бінарний файл на читання і запис, очищує до 0 байтів. Покажчик стоїть на початку файлу.
wb	Відкриває файл для запису в двійковому форматі. Покажчик стоїть на початку файлу. Створює файл з ім’ям ім'я_файлу, якщо такого не існує.
a+	Відкриває файл для додавання і читання. Покажчик стоїть в кінці файлу. Створює файл з ім'я_файлу, якщо такого не існує.

Читання з файлу. Після відкриття файлу можна зчитати його вміст різними способами:

read(size) – читає дані з файлу розміром в size байтів. Якщо size не вказано, читає весь вміст файлу;

readline() – читає один рядок з файлу;

readlines() – читає весь файл і повертає список його рядків.

```
# Читання всього вмісту файлу
content = file.read()
print(content)

# Завжди закривайте файл після завершення роботи з ним
file.close()
```

```
Hello World!
I like to learn Python.
```

Запис у файл. Щоб записати дані в файл, відкрийте його в режимі запису (‘**w**’ для запису, заміщуючи вміст, або ‘**a**’ для додавання в кінець файлу). Використовуйте метод **write()** для запису рядка у файл.

```
file = open('example.txt', 'a')  # Відкриття файлу для запису
file.write('\n\nWorking with files is not that difficult!')
file.close()
```

Тепер при зчитуванні інформації з файлу оримаємо вже новий зміст:

```
# Відкриття файлу для читання ('r' - read)
file = open('example.txt', 'r')

# Читання всього вмісту файлу
content = file.read()
print(content)

# Завжди закривайте файл після завершення роботи з ним
file.close()
```

```
Hello World!
I like to learn Python.

Working with files is not that difficult!
```

Робота з файлами за допомогою менеджера контексту. Для автоматичного закриття файлів після завершення роботи з ними рекомендується використовувати менеджер контексту **with**. Це допомагає уникнути помилок, пов’язаних з тим, що файл залишився відкритим.

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)  # Вміст файлу буде автоматично виведений

# Файл буде автоматично закритий після виходу з блоку with
```

```
Hello World!
I like to learn Python.

Working with files is not that difficult!
```

Робота з файлами в бінарному режимі. Для роботи з бінарними файлами (наприклад зображеннями або виконуваними файлами) використовуйте режими **rb** (читання бінарного файлу) або **wb** (запис бінарного файлу).

Перейменування і видалення файлів. Модуль **os** надає функції для роботи з файловою системою, включаючи перейменування і видалення файлів:

```
import os

os.rename('old_name.txt', 'new_name.txt')  # Перейменування файлу
os.remove('example.txt')  # Видалення файлу
```

Керування шляхом до файлів. У Python керування шляхами до файлів може відбуватися за допомогою кількох підходів та модулів. Два основні модулі, які часто використовуються для цих цілей, – це **os** та **pathlib**.

Модуль os. Модуль os містить функції для роботи з операційною системою, включаючи керування шляхами до файлів.

os.path.join() дозволяє об'єднати кілька частин шляху в один, враховуючи особливості операційної системи. Це забезпечує коректне формування шляху на різних ОС:

```
import os

file_path = os.path.join('my_dir', 'sub_dir', 'file.txt')
print(file_path)  # my_dir/sub_dir/file.txt на Unix-подібних системах
                  # my_dir\sub_dir\file.txt на Windows
```

```
my_dir\sub_dir\file.txt
```

os.path.abspath() повертає абсолютний шлях до файлу або директорії:

```
absolute_path = os.path.abspath('example.txt')
# /path/to/directory/my_file.txt на Unix-подібних системах
# C:\path\to\directory\my_file.txt на Windows

print(absolute_path)
```

```
D:\Python\0_Study_Python\example.txt
```

Модуль pathlib. Модуль pathlib був представлений у Python 3.4 і пропонує об'єктно-орієнтований підхід до керування шляхами файлів. Pathlib забезпечує інтерфейс для простої роботи зі шляхами до файлів та директорій:

```python
from pathlib import Path

# Створення об'єкта Path
p = Path('my_dir/sub_dir')
file_path = p / 'file.txt'  # Об'єднання шляхів за допомогою оператора /

print(file_path)  # my_dir/sub_dir/file.txt
```

```
my_dir\sub_dir\file.txt
```

Наведемо декілька корисних методів об’єктів Path:
.resolve() – отримання абсолютного шляху;
.exists() – перевірка існування файлу або директорії;
.is_file(), .is_dir() – перевірка, чи є об’єкт файлом або директорією відповідно;
.mkdir() – створення нової директорії;
.glob() – пошук файлів у директорії за певним шаблоном.

Вибір між os та pathlib залежить від особистих переваг розробника та конкретних вимог проєкту. **Pathlib** часто вважають більш сучасним та зручним у використанні завдяки своєму об’єктно-орієнтованому інтерфейсу.

Серіалізація. Серіалізація – це процес перетворення об’єкта в потік байтів для зберігання в файлі або передачі через мережу. Десеріалізація, відповідно, перетворює потік байтів назад в об’єкт. Цей процес дозволяє зберігати складні об’єкти з усіма їх властивостями та методами у форматі, придатному для запису у файл або передачі.

Модуль pickle у Python. pickle – це модуль у Python, який забезпечує можливості для серіалізації та десеріалізації об’єктів. pickle дозволяє легко зберігати складні Python-об’єкти у файлі та відновлювати їх стан з файлу згодом. Це може бути корисним для збереження конфігурацій, користувацьких сесій або будь-яких інших даних, структура яких вимагає відтворення в точному вигляді.

Використання pickle просте: для збереження об’єкта використовується функція **pickle.dump()**, а для його завантаження – **pickle.load()**. Однак слід бути обережними з даними, серіалізованими за допомогою pickle, отриманими з ненадійних джерел, оскільки десеріалізація ненадійного або модифікованого вмісту може призвести до виконання шкідливого коду.

Приклад 4.22. Реалізувати серіалізацію об’єкта у файл з допомогою модуля pickle. Відновити дані з файлу і вивести їх на екран.

Як складна структура даних використовується словник, який містить різні типи даних, включаючи список і вкладені словники.

Серіалізація даних:

```python
import pickle

# Створення складної структури даних
data = {
    "userID": 1234,
    "name": "John Doe",
    "roles": ["admin", "user"],
    "profile": {
        "email": "john.doe@example.com",
        "address": "123 Main St, Anytown, USA"
    }
}

# Серіалізація даних і збереження у файл
with open('user_data.pkl', 'wb') as file:
    pickle.dump(data, file)

print("Дані успішно серіалізовані та збережені.")
```

```
Дані успішно серіалізовані та збережені.
```

Цей код серіалізує словник data у бінарний формат за допомогою **pickle.dump()** і зберігає його у файлі **user_data.pkl.**

Відновлення даних з файлу:

```python
# Відновлення даних з файлу
with open('user_data.pkl', 'rb') as file:
    loaded_data = pickle.load(file)

print("Дані успішно відновлені з файлу:")
print(loaded_data)
```

Дані успішно відновлені з файлу:
{'userID': 1234, 'name': 'John Doe', 'roles': ['admin', 'user'], 'profile': {'email': 'john.doe@example.com', 'address': '123 Main St, Anytown, USA'}}.

ПРАКТИЧНА ЧАСТИНА

4.9 Підготовка до завдання

1. Ознайомтеся з теоретичними відомостями щодо реалізації різних видів функцій, lambda-функції, генераторів функцій, декораторів функцій, принципами роботи з файлами мовою Python та керування шляхами.
2. Опрацюйте приклади, наведені в теоретичній частині.

4.10 Практичне завдання

Згідно зі своїм варіантом напишіть програму для виконання наступних завдань.

Варіанти завдання № 1

Варіант	***Завдання***
1	Розробіть функцію, яка перевіряє, чи є пароль сильним. Сильний пароль містить щонайменше одну цифру, одну велику та одну маленьку літеру, і має довжину 8 або більше символів.
2	Створіть функцію, яка приймає список чисел і повертає їх середнє, максимальне та мінімальне значення.
3	Напишіть функцію, яка визначає, чи є задане число простим.
4	Створіть функцію, яка генерує і повертає список перших N чисел Фібоначчі.
5	Створіть функцію, яка конвертує температуру з Цельсія в Фаренгейти та навпаки. Функція приймає число та одиницю виміру ('C' або 'F') і повертає конвертоване значення.
6	Створіть функцію, яка генерує випадкові паролі заданої довжини з можливими символами.
7	Напишіть функцію для сортування списку чисел за зростанням або спаданням.
8	Напишіть функцію для друку таблиці значень виразу $y = \sin(x)$ на відрізку [0; 1] з кроком, що дорівнює 0.1.
9	Напишіть функцію, яка приймає текст як вхід і виводить статистику: кількість слів, кількість унікальних слів, частоту появи кожного слова.
10	Створіть функцію, що генерує випадкові паролі. Користувач вказує довжину паролю, чи повинен пароль містити цифри, символи, великі та малі літери.
11	Створіть рекурсивну функцію, яка приймає рядок і повертає список всіх можливих перестановок символів у рядку.
12	Напишіть функцію, яка приймає текст як аргумент і повертає словник з кількістю входжень кожного слова в тексті.
12	Напишіть функцію, яка перевіряє, чи є рядок паліндромом (читається однаково в обидва боки).
14	Напишіть рекурсивну функцію для пошуку максимального елемента у списку.
15	Напишіть рекурсивну функцію для обчислення факторіала числа.

Завдання № 2

Щасливим називають таке шестизначне число, в якого сума перших трьох цифр дорівнює сумі останніх трьох цифр. Знайдіть усі щасливі числа. Напишіть функцію для визначення суми цифр для тризначного числа.

Завдання № 3

Створіть функцію, яка перекладає рядок з коду Morse в англійський текст і навпаки.

Завдання № 4

Реалізуйте функції для шифрування та дешифрування текстових файлів, використовуючи простий алгоритм (наприклад XOR шифрування з ключем). Перевірте їхню роботу, шифруючи та дешифруючи файл.

Завдання № 5

Створіть рекурсивні функції, які для заданого числа обчислюють: а) суми цифр заданого числа, б) загальну кількість цифр, в) максимальні цифри, г) мінімальні цифри .

4.11 Зміст протоколу роботи

Протокол до виконаної роботи оформлюється у вигляді файлу і містить титульний аркуш із назвою навчального закладу, факультету та випускової кафедри, дисципліни та лабораторної роботи, яка виконується, прізвище та ім’я студента і викладача. Наступна сторінка містить назву роботи та її мету, короткі теоретичні відомості про основні оператори і команди, які використовуються для розв’язання поставлених у роботі задач.

Звіт має містити постановку задачі і програмний код її розв’язання з наданням необхідних коментарів. Наприкінці роботи наводяться висновки щодо отриманих результатів по розв’язанню поставлених задач.

Робота захищається індивідуально.

4.12 Контрольні питання для самоперевірки

1. Що таке функція в Python і для чого вона використовується?
2. Як визначити функцію, що виконує певні дії, але не повертає результат за допомогою оператора return?
3. Що таке глобальні та локальні змінні в контексті Python? Як вони визначаються? Яка різниця між глобальними та локальними змінними в контексті функції?
4. Як можна змінити значення глобальної змінної всередині функції?
5. Які особливості використання ключового слова global?
6. Чому вважається поганою практикою надмірне використання глобальних змінних?
7. Як можна повернути кілька значень з функції у Python?
8. Опишіть механізм повернення декількох значень за допомогою кортежів, списків або словників.
9. Що таке анонімна (lambda) функція і в яких випадках її зручно використовувати?
10. Як можна використати функцію map() у Python? Наведіть приклад. Поясніть механізм роботи функції map() та приклади ситуацій, у яких її застосування є найбільш доцільним.

11. Чим відрізняється використання filter() від map(), і як можна замінити filter() використанням спискових включень (list comprehensions)? Опишіть основні відмінності між цими двома функціями і як можна досягнути подібного результату, використовуючи спискові включення.
12. Як можна використати лямбда-функції з map() та filter()? Наведіть приклад.
13. Чому лямбда-функції вважаються анонімними, і в яких сценаріях їх застосування є особливо зручним?
14. Що таке декоратор у Python, його призначення та створення? Наведіть приклад.
15. Як можна використовувати декоратори для вимірювання часу виконання функції? Наведіть приклад створення декоратора, що дозволяє виміряти і вивести час виконання будь-якої функції.
16. Які є типові випадки застосування декораторів у програмуванні на Python?
17. Що таке рекурсивний алгоритм і в чому його переваги та недоліки?
18. Що таке рекурсивна функція і як уникнути переповнення стека при її використанні?
19. Поясніть, що таке рекурсія та які існують методи оптимізації рекурсивних викликів.
20. Як можна визначити функцію з аргументами, що приймають значення за замовчуванням, і як це впливає на виклик функції?
21. Опишіть, як за допомогою *args та **kwargs можна передати в функцію довільну кількість позиційних або іменованих аргументів відповідно.
22. Як функція може приймати іншу функцію як аргумент? Наведіть приклад.
23. Як використовувати генератори всередині функцій?
24. Чому важливо уникати глобальних змінних у функціях і які є альтернативи?
25. Чому і коли варто використовувати ключове слово pass всередині функції?
26. Як створити генератор у Python і в чому полягає відмінність між генератором та звичайною функцією?
27. Як визначити документацію (docstring) для функції у Python і чому вона важлива?
28. Як відкрити файл для читання в Python і що станеться, якщо файл не існує? Наведіть приклад коду та обговоріть різні режими відкриття файлів.
29. Опишіть процес безпечної роботи з файлами за допомогою менеджера контексту with в Python. Чому використання with вважається кращою практикою порівняно з відкриттям і закриттям файлів вручну?
30. Як здійснюється запис даних у файл в Python? Наведіть приклад коду, який демонструє запис рядків у файл, включаючи додавання нових даних до існуючого файлу без перезапису вмісту.

31. Що таке файлові шляхи в Python і як працювати з абсолютними та відносними шляхами? Продемонструйте, як можна використовувати модуль os.path або pathlib для роботи зі шляхами файлів та директорій.
32. Як у Python видалити файл або директорію? Яких запобіжних заходів варто вжити, щоб уникнути випадкового видалення важливих даних?

4.13 Задачі до практичної роботи № 4

1. Напишіть декоратор, який перевіряє, що вхідні аргументи функції мають відповідний тип. Якщо тип не відповідає очікуваному, виникає виключення.
2. Число називається досконалим, якщо сума його дільників буде дорівнювати самому числу, крім самого цього числа.
   Наприклад, 6, 28 – це досконалі числа, тому що сума їх дільників (крім самих чисел) дасть саме число:
   6 (дільники: 1, 2 і 3), 1 + 2 + 3 = 6,
   28 (дільники: 1, 2, 4, 7, 14), 1 + 2 + 4 + 7 + 14 = 28.
   Напишіть функцію, яка визначає, чи є задане натуральне число n досконалим.
3. Напишіть декоратор, який перехоплює винятки, що виникають при виконанні функції, і виводить їх інформацію.
4. Напишіть функцію-генератор, яка генерує рядки Паскалівського трикутника. Паскалівський трикутник – це трикутник чисел, де кожне число є сумою двох чисел, розташованих прямо над ним у попередньому рядку.
5. Реалізуйте функцію, яка приймає URL-адресу як рядок і повертає словник з компонентами URL (схема, домен, шлях, параметри запиту тощо). Врахуйте можливість відсутності деяких компонентів.
6. Дано список записів (словників), що містять інформацію про студентів (наприклад ім’я, прізвище, вік, бали). Реалізуйте серію функцій для фільтрації, сортування і трансформації цього списку (наприклад знаходження студента з найвищим балом, групування за віком тощо).
7. .Напишіть рекурсивну функцію, яка пробігається по заданому каталогу та його підкаталогах і знаходить всі файли з заданим розширенням.

4.14 Завдання до самостійної роботи

1. Визначення та обробка виключення (exceptions) всередині функцій. Використання блоків try, except, else, finally для обробки помилок та винятків.
2. Визначення власних виключень (exceptions) у Python для обробки помилок у функціях.
3. Поняття перевантаження функцій в Python.

4. Типізація аргументів та значень повернення у функціях Python.
5. Аргументи зі зведеною синтаксичною формою (keyword-only arguments) у Python.
6. Використання декораторів для логування аргументів функції та її результатів.
7. Створення декоратора, що приймає параметри. Функціональність декоратора для забезпечення більшої гнучкості або налаштування поведінки декорованої функції.
8. Кешування результатів функції при застосуванні декораторів.
9. Робота з файловою системою. Використання функцій os.remove(), os.unlink(), а також shutil.rmtree() для директорій. Функції os.listdir(path='.'), os.mkdir(path), os.rmdir(path).
10. Область видимості в Python. Доступ до локальних змінних зовнішньої функції.
11. Бінарні файли і серіалізатори. Модуль pickle.

5 РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОГО ЗОРУ НА PYTHON

Мета практичної роботи № 5: ознайомитися з основними поняттями комп'ютерного зору, його застосуванням і реалізацією за допомогою машинного навчання мовою Python.

ТЕОРЕТИЧНІ ПОЛОЖЕННЯ

5.1 Галузі застосування комп'ютерного зору

Комп'ютерний зір – це галузь штучного інтелекту (ШІ), яка використовує машинне навчання та нейронні мережі, щоб навчити комп'ютери та системи отримувати значущу інформацію з цифрових зображень, відео й інших візуальних вхідних даних, а також давати рекомендації чи вживати заходів, коли вони бачать дефекти чи якісь проблеми. Ця технологія імітує людську здатність інтерпретувати візуальну інформацію, але робить це швидше, що значно перевищує людські можливості.

Якщо ШІ дозволяє комп'ютерам мислити, комп'ютерний зір дозволяє їм бачити, спостерігати та розуміти.

Комп'ютерний зір працює майже так само, як і людський, за винятком того, що люди мають перевагу. Людський зір має перевагу тривалості контексту, щоб навчитися розрізняти об'єкти, наскільки вони віддалені, чи рухаються вони, чи щось не так із зображенням.

Комп'ютерний зір навчає машини виконувати ці функції, але він повинен робити це за набагато менший час за допомогою камер, даних і алгоритмів, а не сітківки, зорових нервів і зорової кори. Оскільки система, навчена перевіряти об'єкти або спостерігати за виробничими процесами, може аналізувати тисячі різноманітних процесів за хвилину, помічаючи непомітні дефекти чи проблеми, вона може швидко перевершити людські можливості.

Комп'ютерний зір застосовується в різноманітних галузях та індустріях, пропонуючи широкий спектр використання – від автоматизації та поліпшення процесів до створення нових можливостей. Наведемо деякі напрями застосування цієї технології.

Медицина. Автоматичний аналіз медичних зображень (наприклад рентгенівські знімки, МРТ) для допомоги в діагностуванні захворювань. Моніторинг стану пацієнтів через аналіз відео для виявлення критичних змін у поведінці або фізичному стані.

Автомобільна промисловість. Системи автономного водіння, які використовують комп'ютерний зір для навігації та уникнення перешкод. Аналіз дорожнього руху та системи безпеки (рисунок 5.1).

Роздрібна торгівля. Системи для аналізу поведінки покупців у магазинах. Автоматизовані каси, які розпізнають продукти без сканування штрих-кодів.

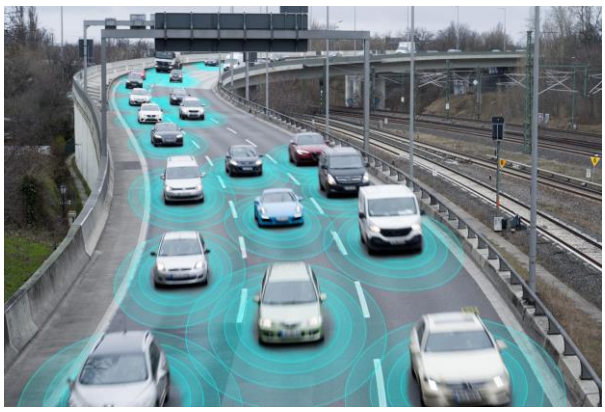


Рисунок 5.1 – Застосування технології комп’ютерного зору
для керування безпілотними автомобілями
Джерело: https://www.forbes.com/sites/peterlyon/2019/03/31/why-the-rush-self-driving-cars-still-have-long-way-to-go-before-safe-integration/?sh=3ecc8549722a

Безпека. Системи відеонагляду з можливістю розпізнавання облич, що можуть ідентифікувати осіб або виявляти підозрілу поведінку. Аналіз відео для автоматичного виявлення надзвичайних ситуацій.

Сільське господарство. Моніторинг стану посівів та визначення необхідності зрошення або обприскування. Автоматизований збір врожаю за допомогою роботів, які «бачать» стиглі плоди.

Виробництво. Автоматичний контроль якості продукції за допомогою візуального аналізу (рисунок 5.2). Оптимізація логістичних процесів через візуальне відстеження вантажів і матеріалів.

Спостереження та розвідка. Виконання детального спостереження, автоматична ідентифікація та відстеження об’єктів, збір розвідданих безпілотними літальними апаратами (БПЛА) з використанням комп’ютерного зору без прямої участі оператора.

Супутникова розвідка. Аналіз зображень, отриманих із супутників, за допомогою комп’ютерного зору, що може автоматизувати виявлення змін на військових базах, переміщення техніки та інші зміни в ландшафті.

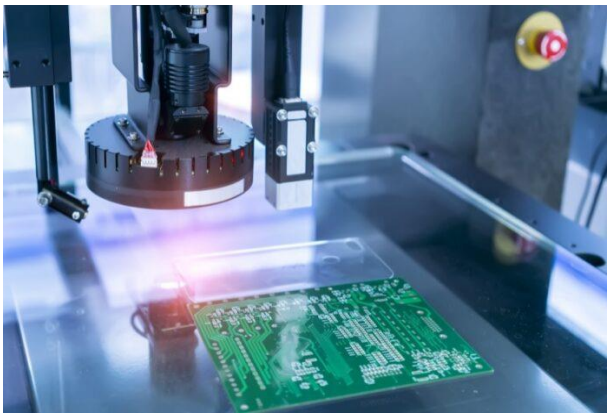


Рисунок 5.2 – Застосування автоматизованої системи контролю
якості продукції
Джерело: https://roboticsbiz.com/machine-vision-systems-1d-2d-and-3d/

Системи протиповітряної оборони. Інтеграція комп'ютерного зору, що може підвищити точність і швидкість реагування систем ППО, автоматизуючи процес виявлення, ідентифікації та відстеження цілей.

Роботизовані наземні системи. Можливість використання комп'ютерного зору для навігації в складних умовах, виявлення та ідентифікації об'єктів, а також для виконання різноманітних бойових та підтримуючих завдань.

Ось кілька прикладів типових завдань для комп'ютерного зору.

Класифікація зображень – бачить зображення і може його класифікувати (собака, яблуко, обличчя людини тощо). Комп'ютерний зір здатний точно передбачити, що певне зображення належить до певного класу. Наприклад, компанія соціальних медіа може використовувати його для автоматичної ідентифікації та відокремлення небажаних зображень, завантажених користувачами.

Виявлення об'єктів – може використовувати класифікацію зображень для ідентифікації певного класу зображень. Приклади включають виявлення пошкоджень на складальній лінії або ідентифікацію обладнання, яке потребує технічного обслуговування, ідентифікацію об'єктів у приміщенні, на вулиці і т.д.

Відстеження об'єкта – стежить за об'єктом після його виявлення. Це завдання часто виконується за допомогою послідовних зображень або відео в реальному часі (рисунок 5.3). Автономним транспортним засобам, наприклад, потрібно не тільки класифікувати та виявляти такі об'єкти, як пішоходи, інші автомобілі та дорожню інфраструктуру, їм потрібно відстежувати їх рух, щоб уникнути зіткнень і дотримуватися правил дорожнього руху.

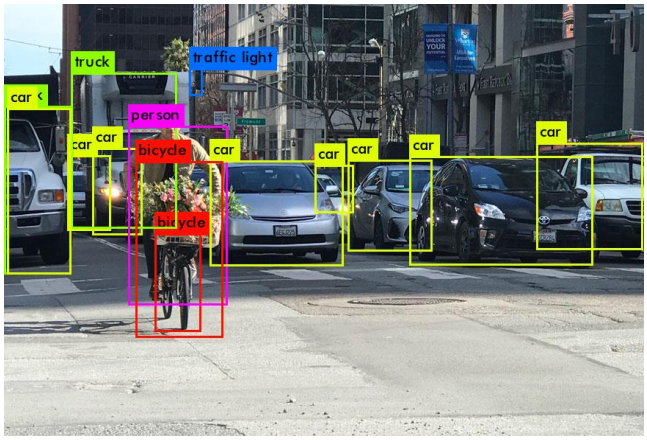


Рисунок 5.3 – Результат виявлення об'єктів та їх ідентифікація

Джерело: https://cryptonews.com/news/atlas-navis-drive2earn-2-0-offers-big-discounts-on-petrol-prices.htm

Пошук зображень – використовує комп'ютерний зір для перегляду, пошуку та отримання зображень із великих сховищ даних на основі вмісту зображень, а не пов'язаних із ними тегів метаданих.

5.2 Основні елементи обробки даних в ML

Основою комп'ютерного зору є здатність комп'ютерних систем виконувати завдання обробки, аналізу, інтерпретації та розуміння зображень або відео з реального світу так, як це робить людський зір. Процес включає кілька ключових етапів і технологій.

Збір даних. Комп'ютерний зір починається зі збору візуальних даних за допомогою камер, сенсорів або інших засобів відеозапису. Ці дані можуть бути у вигляді зображень або відео.

Попередня обробка. Зібрані візуальні дані часто піддаються попередній обробці для покращення якості зображення та видалення шумів. Це може включати корекцію яскравості та контрасту, фільтрацію шуму, а також інші методи обробки зображень.

Сегментація та виділення особливостей. На цьому етапі зображення аналізуються на предмет виділення важливих об'єктів або регіонів інтересу. Сегментація може відбуватися на основі кольору, текстури або інших характеристик. Виділення особливостей полягає в ідентифікації ключових характеристик, таких як кути, краї або специфічні текстури, які допомагають у подальшому аналізі зображень.

Класифікація та розпізнавання. З використанням алгоритмів машинного навчання або глибинного навчання система намагається класифікувати або розпізнати об'єкти на зображенні, базуючись на попередньо навчених моделях. Це може включати розпізнавання облич, ідентифікацію об'єктів, читання тексту тощо.

Розуміння сцени. Останній етап полягає в інтерпретації отриманих даних для розуміння сцени або контексту зображення. Це може включати визначення відносин між об'єктами, розуміння дій, що відбуваються на зображенні, або аналіз змін у послідовності зображень або відео. На рисунку 5.4 наведено типові задачі машинного навчання, серед яких виділяють *класифікацію об'єктів, їх локалізацію, виявлення, сегментацію*.

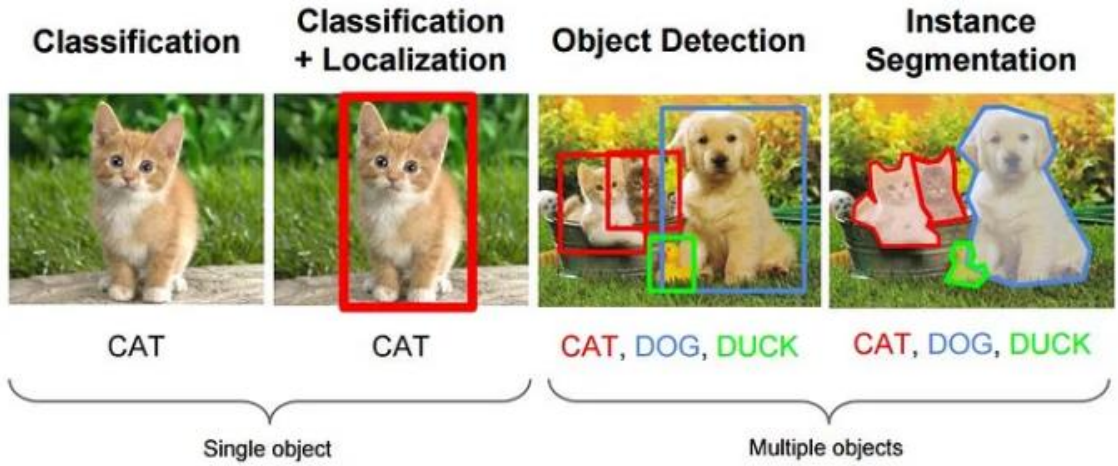


Рисунок 5.4 – Задачі машинного навчання: класифікація, класифікація і локалізація, виявлення об'єктів, сегментація об'єктів

Джерело: https://www.linkedin.com/pulse/what-were-most-significant-machine-learningai-advances-kai-xin-thia/

Технології та інструменти. Розробка систем комп'ютерного зору часто включає використання мов програмування, таких як Python, бібліотек або фреймворків для глибинного навчання й обробки зображень, наприклад, TensorFlow, PyTorch, OpenCV, і базується на технології машинного навчання (**Machine Learning, ML**).

Машинне навчання (**ML**) – це галузь штучного інтелекту, яка дає змогу навчати комп'ютери робити навчання і вдосконалювати свою продуктивність на основі досвіду та даних. В основі машинного навчання лежить ідея, що комп'ютери можуть автоматично аналізувати дані, виявляти закономірності та навчатися від них, щоб приймати рішення або робити прогнози без явного програмування.

Основні характеристики машинного навчання:

Навчання на основі даних. Машинне навчання використовує великі обсяги даних для тренування моделей і вивчення прихованих залежностей у цих даних.

Автоматизована адаптація Моделі машинного навчання можуть адаптуватися до нових даних і змінювати свою продуктивність з часом без необхідності переписування програм.

Здатність до узагальнення. Моделі машинного навчання можуть виявляти залежності у вихідних даних і застосовувати ці знання до нових, раніше не бачених даних.

Вирішення складних завдань. Машинне навчання може застосовуватися до різних завдань, від розпізнавання образів і голосу до аналізу тексту та прогнозування ринків.

Важлива роль даних. Якість та обсяг вхідних даних визначають успішність моделі машинного навчання. Доступ до великих обсягів даних стає ключовим чинником успіху.

Застосування машинного навчання є широкими та всебічними. Воно застосовується в медицині для діагностики хвороб, у фінансах для прогнозування ринків, в автомобільній промисловості для розробки автономних автомобілів, у рекомендаційних системах для рекомендації фільмів і товарів, і в багатьох інших сферах.

Основні методи машинного навчання включають навчання з учителем (**supervised learning**), навчання без учителя (**unsupervised learning**) і з підкріпленням (**reinforcement learning**). Машинне навчання також включає різні типи нейронних мереж, які використовуються в глибокому навчанні (**Deep Learning, DL**). Класифікацію різновидів ML зображено на рисунку 5.5, де також виділено такі задачі, як класифікація, регресія, кластеризація, прийняття рішень і наведено деякі поширені методи для обробки даних.

Одним із різновидів машинного навчання є Deep Learning.

Deep Learning (глибоке навчання, DL) – це підгалузь машинного навчання, яка фокусується на використанні нейронних мереж зі значною кількістю шарів (глибоких мереж) для розв'язання складних завдань аналізу даних (рисунок 5.6).

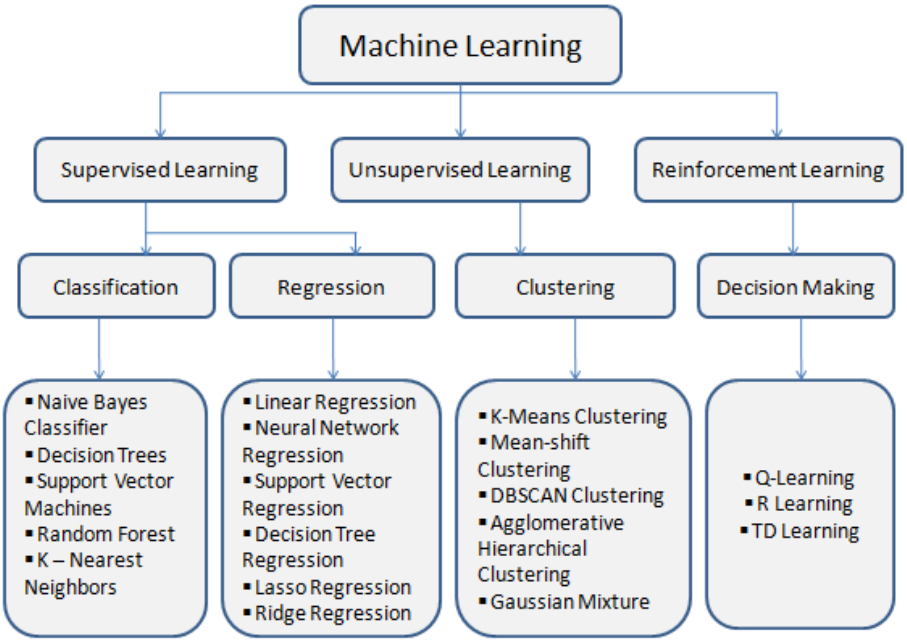


Рисунок 5.5 – Класифікація різновидів машинного навчання

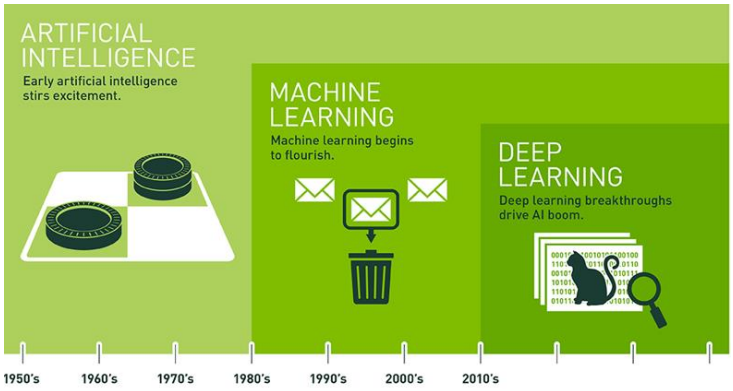


Рисунок 5.6 – Deep Learning як підсистема штучного інтелекту та машинного навчання

Глибоке навчання здійснюється за допомогою нейронних мереж з багатьма прихованими шарами (глибокими шарами), що дозволяє моделям вивчати багато рівнів абстракції та розуміти складні залежності в даних (рисунок 5.7).

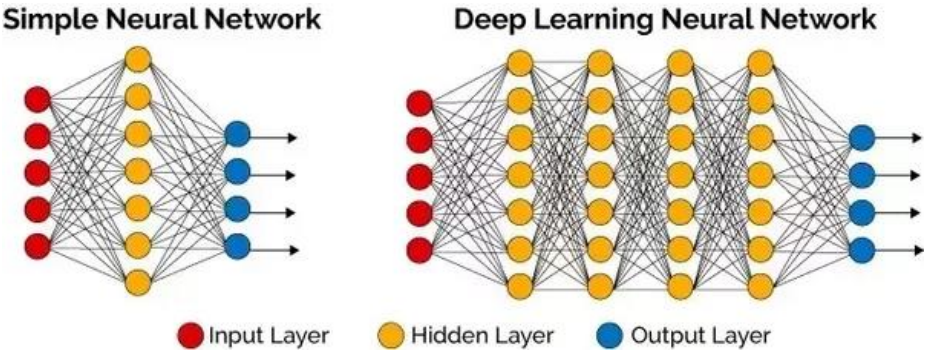


Рисунок 5.7 – Принцип побудови багатошарових перцептронних моделей мереж

Наведемо основні характеристики Deep Learning:

Багатошаровість. Глибокі нейронні мережі мають багато прихованих шарів (від 10 до сотень або навіть тисяч), що дозволяє їм створювати складні моделі залежностей.

Автоматичне вивчення функцій. Deep Learning може вивчати корисні функції з вхідних даних, що дозволяє моделям автоматично визначати, які ознаки або характеристики важливі для розв'язання конкретної задачі.

Здатність до роботи з великими обсягами даних. Глибоке навчання показує високу ефективність при роботі з великими обсягами даних, що робить його ідеальним для завдань, де необхідно аналізувати великі набори інформації, такі як великі тексти, зображення або аудіофайли.

Застосування в багатьох галузях. Deep Learning використовується в різних галузях, включаючи обробку зображень (наприклад розпізнавання об'єктів і обробка медичних зображень), обробку природної мови (переклад текстів, розуміння мови), аналіз тексту, рекомендаційні системи та багато інших.

Основна ідея глибокого навчання – це створення моделей, які вчаться представляти дані на різних рівнях абстракції, поступово збільшуючи складність та інформаційний рівень. Це дозволяє вирішувати завдання, які раніше вважалися важкими або навіть нерозв'язними, за допомогою класичних методів машинного навчання.

5.3 Основи побудови нейронних мереж

Комп'ютерний зір часто реалізується з використанням методів машинного навчання, зокрема глибокого навчання (**DL**), що дозволяє системам ефективно «навчатися» розпізнавати шаблони та об'єкти у великих масивах даних. Ключовим елементом тут є нейронні мережі, особливо згорткові нейронні мережі **(Convolutional Neural Networks, CNN**), які є основою для багатьох сучасних систем комп'ютерного зору. Вони дозволяють точно ідентифікувати об'єкти, класифікувати зображення, розпізнавати текст та виконувати багато інших завдань, пов'язаних з обробкою візуальної інформації.

Нейронна мережа – це серія алгоритмів, які намагаються розпізнавати закономірності в наборі даних за допомогою процесу, що імітує спосіб, яким людський мозок працює. Інакше кажучи, вона може вчитися зі спостережуваних даних та робити складні рішення, використовуючи процес, подібний до того, як людський мозок аналізує та обробляє інформацію.

Наведемо коротку історичну довідку розвитку нейронних мереж.

1943 рік: Уоррен Маккалок і Вальтер Піттс опублікували працю, де було представлено математичну модель нейрону, що заклала основу для розвитку нейронних мереж.

1950-ті роки: Френк Розенблатт розробив перцептрон – один із перших алгоритмів, здатних навчатися та розпізнавати патерни або образи, що стало важливим кроком у розвитку теорії нейронних мереж.

1969 рік: Мінський та Паперт виявили обмеження перцептронів, що привело до першого зниження інтересу до нейронних мереж.

1980-ті роки: Відбулося повторне відродження інтересу до нейронних мереж завдяки введенню алгоритмів зворотного поширення помилки (backpropagation of the error), що дозволило тренувати багатошарові нейронні мережі.

2000-ні роки: Завдяки зростанню обчислювальної потужності та доступності великих обсягів даних глибинне навчання (підмножина нейронних мереж) почало показувати видатні результати в таких сферах, як розпізнавання мови та образів.

Участь у змаганнях з машинного навчання та штучного інтелекту, особливо тих, що зосереджені на застосуванні нейронних мереж, значно сприяла їх розвитку та популяризації. Змагання, такі як ImageNet Large Scale Visual Recognition Challenge (ILSVRC), Kaggle competitions та інші подібні ініціативи, демонструють, як конкурентна атмосфера може стимулювати інновації та покращення в сфері штучного інтелекту.

ImageNet Challenge (ILSVRC) став відомим за свій внесок у розвиток глибокого навчання. В 2012 році команда з Університету Торонто під керівництвом Алекса Кріжевського представила конволюційну нейронну мережу AlexNet, яка значно перевершила всі інші підходи в завданні класифікації зображень. Цей успіх звернув увагу на потенціал глибокого навчання і став поштовхом для багатьох досліджень у цій сфері.

Основою роботи будь-якої нейронної мережі є **перцептрон**, де відбуваються математичні обчислення. Перцептрон – це простий вид штучного нейрона або базової одиниці нейронних мереж, який був розроблений Френком Розенблаттом у 1957 році. Він є базовим блоком для багатьох архітектур нейронних мереж. Перцептрон призначений для вирішення бінарних класифікаційних завдань, таких як розпізнавання образів або визначення, до якого класу належить вхідний об’єкт (рисунок 5.8).

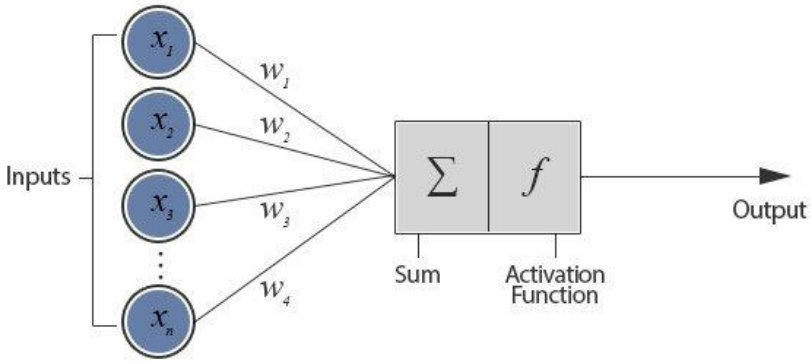


Рисунок 5.8 – Представлення функціонування перцептрона

Основні принципи роботи перцептрона:

Вхідні дані (Inputs). Перцептрон приймає на вхід вектор вхідних даних, який може бути числовим або бінарним. Цей вектор представляє

певні характеристики або ознаки об’єкта, який потрібно класифікувати. На прикладі зображень це можуть бути значення пікселів зображення, яке класифікується. Наприклад, якщо зображення має розмір 28 на 28 пікселів, то загальна кількість пікселів у зображенні буде 784, що і буде вхідним набором даних *Inputs.*

Ваги (*weights, w_i*). Для кожного входу встановлюються ваги (weights), які визначають, наскільки важливий кожний вхід для роботи перцептрона. Ваги є параметрами, які піддаються навчанню.

Вагова сума (Sum). Вхідні дані множаться на відповідні ваги, і всі такі добутки додаються разом. Ця сума називається ваговою сумою (*weighted sum*).

Функція активації (*Activation function*). Вагова сума подається через функцію активації. Функція активації визначає вихідний сигнал перцептрона на основі вагової суми. Популярною функцією активації для перцептрона є функція сигмоїди (*Sigmoid*) або функція Хевісайда (*step function*) (рисунок 5.9).

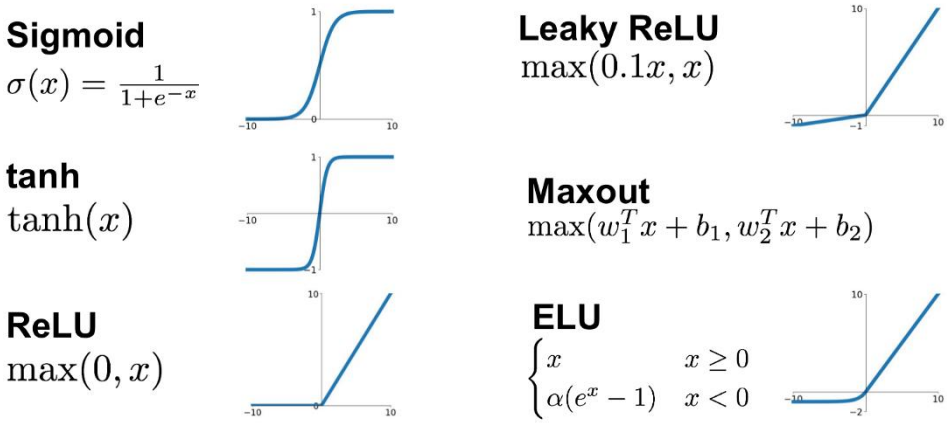


Рисунок 5.9 – Деякі поширені функції активації в ML

Бінарний вихід. Зазвичай вихід перцептрона – це бінарне значення (1 або 0), де 1 може вказувати на один клас, а 0 – на інший. Вихідний сигнал залежить від того, чи перевищує вагова сума певний поріг.

Навчання. Навчання перцептрона полягає в корегуванні ваг з метою покращення його здатності до класифікації. Під час навчання знову обчислюється вихід на основі входів і поточних ваг, і порівнюється з очікуваним результатом. Ваги оновлюються так, щоб зменшити помилку класифікації.

Це основні принципи роботи перцептрона. Важливо враховувати, що класичний перцептрон обмежений у вирішенні складних завдань, але він є основним блоком для більш складних нейронних мереж, таких як багатошарові персептрони (**Multilayer Perceptron – MLP**) (рисунок 5.10) та згорткові нейронні мережі (**Convolutional Neural Network – CNN**) (рисунок 5.11), які використовуються для розв’язання різних завдань машинного навчання.

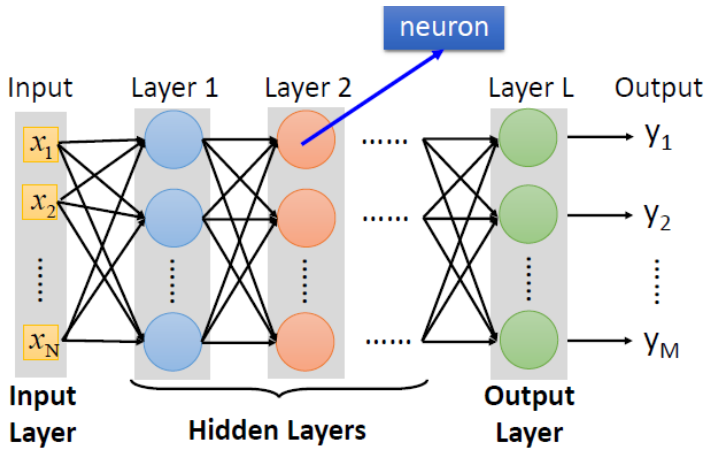


Рисунок 5.10 – Принцип побудови багатошарових перцептронних моделей мереж (MLP)

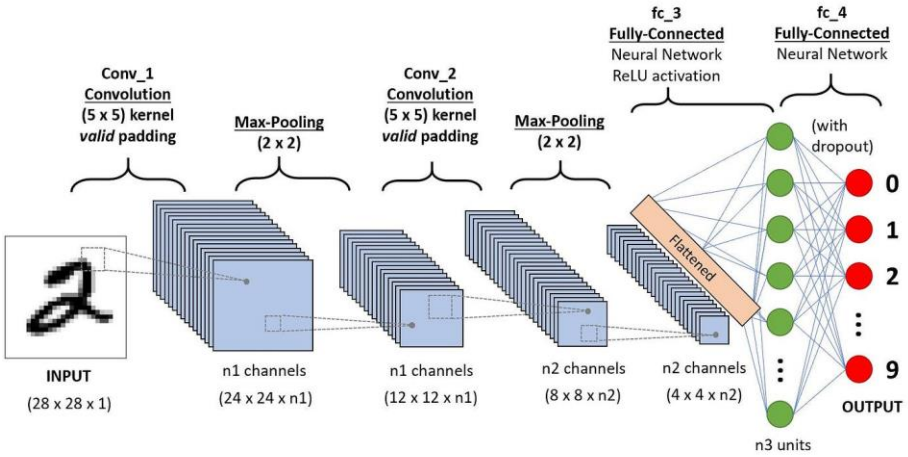


Рисунок 5.11 – Приклад реалізації нейромережевої структури Convolutional Neural Network (CNN) для реалізації задачі класифікації

Джерело: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Процес навчання нейромережевої структури в загальному вигляді можна зобразити на рисунку 5.12, де уточнюються вагові коефіцієнти в результаті багаторазових процедур зворотних обчислень (**Backpropagation**) за певними градієнтними сценаріями оптимізації параметрів, поки не досягнеться бажаний (за певною ймовірністю) збіг вхідних даних з вихідними. З часом алгоритм стає точнішим, а самі етапи проходження навчання називають **епохами.** При такому збігу вважається, що отримано відповідну модель поведінки системи після її навчання, яку можна використовувати для передбачення на інших, перед цим не відомих даних.

Нейронні мережі мають різні архітектури та ролі, і їх класифікація залежить від кількох факторів, таких як структура та зв’язки між шарами, способи навчання та застосування. Деякі основні типи нейронних мереж включають такі:

Feedforward Neural Networks (FNN) – одношарові перцептрони (Single-Layer Perceptrons), мають один вхідний і один вихідний шар, використовуються для бінарної класифікації;

Багатошарові нейронні мережі (Multi-Layer Perceptrons, MLP) – мають один або більше прихованих шарів між вхідним і вихідним шарами. Використовуються для багатьох задач, включаючи розпізнавання образів та обробку природної мови;

Convolutional Neural Networks (CNN) – згорткові нейронні мережі, спеціалізуються на обробці зображень і використовують шари згортки для виділення ознак зображень;

Recurrent Neural Networks (RNN) – рекурентні нейронні мережі, мають зворотний зв'язок між шарами, що дозволяє моделі працювати з послідовними даними, такими як текст або часові ряди. Вони корисні для завдань, де порядок даних має значення;

*Long Short-Term Memory (LSTM) Networks – м*ережі з довготривалою та короткостроковою пам'яттю. Це підтип рекурентних нейронних мереж, який розроблений для вирішення проблеми зникнення та вибування градієнта у навчанні RNN;

Radial Basis Function Networks (RBFN) – мережі з радіальною базисною функцією – використовують радіальні базисні функції як функції активації і використовуються для задач кластеризації та апроксимації функцій;

Autoencoders (Автоенкодери) – використовуються для зменшення розмірності даних та виділення важливих ознак;

Generative Adversarial Networks (GANs) – генеративно-змагальні мережі, складаються з генеративної та дискримінативної мереж, використовуються для створення нових даних.

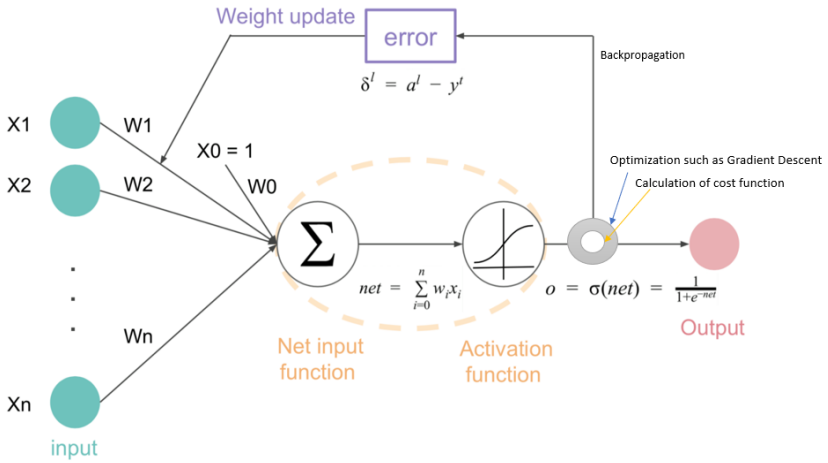


Рисунок 5.12 – Процес навчання нейромережевої структури

Джерело: https://datascience.stackexchange.com/questions/44703/how-does-gradient-descent-and-backpropagation-work-together.

Це лише кілька прикладів типів нейронних мереж. Важливо враховувати, що кожний тип має свої властивості та застосування, і вибір конкретного типу мережі залежить від завдання, яке ви хочете вирішити.

5.4 Основи побудови CNN для обробки зображень

Згорткова **нейронна мережа (ConvNet/CNN)** – це алгоритм глибокого навчання, який може сприймати вхідне зображення, призначати важливість (вагові значення та зміщення) різним об’єктам на зображенні та мати можливість відрізнити один від одного. Попередня обробка, необхідна в ConvNet, набагато менша порівняно з іншими алгоритмами класифікації. Архітектура ConvNet аналогічна структурі підключення нейронів у людському мозку та була натхненна організацією зорової кори. Окремі нейрони реагують на стимули лише в обмеженій області поля зору, відомій як рецептивне поле. Набір таких полів перекривається, щоб охопити всю візуальну область. Структура CNN намагається імітувати спосіб, яким людське зорове сприйняття аналізує візуальні об’єкти, виявляючи шаблони, як-от краї, кути, форми тощо.

Зображення – це не що інше, як матриця значень **пікселів**. Тому зображення, розмір якого, наприклад, – 3 на 3 пікселі, можна перевести у вектор розмірності 9 (рисунок 5.13, а), а для кольорового зображення зробите те саме, тільки для трьох складових зображення RGB (рисунок 5.13, б).

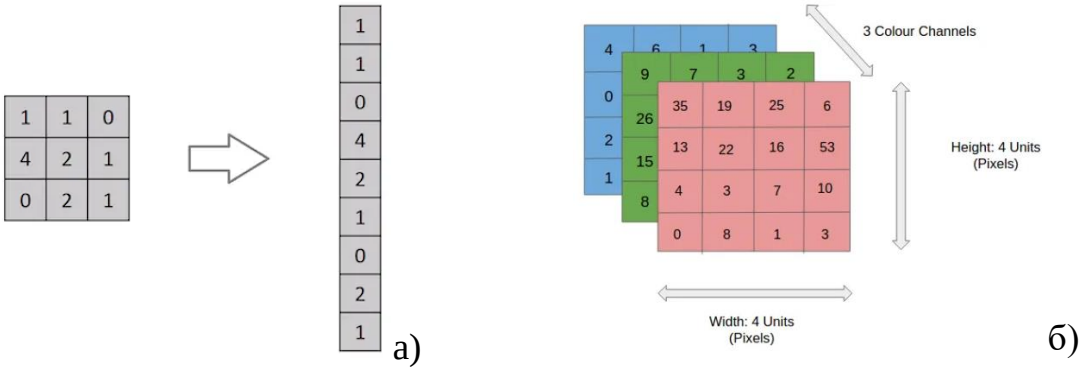


Рисунок 5.13 – Представлення двовимірного зображення у вигляді одновимірного набору даних (а) та обробка кольорового зображення (б)

Піксель (скорочення від "picture element") є основною одиницею виміру в цифровому зображенні або дисплеї. Це найменший контрольований елемент зображення на екрані або в цифровій фотографії. Пікселі є крапками, що разом формують зображення, яке ми бачимо на моніторі, телевізорі, смартфоні чи іншому дисплеї. Кожний піксель може відображати певний колір або відтінок, і коли ми дивимося на зображення з достатньої відстані, наш мозок інтегрує ці кольорові крапки в одне цілісне зображення.

Основними властивостями пікселів є колір та розмір.

Колір – у цифрових зображеннях кольори пікселів зазвичай представлені через комбінацію червоного, зеленого і синього (**RGB**) світла. Змінюючи інтенсивність кожного з цих кольорів, можна отримати широкий спектр інших кольорів.

Розмір – розмір пікселя може варіюватися залежно від роздільної здатності та розміру дисплея. Висока роздільна здатність і малий розмір дисплея зазвичай означають менший розмір пікселів, що сприяє вищій гостроті зображення.

На рисунку 5.14 наведено приклад представлення чорно-білого зображення у вигляді двовимірного набору числових значень яскравості кожного пікселя для машинної обробки, де чорному зображенню відповідає значення «0», а білому – «255», відтінки сірого знаходяться в проміжку між цими значеннями. Для кольорових зображень застосовується така сама технологія, тільки для трьох наборів пікселів – **RGB** (рисунок 5.15).

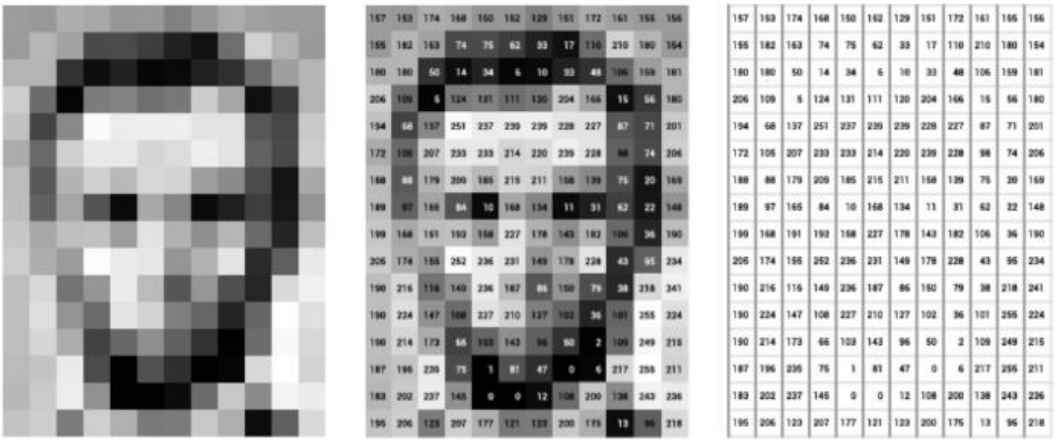


Рисунок 5.14 – Представлення зображення
у вигляді двовимірного набору даних
Джерело: https://www.v7labs.com/blog/what-is-computer-vision

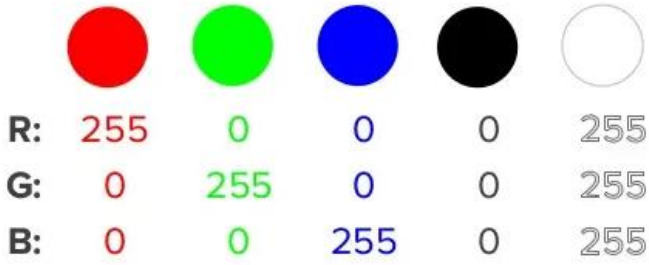


Рисунок 5.15 – Значення яскравості пікселів для RGB зображення

ConvNet може успішно фіксувати просторові та часові залежності в зображенні за допомогою застосування відповідних фільтрів (**ядер – kernel**). Архітектура забезпечує кращу адаптацію до набору даних зображення завдяки зменшенню кількості залучених параметрів і можливості повторного використання вагових коефіцієнтів. Інакше кажучи, мережу можна навчити краще розуміти складність зображення.

Реальні зображення, які підлягають обробці, містять дуже багато інформації, яку потрібно обробити. Роль ConvNet полягає в зменшенні

зображень до форми, яку легше обробляти, без втрати функцій, які є критично важливими для отримання хорошого прогнозу. Це важливо, коли потрібно розробити архітектуру, яка не тільки добре вивчає функції, але й масштабується до масивних наборів даних. Зменшення даних для подальшої обробки відбувається при застосуванні технології згортки, коли на зображення заданої розмірності діє певний фільтр (ядро), в результаті чого отримується зображення меншої розмірності при збереженні основних властивостей про зображення (рисунок 5.16).

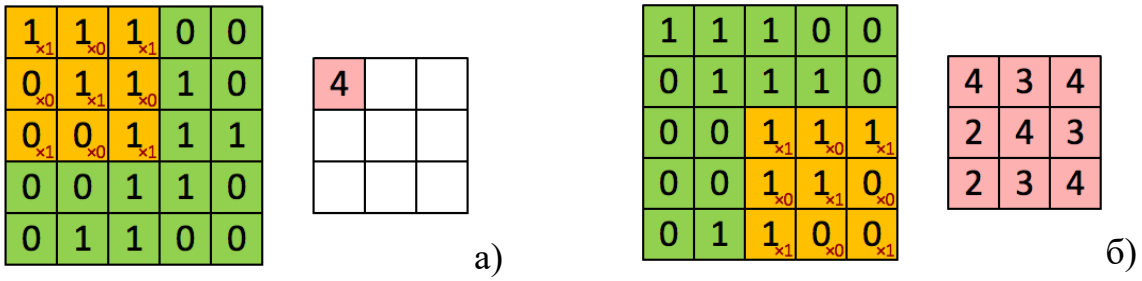


Рисунок 5.16 – Застосування згортки до вхідного зображення

Розміри зображення = 5 (висота) x 5 (ширина) x 1 (кількість каналів, наприклад, для моделі RGB). У наведеній вище демонстрації зелена частина представляє вхідне зображення 5x5x1. Елемент, який бере участь в операції згортки в першій частині згорткового шару, називається **ядром або фільтром (K)** і представлений жовтим кольором. Для прикладу, оберемо ядро **K** (фільтр) як матрицю 3x3x1, яка представлена таким чином:

```
1 0 1
0 1 0
1 0 1
```

.

Ядро зсувається 9 разів через **Stride Length = 1 (Non-Strided)**, кожного разу виконуючи операцію поелементного множення (добуток Адамара) між K і частиною зображення, над якою нависає ядро. В результаті замість зображення, яке містить 25 пікселів (5*5), отримаємо зображення з 9 пікселями (3*3), яке містить у собі основні властивості початкового зображення.

Дійсно, якщо виконаємо операцію згортки заданого ядра K над зображенням (3*3), то отримаємо число 4 (рисунок 5.14, а), якщо виконаємо такі операції над усім зображенням – заповнену матрицю, яка зображеня на рисунку 5.14, б.

Властивості нової отриманої матриці зображення будуть залежати від вибору ядра **K**. Нейромережева структура будується таким чином, щоб можна було задавати як розмір ядра **K**, так і кількість, тим самим, отримується можливість виділення різноманітних ознак з початкового зображення при зменшенні його розміру для подальшої обробки.

На рисунку 5.17 демонструється рух ядра по всьому полю зображення, причому можуть бути різні режими стосовно кроку пересування (**Stride Length**).

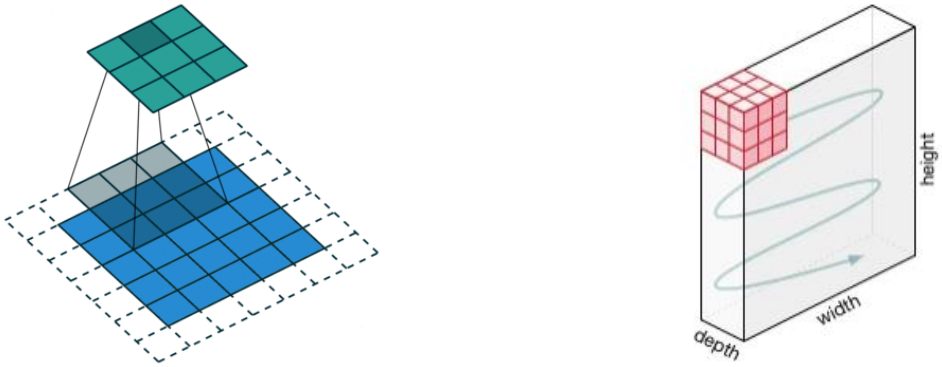


Рисунок 5.17 – Рух ядра по всьому полю зображення

Якщо **Stride = 1**, то фільтр переміщається по зображенню на один піксель за крок. Це забезпечує детальний аналіз зображення, оскільки кожна можлива позиція фільтра розглядається. Вихідна карта особливостей буде відносно великою, оскільки більшість інформації з зображення захоплюється. При **Stride > 1** фільтр переміщається на більшу кількість пікселів (наприклад 2, 3 і т.д.) за крок. Це призводить до «пропускання» деяких областей зображення без їх аналізу, що може зменшити детальність виявлення особливостей, але збільшує обчислювальну ефективність і зменшує розмір вихідної карти особливостей.

При конфігурації конволюційного шару в нейронній мережі, окрім довжини кроку (stride length), можна налаштувати ще декілька важливих параметрів, кожен з яких має вплив на спосіб обробки вхідних даних і формування вихідної карти особливостей. Ось деякі з них:

1. *Розмір Фільтра (Kernel Size)* – визначає розміри фільтра або ядра, яке переміщається по вхідному зображенню. Більші фільтри здатні захоплювати більші особливості або контексти, тоді як менші фільтри концентруються на детальніших особливостях.

2. *Кількість Фільтрів (Number of Filters)* – визначає кількість різних фільтрів, які застосовуються в конволюційному шарі. Кожний фільтр здатний виявляти певні особливості в даних. Збільшення кількості фільтрів веде до більшої глибини вихідної карти особливостей.

3. *Padding (Доповнення)* – визначає, чи будуть додані додаткові нулі до країв вхідного зображення перед застосуванням фільтрації. *"Valid" padding* означає відсутність доповнення, тоді як *"Same" padding* додає достатньо нулів, щоб вихідний розмір відповідав вхідному.

4. *Функція Активації (Activation Function)* – вказує, яка функція активації буде застосована до вихідних даних конволюційного шару. Популярні варіанти включають *ReLU (Rectified Linear Unit)* та її варіації, як-от *Leaky ReLU, PReLU* та інші (рисунок 5.9).

5. *Pooling (Пулінг)* – хоча це не налаштування самого конволюційного шару, пулінг часто використовується після конволюційних шарів для зменшення розміру вихідної карти особливостей і зменшення обчислювальної складності. Існують різні типи пулінгу, як-от *max pooling* і *average pooling* (рисунок 5.18).

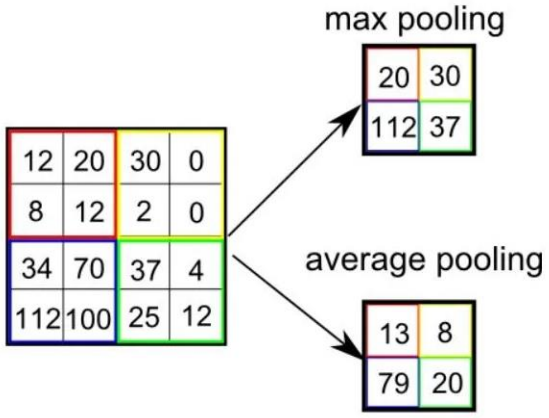


Рисунок 5.18 – Застосування *Pooling*
для зменшення розмірності зображення

Max pooling та average pooling – це два популярні методи пулінгу, використовувані в конволюційних нейронних мережах для зменшення розміру вхідної карти особливостей, зменшення обчислювальної складності та контролю перенавчання.

Max pooling зменшує розмір вхідних даних, вибираючи максимальне значення з кожної підмножини вихідної карти особливостей. Наприклад, якщо застосовувати max pooling з вікном 2x2 до карти особливостей, для кожних чотирьох значень у вікні вибирається найбільше. Цей метод ефективно зберігає найвиразніші особливості (наприклад краї або кути), одночасно зменшуючи розмір даних.

Average pooling також зменшує розмір вхідних даних, але замість вибору максимального значення, він обчислює середнє значення кожної підмножини. Це означає, що для кожного вікна 2x2 (або іншого розміру) на карті особливостей вираховується середнє з чотирьох значень. Цей метод забезпечує більш рівномірне уявлення про особливості зображення, але може втрачати деякі виразні особливості.

6. *Batch Normalization (Нормалізація пакета)*

Нормалізувати можна не тільки вхідні дані. Вхідні дані кожного шару також зазвичай нормалізуються. Ця техніка називається **Batch Normalization (BN)**. Вона була представлена у 2015 році і є одним із найефективніших методів навчання глибоких нейронних мереж. Пакетна нормалізація дозволяє використовувати більш високу швидкість навчання без проблем зі зникненням або вибухом градієнтів. Ця техніка використовується для покращення стабільності та швидкості навчання

шляхом нормалізації входів кожного шару до єдиного розподілу. Вона може бути застосована після конволюційних шарів та перед функцією активації.

Розглянемо ще декілька важливих понять, які використовуються при побудові нейромережевих структур.

Повнозв'язна нейронна мережа (**Fully Connected Neural Network, FCNN**) – це найпростіший тип нейронних мереж, де кожний нейрон у певному шарі з'єднаний з усіма нейронами наступного шару (рисунок 5.19).

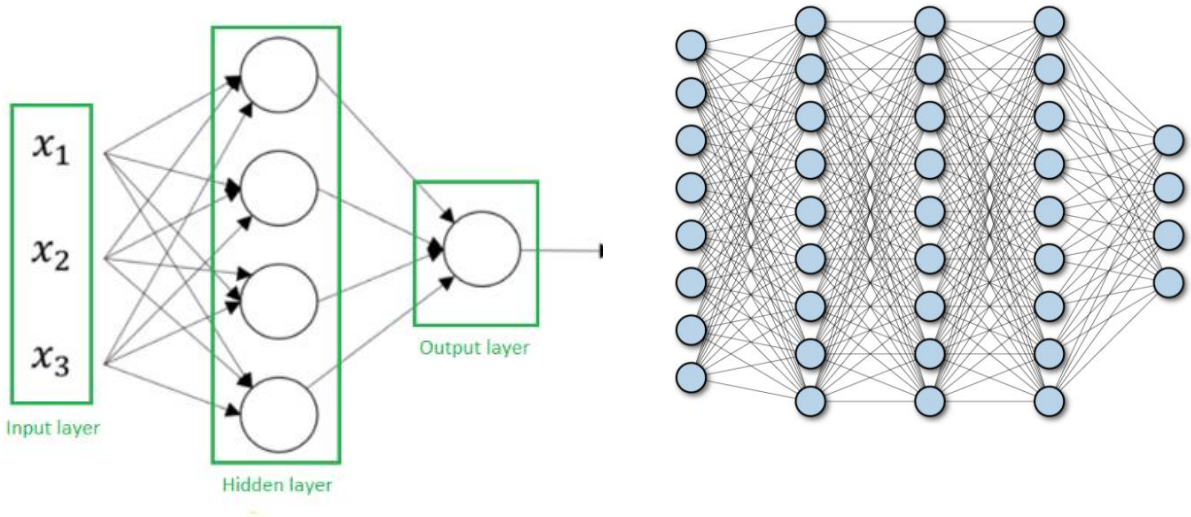


Рисунок 5.19 – Приклад FCNN з одним та багатьма внутрішніми шарами

Ця архітектура дозволяє мережі вивчати складні шаблони та залежності в даних шляхом застосування нелінійних перетворень.

Повнозв'язна нейронна мережа складається з трьох основних типів шарів:

вхідний шар (Input layer) – приймає вхідні дані. Кількість нейронів у цьому шарі відповідає розміру вхідних даних;

приховані шари (Hidden layers) – один або кілька шарів, що виконують обчислення за допомогою ваг, зміщень (bias) і активаційних функцій. Кожний нейрон у прихованому шарі з'єднаний з усіма нейронами попереднього та наступного шарів;

вихідний шар (Output layer) – генерує вихідні дані мережі. Кількість нейронів у цьому шарі зазвичай відповідає кількості класів для задачі класифікації або одному нейрону для задачі регресії.

Функція **Flatten** є важливою складовою для побудови нейронних мереж, особливо у випадках, коли необхідно обробляти дані зображень. Flatten у нейронних мережах використовується для перетворення тривимірних вхідних даних (наприклад зображень або карт особливостей) в одновимірний вектор (рисунок 5.13). Це потрібно для подальшої обробки даних у повнозв'язному шарі (fully connected layer), який приймає на вхід одновимірний вектор.

Приклад 5.1. Нехай є вхідні дані зображення розміром 28x28x3 (наприклад RGB зображення), де 28x28 – розмір зображення у пікселях, а 3 – кількість каналів (RGB). Щоб передати ці дані у повнозв'язний шар, необхідно спочатку перетворити їх у вектор розміром 28*28*3 = 2352.

Функція Flatten вирівнює кожний шар вхідних даних, зберігаючи порядок елементів, але перетворюючи їх в одновимірний вектор.

Функцію Flatten часто використовують в кінці відображення (feature extraction) зображень через згорткові шари (convolutional layers) у конволюційних нейронних мережах. Після проходження зображення через кілька згорткових та пулінгових шарів результат вважається вектором розміром 3D. Щоб передати цей вектор в повнозв'язний шар для класифікації або регресії, його необхідно спочатку перетворити у вектор фіксованого розміру за допомогою функції Flatten.

Dropout – це техніка регуляризації для нейронних мереж, яка допомагає запобігти перенавчанню. Перенавчання – це ситуація, коли нейронна мережа занадто добре «запам'ятовує» навчальні дані, включаючи їх шум та особливості, які не є характерними для загальної вибірки. Це може призвести до поганої загальної продуктивності на нових, раніше не відомих даних.

Під час тренування на кожному кроці (або епосі) кожний нейрон мережі має певну ймовірність (зазвичай від 0.2 до 0.5) тимчасово бути «виключеним», тобто його вихідний сигнал ігнорується (рисунок 5.20).

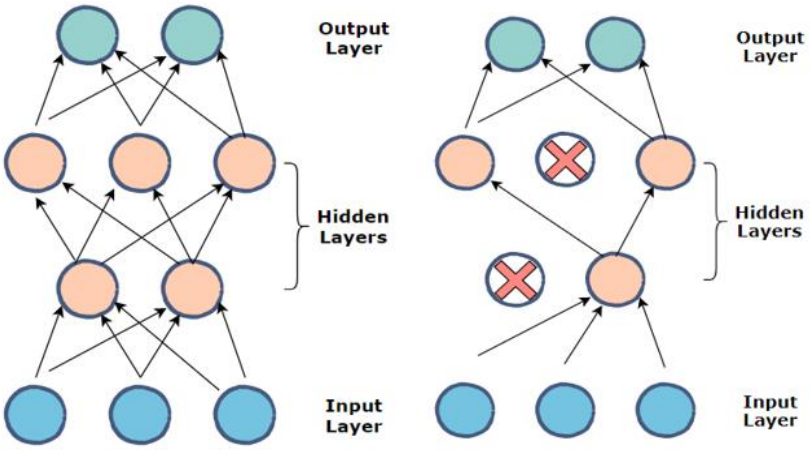


Рисунок 5.20 – Застосування техніки Dropout
для зменшення ефекту перенавчання моделі

Це означає, що стан мережі на кожному кроці тренування є трохи іншим, що змушує мережу навчатися більш робастним шляхом, не покладаючись надто сильно на будь-які конкретні з'єднання між нейронами.

Позитивний ефект від застосування цієї техніки – це зменшення перенавчання.

Dropout зазвичай застосовується у повнозв'язних шарах (fully connected layers), де ризик перенавчання особливо високий через велику

кількість параметрів. Проте його також можна використовувати у згорткових шарах (convolutional layers) для покращення робастності моделі.

Таким чином, при застосування технік, які були наведені вище, загальна структура нейромережевої структури Convolutional Neural Network буде мати вигляд як на рисунку 5.21.

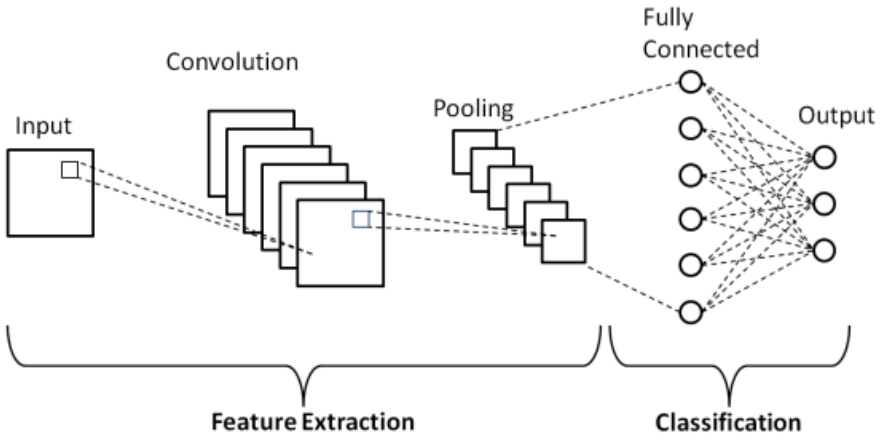


Рисунок 5.21 – Основні перетворення в нейромережевій структурі Convolutional Neural Network (CNN)

Для закріплення представленого матеріалу в практичній роботі буде розглянуто приклад розпізнавання рукописних цифр набору MNIST при застосуванні нейромережевих технологій мовою Python. MNIST (Modified National Institute of Standards and Technology database) – це велика база даних рукописних цифр, яка часто використовується для тренування різноманітних систем обробки зображень, зокрема у сфері машинного навчання та комп'ютерного зору.

5.5 Параметри якості нейромережевої моделі

В машинному навчанні та оцінці продуктивності нейромережевої моделі існує низка показників, які використовуються для їх оцінювання на навчальних і тестових даних. Вибір конкретних показників залежить від типу завдання (класифікація, регресія і т. д.) і характеристик даних. Ось деякі з найбільш поширених показників ефективності моделей для задач класифікації:

Точність (Accuracy) – відсоток правильних класифікацій;

Точність (Precision) – відсоток об'єктів, правильно визначених як позитивні серед усіх об'єктів, визначених як позитивні;

Повнота (Recall) – відсоток позитивних об'єктів, які були правильно визначені моделлю серед усіх позитивних об'єктів;

F1-міра (F1-Score) – гармонічне середнє між точністю та повнотою, забезпечуючи збалансованість обох показників.

Графічне та математичне представлення цих термінів можна зобразити на рисунку 5.22, де TP = істинні позитивні результати, TN = істинні негативні результати, FP = хибні позитивні результати та FN = хибні негативні результати.

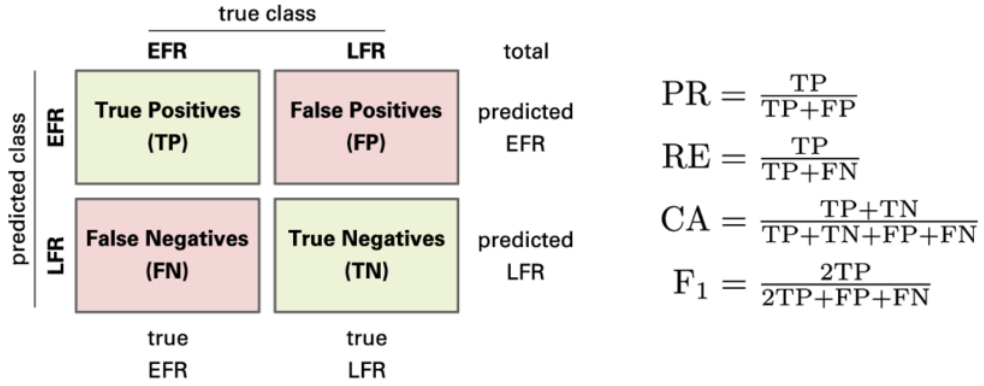


Рисунок 5.22 – Основні показники ефективності в машинному навчанні

Accuracy (точність) – це ключовий показник, що використовується для оцінки продуктивності класифікаційних моделей у машинному навчанні. Вона визначає відсоток правильних прогнозів моделі серед усіх прогнозів і обчислюється за допомогою такої формули:

Accuracy=Кількість правильних прогнозів/Загальна кількість прогнозів.

Для визначення точності спочатку фіксуються такі показники.

Кількість правильних прогнозів – це кількість об’єктів, які були правильно класифіковані моделлю, де передбачені класи збігаються з фактичними класами.

Загальна кількість прогнозів – це загальна кількість об’єктів, для яких модель зробила класифікаційні прогнози.

Після отримання цих значень вони вводяться в формулу для розрахунку точності.

Зазначені параметри будуть використані для оцінювання ефективності отриманої нейромережевої моделі.

Наприклад, нехай є модель, яка була випробувана на 100 об’єктах. Припустимо, якщо вона правильно класифікувала 90 з них, то точність моделі буде:

Accuracy=90/100=0.90.

Отже, точність у цьому випадку становить 90%. Ця метрика важлива для загальної оцінки якості класифікаційної моделі. Разом з тим, потрібно розглядати інші метрики у випадках дисбалансу класів (коли кількість одного класу значно відрізняється від іншого). Для цього потрібні такі оцінки, як точність (precision), повнота (recall) і F1-міра.

Precision (точність) – це критерій, що використовується для оцінки ефективності класифікаційної моделі в машинному навчанні, зокрема в завданнях класифікації. Цей показник визначає відсоток об’єктів, які були

правильно визначені як позитивні серед усіх об’єктів, які модель визначила як позитивні. Точність вказує на те, наскільки «точними» або «чистими» є класифікації позитивних об’єктів моделі.

Формула для розрахунку точності (precision) така:

Precision=Кількість правильно визначених позитивних об’єктів/ Загальна кількість об’єктів, визначених як позитивні.

Наприклад, нехай при класифікації 100 об’єктів модель визначила 95 об’єктів як позитивні. З цих 95 позитивних об’єктів 80 були правильно визначені моделлю. Тоді точність буде:

$$Precision=80/95=0.84.$$

Точність становить 84%. Висока точність означає, що модель мало помиляється при визначенні позитивних об’єктів.

Recall (повнота) – це метрика, що використовується для оцінки якості класифікаційної моделі в машинному навчанні, зокрема в задачах класифікації. Вона вимірює відсоток позитивних об’єктів, які були правильно визначені моделлю серед усіх фактичних позитивних об’єктів. Повнота допомагає визначити, наскільки «комплексним» є визначення позитивних об’єктів моделлю.

Формула для розрахунку повноти (recall) така:

Recall=Кількість правильно визначених позитивних об’єктів/ Загальна кількість фактичних позитивних об’єктів

Припустимо, що при класифікації 100 об’єктів модель визначила 90 об’єктів як позитивні. Загальна кількість фактичних позитивних об’єктів становить 95, з яких визначила лише 85. Тоді повнота буде:

$$Recall=85/95=0.89.$$

F1-міра (F1-Score) – це гармонічний середній показник, який комбінує точність (precision) і повноту (recall) для оцінки якості класифікації моделі в машинному навчанні, зокрема в завданнях класифікації:

$$f1=(2\times Precision\times Recall) / (Precision+Recall)$$

Таким чином, описані показники будуть використані для оцінки ефективності нейромережевої моделі.

5.6 Основні етапи обробки даних при побудові моделі ML

Python є однією з найпопулярніших мов програмування для роботи з нейронними мережами, зокрема завдяки великому вибору бібліотек і фреймворків. Для реалізації нейронних мереж у Python часто використовують такі бібліотеки, як:

TensorFlow – відкрита бібліотека для численних завдань глибинного навчання, розроблена командою Google Brain. Вона дозволяє легко конструювати, тренувати та тестувати нейронні мережі з різноманітними архітектурами;

Keras – високорівневий API, який працює на основі TensorFlow, що зробило його більш доступним для експериментів з нейронними мережами;

PyTorch – бібліотека від Facebook, яка забезпечує гнучкість у проєктуванні та швидкість виконання завдяки динамічним графам обчислень і зручним інструментам.

Застосування нейромереж і машинного навчання включає декілька основних етапів, які можна узагальнити за наступною методикою (рисунок 5.23).

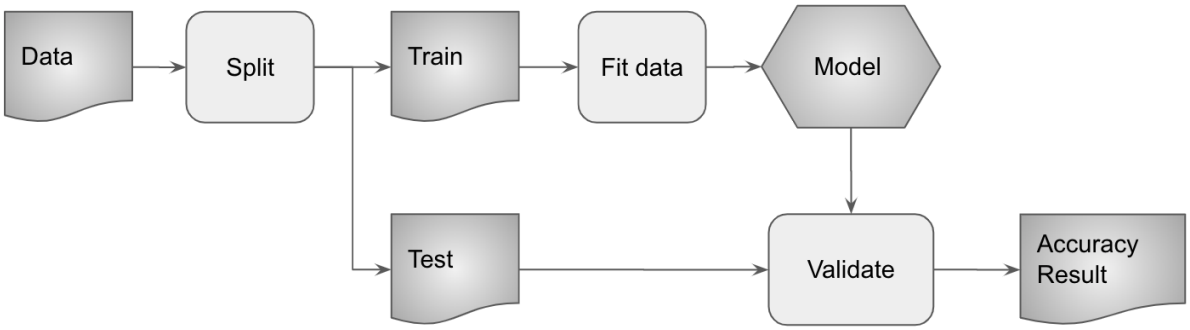


Рисунок 5.23 – Основні етапи обробки даних при побудові моделі ML

Збір та підготовка даних. Збір необхідних даних (*Data*) для вирішення конкретної задачі.

Очищення та попередня обробка даних, така як усунення відсутніх значень, видалення аномальних даних, нормалізація тощо.

Розбиття (*Split*) даних на *тренувальний (Train), валідаційний (Validate)* та *тестовий* (*Test*) набори для оцінки моделі.

Вибір моделі. Вибір відповідної архітектури моделі залежно від характеру задачі: нейронна мережа, згорткова нейронна мережа, рекурентна нейронна мережа тощо.

Визначення гіперпараметрів моделі, таких як кількість шарів, кількість нейронів у кожному шарі, функції активації тощо.

Тренування моделі. Подання тренувальних даних на вхід моделі (*Model*) для навчання (*Fit data*). Використання алгоритму оптимізації для оптимізації параметрів моделі. Моніторинг метрик процесу тренування, таких як втрати та точність, для оцінки продуктивності моделі.

Оцінка моделі. Використання валідаційного набору даних для оцінки моделі під час тренування з метою попередження перенавчання (*Accuracy Result*).

Використання тестового набору даних для фінальної оцінки точності та продуктивності моделі.

Налаштування та вдосконалення моделі. Налаштування гіперпараметрів моделі на основі результатів оцінки. Використання технік регуляризації, таких як Dropout або L2 регуляризація, для підвищення робастності моделі. Експерименти з різними архітектурами моделі для отримання кращих результатів.

Впровадження та використання моделі. Використання навченої моделі для розв'язання реальних завдань або інтеграція її в продуктивне середовище. Моніторинг продуктивності моделі в реальному часі та можливе оновлення в майбутньому.

Налаштування гіперпараметрів і тонке налаштування моделі ML можуть значно покращити її точність. Цей процес може включати зміну параметрів навчання, архітектури моделі або навіть передобробку даних.

5.7 Побудова моделі розпізнавання цифр на прикладі набору даних MNIST для повністю з'єднаної (*Dense*) нейромережі

Розпізнавання рукописних цифр – це процес, який дозволяє машинам розпізнавати цифри, написані людиною. Це непросте завдання для машини, оскільки рукописні цифри не ідеальні, відрізняються від людини до людини та можуть бути створені з багатьма різними стилями.

MNIST – це набір даних, який часто використовується в сфері машинного навчання та комп'ютерного зору для навчання й оцінки алгоритмів класифікації зображень. Він складається з рукописних цифр від 0 до 9, записаних людиною, які мають розмірність 28x28 пікселів. Кожне зображення представлене в чорно-білому форматі (рисунок 5.24).

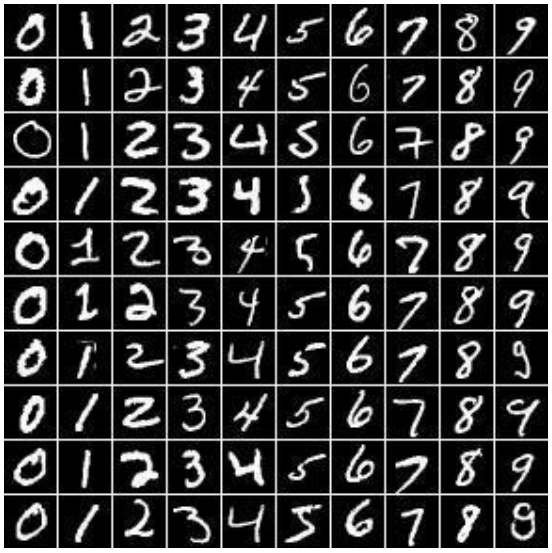


Рисунок 5.24 – Представлення набору MNIST

Розмір набору даних – MNIST містить 60 000 тренувальних зображень і 10 000 тестових зображень. Зображення являють собою нормалізовані (за розміром) і центровані в межах фіксованої рамки (28x28 пікселів) рукописні цифри від 0 до 9.

Формат – **з**ображення зберігаються у вигляді 28x28 піксельних зображень з одним каналом (відтінки сірого), де кожний піксель представляє інтенсивність сірого кольору зі значеннями від 0 до 255.

Застосування – MNIST використовується як базовий набір даних для перевірки алгоритмів машинного навчання та глибинного навчання.

Він служить для оцінки та порівняння продуктивності моделей на задачах класифікації.

Працювати з MNIST можна, використовуючи різноманітні бібліотеки машинного навчання та обробки даних, такі як TensorFlow, PyTorch, Keras, Scikit-learn. Типовий процес включає завантаження набору даних, його передобробку (нормалізація інтенсивностей пікселів, поділ на тренувальну і тестову вибірки), конструювання моделі машинного навчання або глибинної нейронної мережі для розпізнавання цифр, тренування моделі на тренувальній вибірці і, нарешті, оцінювання ефективності моделі на тестовій вибірці.

Розглянемо основні етапи побудови моделі ML та аналіз її ефективності на прикладі розпізнавання рукописних цифр.

1) Команди для встановлення необхідних бібліотек. При встановленому Python на комп'ютері інсталяція додаткових бібліотек відбувається достатньо просто при застосуванні команди **pip** в терміналі.

pip – це система управління пакетами, яка використовується для встановлення та управління програмними пакетами, написаними мовою програмування Python. Пакети, які встановлюються за допомогою pip, зазвичай зберігаються в індексі пакетів Python (PyPI), який є репозиторієм програмного забезпечення для мови програмування Python.

Для подальшої роботи з обробкою зображень і побудови нейромереж потрібно в терміналі (можна викликати через команду **cmd**) виконати команду завантаження потрібної бібліотеки. Формат завантаження:

pip install «бібліотека», наприклад:

```
C:\Users\Admin>pip install numpy
```

Наведемо перелік потрібних завантажень:

pip install numpy

pip install tensorflow

pip install keras

pip install pillow

2) Імпорт бібліотек та набору даних. На початку роботи потрібно імпортувати всі необхідні модулі для навчання майбутньої моделі:

```
import numpy as np

from keras import models

from keras import layers

from tensorflow.keras.datasets import mnist

from keras.utils import to_categorical
```

Можна легко імпортувати набір даних і почати над цим працювати, оскільки бібліотека **Keras** уже містить багато наборів даних, і MNIST є одним із них. Викликаємо функцію **mnist.load_data(),** щоб отримати з

бібліотеки навчальні дані *train_images* з мітками *train_labels*, а також дані для тестування *test_images* з мітками *test_labels*:

(train_images, train_labels), (test_images, test_labels) = mnist.load_data().

Дізнаємося про формат даних, їх розмір за допомогою такої команди:

train_images.shape, test_images.shape, len(train_labels), len(test_labels).

Таким чином, маємо набір для навчання в кількості 60 000 цифр, для тестування (перевірки) якості моделі – 10 000 цифр, кожна з яких має розмірність 28 на 28 пікселів:

```
train_images.shape, test_images.shape, len(train_labels), len(test_labels)
```

```
((60000, 28, 28), (10000, 28, 28), 60000, 10000)
```

Можемо подивитися на перше значення цифри з тренувального набору train_images[0], щоб мати уяву про її вигляд. Це буде зображення цифри «5». Окрім того, нам знадобиться графічна бібліотека matplotlib для реалізації такої візуалізації:

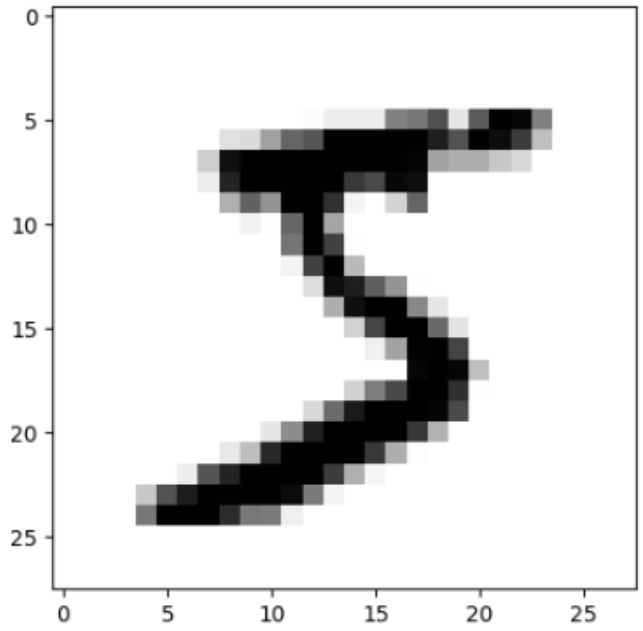
import matplotlib.pyplot as plt

digit = train_images[0]

plt.imshow(digit, cmap=plt.cm.binary)

plt.show()

```
digit = train_images[0]
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```



3) Побудова моделі простої, повністю з'єднаної (*Dense*) нейронної мережі. Для побудови такої моделі для класифікації цифр з датасету MNIST скористаємося деякими типовими функціями, які

дозволяють полегшити нашу роботу. Для цього наберемо такий фрагмент програми:

network = models.Sequential()

*network.add(layers.Flatten(input_shape=(28 * 28,))) # Додано Flatten шар*

network.add(layers.Dense(32, activation='relu'))

network.add(layers.Dense(10, activation='softmax'))

Keras – це високорівнева нейромережева API, написана на Python і здатна працювати поверх TensorFlow, Microsoft Cognitive Toolkit, R, Theano або PlaidML. Це робить Keras дуже гнучкою і зручною для експериментів з глибинним навчанням.

В наведеному фрагменті створюється об'єкт моделі **Sequential**, який дозволить нам послідовно додавати шари:

```
network = models.Sequential()
```

models.Sequential() в Keras використовується для створення лінійного стеку шарів нейронної мережі. Тобто ви можете додавати шар за шаром у вашу модель послідовно, де вихід одного шару є входом для наступного. Це дуже інтуїтивно зрозумілий і простий у використанні спосіб побудови моделей.

```
network.add(layers.Flatten(input_shape=(28 * 28,)))  # Додано Flatten шар
```

У цьому рядку до моделі додається перший **Flatten** шар. Цей шар у TensorFlow/Keras перетворює багатовимірні дані на одновимірний масив. Це важливо, оскільки повнозв'язний шар (**Dense**), який використовується далі у моделі, очікує дані у вигляді одновимірного вектора.

input_shape=(28 * 28,) вказує, що кожне вхідне зображення буде попередньо перетворено в одновимірний масив з 784 елементів (тому що оригінальні зображення MNIST мають розмір 28x28 пікселів).

```
network.add(layers.Dense(32, activation='relu'))
```

Шар **Dense** у Keras – це повнозв'язний шар нейронної мережі, що є одним із найбільш фундаментальних та часто використовуваних типів шарів у глибинному навчанні. Шар Dense інтегрує кожний вхідний елемент з кожним виходом за допомогою лінійної комбінації входів, ваг та зміщень, а потім застосовує нелінійну активаційну функцію до результату. В цьому випадку додається повністю з'єднаний шар (**Dense – fully-connected layer**) з усього 32 нейронами (кількість вузлів). Аргумент **activation='relu'** вказує на використання ReLU (rectified linear unit) як функції активації

```
network.add(layers.Dense(10, activation='softmax'))
```

Другий шар є вихідним шаром мережі, який має 10 нейронів, що відповідає кількості класів (цифр) в датасеті MNIST (від 0 до 9). Функція активації **softmax** використовується для отримання розподілу ймовірностей вихідних класів, що робить цей шар ідеальним для задач класифікації.

Побудована таким чином модель є простою, але ефективною і застосовується для розпізнавання рукописних цифр. Вона вміє вивчати складні шаблони у вхідних даних (зображеннях цифр) завдяки великій кількості нейронів і наявності нелінійної активаційної функції ReLU. Після тренування на великій кількості прикладів зображень і відповідних міток ця модель зможе з достатньою точністю класифікувати нові, раніше не видимі зображення цифр.

4) Компіляція моделі

network.compile(optimizer='rmsprop',

loss='categorical_crossentropy',

metrics=['accuracy'])

Метод **compile** у Keras/TensorFlow використовується для конфігурації процесу навчання моделі перед її тренуванням. Це включає визначення **оптимізатора, функції втрат (loss function) та метрик**, за якими буде вимірюватися ефективність моделі під час навчання та тестування. Наведемо опис параметрів компіляції:

optimizer='rmsprop' – оптимізатор визначає алгоритм, який буде використовуватися для оновлення ваг мережі в процесі тренування. Підходить для багатьох типів задач. Він автоматично налаштовує швидкість навчання в процесі тренування, що робить його ефективним і широко використовуваним вибором;

loss='categorical_crossentropy' – функція втрат визначає, як мережа вимірює різницю між своїми передбаченнями та реальними мітками даних під час тренування. categorical_crossentropy є стандартною функцією втрат для задач багатокласової класифікації, де мітки класів представлені в форматі "one-hot encoding". Ця функція добре підходить для ситуацій, коли кожний приклад може належати лише одному класу;

metrics=['accuracy'] – метрики використовуються для моніторингу ефективності моделі під час тренування та тестування. Вказуючи accuracy як метрику, ви вказуєте Keras вимірювати і виводити точність класифікації – відсоток правильно класифікованих прикладів – під час тренування та оцінки моделі. Це дає змогу легко відстежувати, як добре модель впорається з задачею класифікації на кожному етапі тренування.

У сукупності ці параметри дозволяють налаштувати ключові аспекти тренувального процесу моделі, забезпечуючи необхідні інструменти для ефективного навчання моделі під конкретну задачу.

5) Підготовка даних зображень для навчання та тестування нейронної мережі

*train_images = train_images.reshape((60000, 28 * 28))*

train_images = train_images.astype('float32') / 255

*test_images = test_images.reshape((10000, 28 * 28))*

test_images = test_images.astype('float32') / 255

Датасет MNIST складається з зображень рукописних цифр, кожне з яких має розмір 28x28 пікселів. Перша операція reshape змінює форму масивів зображень з 2D (28x28 пікселів) в 1D (784 пікселі)

*train_images = train_images.reshape((60000, 28 * 28)),*

*test_images = test_images.reshape((10000, 28 * 28)),*

перетворюючи кожне зображення в лінійний вектор. Це необхідно, тому що вхідний шар нейронної мережі, яку ми створюємо, очікує одновимірний вектор даних для кожного прикладу. 60 000 і 10 000 відповідають кількості зображень у навчальному та тестовому наборах даних відповідно.

Нормалізація зображень

train_images = train_images.astype('float32') / 255

test_images = test_images.astype('float32') / 255

Після зміни форми зображень код перетворює тип даних пікселів з цілочисельного (int) у дійсний (float32). Це робиться для того, щоб можна було виконати наступний крок – нормалізацію. Значення пікселів у зображеннях MNIST варіюються від 0 до 255, де 0 відповідає чорному кольору, а 255 – білому. Нормалізація, тобто поділ кожного значення пікселя на 255, переводить ці значення в діапазон від 0 до 1. Нормалізація даних є стандартною практикою в машинному навчанні, оскільки вона допомагає прискорити тренування мережі та досягти кращої збіжності.

6) Перетворення міток класів (labels) у датасеті з цілочисельного формату в бінарний формат класів

train_labels = to_categorical(train_labels)

test_labels = to_categorical(test_labels)

Ці рядки коду використовуються для перетворення міток класів (labels) у датасеті з цілочисельного формату в бінарний формат класів, що є стандартною процедурою підготовки даних при роботі з задачами класифікації у глибинному навчанні. Таке перетворення часто називають "**one-hot encoding**". В контексті датасету MNIST, де мітки являють собою цифри від 0 до 9, "one-hot encoding" перетворює цілочисельну мітку класу в 10-вимірний вектор, де індекс, що відповідає значенню мітки, має значення 1, а всі інші елементи дорівнюють 0.

Наприклад, якщо мітка класу для конкретного зображення є '3', то після перетворення в "one-hot" вектор ми отримаємо:

```
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
```

7) Навчання моделі «network» на тренувальному наборі

Навчання моделі відбувається за таким фрагментом коду:

history = network.fit(train_images, train_labels, epochs=5,

batch_size=32, validation_data=(test_images, test_labels))

Результатом його виконання є набір ітерацій (епох), протягом яких відбувається обрахунок коефіцієнтів моделі:

```
Epoch 1/5
1875/1875 [==============================] - 11s 5ms/step - loss: 0.3531 - accuracy: 0.9003 - val_loss: 0.2151 - val_accuracy:
0.9374
Epoch 2/5
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1976 - accuracy: 0.9429 - val_loss: 0.1705 - val_accuracy:
0.9508
Epoch 3/5
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1602 - accuracy: 0.9550 - val_loss: 0.1543 - val_accuracy:
0.9548
Epoch 4/5
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1411 - accuracy: 0.9602 - val_loss: 0.1498 - val_accuracy:
0.9584
Epoch 5/5
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1272 - accuracy: 0.9639 - val_loss: 0.1439 - val_accuracy:
0.9603
```

Розглянемо призначення функцій і параметрів для побудови моделі:

функція **fit** у Keras/TensorFlow використовується для тренування моделі на заданих вхідних даних, тобто для адаптації ваг нейронної мережі до навчального датасету;

train_images – це вхідні дані для навчання моделі. У нашому випадку це набір зображень рукописних цифр з датасету MNIST, які були перетворені в масиви і нормалізовані для ефективного навчання мережі;

train_labels – це мітки класів для навчального датасету, які вже були перетворені в бінарний формат класів (one-hot encoding) – див. п. 6. Ці мітки використовуються моделлю для визначення, як добре вона виконує свою задачу класифікації під час тренування, і для корегування ваг відповідно до помилки;

epochs=5 – цей параметр визначає, за скільки епох буде виконано процес тренування. Одна епоха означає один прохід всього навчального датасету через мережу в обох напрямках (вперед і назад). Чим більше епох, тим модель має більше можливостей вивчити дані, але існує ризик перенавчання (**overfitting**), коли модель надто добре адаптується до навчальних даних і погано справляється з новими даними;

batch_size=32 – цей параметр вказує на розмір пакета (batch size), тобто на кількість зразків даних, які будуть оброблені за один раз перед оновленням ваг мережі. Використання пакетів допомагає ефективніше використовувати апаратні ресурси, а також може допомогти зі збіжністю процесу тренування;

validation_data=(test_images, test_labels) – це дані, які використовуються для перевірки моделі після кожної епохи навчання. Це допомагає відстежувати, як модель працює на невідомих даних, тобто ви можете бачити, чи виникає перенавчання **(overfitting**).

В результаті виклику **.fit()** отримуємо об'єкт **history**, який містить історію процесу тренування. Цей об'єкт містить деталізовані дані про втрати (**loss**) та метрики (наприклад точність) моделі, обчислені після кожної епохи для навчальних та перевірочних даних. Ми можемо використовувати ці дані для візуалізації процесу навчання моделі, аналізу динаміки зміни точності та втрат, що дуже корисно для тонкої настройки та вдосконалення моделі.

Візуалізацію процесу навчання можемо реалізувати такими функціями:

 # Візуалізація результатів навчання

plt.plot(history.history['accuracy'], label='accuracy')

plt.plot(history.history['val_accuracy'], label='val_accuracy')

plt.title('Model accuracy')

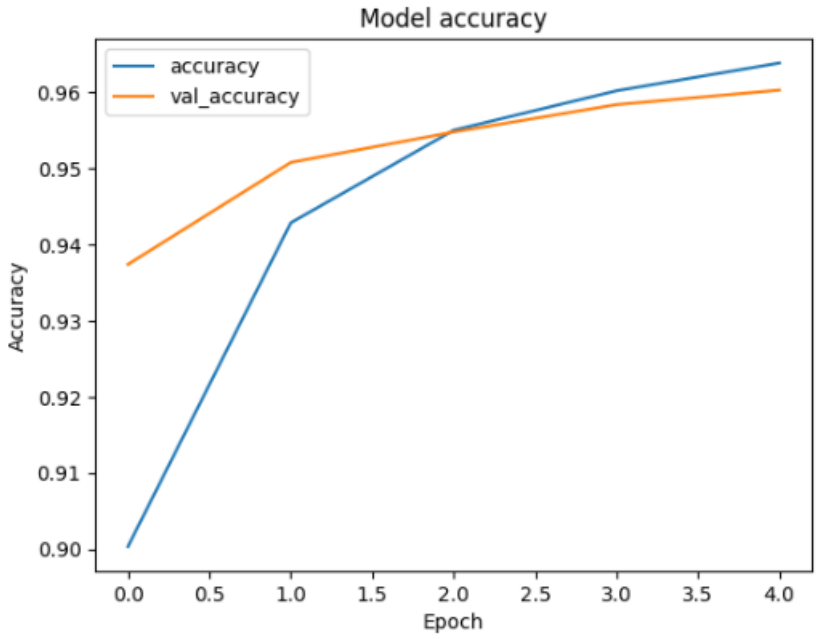
plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(loc='upper left')

plt.show(),

що дасть такий графік:



Таким чином, виклик *network.fit(train_images, train_labels, epochs=5, batch_size=32, validation_data=(test_images, test_labels))* ініціює процес тренування моделі на зображеннях і мітках, відбувається п'ять

повних проходів по датасету (епох) і обробляються дані пакетами по 32 зразки в кожному. Цей процес включає прямий прохід (де модель робить передбачення на основі вхідних даних), обчислення втрат (за допомогою функції втрат, визначеної при компіляції моделі), зворотний прохід (де вираховуються градієнти функції втрат щодо ваг моделі) і оновлення ваг моделі з використанням оптимізатора (також визначеного при компіляції).

8) Оцінка ефективності моделі

test_loss, test_acc = network.evaluate(test_images, test_labels)

print('test_loss:', test_loss, 'test_acc:', test_acc)

Метод **evaluate** використовується для оцінки ефективності навченої моделі нейронної мережі на тестовому наборі даних. Цей метод повертає значення втрати (loss) та метрики (у нашому випадку точності – accuracy), які були вказані при компіляції моделі. Ось що робить кожна частина цього коду:

test_loss, test_acc = network.evaluate(test_images, test_labels): **network** – це назва навченої моделі. Ми викликаємо метод **evaluate**, передаючи йому test_images (зображення з тестового набору даних) та test_labels (правильні мітки для цих зображень). Метод оцінює модель, використовуючи тестові дані, і повертає значення втрати та точності, які зберігаються у змінних test_loss та test_acc.

print('test_loss:', test_loss, 'test_acc:', test_acc): після отримання результатів оцінки цей рядок коду друкує значення втрати та точності. Втрата (loss) показує, наскільки передбачення моделі відрізняються від реальних міток, тоді як точність (accuracy) відображає відсоток зображень, які були правильно класифіковані.

Призначення цього фрагмента коду полягає в тому, щоб забезпечити кінцеву оцінку того, наскільки добре модель працює з новими, не видимими раніше даними, які не використовувались під час тренування. Це дозволяє оцінити загальну здатність моделі до узагальнення та її практичну придатність. У випадку з набором даних MNIST, де завдання полягає в розпізнаванні рукописних цифр, висока точність на тестовому наборі свідчить про те, що модель добре справляється з класифікацією цифр на зображеннях, які вона не бачила під час навчання.

Результатом запуску цього фрагмента кода буде точність (accuracy)=0.96, що свідчить про високий показник ефективнгості моделі (помилка в прогнозуванні на невідомому наборі становить менше 4%).

```
test_loss, test_acc = network.evaluate(test_images, test_labels)
print('test_loss:', test_loss, 'test_acc:', test_acc)
```

```
313/313 [==============================] - 1s 4ms/step - loss: 0.1439 - accuracy: 0.9603
test_loss: 0.14385534822940826 test_acc: 0.9603000283241272
```

9) Виведення загальної інформації про модель

Метод **summary()** у Keras використовується для виведення загальної інформації про модель нейронної мережі, включаючи кількість шарів, типи цих шарів, форму вхідних і вихідних даних для кожного шару, а також загальну кількість параметрів (ваг), які будуть навчатися під час тренування:

network.summary()

Це корисно для отримання швидкого огляду архітектури моделі та для перевірки, що всі шари мають очікувані розміри вхідних і вихідних даних:

```
network.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_10 (Flatten)        (None, 784)               0

 dense_28 (Dense)            (None, 32)                25120

 dense_29 (Dense)            (None, 10)                330

=================================================================
Total params: 25450 (99.41 KB)
Trainable params: 25450 (99.41 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

10) Передбачення рукописних цифр на основі моделі машинного навчання

y_pred = network.predict(train_images)

В результаті виконання такого методу передбачення «*.predict()*» в наборі даних «*train_images*» отримаємо вектор *y_pred,* який містить результат. Так, для рукописної цифри, яка аналізувалася на початку програми, можемо вивести результат її аналізу:

```
y_pred[0]
```

```
array([8.1187460e-11, 4.6883741e-08, 6.7842720e-10, 6.2646599e-05,
       2.2166227e-22, 9.9993730e-01, 7.2664097e-13, 3.0464870e-11,
       1.0347345e-09, 8.0481889e-12], dtype=float32)
```

```
pred_result = np.argmax(y_pred[0], axis=0)
print('Результат передбачення - ',  pred_result)
```

```
Результат передбачення -  5
```

де найбільше значення буде відповідати найбільшій імовірності відповідної цифри. Отже, отримаємо результат передбачення – цифра «5», що відповідає дійсності.

Таким чином, повний код програми, яка розглядалася вище, має вигляд:

Mnist

```
import numpy as np
from keras import models
from keras import layers
from tensorflow.keras.datasets import mnist
from keras.utils import to_categorical
```

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
train_images.shape, test_images.shape, len(train_labels), len(test_labels)
```

```
digit = train_images[0]
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

Model 1 - Sequential ¶

```
network = models.Sequential()
network.add(layers.Flatten(input_shape=(28 * 28,)))  # Додано Flatten шар
network.add(layers.Dense(32, activation='relu'))
network.add(layers.Dense(10, activation='softmax'))
```

```
network.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

```
history = network.fit(train_images, train_labels, epochs=5, batch_size=32,
                      validation_data=(test_images, test_labels))
```

```
# Візуалізація результатів навчання
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='upper left')
plt.show()
```

```
test_loss, test_acc = network.evaluate(test_images, test_labels)
print('test_loss:', test_loss, 'test_acc:', test_acc)
```

```
network.summary()
```

5.8. Практичне застосування ML моделі для розпізнавання рукописних цифр

На основі **моделі ML network** передбачимо значення зображень рукописних цифр з набору MNIST, для чого застосуємо функцію:

y_pred = network.predict(train_images)

Використання методу **predict** моделі Keras, як у нашому прикладі *network.predict(train_images),* дозволяє робити передбачення на основі навчальних даних. У нашому випадку **train_images** є набором зображень, на яких модель уже була натренована, але цей метод часто використовується для нових, раніше не відомих даних.

Коли ви викликаєте **network.predict(train_images)**, модель обробляє кожне зображення з набору **train_images** та видає вектор передбачень для кожного зображення. У контексті класифікації цифр MNIST, де наша модель має вихідний шар з 10 нейронами (кожен відповідає одній із 10 цифр від 0 до 9), **y_pred** буде містити масив розмірності (60 000, 10) (припускаючи, що у нас є 60 000 зображень у train_images). Кожний рядок у **y_pred** буде вектором з 10 значень, які представляють імовірності того, що відповідне зображення належить до кожного з 10 класів. Значення у цьому векторі в сукупності становлять 1.

Для того щоб визначити, до якого класу належить кожне зображення, можна використати **np.argmax(y_pred, axis=1)**, що поверне індекс (від 0 до 9) найбільшого елемента у кожному векторі передбачень, який відповідно буде представляти передбачену цифру. Вибір індексу **y_pred** (наприклад [0]) дозволить передбачити зображення з цим індексом у наборі даних. У цьому випадку це буде цифра «5», яка і викликалася в п. 2:

```
y_pred[0]
```

```
array([3.0412441e-18, 1.7901227e-16, 6.5297160e-15, 9.7498814e-05,
       3.7389118e-25, 9.9990249e-01, 1.7765483e-19, 8.1325566e-17,
       1.7958322e-14, 4.5872551e-13], dtype=float32)
```

```
pred_result = np.argmax(y_pred[0], axis=0)
print('Результат передбачення - ',  pred_result)
```

```
Результат передбачення -  5
```

Це дозволяє оцінити, наскільки добре модель вчилася на навчальному наборі, хоча для перевірки загальної здатності моделі до узагальнення краще використовувати набір даних, який не брав участі у тренуванні, наприклад тестовий набір.

Тепер скористаємося цією моделлю, щоб розпізнати власноруч написану цифру на дошці, папері тощо. Для цього насамперед сфотографуємо таку цифру, наприклад на смартфон, і запям’ятаємо файл, який будемо обробляти. Нехай це буде зображення, яке схоже на цифру **«2».** Спробуємо його розпізнати за допомогою отриманої моделі.

Для початку інсталюємо потрібні бібліотеки для візуалізації фотографії

import cv2

import matplotlib.pyplot as plt

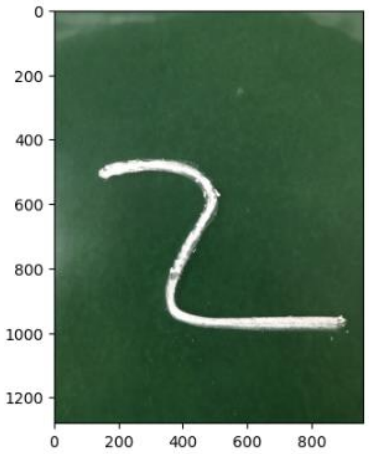
import matplotlib.image as mpimg

img=mpimg.imread('2_class.jpg')

imgplot = plt.imshow(img)

і отримаємо зображення, яке зберігалося в папці:

```
import cv2
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
img=mpimg.imread('2_class.jpg')
imgplot = plt.imshow(img)
```



За допомогою функції *img.shape*

```
img.shape
```

```
(1280, 960, 3)
```

бачимо, що зображення є кольоровим (3 канали) і достатньо великого розміру (1280 на 960), яке відрізняється від тих зображень, на яких тренувалася модель. Перетворимо таке зображення в інше зображення в градаціях сірого (*gray_img*) і з заданим розміром 28 на 28 пікселей, а потім і у чорно-білий формат (*black_white_img*):

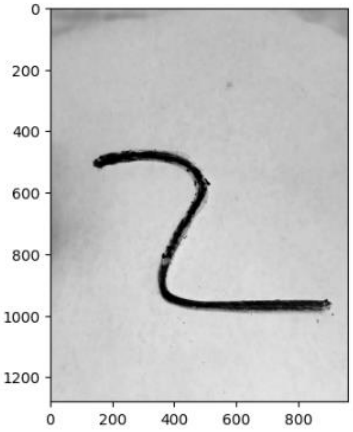
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

resized_img = cv2.resize(gray_img, (28, 28))

_, black_white_img = cv2.threshold(resized_img, 127, 255, cv2.THRESH_BINARY)

plt.imshow(gray_img, cmap=plt.cm.binary)

```
# Перетворення зображення у монохромний режим
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Зміна розміру зображення до 28x28 пікселів
resized_img = cv2.resize(gray_img, (28, 28))
# Порогова обробка (приклад порогу 127)
_, black_white_img = cv2.threshold(resized_img,
                    127, 255, cv2.THRESH_BINARY)
plt.imshow(gray_img, cmap=plt.cm.binary)
```



Порогова обробка перетворює монохромне зображення у бінарне (чорно-біле), де пікселі, які мають значення вище порога (в цьому випадку 127), стають білими (255), а всі інші – чорними (0). Це допомагає виділити важливі аспекти зображення, спрощуючи подальший аналіз.

Зменшене зображення зберігається у змінній *resized_img*, розмір його можна оцінити функцією *resized_img*.shape, і чорно-біле зображення можна побачити при застосуванні функції

plt.imshow(black_white_img, cmap=plt.cm.binary).

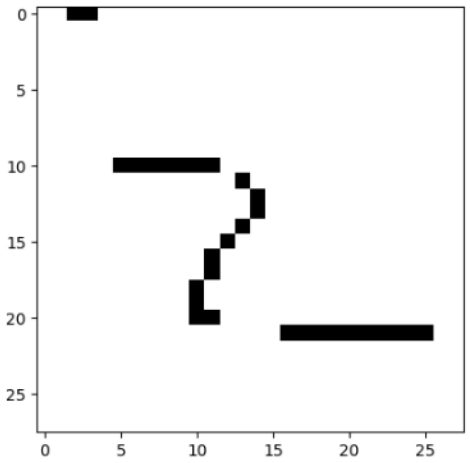
Кожен із цих кроків є частиною типового пайплайну передобробки зображення, особливо коли йдеться про підготовку даних для нейронних мереж, які розпізнають рукописний текст або виконують інші завдання комп'ютерного зору.

```
resized_img.shape
```

```
(28, 28)
```

```
plt.imshow(black_white_img, cmap=plt.cm.binary)
```

```
<matplotlib.image.AxesImage at 0x2afbf274e20>
```



Тепер перетворимо це зображення у формат, який підходить для подачі в нейронну мережу, змінюючи його форму та тип даних:

*images = black_white_img.reshape((1, 28 * 28))*

images = images.astype('float64')#/255

images.shape

Цей фрагмент коду змінює форму зображення з двовимірного масиву (який має розмір 28x28 пікселів) в одновимірний масив розміром 784. Цей крок необхідний, оскільки багато архітектур нейронних мереж очікують на вхід даних у вигляді одновимірного масиву або вектора. Додавання виміру 1 на початку форми масиву підготовлює дані у форматі, який очікує багато реалізацій нейронних мереж, ідентифікуючи цей масив як «пакет» з одним зразком:

```
images = black_white_img.reshape((1, 28 * 28))
images = images.astype('float64')#/255
images.shape
```

```
(1, 784)
```

Змінюючи тип даних на float64, ми готуємо зображення до подальшої обробки, яка може включати нормалізацію або інші вирази, що вимагають чисел з плаваючою комою. В контексті обробки зображень для нейронних мереж конвертація в float64 (хоча більшість задач вимагає меншої точності, наприклад float32) може бути корисною для подальшої нормалізації значень пікселів.

У цьому коді закоментоване ділення на 255, яке зазвичай використовується для нормалізації значень пікселів до діапазону [0, 1]. Ця нормалізація важлива для багатьох моделей машинного навчання, оскільки вона допомагає покращити процес навчання, забезпечуючи, що всі вхідні характеристики мають схожий масштаб.

Залишився останній етап, коли відбувається безпосереднє передбачення цифри за заданим зображенням. Для цього використовується метод *network.predict(images)*

y_pred_some_digit = network.predict(images)

на зміненому та підготовленому зображенні. Цей метод повертає вектор із ймовірностями того, що зображення належить до кожного з можливих класів. У нашому випадку, оскільки мережа навчена розпізнавати цифри від 0 до 9 на основі датасету MNIST, вектор **y_pred_some_digit** міститиме 10 значень, кожне з яких відповідає ймовірності того, що зображення належить до відповідної цифри. Результат виконання цього методу має вигляд:

```
y_pred_some_digit = network.predict(images)
y_pred_some_digit
```

```
1/1 [==============================] - 0s 46ms/step

array([[0., 0., 1., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

Для того щоб автоматизувати виведення результату передбачення, скористаємося методами

pred_result = np.argmax(y_pred_some_digit, axis=1),

де ефективно визначається, до якого класу (або в цьому випадку цифри) належить зображення згідно з передбаченням моделі. Коли вказано axis=1, це означає, що ми шукаємо індекс максимального значення по осі 1 (тобто по кожному рядку, якщо уявити y_pred_some_digit як матрицю, де кожний рядок представляє один пакет або зразок даних):

```
pred_result = np.argmax(y_pred_some_digit, axis=1)
print('Результат ПЕРЕДБАЧЕННЯ цифри - ',  pred_result)
```

```
Результат ПЕРЕДБАЧЕННЯ цифри -  [2]
```

Таким чином, видно, що результат передбачення – це цифра «2», що збігається з очікуваним рукописним варіантом.

Програмна реалізація застосування моделі ML для розпізнавання рукописних цифр наведена нижче:

Read image from file

```
import cv2
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
img=mpimg.imread('2_class.jpg')
imgplot = plt.imshow(img)
```

```
img.shape
```

```
# Перетворення зображення у чорно-білий режим (в монохром)
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Зміна розміру до 28x28 пікселів
resized_img = cv2.resize(gray_img, (28, 28))

# Порогова обробка для отримання чорно-білого зображення (приклад порогу 127)
_, black_white_img = cv2.threshold(resized_img, 127, 255, cv2.THRESH_BINARY)

plt.imshow(gray_img, cmap=plt.cm.binary)
```

```
resized_img.shape
```

```
plt.imshow(black_white_img, cmap=plt.cm.binary)
```

```
images = black_white_img.reshape((1, 28 * 28))
images = images.astype('float64')#/255
images.shape
```

```
y_pred_some_digit = network.predict(_images)
y_pred_some_digit
```

```
pred_result = np.argmax(y_pred_some_digit, axis=1)
print('Результат ПЕРЕДБАЧЕННЯ цифри - ',  pred_result)
```

5.9 Побудова CNN нейромережі для розпізнавання цифр на прикладі набору даних MNIST

Розглянемо інший підхід до обробки зображень на основі теоретичних відомостей, які були викладені в підрозділі 5.4, для побудови **нейронних мереж типу ConvNet/CNN,** які краще адаптовані для задач комп'ютерного зору.

Початкові етапи будуть подібними до тих, які розглядалися в підрозділі 5.7:

1) Імпорт бібліотек та набору даних. Для використання вбудованих в Python функцій імпортуємо необхідні модулі для навчання майбутньої моделі на основі **CNN**:

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to_categorical

Наведемо коротку інформацію про основне призначення бібліотек, які завантажуються:

import numpy as np – NumPy є основною бібліотекою для наукових обчислень у Python. Вона забезпечує підтримку великих, багатовимірних масивів і матриць разом із великим набором високорівневих математичних функцій для роботи з цими масивами. У контексті навчання нейронних мереж NumPy часто використовується для виконання різноманітних математичних операцій на даних перед їх подачею в модель;

import tensorflow as tf – TensorFlow є потужною бібліотекою відкритого коду для чисельних обчислень, що спеціалізується на машинному навчанні та глибинному навчанні. Вона розроблена і підтримується Google і є однією з найпопулярніших бібліотек у цій галузі. TensorFlow використовується для створення, тренування та валідації глибинних нейронних мереж з можливістю автоматичного диференціювання;

from tensorflow.keras import layers, models – layers та models є підмодулями у Keras, високорівневим API для TensorFlow;

l**ayers** використовується для створення різних шарів у нейронних мережах, таких як конволюційні шари, пулінг-шари, повнозв'язні шари тощо;

models дозволяє створювати архітектури моделей, зокрема послідовні моделі (**Sequential**) або складніші графові структури за допомогою функціонального API;

from tensorflow.keras.datasets import mnist – цей модуль надає прямий доступ до популярного набору даних MNIST, який містить зображення рукописних цифр;

from tensorflow.keras.utils import to_categorical – функція to_categorical використовується для перетворення числових міток (наприклад від 0 до 9 для міток класів MNIST) у бінарний векторний формат (one-hot encoding). Це дуже зручно для категорійної класифікації, де кожний вхід може належати тільки одному з кількох класів.

2) Завантаження та підготовка даних MNIST

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

Як і в попередньому розділі, дізнаємося про формат даних, їх розмір за допомогою команди:

train_images.shape, test_images.shape, len(train_labels), len(test_labels),

результатом якої буде виведено формат зображення

```
train_images.shape, test_images.shape, len(train_labels), len(test_labels)
((60000, 28, 28), (10000, 28, 28), 60000, 10000)
```

,

де маємо набір для навчання – 60 000 цифр, а для тестування якості майбутньої моделі – 10 000 цифр, кожна з яких має розмірність 28 на 28 пікселів.

Так само дізнаємося, якою є перша цифра тренувального набору через застосування train_images[0]. При завантаженні графічної бібліотеки matplotlib є можливість візуалізувати цю цифру:

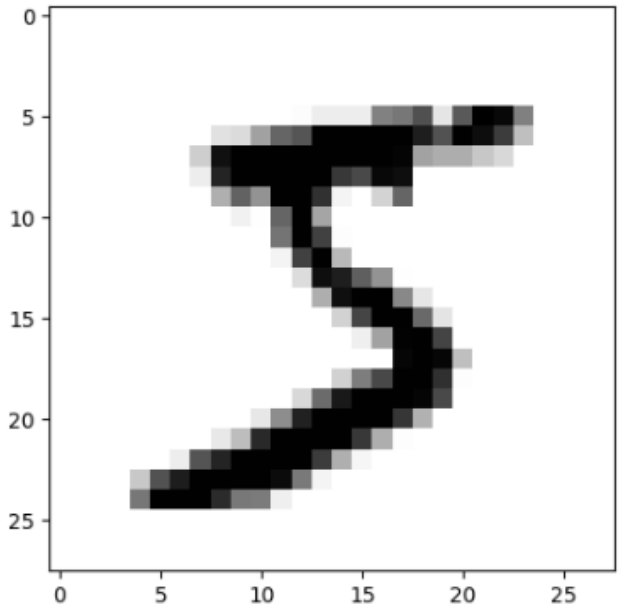
import matplotlib.pyplot as plt

digit = train_images[0]

plt.imshow(digit, cmap=plt.cm.binary)

plt.show()

```
digit = train_images[0]
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```



3) Нормалізація зображень. Цей етап здійснює нормалізацію зображень з набору даних MNIST, що є важливим кроком у передобробці даних для машинного навчання, особливо при роботі з нейронними мережами.

train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255

Перетворення форми зображень (reshape): зображення MNIST початково представлені у вигляді масивів розміром 60 000 (кількість зображень у навчальному наборі) x 28 (висота) x 28 (ширина). Однак для використання в конволюційних нейронних мережах нам потрібно додати ще один вимір, який вказує на кількість каналів зображення.

В MNIST зображення є відтінками сірого, тому мають тільки один канал. Тому масиви перетворюються на форму (60000, 28, 28, 1) для навчальних зображень і (10000, 28, 28, 1) для тестових зображень, де 1 вказує на кількість каналів.

Зміна типу даних і нормалізація (astype('float32') / 255): зображення в MNIST подані в цілочисельному форматі (0–255), де кожне число

представляє інтенсивність пікселя. Для ефективнішого навчання моделей нейронних мереж бажано перетворити ці значення в діапазон від 0 до 1. Це полегшує процес навчання, оскільки більшість функцій активації (наприклад сигмоїд або тангенс гіперболічний) оптимізовані для роботи з малими вхідними даними.

Масив змінюється на тип float32, що дозволяє зберігати дробові числа, після чого відбувається поділ кожного значення пікселя на 255, щоб нормалізувати його до діапазону [0, 1].

4) Кодування міток. Цей блок коду виконує кодування міток (лейблів) для тренувального та тестового наборів даних, використовуючи метод **to_categorical** з бібліотеки Keras. Цей процес також відомий як **one-hot encoding** і є ключовим для підготовки міток при використанні нейронних мереж для задач класифікації:

train_labels = to_categorical(train_labels)

test_labels = to_categorical(test_labels)

Метод **to_categorical** перетворює масив цілочисельних міток на бінарну матрицю. Наприклад, якщо у вас є мітка 3 (для класу, що відповідає цифрі 3 в MNIST), вона перетворюється на масив, де всі елементи дорівнюють 0, крім індексу, що відповідає числу 3, де буде 1.

У випадку MNIST, де є 10 класів (цифри від 0 до 9), мітка 3 буде представлена як [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].

5) Створення моделі. Цей блок коду виконує створення конволюційної нейронної мережі (CNN) за допомогою Keras, яка використовується для класифікації зображень:

model = models.Sequential()

model.add(layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model=models.Sequential() – ініціалізація моделі, відбувається створення екземпляра моделі Sequential, що дозволяє додавати шари послідовно, де вихід одного шару автоматично стає входом наступного.

Створення першого шару Conv2D:

model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape= (28, 28, 1))) – додає перший конволюційний шар, що містить 32 фільтри з розмірами ядра 3x3. Функція активації **'relu'** використовується для додання нелінійності обробки. Параметр **input_shape** вказує форму вхідних даних, тобто зображень з розміром 28x28 пікселів з 1 каналом (чорно-білі).

Наступний шар – MaxPooling2D:

model.add(layers.MaxPooling2D((2, 2))) – додає шар пулінгу з використанням максимального значення (max pooling), який зменшує

розміри вхідної карти ознак в два рази по кожному виміру (з 2x2 області береться максимум).

Шар – Conv2D:

model.add(layers.Conv2D(64, (3, 3), activation='relu')) – додає другий конволюційний шар з 64 фільтрами. Тут не вказується розмір вхідних даних, оскільки він автоматично визначається з виходу попереднього шару.

Знову шар – MaxPooling2D:

model.add(layers.MaxPooling2D((2, 2))) – ще один шар максимального пулінгу для додаткового зменшення просторових розмірів вхідної карти ознак.

Наступний шар – Conv2D:

model.add(layers.Conv2D(64, (3, 3), activation='relu')) – третій конволюційний шар з 64 фільтрами, який надалі обробляє карти ознак, отримані після попередніх шарів.

6) Додавання Dense шарів на кінці нейромережі. Цей блок коду завершує структуру конволюційної нейронної мережі, додаючи повнозв'язні (dense) шари для класифікації:

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

model.add(layers.Flatten()) – цей шар перетворює багатовимірні карти ознак, які були сформовані попередніми конволюційними та пулінговими шарами, в одновимірний вектор. Це необхідно, оскільки повнозв'язні шари вимагають вхідних даних у вигляді одновимірного масиву.

model.add(layers.Dense(64, activation='relu')) – додається повнозв'язний шар із 64 нейронами. Функція активації 'relu' знову ж таки використовується для додавання нелінійності до обчислень. Цей шар приймає векторизовані дані з попереднього Flatten шару і проводить набір лінійних та нелінійних трансформацій.

model.add(layers.Dense(10, activation='softmax')) – останній шар моделі, який складається з 10 нейронів відповідно до кількості класів у задачі класифікації (для MNIST це цифри від 0 до 9). Функція активації 'softmax' використовується для того, щоб перетворити виходи шару в ймовірності, що вказують на те, до якого класу найбільше належить кожний приклад.

7) Компіляція моделі з назвою «model»

```
model.compile(optimizer='adam',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

optimizer='adam' – використання оптимізатора Adam. Adam є популярним вибором оптимізатора, оскільки він комбінує переваги двох інших оптимізаторів – **AdaGrad і RMSProp**. Він ефективно налаштовує швидкість навчання для кожного параметра, використовуючи оцінки перших і других моментів градієнтів.

loss='categorical_crossentropy' – використання кросс-ентропії як функції втрат для багатокласової класифікації. Кросс-ентропія порівнює розподіл ймовірностей, який виходить з моделі, з реальним розподілом міток і намагається мінімізувати розбіжності між ними.

metrics=['accuracy'] – визначення точності як метрики для відстежування прогресу під час навчання. Точність розраховується як відсоток правильно класифікованих образів від загальної кількості образів, що дозволяє легко визначити, наскільки добре модель виконує своє завдання на даних для навчання та перевірки.

8) Тренування моделі – виконує тренування моделі на навчальному наборі даних:

model.fit(train_images, train_labels, epochs=5, batch_size=32, validation_split=0.1)

train_images – це набір зображень, який використовується для тренування моделі. Зображення вже були передопрацьовані і нормалізовані.

train_labels – це мітки класів для навчального набору зображень, які також були перетворені в категоріальний формат.

epochs=5 – вказує кількість епох тренування. Одна епоха означає, що кожний зразок у навчальному наборі даних пройшов (пряме та зворотне поширення) через модель один раз. В цьому випадку процес тренування відбудеться п’ять разів для кожного зразка.

batch_size=32 – кількість зразків, які будуть оброблятися за одну ітерацію алгоритму тренування. У цьому випадку модель використовує 32 зображення для вирахування градієнта та оновлення ваг в одній ітерації перед переходом до наступної партії.

validation_split=0.1 – вказує на те, що 10% даних з навчального набору слід використовувати як валідаційний набір. Ці дані не використовуються для тренування моделі, а замість цього служать для перевірки продуктивності моделі після кожної епохи, дозволяючи змоніторити і запобігти перенавчанню.

Результатом виконання цієї функції є набір ітерацій (епох), протягом яких відбувається обрахунок коефіцієнтів моделі:

```
Epoch 1/5
1688/1688 [==============================] - 89s 51ms/step - loss: 0.1545 - accuracy: 0.9517 - val_loss: 0.0495 - val_accuracy:
0.9862
Epoch 2/5
1688/1688 [==============================] - 83s 49ms/step - loss: 0.0492 - accuracy: 0.9851 - val_loss: 0.0400 - val_accuracy:
0.9882
Epoch 3/5
1688/1688 [==============================] - 83s 49ms/step - loss: 0.0357 - accuracy: 0.9890 - val_loss: 0.0374 - val_accuracy:
0.9897
Epoch 4/5
1688/1688 [==============================] - 85s 50ms/step - loss: 0.0265 - accuracy: 0.9916 - val_loss: 0.0436 - val_accuracy:
0.9893
Epoch 5/5
1688/1688 [==============================] - 86s 51ms/step - loss: 0.0218 - accuracy: 0.9928 - val_loss: 0.0274 - val_accuracy:
0.9928
```

У цьому випадку отримали кращі результати щодо якості моделі (val_accuracy: 0.9928), порівнюючи з попередньою нейромережею, хоча такий результат є відносним і залежить від гіперпараметрів мереж.

9) Оцінка моделі. Використовується для оцінки точності і втрат моделі на тестовому наборі даних після її тренування:

test_loss, test_acc = model.evaluate(test_images, test_labels)

print(f"Точність на тестових даних: {test_acc:.4f}")

model.evaluate – ця функція приймає тестові зображення та мітки як вхід і проводить передбачення моделі на цих даних для оцінки її продуктивності. В результаті повертаються втрати (**loss**) і точність (**accuracy**).

test_images – це масив зображень, який використовується для тестування моделі. Ці зображення також були нормалізовані, як і навчальні зображення.

test_labels – це мітки для тестових зображень у категоріальному форматі.

test_loss, test_acc – змінні, в які зберігаються значення втрат і точності, відповідно, отримані після оцінювання моделі. Втрати показують, наскільки передбачення моделі відрізняються від дійсних міток, тоді як точність показує відсоток правильно класифікованих зразків.

10) model.summary() – TensorFlow дозволяє отримати підсумок структури вашої нейронної мережі, включно з кількістю параметрів, формою входу/виходу та деталями про кожний шар.

11) y_pred = model.predict(train_images) – у TensorFlow є можливість отримати прогнозовані виходи моделі для введених даних. У цьому випадку отримуєте прогнози для зображень **train_images.**

Таким чином, повний код програми, яка реалізує CNN нейромережу для розпізнавання рукописних цифр, має вигляд:

Mnist - CNN

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
```

```
# Завантаження та підготовка даних MNIST
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
train_images.shape, test_images.shape, len(train_labels), len(test_labels)
```

```
((60000, 28, 28), (10000, 28, 28), 60000, 10000)
```

```
digit = train_images[0]
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

```
# Нормалізація зображень
train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255
```

```
# Кодування міток
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

```
# Створення моделі
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

```
# Додавання Dense шарів на кінці
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

```
# Компіляція моделі
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

```
# Тренування моделі
model.fit(train_images, train_labels, epochs=5, batch_size=32, validation_split=0.1)
```

```
# Оцінка моделі
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Точність на тестових даних: {test_acc:.4f}")
```

```
313/313 [==============================] - 6s 17ms/step - loss: 0.0296 - accuracy: 0.9912
Точність на тестових даних: 0.9912
```

```
model.summary()
```

```
Total params: 93322 (364.54 KB)
Trainable params: 93322 (364.54 KB)
Non-trainable params: 0 (0.00 Byte)
```

5.10. Практичне застосування моделі CNN для розпізнавання рукописних цифр

Практичне застосування отриманої моделі **CNN** нейромережі буде збігатися з етапами, які надані в підрозділі 5.8. Основна відмінність буде полягати в тому, що в цьому випадку ми отримали іншу модель, якій дали назву «**model**».

На основі моделі ML «**model**» передбачимо значення зображень рукописних цифр з набору MNIST, для чого застосуємо функцію:

y_pred = model.predict(train_images)

Легко бачити, що перше зображення відповідає цифрі «5», про що було сказано раніше:

```
y_pred =  model.predict(train_images)
```

```
1875/1875 [==============================] - 35s 19ms/step
```

```
y_pred[0]
```

```
array([2.8060847e-09, 7.8878067e-08, 1.8447245e-09, 1.4751495e-02,
       2.9216798e-10, 9.8523396e-01, 1.0048425e-07, 7.4469028e-08,
       2.2528020e-06, 1.1945281e-05], dtype=float32)
```

```
pred_result = np.argmax(y_pred[0], axis=0)
print('Результат передбачення - ',  pred_result)
```

```
Результат передбачення -  5
```

Для того щоб передбачати інші цифри з зображеннь за допомогою отриманої моделі **CNN**, необхідно виконати такі самі операції, які описані в підрозділі 5.8. Відмінність буде полягати у використанні іншої отриманої моделі «**model**»:

```
y_pred_some_digit = model.predict(images)
y_pred_some_digit
```

```
1/1 [==============================] - 0s 48ms/step

array([[0., 0., 1., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

```
pred_result = np.argmax(y_pred_some_digit, axis=1)
print('Результат ПЕРЕДБАЧЕННЯ цифри - ',  pred_result)
```

```
Результат ПЕРЕДБАЧЕННЯ цифри -  [2]
```

Таким чином, наведено відомості щодо побудови іншої нейромережевої структури на основі CNN, яка може бути краще пристосована до обробки зображень.

ПРАКТИЧНА ЧАСТИНА

5.11 Підготовка до виконання завдання:

1. Ознайомтеся з теоретичними відомостями щодо застосування комп'ютерного зору, основних елементів обробки даних в ML, принципами побудови нейронних мереж, етапами обробки даних, застосуванням повністю з'єднаних і CNN нейромереж для розпізнавання цифр на прикладі набору даних MNIST.
2. Опрацюйте приклади, наведені в теоретичній частині.

5.12 Практичне завдання

Завдання № 1

1. Наберіть програму, описану в підрозділі 5.7, для побудови моделі розпізнавання цифр на прикладі набору даних MNIST для повністю з'єднаної (*Dense*) нейромережі.

2. Дослідіть її функціонування, призначення методів та їх параметрів. Отримайте модель для передбачення рукописних цифр і перевірте її якість на тестовому наборі даних. Що демонструє графік «history»?

3. Дослідіть якість різних моделей і кількість параметрів для навчання .summary() залежно від різної кількості шарів, вузлів і параметрів нейромережі. Для цього зменшіть кількість вузлів з 32 до 4

network.add(layers.Dense(4, activation='relu'))

і перевірте якість розпізнавання цифри «5», яка наведена у прикладі. Скільки при цьому отримуємо параметрів нейромережі, які навчаються? Як змінилася точність моделі «accuracy» для тренувального і валідаційного наборів?

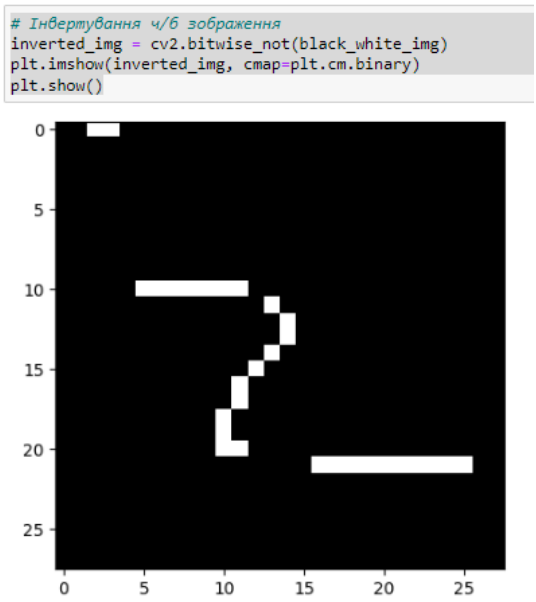
4. Для такої кількості внутрішніх шарів (4) збільшіть кількість епох до epochs=50, дослідіть графік навчання моделі та її точність на прикладі розпізнавання поточної цифри «5».

5. Додайте декілька внутрішніх шарів нейромережі і дослідіть якість розпізнавання. Як змінилася кількість параметрів, які навчаються? Як змінилася точність моделі «accuracy»? Як при цьому змінюється графік «history»?

Завдання № 2

1. Дослідіть якість отриманої моделі розпізнавання власноруч написаної цифри. Для цього згідно з підрозділом 5.8 напишіть цифру (на аркуші паперу, дошці тощо), сфотографуйте її і запишіть файл у поточну директорію, з якої запускається програма. При цьому врахуйте фон, на якому записується цифра. Наприклад, при написанні цифри на світлому аркуші паперу, можливо, потрібно буде провести інвертування зображення (за потреби):

```
# Інвертування ч/б зображення
inverted_img = cv2.bitwise_not(black_white_img)
plt.imshow(inverted_img, cmap=plt.cm.binary)
plt.show()
```



2. Наберіть програму, яка наведена в підрозділі 5.8, дослідіть її роботу. На прикладі попередньо отриманої моделі для варіанта, коли

network.add(layers.Dense(32, activation='relu'))

проаналізуйте якість розпізнавання вашої цифри і порівняйте з реальним значенням.

3. Проведіть спрощення/ускладнення моделі і проаналізуйте якість її роботи.

Завдання № 3

1. Проведіть дослідження нейронної мережі на прикладі структури CNN. Для цього наберіть програму, яка наведена в підрозділі 5.9.

2. Дослідіть призначення методів і результат роботи CNN.

3. Отримайте модель розпізнавання рукописних цифр згідно з підрозділом 5.9.

4. Проаналізуйте якість розпізнавання власноруч написаної цифри, як це було зазначено в завданні 2, виконавши підрозділ 5.10

5. Проведіть порівняння результатів розпізнавання власноруч написаної цифри при використання повністю з'єднаної (*Dense*) нейромережі (завдання 1) і CNN нейромережі.

5.13 Зміст протоколу роботи

Протокол до виконаної роботи оформлюється у вигляді файлу і містить титульний аркуш із назвою навчального закладу, факультету та випускової кафедри, дисципліни та практичної роботи, яка виконується, прізвище та ім'я студента і викладача. Наступна сторінка містить назву роботи та її мету, короткі теоретичні відомості про основні оператори і команди, які використовуються для розв'язання поставлених у роботі задач. Звіт має містити постановку задачі і програмний код її розв'язання з наданням необхідних коментарів. Наприкінці роботи наводяться висновки щодо отриманих результатів по розв'язанню поставлених задач.

Робота захищається індивідуально.

5.14 Контрольні питання для самоперевірки

1. Що таке нейронна мережа, її призначення та можливості?
2. Що таке Dence і CNN нейромережі? Які основні параметри нейромереж?
3. Що таке конволюційний шар у CNN?
4. Які основні типи функцій активації використовуються в нейронних мережах і чому вони важливі?
5. Що таке епоха в контексті навчання нейронної мережі?
6. Назвіть принаймні два типи оптимізаторів, які часто використовуються при тренуванні нейронних мереж, і поясніть їх основні відмінності.

7. Як набір даних MNIST використовується для тренування моделі розпізнавання рукописних цифр? Опишіть процес від завантаження даних до оцінки моделі.
8. Опишіть, як можна використовувати пулінговий шар у CNN, та які існують типи пулінгу?
9. Чому при тренуванні нейронної мережі важливо використовувати валідаційний набір даних?
10. Які основні відмінності між повнозв’язними та конволюційними шарами в нейронних мережах?
11. Поясніть, що таке перенавчання (overfitting) та як можна з ним боротися?
12. Що таке «batch size» та як він впливає на процес тренування нейронної мережі?
13. Які функції втрат (loss functions) зазвичай використовуються при тренуванні нейронних мереж для задач класифікації? Назвіть та опишіть принаймні дві такі функції.
14. Чому при тренуванні моделі важливо проводити нормалізацію даних? Які переваги це дає?
15. Що таке аугментація даних у контексті тренування нейронних мереж для комп’ютерного зору? Наведіть приклади технік аугментації.
16. Як працює механізм Dropout у нейронних мережах? Яка його основна функція?
17. Які переваги надає застосування архітектури CNN порівняно з повнозв'язними нейронними мережами для обробки зображень?
18. Які засоби можна використати для оцінки точності моделі нейронної мережі після її навчання?
19. Опишіть процес, яким конволюційний шар в CNN здійснює виявлення ознак на зображенні. Яку роль відіграють ядра конволюції?
20. Які основні відмінності між оптимізаторами SGD (стохастичний градієнтний спуск) та Adam? Які переваги та недоліки кожного з них?
21. Як можна використовувати техніку переносу навчання в комп’ютерному зорі? Наведіть приклади, де це може бути ефективним.
22. Які основні відмінності між ReLU (rectified linear unit) і sigmoid функціями активації? Чому ReLU часто вважається кращим вибором у нейронних мережах?
23. Поясніть, як зміни в кількості епох навчання впливають на результати нейронної мережі. Які метрики можна використовувати для визначення оптимальної кількості епох?
24. Що таке OpenCV і для яких задач його зазвичай використовують?
25. Як зчитувати та зберігати зображення за допомогою OpenCV? Які функції використовуються для цих операцій?
26. Опишіть, як можна змінити розмір зображення в OpenCV. Яка функція для цього використовується?
27. Як можна реалізувати розпізнавання обличчя в OpenCV? Які основні кроки цього процесу?

28. Як в OpenCV можна аналізувати рух? Які методи чи функції можуть бути використані для цього?
29. Опишіть, як виконати колірну фільтрацію в OpenCV. Які кроки та функції необхідні для ізоляції певного кольору на зображенні?
30. Як можна використати OpenCV для стабілізації відео? Які основні концепції або алгоритми використовуються?

5.15 Завдання до практичної роботи № 5

Проведіть дослідження деякого функціоналу по застосуванню бібліотеки OpenCV.

OpenCV (Open Source Computer Vision Library) є відкритою бібліотекою для комп'ютерного зору та машинного навчання. Вона містить понад 2500 оптимізованих алгоритмів, що охоплюють широкий спектр задач – від простих функцій обробки зображень, таких як фільтрація та морфологічні трансформації, до складних алгоритмів виявлення облич, об'єктів, трасування рухів, аналізу людської пози та багатьох інших.

OpenCV використовується в компаніях, дослідницьких групах і урядових організаціях для розпізнавання облич, розпізнавання жестів, розпізнавання номерних знаків автомобілів, класифікації людей на фотографіях, у робототехніці, автоматичній навігації та багатьох інших додатках.

Платформи: підтримує багато основних платформ, включаючи Windows, Linux, macOS та Android. *Мови програмування:* має інтерфейси для C++, Python, Java та ін., забезпечуючи зручність використання у багатьох розробницьких середовищах. Багато алгоритмів оптимізовані за допомогою C/C++ і використовують апаратне прискорення, коли це можливо, що робить OpenCV дуже швидкою для реалізації в реальному часі.

OpenCV ідеально підходить для стартапів, науковців і корпорацій для прототипування та впровадження систем комп'ютерного зору за допомогою високопродуктивних алгоритмів, доступних для використання на безкоштовній і відкритій основі.

Для наступних практичних завдань запустіть фрагменти програм та прокоментуйте результати їх роботи. Поясніть призначення функцій, які були використані.

1. Візуалізуйте зображення, яке збережене у файлі *'my_image.jpg'*, та запишіть його з новою назвою. Шлях і назва файлу можуть бути змінені:

```
import cv2
students = cv2.imread('my_image.jpg')
cv2.imshow('students', students)
cv2.imwrite('new_students.png', students)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

2. Дослідження деяких можливостей обробки зображень:

а) додавання зображень:

```
import cv2
# Зчитування двох зображень
image_1 = cv2.imread(''my_image_1.jpg')
image_2 = cv2.imread('my_image_2.jpg')
# Додавання двох зображень по всіх каналах
result = cv2.add(image_1, image_2)
# Додавання двох зображень з різними вагами
# result = cv2.addWeighted(image_1, 0.7, image_2, 0.3, 0.0)
cv2.imshow('result', result)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

б) розмиття зображень:

```
import cv2
# Download original image
original_image = cv2.imread('my_image.jpg')
 # Apply guassian blur on src image
dst = cv2.GaussianBlur(original_image,(7,7),cv2.BORDER_DEFAULT)
# Display input and output image
cv2.imshow("Gaussian Smoothing",np.hstack((original_image, dst)))
cv2.waitKey(0) # waits until a key is pressed
cv2.destroyAllWindows() # destroys the window showing image
```

в) зміна масштабу зображень:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('my_image.jpg')

# Збільшення масштабу по ширині і висоті
result_1 = cv2.resize(image, None, fx=1, fy=1,
interpolation=cv2.INTER_CUBIC)
# Зменшення масштабу по ширині і висоті
result_2 = cv2.resize(image, None, fx=0.2, fy=0.2,
interpolation=cv2.INTER_AREA)
# Вивід зображення
# plt.imshow(result_1)
plt.imshow(result_2)
plt.show()
```

г) поворот зображення:

```
import cv2
import matplotlib.pyplot as plt

# Завантаження зображення
image = cv2.imread(''my_image.jpg')
# Встановлення розмірів зображення
```

```
height, width = image.shape[:2]
center = (width/2, height/2)

# Створення матриці повороту
rotate_matrix = cv2.getRotationMatrix2D(center=center, angle=-20, scale=0.5)
# Застосування повороту
rotated_image = cv2.warpAffine(src=image, M=rotate_matrix, dsize=(width, height))
cv2.imshow("rotated image:", rotated_image)
cv2.waitKey(0) # waits until a key is pressed
cv2.destroyAllWindows() # destroys the window showing image
```

д) відтворення зображення в градаціях сірого:

```
def main():
# Завантажуємо зображення (шлях до зображення вказуйте відповідний)
image_path = 'my_image.jpg'
image = cv2.imread(image_path)

if image is None:
    print('Не вдалось завантажити зображення.')
    return

# Перетворюємо зображення в градації сірого
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Показуємо оригінальне і градації сірого зображення
cv2.imshow('Оригінальне зображення', image)
cv2.imshow('Зображення в градаціях сірого', gray_image)

# Чекаємо на будь-яку клавішу перед закриттям вікон
cv2.waitKey(0)
cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

3. Сегментація зображень та виділення обличчя

```
import cv2
# Функція для виявлення обличчя на зображенні
def detect_faces(image_path):
    # Завантажуємо зображення
    image = cv2.imread(image_path)

    # Конвертуємо зображення у відтінки сірого (градації сірого)
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Використовуємо класифікатор каскадів Haar для виявлення обличчя
    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
```

```
    faces = face_cascade.detectMultiScale(gray_image, scaleFactor=1.1,
minNeighbors=5, minSize=(30, 30))

    # Малюємо прямокутники навколо облич
    for (x, y, w, h) in faces:
        cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)

    # Показуємо зображення з відзначеними обличчями
    cv2.imshow('Faces Detected', image)
    cv2.waitKey(0)
cv2.destroyAllWindows()

# Передаємо шлях до зображення для виявлення обличчя
image_path = 'my_image.jpg'  # Замініть це на шлях до вашого зображення
detect_faces(image_path)
```

4. Сегментація зображення та виділення очей на зображенні

```
import cv2
import matplotlib.pyplot as plt

face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_eye.xml')

img = cv2.imread('my_image.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

faces = face_cascade.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(roi_gray)
    for (ex,ey,ew,eh) in eyes:
        cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)
cv2.imshow('img',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

5. Захоплення відеопотоку з відеокамери

```
import numpy as np
import cv2
cap = cv2.VideoCapture(0, cv2.CAP_DSHOW)
cap.set(cv2.CAP_PROP_FPS, 24) # Частота кадрів
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 300) # Ширина кадрів
відеопотоку.
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 200) # Висота кадрів
відеопотоку.
```

```
while(True):
    ret, frame = cap.read()
#    frame = cv2.flip(frame, -1) # Flip camera vertically
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    cv2.imshow('frame', frame)
    cv2.imshow('gray', gray)

    k = cv2.waitKey(30) & 0xff
    if k == 27: # press 'ESC' to quit
        break
cap.release()
cv2.destroyAllWindows()
```

6. Детектування обличчя з відеопотоку відеокамери

```
# Haar Cascade Face detection with OpenCV
import numpy as np
import cv2

faceCascade = cv2.CascadeClassifier(cv2.data.haarcascades + 
'haarcascade_frontalface_default.xml')

cap = cv2.VideoCapture(0, cv2.CAP_DSHOW)
cap.set(cv2.CAP_PROP_FPS, 24) # Частота кадрів
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640) # Ширина кадрів
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480) # Висота кадрів

while True:
    ret, img = cap.read()
#    img = cv2.flip(img, -1)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = faceCascade.detectMultiScale(gray, 1.3, 1)
    faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor=1.2,
        minNeighbors=5,
        minSize=(20, 20))

    for (x,y,w,h) in faces:
        cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = img[y:y+h, x:x+w]

    cv2.imshow('video',img)
    k = cv2.waitKey(30) & 0xff
    if k == 27: # press 'ESC' to quitq
        break

cap.release()
cv2.destroyAllWindows()
```

5.16 Завдання до самостійної роботи

1. Поняття і принципи дії градієнтного спуску. Його застосування для оптимізації нейронних мереж.
2. Поняття "learning rate" і його вплив на процес тренування нейронної мережі.
3. Зворотне поширення помилки (backpropagation) у контексті тренування нейронних мереж.
4. Архітектура ResNet для підвищення ефективності глибоких нейронних мереж. Її застосування і вирішення основних проблем підвишення ефективності нейромереж.
5. Батч-нормалізація та її вплив на процес тренування нейронної мережі.
6. Використання кольорового простору HSV для обробки зображень. Переваги застосування перетворення зображення з RGB до HSV.
7. Архітектури конволюційних нейронних мереж (CNN) та їх застосування у комп'ютерному зорі. Дослідження різних типів CNN, таких як AlexNet, VGG, ResNet.
8. Розширені методи обробки зображень в OpenCV. Морфологічні операції, фільтрація і трансформація зображень та їх практичне застосування.
9. Застосування глибокого навчання для виявлення та класифікації об'єктів. Огляд сучасних підходів та мереж, таких як YOLO, SSD та Faster R-CNN, для задач детекції об'єктів.
10. Використання нейромереж для розпізнавання жестів рук. Дослідження, як комп'ютерний зір може інтерпретувати жести рук як форму несловесної комунікації, з можливими застосуваннями в інтерфейсах людина-комп'ютер.

6 ІНСТАЛЯЦІЯ ТА ОСНОВИ РОБОТИ НА RASPBERRY PI

Мета практичної роботи № 6: отримати практичні навички встановлення операційної системи Raspbian GNU/Linux і виконувати налаштування віддаленого підключення до Raspberry Pi та його функціональними можливостями.

ТЕОРЕТИЧНІ ПОЛОЖЕННЯ

6.1 Відомості про платформу Raspberry Pi

Для того щоб підключити і використовувати мінікомп'ютер **Raspberry Pi,** необхідно попередньо інсталювати операційну систему (ОС), під керуванням якої і буде здійснюватися загальне функціонування пристрою. Комп'ютер без операційної системи – це просто конструкція з металу та полімерів (рисунок 6.1)

Рисунок 6.1 – Загальний вигляд мінікомп'ютера Raspberry Pi

Основні характеристики системи залежать від моделі. Наприклад, деякі характеристики для моделі Raspberry Pi 4 Model B мають такі значення:

- Тип процесора – Broadcom BCM2711;
- Частота процесора – 1,5 ГГц;
- Кількість ядер процесора – 4;
- Об'єм оперативної пам'яті – 8 ГБ;
- Тип пам'яті – LPDDR4;
- Зв'язок Ethernet – 1000 Мбіт/с;
- Безпровідний модуль – Wi-Fi 2,4 ГГц та 5 ГГц стандартів IEEE 802.11.b/g/n/ac;

  – Bluetooth 5.0;

- Підключення та роз’єми:
  порти – USB2 x USB 3.0; 2 x USB 2.0;
  роз’єм живлення – USB-C (5 В на 3 А);
- Наявність слота для CSI камери;
- Збереження програм та операційної системи на SD-карті, із можливістю роботи з флеш-накопичувачами та жорсткими дисками, що підключаються через USB адаптер;
- Формат карт пам’яті – MicroSD;
- Аудіовихід – Micro HDMI; 3.5 mm Jack;
- Відеовихід – Micro-HDMI; 3.5 mm Jack.

На рисунку 6.2 зображено, які пристрої підключаються до відповідних входів-виходів RPI.

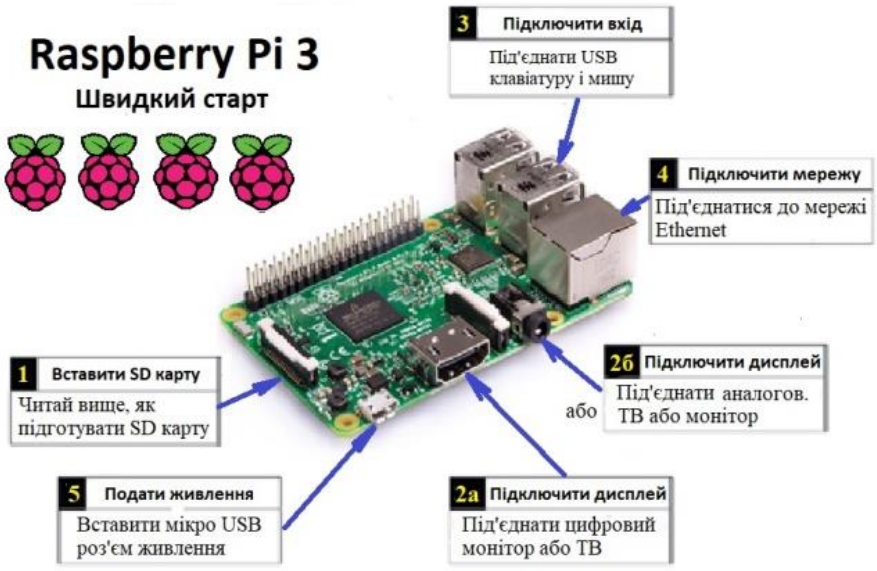


Рисунок 6.2 – Загальний вигляд «швидкого підключення» Raspberry Pi

Raspberry Pi працює в основному на ОС, що базуються на Linux ядрі, таких як:

- Raspbian (модифікація Debian);
- Arch Linux ARM;
- Pidora (модифікація Fedora);
- Kali Linux.

Також можливе використання FreeBSD та Windows 10 IoT Core.

Найпопулярнішою ОС для Raspberry 3 є Raspbian. Вона базується на популярному дистрибутиві Debian. Її відмінність полягає в тому, що в ній «з коробки» існує майже повна підтримка Raspberry. Так, вже після встановлення користувачеві доступний Python і модуль для роботи з GPIO (General Purpose Input Output – низькорівневий інтерфейс вводу-виводу прямого керування).

Завантажити операційну систему для Raspberry Pi 3 можна з офіційного сайту https://www.raspberrypi.com/software/. Всі ОС для RPi є безкоштовними, тому із їх завантаженням не виникне проблем.

6.2 Встановлення ОС Raspbian GNU/Linux

У процесі виконання лабораторних робіт будемо використовувати операційну систему *Raspbian GNU/Linux*. Це відкрита та вільна ОС, адаптація Debian GNU / Linux під архітектуру ARM. Raspbian підтримує легковаговий графічний інтерфейс LXDE і стандартний менеджер пакетів apt-get, так що при встановленні програм з використанням Інтернету не виникає проблем. Так само підтримуються майже всі програми, доступні на Linux для ПК, зокрема компілятори та редактори коду.

Існує два способи встановлення ОС на Raspberry Pi:

- скачування пакета NOOBS на карту та подальша установка;
- монтування образу ОС Raspbian прямо на карту. В цьому випадку можна буде приступати до використання відразу після включення.

Застосуємо перший спосіб, тому що він зручніший, і пакет NOOBS сам виконує необхідні налаштування.

Для початку роботи з Raspberry Pi (RPi) знадобиться кілька речей:

- одноплатний мінікомп'ютер Raspberry Pi;
- блок живлення постійного струму 5В/2А;
- кабель USB – MicroUSB;
- монітор з HDMI-входом;
- кабель HDMI;
- карта пам'яті MicroSD (ємністю від 8 до 64 Гб включно);
- дротова USB-мишка та клавіатура (або бездротовий аналог з Bluetooth адаптером).

Завантажити пакет можна з офіційного сайту https://www.raspberrypi.com/software/ (рисунок 6.3). При скачуванні завантажується дистрибутив ***imager_1.6.2***.

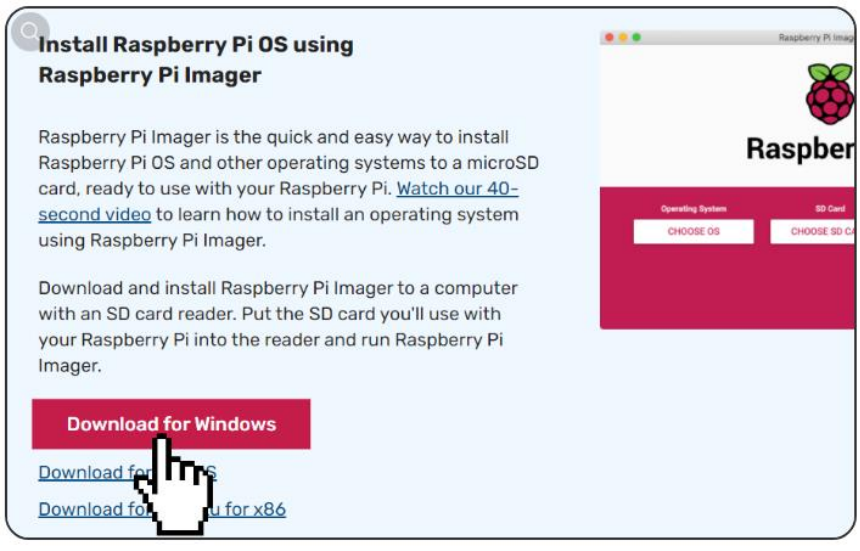


Рисунок 6.3 – Завантаження дистрибутиву з офіційного сайту

Для того щоб встановити ОС на карту, виконаємо такі дії (https://qazf.com.ua/raspberry-pi-install-os/):

1) вставляємо SD-карту в комп'ютер (не в Raspberry Pi) та виконуємо форматування карти; при форматуванні вказуємо файлову систему FAT32;
2) завантажуємо з сайту інсталяційний пакет ***imager_1.6.2***. та запускаємо його (рисунок 6.4);

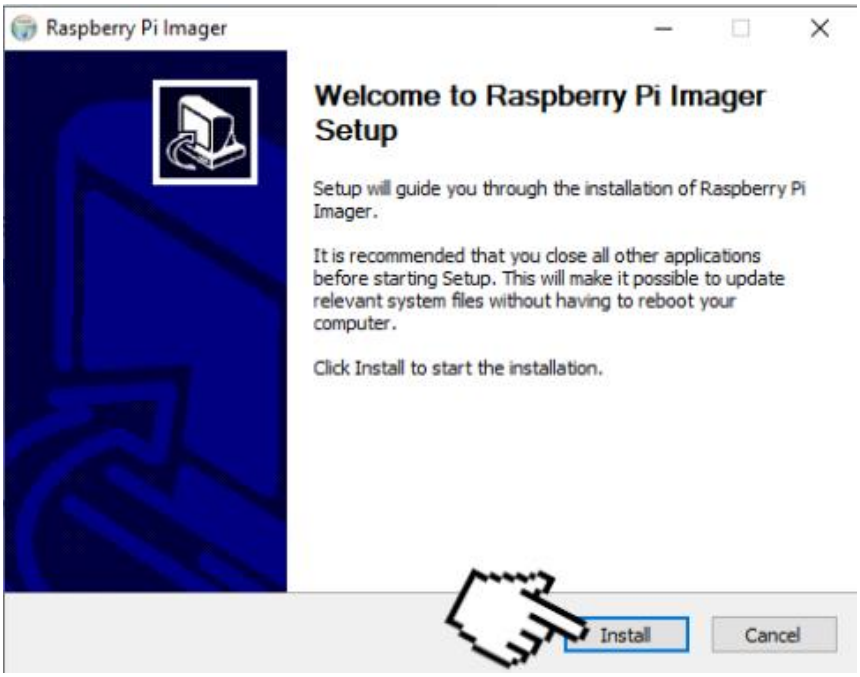


Рисунок 6.4 – Встановлення пакета дистрибутиву ***imager_1.6.2***

3) після встановлення ***imager_1.6.2*** запускаємо його, в результаті з'являється вікно (рисунок 6.5);

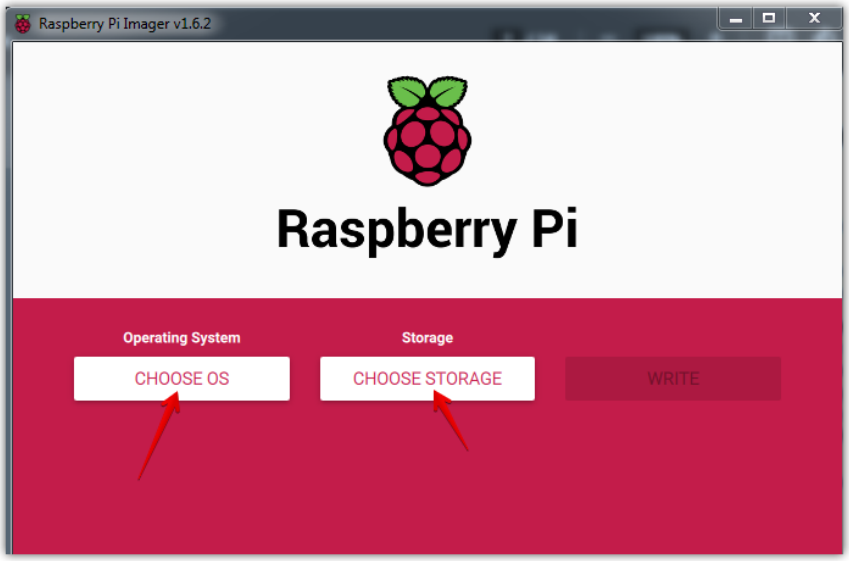


Рисунок 6.5 – Вікно програми для вибору ОС (Choose OS) та карти пам'яті (Choose Storage), яка вставлена в ПК

4) обираємо ОС (рисунок 6.6) і відповідну карту пам’яті (рисунок 6.7), з якою і буде працювати RPi.

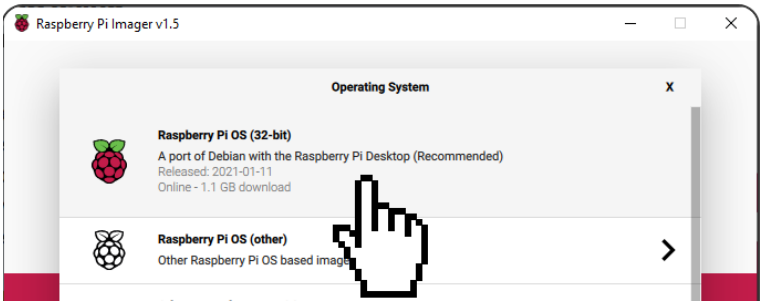


Рисунок 6.6 – Вибір ОС

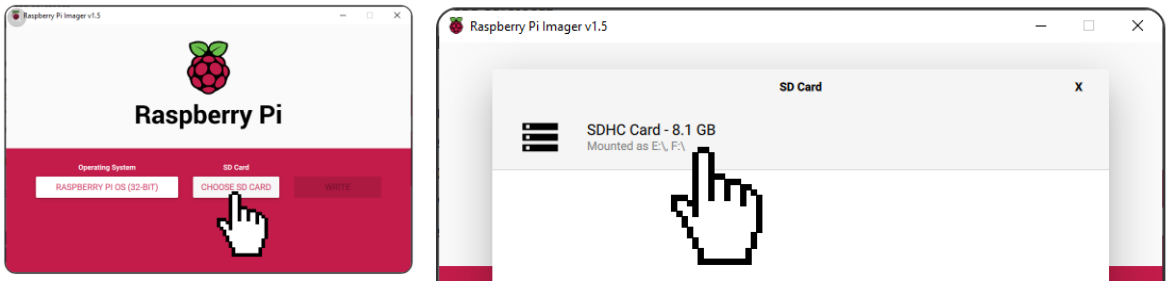


Рисунок 6.7 – Вибір карти пам’яті для інсталяції ОС

Після відповідного вибору натисніть кнопку для запису ОС (рисунок 6.8).

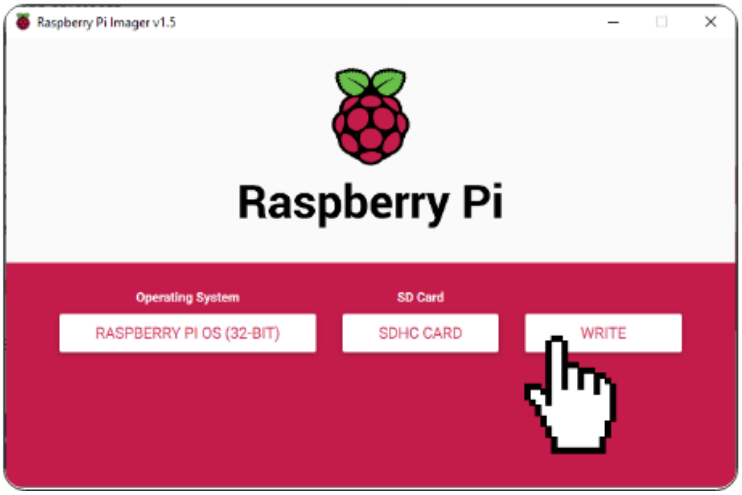


Рисунок 6.8 – Запис ОС на зовнішню (обрану користувачем) карту пам’яті

УВАГА! Майте на увазі, що процес копіювання операційної системи на карту знищить усі дані на ній! Переконайтеся, що це та сама карта, відключіть інші SD карти від ПК, якщо вони є.

Відбувається інсталяція ОС, яка займає деякий час. Далі можна вийняти карту пам’яті і зробити відповідні з’єднання, як показано на рисунку 6.9.

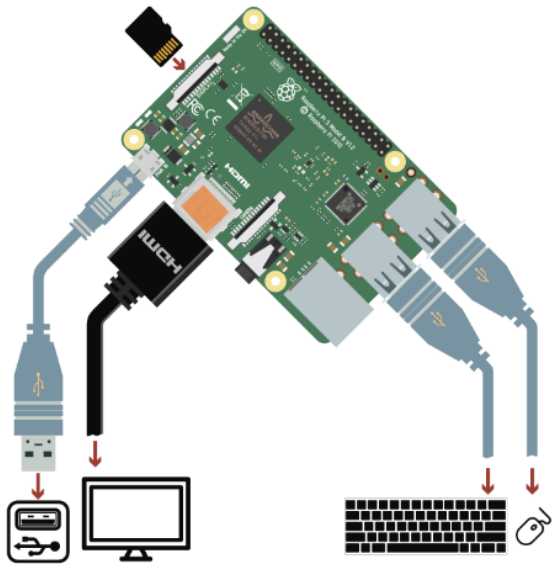


Рисунок 6.9 – З’єднання RPi з периферійними пристроями (живленням +5 В, монітором через кабель HDMI, клавіатурою та мишкою через USB порти)

Помістіть мінікомп’ютер Raspberry Pi на рівну тверду поверхню. Вставте картку пам’яті в гніздо MicroSD на нижній частині плати. Переконайтеся, що вона надійно зафіксована у роз’ємі. Підключіть проводи для клавіатури та мишки до роз’ємів USB. Підключіть монітор до мінікомп’ютера за допомогою кабелю HDMI. Підключіть блок живлення до гнізда MicroUSB. Увімкніть монітор, а потім увімкніть блок живлення Raspberry Pi.

На мінікомп’ютері немає окремої кнопки включення, тому він запуститься відразу після подачі живлення. Якщо все пройшло успішно, можна буде побачити стартовий екран ОС Raspbian (рисунок 6.10). Вітаємо!

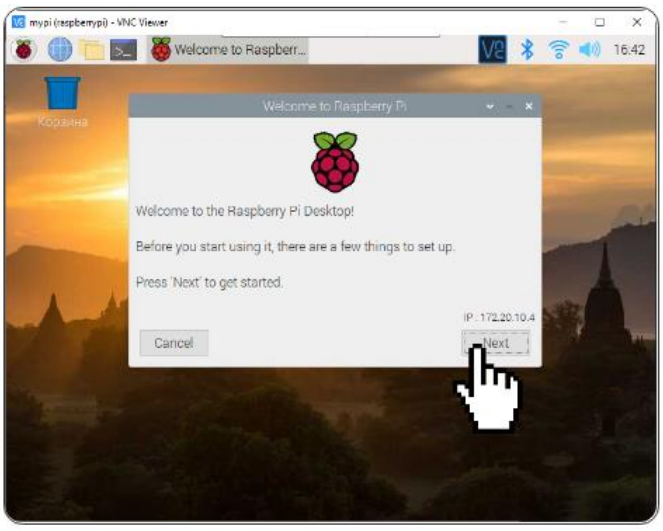


Рисунок 6.10 – Перше включення RPi

При першому завантаженні Raspberry Pi OS **вітає Вас**! в діалоговому вікні. Натисніть "Next" (рисунок 6.10). Виберіть зі списку налаштування, що відповідають вашому регіону. Також надається можливість встановити пароль (Password), але з огляду на навчальне спрямування проєкту і багаторазове використання RPi багатьма учасниками цього робити не рекомендується.

Виберіть вашу точку WiFi, введіть її пароль та натисніть "Next". Якщо у вас немає WiFi або ви підключилися через Ethernet, натисніть "Skip" (рисунок 6.11).

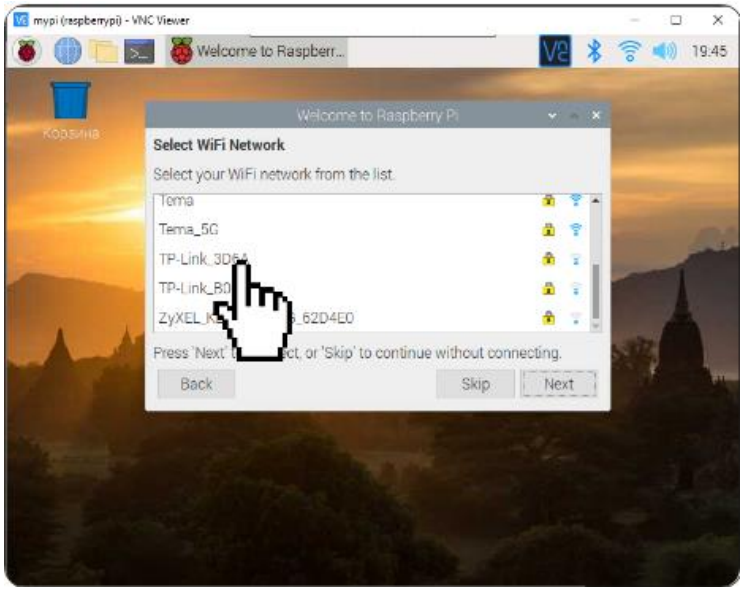


Рисунок 6.11 – Підключення до мережі WiFi

Далі можна оновити систему, натиснувши "Next", або можна зробити це потім, натиснувши "Skip" (рисунок 6.12). Налаштування завершено! Можна використовувати Raspberry Pi OS.

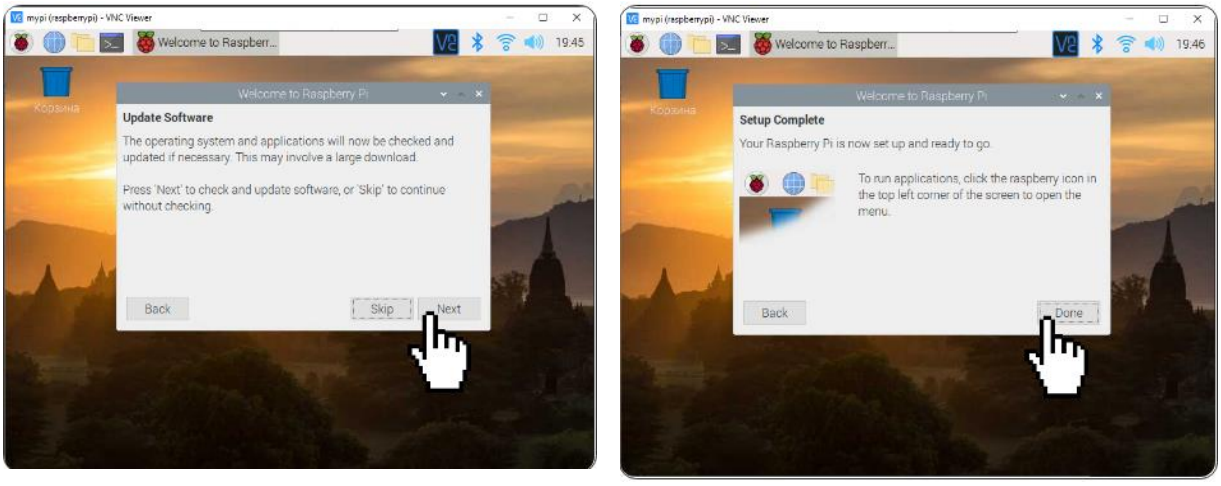


Рисунок 6.12 – Завершення налаштування RPi

Увага! Якщо потрібно, щоб до першого запуску RPi була можливість під'єднатися до бездротової мережі WiFi, то необхідно перед встановленням карти пам'яті до RPi після інсталяції ОС виконати такі дії (https://picockpit.com/raspberry-pi/ru/headless-setup-fro-raspberry-pi/).

Після завершення копіювання системи на карту microSD на вашому ПК з'явиться диск BOOT, його необхідно відкрити і створити в ньому НОВІ файли **wpa_supplicant.conf** і **ssh.txt.**

Перший файл повинен бути порожнім і називатися **ssh.txt** – це необхідно для того, щоб RPi при першому завантаженні зрозуміла, що необхідний доступ по **ssh** протоколу.

Другий файл має називатися **wpa_supplicant.conf** та містити дані для підключення до WiFi. При завантаженні Raspberry сама перенесе дані з цього файлу /etc/wpa_supplicant/wpa_supplicant.conf.

Ось приклад файлу **wpa_supplicant.conf**:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1

network={
    ssid="НАЗВАНИЕ ТОЧКИ ДОСТУПА WiFi"
    psk="ПАРОЛЬ WiFi"
    key_mgmt=WPA-PSK
}
```

Назва точки доступу та пароль пишуться в лапках, наприклад, якщо wifi-точка називається Druidia та пароль 12345, то запис буде мати вигляд:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1

network={
    ssid="Druidia"
    psk="12345"
    key_mgmt=WPA-PSK
}
```

Далі встановлюємо карту пам'яті до RPi і виконуємо дії, які описані вище.

Після того як система запустилася можемо приступити до деяких налаштувань.

6.3 Основні налаштування Raspberry Pi

Налаштування часової зони. Тепер слід налаштувати часову зону. Запускаємо "Start->Preferences->Raspberry Pi Configuration" (рисунок 6.13, а). У програмі переходимо на вкладку "Localization" і натискаємо кнопку "Set Timezone". У новому вікні (рисунок 6.13, б) обираємо Area: Europe і потім Location Kiev. Далі натискаємо OK і ще раз OK та закриваємо програму.

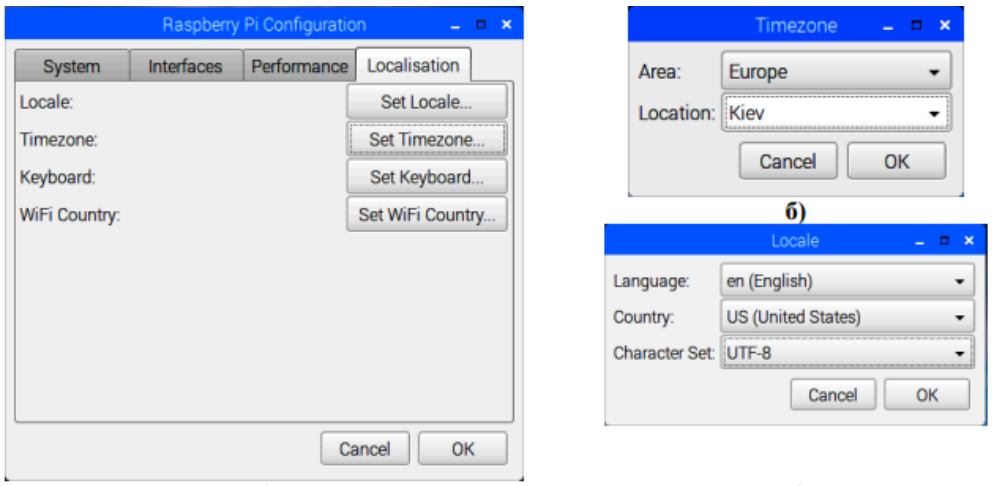


Рисунок 6.13 – Налаштування часової зони

Налаштування клавіатури. Перед початком роботи потрібно встановити додаткову розкладку клавіатури. Тому на етапі початкового налаштування при першому включенні системи рекомендується обрати англійську мову як мову за замовчуванням. Слід підключитися до мережі і лише потім, після налаштування, додавати інші розкладки.

Для цього обираємо розкладку за замовчуванням. У меню Preferences—> Raspberry Pi configuration (рисунок 6.13, а) обираємо закладку Localisation. Натискаємо тепер кнопку «Set Locale.» (рисунок 6.13, в) і обираємо Language – «en (English)», Coutnry – «US (United States)», Character set – «UTF-8». Натискаємо OK, ще раз OK і на запит про перевантаження натискаємо «Yes». Добавляємо аплет на Панель задач.

Клікаємо правою кнопкою на Панель задач у будь-якому вільному місці. Обираємо Add/Remove Panel Items. У новому вікні обираємо кнопку Add та шукаємо у випавшому списку Keyboard Layout Handler. Обираемо його та натискаємо Add. Далі натискаємо OK. В панелі в правому кутку з’явиться прапор США, це і є перемикач.

Робота з портами. Для простого вводу-виводу даних можна використовувати інтерфейс **GPIO**. GPIO (General Purpose Input Output) – це низькорівневий інтерфейс вводу-виводу прямого управління, для цього на платі Raspberry Pi 3 наявний 40-контактний роз’єм GPIO, через який Raspberry може приймати та отримувати сигнали стану логічного 0 та 1. Оскільки інтерфейс низькорівневий, обмінюватися сигналами Raspberry може з будь-якими іншими пристроями – від світлодіода до складних цифрових приладів і датчиків (рисунок 6.14).

Як видно з рисунка, деякі сигнали, крім стандартного простого вводу-виводу, можуть виконувати свою роль, яку називають альтернативною функцією контакта. Крім того, як видно, на контакти роз’єма виведені також сигнали деяких поширених цифрових інтерфейсів, а також напруги живлення та заземлення. Всі контакти GPIO підведені безпосередньо у схему, а сигнали GPIO заведені напряму у процесор, тому слід бути

уважним і не допускати перевантаження сигналів чи подачі на лінії сторонніх напруг, більших за 3.3 В чи менших за 0 В. Сигнали GPIO підтримують режими низькорівневого вводу-виводу та шість спеціальних режимів роботи. Також деякі сигнали GPIO можна використовувати для відправки переривань на процесор.

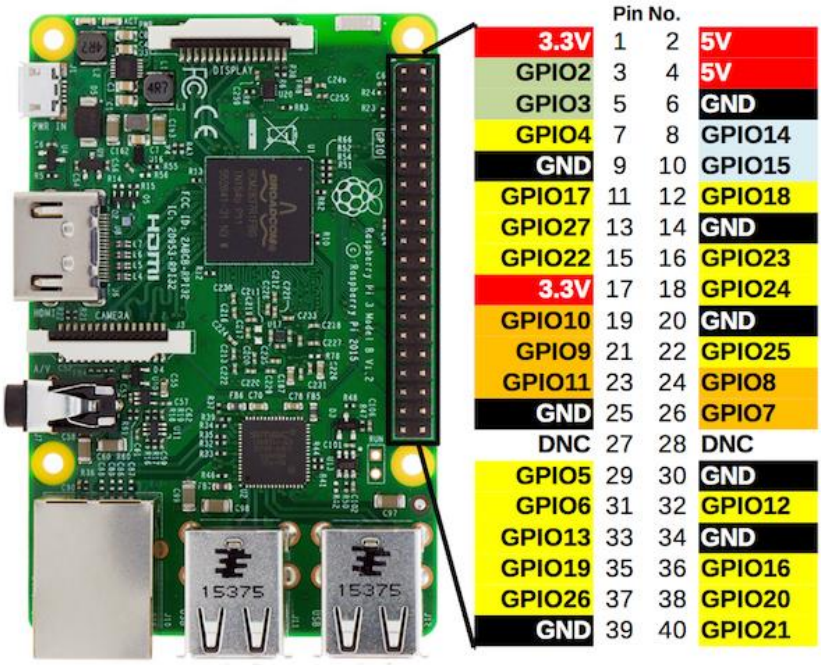


Рисунок 6.14 – GPIO та номери контактів на платі RPi

ПРАКТИЧНА ЧАСТИНА

6.4 Підготовка до виконання завдання

1. Ознайомтеся з теоретичними відомостями інсталяції та початком роботи на RPi, основами початкових налаштувань підключення мережі, зміни часу, розкладки клавіатури тощо.

2. Ознайомтеся з можливостями віддаленого доступу та керування RPI, основними налаштуваннями через **VNC** (Virtual Network Computing) та керуванням RPI через консоль.

6.5 Практичне завдання

1. Проведіть інсталяцію ОС на PRi згідно з описаними вище інструкціями.
2. Проведіть налаштування PRi для підключення до бездротової мережі WIFI.
3. Підключіть PRi згідно з рисунком 6.9 і запустіть мікрокомп’ютер.
4. Проведіть налаштування часової зони, клавіатури.
5. Познайомтеся з меню PRi та основними налаштуваннями.

6. Запустіть редактор Python («Малинка» (Пуск), перемістіть курсор у розділ «Програмування» і клацніть **Thonny Python IDE**) та познайомтеся з основними функціями запуску програм.
7. Зробіть висновки.

6.6 Зміст протоколу роботи

Протокол до виконаної роботи оформлюється у вигляді файлу і містить титульний аркуш із назвою навчального закладу, факультету та випускової кафедри, дисципліни та практичної роботи, яка виконується, прізвище та ім'я студента і викладача. Наступна сторінка містить назву роботи та її мету, короткі теоретичні відомості про основні оператори і команди, які використовуються для розв'язання поставлених у роботі задач.

Звіт має містити постановку задачі і програмний код (або основні інструкції) її розв'язання з наданням необхідних коментарів. Наприкінці роботи наводяться висновки щодо отриманих результатів по розв'язанню поставлених задач.

Робота захищається індивідуально.

6.7 Контрольні питання для самоперевірки

1. Що таке Raspberry Pi, призначення, які особливості застосування?
2. Які компоненти потрібні для запуску Raspberry Pi 3 і як їх підключити?
3. Як завантажити та встановити операційну систему на Raspberry Pi 3 з використанням MicroSD карти? Перелічіть ОС, що підтримуються Raspberry Pi.
4. Які існують способи встановлення ОС на Raspberry Pi? Як виконується встановлення Raspbian GNU/Linux?
5. Наведіть технічні характеристики Raspberry Pi. Наведіть схему пристрою Raspberry Pi.
6. Як здійснити підключення Raspberry Pi 3 до мережі Wi-Fi або Ethernet?
7. Як налаштувати інтерфейс користувача Raspberry Pi (CLI або GUI) для виконання команд та налаштувань?
8. Як запустити базовий програмний код на Raspberry Pi 3, такий як програма на Python, і як вивести результати?
9. Що таке GPIO та яким чином можна нею керувати? Які режими підтримує GPIO?
10. Чим відрізняється GPIO нумерація виводів у стандарті BOARD та BCM?
11. Яким чином можна виконати організацію радіопередавача за допомогою виходу GPIO?
12. Як виконується завантаження та складання програми fmtransmitter на Raspberry Pi?

6.8 Задачі до практичної роботи № 6

1. Зберіть схему згідно з рисунком 6.15.

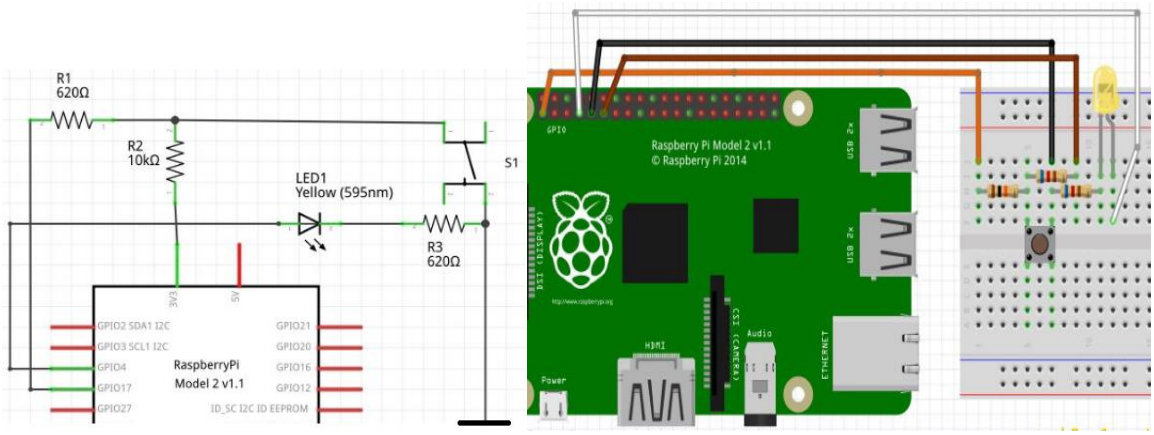


Рисунок 6.15 – Схема для дослідження портів вводу/виводу даних

2. Відкрийте редактор **Thonny** на Raspberry Pi, напишіть програму, яка б реалізовувала функцію зчитування події натискання кнопки, що приведе до миготіння світлодіоду при її натисканні (лістинг програми наведено нижче) (https://ph0en1x.net/86-raspberry-pi-znakomstvo-s-gpio-perekluchatel-i-svetodiod.html).

```
import time
import RPi.GPIO as GPIO
LED = 4
KEY = 17
GPIO.cleanup()
GPIO.setmode(GPIO.BCM)
GPIO.setup(LED, GPIO.OUT)
GPIO.output(LED, GPIO.LOW)
GPIO.setup(KEY, GPIO.IN)
print 'Привіт! Миг...миг...'
try:
    while True:
    if GPIO.input(KEY) == False:
        timeout = 0.1
        print 'Клавішу натиснуто.'
    else:
        timeout = 0.5
    GPIO.output(LED, GPIO.HIGH)
    time.sleep(timeout)
    GPIO.output(LED, GPIO.LOW)
    time.sleep(timeout)
```

3. Розрахуйте величину опору гасячого резистора.

Наприклад, світлодіод розрахований на напругу 2 В і струм споживання 10 мА, тоді гасячий резистор матиме значення:

R = (3,3 – 2 В) / 0,01 А = 130 Ом.

4. Перевірте, чи дійсно програма виконується так, як написано в завданні 3. Якщо ні, то потрібно знайти та виправити помилку.
5. Додайте коментарі до програми.

6.9 Завдання до самостійної роботи

1. Налаштування віддаленого робочого столу VNC на Raspberry Pi (https://itmaster.biz.ua/electronics/raspberry-pi/raspberrypi-vnc.html).
2. Організація віддаленого керування по SSH (https://itmaster.biz.ua/electronics/raspberry-pi/raspberry-pi-zero-2-w-ssh.html).
3. Структура мікрокомп'ютера та тип пам'яті Raspberry Pi.
4. Зовнішні пристрої, які підключаються до Raspberry Pi. Наведіть приклади.
5. Можливе застосування платформи Raspberry Pi для задач автоматизації, робототехніки та телекомунікацій.
6. Системи живлення та охолодження платформи, способи організації.
7. Розробка IoT-проєктів: вивчіть, як використовувати Raspberry Pi 3 для створення проєктів Інтернету речей, включаючи моніторинг погоди, керування освітленням та інші датчикові застосунки.
8. Створення простих робототехнічних систем: ознайомтеся з можливостями підключення робототехнічних компонентів до Raspberry Pi 3 та розробіть прості проєкти роботів, такі як автономний автомобіль або робот-маніпулятор.
9. Створення системи моніторингу: дослідіть, як використовувати Raspberry Pi 3 для створення системи моніторингу, що вимірює температуру, вологість, рівень освітленості тощо та надсилає дані на сервер або зберігає їх локально.
10. Розробка власного годинника: вивчіть, як побудувати годинник на Raspberry Pi 3 з використанням дисплея, модуля часу реального світу (RTC), а також можливість синхронізації з Інтернетом.

7 ОСНОВНІ ПРИНЦИПИ РОБОТИ З GPIO НА ПРИКЛАДІ ПІДКЛЮЧЕННЯ ТА КЕРУВАННЯ ПЕРИФЕРІЙНИМИ ПРИСТРОЯМИ

Мета практичної роботи № 7: отримати практичні навички програмування інтерфейсу вводу-виводу прямого управління GPIO (General Purpose Input Output) на прикладі керування світлодіодами та сервоприводами.

ТЕОРЕТИЧНІ ПОЛОЖЕННЯ

7.1. Відомості про інтерфейс Raspberry Pi

Raspberry Pi – це маленький, розміром із кредитну картку, мікрокомп'ютер (рисунок 7.1). Raspberry Pi побудований на процесорі з архітектурою ARM11 та частотою від 700 МГц до 1,5 ГГц і вище, що забезпечує досягнення прийнятної продуктивності за низького енергоспоживання. Raspberry Pi по суті є повноцінним системним блоком, за допомогою якого можна навчати роботі з комп'ютером, відтворювати відео, програмувати, користуватися Інтернетом, слухати музику тощо.

Одна з головних і привабливих особливостей Raspberry Pi – наявність на платі апаратних портів вводу/виводу GPIO (входи/виходи загального призначення), що відкриває перспективи використання його в робототехнічних проєктах.

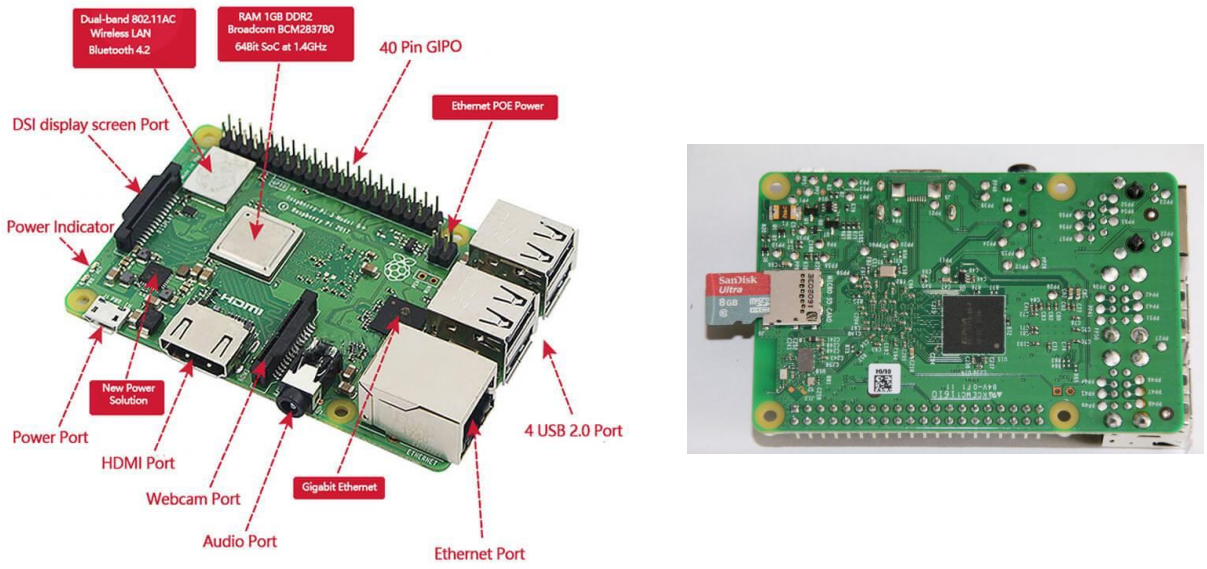


Рисунок 7.1 – Зовнішній вигляд плати Raspberry Pi 3 та призначення елементів вводу/виводу

Наведемо опис складових частин Raspberry Pi, їх характеристик та застосування.

Чіп BCM2837

На Raspberry Pi 3 встановлений 64-бітний процесор Broadcom BCM2837 на архітектурі ARM Cortex-A53 з тактовою частотою 1,2 ГГц і модулем оперативної пам’яті на 1 ГБ. Процесор і пам’ять розміщені за технологією "package-on-package" безпосередньо на процесорі. BCM2837 включає також двоядерний графічний співпроцесор Video Core IV® Multimedia, який забезпечує Open GL ES 2.0, апаратне прискорення Open VG і 1080p30 H.264 декодування.

USB-Ethernet перетворювач LAN9512

Чіп LAN9512 являє собою високошвидкісний USB2.0 Hub і Ethernet контролер.

WiFi та Bluetooth

Інтегрований 802.11n Wi-Fi та Bluetooth 4.1.

HDMI-порт

Роз’єм призначений для виведення цифрового відео та звуку на мультимедійні пристрої. Для комунікації знадобиться HDMI-кабель.

Якщо використовується VGA монітор, то використовується перехідник на HDMI (рисунок 7.2).

Рисунок 7.2 – Перехідник для монітора VGA-HDMI

Аудіо / Відео вихід

3,5 мм роз’єм з додатковим виведенням на композитний відеовихід для підключення навушників або інших пристроїв відтворення звуку та телевізорів.

Роз’єм живлення

Роз’єм micro-USB призначений для живлення Raspberry Pi. Жодних кнопок увімкнення/вимкнення на корпусі плати не передбачено. Якщо необхідно запустити Raspberry Pi, підключаєте USB-живлення, для вимкнення слід висмикнути шнур.

Роз’єми 4×USB2.0

USB-хаб із чотирма роз’ємами для підключення клавіатури, миші, флешок та інших USB-пристроїв.

Ethernet-роз’єм

10/100 Мбіт Ethernet-роз’єм для підключення до мережі через RJ45 патч-корд крученої пари.

Роз’єм камери (CSI-2)

15-контактний плаский гнучкий роз’єм інтерфейсу MIPI CSI-2 для підключення камери.

Роз’єм дисплея (DSI)

15-контактний плаский гнучкий роз’єм, універсальний високошвидкісний інтерфейс для дисплеїв.

Поряд із портом живлення знаходиться слот для **SD-карти**. Без карти пам’яті Raspberry Pi не вмикається, оскільки саме на ній записана ОС. Оскільки власної ОС у внутрішній пам’яті у Raspberry Pi немає, то з цього випливає один позитивний момент – пристрій практично неможливо перетворити на «непрацюючий брухт». Після будь-якого невдалого експерименту достатньо перезаписати дистрибутив на карту пам’яті, і Raspberry Pi знову запрацює як новий.

7.2 Робота з інтерфейсом вводу-виводу прямого управління GPIO (General Purpose Input Output)

Як відзначалося, основною відмінною рисою Raspberry Pi від звичайного ПК є наявність на платі портів загального призначення GPIO. Користувачеві доступна можливість керування цими виводами, а це означає, що до Raspberry Pi можна підключати дисплеї, кнопки, датчики, реле й інші електронні модулі, якими можна маніпулювати на власний розсуд.

Зовні GPIO виконаний у вигляді дворядної штирьової колодки з кроком 2,54 мм, яка розташована на краю плати. Ранні моделі, такі як B та A містять 26 виводів, а більш сучасні – 40. На рисунку 7.3 показано зовнішній вигляд портів загального призначення для плати Raspberry Pi 3B+ із зазначенням нумерації виводів.

Рисунок 7.3 – Нумерація виводів GPIO порту

Щоб повноцінно використовувати GPIO, недостатньо знати їх нумерацію. Необхідно точно розуміти, де розташований той чи інший вивід, як він називається і за що він відповідає.

Як видно з рисунка 7.4, на колодці, крім самих GPIO, виведені штирі з напругою 3,3 V, 5 V, а також виводи GND. Деякі GPIO мають альтернативні функції, призначення яких вказано у синіх блоках. До того ж, не можна порушувати навантаження порту, щоб не вивести Raspberry Pi з ладу. Слід пам'ятати, що GPIO працює з напругою 3,3 V та максимальним струмом навантаження 50 mA на один вивід. Це означає, що будь-яке перевищення зазначених параметрів негативно позначиться на працездатності плати, тому краще використовувати гальванічну розв'язку між GPIO та зовнішнім виконавчим пристроєм. Те саме стосується і вхідних ланцюгів, до яких застосовуються резистивні дільники і всілякі перетворювачі рівнів.

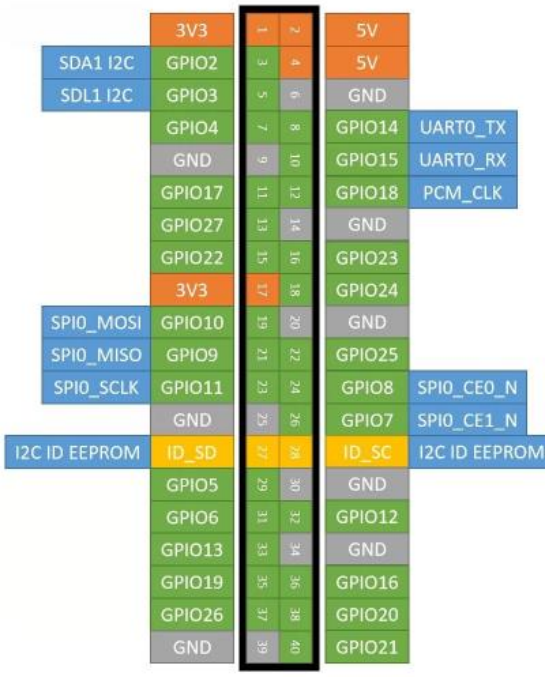
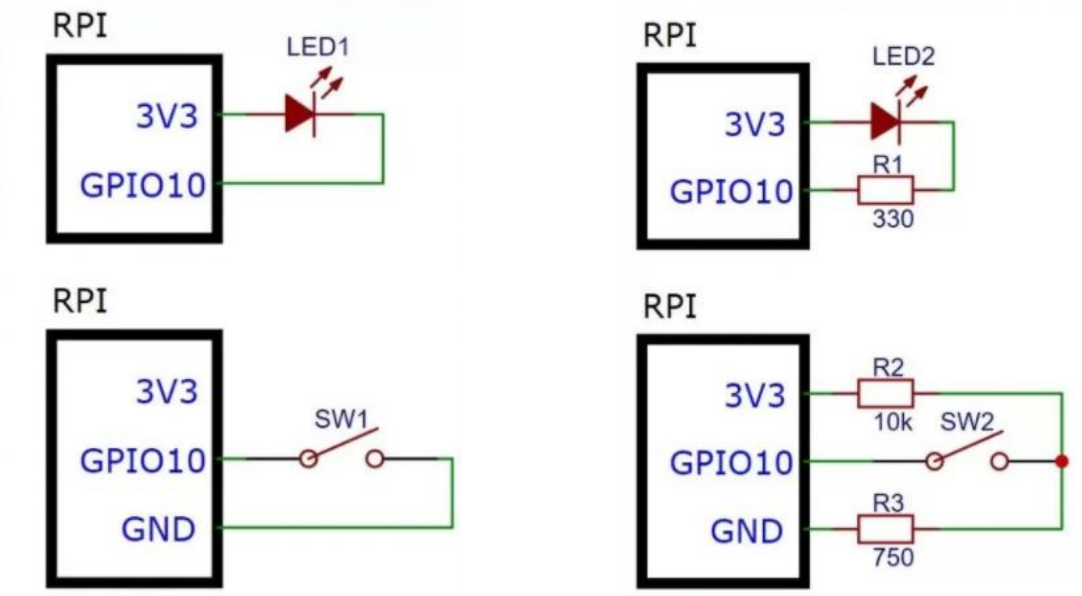


Рисунок 7.4 – Нумерація виводів GPIO порту та їх призначення

На рисунку 7.5 показано приклад правильного та неправильного підключення базових елементів до GPIO порту.У лівій частині рисунка 7.5 пряме підключення світлодіода призведе до перевищення максимально допустимого струму 50 mA. Це, зі свого боку, **виведе GPIO10 з ладу**. У правій частині рисунка додано обмежувальний резистор, який утримуватиме струм у допустимих межах.

Що стосується кнопки SW2 (рисунок 7.5, б), то може виникнути ситуація, коли GPIO10 помилково буде налаштовано на вихід, і її натискання призведе до прямого з'єднання 3,3 V і GND. При додаванні резисторів R2 та R3 усі виводи будуть гарантовано захищені від перевантажень.



а) б)

Рисунок 7.5 – Приклад неправильного (а) та правильного (б) підключення базових елементів до GPIO порту

Виходячи з вищевикладеного, можна дійти висновку, що **ПОТРІБНО РЕТЕЛЬНО ПЛАНУВАТИ З'ЄДНАННЯ**, щоб Raspberry Pi не вийшла з ладу.

7.3 Програмування GPIO на прикладі включення світлодіода

Операційна система Raspbian пропонує користувачам зручний модуль для програмного керування GPIO. Називається він RPi.GPIO і є стандартною програмою. Перед його застосуванням модуль рекомендується оновити. Зробити це можна, набравши в консолі рядки:

```
sudo apt-get update
sudo apt-get install python-rpi.gpio
```

Щоб мати практичне уявлення про роботу з GPIO, створимо невеликий проєкт, який змусить Raspberry Pi блимати світлодіодом раз на секунду, а при натисканні на кнопку збільшувати частоту миготіння в п'ять разів. Схему майбутнього проєкту показано на рисунку 7.6.

За керування світлодіодом відповідатиме GPIO4, а за читання стану – GPIO17.

Як правило, програми для Raspberry Pi пишуться скриптовою мовою програмування **Python**. Особливість її полягає в тому, що для запуску програми не потрібний компілятор. Скрипт запускається і починає роботу відразу, але його необхідно зберегти у файл із подальшим завантаженням у плату. Для цього відкриваємо термінал та прописуємо рядок

```
1 | nano /home/pi/led_key_test.py
```

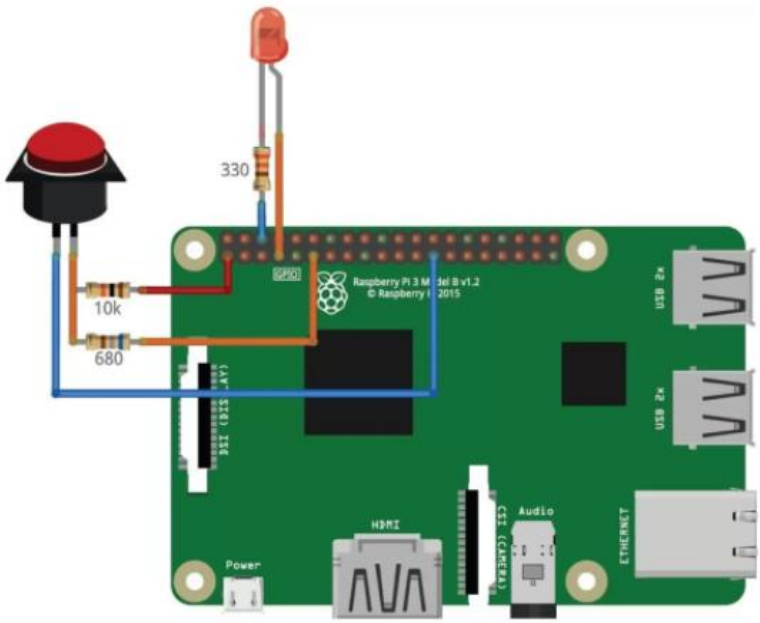


Рисунок 7.6 – Схема керування світлодіодом

Тим самим ми створюємо файл led_key_test.py в директорії /home/pi. Як наслідок, відкриється редактор, в який необхідно написати наведений нижче код:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Підключення бібліотек для роботи з GPIO і #організації затримок по часу
import time
import RPi.GPIO as GPIO
# Визначення виводів GPIO, до яких підключені #світлодіод та кнопка
LED = 4
KEY = 17
# Скидання портів (всі виводи налаштовуються на вхід – #INPUT)
GPIO.cleanup()
# Режим нумерації пінів – по назві (не по порядковому номеру на платі)
GPIO.setmode(GPIO.BCM)
# Налаштування піна LED на вихід (OUTPUT)
GPIO.setup(LED, GPIO.OUT)
# Встановлення логічного (0) на виводі LED
GPIO.output(LED, GPIO.LOW)
# Налаштування піна KEY на вхід (INPUT)
GPIO.setup(KEY, GPIO.IN)
# Вивід вітання на екран
```

```
print ('Hello Raspberry Pi')
# Перевірка на переривання програми по натисненню #(CTRL+C) на клавіатурі
try:
    # Нескінченний цикл
    while True:
    # Якщо кнопка натиснута (на пині KEY логічний 0)
        if GPIO.input(KEY) == False:
            # Встановлюємо затримку 0,1 с
          timeout = 0.1
            print ('Key is pressed.')
        else:
            # в іншому разі затримка - 0,5 с
            timeout = 0.5
        # Увімкнемо світлодіод
        GPIO.output(LED, GPIO.HIGH)
        # Затримка
        time.sleep(timeout)
        # Вимикаємо світлодіод
        GPIO.output(LED, GPIO.LOW)
        time.sleep(timeout)
# Якщо CTRL+C була натиснута – скидаємо порт і #завершуємо роботу програми
except KeyboardInterrupt:
    GPIO.cleanup()
```

Ще однією важливою особливістю програми Python є дотримання відступів (табуляцій) при написанні програм. Зважайте на це правило при створенні свого коду.

Отже, переходимо до останнього етапу. Щоб вийти з редактора, тиснемо CTRL+X і зберігаємо програму натисканням «y» + ENTER. Залишилося тільки зробити скрипт виконуваним. Для цього вводимо в терміналі рядки:

```
1 chmod +x /home/pi/led_key_test.py
2 /home/pi/led_key_test.py
```

Використаємо GPIO для моделювання роботи світлофора при натисканні кнопки, як це робиться на пішохідних переходах, де зазвичай горить зелене світло для транспорту, а пішохід може кнопкою запустити

програму включення червоного світла для транспорту. Алгоритм цієї програми такий: при натисканні кнопки починає блимати зелене світло, потім на короткий час запалюється жовте, потім червоне; червоне світло горить деякий час, потім короткий час горять червоне і жовте, і, нарешті, знову зелене; Далі система чекає чергового натискання кнопки.

Для програмування цього алгоритму скористаємося вбудованою в образ ОС Raspbian інтегрованого середовища розробки (IDE) мовою Python (Пайтон). Мова Python має велику кількість переваг, що робить її дуже потужним інструментом як для програмістів-початківців, так і для професіоналів.

Середовище розробки програм мовою Python запускається з робочого столу послідовним вибором **Menu - Programming - Python 3.** Далі у вікні Python Shell, що відкрилося, слід натиснути File - New File. У вікні редактора, що відкрилося, потрібно набрати або скопіювати такий **текст програми**, звертаючи особливу увагу на відступи в тексті, тому що для програм на Python вони мають принципове значення:

```
#!/usr/bin/python
import RPi.GPIO as GPIO
from time import sleep
RED_PIN = 36
YELLOW_PIN = 32
GREEN_PIN = 29
BUTTON_PIN = 40
print ("RPi.GPIO init start")
GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD)
print ("RPi.GPIO init end")
print ("GPIO setup")
GPIO.setup(RED_PIN, GPIO.OUT)
GPIO.setup(YELLOW_PIN, GPIO.OUT)
GPIO.setup(GREEN_PIN, GPIO.OUT)
GPIO.setup(BUTTON_PIN, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.output(RED_PIN, 0)
GPIO.output(YELLOW_PIN, 0)
GPIO.output(GREEN_PIN, 1)
while True:
   inp = GPIO.input(BUTTON_PIN)
   if inp==0:
     for x in range(0, 5):
        GPIO.output(GREEN_PIN, 1)
```

```
    sleep(0.5)
    GPIO.output(GREEN_PIN, 0)
    sleep(0.5)
GPIO.output(YELLOW_PIN, 1)
sleep(2)
GPIO.output(YELLOW_PIN, 0)
GPIO.output(RED_PIN, 1)
sleep(5)
GPIO.output(YELLOW_PIN, 1)
sleep(1)
GPIO.output(RED_PIN, 0)
GPIO.output(YELLOW_PIN, 0)
GPIO.output(GREEN_PIN, 1)
```

Зібраний проєкт має вигляд, наведений на рисунку 7.7.

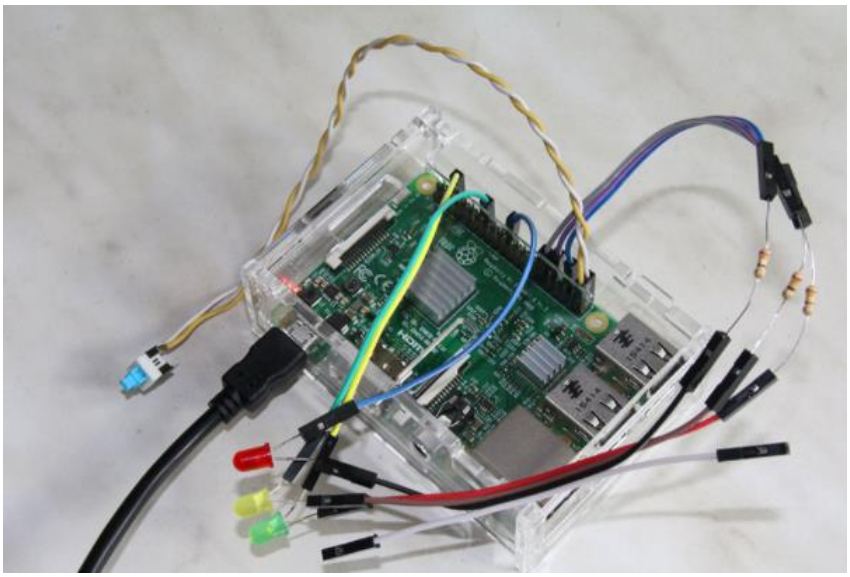


Рисунок 7.7 – Проєкт світлофора на Raspberri Pi

Опис програми. Рядки, що починаються з GPIO.setup, задають режим виходу (OUT) або входу (IN) відповідних пінів, а аргумент pull_up_down=GPIO.PUD_UP включає резистор, що на вході 40, до якого підключена кнопка. Оскільки програма на Python не має стандартного «нескінченного циклу», як, наприклад, у Ардуїно, де завантажена в мікроконтролер програма виконується нескінченно, поки подано живлення, оператор «while True» здійснює цей цикл. Адже нам треба повертати наш світлофор у вихідний стан щоразу після завершення циклу його роботи.

Оператор присвоєння inp = GPIO.input(BUTTON_PIN) записує в змінну *inp* значення на вході 40. Якщо кнопка не натиснута – це «0», якщо

натиснута – «1». Якщо *inp* дорівнює «0», то починається цикл роботи світлофора:
- за допомогою циклу for п’ять разів блимає зелений світлодіод;
- на дві секунди запалюється жовтий (пауза задається оператором *sleep*);
- жовтий гасне, запалюється червоний на п’ять секунд і т.д.

Після закінчення циклу роботи світлофора все починається знову.

Короткі виводи світлодіодів (це мінус) підключаємо до землі – контакти 6, 14, 20; довгі (плюс) через резистори 240 Ом – до контактів 29 (зелений), 32 (жовтий), 36 (червоний).

Кнопку підключаємо до контактів 39 та 40.

Тепер у редакторі з нашою програмою вибираємо Run - Run Modul або натискаємо F5, і програма починає виконуватися, очікуючи натискання кнопки.

Автономний запуск програми. Незручно щоразу запускати програму за допомогою оболонки. Найзручніше, щоб наша програма запускалася при включенні живлення Raspberry, адже тоді пристрій можна використовувати автономно, без монітора, клавіатури та миші.

Для цього необхідно включити програму до автозавантаження операційної системи. Тут нам знадобиться термінал, без нього не обійтись.

Спочатку збережемо нашу програму у вигляді файлу svetofor-rpi.py3 у кореневому каталозі користувача /home/pi.

Тепер запустимо термінал і після запрошення pi@raspberrypi:~ $ наберемо рядок

gksudo leafpad /etc/xdg/autostart/Svetofor.desktop

Тим самим ми викликаємо текстовий редактор leafpad та створимо файл Svetofor.desktop у папці автозапуску.

У текстовому редакторі набираємо рядки:

[Desktop Entry]
Version=1.0
Encoding=UTF-8
Name=Svetofor
Comment=
Exec=sudo python /home/pi/svetofor-rpi.py3
Terminal=false
Type=Application

і зберігаємо файл.

Основне в цьому файлі – рядок, що починається з *Exec*, який запускає інтерпретатор Python на виконання програми svetofor-rpi.py3.

Можна перевірити наявність файлу, зайшовши в папку /etc/xdg/autostart за допомогою файлового менеджера.

Тепер, якщо вимкнути живлення, відключити монітор, мишу та клавіатуру та знову включити живлення, наш світлофор почне працювати в автономному режимі!

Як має працювати світлофор, можна подивитися за посиланням: https://youtu.be/y_l80oFHs7g

7.4 Програмування GPIO на прикладі керування сервоприводами

Продемонструємо керування сервоприводами, які під'єднані до RPi згідно з рисунком 7.8.

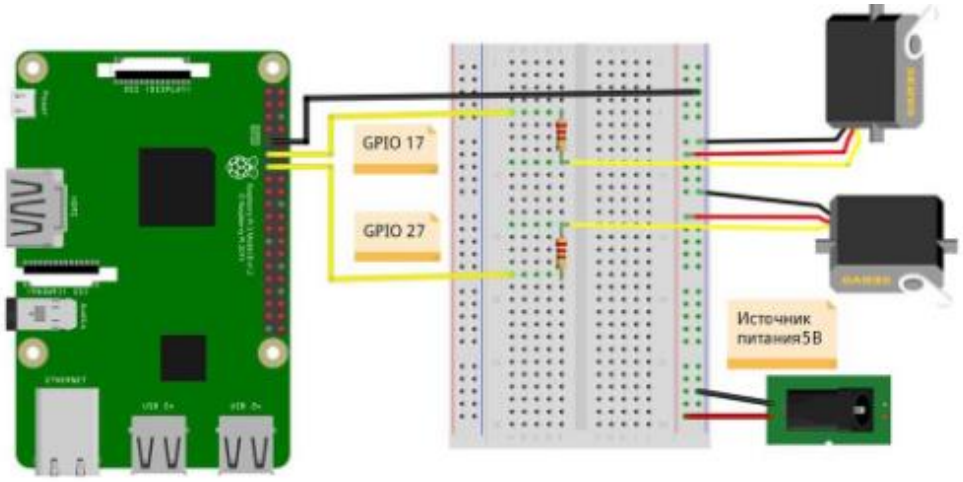


Рисунок 7.8 – Підключення сервоприводів до Rpi

Сервоприводи будуть підключені до зовнішнього джерела живлення 5 В, а керуючий провід – до Raspberry Pi. Кольори дротів у різних сервоприводів можуть відрізнятися, але в цьому випадку:

коричневий – «земля»; червоний – «+5 В»; жовтий – сигнальний.

До GPIO 17 підключається керуючий провід сервоприводу нахилу, GPIO 27 сервопривід повороту. Для захисту виводів Raspberry Pi можна використовувати резистор 1 кОм.

УВАГА! Якщо ви також використовуєте окремі джерела живлення для Raspbery Pi та сервоприводів, обов'язково з'єднайте їх землі, інакше електроніка може вийти з ладу. У цьому випадку земля від зовнішнього джерела живлення для сервоприводів підключена до виводу землі (біля GPIO 17) на Raspberry Pi.

Калібрування сервоприводів. Для цього відкриємо термінал на Raspberry та запустимо редактор Python 3 з правами від суперкористувача (потрібно для роботи з GPIO):

sudo python3

Python Shell:

#Імпортуємо модуль RPI.GPIO і назвемо його GPIO:

import RPi.GPIO as GPIO

Перед початком роботи потрібно визначитися, яка саме схема нумерації виводів буде використовуватися (BCM або BOARD). Наведений приклад подано зі схемою BOARD, тому контакти, що використовуються, будуть фізичними (GPIO 17 – це контакт 11 і GPIO 27 – це контакт 13). Вибираємо:

GPIO.setmode(GPIO.BOARD)

Визначаємо вивід сервоприводу, який використовуватиметься:

tiltPin = 11

Якщо хочете використовувати схему BCM, останні дві команди повинні бути замінені на:

GPIO.setmode(GPIO.BCM)

tiltPin = 17

Тепер вказуємо, що цей вивід працюватиме як вихід:

GPIO.setup(tiltPin, GPIO.OUT)

Налаштовуємо частоту, для SG90 потрібно 50 Гц:

tilt = GPIO.PWM(tiltPin, 50)

Включаємо генерацію сигналу ШІМ на виводі та задаємо початковий коефіцієнт заповнення, який дорівнює нулю:

tilt = start(0)

Тепер ми можемо встановлювати різні значення коефіцієнта заповнення та спостерігати за рухом сервоприводу. Почнемо з 5% і подивимося, що відбувається:

tilt.ChangeDutyCycle(5)

Сервопривід перейшов у «нульовий стан». Сервопривід продовжує обертатись при заповненні аж до 3%. При заповненні 2% сервопривід залишається у тому самому положенні. Отже, 3% – це мінімальне заповнення (позиція «0 градусів») для цього сервоприводу. Потрібно перевірити самостійно це значення, оскільки для різних моделей сервоприводів воно може бути різним.

Встановимо максимальний коефіцієнт заповнення – 10%:

tilt.ChangeDutyCycle(10)

Потім задаємося іншими значеннями – сервопривід продовжує провертатися при встановленні коефіцієнта заповнення до 13%. Таким чином, максимальний коефіцієнт заповнення для цього сервоприводу – це 13%. Кут, на який провертається вал сервоприводу, становить приблизно 180 градусів. Отже, в результаті калібрування отримано такі дані:

0 градусів ==> заповнення 3%;

90 градусів ==> заповнення 8%;

180 градусів ==> заповнення 13%.

Після закінчення калібрування зупиняємо ШІМ та очищаємо GPIO:

```
tilt=stop()
GPIO.cleanup()
```

Для другого сервоприводу процедура калібрування аналогічна. Створимо файл "angleServoCtrl.py" скрипту на Python для виконання тестів:

```
from time import sleep
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
def setServoAngle(servo, angle):
pwm = GPIO.PWM(servo, 50)
pwm.start(8)
dutyCycle = angle / 18. + 3.
pwm.ChangeDutyCycle(dutyCycle)
sleep(0.3)
pwm.stop()
if __name__ == '__main__':
import sys
servo = int(sys.argv[1])
GPIO.setup(servo, GPIO.OUT)
setServoAngle(servo, int(sys.argv[2]))
GPIO.cleanup()
```

Функція setServoAngle(servo, angle) отримує як аргументи номер виводу GPIO, до якого підключено сервопривід, і значення кута, куди сервопривід повинен провернутися.

У консолі цей скрипт запускається так:

sudo python3 angleServoCtrl.py 17 45

Наведена вище команда встановить сервопривід, підключений до GPIO 17, під кутом 45 градусів.

ПРАКТИЧНА ЧАСТИНА

7.5 Підготовка до виконання завдання:

1. Ознайомтеся з основними портами та їх призначенням Rpi.
2. Ознайомтеся з теоретичними відомостями підключення до GPIO, основними правилами, призначенням виводів.

7.6 Практичне завдання

1. Реалізуйте схему підключення світлодіода до RPi та забезпечте керування ним згідно з наведеними вище інструкціями.
2. Реалізуйте схему керування світлофором на RPi згідно з наведеними вище інструкціями.
3. Реалізуйте керування сервоприводами на RPi згідно з наведеними вище інструкціями. Напишіть програму, згідно з якою один сервопривід повертався б як секундна стрілка, а другий – відпрацьовував повороти як хвилинна стрілка.
4. Зробіть висновки.

7.7 Зміст протоколу роботи

Протокол до виконаної роботи оформлюється у вигляді файлу і містить титульний аркуш із назвою навчального закладу, факультету та випускової кафедри, дисципліни та практичної роботи, яка виконується, прізвище та ім’я студента і викладача. Наступна сторінка містить назву роботи та її мету, короткі теоретичні відомості про основні оператори і команди, які використовуються для розв’язання поставлених у роботі задач.

Звіт має містити постановку задачі і програмний код її розв’язання з наданням необхідних коментарів. Наприкінці роботи наводяться висновки щодо отриманих результатів по розв’язанню поставлених задач.

Робота захищається індивідуально.

7.8 Контрольні питання для самоперевірки

1. Які можливості надає бібліотека RPi.GPIO для роботи з GPIO на Raspberry Pi 3? Наведіть приклади основних функцій.
2. Які команди потрібно використовувати в терміналі Raspberry Pi 3 для оновлення системи та встановлення необхідних бібліотек для роботи з GPIO?
3. На який максимальний струм і чому можна навантажувати GPIO?
4. Як ви здійснюєте зберігання та обробку даних, зчитаних із GPIO-пінів на Raspberry Pi 3 за допомогою Python?
5. Які можливості надає бібліотека pigpio для роботи з GPIO на Raspberry Pi 3, і в чому її відмінності від RPi.GPIO?
6. Які існують безпровідні інтерфейси Raspberry Pi 3?
7. Який об’єм оперативної пам’яті у Raspberry Pi 3 та на що вона впливає при функціонуванні пристрою?

8. Які основні характеристики процесора Raspberry Pi 3?
9. Назвіть периферійні пристрої, які можна підключити до Raspberry Pi 3. Які існують способи зберігання та обміну даними між Raspberry Pi 3 та зовнішніми пристроями через GPIO?
10. Поясніть, як налаштувати простий датчик (наприклад температурний або руху) з Raspberry Pi 3 та зчитати його дані за допомогою Python.
11. Які основні переваги використання Raspberry Pi 3 для розробки та впровадження проєктів, що вимагають взаємодії з різними електронними пристроями через GPIO?
12. Які є переваги та недоліки використання GPIO на Raspberry Pi 3, порівнюючи з іншими методами взаємодії з периферійними пристроями?
13. Як ви можете використовувати переривання (interrupts) для обробки подій на пінах GPIO на Raspberry Pi 3?
14. Як ви можете забезпечити захист від перенапруги або перевантаження для пристроїв, підключених до пінів GPIO на Raspberry Pi 3?
15. Які існують можливості використання аналогових сигналів на Raspberry Pi 3, і як вони відрізняються від цифрових сигналів?
16. Які можливості надає управління GPIO через Інтернет (IoT) на Raspberry Pi 3, і які практичні застосунки цього можуть бути?
17. Як ви можете встановити і налаштувати сервопривід на Raspberry Pi 3? Як ви програмуєте Raspberry Pi 3 для керування сервоприводом?
18. Які є стратегії керування сервоприводом для досягнення різних типів рухів або позицій?
19. Як ви можете створити ефект миготіння або плавного затемнення світлодіода за допомогою Raspberry Pi 3?
20. Визначте максимальне значення частоти перемикання світлодіода, коли ми вже не бачимо його миготіння.
21. Як ви можете використовувати широтно-імпульсну модуляцію (PWM) для керування яскравістю світлодіода за допомогою Raspberry Pi 3? Поясніть принцип роботи PWM.
22. Які існують методи вимірювання і контролю споживаної потужності світлодіода, і як це може бути корисним у практичних застосунках?
23. Які існують способи реалізації мультикольорового світлодіода з декількома кольорами на Raspberry Pi 3, і як ви можете керувати кожним кольором окремо?
24. Як ви можете використовувати світлодіод як індикатор стану для інших пристроїв чи процесів, і як це може бути використано в реальних проєктах?
25. Як ви можете розробити систему індикації та сигналізації на основі світлодіодів з використанням Raspberry Pi 3, яка відповідатиме на різноманітні події та стани в системі?
26. Як ви можете реалізувати інтерактивність світлодіодних ефектів з використанням зовнішніх сенсорів або введення від користувача на Raspberry Pi 3?

7.9 Задачі до практичної роботи № 7

1. Доповніть програму фрагментом коду, який дозволить будь-якої миті перервати роботу циклу за допомогою комбінації клавіш Ctrl+C. При цьому програма має завершиться у нормальному режимі та виконати функцію cleanup.

2. Доповніть програму фрагментом коду, який дозволить будь-якої миті змінювати швидкість перемикання світлофора при натисканні на кнопки.

3. Реалізуйте керування сервоприводами при натисканні на кнопки RPi. Напишіть програму, згідно з якою сервопривід буде повертатися в напрямку часової стрілки при натисканні однієї кнопки або навпаки, при натисканні іншої кнопки.

7.10 Завдання до самостійної роботи

1. **Розширення можливостей GPIO:** дослідіть додаткові можливості використання GPIO на Raspberry Pi 3, такі як підключення до різних типів сенсорів, пристроїв введення/виведення та інтерфейсів зовнішніх пристроїв.
2. **Застосування мережевих технологій:** дослідіть використання Raspberry Pi 3 у мережевих застосунках, таких як створення серверів, мережевих моніторів, IoT проєктів та ін.
3. **Робототехніка і автоматизація:** вивчіть як використовувати Raspberry Pi 3 для створення робототехнічних систем, автоматизації домашнього середовища, управління робочими процесами та інші автоматизовані застосунки.
4. **Використання камери і обробка зображень:** дослідіть можливості використання камери Raspberry Pi та вивчіть методи обробки зображень, розпізнавання облич та об’єктів, а також створення систем відеоспостереження.
5. **Розробка проєктів зв’язку та IoT:** дослідіть використання Raspberry Pi 3 для створення проєктів зв’язку, включаючи розробку систем моніторингу, керування та збір даних, побудову мережевих додатків та інтерфейсів IoT.
6. Організуйте на Raspberry Pi 3 smart tv приставку для перегляду передач. Основні етапи реалізації цього проєкту

РЕКОМЕНДОВАНА ЛІТЕРАТУРА ТА ПОСИЛАННЯ

1. Яковенко А. В. Основи програмування. Python. Частина 1 [Електронний ресурс] : підручник для студ. спец. 122 «Комп'ютерні науки», спеціалізації «Інформаційні технології в біології та медицині» / А. В. Яковенко. – Київ : КПІ ім. Ігоря Сікорського, 2018. – 195 с.
2. Креневич А. П. Python у прикладах і задачах. Частина 1. Структурне програмування : навч. посіб. із дисципліни «Інформатика та програмування» / А. П. Креневич. – Київ : ВПЦ «Київський університет», 2017. – 206 с.
3. Програмування числових методів мовою Python : підручник / А. В. Анісімов, А. Ю. Дорошенко, С. Д. Погорілий, Я. Ю. Дорогий ; за ред. А. В. Анісімова. – Київ : ВПЦ «Київський університет», 2014. – 640 с.
4. Костюченко А. О. Основи програмування мовою Python: навч. посіб. / А. О. Костюченко. – Чернігів : ФОП Баликіна С. М., 2020. 180 с.
5. Васильєв О. М. Програмування мовою Python / О. М. Васильєв. – Тернопіль : Навчальна книга-Богдан, 2018. – 504 с.
6. Могильний С. Б. Машинне навчання з використанням мікрокомп'ютерів : навч.-метод. посіб. / С. Б. Могильний ; за ред. О. В. Лісового та ін. – Київ, 2019. – 226 с.
7. Sweigart A. Core Python Applications Programming / A. Sweigart. – 2012. – 888 p.
8. Raspberry Pi: Офіційний логотип Raspberry Pi [Електронний ресурс]. – Режим доступу : https://www.raspberrypi.org/wp-content/uploads/2015/08/raspberry-pilogo.png. (дата звернення: 02.09.2020).
9. Бизли Д. М. Язык программирования Python : справочник / Д. М. Бизли. – Киев : ДиаСофт, 2000. – 336 с.
10. Hunt J. A Beginners Guide to Python 3 Programming / J. Hunt. – Springer, 2019. – 441 p.
11. Ceder N. The Quick Python Book / N. Ceder. – 3rd ed. – Manning Publications Co., 2018. – 332 p.
12. Беррі П. Head First Python / Пол Беррі. – 2-ге вид. – Харків : ВД «Фабула», 2023. – 624 с.
13. Лутц М. Python. Довідник програміста/М. Лутц. – Київ : Науковий Світ, 2024. – 294 с.
14. Євсеєв С. П. Кібербезпека: криптографія з Python : навч. посіб. / С. П. Євсеєв, О. В. Шматко, О. Г. Король. – Харків: Вид-во «Новий Світ-2000», 2024. – 120 с.
15. Висоцька В. А. Python: алгоритмізація та програмування : навч. посіб. / В. А. Висоцька, О. В. Оборська. – Львів : Вид-во ПП «Новий Світ-2000», 2023. – 516 с.
16. Васильєв О. М. Програмування мовою Python / О. М. Васильєв. – Тернопіль : Начальна книга-Богдан, 2022. – 504 с.