ЛЕКЦІЯ 3 · PYTHON

Модулі, файли, ООП та екосистема

Модулі, файли, об’єкти та бібліотеки

Дисципліна «Мова програмування Python»
Викладач: Голенко Максим Юрійович

План лекції 3

01 **Модулі й середовище**

Організація коду та залежностей

02 **Файли й формати**

Текст, CSV, JSON

03 **Об’єктна модель**

Класи, методи, протоколи

04 **Екосистема й підсумок**

Бібліотеки, ШІ, готовий звіт

Модулі та структура програми

Розмістимо логіку, запуск і тести у зрозумілих місцях.

Після впорядкування коду налаштуємо середовище виконання.

Від функцій до проєкту

Проєкт розділяє код за призначенням: запуск, обчислення, робота з файлами та перевірки.

main.py
Запуск

processing.py
Обчислення

io_data.py
Файли

tests/
Перевірки

main.py поєднує потрібні функції. processing.py містить обчислення, io_data.py — читання й запис. tests/ перевіряє ці частини. Порядок блоків не задає порядок виконання.

Модуль: призначення і простір імен

Модуль — одиниця коду з власним простором імен. Власний модуль зазвичай зберігають у файлі .py.

Простір імен пов’язує назви з об’єктами. Після import до функції average звертаються через ім’я її модуля processing.

Власний файл

```
# processing.py
def average(values):
```

Функцію average з лекції 2 переносимо у processing.py. Тут показано початок її визначення.

Імпорт

```
import processing
processing.average(values)
```

Імпорт надає об’єкт модуля. Виклик використовує визначену в ньому функцію та список values.

Виконання

```
print("Модуль завантажено")
```

Якщо такий рядок стоїть у processing.py поза функціями, він виконається під час першого імпорту.

Визначення функції не запускає її тіло. Звичайні повторні імпорти використовують кеш модуля.

Імпорт: три способи звернення

Імпорт відкриває доступ до імен модуля. Форма запису визначає, як звертатися до них у поточному файлі.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
import модуль
from модуль import назва
import модуль as псевдонім
```

Модуль

Звернення має форму модуль.назва. Ім’я модуля показує джерело.

Окреме ім’я

Після from можна використовувати назва без префікса модуля.

Псевдонім (as)

Псевдонім замінює локальне ім’я модуля, але не сам модуль.

У схемах українські позначення замінюють фактичними іменами.

Форми імпорту

Три способи імпорту дають доступ до тих самих засобів math.

PYTHON · ПРИКЛАД

```python
import math
from math import ceil
import math as m

print(math.sqrt(81))
print(ceil(3.2))
print(m.floor(3.8))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
9.0
4
3
```

Пояснення

Простір імен. math.sqrt показує джерело функції. Псевдонім. as дає локальне коротке ім’я.

Запуск модуля: умова __name__

Модуль може бути і бібліотекою для імпорту, і самостійною програмою. Умова відокремлює запуск від визначень.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
def main():
    інструкції

if __name__ == "__main__":
    main()
```

Функція main

Містить дії програми. Ім’я main — домовленість, а не ключове слово.

Умова запуску

При прямому запуску __name__ має значення "__main__".

Тіло функції

Виконується лише після main(). Відступ — 4 пробіли.

Під час імпорту визначення зберігаються, а захищений виклик пропускається.

Точка входу: __name__

Той самий файл можна запустити як програму або імпортувати як модуль.

PYTHON · ПРИКЛАД

```python
def main():
    print("Створюємо звіт")

if __name__ == "__main__":
    main()
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Створюємо звіт
```

Пояснення

Запуск файла. __name__ дорівнює __main__. Імпорт. Захищений виклик не запускається.

Пакет і модуль — різні поняття

Модуль містить код. Пакет об’єднує модулі та підпакети у спільному просторі імен.

У звичайному пакеті каталог містить __init__.py. Наприклад, course.processing — модуль processing у пакеті course.

Каталог

```
course/
    processing.py
    io_data.py
```

Це структура файлів, а не Python-код. Кожний .py містить код окремого модуля.

Звичайний пакет

```
course/__init__.py
```

Цей файл позначає звичайний пакет і може містити код його ініціалізації. Він може бути порожнім.

Використання

```
from course import processing
processing.average(values)
```

Імпортуємо модуль processing із пакета course. Функцію викликаємо як processing.average(values).

Namespace-пакети можуть не мати __init__.py. Це окремий механізм, який тут не деталізуємо.

Імпорти: коротка довідка

Запис	Призначення	Важливий нюанс
`import math`	Модуль за ім’ям	math.sqrt(4)
`from math import sqrt`	Окреме ім’я	sqrt(4)
`import math as m`	Псевдонім	m.sqrt(4)
`__name__`	Сценарій запуску	__main__ для головного файла
`sys.path`	Шляхи пошуку	Залежать від запуску
`ModuleNotFoundError`	Модуль не знайдено	Перевірити ім’я й середовище

Не називаємо власні файли math.py, random.py або json.py.

Середовище та залежності

Відокремимо засоби мови від установлених пакетів.

У налаштованому середовищі під’єднаємо файли.

Три рівні засобів Python

Python має вбудовані засоби, стандартну бібліотеку та сторонні пакети. Вони підключаються по-різному.

Вбудовані функції доступні відразу. Стандартні модулі входять до Python. Сторонні пакети встановлюють у вибране середовище.

Вбудовані засоби

```
print(len([85, 45]))
```

len рахує елементи, print виводить 2. Для цих функцій import і встановлення не потрібні.

Стандартні модулі

```
import math
math.sqrt(81)
```

math входить до стандартної бібліотеки. Після імпорту sqrt(81) повертає 9.0.

Сторонні пакети

```
import pandas as pd
```

Цей імпорт спрацює після встановлення pandas у тому інтерпретаторі, що виконує програму.

Команда термінала python -m pip install pandas встановлює пакет; import підключає його в програмі.

Віртуальне середовище

Віртуальне середовище — окреме середовище Python зі своїм набором установлених пакетів.

venv дає змогу різним проєктам використовувати різні залежності. Команди нижче виконують у терміналі, а не в Python-файлі.

Створення

```
python -m venv .venv
```

У каталозі проєкту створюється .venv. Стандартна бібліотека доступна, а сторонні пакети додаємо окремо.

Linux / macOS

```
source .venv/bin/activate
```

Активація змінює шлях до python для поточної оболонки. Після неї pip працює з цим середовищем.

Windows PowerShell

```
.venv\Scripts\Activate.ps1
```

Це команда активації для PowerShell. У IDE потрібно окремо вибрати інтерпретатор із .venv.

Активація не є обов’язковою: можна запускати інтерпретатор середовища за його повним шляхом.

pip і залежності: команди

Запис	Призначення	Важливий нюанс
`python -m pip --version`	Перевірка pip	Показує середовище
`python -m pip install pytest`	Установлення	У вибраному Python
`python -m pip show pytest`	Опис пакета	Версія та місце
`python -m pip freeze`	Знімок пакетів	Може містити зайве
`python -m pip freeze > requirements.txt`	Зберегти знімок	Команда оболонки
`python -m pip install -r requirements.txt`	Відновити залежності	Прочитати список

Команди виконуються в терміналі, а не в тілі Python-файла.

Файли та шляхи

Прочитаємо й запишемо текст із явним керуванням ресурсами.

Від простого тексту перейдемо до структурованих записів.

Path: шлях як об’єкт

Path із pathlib представляє шлях до файла чи каталогу. Створення цього об’єкта саме по собі не відкриває файл.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
шлях = Path(каталог) / назва_файла
текст = шлях.read_text(encoding=кодування)
```

Каталог

Відносний шлях відраховується від поточного робочого каталогу.

Назва файла

Оператор / приєднує компонент шляху. Це операція Path, а не ділення.

Кодування (encoding)

read_text відкриває, читає й закриває текстовий файл.

Перед використанням виконайте from pathlib import Path.

Path: шлях не дорівнює файлу

Path дозволяє отримати назву й розширення, не відкриваючи файл.

PYTHON · ПРИКЛАД

```python
from pathlib import Path

path = Path("data") / "scores.csv"
print(path.name)
print(path.suffix)
print(path.is_absolute())
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
scores.csv
.csv
False
```

Пояснення

Об’єкт Path. Описує шлях; ще не читає дані. Відносний шлях. Відраховується від робочого каталогу.

open: режими й наслідки

Запис	Призначення	Важливий нюанс
`r`	Читання	Файл має існувати
`w`	Запис	Обрізає наявний файл
`a`	Дописування	Нові дані в кінці
`x`	Створення	Помилка, якщо вже існує
`b / t`	Байти / текст	rb повертає bytes
`+`	Читання та запис	r+ і w+ різні за обрізанням

Перед w перевіряємо, чи потрібно зберегти попередній вміст.

with: загальний запис роботи з файлом

with обмежує час роботи з відкритим файлом. Після виходу з блока файл закривається, зокрема за винятку.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
with open(шлях, режим, encoding=кодування) as файл:
    інструкції
```

Відкриття

шлях задає файл, режим — читання або запис, encoding — кодування.

Ресурс (as)

Ім’я файл посилається на відкритий потік усередині блока.

Тіло блока

Інструкції з відступом виконують читання або запис.

Приклади режимів: "r" — читання, "w" — перезапис, "a" — дописування.

with, UTF-8 і закриття

PYTHON · ПРИКЛАД

```python
from pathlib import Path
from tempfile import TemporaryDirectory
with TemporaryDirectory() as folder:
    path = Path(folder) / "hello.txt"
    path.write_text(
        "Вітаю, студенти!", encoding="utf-8"
    )
    with path.open(encoding="utf-8") as file:
        print(file.read())
    print(file.closed)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Вітаю, студенти!
True
```

Пояснення

Файл закривається після виходу з with. TemporaryDirectory прибирає власні файли.

with, UTF-8 і закриття: пояснення

Приклад записує текст у UTF-8, читає його та перевіряє закриття файла.

TemporaryDirectory керує тимчасовим каталогом. Внутрішній with керує відкритим файлом. Обидва ресурси звільняються після своїх блоків.

Шлях

```
path = (Path(folder)
        / "hello.txt")
```

Об’єкт Path задає файл у тимчасовому каталозі folder. Оператор / приєднує назву файла.

Читання

```
print(file.read())
```

Усередині with читається записаний текст «Вітаю, студенти!». encoding="utf-8" узгоджує читання із записом.

Закриття

```
print(file.closed)
```

Після внутрішнього with властивість closed дорівнює True. Каталог існує до виходу із зовнішнього with.

Результат: «Вітаю, студенти!» і True. Після зовнішнього with тимчасовий каталог видаляється.

Текстовий потік: читання й запис

Текстовий потік працює з рядками str і зберігає поточну позицію. Кожне читання продовжується з цієї позиції.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
текст = файл.read()
рядок = файл.readline()
кількість = файл.write(текст)
файл.writelines(рядки)
файл.seek(позиція)
```

Читання

read читає решту тексту; readline — один рядок, зазвичай разом із \n.

Запис

write повертає кількість символів. writelines не додає роздільників.

Позиція

seek(0) повертає на початок. Інші зміщення для тексту мають обмеження.

Для запису файл має бути відкритий у відповідному режимі.

Читання всього тексту й рядків

Послідовне читання змінює позицію потоку; seek(0) повертає її на початок.

PYTHON · ПРИКЛАД

```python
from io import StringIO

file = StringIO("Олена,85\nІгор,45\n")
print(repr(file.readline()))
print(repr(file.read()))
file.seek(0)
for line in file:
    print(line.rstrip("\n"))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
'Олена,85\n'
'Ігор,45\n'
Олена,85
Ігор,45
```

Пояснення

Позиція читання. read продовжує після readline. Перехід на новий рядок. repr дозволяє побачити символ \n.

write та writelines

write додає один рядок, а writelines записує послідовність рядків.

PYTHON · ПРИКЛАД

```python
from io import StringIO

file = StringIO()
count = file.write("Звіт\n")
file.writelines(["85", "45"])
print(count)
print(repr(file.getvalue()))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
5
'Звіт\n8545'
```

Пояснення

write. Приймає str; повертає кількість символів. writelines. Не додає роздільників чи нових рядків.

Path і файли: довідка

Запис	Призначення	Важливий нюанс
`path.exists()`	Перевірка наявності	Не гарантує подальшого доступу
`path.is_file()`	Це звичайний файл?	Каталог — не файл
`path.mkdir(parents=True)`	Створення каталогів	exist_ok=True для наявного
`path.glob("*.txt")`	Пошук у каталозі	Перебір знайдених шляхів
`read_text / write_text`	Текст цілком	Явне encoding
`read_bytes / write_bytes`	Бінарні дані	Працюють з bytes

Для потокового читання використовуємо open і перебір рядків.

Очікувана файлова помилка

PYTHON · ПРИКЛАД

```python
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as folder:
    path = Path(folder) / "missing.txt"
    try:
        text = path.read_text(
            encoding="utf-8"
        )
    except FileNotFoundError:
        print("Вхідний файл не знайдено")
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Вхідний файл не знайдено
```

Пояснення

Відсутній файл. Перехоплюємо FileNotFoundError. Інші причини. Доступ і кодування потребують інших рішень.

CSV, JSON та збереження записів

Виберемо формат і відділимо читання від предметної валідації.

Після завантаження даних побудуємо об’єктну модель.

Таблиця CSV чи структура JSON

CSV зберігає таблицю записів. JSON представляє дані як масиви, об’єкти та окремі значення.

CSV

Рядки й колонки

```
name,score
Олена,85
Ігор,45
```

Перший рядок задає колонки name і score. Кожний наступний рядок — окремий запис. Типи полів визначає програма після читання.

JSON

Вкладені масиви й об’єкти

```
[{"name": "Олена", "score": 85},
 {"name": "Ігор", "score": 45}]
```

Квадратні дужки об’єднують два записи в масив. Кожний об’єкт у фігурних дужках містить ім’я та числову оцінку.

Формат файла не гарантує відповідності правилам предметної області.

json: типи та функції

Запис	Призначення	Важливий нюанс
`dict → object`	Пари ключ–значення	Ключі JSON — рядки
`list → array`	Послідовність	tuple також стає масивом
`str, int, float`	Текст і числа	Довільний клас не підтримується
`True / False / None`	true / false / null	Різний запис у двох мовах
`loads / dumps`	Рядок ↔ дані	Літера s: string
`load / dump`	Файл ↔ дані	Потрібний файловий об’єкт

Стандартний JSON не зберігає довільний об’єкт Python без перетворення.

JSON: яку функцію вибрати

Функцію вибирають за джерелом або місцем збереження JSON — текстова змінна чи відкритий файл.

Рядок

JSON-текст у змінній

```
текст = json.dumps(дані)
дані = json.loads(текст)
```

dumps повертає JSON-текст типу str. loads читає цей текст і відновлює дані Python. Файл ці функції не відкривають.

Файл

Відкритий файловий об’єкт

```
json.dump(дані, файл)
дані = json.load(файл)
```

dump записує JSON у відкритий файл. load читає з нього дані. Файл попередньо відкривають у відповідному режимі.

s у dumps / loads означає string. Для dump / load передаємо відкритий файл.

JSON: український текст

PYTHON · ПРИКЛАД

```python
import json

record = {"name": "Олена", "score": 85}
text = json.dumps(
    record, ensure_ascii=False, indent=2
)
print(text)
restored = json.loads(text)
print(restored["score"] + 5)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
{
  "name": "Олена",
  "score": 85
}
90
```

Пояснення

ensure_ascii=False. Залишає українські символи читабельними. indent=2. Відступи в JSON, а не блоки Python.

JSON-файл: запис і читання

PYTHON · ПРИКЛАД

```python
import json
from pathlib import Path
from tempfile import TemporaryDirectory
with TemporaryDirectory() as folder:
    path = Path(folder) / "report.json"
    with path.open("w", encoding="utf-8") as file:
        json.dump(
            {"average": 65.0}, file, indent=2
        )
    with path.open(encoding="utf-8") as file:
        report = json.load(file)
    print(report["average"])
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
65.0
```

Пояснення

Працюють з відкритим файлом. Кожен with завершує свою операцію.

JSON-файл: запис і читання: пояснення

json.dump записує дані у файл, а json.load відновлює об’єкт із відкритого файла.

У прикладі report.json зберігає словник із середнім 65.0. Кожний файловий потік відкривається у власному блоці with.

Запис

```
json.dump(
    {"average": 65.0}, file
)
```

file відкрито в режимі "w". dump передає JSON-текст безпосередньо в потік. У повному прикладі indent=2 задає відступи.

Читання

```
report = json.load(file)
```

file відкрито для читання. load читає JSON-текст і повертає словник Python.

Доступ до результату

```
print(report["average"])
```

За ключем average отримуємо число 65.0. Кодування UTF-8 задане під час відкриття обох потоків.

dump/load працюють із файловим потоком; dumps/loads — із рядком. Потоки закриває with.

CSV: рядки, словники й заголовок

CSV зберігає табличні дані як текстові поля. Модуль csv обробляє роздільники й лапки; числові типи задають окремо.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
csv.reader(файл)
csv.DictReader(файл)
csv.DictWriter(файл, fieldnames=назви_полів)
```

Список полів

reader повертає рядок як список. Заголовок потрібно прочитати окремо.

Словник полів

DictReader типово бере ключі з першого рядка таблиці.

Запис словників

DictWriter використовує fieldnames для назв і порядку колонок.

Файли CSV відкривайте з newline="" та узгодженим encoding. Потрібен import csv.

CSV: заголовок і текстові поля

PYTHON · ПРИКЛАД

```python
import csv
from io import StringIO
source = StringIO(
    "name,score\nОлена,85\nІгор,45\n"
)
reader = csv.reader(source)
print(next(reader))
for name, text in reader:
    score = int(text)
    print(name, score + 5)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
['name', 'score']
Олена 90
Ігор 50
```

Пояснення

Повертає поля рядка як текст. int виконуємо явно; можливий ValueError.

CSV: заголовок і текстові поля: пояснення

CSV містить заголовок і текстові поля. Оцінки потрібно перетворити на числа перед обчисленням.

csv.reader читає записи зі StringIO. Перший запис — назви колонок, наступні містять ім’я та оцінку. Фрагменти належать попередньому прикладу.

Заголовок

```
print(next(reader))
```

next читає перший запис ["name", "score"]. Цикл починається з наступного запису.

Перетворення

```
score = int(text)
```

Рядки "85" і "45" стають цілими числами 85 та 45. Некоректний числовий текст спричинив би ValueError.

Обчислення

```
print(name, score + 5)
```

Числове додавання дає «Олена 90» та «Ігор 50». До рядка text число 5 додати не можна.

reader повертає рядкові поля. Типи та допустимі значення перевіряє програма, а не CSV-формат.

DictReader: доступ за назвою поля

PYTHON · ПРИКЛАД

```python
import csv
from io import StringIO

source = StringIO(
    "name,score\nОлена,85\nІгор,45\n"
)
for row in csv.DictReader(source):
    name = row["name"]
    score = int(row["score"])
    print(name, score >= 60)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Олена True
Ігор False
```

Пояснення

Заголовок. Назви колонок стають ключами словника. Валідація. Назва ключа не гарантує правильного типу.

DictWriter: явний порядок полів

PYTHON · ПРИКЛАД

```python
import csv
from io import StringIO

output = StringIO(newline="")
writer = csv.DictWriter(
    output, fieldnames=["name", "score"]
)
writer.writeheader()
writer.writerow(
    {"score": 85, "name": "Олена"}
)
print(output.getvalue().strip())
```

КОНСОЛЬ · РЕЗУЛЬТАТ

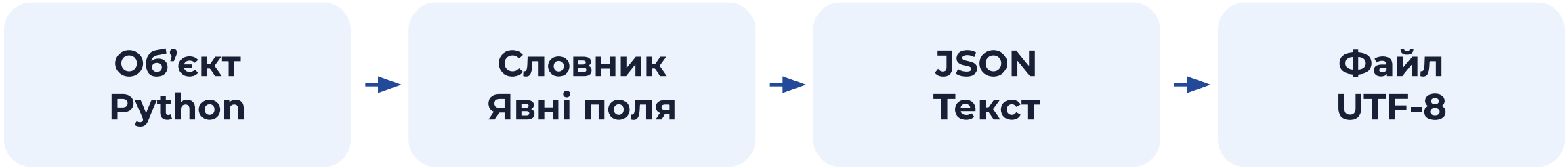
```
name,score
Олена,85
```

Пояснення

fieldnames. Задає порядок і назви колонок. writeheader. Окремо записує рядок заголовка.

Об’єкт і зовнішній формат

Для збереження даних обираємо явні поля, перетворюємо їх на JSON-текст і записуємо у файл.




```
data = {"name": "Олена", "score": 85}
text = json.dumps(data, ensure_ascii=False)
```

Стрілки — послідовність перетворень. Потрібен import json. pickle може виконувати код під час завантаження, тому не читаємо недовірені файли.

Клас та екземпляр

Об’єднаємо стан і пов’язану з ним поведінку.

Від створення об’єкта перейдемо до правил доступу.

Функції чи об’єктна модель

Функції та класи по-різному організовують роботу з даними. Вибір залежить від структури програми.

Функції + дані

Дії отримують аргументи

```
average(scores)
validate(score)
```

Функції отримують оцінки через аргументи. Це зручно для послідовності незалежних перетворень і розрахунків.

Об’єкт

Стан і методи разом

```
student.add_score(85)
student.average()
```

Об’єкт зберігає оцінки у власному стані. Методи додають оцінку та обчислюють середнє саме для цього студента.

ООП — спосіб організації; не кожен скрипт потребує класів.

Клас, екземпляр, атрибути та методи

Клас визначає власний тип. Екземпляр — окремий об’єкт цього класу з конкретними значеннями атрибутів.

Атрибут зберігає дані об’єкта. Метод виконує дію зі станом об’єкта. Нижче — фрагменти класу Student, який далі реалізуємо повністю.

Клас

```
class Student:
```

Заголовок починає визначення типу Student. Атрибути та методи записують у тілі класу з відступом.

Екземпляр

```
student = Student("Олена")
```

Після визначення класу цей виклик створить об’єкт. __init__ заповнить його початковий стан.

Доступ

```
student.name
student.average()
```

Без дужок читаємо атрибут name. Із дужками викликаємо метод average та отримуємо результат.

Склад аргументів Student(...) залежить від визначення __init__. Імена атрибутів і методів визначає клас.

Будова класу: атрибути та методи

Клас описує стан і поведінку об’єктів. __init__ ініціалізує створений екземпляр; self посилається на цей екземпляр.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
class НазваКласу:
    def __init__(self, значення):
        self.атрибут = значення
    def метод(self):
        return self.атрибут
```

Тип об’єкта

class задає новий тип. НазваКласу(аргумент) створює його екземпляр.

Стан

self.атрибут зберігає дані поточного екземпляра.

Поведінка

метод викликають як екземпляр.метод(). self передається автоматично.

Відступи у схемі: 4 пробіли — тіло класу, 8 — тіло методу.

__init__ та self

PYTHON · ПРИКЛАД

```python
class Student:
    def __init__(self, name):
        self.name = name
        self.scores = []

olena = Student("Олена")
ihor = Student("Ігор")
print(olena.name, ihor.name)
print(olena.scores is ihor.scores)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Олена Ігор
False
```

Пояснення

__init__ задає name і створює новий список scores для кожного екземпляра. self посилається на об’єкт поточного виклику. olena.scores is ihor.scores повертає False, бо це два різні списки.

Атрибути класу та екземпляра

Атрибут класу задає спільне значення. Атрибут екземпляра зберігає стан конкретного об’єкта.

Клас

Спільне правило

```
class Student:
    pass_mark = 60
```

Student.pass_mark = 60 читають екземпляри без власного pass_mark.

Екземпляр

Окремий стан

```
class Student:
    def __init__(self, name):
        self.name = name
        self.scores = []
```

__init__ записує name та створює self.scores. Кожний екземпляр отримує окремий список.

Змінний атрибут класу може ненавмисно стати спільним станом.

self: від визначення до виклику

self — параметр методу, що посилається на екземпляр. Python передає його автоматично під час звичайного виклику.

Визначення

Параметри методу

```
def метод(self, аргумент):
    інструкції
```

self отримує екземпляр, аргумент — передане значення. Імена тут є українськими позначеннями схеми.

Виклик

Рівнозначні форми виклику

```
екземпляр.метод(значення)
Клас.метод(екземпляр, значення)
```

Перша форма передає екземпляр неявно, друга — явно. Йдеться про звичайний метод екземпляра.

self — прийняте ім’я, а не ключове слово. Воно дозволяє читати та змінювати атрибути поточного об’єкта.

Метод працює зі станом

PYTHON · ПРИКЛАД

```python
class Student:
    def __init__(self, name):
        self.name = name
        self.scores = []
    def add_score(self, score):
        if not 0 <= score <= 100:
            raise ValueError(
                "Оцінка поза діапазоном"
            )
        self.scores.append(score)
s = Student("Олена")
s.add_score(85)
print(s.scores)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
[85]
```

Пояснення

s передається як self автоматично. Зміна стану відбувається після перевірки.

Метод працює зі станом: пояснення

Метод add_score перевіряє оцінку та змінює список конкретного екземпляра Student.

У прикладі s — об’єкт для Олени. Виклик s.add_score(85) передає цей об’єкт як self, а число 85 — як score.

Виклик

```
s.add_score(85)
```

Метод отримує доступ до стану s через self. Ім’я self не позначає клас загалом.

Перевірка

```
if not 0 <= score <= 100:
```

За хибної допустимості наступний raise породжує ValueError. До append виконання не доходить.

Зміна стану

```
self.scores.append(score)
```

Для 85 перевірка проходить. Оцінка додається до списку саме цього об’єкта, і print(s.scores) дає [85].

Перевірка стоїть перед зміною стану. Недопустима оцінка не додається до списку.

Два незалежні екземпляри

PYTHON · ПРИКЛАД

```python
class Student:
    def __init__(self, name, scores):
        self.name = name
        self.scores = list(scores)
    def average(self):
        if not self.scores:
            raise ValueError("Немає оцінок")
        return sum(self.scores) / len(self.scores)
a = Student("Олена", [80, 90])
b = Student("Ігор", [45])
a.scores.append(100)
print(a.average(), b.average())
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
90.0 45.0
```

Пояснення

list(scores) створює власний контейнер. Метод має явне правило для порожніх даних.

Два незалежні екземпляри: пояснення

Екземпляри a та b зберігають окремі списки оцінок і обчислюють власне середнє.

a отримує оцінки [80, 90], b — [45]. Конструктор копіює переданий список. Метод average перевіряє, що оцінки є.

Окремий контейнер

```
self.scores = list(scores)
```

list створює новий список. Зміни контейнера всередині одного екземпляра не змінюють список іншого.

Зміна a

```
a.scores.append(100)
```

Оцінки a стають [80, 90, 100]. Список b залишається [45].

Обчислення

```
a.average()
b.average()
```

Середнє a дорівнює 270 / 3 = 90.0. Середнє b — 45.0. Для порожнього списку виникне ValueError.

Консоль: 90.0 45.0. Копія контейнера є поверхневою; у цьому прикладі його елементи — числа.

Доступ і різновиди методів

Задамо інваріанти й зрозумілий інтерфейс класу.

Від окремого класу перейдемо до зв’язків між класами.

Інкапсуляція та інваріант

Інкапсуляція поєднує стан об’єкта з методами та визначає, як із цим станом працюють.

Інваріант — правило, яке зберігають допустимі операції. Для оцінки це 0 ≤ score ≤ 100. Назва атрибута сама по собі не перевіряє діапазон.

Публічний атрибут

```
self.score
```

score — частина інтерфейсу. Дані змінюють через передбачені класом операції, які мають зберігати інваріант.

Внутрішній атрибут

```
self._score
```

_score позначає внутрішній стан за домовленістю. Технічно зовнішній доступ можливий; перевірку реалізують у коді.

Подвійне підкреслення

```
self.__score
```

__score змінюється на ім’я на зразок _Result__score. Це зменшує конфлікти імен у підкласах, а не гарантує приватність.

Контроль значення виконує метод або setter. Підкреслення не є механізмом перевірки.

Методи й доступ: довідка

Запис	Призначення	Важливий нюанс
`def method(self, x)`	Метод екземпляра	obj.method(x)
`@classmethod`	Метод класу	Перший параметр cls
`@staticmethod`	Статичний метод	Немає self або cls
`@property`	Читання як атрибута	obj.score
`@score.setter`	Контроль присвоєння	obj.score = value
`class E(ValueError)`	Власний виняток	Конкретна предметна причина

Вид методу визначає контекст, який він отримує автоматично.

property: читання і присвоєння

property керує доступом до атрибута через методи. Зовні використовують об’єкт.атрибут, без виклику дужками.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
@property
def атрибут(self):
    return self._значення
@атрибут.setter
def атрибут(self, нове):
    self._значення = нове
```

Читання

@property задає метод отримання значення (getter).

Присвоєння

@атрибут.setter задає метод для нового значення (setter).

Внутрішній стан

_значення зберігає дані. Перед присвоєнням їх можна перевірити.

У цій схемі методи належать класу; тіло кожного методу має додатковий відступ.

property: кероване присвоєння

PYTHON · ПРИКЛАД

```python
class Result:
    def __init__(self, score):
        self.score = score
    @property
    def score(self):
        return self._score
    @score.setter
    def score(self, value):
        if not 0 <= value <= 100:
            raise ValueError(
                "Оцінка поза діапазоном"
            )
        self._score = value
r = Result(85)
r.score = 90
print(r.score)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
90
```

Пояснення

property викликає getter. setter перевіряє нове значення.

property: кероване присвоєння: пояснення

Властивість score поєднує звичний доступ до атрибута з перевіркою під час присвоєння.

Result зберігає число у _score. Читання score викликає getter, а присвоєння score — setter, що перевіряє межі 0–100.

Читання

```
print(r.score)
```

@property дозволяє читати значення без дужок. Getter повертає self._score.

Присвоєння

```
r.score = 90
```

Викликається setter, позначений @score.setter. Число 90 проходить перевірку діапазону.

Збереження

```
self._score = value
```

Цей рядок виконується в setter після перевірки. Саме внутрішній атрибут зберігає нове число.

Результат: 90. Початкова ініціалізація через self.score також проходить перевірку setter.

classmethod: створення з даних

PYTHON · ПРИКЛАД

```python
class Student:
    def __init__(self, name, score):
        self.name = name
        self.score = score
    @classmethod
    def from_dict(cls, record):
        return cls(
            record["name"], int(record["score"])
        )
s = Student.from_dict(
    {"name": "Олена", "score": "85"}
)
print(s.name, s.score)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Олена 85
```

Пояснення

Отримує клас, для якого викликали метод. from_dict читає узгоджену структуру.

classmethod: створення з даних: пояснення

classmethod створює Student із даних словника, передаючи клас через параметр cls.

У словнику name містить "Олена", а score — рядок "85". Метод from_dict готує аргументи та повертає новий екземпляр.

Клас замість self

```
@classmethod
```

Декоратор передає Student як cls. Для цього виклику не потрібний заздалегідь створений екземпляр.

Числове поле

```
int(record["score"])
```

Текст "85" перетворюється на число 85. Некоректний текст спричинить ValueError.

Створення

```
return cls(
    record["name"],
    int(record["score"]))
```

cls викликається як конструктор. Створений об’єкт має name="Олена" та score=85.

Консоль: Олена 85. Перетворення типу саме по собі не перевіряє допустимий діапазон оцінки.

staticmethod: без self та cls

Статичний метод перевіряє дані без доступу до екземпляра чи класу.

PYTHON · ПРИКЛАД

```python
class ScoreRules:
    @staticmethod
    def is_valid(value):
        return 0 <= value <= 100

print(ScoreRules.is_valid(85))
print(ScoreRules.is_valid(110))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
True
False
```

Пояснення

Без неявного аргументу. Передаємо лише value. Альтернатива. Окрема функція модуля теж доречна.

Власний виняток

PYTHON · ПРИКЛАД

```python
class InvalidScoreError(ValueError):
    pass
def validate(score):
    if not 0 <= score <= 100:
        raise InvalidScoreError("Оцінка поза 0–100")
    return score
try:
    validate(110)
except InvalidScoreError as error:
    print(type(error).__name__)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
InvalidScoreError
```

Пояснення

Новий тип є різновидом ValueError. Зовнішній код може обробити саме цю причину.

Власний виняток: пояснення

Власний InvalidScoreError уточнює, що помилка пов’язана з недопустимою оцінкою.

Клас винятку успадковує ValueError. validate(110) порушує межі 0–100, тому замість повернення оцінки породжується виняток.

Тип помилки

```
class InvalidScoreError(
        ValueError):
    pass
```

Новий тип позначає конкретну помилку, залишаючись різновидом ValueError.

Недопустиме значення

```
validate(110)
```

Число 110 не проходить перевірку. raise усередині validate перериває звичайне виконання.

Перехоплення

```
except InvalidScoreError:
```

Обробник отримує об’єкт винятку як error. type(error).__name__ повертає InvalidScoreError.

Цю помилку перехопить і except ValueError. Специфічні обробники ставлять перед загальними.

Наслідування та композиція

Зіставимо «є різновидом» і «містить» у моделях.

Далі під’єднаємо об’єкти до стандартних операцій Python.

Наслідування: що отримує похідний клас

Наслідування дозволяє похідному класу використовувати та доповнювати поведінку базового.

У моделі Person описує людину, а Student — її різновид. Студент має спільні властивості людини й додаткові дані для навчання.

Базовий клас

```
class Person:
    def summary(self):
```

Person визначає спільну поведінку, наприклад подання імені через summary. Тут показано заголовки, а не повні класи.

Похідний клас

```
class Student(Person):
    def average(self):
```

Назва Person у дужках задає базовий клас. Student може додати average або перевизначити summary.

Успадкований метод

```
student.summary()
```

Якщо Student не перевизначає summary, використовується реалізація Person. Власний average додає навчальні обчислення.

Наслідування має описувати зв’язок моделі, а не лише економію рядків.

Наслідування: базовий і похідний клас

Похідний клас використовує поведінку базового й може перевизначити його методи. super() дає доступ до наступної реалізації.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
class Похідний(Базовий):
    def метод(self):
        результат = super().метод()
        return результат
```

Базовий клас

Назва в дужках задає, від якого класу успадковують поведінку.

Перевизначення

Метод із тим самим ім'ям задає реалізацію похідного класу.

Виклик super

За простого наслідування звертається до базового методу.

Наслідування описує відношення «є різновидом»; композиція — «містить».

Перевизначення та super

PYTHON · ПРИКЛАД

```python
class Person:
    def __init__(self, name):
        self.name = name
    def summary(self):
        return self.name
class Student(Person):
    def __init__(self, name, score):
        super().__init__(name)
        self.score = score
    def summary(self):
        return f"{super().summary()}: {self.score}"
print(Student("Олена", 85).summary())
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Олена: 85
```

Пояснення

Викликає наступну реалізацію за MRO. Student уточнює summary.

Перевизначення та super: пояснення

Student розширює Person: зберігає ім’я та оцінку і перевизначає метод summary.

super звертається до наступної реалізації за порядком MRO. У цьому простому прикладі наступний клас — Person.

Ініціалізація

```
super().__init__(name)
```

Метод Person.__init__ записує self.name у той самий екземпляр Student.

Власний стан

```
self.score = score
```

Student додає до успадкованого стану оцінку. Для прикладу вона дорівнює 85.

Спільна поведінка

```
super().summary()
```

Метод Person повертає "Олена". Перевизначений summary додає до цього імені оцінку.

Student("Олена", 85).summary() повертає «Олена: 85». super не створює окремий об’єкт Person.

Поліморфізм і duck typing

PYTHON · ПРИКЛАД

```python
class Student:
    def summary(self):
        return "Студент: Олена"

class Group:
    def summary(self):
        return "Група: ІПЗ-1"

for item in [Student(), Group()]:
    print(item.summary())
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Студент: Олена
Група: ІПЗ-1
```

Пояснення

Student і Group не мають спільного базового класу, але обидва реалізують summary(). Цикл працює з їхньою спільною поведінкою. Це duck typing — використання потрібних операцій незалежно від назви типу.

Композиція: група містить студентів

PYTHON · ПРИКЛАД

```python
class Student:
    def __init__(self, name):
        self.name = name
class Group:
    def __init__(self, students):
        self.students = list(students)
    def names(self):
        return [s.name for s in self.students]
group = Group([
    Student("Олена"), Student("Ігор")
])
print(group.names())
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
['Олена', 'Ігор']
```

Пояснення

Group містить об’єкти Student. Самих студентів глибоко не копіюємо.

Композиція: група містить студентів: пояснення

Group містить об’єкти Student. Це композиція, а не наслідування від Student.

Контейнер студентів належить групі, але його елементи — посилання на передані об’єкти. Копія списку не створює копій студентів.

Контейнер групи

```
self.students =
list(students)
```

Group створює свій список. Зміна цього списку не змінює початковий контейнер.

Доступ до імен

```
s.name
```

У методі names ім’я s по черзі посилається на кожний об’єкт Student зі списку.

Результат

```
print(group.names())
```

Метод збирає імена у новий список. У консолі отримуємо ["Олена", "Ігор"].

Група має студентів, але сама не є студентом. Посилання на об’єкти Student залишаються спільними.

ABC: явний абстрактний контракт

PYTHON · ПРИКЛАД

```python
from abc import ABC, abstractmethod
class Report(ABC):
    @abstractmethod
    def summary(self):
        pass
class TextReport(Report):
    def summary(self):
        return "Звіт готовий"
print(TextReport().summary())
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Звіт готовий
```

Пояснення

Базовий абстрактний тип. Похідний клас визначає summary.

ABC: явний абстрактний контракт: пояснення

Абстрактний Report задає вимогу до методу summary, а TextReport реалізує цей метод.

ABC використовують, коли потрібна явна вимога до конкретних класів. Для простого навчального класу цей механізм не обов’язковий.

Контракт

```
@abstractmethod
def summary(self):
```

У Report цей метод позначений як абстрактний. Клас із нереалізованим методом не можна створити.

Реалізація

```
class TextReport(Report):
```

У тілі TextReport метод summary перевизначений і повертає "Звіт готовий".

Виклик

```
TextReport().summary()
```

Конкретний клас можна створити та викликати його метод. print виводить «Звіт готовий».

Report() у наведеному визначенні породив би TypeError.

MRO: порядок пошуку методів

PYTHON · ПРИКЛАД

```python
class Base:
    pass
class Left(Base):
    pass
class Right(Base):
    pass
class Child(Left, Right):
    pass

print([c.__name__ for c in Child.__mro__])
```

Пояснення

__mro__ задає порядок пошуку методів: Child → Left → Right → Base → object. super() враховує цей порядок. Спільний клас Base входить до послідовності один раз.

КОНСОЛЬ · РЕЗУЛЬТАТ

```
['Child', 'Left', 'Right', 'Base', 'object']
```

Протоколи та dataclass

Зв’яжемо об’єкти з print, len, перебором і збереженням.

Готову модель використаємо у звіті та бібліотеках.

Спеціальні методи: довідка

Запис	Призначення	Важливий нюанс
`__init__`	Ініціалізація	Повертає None
`__str__`	str(obj), print(obj)	Повертає str
`__repr__`	repr(obj)	Діагностичний str
`__len__`	len(obj)	Невід’ємне ціле
`__iter__`	iter(obj), for	Повертає ітератор
`__eq__`	obj == other	Узгоджена рівність

Протокол задає очікуваний контракт, а не тільки назву методу.

__str__ та __repr__

PYTHON · ПРИКЛАД

```python
class Student:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return f"Студент: {self.name}"
    def __repr__(self):
        return f"Student({self.name!r})"
s = Student("Олена")
print(s)
print(repr(s))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Студент: Олена
Student('Олена')
```

Пояснення

Читабельне представлення для користувача. Однозначне представлення для діагностики.

__str__ та __repr__: пояснення

Один об’єкт може мати подання для користувача та окреме діагностичне подання.

__str__ використовується для читабельного тексту, __repr__ — для діагностики. У прикладі обидва методи визначені в Student.

Для користувача

```
print(s)
```

Викликається __str__. Результат — «Студент: Олена».

Для діагностики

```
print(repr(s))
```

Викликається __repr__. Результат — Student('Олена'). Це рядок, який описує об’єкт.

Лапки в поданні

```
f"Student({self.name!r})"
```

!r застосовує repr до імені. Тому рядок "Олена" у діагностичному записі має лапки.

__str__ та __repr__ мають повертати str. repr не обов’язково є виконуваним кодом.

__len__ та __iter__

PYTHON · ПРИКЛАД

```python
class Group:
    def __init__(self, names):
        self.names = list(names)
    def __len__(self):
        return len(self.names)
    def __iter__(self):
        return iter(self.names)
g = Group(["Олена", "Ігор"])
print(len(g))
print(list(g))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
2
['Олена', 'Ігор']
```

Пояснення

Викликає протокол довжини. __iter__ повертає ітератор.

__len__ та __iter__: пояснення

Спеціальні методи дозволяють Group підтримувати стандартні len і перебір елементів.

Об’єкт g містить імена ["Олена", "Ігор"]. Python викликає __len__ для len(g), а __iter__ — під час отримання ітератора.

Кількість

```
return len(self.names)
```

Цей рядок належить __len__. Для двох імен метод повертає ціле число 2.

Ітератор

```
return iter(self.names)
```

Цей рядок належить __iter__. Перебір делегується внутрішньому списку імен.

Використання

```
print(len(g))
print(list(g))
```

Перший виклик друкує 2. Другий збирає елементи об’єкта у список і друкує імена.

Group підтримує ці операції через протокол методів. Наслідування від list для цього не потрібне.

dataclass: клас для зберігання даних

Декоратор dataclass генерує типові методи за оголошеними полями. Методи перевірки й обчислення можна додати самостійно.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
@dataclass
class НазваКласу:
    поле: тип
    список: list = field(default_factory=list)
```

Декоратор

Типово додає __init__, __repr__ та __eq__. Не перевіряє типи автоматично.

Поля

Анотація поле: тип описує дані. Порядок полів впливає на __init__.

Окремі списки

default_factory=list створює новий список для кожного екземпляра.

Потрібен from dataclasses import dataclass, field. Змінні типові значення потребують уваги.

dataclass і default_factory

PYTHON · ПРИКЛАД

```python
from dataclasses import dataclass, field
@dataclass
class Student:
    name: str
    scores: list[int] = field(
        default_factory=list
    )
a = Student("Олена")
b = Student("Ігор")
a.scores.append(85)
print(a)
print(b.scores)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Student(name='Олена', scores=[85])
[]
```

Пояснення

dataclass формує службові методи. default_factory створює окремий список для кожного екземпляра.

dataclass і default_factory: пояснення

dataclass формує службові методи, а default_factory створює окремий список для кожного об’єкта.

У Student поле name зберігає ім’я. Поле scores отримує новий список, якщо під час створення екземпляра його не передали.

Службові методи

```
@dataclass
class Student:
```

@dataclass за полями генерує __init__, __repr__ та __eq__. Метод __repr__ дає читабельне подання об’єкта під час print(a).

Новий список

```
field(default_factory=list)
```

list викликається для кожного нового екземпляра. Фрагмент задає типове значення поля scores.

Незалежний стан

```
a.scores.append(85)
print(b.scores)
```

Змінюється список a. Список b залишається порожнім, і другий print виводить [].

Анотація list[int] описує тип, але dataclass автоматично не перевіряє типи чи діапазон оцінок.

asdict: підготовка до JSON

PYTHON · ПРИКЛАД

```python
from dataclasses import dataclass, asdict
import json
@dataclass
class Result:
    name: str
    score: int
r = Result("Олена", 85)
data = asdict(r)
print(data["score"])
print(json.dumps(data, ensure_ascii=False))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
85
{"name": "Олена", "score": 85}
```

Пояснення

Перетворює dataclass на словник. Поля мають бути придатними для JSON.

asdict: підготовка до JSON: пояснення

asdict готує поля dataclass для подальшого перетворення у JSON.

r — Result із name="Олена" та score=85. Спочатку отримуємо словник Python, потім перетворюємо його на JSON-рядок.

Поля об’єкта

```
data = asdict(r)
```

asdict створює словник із полів dataclass. Ключами стають назви name та score.

Значення поля

```
print(data["score"])
```

Доступ до словника за ключем повертає число 85. Це ще об’єкт Python, а не JSON-текст.

JSON-текст

```
json.dumps(data,
    ensure_ascii=False)
```

dumps повертає рядок. ensure_ascii=False залишає українські символи у читабельному вигляді.

asdict не гарантує придатність усіх значень для JSON. Непідтримувані типи перетворюють окремо.

Перевірка незалежності й валідації

PYTHON · ПРИКЛАД

```python
from dataclasses import (
    dataclass, field
)
@dataclass
class Student:
    scores: list[int] = field(
        default_factory=list
    )
    def add_score(self, value):
        if not 0 <= value <= 100:
            raise ValueError(
                "Оцінка поза "
                "діапазоном"
            )
        self.scores.append(value)
```

PYTHON · ПРИКЛАД

```python
a, b = Student(), Student()
a.add_score(85)
assert b.scores == []
try:
    a.add_score(110)
except ValueError:
    assert a.scores == [85]
else:
    raise AssertionError(
        "Очікували ValueError"
    )
print(
    "Стан незалежний; "
    "помилку відхилено"
)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Стан незалежний; помилку відхилено
```

Перевірка незалежності й валідації: пояснення

Перевірки підтверджують незалежність списків та відхилення недопустимої оцінки.

Після a.add_score(85) у b не має бути оцінок. Спроба додати 110 має породити ValueError і залишити стан a без змін.

Незалежність

```
assert b.scores == []
```

Перевіряємо, що зміна a не торкнулася b. Це тест поведінки окремих контейнерів.

Збереження стану

```
assert a.scores == [85]
```

Це твердження стоїть у except ValueError після спроби a.add_score(110). Число 110 не додано.

Обов’язковий виняток

```
raise AssertionError(
    "Очікували ValueError")
```

Цей рядок належить else блока try. Він позначає провал перевірки, якщо ValueError не виник.

За успіху друкується «Стан незалежний; помилку відхилено». assert не використовують для даних користувача.

Стандартна бібліотека та прикладні інструменти

Зіставимо типові задачі з потрібними засобами.

Окремо розглянемо дані, штучний інтелект і конкурентність.

Математика й випадкові значення

Запис	Призначення	Важливий нюанс
`math.sqrt(x)`	Квадратний корінь	Для від'ємного потрібен інший підхід
`math.ceil / floor`	Округлення	Угору / униз
`math.pi`	Стала π	Значення float
`random.randint(a, b)`	Випадкове ціле	Обидві межі враховано
`random.choice(items)`	Вибір елемента	Непорожня послідовність
`secrets.token_hex(n)`	Секретний токен	n байтів → 2n hex-символів

random — для моделювання; secrets — для секретних випадкових значень.

Counter і timedelta

Counter підраховує повторення, timedelta представляє тривалість.

PYTHON · ПРИКЛАД

```python
from collections import Counter
from datetime import timedelta

words = ["python", "data", "python"]
counts = Counter(words)
print(counts["python"])
print(counts["web"])
print(timedelta(hours=48).days)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
2
0
2
```

Пояснення

Counter. Рахує входження елементів. timedelta. Представляє тривалість, а не часову мітку.

re та аргументи командного рядка

PYTHON · ПРИКЛАД

```python
import re
import argparse
print(re.findall(r"\d+", "Оцінки: 85 і 45"))
parser = argparse.ArgumentParser()
parser.add_argument(
    "--input", default="scores.csv"
)
args = parser.parse_args(["--input", "data.csv"])
print(args.input)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
['85', '45']
data.csv
```

Пояснення

\d+ знаходить послідовності цифр. Описує параметри запуску програми.

re та аргументи командного рядка: пояснення

re знаходить числові фрагменти тексту, а argparse отримує параметри запуску програми.

Приклад демонструє дві незалежні операції. Знайдені цифри повертаються як рядки; параметр --input задає ім’я вхідного файла.

Пошук

```
re.findall(r"\d+",
    "Оцінки: 85 і 45")
```

Шаблон \d+ означає одну або більше цифр. Результат — рядки ["85", "45"].

Параметр

```
parser.add_argument(
    "--input",
    default="scores.csv")
```

Описуємо іменований аргумент і його типове значення. Аргумент доступний як args.input.

Звичайний запуск

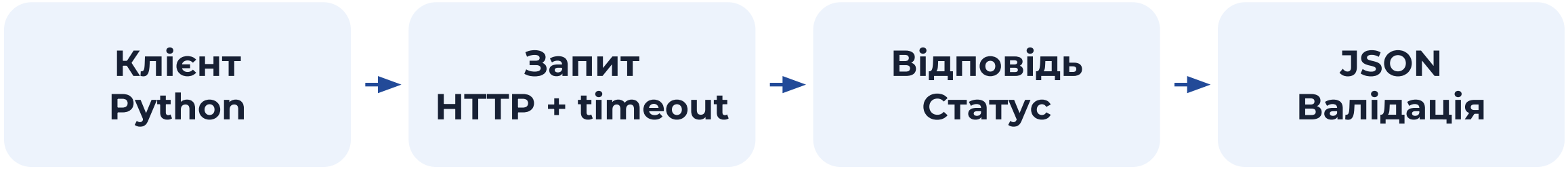
```
args = parser.parse_args()
```

Ця форма читає командний рядок. У повному прикладі список аргументів передано явно для демонстрації.

Повний приклад виводить ['85', '45'] і data.csv. Для числових обчислень застосовують int.

HTTP: від запиту до даних

HTTP-запит передає дані між Python-програмою та сервером. Відповідь потрібно перевірити перед використанням.



Стрілки показують етапи обміну: надсилання запиту → відповідь зі статусом → розбір JSON. timeout обмежує очікування; коректні статус і JSON не замінюють перевірку полів.

Автоматизація, веб і GUI

Запис	Призначення	Важливий нюанс
`Requests`	HTTP-клієнт	Потрібне встановлення
`Beautiful Soup`	Розбір HTML	Не виконує JavaScript браузера
`tkinter`	Настільний GUI	Доступність залежить від інсталяції
`Django / Flask`	Вебзастосунки	Сторонні фреймворки
`FastAPI`	HTTP API	Сторонній фреймворк
`pathlib + csv + json`	Локальна автоматизація	Стандартна бібліотека

Python застосовується і в клієнтських скриптах, і в серверній логіці.

sqlite3: параметризований SQL

PYTHON · ПРИКЛАД

```python
import sqlite3
from contextlib import closing
with closing(sqlite3.connect(":memory:")) as db:
    with db:
        db.execute(
            "CREATE TABLE scores "
            "(name TEXT, score INTEGER)"
        )
        db.executemany(
            "INSERT INTO scores VALUES (?, ?)",
            [("Олена", 85), ("Ігор", 45)],
        )
    result = db.execute(
        "SELECT AVG(score) FROM scores"
    )
    print(result.fetchone()[0])
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
65.0
```

Пояснення

Значення передаються окремо від SQL. with db — транзакція; closing — закриття.

sqlite3: параметризований SQL: пояснення

sqlite3 передає дані окремо від SQL та розділяє керування транзакцією і закриття з’єднання.

У таблицю scores записано оцінки 85 і 45. Знаки ? у команді INSERT — місця для значень, а не текстове склеювання SQL.

Транзакція

```
with db:
```

За нормального виходу зміни підтверджуються, за винятку відкочуються. Цей with не закриває з’єднання.

Закриття

```
with closing(sqlite3.connect(
        ":memory:")) as db:
```

closing завершує роботу з’єднання після блока. У прикладі база існує лише в пам’яті.

Середнє

```
print(result.fetchone()[0])
```

Запит SELECT AVG(score) повертає один рядок. Індекс 0 бере його перше поле — 65.0.

Значення для ? передають другим аргументом execute або через записи executemany.

Додаткові бібліотеки та можливості Python

Познайомимося із засобами для числових даних, таблиць і графіків та іншими можливостями Python.

Ці матеріали допоможуть вибрати інструменти для подальших навчальних проєктів.

Бібліотеки для даних і графіків

Кожна бібліотека розв’язує своє коло задач і доповнює базові засоби Python.

NumPy працює з числовими масивами, pandas — із таблицями, Matplotlib — із графічним поданням даних.

NumPy

```
import numpy as np
```

ndarray зберігає числовий масив. Операції можна застосувати одразу до його елементів, без ручного циклу.

pandas

```
import pandas as pd
```

DataFrame представляє таблицю з колонками. Бібліотека читає CSV, відбирає рядки й обчислює підсумки.

Matplotlib

```
from matplotlib import pyplot
```

pyplot надає функції для графіків. Можна показати залежність, порівняти значення й підписати осі.

У Google Colab ці бібліотеки зазвичай доступні. Кожний приклад починається з потрібного import.

NumPy-масив і список Python

list

Універсальна колекція

`[1, 2] * 2 → [1, 2, 1, 2]`
Може містити різні типи
Універсальна колекція

NumPy ndarray

Числовий масив

`np.array([1, 2]) * 2 → [2, 4]`
Тип і форма масиву
Операції над масивом

Для NumPy потрібен import numpy as np. * для list повторює елементи, для ndarray множить значення.

pandas: відбір і середнє

PYTHON · ПРИКЛАД

```python
import pandas as pd
from io import StringIO
source = StringIO(
    "name,score\nОлена,85\n"
    "Ігор,45\nМарія,110\n"
)
df = pd.read_csv(source)
clean = df[df["score"].between(0, 100)]
print(len(clean))
print(clean["score"].mean())
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
2
65.0
```

Пояснення

Таблиця з колонками та індексом. Потрібен пакет pandas у середовищі.

pandas: відбір і середнє: пояснення

pandas читає таблицю, відбирає допустимі оцінки та обчислює середнє її колонки.

df містить оцінки 85, 45 і 110. Булева маска позначає допустимі рядки, а clean містить лише відібрані записи.

Читання

```
df = pd.read_csv(source)
```

CSV читається у DataFrame з колонками name та score. source у прикладі — текстовий потік StringIO.

Відбір

```
df["score"].between(0, 100)
```

Маска має значення True, True, False. Обидві межі враховуються. df[маска] формує clean.

Середнє

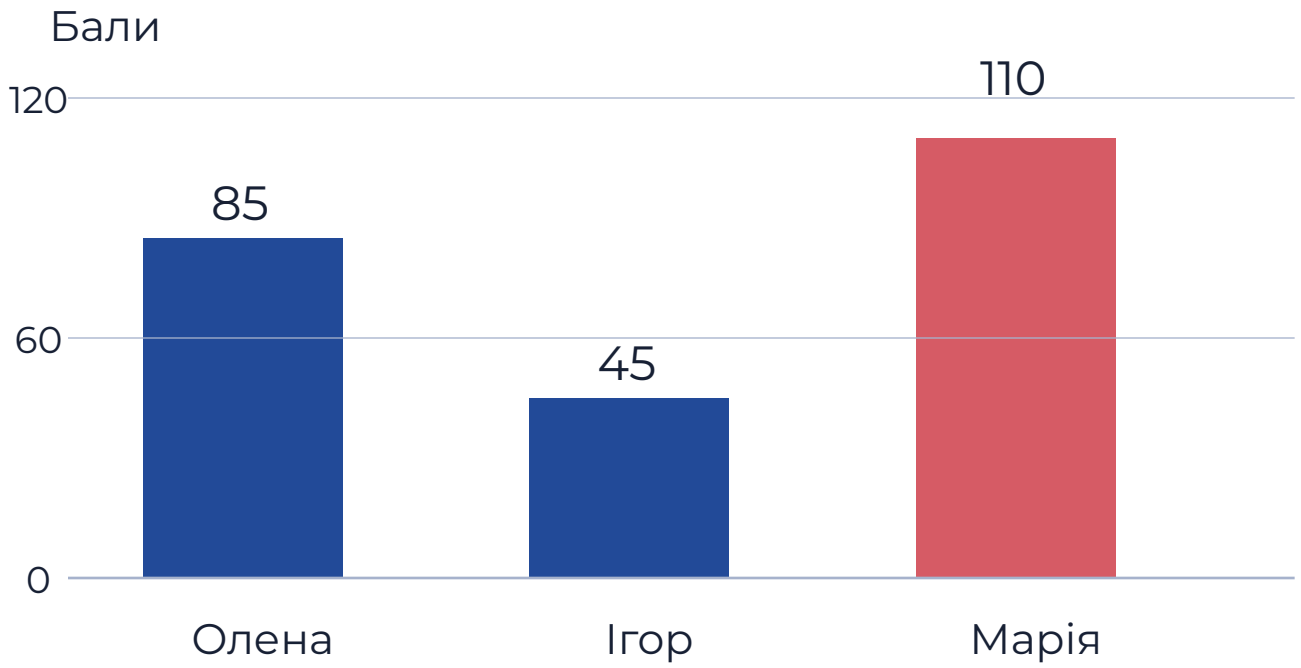
```
clean["score"].mean()
```

Для оцінок 85 та 45 результат дорівнює 65.0. len(clean) повертає кількість рядків 2.

Консоль: 2 і 65.0. pandas — сторонній пакет; для базового звіту його використовувати не обов’язково.

Графік у Google Colab

names містить імена, scores — оцінки 85, 45, 110. Червоний стовпчик позначає значення поза межами 0–100.



Matplotlib

```
import matplotlib.pyplot as plt
names = ["Олена", "Ігор", "Марія"]
scores = [85, 45, 110]
colors = ["C0", "C0", "C3"]
plt.bar(names, scores, color=colors)
plt.ylabel("Бали")
plt.ylim(0, 120)
plt.show()
```

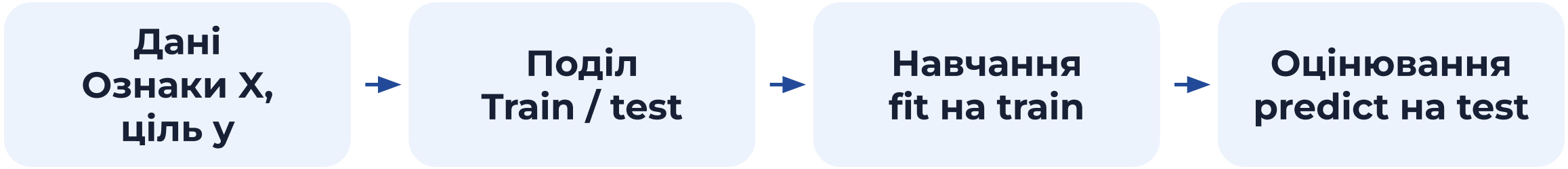
Google Colab

Виконайте клітинку з кодом. Для перевірки виконайте всі клітинки згори вниз.

Для підсумкового звіту оцінку 110 відкидають до обчислень; тут її показано для порівняння.

Машинне навчання: окремий процес

Модель навчається на навчальних даних, а її результат оцінюють на відокремлених тестових даних.



Стрілки показують етапи процесу. fit змінює параметри моделі на train; predict формує прогноз для test. Перетворення налаштовують лише на train, щоб уникнути витоку даних.

Навчання, inference і зовнішній API

Локальна модель

PyTorch / TensorFlow

Навчання змінює параметри
Inference використовує готові
Потрібні модель і ресурси

Зовнішній сервіс

HTTP API моделі

Python надсилає запит
Обчислення виконує сервіс
Є мережа й правила доступу

NLP, комп’ютерний зір і генеративні моделі спираються на дані та оцінювання.

Потоки, процеси й asyncio

Запис	Призначення	Важливий нюанс
`threading`	Конкурентні потоки	Часто для очікування I/O
`multiprocessing`	Окремі процеси	Можлива паралельність CPU
`asyncio`	Цикл подій	Кооперативне очікування
`async def`	Корутина	Виклик ще не виконує її всю
`await`	Очікування awaitable	Може передати керування циклу
`GIL`	Властивість збірки CPython	Є також free-threaded збірки

async/await саме по собі не гарантує паралельного виконання обчислень.

Асинхронність: async def та await

Корутина може призупинити виконання, поки очікує асинхронну операцію. Це корисно для очікування введення й виведення.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
async def дія():
    результат = await операція()
    return результат
asyncio.run(дія())
```

Корутина

async def визначає корутинну функцію. Виклик дія() створює корутину.

Очікування

await працює всередині async def з об'єктом, який підтримує очікування.

Запуск

У клітинці Google Colab корутину запускають через await дія(). У звичайному скрипті — через asyncio.run(дія()).

await не перетворює звичайну блокувальну операцію на неблокувальну.

asyncio: запуск у Google Colab

Корутина повертає результат після асинхронного очікування.

PYTHON · ПРИКЛАД

```python
import asyncio

async def report():
    await asyncio.sleep(0)
    return "Звіт готовий"

print(await report())
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Звіт готовий
```

Пояснення

У Google Colab await report() очікує результат корутини. asyncio.sleep(0) передає керування циклу подій. У звичайному скрипті останній рядок — print(asyncio.run(report())).

Карта подальшого навчання

Запис	Призначення	Важливий нюанс
`Автоматизація`	Файли, винятки	CSV-звіт за розкладом
`Веб і API`	HTTP, функції, типи	Сервіс обробки записів
`Аналіз даних`	Таблиці, статистика	Дослідити відкритий набір
`ШІ`	Математика, дані, метрики	Оцінити просту модель
`GUI`	Події, стан, класи	Форма введення даних
`Якість коду`	Тести, типи, модулі	Перевірений пакет правил

Оберіть один маленький проєкт і поглиблюйте потрібні для нього засоби.

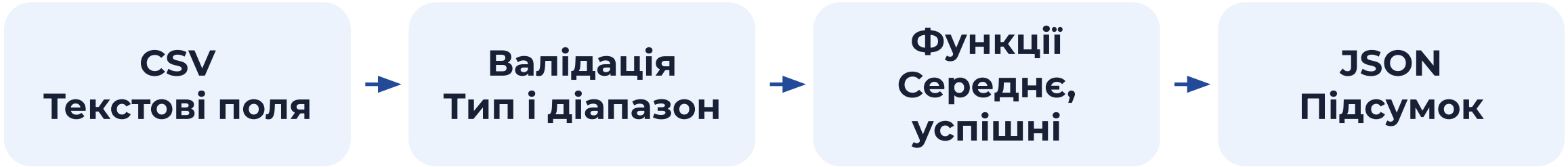
Завершена програма та підсумок курсу

За матеріалами трьох лекцій створимо програму, що читає CSV, перевіряє оцінки, обчислює підсумок і записує JSON.

Три блоки коду «Крок 1–3» разом утворюють report.py. Виконуйте їх у порядку подання.

Архітектура звіту

Підготуйте scores.csv і збережіть три блоки коду в одному report.py. Запускайте з каталогу цих файлів.



scores.csv у UTF-8: перший рядок name,score; наступні — Олена,85; Ігор,45; Марія,110 (кожен на новому рядку).
Запуск у терміналі: python report.py. Стрілки показують рух даних. Результат — report.json у тому самому каталозі.

Крок 1: читання та валідація

PYTHON · ПРИКЛАД

```python
import csv
from pathlib import Path
clean = []
with Path("scores.csv").open(
    encoding="utf-8", newline=""
) as file:
    for row in csv.DictReader(file):
        try:
            score = int(row["score"])
            if not 0 <= score <= 100:
                raise ValueError("поза діапазоном")
        except ValueError:
            print("Пропущено:", row["name"])
        else:
            clean.append({
                "name": row["name"], "score": score
            })
print(len(clean))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Пропущено: Марія
2
```

Пояснення

Крок 1 читає справжній scores.csv. clean зберігає два допустимі записи для кроку 2. Потрібні колонки name і score; файл має існувати.

Крок 1: читання та валідація: пояснення

Перший крок звіту читає CSV та залишає лише записи з допустимими числовими оцінками.

DictReader читає словники з відкритого scores.csv. int перетворює оцінку, if перевіряє діапазон, except пояснює пропуск. with закриває файл.

Число

```
score = int(row["score"])
```

З рядкового поля отримуємо ціле число. Текст, який не є цілим числом, спричинить ValueError.

Межі

```
if not 0 <= score <= 100:
```

Для Марії оцінка 110 не проходить перевірку. Наступний raise передає керування обробнику помилки.

Допустимі записи

```
clean.append({
    "name": row["name"],
    "score": score})
```

Цей фрагмент стоїть у else після try. Він виконується лише за успішної перевірки запису.

Консоль: «Пропущено: Марія» і 2. У clean залишаються Олена з 85 та Ігор із 45.

Крок 2: обчислювальне ядро

PYTHON · ПРИКЛАД

```python
def make_report(records):
    if not records:
        raise ValueError(
            "Немає допустимих записів"
        )
    scores = [r["score"] for r in records]
    return {
        "count": len(scores),
        "average": sum(scores) / len(scores),
        "passed": sum(s >= 60 for s in scores),
    }
report = make_report(clean)
print(
    report["count"],
    report["average"], report["passed"]
)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
2 65.0 1
```

Пояснення

Продовження report.py: clean отримано в кроці 1. make_report повертає словник report для запису в кроці 3.

Крок 2: обчислювальне ядро: пояснення

make_report обчислює підсумок із уже перевірених записів і повертає словник результатів.

clean надходить із кроку 1 без повторного створення. Для оцінок 85 та 45 функція повертає count=2, average=65.0, passed=1.

Вхідні оцінки

```
scores = [r["score"]
          for r in records]
```

З кожного словника беремо поле score. Перед цим порожній records перевіряється та спричиняє ValueError.

Середнє

```
sum(scores) / len(scores)
```

Сума дорівнює 130, кількість — 2. Частка дає 65.0. count окремо дорівнює len(scores).

Успішність

```
sum(s >= 60 for s in scores)
```

True враховується як 1, False — як 0. Серед 85 та 45 одна оцінка не менша за 60.

Консоль: 2 65.0 1. Читання, обчислення та збереження звіту залишаються окремими операціями.

Крок 3: JSON-звіт

PYTHON · ПРИКЛАД

```python
import json
path = Path("report.json")
with path.open("w", encoding="utf-8") as file:
    json.dump(
        report, file,
        ensure_ascii=False, indent=2
    )
restored = json.loads(
    path.read_text(encoding="utf-8")
)
print(path.name)
print(restored == report)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
report.json
True
```

Пояснення

Продовження report.py: report отримано в кроці 2, Path імпортовано в кроці 1. report.json залишається після запуску. Режим w перезаписує наявний файл.

Крок 3: JSON-звіт: пояснення

Завершальний крок записує словник звіту в JSON та перевіряє читання назад.

report надходить із кроку 2. Path("report.json") задає файл у робочому каталозі. Записаний JSON залишається після завершення програми.

Запис

```
json.dump(report, file,
    ensure_ascii=False,
    indent=2)
```

У відкритий для запису потік передаємо словник. indent=2 задає відступи, ensure_ascii=False зберігає читабельні символи.

Читання

```
restored = json.loads(
    path.read_text(
        encoding="utf-8"))
```

read_text повертає рядок із файла. loads розбирає цей JSON-текст та відновлює словник.

Перевірка

```
print(restored == report)
```

Порівнюємо значення відновленого та початкового словників. True підтверджує їхню рівність.

Повний вивід report.py: «Пропущено: Марія», 2, «2 65.0 1», report.json, True.

Типові помилки модулів і файлів

Помилка

Порушений порядок роботи

```
# Запуск під час імпорту
main()
# Екземпляр замість JSON-даних
json.dumps(student)
# Читання після закриття
with open(path) as file:
    pass
text = file.read()
```

Імпорт запускає main. JSON не приймає цей екземпляр. Після with файл уже закрито.

Виправлення

Явний запуск і підготовка даних

```
if __name__ == "__main__":
    main()
data = {"name": student.name}
json.dumps(data, ensure_ascii=False)
with open(path, encoding="utf-8") as file:
    text = file.read()
```

Захистіть запуск, виберіть поля об’єкта для JSON і читайте файл всередині with.

Фрагменти передбачають import json, визначені main, student і path; кожна пара показує окрему помилку.

Контрольні питання

Перевірте, чи можете пояснити організацію коду, роботу з файлами та основи об’єктної моделі.

Модулі

Чим import модуль відрізняється від from модуль import назва?

Для чого перевіряють __name__ == "__main__"?

Файли й формати

Що забезпечує with під час роботи з файлом?

Чим JSON-рядок відрізняється від відкритого файла?

Як CSV-поля перетворюють на числа та перевіряють?

Об’єкти

Чим клас відрізняється від екземпляра?

Які ролі виконують self та __init__?

Коли доречні наслідування, а коли композиція?

Самостійна робота

Створіть невелику програму звіту, що читає оцінки з CSV, перевіряє їх та зберігає підсумок у JSON.

Вхідні дані

Створіть CSV із колонками name і score та оцінками 85, 45, 110.

Перетворіть оцінки на числа. Значення поза 0–100 пропустіть із поясненням.

Підсумок і файл

Обчисліть count, average, passed. Очікувані значення — 2, 65.0, 1.

Збережіть результат у report.json у UTF-8 та перевірте читання назад.

Перевірки

Перевірте межі діапазону, порожній набір і роботу програми з файлом.

За бажанням додайте клас або інший формат після перевірки базової версії.

Основна програма використовує стандартну бібліотеку Python. Спочатку перевірте базову версію.