ЛЕКЦІЯ 2 · PYTHON

Функції, дані та надійність

Від повторюваного коду до перевіреної програми

Дисципліна «Мова програмування Python»
Викладач: Голенко Максим Юрійович

План лекції 2

01 **Функції**

Параметри, результат, області видимості

02 **Обробка даних**

Сортування, comprehensions, генератори

03 **Надійність**

Винятки, ресурси та діагностика

04 **Практика**

Тести й спільний приклад

Від алгоритму до функції

Організуємо знайомі обчислення у повторно використовувані дії.

Почнемо з даних і циклів першої лекції.

Що вже вміємо

Дані, умови та цикли
утворюють основу алгоритму.

Функція об’єднує інструкції в окрему дію. Вона допомагає повторно використовувати код і перевіряти обчислення незалежно.

Дані

Список оцінок або словників.

Перевірка

Допустимий діапазон 0–100.

Повторення

Цикл обробляє всі записи.

Тепер дамо окремим діям імена.

Навіщо потрібні функції

Функція дає ім’я окремій дії, приймає дані та може повертати результат. Одне визначення можна викликати багато разів.

Повторювана дія → **Окрема функція** → **Кілька викликів** → **Незалежний тест**

Одне правило реалізуємо в одному місці.

Функція: визначення та синтаксис

Функція — об’єкт, який можна викликати для виконання окремої дії. Параметри передають їй дані, return повертає результат.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
def назва_функції(параметри):
    інструкції
    return результат
```

Заголовок

def, ім’я, параметри в дужках і двокрапка. Визначення не запускає тіло.

Тіло

Інструкції мають однаковий відступ. За домовленістю використовують 4 пробіли.

Результат

return завершує виклик. Без явного повернення результатом є None.

Виклик має форму назва_функції(аргументи). Параметри та аргументи — різні поняття.

Визначення та виклик

Визначення функції та передавання списку під час виклику.

PYTHON · ПРИКЛАД

```python
def average(scores):
    return sum(scores) / len(scores)

values = [80, 90]
result = average(values)
print(result)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
85.0
```

Пояснення

average(values) передає список [80, 90] параметру scores. sum(scores) / len(scores) обчислює середнє 85.0; return повертає його, а print виводить результат.

return та print

PYTHON · ПРИКЛАД

```python
def calculate(a, b):
    return a + b

def show(a, b):
    print(a + b)

print(calculate(2, 3) * 2)
result = show(2, 3)
print(result)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
10
5
None
```

Пояснення

calculate повертає 5, тому результат можна помножити на 2. show лише друкує 5 і не має return, тож повертає None. Саме None отримує змінна result.

Повернення: коротка довідка

Запис	Призначення	Важливий нюанс
`return value`	Повернути значення	Завершує поточний виклик
`return`	Повернути None	Також завершує функцію
`Без return`	Неявний None	Якщо виконання дійшло до кінця
`print(value)`	Вивести значення	Сам print повертає None
`return a, b`	Повернути кортеж	Можна розпакувати результат
`Після return`	Недосяжна ділянка	Не виконується в цьому шляху

Виведення та обчислення варто розділяти.

Кілька результатів

Одна функція повертає кортеж із двох значень.

PYTHON · ПРИКЛАД

```python
def limits(scores):
    return min(scores), max(scores)

low, high = limits([80, 45, 90])
print(low, high)
print(limits([60, 70]))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
45 90
(60, 70)
```

Пояснення

return min(scores), max(scores) формує один кортеж. Присвоєння low, high розпаковує його в окремі змінні; прямий print показує кортеж у дужках.

Документування функції

Документаційний рядок описує призначення функції.

PYTHON · ПРИКЛАД

```python
def average(scores):
    """Повертає середнє непорожнього списку."""
    return sum(scores) / len(scores)

print(average.__doc__)
print(average([60, 80]))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Повертає середнє непорожнього списку.
70.0
```

Пояснення

Перший рядковий літерал у тілі є docstring. __doc__ містить його текст, а help(average) показує довідку. Приклад обчислює середнє непорожнього списку; перевірку порожності тут не реалізовано.

Документування функції: пояснення

Документаційний рядок описує призначення функції.

Документація відрізняється від коментаря: docstring належить функції та доступний програмно. Коментар із # не потрапляє до __doc__.

Опис

```
"""Повертає середнє
непорожнього списку."""
```

Перший рядковий літерал у тілі average є її docstring. Він описує призначення функції.

Довідка

```
print(average.__doc__)
```

__doc__ зберігає документаційний рядок. help(average) показує довідку про функцію.

Передумова

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

Ділення коректне лише для непорожнього scores. У цьому прикладі перевірку ще не додано.

Документація описує передумову, але сама не перевіряє вхідні дані.

Сигнатура функції

Запис	Призначення	Важливий нюанс
`def f(x, y=0)`	Параметри	y має типове значення
`f(5, 2)`	Позиційні аргументи	Порядок має значення
`f(x=5, y=2)`	Іменовані аргументи	Імена мають відповідати
`*args`	Кортеж аргументів	Довільна кількість позиційних
`**kwargs`	Словник аргументів	Довільні іменовані
`f(*items, **options)`	Розпакування виклику	Не дублювати одне значення

Параметри — у визначенні; аргументи — у виклику.

Аргументи та області видимості

Уточнимо передачу даних і доступ до імен.

Після цього застосуємо функції до наборів даних.

Параметри та аргументи: синтаксис

Параметр — ім’я у визначенні функції. Аргумент — значення, передане під час її виклику.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
def функція(параметр, поріг=60):
    return вираз

функція(значення, поріг=70)
```

Параметр

параметр отримує передане значення. Назва існує в тілі функції.

Типове значення

поріг=60 застосовується, якщо аргумент поріг не передали.

Аргумент

значення передають за позицією; поріг=70 — за іменем.

Позначення у схемі замінюють реальними іменами та виразами.

Параметри й аргументи

Аргументи можна передати за позицією або за іменем.

PYTHON · ПРИКЛАД

```python
def passed(score, threshold):
    return score >= threshold

print(passed(75, 60))
print(passed(threshold=60, score=45))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
True
False
```

Пояснення

У passed(75, 60) аргументи відповідають порядку параметрів score і threshold. В іменованому виклику порядок запису не важливий: назви визначають, яке значення отримує параметр.

Значення за замовчуванням

Типове значення використовується лише за відсутності аргументу.

PYTHON · ПРИКЛАД

```python
def passed(score, threshold=60):
    return score >= threshold

print(passed(65))
print(passed(65, threshold=70))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
True
False
```

Пояснення

У passed(65) параметр threshold отримує типове значення 60. У другому виклику threshold=70 замінює його, тому перевірка 65 >= 70 повертає False.

Змінний типовий аргумент

PYTHON · ПРИКЛАД

```python
def bad(value, items=[]):
    items.append(value)
    return items

print(bad(1))
print(bad(2))

def good(value, items=None):
    if items is None:
        items = []
    items.append(value)
    return items

print(good(1), good(2))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

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

Пояснення

Типовий список спільний для викликів. Деталі — на наступному слайді.

Змінний типовий аргумент: пояснення

Типовий змінний об’єкт може накопичувати дані між викликами.

Типові аргументи обчислюються під час визначення функції. Для змінних об’єктів це може спричинити небажаний спільний стан між викликами.

Спільний список

```
def bad(value, items=[]):
```

Типовий список створюється один раз. Наступний виклик bad використовує той самий об’єкт.

Новий список

```
if items is None:
    items = []
```

Якщо список не передали, good створює новий список для поточного виклику.

Переданий список

```
items.append(value)
```

Якщо передали список, append змінює саме його. Повертається той самий об’єкт.

None розділяє випадки «список не передали» та «передали власний список».

Збирання та розпакування аргументів

У визначенні

Зібрати передані аргументи

```
def f(*args, **kwargs):
```

*args → кортеж позиційних.
**kwargs → словник іменованих.
args і kwargs — звичні назви.

У виклику

Розгорнути готові дані

```
f(*values, **fields)
```

*values → окремі аргументи.
**fields → іменовані аргументи.
Ключі fields мають бути рядками.

Зірочки збирають у визначенні, але розпаковують у виклику.

*args та **kwargs

Зірочки збирають довільну кількість аргументів.

PYTHON · ПРИКЛАД

```python
def total(*scores):
    return sum(scores)

def describe(**fields):
    return sorted(fields)

print(total(60, 80, 90))
print(describe(name="Олена", score=85))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
230
['name', 'score']
```

Пояснення

*scores збирає позиційні аргументи у кортеж, який підсумовує sum. **fields збирає іменовані аргументи у словник; sorted(fields) повертає відсортовані ключі, а не значення.

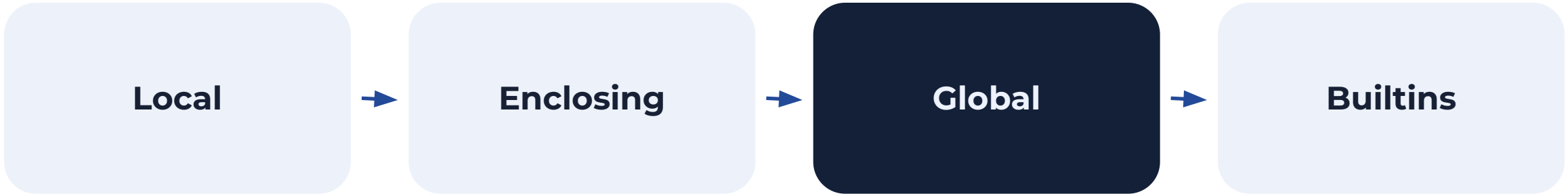
Розпакування та обмеження

Запис	Призначення	Важливий нюанс
`f(*values)`	Розпакувати позиційні	Елементи стають аргументами
`f(**fields)`	Розпакувати іменовані	Ключі мають бути назвами
`def f(x, /)`	Лише позиційний x	f(x=...) не дозволено
`def f(*, x)`	Лише іменований x	Потрібно f(x=...)
`def f(x, *, y=0)`	Явний режим для y	y передаємо за назвою
`Дубль аргументу`	Помилка TypeError	Не передавати x двічі

Оглядові правила: читайте їх у підписах бібліотечних функцій.

Області видимості: LEGB

Область видимості визначає, де доступне ім’я. LEGB описує порядок пошуку: локальна, зовнішня функція, модуль, вбудовані імена.



Пошук імені: локальне → зовнішнє → модуль → вбудоване.

Зміна аргументу та переприсвоєння

Зміна спільного об’єкта відрізняється від переприсвоєння імені.

PYTHON · ПРИКЛАД

```python
def modify(items):
    items.append(90)
    items = [0]
    print(items)

scores = [80]
modify(scores)
print(scores)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
[0]
[80, 90]
```

Пояснення

items.append(90) змінює список, на який посилається scores. Присвоєння items = [0] прив’язує лише локальне ім’я до нового списку. Початковий scores залишається [80, 90].

Анотації типів: загальний запис

Анотації описують очікувані типи параметрів і результату. Python сам не забезпечує їх перевірку.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
def функція(параметр: тип) -> тип_результату:
    return вираз
```

Параметр

Двокрапка відділяє ім’я параметра від анотації його типу.

Тип

Наприклад, int, float, str. Анотацію аналізують редактори й інструменти.

Результат

Після -> записують тип значення, яке має повернути функція.

Перевірку даних під час виконання потрібно реалізовувати окремо.

Анотації типів

Анотації пояснюють очікувані типи без автоматичної перевірки.

PYTHON · ПРИКЛАД

```python
def passed(
    score: int, threshold: int = 60
) -> bool:
    return score >= threshold

print(passed(75))
print(passed(75.5))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
True
True
```

Пояснення

score: int та -> bool описують параметр і результат. Виклик із 75.5 також виконується: анотація сама не забороняє float. Перевірка порога повертає True в обох випадках.

Обробка та сортування даних

Застосуємо функції до наборів значень.

Спочатку розглянемо явні операції, потім компактні форми.

sum(), any(), all(): призначення

Агрегування об’єднує значення набору в один результат. Джерелом може бути список або інший ітерований об’єкт.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
sum(значення)
any(результати_перевірок)
all(результати_перевірок)
```

sum

Підсумовує числові значення. Для порожнього набору типовий результат — 0.

any

True, якщо хоча б одна перевірка істинна. Для порожнього набору — False.

all

True, якщо всі перевірки істинні. Для порожнього набору — True.

У цьому прикладі передаємо список значень True/False. Генераторний вираз, який обчислює перевірки під час перебору, розглянемо далі.

Агрегування: sum, any, all

Підсумовування та логічні перевірки для набору оцінок.

PYTHON · ПРИКЛАД

```python
scores = [80, 45, 90]
passed = [True, False, True]
valid = [True, True, True]
print(sum(scores), len(scores))
print(sum(scores) / len(scores))
print(any(passed))
print(all(valid))
print(any([]), all([]))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
215 3
71.66666666666667
True
True
False True
```

Пояснення

passed містить результати score >= 60 для оцінок 80, 45, 90; valid — перевірок 0 <= score <= 100. any шукає хоча б одне True, all — перевіряє всі. sum дає 215, len — 3. Для порожнього набору any є False, all — True.

Сортування: синтаксис і параметри

sorted() створює новий список. Метод list.sort() змінює наявний список і повертає None.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
sorted(джерело, key=функція_ключа, reverse=False)
список.sort(key=функція_ключа, reverse=False)
```

Джерело

sorted приймає ітерований об’єкт; sort викликають лише для списку.

Ключ (key)

Функція отримує елемент і повертає значення для порівняння. Типово key=None.

Напрям (reverse)

False — за зростанням, True — за спаданням. Сортування є стабільним.

Передавайте функцію без виклику: key=функція_ключа. Дужки після її імені запускають функцію одразу.

sorted та list.sort

sorted зберігає оригінал, а sort змінює список.

PYTHON · ПРИКЛАД

```python
scores = [80, 45, 90]
ordered = sorted(scores, reverse=True)
print(scores)
print(ordered)
result = scores.sort()
print(scores, result)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
[80, 45, 90]
[90, 80, 45]
[45, 80, 90] None
```

Пояснення

sorted(scores, reverse=True) створює новий список за спаданням. scores.sort() змінює scores за зростанням і повертає None. Тому result не містить відсортованого списку.

Функція як значення

PYTHON · ПРИКЛАД

```python
def double(value):
    return value * 2

action = double
print(action(5))

def apply(function, value):
    return function(value)

print(apply(double, 7))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
10
14
```

Пояснення

action = double зберігає посилання на функцію. action(5) викликає її й повертає 10. apply(double, 7) отримує функцію параметром function та викликає її для значення 7.

def і lambda: два способи визначення

def

Іменована функція з тілом

```
def double(x):
    return x * 2
```

Інструкції та явний return.
Доречно для складнішої логіки.

lambda

Функція з одним виразом

```
lambda x: x * 2
```

x — параметр; після : — вираз.
Його значення є результатом.
Окремого return немає.

Обидва записи створюють функцію. lambda містить лише один вираз; def допускає кілька інструкцій.

Ключ сортування

PYTHON · ПРИКЛАД

```python
def get_score(record):
    return record["score"]
records = [
    {"name": "Олена", "score": 85},
    {"name": "Ігор", "score": 45},
]
ordered = sorted(records, key=get_score)
for record in ordered:
    print(record["name"])
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Ігор
Олена
```

Пояснення

get_score отримує один словник і повертає його поле score. У key=get_score передають саму функцію без дужок. sorted порівнює оцінки 45 і 85 та розміщує Ігоря перед Оленою.

Lambda для короткого key

PYTHON · ПРИКЛАД

```python
records = [
    {"name": "Олена", "score": 85},
    {"name": "Ігор", "score": 45},
]
ordered = sorted(
    records, key=lambda r: r["score"],
    reverse=True,
)
for record in ordered:
    print(record["name"])
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Олена
Ігор
```

Пояснення

lambda r: r["score"] повертає оцінку поточного запису. reverse=True змінює напрям на спадання, тому першою є Олена. Для кількох інструкцій або складної логіки краще використовувати def.

map() і filter(): загальний запис

map застосовує перетворення до елементів, а filter залишає елементи, для яких перевірка істинна.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
map(перетворення, джерело)
filter(перевірка, джерело)
```

Перетворення

Функція повертає нове значення для кожного елемента.

Перевірка

Функція повертає значення, істинність якого визначає відбір.

Джерело

Набір початкових елементів. Результат map і filter — ітератор.

Ітератор вичерпується. Після повного перебору повторне list(...) дає порожній список.

map та filter

map і filter обчислюють значення під час перебору.

PYTHON · ПРИКЛАД

```python
texts = ["80", "45", "90"]
numbers = map(int, texts)
print(list(numbers))
print(list(numbers))
positive = filter(
    lambda x: x > 0, [-2, 0, 3]
)
print(list(positive))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
[80, 45, 90]
[]
[3]
```

Пояснення

Перший list(numbers) перетворює текстові числа на int та вичерпує map. Повторний list(numbers) дає []. filter залишає лише 3, бо тільки для цього елемента x > 0 є істинним.

Обробка наборів: довідка

Запис	Призначення	Важливий нюанс
`sum(items, start=0)`	Сума значень	Порожній набір → start
`min / max`	Крайні значення	default для порожнього iterable
`any / all`	Логічний підсумок	Коротке обчислення
`sorted(items, key=…)`	Новий список	reverse=False за замовчуванням
`map(f, items)`	Перетворення	Результат — ітератор
`filter(f, items)`	Відбір	Зберігає початкові елементи

Для середнього перевіряємо непорожність перед діленням.

Comprehensions та вибір за зразком

Стисло запишемо відбір і перетворення.

Компактність не повинна приховувати логіку.

Comprehension: перетворення та відбір

Списковий вираз створює список, поєднуючи перебір, перетворення й необов'язковий відбір.

Створення

Елемент із кожного значення

```
[вираз for елемент in джерело]
```

джерело — набір початкових значень.
елемент — поточне значення.
вираз — перетворення цього значення.

Відбір

Елемент лише за умовою

```
[вираз for елемент in джерело
      if умова]
```

Умову перевіряють перед обчисленням виразу.
False → значення пропускається.
True → результат додається.

[A if умова else B for елемент in джерело] обирає A або B для кожного елемента джерела.

List comprehension

Списковий вираз замінює цикл накопичення результатів.

PYTHON · ПРИКЛАД

```python
numbers = [1, 2, 3]
squares = []
for x in numbers:
    squares.append(x * x)

compact = [x * x for x in numbers]
print(squares)
print(compact)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
[1, 4, 9]
[1, 4, 9]
```

Пояснення

Обидва записи обчислюють x * x для кожного x з numbers та створюють [1, 4, 9]. У comprehension вираз нового елемента стоїть перед for; порядок результатів відповідає перебору.

Відбір та умовний результат

Умова відбору і умовний вираз мають різні результати.

PYTHON · ПРИКЛАД

```python
scores = [80, 45, 90]
passed = [s for s in scores if s >= 60]
labels = [
    "Так" if s >= 60 else "Ні"
    for s in scores
]
print(passed)
print(labels)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
[80, 90]
['Так', 'Ні', 'Так']
```

Пояснення

if після for пропускає оцінку 45, тому passed містить два елементи. Умовний вираз перед for обирає «Так» або «Ні» для кожної оцінки, тому labels містить три елементи.

Set і dict comprehensions

PYTHON · ПРИКЛАД

```python
words = [" Python ", "python", "DATA"]
unique = {
    w.strip().lower() for w in words
}
records = [("Олена", 85), ("Ігор", 45)]
by_name = {
    name: score for name, score in records
}
print(sorted(unique))
print(by_name["Олена"])
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
['data', 'python']
85
```

Пояснення

unique об’єднує однакові нормалізовані слова. by_name зіставляє ім’я з оцінкою через ключ: значення. sorted(unique) задає передбачуваний порядок виведення.

Вкладений comprehension

PYTHON · ПРИКЛАД

```python
matrix = [[1, 2], [3, 4]]
flat = [
    value for row in matrix for value in row
]
print(flat)

expanded = []
for row in matrix:
    for value in row:
        expanded.append(value)
print(expanded)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

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

Пояснення

Зовнішній перебір записаний першим. Деталі — на наступному слайді.

Вкладений comprehension: пояснення

Порядок for у comprehension відповідає вкладеним циклам.

У comprehension порядок for читають зліва направо, як порядок вкладених циклів. Кожен внутрішній перебір виконується для поточного рядка.

Порядок перебору

```
for row in matrix:
    for value in row:
```

Спочатку вибирається рядок matrix, потім — кожне значення цього рядка.

Однаковий результат

```
flat == expanded
# True
```

flat та expanded містять [1, 2, 3, 4]. Ця перевірка порівнює вміст обох списків.

Збирання списку

```
expanded.append(value)
```

У розгорнутому записі append додає поточне значення. Цикли задають той самий порядок.

Для складних умов розгорнуті цикли часто легше читати й налагоджувати.

match / case: загальний синтаксис

match зіставляє значення зі зразками та виконує першу відповідну гілку. Потрібен Python 3.10 або новіший.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
match значення:
    case зразок_1:
        дія_1
    case зразок_2:
        дія_2
    case _:
        інша_дія
```

match

значення обчислюється перед зіставленням зі зразками.

case

Гілки перевіряються послідовно. Після заголовка потрібні двокрапка й відступ.

Зразок _

Охоплює решту значень. Текстові літерали записують у лапках.

match / case

PYTHON · ПРИКЛАД

```python
command = "report"
match command:
    case "report":
        message = "Створити звіт"
    case "exit":
        message = "Завершити"
    case _:
        message = "Невідома команда"
print(message)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Створити звіт
```

Пояснення

Значення command дорівнює "report", тому спрацьовує перший case. Зразок _ охоплює решту значень. Літерали рядків записують у лапках; звичайне ім’я в case може захоплювати значення.

Рекурсія, ітератори та генератори

Розглянемо різні способи отримувати значення.

Важливі завершення та стан перебору.

Рекурсія: завершення та повернення

Базовий випадок

Відповідь без нового виклику

```
if n == 0:
    return 1
```

Зупиняє рекурсивне заглиблення.
Має бути досяжним.

Рекурсивний крок

Перехід до меншої задачі

```
return n * factorial(n - 1)
```

Новий виклик має власні дані.
Після нього попередній триває.
Результати повертаються назад.

Рекурсія — розв’язання задачі через виклики тієї самої функції.

Рекурсія: базовий випадок

PYTHON · ПРИКЛАД

```python
def factorial(n):
    if n < 0:
        raise ValueError("n має бути невід'ємним")
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(4))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
24
```

Пояснення

n == 0 зупиняє виклики. Деталі — на наступному слайді.

Рекурсія: базовий випадок: пояснення

Рекурсивний виклик зменшує задачу до базового випадку.

Кожен рекурсивний виклик має власне n. Повернення результатів починається з базового випадку й завершує попередні виклики у зворотному порядку.

Рекурсивний крок

```
return n * factorial(n - 1)
```

factorial(4) обчислює 4 · 3 · 2 · 1 і повертає 24. Кожен виклик зменшує n на один.

Недопустиме n

```
if n < 0:
```

Для від'ємного n функція породжує ValueError замість продовження рекурсії.

Базовий випадок

```
if n == 0:
    return 1
```

Для n = 0 повертається 1. Ця гілка припиняє рекурсивні виклики.

Приклад розрахований на ціле n. Перевірку типу аргументу тут не реалізовано.

Ітерований об’єкт, ітератор, генератор

Ітерований об’єкт

Можна отримати ітератор

```
iter(джерело) → ітератор
```

Список, рядок, range — джерела.
Для списку новий iter(джерело)
починає новий перебір.

Ітератор

Має стан поточного перебору

```
next(ітератор) → елемент
```

Після кінця → StopIteration.
Генератор — різновид ітератора.
Нові значення дає за запитом.

return завершує виклик; yield видає значення й призупиняє генератор.

Ітератор: iter та next

Ітератор видає елементи по одному та зберігає стан.

PYTHON · ПРИКЛАД

```python
iterator = iter([80, 90])
print(next(iterator))
print(next(iterator))
print(next(iterator, "Кінець"))
print(list(iterator))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
80
90
Кінець
[]
```

Пояснення

Перші два next отримують 80 і 90. Третій повертає типове значення «Кінець», бо елементи закінчилися. list(iterator) після цього дає []; без типового значення виник би StopIteration.

Генераторна функція: синтаксис yield

Функція з yield створює генератор. Її тіло починає виконуватися під час запиту наступного значення.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
def генератор(джерело):
    for елемент in джерело:
        yield вираз
```

Джерело

Звідси отримують елементи для обчислень. Виклик повертає об’єкт генератора.

yield

Видає значення виразу та призупиняє виконання, зберігаючи стан.

Вираз

Обчислюється для поточного елемента. yield видає його результат як наступне значення.

next(...) або цикл for відновлює генератор. return завершує його; next(...) після вичерпання породжує StopIteration.

Генераторна функція: yield

yield призупиняє виконання після кожного отриманого квадрата.

PYTHON · ПРИКЛАД

```python
def squares(numbers):
    for x in numbers:
        yield x * x

values = squares([1, 2, 3])
print(next(values))
print(next(values))
print(list(values))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
1
4
[9]
```

Пояснення

Перші два next(values) отримують 1 і 4. list(values) продовжує перебір з поточного місця та отримує лише [9]. Повторний перебір цього генератора не почнеться заново.

Генераторний вираз

Генераторний вираз не створює готовий список результатів.

PYTHON · ПРИКЛАД

```python
numbers = [1, 2, 3]
values = (x * x for x in numbers)
print(sum(values))
print(list(values))
print(sum(x * x for x in numbers))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
14
[]
14
```

Пояснення

sum(values) послідовно отримує квадрати й повертає 14. Після цього values вичерпаний, тому list(values) дає []. Останній sum використовує новий генераторний вираз і знову повертає 14.

Готовий список чи генератор

Список

Значення вже збережені

```
data = [1, 2, 3]
values = [x * x for x in data]
print(values[0])
```

Значення вже обчислені.
Є індекси та повторний перебір.

Генератор

Наступне значення за запитом

```
data = [1, 2, 3]
values = (x * x for x in data)
print(next(values))
```

Обчислення під час перебору.
Індексів немає. Вичерпується.

Для data = [1, 2, 3] обидва виклики виведуть 1. Спосіб зберігання результатів різний.

Декоратори та керування ресурсами

Побачимо обгортання дій і життєвий цикл ресурсу.

Ці ідеї підготують роботу з діагностикою й файлами.

Замикання: збереження зовнішніх даних

Замикання — вкладена функція, що має доступ до імен зовнішньої функції після завершення її виклику.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
def зовнішня(налаштування):
    def внутрішня(значення):
        return вираз
    return внутрішня
```

Налаштування

Параметр зовнішньої функції. Його значення доступне вкладеній функції.

Параметр

значення передають під час виклику внутрішньої функції.

Повернення функції

return внутрішня передає функцію без виклику; дужки запустили б її.

Зберігається доступ до зовнішніх імен, а не обов’язково незмінна копія їхніх значень.

Вкладена функція та замикання

Вкладена функція зберігає доступ до заданого порога.

PYTHON · ПРИКЛАД

```python
def make_check(threshold):
    def check(score):
        return score >= threshold
    return check

passed = make_check(60)
print(passed(75), passed(45))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

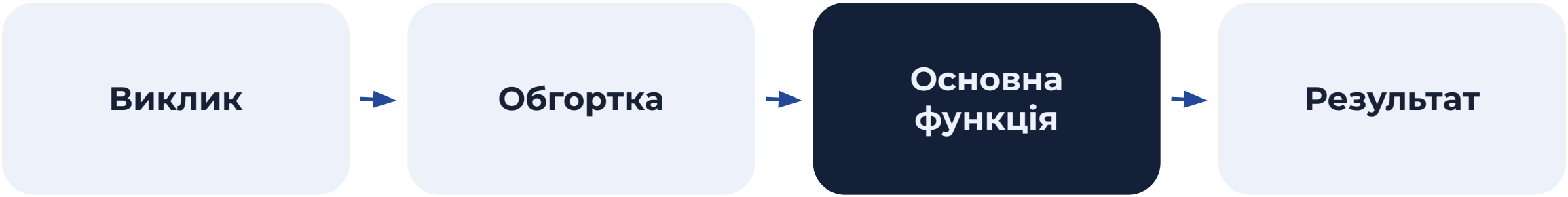
```
True False
```

Пояснення

make_check(60) повертає функцію check без її виклику. У passed зберігається ця функція, що пам’ятає доступ до threshold. Наступні виклики порівнюють 75 і 45 із порогом 60.

Ідея декоратора

Декоратор викликається під час визначення функції та пов’язує її ім’я з отриманим результатом. Обгортка додає дії під час наступних викликів.



Темний блок — основна функція; обгортка додає дії навколо її виклику. @decorator відповідає f = decorator(f).

Декоратор: що відбувається з функцією

Із @

Застосування декоратора

```
@декоратор
def функція(параметр):
    ...
```

Спочатку створюється функція. Потім її ім’я пов’язується з результатом декоратора.

Без @

Еквівалентний запис

```
def функція(параметр):
    ...
функція = декоратор(функція)
```

Декоратор отримує функцію. Повернений об’єкт присвоюється тому самому імені.

У типових прикладах декоратор повертає функцію-обгортку, що додає поведінку під час виклику.

Мінімальний декоратор

PYTHON · ПРИКЛАД

```python
from functools import wraps

def announce(function):
    @wraps(function)
    def wrapper(*args, **kwargs):
        print("Виклик:", function.__name__)
        return function(*args, **kwargs)
    return wrapper

@announce
def double(value):
    return value * 2

print(double(5))
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Виклик: double
10
```

Пояснення

Передає всі аргументи далі. Деталі — на наступному слайді.

Мінімальний декоратор: пояснення

Декоратор додає повідомлення перед викликом основної функції.

announce створює wrapper та повертає його без виклику. Після декорування double посилається на wrapper, а початкова функція доступна через замикання.

Декорування

```
@announce
def double(value):
```

announce отримує функцію double. Ім’я double починає посилатися на повернуту обгортку.

Виклик обгортки

```
return function(
    *args, **kwargs)
```

wrapper передає аргументи початковій функції. Для double(5) результат дорівнює 10.

Метадані

```
@wraps(function)
```

wraps переносить ім’я й docstring початкової функції на wrapper.

Обгортка додає повідомлення перед викликом, а результат повертає без змін.

with: призначення та синтаксис

Контекстний менеджер організовує вхід у блок і вихід із нього. Для файлу це забезпечує його закриття.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
with менеджер as ресурс:
    інструкції
```

Менеджер

Вираз створює або повертає контекстний менеджер.

Ресурс

as прив’язує результат входу до імені. Ця частина необов’язкова.

Тіло

Інструкції з відступом виконуються всередині контексту.

Якщо вхід у контекст успішний, під час виходу викликається його процедура завершення, зокрема за винятку.

with: життєвий цикл ресурсу

with передає керування ресурсом контекстному менеджеру. Завершення контексту не потрібно дублювати в кожній гілці програми.

Вхід у with → **Отримання ресурсу** → **Робота в блоці** → **Завершення й очищення**

Очищення організовує контекстний менеджер, навіть за винятку.

with: приклад керування ресурсом

Контекстний менеджер закриває ресурс після виходу з блоку.

PYTHON · ПРИКЛАД

```python
from io import StringIO

with StringIO("Python") as stream:
    print(stream.read())
    print(stream.closed)
print(stream.closed)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Python
False
True
```

Пояснення

StringIO — потік у пам’яті. У блоці with він відкритий, тому closed є False. Після виходу менеджер закриває потік і closed стає True. Такий самий принцип застосовують до файлів.

Помилки та винятки

Відрізнятимемо збій від очікуваного результату.

Після обробки помилки подбаємо про діагностику.

Три види помилок

Різні причини потребують
різних способів виявлення.

Синтаксичну помилку виявляють до виконання. Виняток виникає під час виконання, а логічна помилка може дати неправильний результат без винятку.

Синтаксис

Пропущена двокрапка: код не запускається.

Виконання

int("8.5") породжує ValueError.

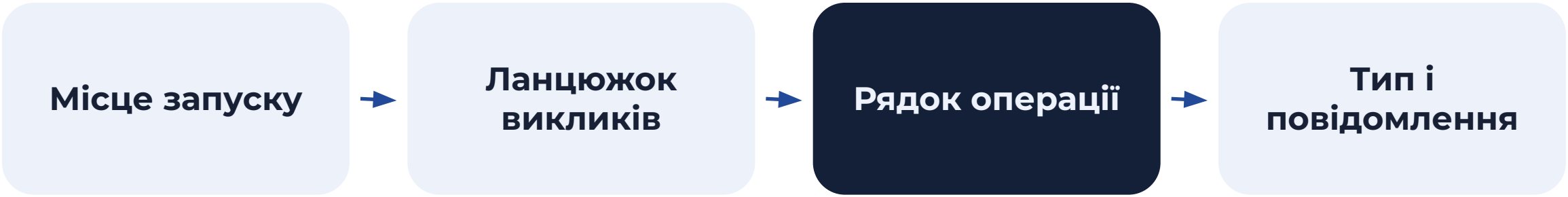
Логіка

Неправильний поріг дає хибний результат.

Відсутність винятку ще не означає правильності.

Як читати traceback

Traceback показує послідовність викликів, що привела до винятку. Останній рядок містить тип винятку та його повідомлення.



Темний блок — рядок, що спричинив виняток.
Приклад: main → read_score → int("8.5") → ValueError.

Виняток: виникнення й обробка

Виникнення

Звичайний шлях переривається

```
raise ValueError("Причина")
```

Виняток — об’єкт опису збою.
Невиконані інструкції try
після збою пропускаються.

Обробка

Вибір відповідного except

```
except ValueError as error:
```

Перший відповідний обробник.
else — якщо try успішний.
finally — під час виходу.

raise породжує виняток; except реагує на нього; logging записує подію.

try / except / else / finally: будова

try містить операції, які можуть породити виняток. except обробляє виняток відповідного типу.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
try:
    операція
except ValueError as error:
    обробка
else:
    успішна_дія
finally:
    завершення
```

try / except

ValueError у try запускає except. as error зберігає виняток в імені error.

else

Виконується після нормального завершення try. Винятки в else не обробляє цей except.

finally

Виконується під час виходу. Після нього необроблений виняток передається далі.

try та конкретний except

Очікувану помилку перетворення обробляємо окремо.

PYTHON · ПРИКЛАД

```python
for text in ["80", "8.5", "90"]:
    try:
        score = int(text)
    except ValueError:
        print("Некоректно:", text)
        continue
    print("Оцінка:", score)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Оцінка: 80
Некоректно: 8.5
Оцінка: 90
```

Пояснення

int("8.5") породжує ValueError. except друкує повідомлення, а continue переходить до наступного тексту. Для "80" та "90" перетворення успішне, тому виконується print після try.

Кілька обробників

Різні типи винятків отримують різні повідомлення.

PYTHON · ПРИКЛАД

```python
for text in ["2", "0", "abc"]:
    try:
        value = int(text)
        print(10 / value)
    except ValueError as error:
        print("Не ціле число")
    except ZeroDivisionError as error:
        print("Ділення на нуль")
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
5.0
Ділення на нуль
Не ціле число
```

Пояснення

Для "0" перетворення на int успішне, але ділення спричиняє ZeroDivisionError. Для "abc" виникає ValueError. as error прив’язує об’єкт винятку до імені, хоча цей приклад його не друкує.

else та finally

PYTHON · ПРИКЛАД

```python
for text in ["80", "bad"]:
    try:
        value = int(text)
    except ValueError:
        print("Помилка")
    else:
        print("Успіх:", value)
    finally:
        print("Завершення спроби")
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Успіх: 80
Завершення спроби
Помилка
Завершення спроби
```

Пояснення

Для "80" виконуються try, else і finally. Для "bad" — try, except і finally. else працює лише після нормального завершення try; finally виконується під час виходу з конструкції.

raise: перевірка аргументу

PYTHON · ПРИКЛАД

```python
def validate(score):
    if not 0 <= score <= 100:
        raise ValueError("Оцінка поза 0–100")
    return score

for score in [85, 110]:
    try:
        print(validate(score))
    except ValueError as error:
        print(error)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
85
Оцінка поза 0–100
```

Пояснення

validate повертає оцінку 85, але для 110 породжує ValueError. Обробник отримує об’єкт винятку через as error та виводить його повідомлення. Перевірка діапазону не є перевіркою типу.

Винятки: довідка

Запис	Призначення	Важливий нюанс
`ValueError`	Тип підходить, значення ні	int("abc"); оцінка 110
`TypeError`	Невідповідна операція	"80" + 5
`IndexError`	Недопустимий індекс	items[10] у короткому списку
`KeyError`	Відсутній ключ	record["missing"]
`ZeroDivisionError`	Ділення на нуль	Середнє порожнього набору
`StopIteration`	Ітератор завершився	next без запасного значення

Перехоплюйте очікуваний тип, а не приховуйте все.

Логування

Збережемо діагностику окремо від результатів програми.

Журнал пояснює хід роботи; тести перевіряють правильність.

Навіщо потрібен журнал

Повідомлення користувачу
та діагностика мають різні ролі.

Результати потрібні користувачу. Діагностика допомагає з’ясувати причини проблем і хід обробки даних.

print

Результат або запрошення для користувача.

```
print("Результат:", result)
```

logging

Події, попередження та причини збоїв.

```
logging.warning("Збій")
```

logging із контекстом

Повідомлення уточнює, який запис обробляли.

```
logging.warning(
    "Запис %s: збій", record_id
)
```

Журнал не замінює повернення результату.

Рівні logging

Запис	Призначення	Важливий нюанс
`DEBUG`	Докладні кроки	Для діагностики розробником
`INFO`	Нормальна подія	Початок або завершення обробки
`WARNING`	Потенційна проблема	Запис пропущений; робота триває
`ERROR`	Дія не виконана	Потрібна увага до збою
`CRITICAL`	Критична проблема	Продовження суттєво порушене
`Поріг рівня`	Фільтр повідомлень	INFO приховує DEBUG

Рівень обираємо за наслідком події.

logging: налаштування й повідомлення

Модуль logging записує діагностичні події. Поріг рівня визначає, які повідомлення потрапляють до журналу.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
logging.basicConfig(level=logging.INFO)
logging.info(повідомлення)
logging.warning(шаблон, значення)
```

Поріг (level)

INFO пропускає INFO та важливіші рівні; DEBUG відсіюється.

Повідомлення

Текст події має пояснювати, що відбулося.

Аргументи

У шаблоні можна використати %s або %r та передати значення окремо.

Типово журнал виводиться у stderr. print виводить результат у stdout. Це різні потоки.

Мінімальне налаштування logging

PYTHON · ПРИКЛАД

```python
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(levelname)s: %(message)s",
    force=True,
)
logging.debug("Докладний крок")
logging.info("Обробку розпочато")
logging.warning("Оцінку 110 пропущено")
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
INFO: Обробку розпочато
WARNING: Оцінку 110 пропущено
```

Пояснення

basicConfig показує рівень і текст події. Поріг INFO приховує DEBUG, але пропускає INFO та WARNING. force=True перестворює налаштування кореневого логера.

Журналювання очікуваної помилки

PYTHON · ПРИКЛАД

```python
import logging

logging.basicConfig(
    level=logging.WARNING,
    format="%(levelname)s: %(message)s",
    force=True,
)
try:
    score = int("bad")
except ValueError:
    logging.error(
        "Некоректна оцінка: %r", "bad"
    )
    # logging.exception("Збій")
    # додає traceback
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
ERROR: Некоректна оцінка: 'bad'
```

Пояснення

%r показує подання вхідного тексту. Деталі — на наступному слайді.

Журналювання очікуваної помилки: пояснення

Журнал зберігає контекст помилки перетворення.

Повідомлення error описує відому проблему даних. exception у блоці except додає traceback, коли потрібні деталі місця та послідовності збою.

Причина

```
score = int("bad")
```

Рядок "bad" не можна перетворити на ціле число. Виконання переходить до except ValueError.

Повідомлення

```
logging.error(
    "Некоректна оцінка: %r",
    "bad")
```

%r підставляє представлення "bad". У журналі з’являється повідомлення рівня ERROR.

Traceback

```
# logging.exception("Збій")
```

Цей рядок закоментований. Якщо виконати його в except, журнал міститиме також traceback.

logging.error записує повідомлення; logging.exception додає також дані про поточний виняток.

Модульне тестування

Перевіримо поведінку функцій на відомих прикладах.

Тести допомагають безпечно об’єднувати окремі дії.

Що таке модульний тест

Модульний тест перевіряє окрему дію на відомих даних. Очікуваний результат визначають незалежно від реалізації функції.

Вхідні дані → **Виклик функції** → **Фактичний результат** → **Порівняння з очікуванням**

Тест перевіряє правило, а не повторює його реалізацію.

Вибір тестових випадків

Запис	Призначення	Важливий нюанс
`Звичайний`	Оцінка 85	Очікуємо успішність
`Перед межею`	Оцінка 59	False
`На межі`	Оцінка 60	True
`Після межі`	Оцінка 61	True
`Порожній набір`	average([])	Узгоджений виняток
`Некоректний`	Оцінка 110	ValueError

Кожен випадок перевіряє окрему вимогу.

assert: синтаксис і обмеження

assert перевіряє припущення програми. Якщо умова хибна, виникає AssertionError.

СХЕМА · ПОЗНАЧЕННЯ ЗАМІНЮЮТЬ ДАНИМИ

```
assert умова
assert умова, повідомлення
```

Умова

Істинна — виконання триває. Хибна — виникає AssertionError.

Повідомлення

Необов'язковий опис допомагає зрозуміти причину невдалої перевірки.

Обмеження

За запуску Python з -O перевірки assert вимикаються.

Для перевірки даних користувача застосовуйте if та raise, а не assert.

assert: проста перевірка

Три перевірки охоплюють межу успішності.

PYTHON · ПРИКЛАД

```python
def passed(score):
    return score >= 60

assert passed(59) is False
assert passed(60) is True
assert passed(61) is True
print("Три перевірки пройдено")
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Три перевірки пройдено
```

Пояснення

Для 59 очікується False, для 60 і 61 — True. Якщо хоча б одна перевірка хибна, assert породжує AssertionError. Повідомлення наприкінці друкується лише після успіху всіх перевірок.

unittest: структура тесту

PYTHON · ПРИКЛАД

```python
import unittest

def passed(score):
    if not 0 <= score <= 100:
        raise ValueError("Некоректна оцінка")
    return score >= 60

class TestPassed(unittest.TestCase):
    def test_boundary(self):
        self.assertFalse(passed(59))
        self.assertTrue(passed(60))
    def test_invalid(self):
        with self.assertRaises(ValueError):
            passed(110)

print(TestPassed.__name__)
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
TestPassed
```

Пояснення

Клас групує методи тестування. Деталі — на наступному слайді.

unittest: структура тесту: пояснення

Тестовий клас описує перевірку порога та очікуваного винятку.

TestPassed успадковує unittest.TestCase; self — об’єкт тесту. Поки використовуємо готовий шаблон. Будову класів і self докладно розглянемо в лекції 3.

Поріг успішності

```
self.assertTrue(passed(60))
```

Перевірка очікує True для оцінки 60. Друге твердження в прикладі перевіряє оцінку 59.

Друк класу

```
print(TestPassed.__name__)
```

Цей виклик друкує TestPassed. Він не запускає методи тестування.

Очікуваний виняток

```
with self.assertRaises(
        ValueError):
    passed(110)
```

Оцінка 110 виходить за допустимі межі. Тест перевіряє, що passed породжує ValueError.

Запуск тестів: python -m unittest -v test_passed. Друк назви класу їх не виконує.

Запуск: успіх, failure, error

Команда

```
python -m unittest -v test_passed
```

КОНСОЛЬ · РЕЗУЛЬТАТ (СКОРОЧЕНО)

```
test_boundary ... ok
test_invalid ... ok
Ran 2 tests
OK
```

Збережіть попередній приклад у test_passed.py. Очікуються два успішні тести. OK означає, що всі перевірки виконано.

Навмисно змініть очікування для 60 та поясніть FAIL.

Невдачі

FAIL та ERROR

FAIL: твердження не збіглося.
ERROR: неочікуваний виняток.
Читаємо ім’я тесту й traceback.

pytest: тести-функції

PYTHON · ПРИКЛАД

```python
import pytest

def passed(score):
    if not 0 <= score <= 100:
        raise ValueError("Некоректна оцінка")
    return score >= 60

def test_boundary():
    assert passed(60) is True

def test_invalid():
    with pytest.raises(ValueError):
        passed(110)
```

КОНСОЛЬ · КОМАНДА

```
python -m pytest -q
```

Пояснення

Встановлення: python -m pip install pytest. Деталі — на наступному слайді.

pytest: тести-функції: пояснення

pytest знаходить функції з іменами test_ та виконує їх.

pytest знаходить функції з префіксом test_. assert перевіряє результат, а pytest.raises — очікуваний виняток.

Поріг успішності

```
assert passed(60) is True
```

Звичайне assert перевіряє результат passed(60). У разі хибного результату тест не проходить.

Очікуваний виняток

```
with pytest.raises(
        ValueError):
    passed(110)
```

pytest.raises перевіряє, що passed(110) породжує саме ValueError.

Запуск

```
python -m pytest -q
```

Збережіть код у test_passed.py. pytest знаходить і виконує обидві функції test_*.

pytest — сторонній пакет. Установлення: python -m pip install pytest.

Тести: коротка довідка

Запис	Призначення	Важливий нюанс
`assertEqual(a, b)`	Рівність значень	unittest
`assertTrue(x)`	Істинність	Не завжди тотожність True
`assertRaises(Error)`	Очікуваний виняток	with навколо виклику
`assert expression`	Проста перевірка	pytest або навчальний приклад
`test_*.py`	Пошук файлів	Звичне ім’я для запуску
`Незалежність`	Окремі вхідні дані	Без спільного прихованого стану

Результат, межа й виняток — різні вимоги.

Завершення другої лекції

Об’єднаємо функції, перевірки та тести.

Далі організуємо їх у модулі й під’єднаємо файли.

Наскрізний приклад: функції

PYTHON · ПРИКЛАД

```python
def valid(records):
    return [
        r for r in records
        if 0 <= r["score"] <= 100
    ]
def average(records):
    if not records:
        raise ValueError("Немає оцінок")
    return (sum(r["score"] for r in records)
            / len(records))
records = [
    {"name": "Олена", "score": 85},
    {"name": "Ігор", "score": 45},
    {"name": "Марія", "score": 110},
]
clean = valid(records)
print(len(clean), f"{average(clean):.2f}")
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
2 65.00
```

Пояснення

Відбір та середнє — різні функції. Деталі — на наступному слайді.

Наскрізний приклад: функції: пояснення

Окремі функції відбирають записи й обчислюють середнє.

Кожна функція має окрему відповідальність. Це дозволяє перевірити відбір незалежно від середнього та використовувати результати обчислень без друку.

Відбір

```
clean = valid(records)
```

valid вилучає запис з оцінкою 110. У clean залишаються оцінки 85 і 45.

Обчислення

```
average(clean)  # 65.0
```

average ділить суму 130 на кількість записів 2. Результат — число 65.0.

Форматування

```
f"{average(clean):.2f}"
```

Специфікатор .2f формує текст "65.00". Саме число при цьому не змінюється.

Підсумковий вивід: 2 65.00. Недопустима оцінка 110 не бере участі в обчисленні.

Наскрізний приклад: надійність

PYTHON · ПРИКЛАД

```python
def average(scores):
    if not scores:
        raise ValueError("Немає оцінок")
    return sum(scores) / len(scores)

assert average([85, 45]) == 65
for scores in [[85, 45], []]:
    try:
        result = average(scores)
        print(f"Середнє: {result:.2f}")
    except ValueError as error:
        print(
            "Звіт не сформовано:", error
        )
```

КОНСОЛЬ · РЕЗУЛЬТАТ

```
Середнє: 65.00
Звіт не сформовано: Немає оцінок
```

Пояснення

Перевірене середнє непорожнього набору. Деталі — на наступному слайді.

Наскрізний приклад: надійність: пояснення

Успішний розрахунок і відсутність даних мають явні результати.

Порожній набір — окремий сценарій. Явний ValueError відділяє відсутність даних від нормального числового результату, який можна використовувати далі.

Відомий результат

```
assert average(
    [85, 45]) == 65
```

assert перевіряє, що середнє оцінок 85 і 45 дорівнює 65. Хибна перевірка дасть AssertionError.

Порожній список

```
average([])
# ValueError: Немає оцінок
```

Перевірка if not scores у функції зупиняє обчислення до ділення на нуль.

Реакція на збій

```
except ValueError as error:
```

error містить причину збою. print виводить «Звіт не сформовано» та текст винятку.

Для [85, 45] виводиться 65.00; для [] — повідомлення про помилку без аварійного завершення.

Типові помилки функцій

Помилка

Прихований контракт

```
# Результат не повертається
print(sum(values))
# Спільний список: items=[]
items.append(value)
# Виняток приховано
except:
    pass
```

Друк не повертає значення. Список між викликами накопичує зміни. Помилка залишається без реакції.

Виправлення

Явний результат

```
return sum(values)
# Параметр функції: items=None
if items is None:
    items = []
items.append(value)
except ValueError:
    print("Некоректне число")
```

Повертаємо результат, створюємо окремий список і обробляємо конкретний виняток із повідомленням.

Показано окремі фрагменти функцій і try/except. Перевіряйте повернення, стан та обробку помилок.

Підсумок і самоперевірка

Перевірте, чи можете пояснити основні конструкції лекції.

Питання 1

Чим параметр функції відрізняється від аргументу?

Питання 2

Чому присвоєння a = a.sort() зберігає None?

Питання 3

Які перевірки потрібні для оцінок поблизу порога 60?

Самостійна робота перед лекцією 3

Використайте матеріал лекції, щоб підготувати власні відповіді та приклади.

Питання 1

Як розділити перевірку оцінки, визначення успішності та обчислення середнього на окремі функції?

Питання 2

Як обробити порожній набір оцінок і значення за межами 0–100?

Питання 3

Як перевірити функції для оцінок 59, 60, 61 та 110?