Національний університет біоресурсів і
природокористування України

Структуризація програмного коду в Python: функції, модулі та пакети

Частина 2

Лектор: к.ф.-м.н., доцент Кириченко Віктор Вікторович

Зміст лекції

Рекурсія
Поняття, приклади та особливості реалізації у Python

Модулі в Python
Поняття, структура та призначення

Вбудовані модулі
math, os, sys, datetime тощо

Імпорт модулів
Способи, особливості та найкращі практики

Створення власних модулів
Практичні аспекти розробки

Пакети в Python
Структура, файл __init__.py, організація

Створення власних пакетів
Кроки та рекомендації

Кращі практики
Структуризація проектів

Рекурсія в Python

Що таке рекурсія?

Рекурсія — це метод програмування, де функція викликає саму себе для вирішення задачі.

Кожен рекурсивний виклик працює з меншою підзадачею, наближаючись до базового випадку.

Ключові компоненти рекурсії:

- **Базовий випадок** — умова завершення рекурсії
- **Рекурсивний крок** — виклик функції самої себе
- **Зміна стану** — наближення до базового випадку

Переваги та недоліки

Переваги

- Елегантний код
- Простота для певних задач
- Природність для рекурсивних структур

Недоліки

- Високе споживання пам'яті
- Ризик переповнення стеку
- Складність налагодження

Обмеження в Python

- Максимальна глибина рекурсії за замовчуванням: ~1000
- Можна змінити за допомогою `sys.setrecursionlimit()`
- Python не оптимізує хвостову рекурсію

Приклади рекурсії в Python

Факторіал числа

```python
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)

# Приклад виклику
print(factorial(5)) # Виведе: 120
```

Послідовність Фібоначчі

```python
def fibonacci(n):
  if n <= 1:
    return n
  else:
    return fibonacci(n-1) + fibonacci(n-2)

# Приклад виклику
print(fibonacci(7)) # Виведе: 13
```

Пояснення роботи

- **Базовий випадок:** n == 0
- **Рекурсивний крок:** n * factorial(n-1)
- **Зміна стану:** зменшення n на 1

Особливості реалізації

- **Базовий випадок:** n <= 1
- **Рекурсивний крок:** fibonacci(n-1) + fibonacci(n-2)
- **Недолік:** експоненційна складність
- **Оптимізація:** мемоізація або динамічне програмування

Особливості та обмеження рекурсії в Python

Глибина рекурсії

```python
import sys
print(sys.getrecursionlimit())
# Виведе: 1000 (за замовчуванням)
def infinite_recursion():
  infinite_recursion()
infinite_recursion() # RecursionError
```

- Обмеження для захисту від переповнення стеку
- Можна змінити: `sys.setrecursionlimit()`

Використання пам'яті

- Кожен рекурсивний виклик створює новий кадр стеку
- Локальні змінні зберігаються в кожному кадрі
- Високе споживання пам'яті при великій глибині
- Ризик виникнення помилки `MemoryError`

Продуктивність

Рекурсія	Ітерація
Повільніше	Швидше
Більше пам'яті	Менше пам'яті
Простіший код	Складніший код

Python не оптимізує хвостову рекурсію, на відміну від деяких інших мов

Оптимізація рекурсії

- **Мемоізація** — кешування результатів
- **Декоратори** для автоматичного кешування
- **Динамічне програмування** для зменшення кількості викликів
- **Альтернатива** — використання ітерацій там, де це можливо

Модулі в Python

Що таке модуль?

Модуль — це файл з розширенням `.py`, що містить визначення Python (функції, класи, змінні) та виконуваний код.

Модулі дозволяють логічно організувати код, групуючи пов'язані функції, класи та змінні.

Призначення модулів

- **Ре використання коду** — один модуль можна використовувати в багатьох програмах
- **Організація коду** — логічне групування функціональності
- **Інкапсуляція** — приховування деталей реалізації
- **Уникнення конфліктів імен** — простори імен

Структура модуля

```python
# my_module.py
"""Це приклад модуля Python"""

def greet(name):
  return f"Привіт, {name}!"
  class Calculator:
    def add(self, a, b):
      return a + b
      PI = 3.14159
      # Код, що виконується при імпорті
      print("Модуль завантажено")
      Модуль може містити функції, класи, змінні та
      виконуваний код.
```

Використання модулів

```python
import my_module
print(my_module.greet("Світ"))
calc = my_module.Calculator()
print(calc.add(5, 3))
print(my_module.PI)
```

Вбудовані модулі Python (Частина 1)

math

Математичні функції та константи

```
import math
print(math.pi)
print(math.sqrt(16))
print(math.sin(math.pi/2))
```

- Константи: pi, e
- Функції: sqrt(), sin(), cos()
- Округлення: ceil(), floor()

os

Взаємодія з операційною системою

```
import os
print(os.getcwd())
print(os.listdir())
print(os.path.exists("file.txt"))
```

- Робота з файлами та каталогами
- Змінні середовища
- Керування процесами

sys

Доступ до системних параметрів

```
import sys
print(sys.version)
print(sys.platform)
print(sys.path)
```

- Інформація про Python
- Шляхи пошуку модулів
- Параметри командного рядка

Переваги використання вбудованих модулів

Перевірена якість

Висока продуктивність

Кросплатформенність

Вбудовані модулі Python (Частина 2)

datetime

Робота з датами та часом

```
import datetime
now = datetime.datetime.now()
print(now.strftime("%Y-%m-%d"))
delta = datetime.timedelta(days=7)
```

- date, time, datetime
- Форматування та парсинг
- Робота з часовими інтервалами

random

Генерація випадкових значень

```
import random
print(random.random())
print(random.randint(1, 10))
items = ['a', 'b', 'c']
print(random.choice(items))
```

- Випадкові числа
- Випадковий вибір зі списку
- Перемішування послідовностей

json

Робота з JSON форматом

```
import json
data = {"name": "John"}
json_str = json.dumps(data)
parsed = json.loads(json_str)
```

- Серіалізація об'єктів
- Десеріалізація JSON
- Робота з файлами

collections

Спеціалізовані контейнери даних

- Counter - для підрахунку об'єктів
- defaultdict - словник зі значенням за замовчуванням
- namedtuple - кортеж з іменованими полями

HTTP requests

HTTP-запити (встановлюється окремо)

- Простий інтерфейс для HTTP-запитів
- Підтримка сесій та cookies
- Обробка JSON-відповідей

Імпорт модулів

Основні способи імпорту

1. Імпорт всього модуля

```
import math
print(math.pi)
print(math.sqrt(16))
```

2. Імпорт конкретних об'єктів

```
from math import pi, sqrt
print(pi)
print(sqrt(16))
```

3. Імпорт з псевдонімом

```
import numpy as np
arr = np.array([1, 2, 3])
```

Додаткові способи імпорту

4. Імпорт всіх об'єктів

```
from math import *
print(pi)
print(sqrt(16))
```

⚠️ Не рекомендується через можливі конфлікти імен

5. Умовний імпорт

```
try:
  import module_name
except ImportError:
  # Альтернативний код
```

6. Імпорт з підпакетів

```
from package.subpackage import module
import package.subpackage.module
```

Пошук модулів при імпорті

Порядок пошуку

1. Поточна директорія
2. Директорії у змінній PYTHONPATH
3. Стандартні директорії встановлення

Перевірка шляхів пошуку

```
import sys
print(sys.path)
```

Найкращі практики імпорту модулів

Організація імпортів

1. Стандартні бібліотеки Python
2. Сторонні бібліотеки (встановлені через pip)
3. Власні модулі та пакети проєкту

```
# Стандартні бібліотеки
import os
import sys

# Сторонні бібліотеки
import numpy as np
import pandas as pd

# Власні модулі
from my_project import utils
```

Стиль та форматування

- Використовуйте `isort` для автоматичного сортування
- Розділяйте групи імпортів порожнім рядком
- Уникайте імпорту з `*` (крім особливих випадків)
- Розміщуйте імпорти на початку файлу

Уникнення проблем

Циклічні імпорти

module_a.py: `from module_b import func_b`
module_b.py: `from module_a import func_a`

Рішення: перенесення імпорту всередину функції

Імпорт під час виконання

```
def load_module(name):
  try:
    module = __import__(name)
  except ImportError:
    return None
```

Продуктивність

- Модулі імпортуються лише один раз
- Використовуйте `importlib.reload()` для повторного завантаження
- Імпортуйте лише необхідні об'єкти
- Відкладений імпорт для важких модулів

Створення власних модулів

Структура модуля

Модуль — це файл Python з розширенням `.py`

```python
# mymodule.py
"""Мій перший модуль"""
def greet(name):
  return f"Привіт, {name}!"
class Calculator:
  def add(self, a, b):
    return a + b
VERSION = "1.0"
```

Ім'я файлу стає назвою модуля

Використання модуля

```python
import mymodule
result = mymodule.greet("Світ")
calc = mymodule.Calculator()
print(calc.add(5, 3))
```

Найкращі практики

- **Документування** — docstring
- **Найменування** — малі літери
- **Унікальність** — без конфліктів
- **Організація** — групування

Важливо!

```python
if __name__ == "__main__":
```

Способи імпорту

1. З поточної директорії

```python
import mymodule
```

2. З іншої директорії

```python
sys.path.append("/path")
```

3. PYTHONPATH

Змінна середовища

Приклад власного модуля

Створення модуля

```python
# string_utils.py
"""Утиліти для рядків"""
def reverse_string(s):
  return s[::-1]
def count_vowels(s):
  vowels = 'aeiou'
  return sum( 1 for char in s if char in vowels )
```

Використання модуля

```python
from string_utils import *
text = "Python чудово!"
reversed_text = reverse_string(text)
vowel_count = count_vowels(text)
print(f"Результат: {reversed_text}")
```

Переваги модулів

- Ре використання коду
- Розширюваність
- Інкапсуляція логіки
- Простота тестування

Приклад результатів

```
Оригінал: Python чудово!
Реверс: !оводучч nohtyP
Голосних: 4
Паліндром "радар": Так
Паліндром "код": Ні
```

Ключові принципи створення модулів

- Єдиний файл для кожного модуля
- Документування через docstring
- Перевірка __name__ == "__main__"

Пакети в Python

Що таке пакет?

Пакет — це спосіб організації пов'язаних модулів в ієрархічну структуру у вигляді директорій.

Пакети дозволяють уникнути конфліктів імен та логічно групувати функціональність у великих проєктах.

Переваги використання пакетів

- **Організація коду** — логічне групування пов'язаних модулів
- **Уникнення конфліктів імен** — простори імен для модулів
- **Масштабованість** — легке розширення проєкту
- **Інкапсуляція** — контроль доступу до функціональності
- **Перерозповсюдження** — легке поширення пов'язаного коду

Структура пакету

```
mypackage/
├── __init__.py
├── module1.py
├── module2.py
└── subpackage/
├── __init__.py
└── module3.py
```

- Кожна директорія з файлом `__init__.py` є пакетом
- Можна створювати вкладені підпакети

Використання пакетів

```
import mypackage
from mypackage import module1
from mypackage.module1 import function_name
from mypackage.subpackage import module3
```

Приклад реального пакету

Пакет `numpy` містить безліч модулів:

- `numpy.core` — базові функції
- `numpy.linalg` — лінійна алгебра
- `numpy.random` — генератори випадкових чисел

Файл __init__.py

Призначення файлу

__init__.py — це спеціальний файл, який перетворює директорію на пакет Python.

- Позначає директорію як пакет
- Виконується при імпорті пакету
- Може бути порожнім або містити код

Структура з __init__.py

```
mypackage/
├── __init__.py # ← Цей файл робить директорію пакетом
├── module1.py
├── module2.py
└── subpackage/
├── __init__.py # ← І цей теж
└── module3.py
```

Функціональність __init__.py

1. Ініціалізація пакету

```python
# mypackage/__init__.py
"""Мій пакет для демонстрації"""
print("Пакет mypackage ініціалізовано")
```

2. Визначення публічного інтерфейсу

```python
# mypackage/__init__.py
from .module1 import func1, MyClass
from .module2 import helper_function
```

3. Керування змінною __all__

```python
# mypackage/__init__.py
__all__ = ['func1', 'MyClass', 'helper_function']
```

Приклад використання

```python
import mypackage
# Виведе: Пакет mypackage ініціалізовано

from mypackage import func1, MyClass
from mypackage import *
# Імпортує тільки об'єкти з __all__
```

Структура пакетів

Найкращі практики організації

- **Ієрархічна структура** — логічне групування модулів
- **Чіткі назви** — зрозумілі та змістовні імена пакетів
- **Єдиний стиль** — послідовність у структурі
- **Модульна архітектура** — слабка зв'язність між модулями

Приклад структури

```
myproject/
├── __init__.py
├── main.py
├── core/
│   ├── __init__.py
│   ├── database.py
│   └── config.py
├── utils/
│   ├── __init__.py
│   ├── helpers.py
│   └── validators.py
└── api/
    ├── __init__.py
    ├── routes.py
    └── models.py
```

Рекомендації щодо назв

Пакети

- Використовуйте малі літери
- Уникайте підкреслень на початку
- Короткі та змістовні назви

Модулі

- Також малі літери та підкреслення
- Назви, що відображають функціональність
- Уникайте конфліктів зі стандартними модулями

Організація модулів

Розділення за функціональністю

- **core/** — основна логіка
- **utils/** — допоміжні функції
- **api/** — інтерфейси API

Розділення за рівнем абстракції

- **low_level/** — низькорівневі операції
- **business_logic/** — бізнес-логіка
- **ui/** — користувацький інтерфейс

Переваги правильної структури

Легкість пошуку

Простота розширення

Зручність командної роботи

Простота тестування

Створення власних пакетів

Кроки створення пакету

1. **Створення структури директорій**
   Створіть головну директорію з назвою пакету
2. **Додавання файлів __init__.py**
   Додайте __init__.py у кожну директорію пакету
3. **Створення модулів**
   Додайте модулі (.py файли) з функціональністю
4. **Налаштування імпорту**
   Визначте публічний інтерфейс у __init__.py

Приклад структури пакету

```
myapp/ # Головна директорія пакету
├── __init__.py # Ініціалізація пакету
├── main.py # Основний модуль
├── utils/ # Підпакет утиліт
│ ├── __init__.py
│ ├── helpers.py
│ └── validators.py
└── models/ # Підпакет моделей
├── __init__.py
└── user.py
```

Налаштування __init__.py

Головний __init__.py

```python
# myapp/__init__.py
"""Мій додаток – приклад пакету"""

from .main import run_app
from .utils.helpers import format_data
from .models.user import User

__version__ = "1.0.0"
__all__ = ['run_app', 'format_data', 'User']
__init__.py підпакета

 # myapp/utils/__init__.py
 """Утиліти для мого додатку"""

 from .helpers import format_data, clean_text
 from .validators import validate_email
```

Використання пакету

Імпорт з пакету

```python
import myapp
from myapp import run_app, User
from myapp.utils import format_data
from myapp.models.user import User
```

Використання функціональності

```python
user = User("Іван", "ivan@example.com")
formatted = format_data(user.data)
run_app()
```

Додаткові кроки для розповсюдження

- Створення файлу setup.py
- Додавання README.md та ліцензії
- Публікація на PyPI

Структура проекту

Рекомендована структура

```
myproject/
├── README.md
├── requirements.txt
├── setup.py
├── .gitignore
├── mypackage/
│   ├── __init__.py
│   ├── module1.py
│   └── submodule/
│   ├── __init__.py
│   └── module2.py
├── tests/
│   ├── test_module1.py
│   └── test_module2.py
├── docs/
│   └── usage.md
└── static/
├── css/
└── js/
```

Типові шаблони проектів

1. Пакет (Package)

- Призначений для розповсюдження
- Містить setup.py для встановлення
- Може бути встановлений через pip

2. Додаток (Application)

- Самостійна програма
- Містить точку входу (main.py)
- Може включати конфігураційні файли

3. Гібридний (Hybrid)

- Поєднує пакет та додаток
- Може бути імпортований або запущений

Важливі файли проекту

R **README.md**
Опис проекту, інструкції з встановлення та використання

S **setup.py**
Метадані та інструкції для встановлення пакету

R **requirements.txt**
Список залежностей проекту

Переваги правильної структури проекту

Швидка розробка

Легка командна робота

Простота оновлення

Простота інтеграції

Приклад структури проекту

Приклад структури веб-додатку

```
blog_app/
├── README.md
├── requirements.txt
├── setup.py
├── .gitignore
├── blog/
│   ├── __init__.py
│   ├── config.py
│   ├── app.py
│   ├── models/
│   │   ├── __init__.py
│   │   ├── user.py
│   │   └── post.py
│   ├── views/
│   │   ├── __init__.py
│   │   ├── main.py
│   │   └── auth.py
│   └── utils/
│       ├── __init__.py
│       └── helpers.py
├── tests/
│   ├── test_models.py
│   └── test_views.py
├── docs/
│   └── usage.md
└── static/
    ├── css/
    └── js/
```

Опис компонентів

Файли конфігурації

- **README.md** — документація проекту
- **requirements.txt** — залежності
- **setup.py** — інструкції встановлення

Основний пакет

- **blog/** — головний пакет додатку
- **app.py** — точка входу додатку
- **config.py** — конфігурація

Підпакети

- **models/** — моделі даних
- **views/** — представлення та логіка
- **utils/** — допоміжні функції

Додаткові директорії

tests/

Модулі для тестування коду

- Юніт-тести
- Інтеграційні тести

docs/

Документація проекту

- Керівництво користувача
- API документація

static/

Статичні файли

- CSS стилі
- JavaScript файли

.gitignore

Файл для Git, що вказує які файли ігнорувати

Переваги такої структури

- Чіткий поділ за функціями
- Легкість внесення змін
- Простота розуміння коду

Поради з організації коду

Документування коду

- **Docstrings** — для функцій, класів та модулів
- **Коментарі** — пояснення складних алгоритмів
- **README.md** — опис проекту та інструкції
- **Документація API** — для публічних інтерфейсів

Версіонування коду

- **Git** — система контролю версій
- **Теги** — для позначення версій
- **Гілки** — для розробки нових функцій
- **Pull requests** — для об'єднання змін

Тестування

- **Юніт-тести** — для окремих функцій та класів
- **Інтеграційні тести** — для перевірки взаємодії
- **Тести покриття** — для перевірки якості тестів
- **Неперервна інтеграція** — автоматичне тестування

Якість коду

- **Стиль коду** — дотримання PEP 8
- **Лінтери** — перевірка якості коду
- **Безпека** — перевірка вразливостей
- **Продуктивність** — оптимізація коду

Додаткові інструменти та практики

Віртуальні середовища

Менеджери залежностей

Аналіз коду

Шаблони проєктування

Висновки

Рекурсія

- Рекурсія — це метод програмування, де функція викликає саму себе
- Кожна рекурсивна функція повинна мати базовий випадок
- Рекурсія в Python має обмеження по глибині викликів
- Рекурсивні рішення часто є елегантними, але менш ефективними

Пакети

- Пакет — це структура директорій, що містить модулі
- Файл __init__.py визначає директорію як пакет
- Пакети дозволяють створювати ієрархічну структуру коду
- Пакети спрощують управління великими проєктами

Модулі

- Модуль — це файл Python, що містить визначення та інструкції
- Python має багато вбудованих модулів для різних завдань
- Існують різні способи імпорту модулів
- Створення власних модулів покращує організацію коду

Структура проекту

- Правильна структура проекту покращує читабельність коду
- Розділення коду за функціональністю спрощує розробку
- Слід дотримуватись єдиного стилю та найкращих практик
- Грамотна структуризація коду забезпечує масштабованість

Загальні висновки

Структуризація коду — основа якісних програм

Модулі та пакети забезпечують ре використання коду

Правильна організація спрощує розширення проєктів