Національний університет біоресурсів і природокористування України

Структуризація програмного коду в Python: функції, модулі та пакети

Частина 1

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

План лекції (Частина 1)

Σ Поняття функції та процедури

Основні визначення та відмінності

<> Створення та виклик функцій

Синтаксис та основні принципи

Аргументи та параметри функцій

Позиційні, іменовані, значення за замовчуванням

Область видимості змінних

Локальні та глобальні змінні

↩ Повернення результатів функції

Ключове слово return та його використання

Анонімні функції (lambda)

Створення компактних функцій

Замикання функцій

Поняття замикань та їх практичне застосування

Поняття функції та процедури

Σ Функція

- Підпрограма, що повертає значення
- Містить ключове слово `return`
- Використовується для обчислення результату
- Може бути частиною виразу

Процедура

- Підпрограма, що не повертає значення
- Виконує певну дію або послідовність дій
- Використовується для зміни стану програми
- Викликається як окрема інструкція

Відмінності у Python

- У Python немає чіткого розмежування між функціями та процедурами
- Процедура - це функція, що повертає значення `None`
- Усі підпрограми в Python оголошуються за допомогою ключового слова `def`

Створення та виклик функцій у Python (Частина 1)

<> Синтаксис функції

- Ключове слово `def` для оголошення
- Назва функції повинна бути змістовною
- Параметри в дужках (можуть бути відсутні)
- Тіло функції з відступом (4 пробіли)

▶ Виклик функції

- Назва функції з дужками `function_name()`
- Аргументи в дужках (якщо потрібно)
- Функція має бути визначена до виклику
- Може бути частиною виразу

❞ Проста функція

```
def greet():
    """Привітальна функція"""
    print("Привіт, світе!")

# Виклик функції
greet()

# Результат:
Привіт, світе!
```

Створення та виклик функцій у Python (Частина 2)

Кращі практики

- Використовуйте змістовні імена функцій
- Додавайте docstring для документації
- Дотримуйтесь принципу єдиної відповідальності
- Уникайте глобальних змінних у функціях

Рекурсивні функції

- Функція, що викликає саму себе
- Потрібна умова виходу з рекурсії
- Ефективна для задач, що мають рекурсивну структуру
- Обмеження глибини рекурсії в Python

Приклад рекурсивної функції

```python
def factorial(n):
    """Обчислення факторіала числа"""
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

# Виклик функції
result = factorial(5)
print(result) # Виведе: 120
```

Аргументи та параметри функцій (Частина 1)

Позиційні аргументи

- Передаються у визначеному порядку
- Кількість має збігатися з кількістю параметрів
- Порядок передачі важливий

```
def greet(name, age):
    print(f"Привіт, {name}! Тобі {age} років.")

# Виклик з позиційними аргументами
greet("Олена", 25)
```

Іменовані аргументи

- Передаються з явним вказанням імені параметра
- Порядок передачі неважливий
- Покращують читабельність коду

```
def greet(name, age):
    print(f"Привіт, {name}! Тобі {age} років.")

# Виклик з іменованими аргументами
greet(age=25, name="Олена")
```

Комбінування аргументів

- Можна комбінувати позиційні та іменовані аргументи
- Спочатку йдуть позиційні, потім іменовані

```
def describe_person(name, age, city="Київ"):
    print(f"{name}, {age} років, місто: {city}")

# Комбінований виклик
describe_person("Іван", 30, city="Львів")
```

Аргументи та параметри функцій (Частина 2)

Значення за замовчуванням

- Параметри з визначеними значеннями за замовчуванням
- Можна не передавати при виклику функції
- Значення за замовчуванням обчислюються один раз

```
def greet(name, greeting="Привіт"):
    print(f"{greeting}, {name}!")

# Виклики з різними аргументами
greet("Олена")
greet("Іван", "Доброго дня")
```

Важливі зауваження

- Змінні об'єкти як значення за замовчуванням
- Потенційні проблеми зі змінюваними об'єктами
- Рекомендоване рішення: використання `None`

```
# Небезпечний підхід
def add_item(item, list=[]):
    list.append(item)
    return list

# Правильний підхід
def add_item(item, list=None):
    if list is None:
        list = []
    list.append(item)
    return list
```

Примусові іменовані аргументи

- Використання символу `*` для розділення параметрів
- Параметри після `*` мають передаватися тільки за іменем

```
def create_user(name, *, age, city):
    print(f"Створено користувача: {name}, {age} років, {city}")

# Правильний виклик
create_user("Марія", age=28, city="Одеса")

# Неправильний виклик (викличе помилку)
# create_user("Марія", 28, "Одеса")
```

Аргументи та параметри функцій (Частина 3)

*args - змінна кількість аргументів

- Дозволяє передавати будь-яку кількість позиційних аргументів
- Аргументи збираються в кортеж (tuple)
- Зазвичай використовується як останній параметр

```
def sum_numbers(*numbers):
    result = 0
    for num in numbers:
        result += num
    return result

# Виклик функції
print(sum_numbers(1, 2, 3)) # 6
print(sum_numbers(10, 20)) # 30
```

**kwargs - змінна кількість іменованих аргументів

- Дозволяє передавати будь-яку кількість іменованих аргументів
- Аргументи збираються в словник (dictionary)
- Зазвичай використовується як останній параметр

```
def print_info(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

# Виклик функції
print_info(name="Олена", age=25, city="Київ")
# Виведе:
# name: Олена
# age: 25
# city: Київ
```

Комбінування *args та **kwargs

- Можна використовувати разом у одній функції
- Порядок параметрів: звичайні, *args, **kwargs

```
def process_data(name, *args, **kwargs):
    print(f"Обробка даних для: {name}")
    if args:
        print("Додаткові значення:", args)
    if kwargs:
        print("Параметри:")
        for key, value in kwargs.items():
            print(f" {key}: {value}")

# Виклик функції
process_data("Датасет", 10, 20, 30, format="csv", verbose=True)
```

Область видимості змінних (Частина 1)

Глобальні змінні

- Доступні в усьому модулі
- Оголошуються поза функціями
- Для читання доступні без обмежень
- Для зміни потрібне ключове слово `global`

```python
# Глобальна змінна
global_var = "Я глобальна"

def show_global():
    print(global_var)

def change_global():
    global global_var
    global_var = "Змінено глобальну"
```

Локальні змінні

- Доступні лише всередині функції
- Створюються при виклику функції
- Знищуються після завершення функції
- Не доступні поза функцією

```python
def calculate():
    # Локальні змінні
    x = 10
    y = 20
    result = x + y
    print(f"Результат: {result}")
    return result

# Помилка: змінні x, y, result недоступні
# print(x) # NameError
```

Область видимості змінних (Частина 2)

Правила доступу до змінних (LEGB)

- **L**ocal (локальні) - змінні, визначені всередині функції
- **E**nclosing (зовнішні) - змінні з зовнішньої функції (для вкладених функцій)
- **G**lobal (глобальні) - змінні на рівні модуля
- **B**uilt-in (вбудовані) - вбудовані функції та змінні Python

```
# Приклад пошуку змінних за правилом LEGB
x = "глобальна" # Глобальна область

def outer():
    x = "зовнішня" # Зовнішня область

    def inner():
        x = "локальна" # Локальна область
        print(x) # Виведе: "локальна"

    inner()
    print(x) # Виведе: "зовнішня"

outer()
print(x) # Виведе: "глобальна"
```

Область видимості змінних (Частина 3)

Вкладені функції

- Функція, визначена всередині іншої функції
- Має доступ до змінних зовнішньої функції
- Не може бути викликана поза зовнішньою функцією

```
def outer():
    msg = "Зовнішнє повідомлення"

    def inner():
        print(msg) # Доступ до змінної

    inner() # Виклик внутрішньої функції

outer() # Виведе: "Зовнішнє повідомлення"
```

Ключове слово nonlocal

- Використовується для зміни змінних з зовнішньої області
- Не працює для глобальних змінних
- Шукає змінну в найближчій зовнішній області видимості

```
def outer():
    count = 0

    def inner():
        nonlocal count
        count += 1
        print(count)

    inner() # Виведе: 1
    inner() # Виведе: 2

outer()
```

Область видимості змінних (Частина 4)

Правила роботи з областями видимості

- Змінні, створені всередині функції, є локальними для цієї функції
- Для доступу до глобальних змінних використовуйте `global`
- Для доступу до змінних з зовнішньої функції використовуйте `nonlocal`

```
# Приклад з global та nonlocal
x = "глобальна"

def outer():
    x = "зовнішня"

    def inner():
        nonlocal x
        global global_x
        x = "нова зовнішня"
        global_x = "нова глобальна"

    inner()
    print(x) # Виведе: "нова зовнішня"

outer()
print(global_x) # Виведе: "нова глобальна"
```

Область видимості змінних (Частина 5)

Вкладені функції

- Функція, визначена всередині іншої функції
- Має доступ до змінних зовнішньої функції
- Не може бути викликана поза зовнішньою функцією

```
def outer():
    msg = "Зовнішнє повідомлення"

    def inner():
        print(msg) # Доступ до змінної

    inner() # Виклик внутрішньої функції

outer() # Виведе: "Зовнішнє повідомлення"
```

Ключове слово nonlocal

- Використовується для зміни змінних з зовнішньої області
- Не працює для глобальних змінних
- Шукає змінну в найближчій зовнішній області видимості

```
def outer():
    count = 0

    def inner():
        nonlocal count
        count += 1
        print(count)

    inner() # Виведе: 1
    inner() # Виведе: 2

outer()
```

Область видимості змінних (Частина 6)

Правила роботи з областями видимості

- Змінні, створені всередині функції, є локальними для цієї функції
- Для доступу до глобальних змінних використовуйте `global`
- Для доступу до змінних з зовнішньої функції використовуйте `nonlocal`

```
# Приклад з global та nonlocal
x = "глобальна"

def outer():
    x = "зовнішня"

    def inner():
        nonlocal x
        global global_x
        x = "нова зовнішня"
        global_x = "нова глобальна"

    inner()
    print(x) # Виведе: "нова зовнішня"

outer()
print(global_x) # Виведе: "нова глобальна"
```

Повернення результатів функції (Частина 1)

↩ Ключове слово return

- Завершує виконання функції
- Повертає значення у точку виклику
- Може повертати будь-який тип даних
- Функція без return повертає None

```
def add(a, b):
    result = a + b
    return result

# Виклик функції
sum_result = add(5, 3)
print(sum_result) # Виведе: 8
```

Раннє повернення

- Функція може мати кілька операторів return
- Використовується для умовного повернення
- Покращує читабельність коду

```
def check_number(num):
    if num < 0:
        return "Від'ємне"
    elif num > 0:
        return "Додатне"
    else:
        return "Нуль"

print(check_number(-5)) # Виведе: "Від'ємне"
```

Повернення результатів функції (Частина 2)

Σ Повернення різних типів даних

- Функція може повертати різні типи даних залежно від умов
- Повернення колекцій (списки, кортежі, словники)

```
def get_student_info(request_type):
    if request_type == "name":
        return "Іван Петренко"
    elif request_type == "grades":
        return [90, 85, 92]
    elif request_type == "details":
        return {
            "name": "Іван Петренко",
            "age": 20,
            "faculty": "Інформатика"
        }
    else:
        return None

# Виклики функції
name = get_student_info("name")
grades = get_student_info("grades")
details = get_student_info("details")
```

Повернення результатів функції (Частина 3)

Кілька значень return

- Функція може повертати кілька значень
- Фактично повертає кортеж (tuple)
- Можна розпакувати в окремі змінні

```
def min_max(numbers):
    return min(numbers), max(numbers)

# Виклик функції
nums = [4, 2, 9, 7, 5]
minimum, maximum = min_max(nums)
print(minimum) # Виведе: 2
print(maximum) # Виведе: 9
```

Повернення функцій

- Функція може повертати іншу функцію
- Корисно для створення фабрик функцій
- Дозволяє динамічно створювати функції

```
def make_multiplier(n):
    def multiplier(x):
        return x * n
    return multiplier

# Виклик функції
times3 = make_multiplier(3)
times5 = make_multiplier(5)
print(times3(10)) # Виведе: 30
print(times5(10)) # Виведе: 50
```

Повернення результатів функції (Частина 4)

Розширені можливості return

- Повернення генераторів та ітераторів
- Використання yield для створення генераторів

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# Виклик функції-генератора
fib_gen = fibonacci(10)
for num in fib_gen:
    print(num, end=" ")
# Виведе: 0 1 1 2 3 5 8 13 21 34
```

Анонімні функції (lambda) (Частина 1)

Синтаксис lambda

- Ключове слово `lambda` для створення
- Містить лише один вираз
- Не потребує імені
- Автоматично повертає результат

```
# Базовий синтаксис
lambda аргументи: вираз

# Приклад
square = lambda x: x**2
print(square(5)) # 25
```

Lambda vs звичайна функція

- Обмеження: лише один вираз
- Немає інструкцій (if, for, while)
- Немає анотацій типів
- Короткий синтаксис для простих операцій

```
# Звичайна функція
def add(a, b):
    return a + b

# Еквівалентна lambda
add_lambda = lambda a, b: a + b

# Виклик
print(add(3, 5)) # 8
print(add_lambda(3, 5)) # 8
```

Анонімні функції (lambda) (Частина 2)

Коли використовувати lambda

- Для коротких, одноразових функцій
- Як аргументи функцій вищого порядку (map, filter, sorted)
- Для функцій зворотного виклику (callbacks)

```
# Використання lambda з map()
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared) # [1, 4, 9, 16, 25]

# Використання lambda з filter()
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers) # [2, 4]

# Використання lambda з sorted()
points = [(1, 2), (3, 1), (2, 3)]
sorted_points = sorted(points, key=lambda p: p[1])
print(sorted_points) # [(3, 1), (1, 2), (2, 3)]
```

Анонімні функції (lambda) (Частина 3)

Функції вищого порядку

- `reduce()` - накопичення результатів
- `sorted()` - кастомне сортування
- `max()` / `min()` - з ключем

```
# Використання lambda з reduce()
from functools import reduce
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # 120

# Використання lambda з max()
students = [
  {"name": "Олена", "grade": 90},
  {"name": "Іван", "grade": 85}
]
top_student = max(students, key=lambda s: s["grade"])
print(top_student)  # {"name": "Олена", "grade": 90}
```

Умовні вирази в lambda

- Використання тернарного оператора
- Логічні операції (and/or)
- Обмеження складності умов

```
# Тернарний оператор
absolute = lambda x: x if x >= 0 else -x
print(absolute(-5))  # 5

# Логічні операції
is_even = lambda x: x % 2 == 0
check_even = lambda x: "Парне" if is_even(x) else "Непарне"
print(check_even(4))  # "Парне"
print(check_even(5))  # "Непарне"
```

Анонімні функції (lambda) (Частина 4)

Практичні приклади використання

- Обробка даних у Pandas DataFrame
- Створення обробників подій
- Функціональне програмування

Приклад з Pandas

```
# Приклад з Pandas
import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df['C'] = df.apply(lambda row: row['A'] + row['B'], axis=1)
```

Функціональний підхід

```
# Функціональний підхід
numbers = [1, 2, 3, 4, 5, 6]
result = list(map(
    lambda x: x**2,
    filter(lambda x: x % 2 == 0, numbers)
))
print(result)  # [4, 16, 36]
```

Замикання функцій (Частина 1)

Поняття замикання

- Функція, що "запам'ятовує" своє оточення
- Має доступ до змінних з зовнішньої області видимості
- Зберігає стан між викликами
- Створюється при визначенні вкладеної функції

```
# Простий приклад замикання
def outer(msg):
    message = msg

    def inner():
        print(message)

    return inner

# Створення замикання
hello_func = outer("Привіт!")
hello_func() # Виведе: "Привіт!"
```

Умови створення замикання

- Повинна бути вкладена функція
- Вкладена функція має посилатися на змінні з зовнішньої функції
- Зовнішня функція має повертати вкладену функцію

```
# Перевірка наявності замикання
def multiplier(n):
    def inner(x):
        return x * n
    return inner

times3 = multiplier(3)

# Перевірка атрибута __closure__
print(times3.__closure__) # (,)
print(times3.__closure__[0].cell_contents) # 3
```

Замикання функцій (Частина 2)

Збереження стану в замиканні

- Замикання дозволяє створювати функції зі станом
- Змінні зовнішньої функції зберігаються між викликами

```
# Лічильник з використанням замикання
def make_counter():
    count = 0

    def counter():
        nonlocal count
        count += 1
        return count

    return counter

# Створення та використання лічильника
counter1 = make_counter()
print(counter1()) # 1
print(counter1()) # 2
print(counter1()) # 3

# Створення незалежного лічильника
counter2 = make_counter()
print(counter2()) # 1
```

Замикання функцій (Частина 3)

Декоратори на основі замикань

- Декоратори - це замикання, що приймають функцію
- Дозволяють розширювати функціональність функцій
- Не змінюють початковий код функції

```
# Простий декоратор
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Викликано {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Завершено {func.__name__}")
        return result
    return wrapper

# Застосування декоратора
@logger
def add(a, b):
    return a + b

add(3, 5) # Виведе логування
```

Інкапсуляція даних

- Замикання для приховування даних
- Створення приватних змінних
- Захист від несанкціонованого доступу

```
# Приховання даних за допомогою замикання
def make_account(initial_balance):
    balance = initial_balance

    def get_balance():
        return balance

    def deposit(amount):
        nonlocal balance
        balance += amount
        return balance

    def withdraw(amount):
        nonlocal balance
        if amount <= balance:
            balance -= amount
            return True
        return False

    return {'get': get_balance, 'deposit': deposit, 'withdraw':
withdraw}
```

Замикання функцій (Частина 4)

Практичні застосування замикань

- Створення фабрик функцій
- Кешування результатів обчислень
- Реалізація шаблону "Модуль" для інкапсуляції

```
# Фабрика функцій для обчислення степенів
def power_factory(exponent):
    def power(base):
        return base ** exponent
    return power

square = power_factory(2)
cube = power_factory(3)

print(square(4)) # 16
print(cube(3)) # 27

# Кешування результатів
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper
```

Практичні приклади (Частина 1)

Функція для обчислення статистики

```python
def calculate_statistics(numbers):
    # Обчислення середнього значення
    average = sum(numbers) / len(numbers)

    # Обчислення мінімуму та максимуму
    min_val = min(numbers)
    max_val = max(numbers)

    # Обчислення медіани
    sorted_nums = sorted(numbers)
    n = len(sorted_nums)
    if n % 2 == 0:
        median = (sorted_nums[n//2-1] + sorted_nums[n//2]) / 2
    else:
        median = sorted_nums[n//2]

    return {
        'average': average,
        'min': min_val,
        'max': max_val,
        'median': median
    }

# Використання функції
data = [12, 15, 18, 22, 25, 30]
stats = calculate_statistics(data)
print(stats)
```

Функція фільтрації та трансформації

```python
def process_data(data, filter_func=None, transform_func=None):
    # Застосування фільтра, якщо він наданий
    if filter_func:
        data = [item for item in data if filter_func(item)]

    # Застосування трансформації, якщо вона надана
    if transform_func:
        data = [transform_func(item) for item in data]

    return data

# Функції-фільтри та трансформації
is_even = lambda x: x % 2 == 0
square = lambda x: x ** 2

# Використання функції
numbers = [1, 2, 3, 4, 5, 6]
result = process_data(numbers, is_even, square)
print(result) # [4, 16, 36]
```

Практичні приклади (Частина 2)

Функція з параметрами за замовчуванням та змінною кількістю аргументів

```
def create_user_profile(name, age, *, city="Київ", **kwargs):
    # Створення базового профілю
    profile = {
        'name': name,
        'age': age,
        'city': city
    }

    # Додавання додаткових параметрів
    for key, value in kwargs.items():
        profile[key] = value

    # Функція для форматування профілю
    def format_profile():
        result = f"Профіль користувача:\n"
        for key, value in profile.items():
            result += f" {key}: {value}\n"
        return result

    return format_profile()

# Використання функції
profile1 = create_user_profile("Олена", 25)
profile2 = create_user_profile("Іван", 30, city="Львів", occupation="Інженер")

print(profile1)
print(profile2)
```

Практичні приклади (Частина 3)

Комбінування lambda та замикань

```
# Створення функції з параметром для lambda
def make_operation(operation):
    # Повертаємо lambda, що виконує операцію
    return lambda x, y: operation(x, y)

# Створення операцій
add = make_operation(lambda a, b: a + b)
multiply = make_operation(lambda a, b: a * b)
power = make_operation(lambda a, b: a ** b)

# Використання
print(add(5, 3)) # 8
print(multiply(5, 3)) # 15
print(power(5, 3)) # 125
```

Σ Функція вищого порядку

```
# Функція, що приймає та повертає функції
def compose(*functions):
    # Повертаємо нову функцію
    def inner(x):
        result = x
        # Застосовуємо всі функції послідовно
        for f in functions:
            result = f(result)
        return result
    return inner

# Визначення простих функцій
add5 = lambda x: x + 5
multiply2 = lambda x: x * 2

# Композиція функцій
transform = compose(add5, multiply2)
print(transform(10)) # 30
```

Практичні приклади (Частина 4)

Комплексний приклад з декоратором

```python
# Декоратор для кешування результатів
def memoize(func):
    cache = {}
    def wrapper(*args, **kwargs):
        # Створення ключа кешу з аргументів
        key = str(args) + str(kwargs)
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]
    return wrapper

# Функція для обчислення чисел Фібоначчі
@memoize
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Функція для обчислення факторіала
@memoize
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n-1)

# Використання функцій
print(fibonacci(10)) # 55
print(factorial(10)) # 3628800
```

Підсумки

Функції та процедури

- Функція - підпрограма, що повертає значення
- Процедура - підпрограма, що не повертає значення
- В Python обидві реалізуються через `def`

Аргументи та параметри

- Позиційні та іменовані аргументи
- Значення за замовчуванням
- `*args` та `**kwargs` для змінної кількості аргументів

Область видимості змінних

- Локальні та глобальні змінні
- Ключові слова `global` та `nonlocal`
- Правило LEGB для пошуку змінних

Повернення результатів

- Оператор `return` для повернення значень
- Повернення кількох значень (кортеж)
- Повернення функцій та генераторів

Анонімні функції

- Ключове слово `lambda` для створення
- Використання з функціями вищого порядку
- Обмеження: лише один вираз

Замикання функцій

- Функції, що "запам'ятовують" своє оточення
- Зберігають стан між викликами
- Використання для декораторів та інкапсуляції