Конспект лекції (7)

▼ Навіщо потрібні функції

Функції в Python є важливим елементом програмування і використовуються для реалізації багатьох функціональностей у програмах. Ось деякі основні причини, чому функції в Python є корисними:

1. **Підтримка коду:**
    - **Повторне використання:** Функції дозволяють визначити частину коду і використовувати її багато разів у програмі, що сприяє повторному використанню коду та полегшує зміни.
    - **Модульність:** Розділення програми на функції дозволяє розробляти і тестувати їх незалежно, що сприяє більшій читабельності та роботі над окремими частинами коду.
2. **Структурування програм:**
    - Функції допомагають впорядковувати код і структурувати його у логічні блоки.
    - Визначення функцій надає програмі структуру і розділяє її на окремі завдання.
3. **Абстракція:**
    - Функції дозволяють абстрагувати конкретні дії або операції, роблячи їх інтерфейсом для користувача (іншого програміста).
    - Абстрагування полегшує розуміння та використання коду, оскільки деталі реалізації можуть бути приховані.
4. **Параметри та повернення значень:**
    - Функції можуть приймати параметри, що дозволяє їм обробляти різні дані залежно від контексту виклику.
    - Функції можуть повертати значення, що дозволяє їм передавати результати своєї роботи.
5. **Модульність та імпорт:**
    - Визначені функції можуть бути розміщені у різних модулях, і їх можна імпортувати для використання в інших програмах або модулях.
    - Це сприяє розбиттю коду на логічно пов'язані компоненти.
6. **Зменшення дублювання коду:**
    - Функції дозволяють визначити один раз фрагмент коду та використовувати його в різних частинах програми, уникнувши дублювання коду.
7. **Обробка подій:**

- Функції використовуються для обробки подій, таких як натискання кнопок, миші тощо.

8. **Рекурсія:**
    - Функції можуть викликати сами себе, що дозволяє вирішувати складні завдання шляхом поділу їх на більш прості підзадачі.

Функції є потужним інструментом в Python, який полегшує розробку програм та підтримку коду.

> 💡 **Функція ЗАВЖДИ повертає результат своєї роботи.**
> **Якщо результат не вказаний в `return`,**
> **то за замовченням це буде `None`**

▼ Вбудовані функції python

```
abs()  #    Returns the absolute value of a number
# повертає абсолютне значення (модуль) числа
all()  #  Returns True if all items in an iterable object are true
any()  #  Returns True if any item in an iterable object is true
ascii()  #  Returns a readable version of an object.
# Replaces none-ascii characters with escape character
bin()  #  Returns the binary version of a number
bytearray()  #  Returns an array of bytes
bytes()  #  Returns a bytes object
callable()  # Returns True if the specified object is callable, otherwise False
chr()  #  Returns a character from the specified Unicode code.
classmethod()  #  Converts a method into a class method
compile()  #  Returns the specified source as an object, ready to be executed
complex()  #  Returns a complex number
delattr()  #  Deletes the specified attribute (property or method) from the specified object
dict()  # Returns a dictionary
dir()  #  Returns a list of the specified object's properties and methods
divmod()  # Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()  #  Takes a collection (e.g. a tuple) and returns it as an enumerate object
# повертає пари `(індекс, елемент)` для кожного елемента в ітерабельному об'єкті.
eval()  # Evaluates and executes an expression
exec()  # Executes the specified code (or object)
filter()  # Use a filter function to exclude items in an iterable object
float()  #  Returns a floating point number
format()  # Formats a specified value
frozenset()  #  Returns a frozenset object
getattr()  #  Returns the value of the specified attribute (property or method)
globals()  #  Returns the current global symbol table as a dictionary
hasattr()  #  Returns True if the specified object has the specified attribute (property/method)
# hasattr(object_name, "pop")
hash()  # Returns the hash value of a specified object
help()  # Executes the built-in help system
hex()  #  Converts a number into a hexadecimal value
id()  # Returns the id of an object
input()  #  Allowing user input
int()  #  Returns an integer number
```

```
isinstance()  # Returns True if a specified object is an instance of a specified object
issubclass()  # Returns True if a specified class is a subclass of a specified object
iter()  # Returns an iterator object
len()  #  Returns the length of an object
list()  # Returns a list
locals()  # Returns an updated dictionary of the current local symbol table
map()  #  Returns the specified iterator with the specified function applied to each item
max()  #  Returns the largest item in an iterable
memoryview()  # Returns a memory view object
min()  #  Returns the smallest item in an iterable
next()  # Returns the next item in an iterable
object()  # Returns a new object
oct()  #  Converts a number into an octal
open()  # Opens a file and returns a file object
ord()  #  Convert an integer representing the Unicode of the specified character
pow()  #  Returns the value of x to the power of y
print()  #  Prints to the standard output device
property()  # Gets, sets, deletes a property
range()  #  Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()  # Returns a readable version of an object
reversed()  # Returns a reversed iterator
round()  #  Rounds a numbers
set()  #  Returns a new set object
setattr()  #  Sets an attribute (property/method) of an object
slice()  #  Returns a slice object
sorted()  # Returns a sorted list
staticmethod()  # Converts a method into a static method
str()  #  Returns a string object
sum()  #  Sums the items of an iterator
super()  #  Returns an object that represents the parent class
tuple()  #  Returns a tuple
type()  # Returns the type of an object
vars()  # Returns the **dict** property of an object
zip()  #  Returns an iterator, from two or more iterators
```

▼ Деякі корисні вбудовані функції та особливості їх застосування

▼ Функція map()

`map()` - це вбудована функція в Python, яка дозволяє застосовувати функцію до кожного елемента в ітерованому об'єкті та повернути новий ітератор, що містить результати.

Ось приклад використання `map()` :

```
base_numbers = [2, 4, 6, 8, 10]
powers = [1, 2, 3, 4, 5]
numbers_powers = list(map(pow, base_numbers, powers))

print(numbers_powers)  # [2, 16, 216, 4096, 100000]
```

Цей код використовує функцію `map()` для обчислення ступенів чисел зі списку `base_numbers` згідно з відповідними елементами списку `powers`. Результатом є новий список, в якому кожен елемент є результатом піднесення до ступеня відповідного числа з `base_numbers` за допомогою відповідного елемента з `powers`.

Давайте розглянемо кожен крок:

1. `base_numbers = [2, 4, 6, 8, 10]`: Створення списку `base_numbers` з певними числами.
2. `powers = [1, 2, 3, 4, 5]`: Створення списку `powers` з експонентами (ступенями), які будуть використовуватися для підняття до ступеня чисел з `base_numbers`.
3. `numbers_powers = list(map(pow, base_numbers, powers))`: Застосування функції `pow()` до кожної пари чисел з `base_numbers` і `powers`. Результатом є новий список `numbers_powers`, в якому кожен елемент є результатом підняття до ступеня відповідного числа з `base_numbers` за допомогою відповідного елемента з `powers`.
4. `print(numbers_powers)`: Виведення результату - список чисел, піднятих до відповідних ступенів:

```
[2, 16, 216, 4096, 100000]
```

Отже, `numbers_powers` містить результат підняття кожного числа з `base_numbers` до відповідного ступеня з `powers`.

Ось ще один приклад використання `map()` для конвертації рядків у верхній регістр:

```
list_of_words = ['apple', 'banana', 'cherry']
upper_words = list(map(str.upper, list_of_words))
print(upper_words) # ['APPLE', 'BANANA', 'CHERRY']
```

В цьому прикладі ми використовуємо вбудовану функцію `str.upper()`, щоб перетворити кожен елемент списку `words` у верхній регістр. Результати також зберігаються в ітераторі, який ми перетворюємо на список за допомогою `list()`.

map vs list comprehension

Як правило, замість map можна використовувати list comprehension. Найчастіше варіант з list comprehension більш зрозумілий, а в деяких випадках навіть швидше.

Але map може бути ефективнішим у тому випадку, коли треба згенерувати велику кількість елементів, тому що map - ітератор, а list comprehension генерує список.

Приклади, аналогічний наведеним вище, у варіанті з list comprehension.

Перекласти всі рядки у верхній регістр:

```
list_of_words = ['apple', 'banana', 'cherry']
upper_words = [word.upper() for word in list_of_words]
print(upper_words) # ['APPLE', 'BANANA', 'CHERRY']
```

▼ Функція zip()

`zip()` - це вбудована функція в Python, яка дозволяє об'єднати декілька ітерованих об'єктів (списків, кортежів тощо) в один ітератор, в якому кожен елемент буде містити відповідні елементи з кожного вхідного об'єкта.

Ось приклад використання `zip()` для об'єднання двох списків:

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
zipped = zip(list1, list2)
print(list(zipped)) # [(1, 'a'), (2, 'b'), (3, 'c')]
```

В цьому прикладі ми використовуємо `zip()` для об'єднання двох списків `list1` та `list2`. Кожен елемент ітератора `zipped` міститиме кортеж з елементами з відповідних позицій у `list1` та `list2`.

Якщо довжина вхідних об'єктів не співпадає, `zip()` буде зупинятись при досягненні кінця найкоротшого вхідного об'єкту:

```
list1 = [1, 2, 3]
list2 = ['a', 'b']
zipped = zip(list1, list2)
print(list(zipped)) # [(1, 'a'), (2, 'b')]
```

`zip()` може бути корисним, коли потрібно обробити декілька ітерованих об'єктів паралельно. Наприклад, можна використовувати `zip()` для обчислення суми відповідних елементів у декількох списків:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
sums = [a + b for a, b in zip(list1, list2)]
print(sums) # [5, 7, 9]
```

▼ Коли вживати isinstance() а коли type()

`type()` і `isinstance()` - це дві різні функції, які можуть бути використані для отримання інформації про типи об'єктів, проте вони мають різні застосування:

1. **`type()`**:
   - `type(obj)` повертає конкретний тип об'єкта `obj`. Наприклад, `type(42)` поверне `<class 'int'>`.
   - Часто використовується, коли вам потрібно точно визначити тип об'єкта, і ви вже знаєте, якого типу об'єкт вам треба порівняти.
2. **`isinstance()`**:
   - `isinstance(obj, class_or_tuple)` перевіряє, чи є `obj` екземпляром класу чи класів, які передаються у `class_or_tuple`.

- Використовується, коли вам потрібно визначити, чи об'єкт є екземпляром певного класу або **одного з декількох класів.**

```
x = 5
print(isinstance(x, int)) # True
```

- Може бути корисним у випадках, коли вам необхідно перевірити, чи об'єкт є екземпляром класу, який може мати декілька базових класів або налагоджуваних класів.

Вибір між `type()` і `isinstance()` залежить від конкретних потреб вашого коду. В більшості випадків `isinstance()` більш гнучка, оскільки вона дозволяє перевіряти налагодженість об'єкта відносно кількох класів.

▼ Різниця sort() та sorted()

Обидві функції — `sort()` та `sorted()` — призначені для сортування послідовностей в Python, але вони мають деякі важливі відмінності:

1. **`sorted(iterable, key=None, reverse=False)`**:
   - `sorted()` є вбудованою функцією, яка повертає новий список, що містить всі елементи ітерабельного об'єкта, відсортовані за певним порядком.
   - Не змінює оригінальний об'єкт, але повертає новий відсортований список.
   - Приймає параметр `key`, який можна використовувати для визначення функції, яка визначає порядок сортування.
   - Приймає параметр `reverse`, який, якщо встановлений в `True`, сортує в зворотньому порядку.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
sorted_numbers = sorted(numbers)
print(sorted_numbers) # Виведе: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]`
```

2. **`list.sort(key=None, reverse=False)`**:
   - `sort()` — це метод списку, який змінює оригінальний список, сортуючи його елементи.
   - Також приймає параметри `key` і `reverse`, але вони використовуються так само, як у `sorted()`.
   - Оскільки `sort()` змінює сам список, він повертає `None`.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
numbers.sort()
print(numbers) # Виведе: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

Отже, основна відмінність між ними полягає в тому, що `sorted()` повертає новий відсортований список, залишаючи оригінал без змін, тоді як `sort()` сортує сам список, змінюючи його. Якщо вам потрібно просто отримати відсортовану версію послідовності, і вам цікава вхідна послідовність, використовуйте `sorted()`. Якщо вам потрібно сортувати і зберігати відсортовані дані в тому самому списку, використовуйте `sort()`.

▼ Створення функції

Функції визначаються за допомогою ключового слова **def**.
Для того, щоб створити функцію потрібно розмістити ключове слово `def` перед ідентифікатором функції (її ім’ям), потім вказати пару дужок всередині яких можуть міститися імена змінних (аргументи функції), в кінці рядка дві крапки.

Якщо ви хочете створити функцію в Python, ви можете визначити функцію з порожніми круглими дужками, як показано нижче:

```
def print_lyrics():
    """Друкує пісню"""
    print("Ой у лузі червона калина похилилася")
    print("Чогось наша славна Україна зажурилася")
```

У цьому прикладі `my_function` - це функція без аргументів. Вона виводить просте повідомлення у терміналі. При виклику функції `my_function()` ви побачите вивід.

▼ Функція з аргументами

Щоб створити функцію заргументами (параметрами) просто вкажіть в дужках після назви функції всі необхідні аргументи, розділивши їх комами.

Параметр (аргумент) – це змінна, яка отримує конкретне значення під час звернення до функції. Параметри вказувати не обов'язково, але при цьому круглі дужки опускати не можна.

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    return f"My {animal_type}'s name is {pet_name.title()}."
```

Також в разі необхідності для аргументів можна встановити значення за замовченням:

```
def greet(name, greeting="Привіт"):
    """
    Функція виводить привітання для заданого імені.

    :param name: Ім'я для привітання
    :param greeting: Привітання (за замовчуванням "Привіт")
    """
    print(f"{greeting}, {name}!")

# Виклик функції з аргументами
```

```
greet("Юрій")  # Виведе: Привіт, Юрій!
greet("Оксана", "Доброго дня")  # Виведе: Доброго дня, Оксана!
```

▼ `*args` і `**kwargs`

В Python ви можете використовувати `*args` і `**kwargs` для створення функцій з довільною кількістю аргументів.

1. `args` використовується для передачі довільної кількості позиційних аргументів:

```
def print_args(*args):
    for arg in args:
        print(arg)

# Приклад виклику функції
print_args(1, "hello", 3.14, [1, 2, 3])
```

2. `**kwargs` використовується для передачі довільної кількості іменованих аргументів (ключ-значення):

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

# Приклад виклику функції
print_kwargs(name="John", age=25, city="New York")
```

3. Комбінація `*args` і `**kwargs` в тілі функції дозволяє приймати будь-яку кількість позиційних та іменованих аргументів:

```
def print_args_and_kwargs(*args, **kwargs):
    for arg in args:
        print(arg)

    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Приклад виклику функції
print_args_and_kwargs(1, "hello", 3.14, name="John", age=25)
```

▼ Позиційні та ключові параметри

У Python параметри функції можуть бути передані як позиційні (за замовчуванням) або як ключові.

1. **Позиційні параметри:**

   Позиційні параметри передаються у порядку, в якому вони оголошені у визначенні функції.

```
def add_numbers(a, b):
    return a + b

result = add_numbers(2, 3)
print(result)  # 5
```

- **Ключові параметри:**

  Ключові параметри передаються з вказанням імені параметра. Це дозволяє пропустити певні параметри або змнювати порядок передачі параметрів.

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

greet("Alice", "Hello")  # Hello, Alice!
greet(greeting="Good morning", name="Bob")  # Good morning, Bob!
```

- **Комбінація позиційних та ключових параметрів:**

  Ви можете комбінувати позиційні та ключові параметри, але позиційні повинні йти перед ключовими.

```
def describe_person(name, age, country="Unknown"):
    print(f"{name} is {age} years old and is from {country}.")

describe_person("Alice", 30)  # Alice is 30 years old and is from Unknown.
describe_person("Bob", 25, country="USA")  # Bob is 25 years old and is from USA.
```

  Важливо зауважити, що всі позиційні параметри повинні бути передані перед ключовими, інакше ви отримаєте помилку. Наприклад:

```
# Помилка: SyntaxError: positional argument follows keyword argument
describe_person("Charlie", country="Canada", age=28)
```

  Зверніть увагу, що визначення функції також може використовувати `*args` і `**kwargs` для прийняття довільної кількості позиційних та іменованих аргументів.

▼ Лямбда-функції

Лямбда-функції (також відомі як анонімні функції) в Python - це короткі функції, які можна визначити за допомогою ключового слова `lambda`. Вони часто використовуються для визначення малих функцій на льоту, де визначення повноцінної функції не є необхідним.

Синтаксис лямбда-функції виглядає наступним чином:

```
lambda arguments: expression
```

Приклад з одним аргументом:

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

Приклад з двома або більше аргументами:

```
add = lambda x, y: x + y
print(add(3, 4))  # Виведе: 7
```

Лямбда-функції корисні для коротких операцій, де повна функція з визначенням може виглядати зайвою. Однак, слід використовувати їх обережно, оскільки **вони можуть зменшити читабельність коду**, якщо їх використання стає занадто складним або непрозорим.