

ПРОГРАМУВАННЯ (PYTHON)



ПРОЦЕДУРНЕ ПРОГРАМУВАННЯ (ФУНКЦІЇ)

Поняття функції та її призначення

Функція - універсальний засіб групування інструкцій з метою їх подальшого багаторазового використання

- запобігання надлишковості коду
- зменшення трудомісткості програмування
- процедурна декомпозиція

функція ~ процедура ~ метод ~ підпрограма

Процедурне програмування

```
x = exact_value(arguments)
y = calculated_value(arguments)
display_relative_error(x,y)
```

Ми вже неодноразово використовували функції

`print()`

`len()`

`open()`

Тепер навчимося створювати їх

Для цього достатньо двох нових інструкцій: `def` і `return`

Створення функції: інструкція def

```
def <ім'я_функції>(<ім'яАргумента1>, <ім'яАргумента2>, ...):  
    # тіло функції  
    <інструкція 1>  
    <інструкція 2>  
    ...
```

Виклик функції

Тіло функції починає виконуватися в момент виклику функції, синтаксис якого:

<ім'я_функції>(<значенняАргумента1>, <значенняАргумента2>, ...)

Викликати функцію можна з будь-якого місця програми (у тому числі й з тіла іншої або навіть тієї ж самої функції), але після інструкції **def**, якою ця функція була створена.

return

Якщо функція повертає якісь значення, то вони повинні вказуватись в інструкції **return** у тілі функції.

Якщо функція виконує певні операції над об'єктами, але не повертає ніяких значень (в інших мовах програмування такі функції зазвичай називаються процедурами), то інструкція **return** може бути відсутня.

```
def average_value(listOfNumbers):  
    """Повертає середнє арифметичне чисел у списку"""  
    return sum(listOfNumbers) / len(listOfNumbers)
```

```
ages = [25, 30, 35]
```

```
averageAge = average_value(ages)
```

```
print(averageAge) # 30.0
```

```
print(average_value([25, 30, 35])) # 30.0
```


Функція може бути без аргументів

```
def birthday_greeting():  
    print('Вітаю з днем народження!')
```

```
birthday_greeting()
```

А якщо запишемо у змінну?

```
def birthday_greeting():  
    print('Вітаю з днем народження!')  
greet = birthday_greeting()           # вітання  
  
print(greet)                           # None  
  
print(birthday_greeting())             # вітання і None
```

return None

```
def birthday_greeting():  
    print('Вітаю з днем народження!')  
    return None    # додається прихована інструкція
```

Але!!!

```
def birthday_greeting():  
    print('Вітаю з днем народження!')
```

```
greet = birthday_greeting      # ще одне ім'я для функції
```

```
print(greet) # <function birthday_greeting at 0x7f35a61d9ea0>
```

```
greet() #  
вітання
```

Результат виконання функцій з відсутньою інструкцією **return** не має змісту записувати у змінну — такі функції зазвичай викликаються безпосередньо.

```
def interchange(L, i, j):  
    L[i], L[j] = L[j], L[i]
```

```
myList = [1, 2, 3, 4, 5]  
interchange(myList, 0, 1)  
print(myList)
```

```
# [2, 1, 3, 4, 5]
```

“Мертвий” код

Інструкція **return** може міститися у будь-якому місці тіла функції. Також у тілі функції можуть міститися кілька інструкцій **return**. Як тільки виконається одна з них, функція припиняє роботу і повертає результат програмі, яка її викликала.

```
def is_empty_list(someList):  
    if someList:  
        return False  
    return True                # else не потрібне!
```

Повертання більше ніж одного значення

```
def modul(k):  
    """Розв'язок рівняння  $|x| = k$ """  
    if k < 0:  
        return None  
    if k == 0:  
        return 0  
    return -k, k  
  
solution = modul(5)  
print(solution)           # (-5, 5)  
print(solution[0])        # -5
```

Рекурсивні функції

```
def list_sum(L):  
    if not L:  
        return 0  
    return L[0] + list_sum(L[1:])
```


Анотування функцій

Анотації функцій документують типи аргументів і тип значення, яке повертає функція. Анотування здійснюється у заголовку функції. Типи аргумента вказується через двокрапку після імені цього аргумента, а тип значення, яке повертається, після символів “->” в кінці заголовку функції, але перед двокрапкою:

```
def count_of_digits(s:str) -> int:
    """Обчислює кількість цифр у рядку"""
    n = 0
    for symbol in s:
        if symbol.isdigit():
            n += 1
    return n
```

```
print(count_of_digits('У 2018 р. населення України складало 40 млн.')) # 6
```

Анотація наведеної вище функції інформує про те, що їй передається рядок (тип str), а вона повертає ціле число (тип int).

Анотування функцій необов'язкове. Анотації описують особливості функції, але не впливають на виконання, так як інтерпретатор не перевіряє відповідність типів. Основне призначення анотацій — полегшити користувачам розуміння написаного коду.

Лямбда-функції

Лямбда-функції — це невеликі функції, які можуть повертати значення лише одного виразу і оголошуються ключовим словом `lambda`. Загальний синтаксис лямбда-функції:

`lambda аргумент_1, аргумент_2, ..., аргумент_n: вираз`

Наприклад, лямбда-функцію для обчислення суми, яка буде на рахунку через `years` років при початковому внеску `deposit` та `pc` складних річних відсотках, можна оголосити та викликати так:

```
money = lambda deposit, years, pc: deposit * (1 + pc/100)**years  
my_money = money(1000,2,10)
```

Декомпозиція

Зазвичай функції використовуються для декомпозиції задачі. Тобто, задача розбивається на складові частини (підзадачі), для кожної з яких створюється своя функція.

Часто функції описуються в окремому файлі (модулі) і імпортуються в основну програму. Це дає змогу багаторазово використовувати функції в різних програмах, імпортуючи їх з потрібного модуля.

ОБЛАСТІ ВИДИМОСТІ.

РЕЖИМИ ПЕРЕДАВАННЯ АРГУМЕНТІВ

Локальні та глобальні змінні

Кожне ім'я (змінної, функції тощо) має свою **область видимості** — місце у програмному коді, де цьому імені було присвоєно певне значення.

Змінні, яким були присвоєні значення всередині функції, включаючи аргументи функції, з'являються в момент виклику функції і зникають, коли функція завершує роботу і передає керування програмі, яка її викликала. Через цю свою особливість вони називаються **локальними** змінними.

Змінні, яким присвоєні значення поза усіма інструкціями **def**, називаються **глобальними**. Глобальна область видимості охоплює єдиний файл (модуль), у якому збережено код. Якщо потрібно використовувати імена з іншого модуля, його потрібно явно імпортувати.

Області видимості

Вбудовані імена

Модуль

Охоплююча функція

Функція

```
x = 99
```

```
y = 99
```

```
def func(y):
```

```
    y += 1
```

```
    z = x + y
```

```
    return z
```

```
print(func(0))
```

```
print(y)
```

Незмінюваний тип

```
def add_n(a, n):  
    a += n
```

```
x = 5  
add_n(x, 1)  
print(x)      # 5
```

```
x = 5  
a = x; n = 1  
a = a + n
```

Змінюваний тип

```
def add_to_list(lst, item):  
    lst.append(item)
```

```
L = [1,2,3,4,5]  
add_to_list(L, 6)  
print(L)      # [1,2,3,4,5,6]
```

```
L = [1,2,3,4,5]  
lst = L; item = 6  
lst.append(item)
```

Інструкція **global**

Щоб дати змогу функції змінювати глобальну змінну незмінюваного типу, ім'я цієї змінної оголошується всередині функції інструкцією **global**:

```
name = 'Роман'
```

```
def rename (newName) :  
    global name  
    name = newName
```


Позиційні аргументи

```
from time import sleep
```

```
def timer(seconds, step):  
    for second in range(seconds, 0, -step):  
        print(second, end='-')  
        sleep(step)  
    print('stop')
```

```
timer(10,1)    # 10-9-8-7-6-5-4-3-2-1-stop
```

```
timer(10)
```

```
# TypeError: timer() missing 1 required positional argument: 'step'
```

```
timer(10,1,2)
```

```
# TypeError: timer() takes 2 positional arguments but 3 were given
```

Іменовані аргументи

Іменовані аргументи дають змогу співставити аргументи за їхніми іменами. Порядок аргументів під час виклику функції у цьому випадку не має значення:

```
def timer(seconds, step):
```

```
    ...
```

```
timer(step=2, seconds=10)           # 10-8-6-4-2-stop
```

Можна змішувати позиційні та іменовані аргументи. Тоді спочатку будуть співставлені зліва направо всі позиційні, а потім іменовані.

Значення за замовчуванням

Значення аргументів за замовчуванням використовується, якщо під час виклику функції цим аргументам не передаватиметься конкретне значення:

```
from time import sleep
```

```
def timer(seconds, step=1):  
    for second in range(seconds, 0, -step):  
        print(second, end=' - ')  
        sleep(step)  
    print('stop')
```

```
timer(10)                                # 10-9-8-7-6-5-4-3-2-1-stop
```

```
timer(10, 2)                             # 10-8-6-4-2-stop
```

Передавання довільної кількості аргументів

Функція може приймати довільну кількість аргументів за допомогою параметрів ***args** і ****kwargs**.

***args** збирає додаткові позиційні аргументи в **кортеж args**,

****kwargs** збирає додаткові іменовані аргументи в **словник kwargs**.

Насправді імена args і kwargs просто домовленість, вони можуть бути іншими. Синтаксис визначається лише “зірочками”.

```
def print_friends(person, *friends):  
    """Виводить на екран друзів особи person"""  
    s = f"{person}'s friends:\n"  
    for friend in friends:  
        s += f"{friend}, "  
    print(s[:-2] + '.')
```

```
print_friends('Alice', 'Bob', 'Eve', 'John')
```

```
alice_friends = ['Bob', 'Eve', 'John']  
print_friends('Alice', *alice_friends)
```

Alice's friends:
Bob, Eve, John.

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

```
print_info(name = 'Alice', age = 20)
```

```
name: Alice
```

```
age: 20
```

Оскільки параметри ***args** і ****kwargs** працюють з різними типами аргументів (позиційними та іменованими відповідно), то їх можна використовувати одночасно в межах однієї функції.

Розпаковування аргументів

Якщо * або ** розмістити перед ітерабельним об'єктом під час виклику функції, то елементи цього об'єкту будуть розпаковані та передані функції як позиційні (іменовані) аргументи:

```
def person_info(name, year):  
    print(f'My name is {name}. I was born in {year}.')
```

```
my_info = ('Roman', 1976)  
person_info(*my_info)
```

```
my_info = {'year': 1976, 'name': 'Roman'}  
person_info(**my_info)
```


ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ

Оперування функціями як об'єктами

Функції нічим не відрізняються від інших об'єктів Python — вони мають атрибути, їх можна присвоювати змінним, зберігати в структурах даних, передавати аргументами, повертати як результат виконання інших функцій, оголошувати всередині інших функцій та ін. Для демонстрації сказаного оголосимо функцію `new_price`, яка застосовує до старої ціни `old_price` знижку `discount`:

```
def new_price(old_price, discount):  
    return old_price * (1 - discount/100)
```

Присвоїмо ім'я функції новій змінній:

```
new_cost = new_price
```

Тепер наступні два рядки коду даватимуть ідентичні результати:

```
print(new_price(50,5))      # 47.5  
print(new_cost(50,5))       # 47.5
```

При цьому другий виклик працюватиме навіть після вилучення першого імені функції, оскільки об'єкт функції та її ім'я не одне й те саме:

```
del new_price  
print(new_price(50,5))      # NameError: name 'new_price' is not defined  
print(new_cost(50,5))       # 47.5
```

Функція може бути аргументом іншої функції⁸⁴, наприклад:

```
def new_price(func, price, percent):  
    return func(price, percent)  
  
print(new_price(new_cost,50,5))      # 47.5
```

```
def apply_discount(percent):  
    def discount(price):  
        return price * (1 - percent/100)  
    return discount
```

По-перше, у ньому всередині функції `apply_discount` оголошено іншу — `discount`. По-друге, результатом виконання функції `apply_discount` є не якесь значення, а функція. Це дає змогу створити функції для конкретної (у відсотках) знижки і надалі передавати їм аргументом початкову ціну:

```
discount_5_percent = apply_discount(5)  
discount_10_percent = apply_discount(10)  
print(discount_5_percent(50))      # 47.5  
print(discount_10_percent(50))     # 45.0
```

Оскільки ім'я вкладеної функції `discount` ніде не фігурує, то функцію `apply_discount` можна переписати простіше з використанням анонімної лямбда-функції:

```
def apply_discount(percent):  
    return lambda price: price * (1 - percent/100)
```

Фактично функція `apply_discount` слугує фабрикою для створення інших функцій. Вони можуть бути збережені, наприклад, у списку, і викликатися у циклі:

```
discounts = [discount_5_percent, discount_10_percent]  
for func in discounts:  
    print(func(50))      # 47.5  
                        # 45.0
```

Функції як об'єкти мають атрибути⁸⁵. Крім наперед визначених атрибутів, можна приєднувати і свої. Оскільки атрибути пов'язані з об'єктами, а не з областями видимості, то, хоча вони є локальними відносно функції, проте зберігають свої значення після виходу з неї.

```
from time import time

def f(x):
    print(x)
    f.counter += 1
    f.logfile.write(f'{x}: {time()}\n')

f.counter = 0
f.logfile = open('log.txt', 'w')
for i in range(10):
    f(i)

print(' ', f.counter)
f.logfile.close()
```

```
old_list = ['1', '2', '3', '4']  
new_list = list(map(int, old_list))  
print (new_list) # [1, 2, 3, 4]
```

```
a = [1,2,3]; b = "xyz"; c = (None, True)  
res = list(zip(a, b, c))  
print (res) # [(1, 'x', None), (2, 'y', True)]
```

```
words = ['www', 'office', 'game', 'ink']  
res = list(filter(lambda x: len(x)==3, words))  
print(res) # ['www', 'ink']
```