



ОРГАНІЗАЦІЯ PYTHON-КОДУ ЗА ДОПОМОГОЮ ФУНКЦІЙ

Питання 7.2

Функції в мовах C та Python

C

- Приймає будь-яку кількість параметрів та повертає один будь-який результат.


The diagram shows the C function signature `float findArea(float length, float width);` with arrows pointing to its components: `float` is the return type; `findArea` is the function name; `float` and `length` are parameter type and parameter name respectively; `float` and `width` are parameter type and parameter name respectively; and `;` is the terminating semicolon.

Python

- Приймає будь-яку кількість параметрів та повертає будь-яку кількість будь-яких результатів.
 - Для визначення функції потрібно написати `def`, назву функції, вхідні параметри в дужках та двокрапку (`:`)
 - Приклад найпростішої функції:
 - `def do_nothing():`
 `pass`
 - Функцію можна викликати, написавши її назву та дужки:
 - `>>> do_nothing()`

Аргументи та параметри функцій

- Значення, що передаються в функцію при виклику, називаються *аргументами*.
 - Їх значення копіюються у відповідні *параметри* всередині функцій.



```
>>> def echo(anything):  
...     return anything + ' ' + anything  
...
```

Параметр

Рядок 'Rumplestiltskin' копіюється всередині функції echo() в параметр anything, а потім повертається на викликаючу сторону.



```
>>> echo('Rumplestiltskin')  
'Rumplestiltskin Rumplestiltskin'
```

Аргумент

- Функція, яка не має параметрів, проте повертає значення:

```
>>> def agree():  
...     return True  
...
```

Функції в мовах C та Python

C

- Прототип:
 - `void show_n_char(char ch, int num)`
 - Змінні `ch` та `num` називають *формальними параметрами*.
 - формальні параметри є локальними змінними для функції.
 - форма ANSI C вимагає, щоб **кожній змінній** передувала її тип.
- Функції повинні оголошуватись із вказуванням типів.
 - Тип значення, яке функція повертає, повинен співпадати з оголошеним типом повернення (return type).
 - Функція без типу повернення повинна оголошуватись з типом `void`.

Python

- Функція може приймати будь-яку кількість аргументів (включаючи 0) будь-якого типу.
- Вона може повертати довільну кількість результатів (також включаючи 0) будь-якого типу.
- Якщо функція не викликає `return` явно, викликаючий код отримає результат `None`.

```
>>> print(do_nothing())  
None
```

Значення None

- None — це спеціальне значення в Python, яке заповнює порожнє місце, якщо функція нічого не повертає.
 - Воно не є булевим значенням False, хоч і схоже при перевірці булевої змінної.

```
>>> thing = None
>>> if thing:
...     print("It's some thing")
... else:
...     print("It's no thing")
...
It's no thing
```

```
>>> if thing is None:
...     print("It's nothing")
... else:
...     print("It's something")
...
It's nothing
```

- None потрібен, щоб відрізнити порожнє значення від відсутнього.

Передача параметрів у функцію

```
def main():
    n = 9001
    print(f"Initial address of n: {id(n)}")
    increment(n)
    print(f"    Final address of n: {id(n)}")

def increment(x):
    print(f"Initial address of x: {id(x)}")
    x += 1
    print(f"    Final address of x: {id(x)}")

main()
```

```
= RESTART: F:/GDisk/[Коледж]/[Основи програмування та алгоритмі]
Initial address of n: 2390124837296
Initial address of x: 2390124837296
    Final address of x: 2390124837392
    Final address of n: 2390124837296
```

- Вбудована функція `id()` повертає ціле число, що представляє адресу пам'яті, за якою розміщується шуканий об'єкт. За допомогою цього можна перевірити наступне:
 - Аргументи функції спочатку посилаються на ту ж адресу, що і відповідні оригінальні змінні.
 - Переприсвоєння значення аргумента всередині функції надає аргументу нову адресу в той час, як початкова змінна залишається за старою адресою незмінною.
- Оскільки початкові адреси `n` та `x` залишаються тими ж при виклику `increment()`, аргумент `x` передається НЕ за значенням.
 - Інакше, `n` та `x` мали б різні адреси пам'яті.



Передача параметрів у функцію

```
def main():
    counter = 0
    print(greet("Alice", counter))
    print(f"Counter is {counter}")
    print(greet("Bob", counter))
    print(f"Counter is {counter}")

def greet(name, counter):
    counter += 1
    return f"Hi, {name}!"

main()
```

```
>>>
===== RESTART: F:/GDisk/[Колект]/[Осно]
Hi, Alice!
Counter is 0
Hi, Bob!
Counter is 0
```

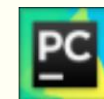
- counter не інкрементується, оскільки Python не може передавати значення за посиланням.
 - Допоможе повернення кількох значень.

```
def main():
    counter = 0
    greeting, counter = greet("Alice", counter)
    print(f"{greeting}\nCounter is {counter}")
    greeting, counter = greet("Bob", counter)
    print(f"{greeting}\nCounter is {counter}")

def greet(name, counter):
    return (f"Hi, {name}!", counter + 1)

main()
```

```
===== RESTART: F:/GDisk/[Колект]
Hi, Alice!
Counter is 1
Hi, Bob!
Counter is 2
```



pass_by_ref.py

Передача параметрів у функцію

- Python передає аргументи за присвоєнням: при виклику функції кожний аргумент стає змінною, якій буде присвоєне передане значення.
 - Усі Python-об'єкти реалізуються з конкретною структурою, зокрема, з лічильником посилань (reference counter), який визначає, скільки посилань (імен) прив'язано до об'єкта.
- Розглянемо інструкцію **x = 2**:
 - Якщо об'єкт, що представляє значення 2 вже існує, він отримується. Інакше – створюється.
 - Лічильник посилань на цей об'єкт збільшується на 1.
 - У поточний простір імен додається «запис» (entry, пара «ключ-значення» словника), що пов'язує ідентифікатор x з об'єктом «2».
 - Словник повертається функціями locals() або globals().
- Якщо присвоїти інше значення для **x**:
 - Лічильник посилань на об'єкт «2» зменшується на 1.
 - Лічильник посилань на об'єкт, що представляє нове значення, збільшується на 1.
 - Словник для поточного простору імен оновлюється, пов'язуючи x із об'єктом, що представляє нове значення.

Передача параметрів у функцію

```
from sys import getrefcount

print("--- Before assignment ---")
print(f"References to value_1: {getrefcount('value_1')}")
print(f"References to value_2: {getrefcount('value_2')}")
x = "value_1"
print("--- After assignment ---")
print(f"References to value_1: {getrefcount('value_1')}")
print(f"References to value_2: {getrefcount('value_2')}")
x = "value_2"
print("--- After reassignment ---")
print(f"References to value_1: {getrefcount('value_1')}")
print(f"References to value_2: {getrefcount('value_2')}")
```

```
>>>
===== RESTART: F:/GDisk/[Коледж]/[Осно:
--- Before assignment ---
References to value_1: 2
References to value_2: 2
--- After assignment ---
References to value_1: 3
References to value_2: 2
--- After reassignment ---
References to value_1: 2
References to value_2: 3
```

- При присвоєнні багатьом змінним одного значення Python інкрементує лічильник посилань існуючого об'єкта та оновлює поточний простір імен замість створення дублікатів у пам'яті.
 - Аргументи функцій у Python є локальними змінними:
 - Скрипт демонструє збільшення кількості посилань не на 1, а на 2, що пояснюється викликом ще й sys.getrefcount()

```
>>>
===== RESTART: F:/GDisk/[Коледж]/[Основи п
{'my_arg': 'arg_value', 'my_local': True}
2
4
```

```
from sys import getrefcount

def show_refcount(my_arg):
    return getrefcount(my_arg)

def show_locals(my_arg):
    my_local = True
    print(locals())

show_locals("arg_value")
print(getrefcount("my_value"))
print(show_refcount("my_value"))
```

Значення None

- Цілочисельні нулі, нулі з плаваючою крапкою, порожні рядки ("), списки ([]), кортежі ((,)), словники ({}), і множини (set()) рівні False, але не None.

```
>>> def is_none(thing):  
...     if thing is None:  
...         print("It's None")  
...     elif thing:  
...         print("It's True")  
...     else:  
...         print("It's False")  
... 
```

```
>>> is_none(None)  
It's None  
>>> is_none(True)  
It's True  
>>> is_none(False)  
It's False  
>>> is_none(0)  
It's False  
>>> is_none(0.0)  
It's False  
>>> is_none(())  
It's False  
>>> is_none([])  
It's False  
>>> is_none({})  
It's False  
>>> is_none(set())  
It's False
```

Позиційні аргументи

- Аргументи, чиї значення копіюються у відповідні параметри відповідно до порядку слідування.

```
>>> def menu(wine, entree, dessert):  
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}  
...  
>>> menu('chardonnay', 'chicken', 'cake')  
{'dessert': 'cake', 'wine': 'chardonnay', 'entree': 'chicken'}
```

- Недолік: слід запам'ятовувати значення кожної позиції.
 - Якби викликали функцію menu(), передавши останнім аргументом марку вина, обід вийшов би геть іншим:

```
>>> menu('beef', 'bagel', 'bordeaux')  
{'dessert': 'bordeaux', 'wine': 'beef', 'entree': 'bagel'}
```

Іменовані аргументи (keyword arguments)

- Для уникнення плутанини можна вказати аргументи за допомогою імен відповідних параметрів.
 - Порядок слідування може бути іншим:

```
>>> menu(entree='beef', dessert='bagel', wine='bordeaux')  
{'dessert': 'bagel', 'wine': 'bordeaux', 'entree': 'beef'}
```

- Можна об'єднувати позиційні та іменовані аргументи, проте позиційні аргументи необхідно вказувати першими.

```
>>> menu('frontenac', dessert='flan', entree='fish')  
{'entree': 'fish', 'dessert': 'flan', 'wine': 'frontenac'}
```

Значення параметру за умовчанням

- Використовуються, якщо викликаюча сторона не надала відповідний аргумент.

```
>>> def menu(wine, entree, dessert='pudding'):  
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

- Викличемо функцію menu(), не передавши їй аргумент dessert:

```
>>> menu('chardonnay', 'chicken')  
{'dessert': 'pudding', 'wine': 'chardonnay', 'entree': 'chicken'}
```

- Надавши аргумент, він замінить значення за умовчанням:

```
>>> menu('dunkelfelder', 'duck', 'doughnut')  
{'dessert': 'doughnut', 'wine': 'dunkelfelder', 'entree': 'duck'}
```

Приклад: функція buggy()

- Очікується, що функція буде кожен раз запускатись з новим порожнім списком `result`, додавати в нього аргумент `arg` та виводити список з одного елементу на екран.

```
>>> def buggy(arg, result=[]):  
...     result.append(arg)  
...     print(result)  
...
```

```
>>> buggy('a')  
['a']  
>>> buggy('b')  
['a', 'b']
```

- Проте є баг: при першому виклику список порожній, а далі - ні. Варіанти виправлень:

```
>>> def works(arg):  
...     result = []  
...     result.append(arg)  
...     return result  
...  
>>> works('a')  
['a']  
>>> works('b')  
['b']
```

```
>>> def nonbuggy(arg, result=None):  
...     if result is None:  
...         result = []  
...     result.append(arg)  
...     print(result)  
...  
>>> nonbuggy('a')  
['a']  
>>> nonbuggy('b')  
['b']
```

Отримання позиційних аргументів (оператор *)

- Символ * всередині функції з параметром дозволяє згрупувати довільну кількість позиційних аргументів у кортеж.

- У прикладі args – кортеж параметрів, створений з аргументів, переданих у функцію print_args():

```
>>> def print_args(*args):  
...     print('Positional argument tuple:', args)  
...
```

- При виклику без аргументів буде порожній кортеж:

```
>>> print_args()  
Positional argument tuple: ()
```

- Всі передані аргументи виводяться на екран як кортеж args:

```
>>> print_args(3, 2, 1, 'wait!', 'uh...')  
Positional argument tuple: (3, 2, 1, 'wait!', 'uh...')
```

Отримання позиційних аргументів (оператор *)

- Якщо у функції є обов'язкові позиційні аргументи, *args відправиться в кінець списку та отримає решту аргументів:

```
>>> def print_more(required1, required2, *args):  
...     print('Need this one:', required1)  
...     print('Need this one too:', required2)  
...     print('All the rest:', args)  
...
```

```
>>> print_more('cap', 'gloves', 'scarf', 'monocle', 'mustache wax')  
Need this one: cap  
Need this one too: gloves  
All the rest: ('scarf', 'monocle', 'mustache wax')
```


Отримання іменованих аргументів за допомогою оператора **

- Елементи групуються в словник, де назви параметрів стають ключами, а передані значення – відповідними значеннями у словнику.

```
>>> def print_kwargs(**kwargs):  
...     print('Keyword arguments:', kwargs)  
...
```

- Викличемо функцію:

```
>>> print_kwargs(wine='merlot', entree='mutton', dessert='macaroon')  
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot', 'entree': 'mutton'}
```

- kwargs є словником.
- Якщо використовуються позиційні та іменовані аргументи (*args і **kwargs), вони повинні слідувати в цьому ж порядку.
- Як і для args, не обов'язково називати цей словник kwargs, проте це розповсюджена практика.

Внутрішні функції

- Можна визначити одну функцію всередині іншої:

```
>>> def outer(a, b):  
...     def inner(c, d):  
...         return c + d  
...     return inner(a, b)  
...  
>>>  
>>> outer(4, 7)  
11
```

- Внутрішні функції корисні при виконанні деяких складних задач понад 1 раз всередині іншої функції.
 - Дозволяє уникати дублювання коду.

Приклад роботи з рядком

- Внутрішня функція додає текст до свого аргументу:

```
>>> def knights(saying):  
...     def inner(quote):  
...         return "We are the knights who say: '%s'" % quote  
...     return inner(saying)  
...  
>>> knights('Ni!')  
"We are the knights who say: 'Ni!'"
```

Рекурсія vs ітерація

Recursion

```
>>> def countdown (n):  
...     if n <= 0:  
...         print 'Blastoff!'  
...     else:  
...         print n  
...         countdown (n - 1)  
>>>  
>>> countdown (3)  
3  
2  
1  
Blastoff!  
>>>
```

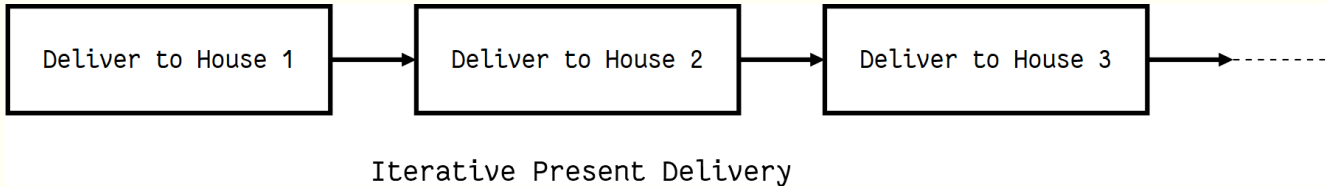
For Loop

```
>>> def countdown (n):  
...     for i in range (n, -1, -1):  
...         if i <= 0:  
...             print "Blastoff!"  
...         else:  
...             print i  
...  
>>> countdown (3)  
3  
2  
1  
Blastoff!  
>>>
```

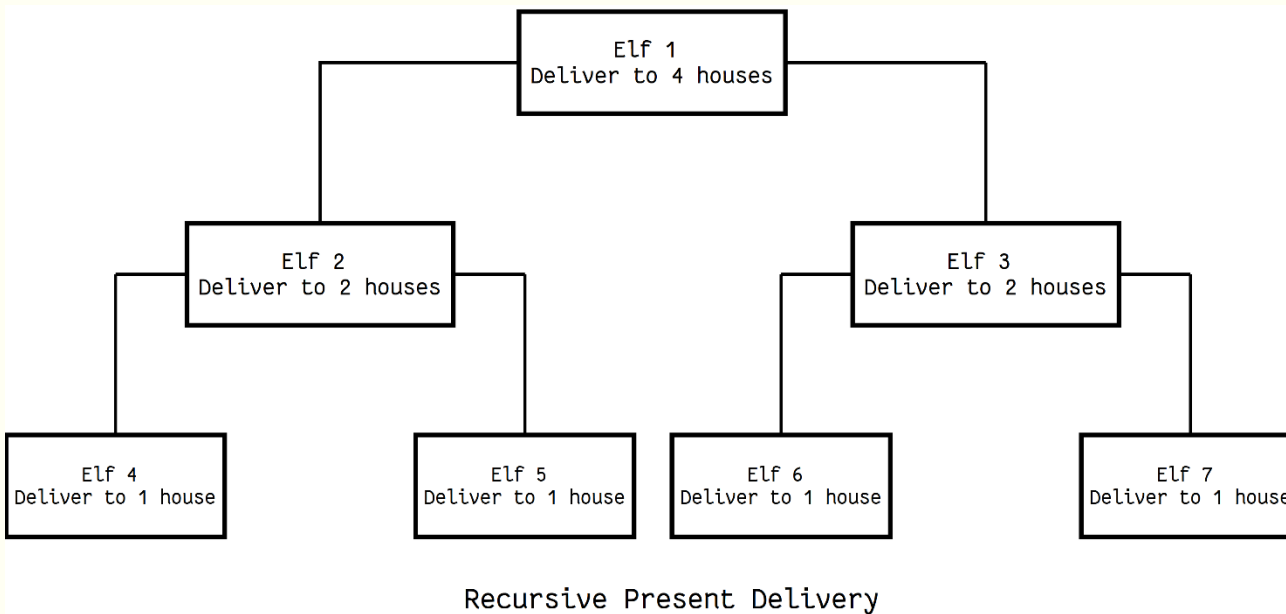
While Loop

```
>>> def countdown (n):  
...     while n > 0:  
...         print n  
...         n = n - 1  
...         print "Blastoff!"  
>>>  
>>> countdown (3)  
3  
2  
1  
Blastoff!  
>>>
```

Ітераційне та рекурсивне розбиття задачі



```
houses = ["Eric's house", "Kenny's house", "Kyle's house", "Stan's house"]  
def deliver_presents_iteratively():  
    for house in houses:  
        print("Delivering presents to", house)
```



```
houses = ["Eric's house", "Kenny's house", "Kyle's house", "Stan's house"]  
  
# Each function call represents an elf doing his work  
def deliver_presents_recursively(houses):  
    # Worker elf doing his work  
    if len(houses) == 1:  
        house = houses[0]  
        print("Delivering presents to", house)  
  
    # Manager elf doing his work  
    else:  
        mid = len(houses) // 2  
        first_half = houses[:mid]  
        second_half = houses[mid:]  
  
        # Divides his work among two elves  
        deliver_presents_recursively(first_half)  
        deliver_presents_recursively(second_half)
```

Характеризуємо рекурсію

- Переваги рекурсії
 - Рекурсивні функції можуть зробити код чистим та елегантним.
 - Складна задача розбивається на простіші підзадачі.
 - Генерування послідовностей простіше виконати рекурсивно, ніж ітераційно.
- Недоліки рекурсії
 - Інколи логіка рекурсії складна для розуміння в конкретних задачах.
 - Рекурсивні виклики затратні (неефективні), оскільки споживають багато пам'яті та часу.
 - Рекурсивні функції важко налагодити.

Обмеження рекурсії в мові Python

- Кількість рекурсивних викликів у мові Python за умовчанням обмежується значенням 1000.
 - Причина – можливе переповнення стеку через відсутність спеціальної оптимізації для хвостової рекурсії (tail recursion).
 - Дане обмеження – глобальне для всіх Python-додатків, його зміна також зачіпає всі додатки.
 - Функції `getrecursionlimit()` та `setrecursionlimit()` з модуля `sys` дозволяють змінювати рекурсивний ліміт.
- Приклад коду:

```
import sys
sys.getrecursionlimit()
sys.setrecursionlimit(1500)
```

```
>>> import sys
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(1500)
```



ДЯКУЮ ЗА УВАГУ!

Наступне питання: стратегії налагодження Python-коду