



ОРГАНІЗАЦІЯ PYTHON-КОДУ ЗА ДОПОМОГОЮ ФУНКЦІЙ

Питання 6.2

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 labels and arrows pointing to its components:


- `float` (return type) is labeled "return type" with an upward arrow.
- `findArea` (function name) is labeled "function name" with a downward arrow.
- `float` (parameter type) is labeled "parameter type" with an upward arrow.
- `length` (parameter name) is labeled "parameter name" with a downward arrow.
- `float` (parameter type) is labeled "parameter type" with an upward arrow.
- `width` (parameter name) is labeled "parameter name" with a downward arrow.
- `;` (terminating semicolon) is labeled "terminating semicolon" with an upward arrow.

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.

# 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'}
```

# Аргументи – ключові слова

---

- 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 аргументи та аргументи — ключові слова.
  - Проте позиційні аргументи необхідно вказувати першими.

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



## Значення параметру за замовчуванням

---

- 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 замовчуванням:

```
>>> 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')
['a', '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

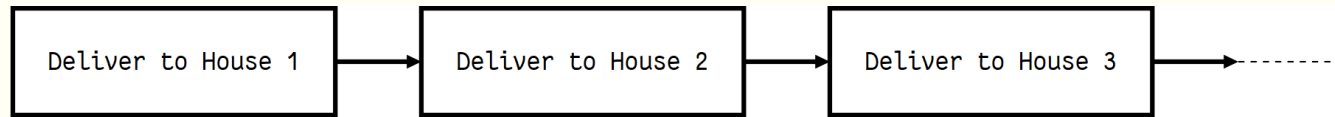
```
>>> def countdown (n):
... for i in range (n, -1, -1):
... if i <= 0:
... print "Blastoff!"
... else:
... print i
...
>>> countdown (3)
3
2
1
Blastoff!
>>>
```

## While Loop

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

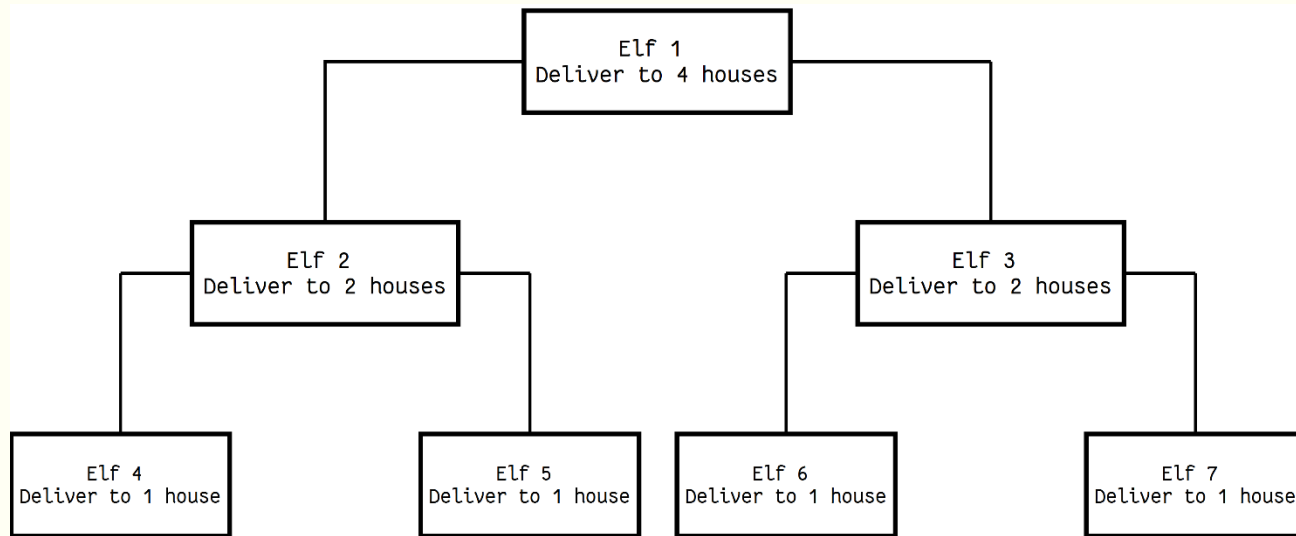


# Ітераційне та рекурсивне розбиття задачі



Iterative Present Delivery

```
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)
```



Recursive Present Delivery

```
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