МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Кафедра “Захист інформації”

Створення та використання функцій

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 7
з курсу «Програмування скриптовими мовами»
для студентів спеціальності
«Кібербезпека»

Затверджено
на засіданні кафедри
"Захист інформації"
протокол № 01 від 29.08.2024 р.

Львів – 2024

Створення та використання функцій: Методичні вказівки до лабораторної роботи № 7 з курсу «Програмування скриптовими мовами» для студентів спеціальності «Кібербезпека» / Укл. *Я. Р. Совин* – Львів: Національний університет "Львівська політехніка", 2024. – 38 с.

Укладач: Я. Р. Совин, канд. техн. наук, доцент

Відповідальний за випуск: В. Б. Дудикевич, д.т.н., професор

Рецензенти: А. Я. Горпенюк, канд. техн. наук, доцент
Ю. Я. Наконечний, канд. техн. наук, доцент

Мета роботи – ознайомитись з можливостями і застосуванням функцій у мові Python**.**

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

Функція – це фрагмент коду, який можна викликати з будь-якого місця програми. Функції у Python відносяться до об’єктів – їх можна присвоювати змінним, зберігати в структурах даних, передавати в якості аргументів іншим функціям і навіть повертати їх у якості значень з інших функцій. У цій лекції ми розглянемо створення користувацьких функцій, які дозволять зменшити надмірність програмного коду і підвищити його структурованість.

1.1. Визначення функції та її виклик

Функція створюється (визначається) за допомогою ключового слова *def* в наступному форматі:

```
def <Ім'я функції> ([<Параметр1>[,...,<ПараметрN>]]):
    [""" Рядок документування """]
    <Тіло функції>
    [return <Результат>]
```

Ім’я функції має бути унікальним ідентифікатором, що складається з латинських букв, цифр і знаків підкреслення, причому ім’я формальної процедури не може починатися з цифри. В якості імені функції не можна використовувати ключові слова, крім того, слід уникати збігів з назвами вбудованих ідентифікаторів. Регістр літер в імені функції також має значення.

Після імені функції в круглих дужках через кому можна вказати один або кілька параметрів, а якщо функція не приймає параметри, вказуються тільки круглі дужки. Після круглих дужок ставиться двокрапка.

Тіло функції являє собою вкладену конструкцію. Як і в будь-якої вкладеної конструкції, інструкції всередині функції виділяються однаковою кількістю пробілів зліва. Кінцем функції вважається інструкція, перед якою знаходиться менша кількість пробілів. Якщо тіло функції не містить інструкцій, то всередині неї необхідно розмістити оператор *pass*, який не виконує ніяких дій. Цей оператор зручно використовувати на етапі розробки програми, коли ми визначили функцію, а тіло вирішили дописати пізніше. Ось приклад функції, яка нічого не робить:

```
def func ():
    pass
```

Необов’язкова інструкція *return* дозволяє повернути з функції будь-яке значення в якості результату. Після виконання цієї інструкції виконання функції буде зупинено, і наступні інструкції ніколи не будуть виконані:

```
def func():
    print("Текст до інструкції return")
    return "Значення, що повертається"
    print("Ця інструкція ніколи не буде виконана")

print(func()) # Викликаємо функцію
```

Результат виконання:

```
Текст до інструкції return
Значення, що повертається
```

Інструкції *return* може не бути взагалі. У цьому випадку виконуються всі інструкції всередині функції, і як результат повертається значення *None*.

Для прикладу створимо три функції:

```
def print_ok():
    """Приклад функції без параметрів"""
    print("Повідомлення при вдало виконаної операції")

def echo(m):
    """Приклад функції з параметром"""
    print(m)

def summa(x, y):
    """Приклад функції з параметрами,
    що повертає суму двох змінних"""
    return x + y
```

При виклику функції значення її параметрів вказуються всередині круглих дужок через кому. Якщо функція не приймає параметрів, залишаються тільки круглі дужки. Необхідно також зауважити, що кількість параметрів у визначенні функції має збігатися з кількістю параметрів при виклику, інакше буде виведено повідомлення про помилку.

Викликати визначені вище функції можна наступними способами:

```
print_ok()            # Викликаємо функцію без параметрів
echo("Повідомлення") # Функція виведе повідомлення
x = summa(5, 2)       # Змінній x буде присвоєно значення 7
a, b = 10, 50
y = summa (a, b)      # Змінній y буде, присвоєно значення 60
```

Як видно з останнього прикладу, ім’я змінної у виклику функції може не збігатися з ім’ям відповідного параметра у визначенні функції. Крім того, *глобальні* змінні *x* і *y* не конфліктують з однойменними змінними, створеними у визначенні функції, оскільки вони розташовані в різних областях видимості. Змінні, зазначені у визначенні функції, є локальними і доступні тільки всередині функції. Більш детально області видимості ми розглянемо пізніше.

Оператор +, який використовується в функції *summa()*, служить не тільки для додавання чисел, а й дозволяє об’єднати послідовності. Тобто функція *summa()* може використовуватися не тільки для додавання чисел. Як приклад виконаємо конкатенацію рядків і об’єднання списків:

```
def summa(x, y):
    return x + y

print(summa("str", "ing"))   # Виведе: string
print(summa([1, 2], [3, 4])) # Виведе: [1, 2, 3, 4]
```

Функція може повертати більш ніж одне значення:

```
def f(x):
    return x ** 2, x ** 3, x ** 4

p0, p1, p2 = f(2)
>>> print(p0, p1, p2) # Виведе 4 8 16
```

Оскільки все в мові Python представляє собою об’єкти не є винятком і функції. Інструкція *def* створює об’єкт, що має тип *function*, і зберігає посилання на нього в ідентифікаторі, зазначеному після інструкції *def*. Таким чином ми можемо зберегти посилання на функцію в іншій змінній – для цього назва функції вказується без круглих дужок. Збережемо посилання в змінній і викличемо функцію через неї:

```
def summa(x, y):
    return x + y

f = summa     # Зберігаємо посилання в змінній f
v = f(10, 20) # Викликаємо функцію через змінну f
```

Можна також передати посилання на функцію іншій функції в якості параметра. Функції, що передаються по посиланню, зазвичай називаються *функціями зворотного виклику (callback)*.

```
def summa(x, y):
    return x + y

def func(f, a, b):
    """Через змінну f буде доступне посилання на
    функцію summa()"""
    return f(a, b) # Викликаємо функцію summa()

# Передаємо посилання на функцію в якості параметра
v = func(summa, 10, 20)
```

Ще приклад *callback*-функцій:

```
def someAction(x, y, someCallback):
    return someCallback(x, y)

def calcProduct(x, y):
    return x * y

def calcSum(x, y):
    return x + y

# Виводить 75, добуток 5 та 15
print (someAction(5, 15, calcProduct))

# Виводить 20, суму 5 та 15
print (someAction(5, 15, calcSum))
```

Об’єкти функцій підтримують багато атрибутів, звернутися до яких можна, вказавши атрибут після назви функції через крапку. Наприклад, через атрибут *__name__* можна отримати ім'я функції у вигляді рядка, через атрибут *__doc __* рядок документування і т. д. Для прикладу виведемо назви всіх атрибутів функції за допомогою вбудованої функції *dir()*:

```
def summa(x, y):
    """Підсумовування двох чисел"""
    return x + y

print(dir(summa))
```

Результа:

```
['__annotations__',     '__call__',     '__class__',     '__closure__',
'__code__',  '__defaults__',  '__delattr__',  '__dict__',  '__dir__',
'__doc__',     '__eq__',     '__format__',     '__ge__',     '__get__',
'__getattribute__', '__globals__', '__gt__', '__hash__', '__init__',
'__init_subclass__',     '__kwdefaults__',     '__le__',     '__lt__',
'__module__',   '__name__',   '__ne__',   '__new__',   '__qualname__',
'__reduce__',     '__reduce_ex__',     '__repr__',     '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
>>> summa.__name__
'summa'
>>> summa.__code__.co_varnames
('x', 'y')
>>> summa.__doc__
'Підсумовування двох чисел'
```

1.2. Розташування визначень функцій

Всі інструкції в програмі виконуються послідовно. Це означає, що, перш ніж використовувати в програмі ідентифікатор, його необхідно попередньо визначити, присвоївши йому значення. Тому визначення функції має бути розташоване перед викликом функції.

Правильно:

```
def summa(x, y):
    return x + y

v = summa(10, 20) # Викликаємо після визначення. Все нормально
```

Неправильно:

```
v = summa(10, 20) # Ідентифікатор ще не визначений. Це помилка!!!

def summa(x, y):
    return x + y
```

В останньому випадку буде виведено повідомлення про помилку: *NameError: name 'summa' is not defined*. Щоб уникнути помилки, визначення функції розміщують на самому початку програми після підключення модулів або в окремому модулі.

За допомогою умовного оператора *if* можна змінити порядок виконання програми - наприклад, розмістити всередині умови кілька визначень функцій з однаковою назвою,але різною реалізацією:

```
n = input("Введіть 1 для виклику першої функції:")

if n == "1":
    def echo():
        print("Ви ввели число 1")
else:
    def echo():
        print("Альтернативна функція")

echo() # Викликаємо функцію
input()
```

При введенні числа 1 ми отримаємо повідомлення *"Ви ввели число 1"*, в іншому випадку - *"Альтернативна функція"*.

Пам’ятайте, що інструкція *def* всього лише присвоює посилання на об’єкт функції ідентифікатору, розташованому після ключового слова *def*. Якщо визначення однієї функції зустрічається в програмі кілька разів, буде використовуватися функція, яка була визначена останньою:

```
def echo():
    print("Ви ввели число 1")

def echo():
    print("Альтернативна функція")

echo() # Завжди виводить "Альтернативна функція"
```

1.3. Необов'язкові параметри і зіставлення по ключам

Щоб зробити деякі параметри функції необов’язковими, слід у визначенні функції присвоїти цим параметрам початкове значення. Переробимо функцію сумування двох чисел і зробимо другий параметр необов’язковим:

```
def summa(x, y = 2): # y - необов'язковий параметр
    return x + y

a = summa(5)       # Змінній a буде присвоєно значення 7
b = summa(10, 50) # Змінній b буде присвоєно значення 60
```

Таким чином, якщо другий параметр не заданий, він отримає значення 2. Зверніть увагу на те, що необов’язкові параметри повинні слідувати після обов’язкових, інакше буде виведено повідомлення про помилку.

До сих пір ми використовували *позиційну передачу параметрів* у функцію:

```
def summa(x, y):
    return x + y

print(summa(10, 20)) # Виведе: 30
```

Змінній *x* при зіставленні буде присвоєно значення 10, а змінній *y* – значення 20. Але мова Python дозволяє також передати значення в функцію, використовуючи *зіставлення за ключами (іменовані аргументи)*, коли передається пара “ім’я-значення”. Для цього при виконанні функції параметрам присвоюються значення, причому послідовність вказання параметрів у цьому випадку може бути довільною:

```
def summa(x, y):
    return x + y

print(summa(y = 20, x = 10)) # Зіставлення за ключами
```

Зіставлення за ключами дуже зручно використовувати, якщо функція має кілька необов’язкових параметрів. В цьому випадку не потрібно вказувати всі значення, а досить присвоїти значення потрібному параметру:

```
def summa (a = 2, b = 3, c = 4): # Всі параметри є необов'язковими
    return a + b + c

print(summa(2, 3, 20)) # Позиційне присвоювання
```

```
print(summa(c = 20))   # Зіставлення за ключами
```

Якщо значення параметрів, які планується передати в функцію, містяться в кортежі або списку, то перед цим кортежем або списком слід вказати символ *. Приклад:

```
def summa(a, b, c):
    return a + b + c

t1, arr = (1, 2, 3), [1, 2, 3]

print(summa(*t1))     # Розпаковуємо кортеж
print(summa(*arr))    # Розпаковуємо список
t2 = (2, 3)
print(summa(1, *t2))  # Можна комбінувати значення
```

Починаючи з Python 3.5, можна передавати таким чином параметри в функцію з декількох списків або кортежів:

```
def summa(a, b, c, d, e):
    return a + b + c + d + e

arr1, arr2 = [1, 2, 3], [4, 5]
print(summa(*arr1, *arr2)) # Розпаковуємо два списки
t11, t12 = (1, 2), (4, 5, 3)
print(summa(*t11, *t12))   # Розпаковуємо два кортежі
```

Якщо значення параметрів містяться в словнику, то перед ним слід поставити дві зірочки:

```
def summa(a, b, c):
    return a + b + c

d1 = {"a": 1, "b": 2, "c": 3}
print(summa(**d1))        # Розпаковуємо словник
t, d2 = (1, 2), {"c": 3}
print(summa(*t, **d2))    # Можна комбінувати значення
```

У Python 3.5 також з’явилася можливість передавати значення параметрів в функцію з декількох словників:

```
def summa(a, b, c, d, e):
    return a + b + c + d + e

d1, d2 = {"a": 1, "b": 2, "c": 3}, {"d": 4, "e": 5}
print(summa(**d1, **d2)) # Розпаковуємо два словника
```

Об’єкти в функцію передаються за посиланням. Якщо об’єкт відноситься до незмінного типу, то зміна значення всередині функції не торкнеться значення змінної поза функцією:

```
def func(a, b):
    a, b = 20, "str"

x, s = 80, "test"
func(x, s)  # Значення змінних x і s не змінюються
print(x, s) # Виведе: 80 test
```

У цьому прикладі значення в змінних *x* і *s* не змінилися. Однак, якщо об’єкт відноситься до змінюваного типу, ситуація буде іншою:

```
def func (a, b):
    a[0], b["a"] = "str", 800
```

```
x = [1, 2, 3]         # Список
y = {"a": 1, "b": 2} # Словник
func (x, y)           # Значення будуть змінені!!!
print(x, y)           # Виведе: ['str', 2, 3] {'a': 800, 'b': 2}
```

Як видно з прикладу, значення в змінних *x* і *y* змінилися, оскільки список і словник відносяться до змінюваних типів.

Ще приклад поведінки функцій демонструє наступний код і рис.:

```
def f(n, list1, list2):
    list1.append(3)
    list2 = [4, 5, 6]
    n = n + 1

x, y, z = 5, [1, 2], [4, 5]
f(x, y, z)
print(x, y, z) # Виведе: 5 [1, 2, 3] [4, 5]
```

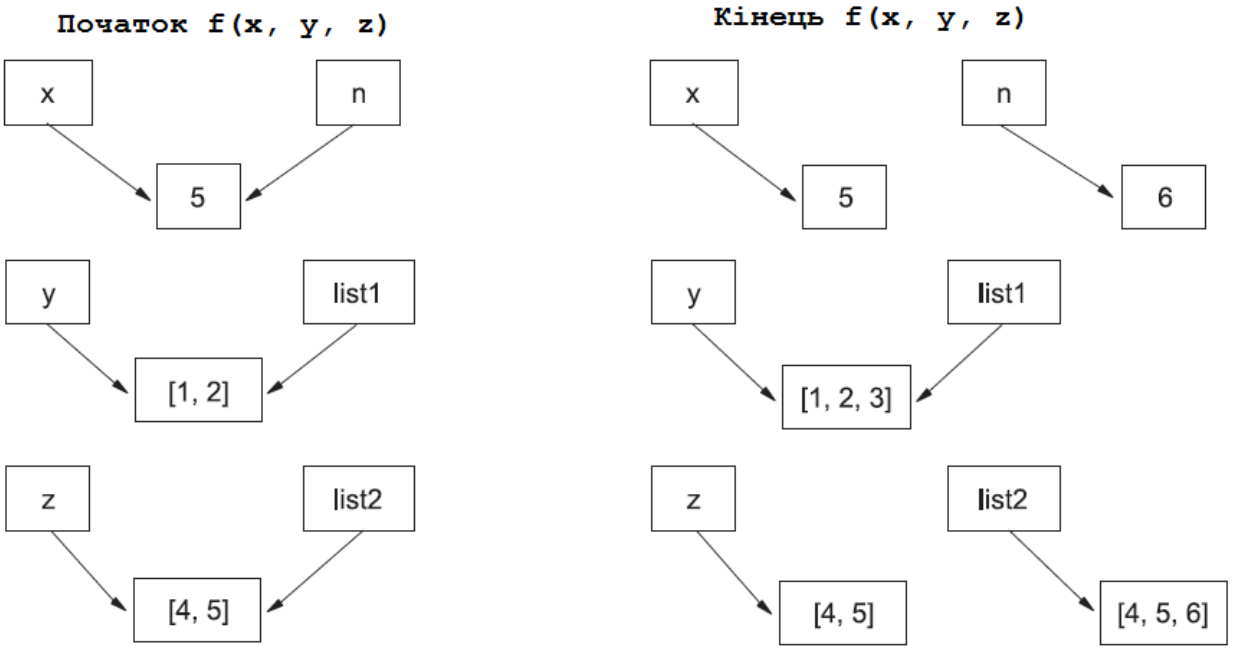

Рис. Передача параметрів функції за посиланням

Щоб уникнути зміни значень, всередині функції слід створити копію об’єкту:

```
def func(a, b):
    a = a[:]      # Створюємо поверхневу копію списку
    b = b.copy()  # Створюємо поверхневу копію словника
    a[0], b["a"] = "str", 800

x = [1, 2, 3]         # Список
y = {"a": 1, "b": 2} # Словник
func (x, y)           # Значення залишаться колишніми
print(x, y)           # Виведе: [1, 2, 3] {'a': 1, 'b': 2}
```

Можна також передати копію об’єкту безпосередньо у виклику функції:

```
func (x[:], y.copy ())
```

Якщо вказати об’єкт, що має змінний тип, як значення параметра за замовчуванням, цей об’єкт буде зберігатися між викликами функції:

```
def func(a = []):
```

```
    a.append(2)
    return a

print(func()) # Виведе: [2]
print(func()) # Виведе: [2, 2]
print(func()) # Виведе: [2, 2, 2]
```

Як видно з прикладу, значення накопичуються всередині списку. Обійти цю проблему можна, наприклад, наступним чином:

```
def func(a = None):
    # Створюємо новий список, якщо значення дорівнює None
    if a is None:
        a = []
    a.append(2)
    return a

print(func())    # Виведе: [2]
print(func([1])) # Виведе: [1, 2]
print(func())    # Виведе: [2]
```

Щоб покращити читабельність коду можна примусово задати, які аргументи є позиційні, а які – ключові, і це можна легко встановити глянувши на визначення функції.

Щоб примусово вказати, які аргументи є позиційні, а які – ключові у визначенні функції використовуються / та *.

Примусово позиційні (positional-only) параметри розташовуються перед / у визначенні функції (починаючи з версії Python 3.8).

Параметри, що йдуть після * є примусово ключові (keyword-only).

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

Розглянемо приклад.

```
def incr(x, /):                          # x - pos
    return x + 1
incr(1)                  # OK
incr(x=1)                # Error

def incr(x, /, start=0):                 # x - pos, start - pos/key
    return start + x + 1
incr(1)                  # OK
incr(1, 5)               # OK
incr(1, start=5)         # OK

def incr(x, /, start=0, *, step=1):      # x/start-див. вище, step-key
    return start + x + step
incr(1)                  # OK
incr(1, 5)               # OK
incr(1, start=5)         # OK
```

```
incr(1, start=5, step=1) # OK
incr(1, start=5, 1)      # Error
```

1.4. Змінна кількість параметрів у функції

Якщо перед параметром у визначенні функції вказати символ *, то функції можна буде передати будь-яку кількість параметрів. Всі передані параметри зберігаються в кортежі. Для прикладу напишемо функцію підсумовування довільної кількості чисел:

```
def summa(*t):
    """Функція приймає будь-яку кількість параметрів"""
    res = 0
    for i in t: # Перебираємо кортеж з переданими параметрами
        res += i
    return res

print(summa(10, 20))                  # Виведе: 30
print(summa(10, 20, 30, 40, 50, 60)) # Виведе: 210
```

Тепер створимо функцію знаходження максимального значення:

```
def maximum(*numbers):
    if len(numbers) == 0:
        return None
    else:
        maxnum = numbers[0]
        for n in numbers[1:]:
            if n > maxnum:
                maxnum = n
        return maxnum

print(maximum(3, 2, 8))         # Виведе: 8
print(maximum(1, 5, 9, -2, 2)) # Виведе: 9
```

І нарешті напишемо функцію для формування піци:

```
def make_pizza(*toppings):
    """Виводить опис піци."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print("- " + topping)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

Результат:

```
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Можна також спочатку вказати кілька обов’язкових параметрів і параметрів, що мають значення за замовчуванням:

```
def summa(x, y = 5, *t): # Комбінація параметрів
```

```
    res = x + y
    for i in t: # Перебираємо кортеж з переданими параметрами
        res += i
    return res

print(summa(10))                        # Виведе: 15
print(summa(10, 20, 30, 40, 50, 60)) # Виведе: 210
```

Якщо перед параметром у визначенні функції вказати дві зірочки **, то всі іменовані параметри будуть збережені в словнику:

```
def func(**d):
    for i in d: # Перебираємо словник з переданими параметрами
        print("{0} => {1}".format(i, d[i]), end = " ")

func (a = 1, b = 2, c = 3) # Виведе: a => 1 b => 2 c => 3
```

При комбінуванні параметрів параметр з двома зірочками записується найостаннішим. Якщо у визначенні функції вказується комбінація параметрів з однією зірочкою і двома зірочками, то функція прийме будь-які передані їй параметри:

```
def func(*t, **d):
    """Функція прийме будь-які параметри"""
    for i in t:
        print(i, end = " ")
    for i in d: # Перебираємо словник з переданими параметрами
        print("{0} => {1}".format(i, d[i]), end = " ")

func(35,10,a = 1,b = 2,c = 3) # Виведе: 35 10 a => 1 b => 2 c => 3
func(10)                      # Виведе: 10
func(a = 1, b = 2)            # Виведе: a => 1 b => 2
```

У визначенні функції можна вказати, що деякі параметри передаються тільки по іменах. Такі параметри повинні вказуватися після параметра з однією зірочкою, але перед параметром з двома зірочками. Іменовані параметри можуть мати значення за замовчуванням:

```
def func(*t, a, b = 10, **d):
    print (t, a, b, d)

func(35, 10, a = 1, c = 3) # Виведе: (35, 10) 1 10 {'c': 3}
func(10, a = 5)            # Виведе: (10,) 5 10 {}
func(a = 1, b = 2)         # Виведе: () 1 2 {}
func(1, 2, 3)              # Помилка. Параметр a обов'язковий!
```

У цьому прикладі змінна *t* прийме будь-яку кількість значень, які будуть об’єднані в кортеж. Змінні *a* та *b* повинні передаватися тільки по іменах, причому змінній *a* при виклику функції обов’язково потрібно передати значення. Змінна *b* має значення за замовчуванням, тому при виклику допускається не передавати їй значення, але якщо значення передається, воно повинно бути зазначено після назви параметра і оператора =. Змінна *d* прийме будь-яку кількість іменованих параметрів і збереже їх в словнику. Зверніть увагу на те, що, хоча змінні *a* та *b* є іменованими, вони не потраплять в цей словник.

Параметра з двома зірочками у визначенні функції може не бути, а ось параметр з однією зірочкою при вказанні параметрів, що передаються тільки по іменах, повинен бути присутнім обов’язково. Якщо функція не повинна приймати

змінну кількість параметрів, але необхідно використовувати змінні, що передаються тільки по іменах, то можна вказати тільки зірочку без змінної:

```
def func(x = 1, y = 2, *, a, b = 10):
    print(x, y, a, b)

func(35, 10, a = 1)        # Виведе: 35 10 1 10
func(10, a = 5)            # Виведе: 10 2 5 10
func(a = 1, b = 2)         # Виведе: 1 2 1 2
func(a = 1, y = 8, x = 7) # Виведе: 7 8 1 10
func(1, 2, 3)              # Помилка. Параметр a обов'язковий!
```

У цьому прикладі значення змінним *x* і *y* можна передавати як по позиціях, так і по іменах. Оскільки змінні мають значення за замовчуванням, допускається взагалі не передавати їм значень. Змінні *a* та *b* розташовані після параметра з однією зірочкою, тому передати значення при виклику можна тільки по іменах. Так як змінна *b* має значення за замовчуванням, допускається не передавати їй значення при виклику, а ось змінна *a* обов'язково повинна отримати значення, причому тільки по імені.

Зазвичай для позначення позиційних та ключових довільних аргументів використовують імена ***args* та ****kwargs*.

Таким чином узагальнений синтаксис функції з довільним число аргументів буде такий:

```
[[[[[mandatory-positional-args], *[args]], mandatory-keyword-args],
**kwargs]]
```

1.5. Анонімні (лямбда) функції

Крім звичайних, мова Python дозволяє використовувати *анонімні функції*, які також називаються *лямбда-функціями*. Анонімна функція не має імені і описується за допомогою ключового слова *lambda* в наступному форматі:

```
lambda [<Параметр1>[,...,<ПараметрN>]]: <Значення, що повертається>
```

Після ключового слова *lambda* можна вказати передані параметри. Як параметр *<Значення, що повертається>* вказується вираз, результат виконання якого буде повернений функцією.

В якості значення анонімна функція повертає посилання на об’єкт-функцію, яке можна зберегти у змінній або передати в якості параметра іншій функції. Викликати анонімну функцію можна, як і звичайну, за допомогою круглих дужок, усередині яких розташовані передані параметри. Приклад використання анонімних функцій:

```
f1 = lambda: 10 + 20           # Функція без параметрів
f2 = lambda x, y: x + y        # Функція з двома параметрами
f3 = lambda x, y, z: x + y + z # Функція з трьома параметрами

print(f1())          # Виведе: 30
print(f2(5, 10))     # Виведе: 15
print(f3(5, 10, 30)) # Виведе: 45
```

Як і у звичайних функцій, деякі параметри анонімних функцій можуть бути необов’язковими. Для цього параметрам у визначенні функції присвоюється значення за замовчуванням

```
f = lambda x, y = 2: x + y
print(f(5))    # Виведе: 7
print(f(5, 6)) # Виведе: 11
```

Найчастіше анонімну функцію не зберігають в змінній, а відразу передають як параметр в іншу функцію. Наприклад, метод списків *sort()* дозволяє вказати призначену для користувача функцію в параметрі *key*. Відсортуємо список без урахування регістру символів, вказавши в якості параметра анонімну функцію:

```
arr = ["одиниця1", "Одиничний", "Одиниця2"]
arr.sort(key = lambda s: s.lower())
for i in arr:
    print (i, end = " ")
# Результат виконання: одиниця1 Одиниця2 Одиничний
```

Можливо також не збергіати анонімну функцію у змінну, а відразу викликати лямбда-функцію:

```
>>> total = (lambda x, y: x + y)(5, 3)
>>> total
8
```

Lambda-вирази можна використовувати для створення таблиць переходів, які представляють собою списки або словники дій, виконуваних на вимогу. Наприклад:

```
L = [lambda x: x ** 2, # Вбудовані визначення функцій
     lambda x: x ** 3,
     lambda x: x ** 4] # Список з трьох функцій

for f in L:
    print(f(2)) # Виведе 4, 8, 16

print(L[0](3)) # Виведе 9
```

Той же приклад без використання *lambda*:

```
def f1(x): return x ** 2
def f2(x): return x ** 3 # Визначення іменованих функцій
def f3(x): return x ** 4

L = [f1, f2, f3] # Посилання по імені

for f in L:
    print(f(2)) # Виведе 4, 8, 16

print(L[0](3)) # Виведе 9
```

Подібні таблиці дій в мові Python можна створювати за допомогою словників та інших структур даних. Приклад:

```
>>> key = 'got'
>>> {'already': (lambda: 2 + 2),
    'got': (lambda: 2 * 4),
    'one': (lambda: 2 ** 6)}[key]()
8
```

Той же приклад без використання lambda:

```
>>> def f1(): return 2 + 2
>>> def f2(): return 2 * 4
>>> def f3(): return 2 ** 6
>>> key = 'one'
>>> {'already': f1, 'got': f2, 'one': f3}[key]()
```

```
64
```

Лямбда-функції часто використовуються для сортування інтерованих об’єктів за альтернативним ключем:

```
>>> sorted(range(-5, 6), key = lambda x: x * x)
[0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5]
```

1.6. Функції-генератори

Функцією-генератором називається функція, яка при послідовних викликах повертає черговий елемент будь-якої послідовності. Призупинити виконання функції і перетворити функцію в генератор дозволяє ключове слово *yield*. Для прикладу напишемо функцію, яка зводить елементи послідовності в зазначену ступінь:

```
def func(x, y):
    for i in range(1, x + 1):
        yield i ** y

for n in func (10, 2):
    print (n, end = " ") # Виведе: 1 4 9 16 25 36 49 64 81 100
print() # Вставляємо порожній рядок

for n in func(10, 3):
    print(n, end = " ") # Виведе: 1 8 27 64 125 216 343 512 729 1000
```

Функції-генератори підтримують метод *__next__()*, який дозволяє отримати наступне значення. Коли значення закінчуються, метод генерує виняток *StopIteration*. Виклик методу *__next__()* в циклі *for* проводиться непомітно для нас. Для прикладу перепишемо попередню програму, використавши метод *__next__()* замість циклу

```
def func(x, y):
    for i in range(1, x + 1):
        yield i ** y

i = func(3, 3)
print(i.__next__()) # Виведе: 1  (1 ** 3)
print(i.__next__()) # Виведе: 8  (2 ** 3)
print(i.__next__()) # Виведе: 27 (3 ** 3)
print(i.__next__()) # Виняток StopIteration
```

Виходить, що за допомогою звичайних функцій ми можемо повернути всі значення відразу у вигляді списку, а за допомогою функцій-генераторів – тільки одне значення за раз. Така особливість дуже корисна при обробці великої кількості значень, оскільки при цьому не знадобиться завантажувати в пам’ять весь список зі значеннями.

Існує можливість викликати одну функцію-генератор з іншої. Для цього застосовується розширений синтаксис ключового слова *yield*:

```
yield from <Функція-генератор, що викликається>
```

Розглянемо наступний приклад. Нехай у нас є список чисел, і нам потрібно отримати інший список, що включає числа в діапазоні від 1 до кожного з чисел в першому списку:

```
def gen(l):
    for e in l:
        yield from range(1, e + 1)

l = [5, 10]
for i in gen([5, 10]): print(i, end = " ")
```

Тут ми в функції-генераторі *gen* перебираємо переданий їй як параметр список і для кожного його елемента викликаємо іншу функцію-генератор. В якості останньої виступає вираз, що створює діапазон від 1 до значення чергового елемента, збільшеного на одиницю (щоб це значення увійшло в діапазон). В результаті на виході ми отримаємо цілком очікуваний результат:

```
1 2 3 4 5 1 2 3 4 5 6 7 8 9 10
```

Ускладнимо завдання, включивши в результуючий список числа, помножені на 2:

```
def gen2(n):
    for e in range(1, n + 1):
        yield e * 2

def gen(l):
    for e in l:
        yield from gen2(e)

l = [5, 10]
for i in gen([5, 10]): print(i, end = " ")
```

Тут ми викликаємо з функції-генератора *gen* написану нами самими функцію-генератор *gen2*. Остання створює діапазон, перебирає всі вхідні в нього числа і повертає їх помноженим на 2. Результат роботи наведеного в лістингу коду такий:

```
2 4 6 8 10 2 4 6 8 10 12 14 16 18 20
```

Функція-генератор це особливий різновид функції, яку можна використовувати для визначення власних ітераторів. При визначенні функцій-генераторів значення кожної ітерації повертається ключовим словом *yield*. Генератор перестає повертати значення, коли ітерацій більше немає, при досягненні порожньої команди *return* або кінця функції. Локальні змінні в функції-генераторі зберігаються між викликами (на відміну від звичайних функцій):

```
def four():
    x = 0
    while x < 4:
        print("in generator, x =", x)
        yield x # Повертає поточне значення x
        x += 1

for i in four():
    print(i)
```

Результат:

```
in generator, x = 0
0
```

```
in generator, x = 1
1
in generator, x = 2
2
in generator, x = 3
3
```

Функцію-генератор також можна використовувати з оператором *in* для перевірки того, чи входить значення в серію, створену генератором:

```
>>> 2 in four()
in generator, x = 0
in generator, x = 1
in generator, x = 2
True
>>> 5 in four()
in generator, x = 0
in generator, x = 1
in generator, x = 2
in generator, x = 3
False
```

1.7. Декоратори функцій

Декоратори – це функції, які приймають як параметр іншу функцію та розширюють її поведінку без явної модифікації.

Декоратори дозволяють змінити поведінку звичайних функцій – наприклад, виконати будь-які дії перед або після виконання функції. Типові приклади використання:

- для вимірювання продуктивності функції: часу виконання, необхідної пам’яті тощо;
- для логування викликів функції;
- кешування результатів;
- перевірки дозволів, конвертації вхідних/вихідних даних, тестування тощо.

Створення декотораторів можливе завдяки тому, що функції в Python можуть присвоюватися змінним, передаватися в інші функції як аргумент, повертатися з функції та декларуватися всередині функцій.

Спочатку розглянемо ручне створення декоратора, який має таку структуру:

```
def decorator_name(original_function):  # Декоратор

    def wrapper():
        # Код, що виконується перед викликом original_function
        original_function()
        # Код, що виконується після виклику original_function

    return wrapper

def original_function():
    ...

decorated_function = decorator_name(original_function)
decorated_function()
```

Для прикладу створимо декоратор для фунеції без параметрів *func():*

```
def start_end_decorator(func):

    def wrapper():
        print('Start decorator')
        func()
        print('End decorator')
    return wrapper

def print_hello():
    print('Hello!!!')

print_hello = start_end_decorator(print_hello)
print_hello()
```

Виведе:

```
Start decorator
Hello!!!
End decorator
```

Проте є більш простіший шлях створення декоратора з допомогою синтаксису декоратора з символом @:

```
def decorator_name(original_function):  # Декоратор

    def wrapper():
        # Код, що виконується перед викликом original_function
        original_function()
        # Код, що виконується після виклику original_function

    return wrapper

@decorator_name     # Оголошуємо декоратор для original_function
def original_function():
    ...

original_function() # Виклик декорованої функції
```

Перепишемо приколад з func() використовуючу синтаксис декораторів:

```
def start_end_decorator(func):

    def wrapper():
        print('Start decorator')
        func()
        print('End decorator')

    return wrapper

@start_end_decorator
def print_hello():
    print('Hello!!!')

print_hello()
```

Виведе:

```
Start decorator
```

```
Hello!!!
End decorator
```

Створимо декоратор для вимірювання часу виконання функції:

```
import time

def timer_decorator(func):

    def wrapper():
        start = time.time()
        func()
        end = time.time()
        print(f"{func.__name__} ran in {end - start} seconds")

    return wrapper

@timer_decorator
def print_hello():
    print('Hello!!!')

print_hello()
```

Виведе:

```
print_hello ran in 0.0059833526611328125 seconds
```

Якщо декорована функція приймає параметри чи повертає значення, то їх обробляють таким чином:

```
import time
def timer_decorator(func):
    def wrapper(arg1, arg2):
        start = time.time()
        result = func(arg1, arg2)
        end = time.time()
        print(f"{func.__name__} ran in {end - start} seconds")
        return result
    return wrapper

@timer_decorator
def summa(x, y):
    return x + y

print(summa(2, 3))
```

Виведе:

```
summa ran in 0.0 seconds
5
```

Якщо потрібно створити універсальний декоратор для будь-якої функції з довільною кількістю параметрів, то структура декоратора виглядатиме так:

```
def general_decorator(original_function):

    def wrapper(*args, **kwargs):

        # Код, що виконується перед викликом original_function
        result = original_function(*args, **kwargs)
        # Код, що виконується після виклику original_function

        return result
```

```
    return wrapper

@general_decorator
def original_function(args, kwargs):
    ...
```

Створимо універсальний декоратор для вимірювання часу виконання функції:

```
import time
def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} ran in {end - start} seconds")
        return result
     return wrapper

@timer_decorator
def summa(a, b, c, d):
    return a + b + c + d

print(summa(2, 3, c=4, d=5))
```

Виведе:

```
summa ran in 0.0 seconds
14
```

Проте є одна проблема – рядок документування та атрибут імені функції вказують на обгортку, а не первинну декоровану функцію:

```
def start_end_decorator(func):

    def wrapper():
        """Wrapper docstring"""
        print('Start decorator')
        func()
        print('End decorator')
    return wrapper

@start_end_decorator
def print_hello():
    """print_hello docstring"""
    print('Hello!!!')

print(print_hello.__name__)
print(print_hello.__doc__)
```

Виведе:

```
wrapper
Wrapper docstring
```

Щоб виправити це можна використати декоратор *wraps* з бібліотеки *functools*, який зберігає інформацію про оригінальну функцію:

```
import functools
def start_end_decorator(func):
    @functools.wraps(func)
```

```
    def wrapper():
        """Wrapper docstring"""
        print('Start decorator')
        func()
        print('End decorator')
    return wrapper
@start_end_decorator
def print_hello():
    """print_hello docstring"""
    print('Hello!!!')

print(print_hello.__name__)
print(print_hello.__doc__)
```

Виведе:

```
print_hello
print_hello docstring
```

Можна вказати відразу кілька функцій-декораторів:

```
def star_decorator(func):
    def inner(*args, **kwargs):
        print("*" * 15)
        func(*args, **kwargs)
        print("*" * 15)
    return inner

def percent_decorator(func):
    def inner(*args, **kwargs):
        print("%" * 15)
        func(*args, **kwargs)
        print("%" * 15)
    return inner

@star_decorator
@percent_decorator
def show(msg):
    print(msg)

show("Hello")
```

Тут спочатку буде викликана функція *star_decorator()*, а потім функція *percent_decorator()*. Цей код еквівалентний:

```
show = star(percent(show))
```

Результат:

```
***************
%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%
***************
```

```
@percent_decorator
@star_decorator
def show(msg):
    print(msg)

show("Hello")
```

Тепер спочатку буде викликана функція *percent_decorator()*, а потім функція *star_decorator()*.

Цей код еквівалентний:

```
show = percent(star(show))
```

Результат:

```
%%%%%%%%%%%%%%%
***************
Hello
***************
%%%%%%%%%%%%%%%
```

Можна використовувати декоратори з аргументами. Наприклад, будем міряти час виконання функції при заданому числі повторів:

```
def timer_decorator_with_repeat(num_repeats=2):
  def timer_decorator(func):
      def wrapper(*args, **kwargs):
          start = time.time()
          for _ in range(num_repeats):
              result = func(*args, **kwargs)
          end = time.time()
          print(f"{func.__name__} ran in {(end - start)/num_repeats}
seconds")
          return result
      return wrapper
  return timer_decorator

@timer_decorator_with_repeat(5)
def long_loop(loop_value):
    for i in range(loop_value):
        pass

long_loop(100_000)    # Виведе: long_loop ran in 0.008377695083618164
seconds
```

1.8. Вкладені функції

Одну функцію можна вкласти в іншу функцію, причому рівень вкладеності не обмежений. Такі функції називаються *вкладеними функціями* (nested functions), або *внутрішніми функціями* (inner functions). При цьому вкладена функція отримує свою власну локальну область видимості і має доступ до змінних, оголошених всередині функції, в яку вона вкладена (*функції-батька*). Розглянемо вкладення функцій на прикладі:

```
def func1(x):
    def func2():
        print(x)
    return func2

f1 = func1(10)
f2 = func1(99)
f1() # Виведе: 10
f2() # Виведе: 99

def outer(a, b):
```

```
    def inner(c, d):
        return c + d
    return inner(a, b)

print(outer(4, 7)) # Виведе 11
```

Тут ми визначили функцію *func1()*, приймаючу один параметр, а всередині неї - вкладену функцію *func2()*. Результатом виконання функції *func1()* буде посилання на цю вкладену функцію. Усередині функції *func2()* ми здійснюємо вивід значення змінної *x*, яка є локальною в функції *func1()*. Таким чином, крім локальної, глобальної та вбудованої областей видимості, додається вкладена область видимості. При цьому пошук ідентифікаторів спочатку проводиться всередині вкладеної функції, потім всередині функції-батька, далі у функціях вищого рівня і лише потім в глобальній і вбудованій областях видимості. У нашому прикладі змінна *x* буде знайдена в області видимості функції *func1()*.

Ще приклади:

```
def maker(n):
    def action(x):    # Make and return action
        return x ** n # action retains n from enclosing scope
    return action

f = maker(2) # Передаємо 2 як аргумент n
print(f)
print(f(3))  # Передає 3 до x, n запамяталося як 2: 3 ** 2
print(f(4))  # 4 ** 2
```

Результат:

```
<function maker.<locals>.action at 0x00000000005EC1E0>
9
16
```

Можна повертати не вкладену функції, а її результат:

```
def speak(text):
    def whisper(t):
        return t.lower() + '...'
    return whisper(text)

print(speak('Hello, World')) # Виведе: hello, world...
```

Можна вкласти декілька функцій:

```
def get_speak_func(volume):
    def whisper(text):
        return text.lower() + '...'
    def yell(text):
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper
```

Зверніть увагу на те, як функція *get_speak_func* фактично не викликає ні одну зі своїх внутрішніх функцій - вона просто вибирає відповідну внутрішню функцію на основі аргументу *volume* і потім повертає об'єкт-функцію:

```
>>> get_speak_func(0.3)
<function get_speak_func.<locals>.whisper at 0x0000000001CDC1E0>
>>> get_speak_func(0.7)
```

```
<function get_speak_func.<locals>.yell at 0x0000000002E7CF28>
```

Можна викликати повернуту функцію безпосередньо, або спочатку присвоївши її змінній:

```
>>> speak_func = get_speak_func(0.7)
>>> speak_func('Hello')
'HELLO!'
>>> (get_speak_func(0.2))('Hello')
'hello...'
```

Слід враховувати, що в момент визначення функції зберігаються посилання на змінні, а не їх значення. Наприклад, якщо після визначення функції *func2()* провести зміну змінної *x*, то буде використовуватися це нове значення:

```
def func1(x):
    def func2():
        print(x)
    x = 30
    return func2

f1 = func1(10)
f2 = func1(99)
f1() # Виведе: 30
f2() # Виведе: 30
```

Зверніть увагу на результат виконання. В обох випадках ми отримали значення 30. Якщо необхідно зберегти саме значення змінної при визначенні вкладеної функції, слід передати значення як значення за замовчуванням:

```
def func1(x):
    def func2(x = x): # Зберігаємо поточне значення, а не посилання
        print(x)
    x = 30
    return func2

f1 = func1(10)
f2 = func1(99)
f1() # Виведе: 10
f2() # Виведе: 99
```

Тепер спробуємо з вкладеної функції *func2()* змінити значення змінної *x*, оголошеної всередині функції *func1()*. Якщо всередині функції *func2()* присвоїти значення змінній *x*, буде створена нова локальна змінна з таким же ім'ям. Якщо всередині функції *func2()* оголосити змінну як глобальну і присвоїти їй значення, то зміниться значення глобальної змінної, а не значення змінної *x* всередині функції *func1()*. Таким чином, жоден з вивчених раніше способів не дозволяє з вкладеної функції змінити значення змінної, оголошеної всередині функції-батька. Щоб вирішити цю проблему, слід оголосити необхідні змінні за допомогою ключового слова *nonlocal*:

```
def func1(a):
    x = a
    def func2(b):
        nonlocal x # Оголошуємо змінну як nonlocal
        print(x)
        x = b # Можемо змінити значення x в func1()
    return func2
```

```
f = func1(10)
f(5)  # Виведе: 10
f(12) # Виведе: 5
f(3)  # Виведе: 12
```

При використанні ключового слова *nonlocal* слід пам'ятати, що змінна обов'язково повинна існувати всередині функції-батька. В іншому випадку буде виведено повідомлення про помилку.

Локальні функції формують так звані *замикання (closure)*. Замикання пам'ятає об'єкт з області замикання, який потребує локальна функція. Python імплементує замикання з допомогою спеціального атрибуту *__closure__*. Якщо функція замикає якийсь об'єкт, тоді ця функція має атрибут *__closure__*, який підтримує посилання на цей об'єкт.

```
def enclosing():
    x = 'closed over'
    def local_func():
        print(x)
    return local_func
>>> lf = enclosing()
>>> lf()
closed over
>>> print(lf.__closure__)
(<cell at 0x0000000002C18438: str object at 0x0000000002C2BC30>,)
```

Атрибут *__closure__* функції *lf* вказує, що *lf* є замиканням, і замикання посилається на один об'єкт – змінну *x* визначену у функції *lf*.

Замикання часто використовуються у *фабриках-функцій (function factories).* Ці фабрики це функції, які повертають спеціалізовані функції, залежно від аргументів фабрики. Інакше кажучи, фабрики-функцій приймають певні аргументи та створюють локальні функції, що приймають свої власні аргументи, проте використовують і аргументи передані фабриці. Приклад:

```
def raise_to(exp):
    def raise_to_exp(x):
        return pow(x, exp)
    return raise_to_exp
```

raise_to примає один аргумент- степінь *exp* і повертає функцію, що підносить аргумент до цієї степені. Локальна функція *raise_to_exp* звертається до *exp* в своїй імплементації і це означає, що Python створить замикання для посилання на цей об'єкт. Це можна перевірити викликавши *raise_to*:

```
>>> square = raise_to(2)
>>> square.__closure__
(<cell at 0x0000000002C38438: int object at 0x000007FEEAFDD440>,)
```

Тепер, коли ми передали *exp = 2* піднесемо до квадрату:

```
>>> square(5)
25
>>> square(9)
81
```

Ми можемо створити функцію піднесення до кубу таким самим чином:

```
>>> cube = raise_to(3)
>>> cube(3)
27
>>> cube(10)
```

```
1000
```

Наприклад, виконаємо сортування рядків за останньою літерою трьома способами: окремими функціями, вкладеними функціями і анонімними функціями:

```
def last_letter(s):
    return s[-1]

def sort_by_last_letter(strings):
    return sorted(strings, key = last_letter)

print(sort_by_last_letter(['hello','from','a','local','function']))

def sort_by_last_letter(strings):
    def last_letter(s):
        return s[-1]
    return sorted(strings, key = last_letter)

print(sort_by_last_letter(['hello','from','a','local','function']))

def sort_by_last_letter(strings):
    return sorted(strings, key = lambda s: s[-1])

print(sort_by_last_letter(['hello','from','a','local','function']))
```

Результат для всіх трьох варіантів буде:

```
['a', 'local', 'from', 'function', 'hello']
```

1.9. Глобальні і локальні змінні

Глобальні змінні – це змінні, оголошені в програмі поза функцією. У Python глобальні змінні видно в будь-якій частині модуля, включаючи функції:

```
def func(glob2):
    print("Значення глобальної змінної globl =", glob1)
    glob2 += 10
    print("Значення локальної змінної glob2 =", glob2)

glob1, glob2 = 10, 5
func(77) # Викликаємо функцію
print("Значення глобальної змінної glob2 =", glob2)
```

Результат виконання:

```
Значення глобальної змінної globl = 10
Значення локальної змінної glob2 = 87
Значення глобальної змінної glob2 = 5
```

Змінній *glob2* всередині функції присвоюється значення, передане при виклику функції. В результаті створюється нова змінна з тим же ім’ям, але яка є локальною. Всі зміни цієї змінної всередині функції не торкнуться значення однойменної глобальної змінної.

Локальні змінні – це змінні, які оголошуються всередині функцій. Якщо ім’я локальної змінної співпадає з назвою глобальної змінної, то всі операції всередині функції здійснюються з локальною змінною, а значення глобальної змінної не змінюється. Локальні змінні видно тільки всередині тіла функції:

```
def func():
    local1 = 77 # Локальна змінна
    glob1  = 25 # Локальна змінна
    print("Значення glob1 всередині функції =", glob1)

glob1 = 10 # Глобальна змінна
func()     # Викликаємо функцію
print("Значення glob1 поза функцією =", glob1)

try:
    print(local1) # Викличе виняток NameError
except NameError: # Обробляємо виняток
    print("Змінна local1 не видима поза функцією")
```

Результат виконання:

```
Значення glob1 всередині функції = 25
Значення glob1 поза функцією = 10
Змінна local1 не видима поза функцією
```

Як видно з прикладу, змінна *local1*, оголошена всередині функції *func()*, недоступна поза функцією. Оголошення всередині функції локальної змінної *glob1* не змінило значення однойменної глобальної змінної.

Якщо звернення до змінної проводиться до присвоєння їй значення (навіть якщо існує однойменна глобальна змінна), буде згенеровано виняток *unboundLocalError*:

```
def func():
    # Локальна змінна ще не визначена
    print(glob1) # Цей рядок викличе помилку!!!
    glob1 = 25   # Локальна змінна

glob1 = 10 # Глобальна змінна
func()     # Викликаємо функцію
# Результат виконання:
#  UnboundLocalError: local variable 'glob1' referenced before
assignment
```

Для того щоб значення глобальної змінної можна було змінити всередині функції, необхідно оголосити змінну глобальною за допомогою ключового слова *global*. Продемонструємо це на прикладі:

```
def func():
    # Оголошуємо змінну glob1 глобальною
    global glob1
    glob1 = 25 # Змінюємо значення глобальної змінної
    print("Значення glob1 всередині функції =", glob1)

glob1 = 10 # Глобальна змінна
print("Значення glob1 поза функцією =", glob1)
func()     # Викликаємо функцію
print("Значення glob1 після функції =", glob1)
```

Результат виконання:

```
Значення glob1 поза функцією = 10
Значення glob1 всередині функції = 25
Значення glob1 після функції = 25
```

Таким чином, пошук ідентифікатора, що використовується всередині функції, буде проводитися в наступному порядку:

1. Пошук оголошення ідентифікатора всередині функції (в локальній області видимості).

2. Пошук оголошення ідентифікатора у батьківських функціях.

3. Пошук оголошення ідентифікатора в глобальній області.

4. Пошук у вбудованій області видимості (вбудовані функції, класи і т. д.).

У Python ці правила відомі як LEGB (Local-Enclosing-Global-Built-in):

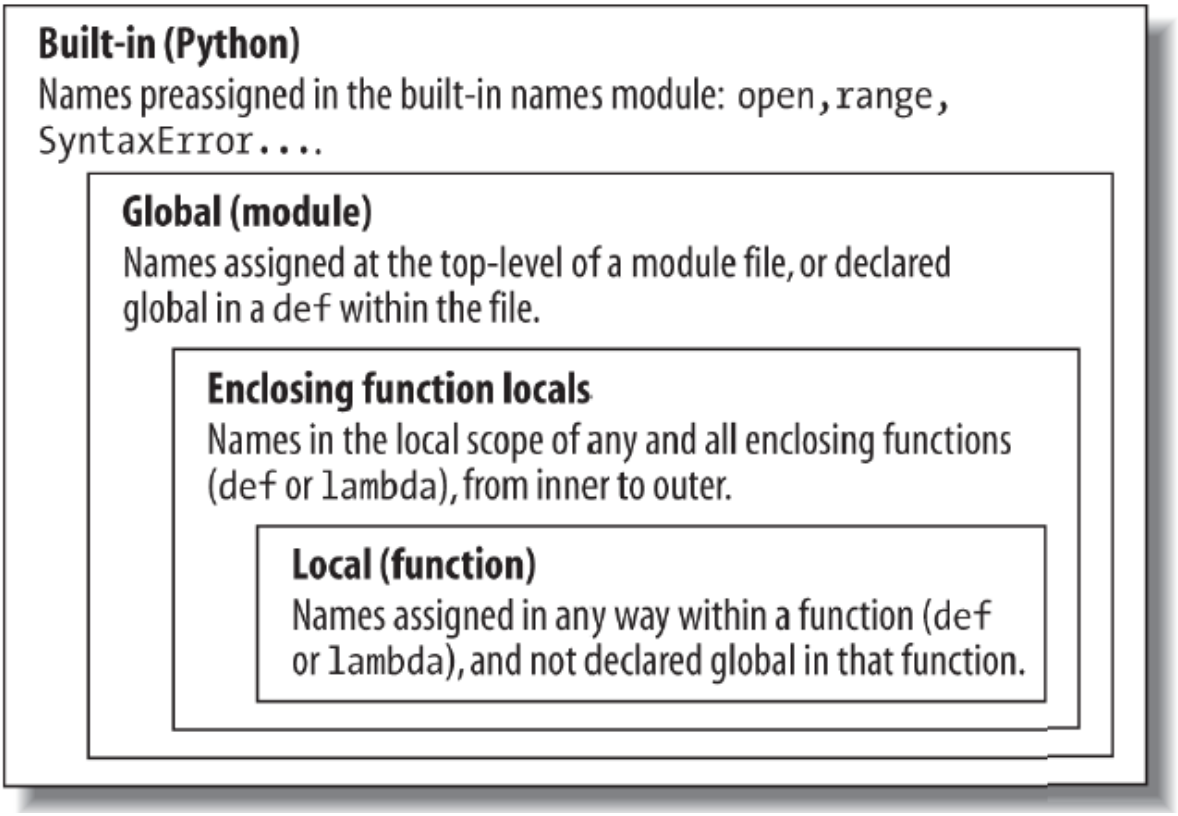

Рис. Області видимості в Python

Локальні функції теж працюють за правилами LEGB. Приклад:

```
g = 'global'

def outer(p = 'param'):
    l = 'local'
    def inner():
        print(g, p, l)
    inner()
outer()  # Виведе: global param local
```

Роботу з локальними, вкладеними та глобальними змінними демонструє наступний приклад:

```
message = 'global'

def enclosing():
    message = 'enclosing'

    def local():
        message = 'local'

    print('enclosing message:', message)
    local()
    print('enclosing message:', message)
```

```
print('global message:', message)
enclosing()
print('global message:', message)
```

Результат:

```
global message: global
enclosing message: enclosing
enclosing message: enclosing
global message: global
```

Щоб змінити глобальну змінну в локальній функції використаємо ключове слово *global*:

```
message = 'global'

def enclosing():
    message = 'enclosing'

    def local():
        global message
        message = 'local'

    print('enclosing message:', message)
    local()
    print('enclosing message:', message)

print('global message:', message)
enclosing()
print('global message:', message)
```

Результат:

```
global message: global
enclosing message: enclosing
enclosing message: enclosing
global message: local
```

Щоб змінити вкладену змінну в локальній функції використаємо ключове слово *nonlocal*:

```
message = 'global'

def enclosing():
    message = 'enclosing'

    def local():
        nonlocal message
        message = 'local'

    print('enclosing message:', message)
    local()
    print('enclosing message:', message)

print('global message:', message)
enclosing()
print('global message:', message)
```

Результат:

```
global message: global
enclosing message: enclosing
```

```
enclosing message: local
global message: global
```

Розглянемо більш практичний варіант використання вкладених функцій та нелокальних змінних на прикладі створення таймеру:

```
import time

def make_timer():
    last_called = None # Never

    def elapsed():
        nonlocal last_called
        now = time.time()
        if last_called is None:
            last_called = now
            return None
        result = now - last_called
        last_called = now
        return result
    return elapsed
>>> t = make_timer()
>>> t()
>>> t()
2.629150390625
>>> t()
4.409252166748047
```

При використанні анонімних функцій слід враховувати, що при вказанні всередині функції глобальної змінної буде збережено посилання на цю змінну, а не її значення в момент визначення функції:

```
x = 5
func = lambda: x # Зберігається посилання, а не значення x!!!
x = 80 # Змінили значення
print(func()) # Виведе: 80, а не 5
```

Якщо необхідно зберегти саме поточне значення змінної, можна скористатися наступним способом:

```
x = 5
func = (lambda y: lambda: y)(x) # Зберігається значення змінної x
x = 80 # Змінили значення
print(func()) # Виведе: 5
```

Зверніть увагу на другий рядок прикладу. У ній ми визначили анонімну функцію з одним параметром, що повертає посилання на вкладену анонімну функцію. Далі ми викликаємо першу функцію за допомогою круглих дужок і передаємо їй значення змінної *x*. В результаті зберігається поточне значення змінної, а не посилання на неї.

Зберегти поточне значення змінної також можна, вказавши глобальну змінну в якості значення параметра за замовчуванням у визначенні функції:

```
x = 5
func = lambda x = x: x # Зберігається значення змінної x
x = 80                 # Змінили значення
print(func())          # Виведе: 5
```

Отримати всі ідентифікатори та їх значення дозволяють наступні функції:

♦ *globals()* - повертає словник з глобальними ідентифікаторами;

♦ *locals()* - повертає словник з локальними ідентифікаторами.

Приклад використання обох цих функцій:

```
def func():
    local1 = 54
    glob2  = 25
    print("Глобальні ідентифікатори всередині функції")
    print(sorted(globals().keys()))
    print("Локальні ідентифікатори всередині функції")
    print(sorted(locals().keys()))

glob1, glob2 = 10, 88
func()
print("Глобальні ідентифікатори поза функцією")
print(sorted(globals().keys()))
```

Результат виконання:

```
Глобальні ідентифікатори всередині функції
['__annotations__',     '__builtins__',     '__doc__',     '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'func', 'glob1',
'glob2']
Локальні ідентифікатори всередині функції
['glob2', 'local1']
Глобальні ідентифікатори поза функцією
['__annotations__',     '__builtins__',     '__doc__',     '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'func', 'glob1',
'glob2']
```

♦ *vars([<Об'єкт>])* - якщо викликається без параметра всередині функції, повертає словник з локальними ідентифікаторами. Якщо викликається без параметра поза функцією, повертає словник з глобальними ідентифікаторами. При вказівці об'єкта в якості параметра повертає ідентифікатори цього об'єкта (еквівалентно виклику *<об'єкт>.__dict__*). Приклад використання цієї функції:

```
def func():
    local1 = 54
    glob2  = 25
    print("Локальні ідентифікатори всередині функції")
    print(sorted(vars().keys()))

glob1, glob2 = 10, 88
func()
print("Глобальні ідентифікатори поза функцією")
print(sorted (vars().keys()))
print("Вказівка об'єкта в якості параметра")
print(sorted(vars(dict).keys()))
print("Альтернативний виклик")
print(sorted(dict.__dict__.keys()))
```

1.10. Анотації функцій

У Python функції можуть містити *анотації*, які вводять новий спосіб документування. Тепер в заголовку функції допускається вказувати призначення кожного параметру, його тип даних, а також тип значення, що повертається функцією. Анотації мають такий вигляд:

```
def <Ім'я функції>(
```

```
    [<Параметр1> [: <Вираз>] [= <Значення за замовчуванням>] [, ...,
     <ПараметрN> [: <Вираз>] [=<Значення за замовчуванням>]]]
    ) -> <Значення, що повертається>:
    <Тіло функції>
```

В параметрах *<Вираз>* та *<Значення, що повертається>* можна вказати будь-який припустимий вираз мови Python. Цей вираз буде виконано при створенні функції.

Приклад вказання анотацій:

```
>>> def func(a: "Параметр1", b: 10 + 5 = 3) -> None:
    print(a, b)
```

У цьому прикладі для змінної *a* створено опис *"Параметр1"*. Для змінної *b* вираз 10 + 5 є описом, а число 3 - значенням параметра за замовчуванням. Крім того, після закриваючої круглої дужки вказано тип значення, що повертається функцією: *None*. Після створення функції всі вирази будуть виконані, і результати збережуться у вигляді словника в атрибуті *__annotations__* об'єкта функції.

Для прикладу виведемо значення цього атрибута:

```
>>> def func(a: "Параметр1", b: 10 + 5 = 3) -> None:
    pass

>>> func.__annotations__
{'a': 'Параметр1', 'b': 15, 'return': None}
```

2. ЗАВДАННЯ

2.1. Домашня підготовка до роботи

1. Вивчити теоретичний матеріал.

2.2. Виконати в лабораторії

1. Встановити з допомогою *pip* статичний аналізатор типів *mypy*. Для *mypy* правила перевірки задаються у файлі *mypy.ini*, який розташуйте разом з файлом *test_mypy.py*. Файли *test_mypy.py* та *mypy.ini* є в матеріалах до лабораторної роботи.
2. Переглянути файл *test_mypy.py*, самостійно відзначити порушення правил анотацій типів в ньому.
3. Скоригуйте знайдені порушення та збережіть виправлений файл як *test_mypy_my.py*.
4. Перевірте файл *test_mypy.py* з допомогою *mypy* і порівняйте їх результати зі своїми.
5. Перевірте файл *test_mypy_my.py* з допомогою *mypy* і оцініть наскільки добре ви провели виправлення. Усуньте всі помилки і зауваження, які видав чекер.
6. Написати програму валідації введеного паролю з Лабораторної роботи №4 з використанням функцій. Користувач вводить пароль, програма має перевірити наявність у ньому лише заданих типів символів у вказаних пропорціях і з дотримання додаткових правил згідно варіанту у табл. 1 і

вивести інформацію про результати перевірки у форматі як показано на рис.

```
Введіть пароль довжиною не менше 12 символів.
Вимоги до паролю:
1.  Маленькі латинські літери
2.  Великі латинські літери
3.  Цифри
4.  Спеціальні символи !@#$_-%^&*
5.  Не менше 3 і не більше 5 маленьких латинських літер
6.  Не менше 3 і не більше 5 великих латинських літер
7.  Не менше 2 і не більше 4 цифр
8.  Не менше 2 і не більше 4 спеціальних символів
9.  Не більше 3 однакових спеціальних символів
10. Не більше 3 однакових маленьких латинських літер підряд
> 67п!!!!aVFk
Довжина не менше 12 символів - FAIL!
Пароль містить лише допустимі символи - FAIL!
Маленькі латинські літери - FAIL!
Великі латинські літери - FAIL!
Цифри - OK!
Спеціальні символи - OK!
Не більше 3 однакових спеціальних символів - FAIL!
Не більше 3 однакових маленьких латинських літер підряд - OK!

Пароль не валідний!

Введіть пароль довжиною не менше 12 символів.
Вимоги до паролю:
1.  Маленькі латинські літери
2.  Великі латинські літери
3.  Цифри
4.  Спеціальні символи !@#$_-%^&*
5.  Не менше 3 і не більше 5 маленьких латинських літер
6.  Не менше 3 і не більше 5 великих латинських літер
7.  Не менше 2 і не більше 4 цифр
8.  Не менше 2 і не більше 4 спеціальних символів
9.  Не більше 3 однакових спеціальних символів
10. Не більше 3 однакових маленьких латинських літер підряд
> *6sss4SD*5ZZ*
Довжина не менше 12 символів - OK!
Пароль містить лише допустимі символи - OK!
Маленькі латинські літери - OK!
Великі латинські літери - OK!
Цифри - OK!
Спеціальні символи - OK!
Не більше 3 однакових спеціальних символів - OK!
Не більше 3 однакових маленьких латинських літер підряд - OK!

Пароль валідний!
```

7. Написати програму генерації паролю. з Лабораторної роботи №3 з використанням функцій. Користувач повинен ввести кількість різних типів символів у паролі згідно варіанту у табл. 2 і вивести згенерований пароль у форматі як показано на рис.

```
Петренко Олег Степанович, КБ-101, 2024. Варіант 16

Введіть кількість великих латинських літер в паролі: 3
Введіть кількість малих латинських літер в паролі: 3
Введіть кількість цифр в паролі: 2
Введіть кількість спеціальних символів !@#$_-%^&* в паролі: 1
Password: b1JX&w9Cs
```

8. Написати програму яка створює і виводить двовимірний список з 5 елементів. Кожен елемент списку представляє собою список, який містить опис атрибутів об’єкту згідно таблиці 3. Організуйте діалоговий режим із вводом з клавіатури, який дозволяє робити такі операції:
    a. Вивести весь список.
    b. Додавати елементи до списку.
    c. Відсортувати список за заданим атрибутом.
    d. Видаляти елементи за заданим атрибутом.
    e. Видаляти елемент за заданим індексом.
    f. Виводити всі елементи за заданим атрибутом.

Всі операції для пунктів 6-8 повинні бути оформлені у вигляді функцій з анотаціями і проходити перевірку *туру*.
Номер варіанту відповідає номеру в списку групи.

3. ЗМІСТ ЗВІТУ

1. Мета роботи.
2. Повний текст завдання згідно варіанту.
3. Лістинг програми.
4. Результати роботи програм (у текстовій формі та скріншот).
5. Висновок.

У якості наборів символів можуть виступати:
upp_char = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
low_char = "abcdefghijklmnopqrstuvwxyz"
num_char = "0123456789"
spc_char = "!@#$%^&*_-"

Додаткові правила для формування паролів:
1. Не менше 2 маленьких латинських літер
2. Не менше 3 маленьких латинських літер
3. Не менше 4 маленьких латинських літер
4. Не менше 5 маленьких латинських літер
5. Не менше 2 великих латинських літер
6. Не менше 3 великих латинських літер

7. Не менше 4 великих латинських літер
8. Не менше 5 великих латинських літер
9. Не менше 2 цифр
10. Не менше 3 цифр
11. Не менше 2 спеціальних символів
12. Не менше 3 спеціальних символів
13. Не більше 4 маленьких латинських літер
14. Не більше 5 маленьких латинських літер
15. Не більше 6 маленьких латинських літер
16. Не більше 7 маленьких латинських літер
17. Не більше 4 великих латинських літер
18. Не більше 5 великих латинських літер
19. Не більше 6 великих латинських літер
20. Не більше 4 цифр
21. Не більше 5 цифр
22. Не більше 3 спеціальних символів
23. Не більше 4 спеціальних символів
24. Не більше 2 однакових маленьких латинських літер
25. Не більше 3 однакових маленьких латинських літер
26. Не більше 2 однакових великих латинських літер
27. Не більше 3 однакових великих латинських літер
28. Не більше 2 однакових цифр
29. Не більше 2 однакових спеціальних символів
30. Не більше 3 однакових спеціальних символів
31. Не більше 3 однакових маленьких латинських літер підряд
32. Не більше 3 однакових великих латинських літер підряд
33. Не більше 3 однакових цифр підряд
34. Не більше 3 однакових спеціальних символів підряд

Табл. 1

Варіанти завдань

Варіант	Довжина, символів	Набір символів	Додаткові правила
1.	12	low_char + spc_char + upp_char	3, 15, 11, 23, 7, 18, 25, 32
2.	15	low_char + num_char + upp_char	4, 16, 9, 21, 8, 19, 27, 33
3.	10	low_char + num_char + spc_char + upp_char	1, 14, 10, 20, 11, 22, 5, 17, 28, 34
4.	9	spc_char + upp_char	12, 23, 6, 17, 26, 32
5.	12	low_char + num_char + spc_char + upp_char	2, 14, 9, 20, 11, 23, 6, 18, 30, 31
6.	10	low_char + num_char + spc_char	1, 13, 10, 20, 12, 22, 29, 33
7.	14	low_char + num_char + spc_char + upp_char	3, 15, 10, 21, 11, 22, 7, 18, 29, 31
8.	9	low_char + upp_char	3, 14, 7, 18, 24, 32
9.	8	num_char + spc_char	9, 21, 11, 23, 30, 33
10.	9	low_char + spc_char	2, 15, 12, 23, 25, 34
11.	8	num_char + spc_char + upp_char	9, 20, 11, 22, 5, 17, 26, 33

12.	11	low_char + spc_char + upp_char	2, 13, 12, 23, 7, 18, 25, 32
13.	14	low_char + num_char + spc_char + upp_char	3, 16, 10, 21, 11, 23, 7, 19, 27, 34
14.	10	low_char + spc_char + upp_char	3, 15, 11, 22, 6, 18, 30, 31
15.	11	low_char + num_char + spc_char + upp_char	1, 13, 9, 20, 11, 22, 5, 17, 28, 32
16.	13	low_char + num_char + spc_char + upp_char	4, 15, 9, 21, 12, 23, 5, 18, 24, 34
17.	13	low_char + num_char + spc_char + upp_char	3, 16, 10, 20, 11, 22, 6, 19, 29, 33
18.	11	low_char + upp_char	2, 14, 6, 18, 25, 32
19.	12	low_char + num_char + spc_char + upp_char	2, 13, 9, 20, 11, 22, 5, 17, 27, 31
20.	11	low_char + spc_char + upp_char	2, 13, 12, 23, 6, 18, 26, 34
21.	6	num_char + spc_char	9, 20, 11, 23, 28, 34
22.	14	low_char + num_char + spc_char + upp_char	3, 15, 10, 21, 12, 23, 6, 18, 24, 32
23.	9	low_char + num_char + spc_char + upp_char	1, 14, 9, 21, 11, 23, 5, 17, 29, 31
24.	10	spc_char + upp_char	12, 23, 8, 19, 30, 32
25.	9	low_char + num_char + upp_char	3, 14, 9, 20, 5, 18, 25, 33
26.	7	num_char + upp_char	9, 20, 5, 17, 28, 32
27.	10	low_char + num_char + spc_char + upp_char	1, 13, 9, 20, 11, 22, 5, 18, 27, 34
28.	15	low_char + num_char + spc_char + upp_char	4, 16, 10, 21, 12, 23, 6, 19, 26, 33
29.	8	num_char + spc_char + upp_char	9, 20, 11, 22, 6, 18, 29, 32
30.	13	low_char + num_char + upp_char	3, 15, 10, 21, 7, 19, 25, 33

upp_char = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
low_char = "abcdefghijklmnopqrstuvwxyz"
num_char = "0123456789"
spc_char = "!@#$%^&*_-"

У душках в табл. 2 вказано кількість символів з даного набору. Напр. low_char(3) означає, що пароль має містити 3 маленькі латинські літери.

Табл. 2

Варіанти завдань

Варіант	**Довжина паролю, символів**	**Набір символів**
1.	12	low_char(5) + spc_char(3) + upp_char(4)
2.	15	low_char(7) + num_char(4) + upp_char(4)
3.	7	low_char(3) + num_char(2) + spc_char(1) + upp_char(1)
4.	7	spc_char(2) + upp_char(5)
5.	8	low_char(3) + num_char(3) + spc_char(1) + upp_char(1)
6.	7	low_char(2) + num_char(3) + spc_char(2)
7.	14	low_char(6) + num_char(3) + spc_char(2) + upp_char(3)
8.	9	low_char(5) + upp_char(4)
9.	12	num_char(7) + spc_char(5)
10.	13	low_char(9) + spc_char(4)
11.	8	num_char(3) + spc_char(2) + upp_char(3)
12.	10	low_char(4) + spc_char(2) + upp_char(4)
13.	14	low_char(5) + num_char(4) + spc_char(2) + upp_char(3)
14.	9	low_char(3) + spc_char(1) + upp_char(5)
15.	8	low_char(2) + num_char(1) + spc_char(1) + upp_char(4)
16.	13	low_char(4) + num_char(3) + spc_char(2) + upp_char(4)
17.	13	low_char(3) + num_char(5) + spc_char(3) + upp_char(2)

18.	11	low_char(7) + upp_char(4)
19.	11	low_char(2) + num_char(1) + spc_char(2) + upp_char(6)
20.	11	low_char (2)+ spc_char(2) + upp_char(7)
21.	6	num_char(4) + spc_char(2)
22.	12	low_char(6) + num_char(1) + spc_char(2) + upp_char(3)
23.	9	low_char(3) + num_char(3) + spc_char(2) + upp_char(1)
24.	13	spc_char(4) + upp_char(9)
25.	9	low_char(4) + num_char(3) + upp_char(2)
26.	7	num_char(5) + upp_char(2)
27.	8	low_char(4) + num_char(2) + spc_char(1) + upp_char(1)
28.	15	low_char(5) + num_char(3) + spc_char(2) + upp_char(5)
29.	7	num_char(3) + spc_char(2) + upp_char(2)
30.	13	low_char(8) + num_char(3) + upp_char(2)

Табл. 3

Варіант	**Об'єкт**	**Атрибути**
1	Автомобіль	Виробник, модель, рік випуску, пробіг, макс. Швидкість
2	Книга	Автор, назва, видавництво, рік видання, кількість сторінок
3	Пасажирський літак	Виробник, модель, рік випуску, кількість пасажирів, макс. Швидкість
4	Студент	ПІБ, група, рік вступу, середній бал
5	Користувач	Логін, пароль, телефон, е-мейл, кількість авторизацій
6	Фільм	Назва, режисер, рік випуску, бюджет, тривалість, розмір файлу
7	Пиво	Назва, виробник, міцність, ціна, термін зберігання в днях
8	Працівник	Назва фірми, посада, телефон, е-мейл, оклад
9	Пісня	Виконавець, назва, альбом, тривалість, розмір файлу
10	Смартфон	Виробник, модель, ціна, ємність батареї, обсяг пам'яті
11	Процесор	Виробник, модель, тактова частота, ціна, розсіювана потужність
12	Квартира	Власник, адреса, поверх, площа, кількість кімнат
13	Собака	Порода, кличка, вік, вага
14	Ноутбук	Виробник, процесор, ціна, діагональ, час роботи від акумулятора
15	Країна	Назва, столиця, населення, площа, ВВП
16	Нобелівський лауреат	Прізвище, рік народження, країна, рік вручення, галузь
17	Купюра	Валюта, номінал, рік випуску, курс до долара
18	Дисципліна	Назва, викладач, семестр, балів за поточний контроль, балів за екзамен
19	Річка	Назва, континент, довжина, басейн, середньорічний стік
20	Місто	Назва, країна, населення, рік заснування, площа

4. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Які переваги дає використання списку?
2. Як можна звернутися до окремого елементу списку?
3. Які можливі способи заповнення списку?

5. СПИСОК ЛІТЕРАТУРИ

1. Learn to Program with Python 3. A Step-by-Step Guide to Programming, Second Edition / Irv Kalb. – Mountain View: Apress, 2018. – 361 p.
2. The Python Workbook. A Brief Introduction with Exercises and Solutions, Second Edition / Ben Stephenson. – Cham: Springer, 2014. – 218 p.
3. Python Pocket Reference, Fifth Edition / Mark Lutz. – Sebastopol: O'Reilly Media, Inc., 2014. – 264 p.
4. Learn Python 3 the Hard Way / Zed A. Shaw. – Boston: Addison-Wesley, 2017. – 321 p.
5. A Python Book: Beginning Python, Advanced Python, and Python Exercises / Dave Kuhlman. – Boston: MIT, 2013. – 278 p.

НАВЧАЛЬНЕ ВИДАННЯ

Створення та використання функцій

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 7
з курсу «Програмування скриптовими мовами»
для студентів спеціальності
«Кібербезпека»

Укладач: Я. Р. Совин, канд. техн. наук, доцент