Лекція 16
Функції користувача

Контрольна робота №6 (визначення варіанту)

ІО	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
61	15	1	2	3	4	5	6	7	8	9	10	11	12	13	14
62	13	14	15	1	2	3	4	5	6	7	8	9	10	11	12
63	11	12	13	14	15	1	2	3	4	5	6	7	8	9	10
64	9	10	11	12	13	14	15	1	2	3	4	5	6	7	8
65	7	8	9	10	11	12	13	14	15	1	2	3	4	5	6

Червоний колір – номер у списку групи

Чорний та синій колір – номер варіанту

ІО	16	17	18	19	20	21	22	23	24	25	26	27	28	29
61	15	1	2	3	4	5	6	7	8	9	10	11	12	13
62	13	14	15	1	2	3	4	5	6	7	8	9	10	11
63	11	12	13	14	15	1	2	3	4	5	6	7	8	9
64	9	10	11	12	13	14	15	1	2	3	4	5	6	7
65	7	8	9	10	11	12	13	14	15	1	2	3	4	5

Приклад виконання контрольної роботи

Написати програму для отримання зазначеного результату:

Умова.

17	Дано: d1 = { "third": 12,"first": 13, "second": 21}. Результат: first-second-third-

Розв'язок.

```
for k in sorted(d1.keys()): print(k, end="-")
```

Написати програму для отримання зазначеного результату:

1.	Дано: d = {1: "int", "a": "str", 1.1: "float"}. Результат: ('float', 'str').
2.	Дано: d = { "a": 1, "b": 2}. Результат: True
3.	Дано: d = { "a": 1, "b": 2}. Результат: False
4.	Дано: d = { "a": 1, "b": 2}. Результат: None
5.	Дано: d = {"a": 1, "b": 2, "c": 3}. Результат: 1 {'c': 3, 'b': 2}
6.	Дано: d = { "a": 1, "b": 2}. Результат: 2
7.	Дано: d = { "a": 1, "b": 2}. Результат: {'a': 1}
8.	Дано: d = {"a": 1, "b": 2}. Результат: { }
9.	Дано: d = {"a": 1, "b": 2}. Результат: {'c': 3, 'b': 2, 'd': 4, 'a': 1}
10.	Дано: d1 = {"a": 1, "b": 2}. Результат: (dict_keys(['a', 'b'])
11.	Дано: d1 = {"a": 1, "b": 2}. Результат: a b
12.	Дано: d = {"a": 1, "b": 2}. Результат: [1, 2]
13.	Дано: d = {"a": 1, "b": 2}. Результат: dict_values([1, 2])
14.	Дано: d = {"a": 1, "b": 2}. Результат: [('a', 1), ('b', 2)]
15.	Дано: d = { "a": 1, "b": 2}. Результат: {'c': 'string', 'b': 2, 'a': 800}

Навіщо потрібні функції ?

Функція – це фрагмент коду, який можна викликати з будь-якого місця програми.

У попередніх лекціях ми вже багато раз використовували, так звані, вбудовані функції мови Python.
Наприклад:
Функція `len()` – дозволяє одержати кількість елементів послідовності.
Функція `str()` – перетворює будь-який об'єкт у рядок.
Функція `split()` – розділяє рядок на підрядки.

Розглянемо створення функцій користувача
Такі функції дозволяють:
зменшити надмірність програмного коду й
підвищити його структурованість.

Визначення функції і її виклик

Функцію створюють (або, як говорять програмісти, визначають) за допомогою ключового слова **def** за наступною схемою:

Приклад 1.

```
def <Ім'я функції> ([<Параметри>]):
    ["""  Рядок документування  """]
    <Тіло функції>
    [return <Результат>]
```

<Ім'я функції> повинно бути унікальним ідентифікатором, який формується за правилами:

1. Ім'я функції може містити:
 латинські букви, цифри, знаки підкреслення.
2. Ім'я функції не може починатися з цифри, не можна використовувати ключові слова, слід уникати збігів з назвами вбудованих ідентифікаторів.
3. Регістр символів у назві має значення.

Визначення функції (продовження)

Після імені записують параметри `([<Параметри>])` функції в круглих дужках.

1. Можна вказати один або кілька параметрів через кому (q,z)
2. Якщо функція не приймає параметри, вказують тільки круглі дужки ()
3. Після круглих дужок ставлять двокрапку.

```
def <Ім'я функції> ([<Параметри>]):
```

Наприклад: `def func(a,b):`

Тіло функції є складеною конструкцією.

1. Інструкції всередині функції виділяють однаковою кількістю пробілів ліворуч.
2. Кінцем функції вважається інструкція, перед якою перебуває менша кількість пробілів.
3. Якщо тіло функції не містить інструкцій, то усередині неї необхідно розмістити оператор `pass`, який не виконує ніяких дій.

Визначення функції (продовження)

Оператор `pass` – оператор зручно використовувати на етапі налагодження програми, коли функція визначена, а тіло вирішено дописати пізніше. Приклад функції, яка нічого не робить:

Приклад 2.

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

Необов'язкова інструкція `return` дозволяє повернути з функції яке-небудь значення як результат.

1. Після виконання цієї інструкції виконання функції буде зупинено.
2. Це означає, що інструкції, що слідують після оператора **`return`**, ніколи не будуть виконані.

Визначення функції (продовження)

Приклад 3.

```
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 усередині функції, і як результат повертається значення **None**.

Приклади визначення функцій.

Приклад 4.

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

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

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

Виклик функції

Виклик функції записують у форматі

```
<Ім'я функції> ([<Параметри>])
```

1. При виклику функції значення передають через параметри усередині круглих дужок

2. Параметри записуються через кому.

3. Якщо функція не приймає параметрів, то вказують тільки круглі дужки.

4. Кількість параметрів у визначенні функції повинна збігатися з кількістю параметрів при виклику, інакше буде виведене повідомлення про помилку.

Приклади виклику функцій, визначених у прикладі 4

Приклад 5.

```
print_ok()
Результати роботи:
Повідомлення при вдало виконаній операції

x = suma(5, 2)
print(x)
Результати роботи:
7

m="довільний рядок"
print(echo(m))
Результати роботи:
довільний рядок
None
```

Формальні та фактичні параметри

1. Ім'я змінної у виклику функції може не збігатися з іменем змінної у визначенні функції.

Приклад 6.

```
def echo(m) :
    print(m)

n="Параметр має інше ім’я"
echo(n)
```

```
Результати роботи:
Параметр має інше ім’я
```

Глобальні та локальні змінні

2. Крім того, *глобальні* змінні `x` и `y` не конфліктують з однойменними змінними у визначенні функції, оскільки вони розташовані в різних областях видимості.

Приклад 7.

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

x = 20
y = 45
z = 20 + 45
print(z)
a = 10
b = 11
print(suma(a,b))
Результат:
65
21
```

Функція може приймати ті типи значень, які відповідають операторам в її тілі

Оператор **+**, використовуваний у функції `suma()`, застосовується не тільки для додавання чисел, але й дозволяє об'єднати послідовності.

Тобто, функція `suma()` може використовуватися не тільки для додавання чисел.

Приклад 8.

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

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

Функція як об'єкт мови Python

У мові Python усі елементи є об’єктами:

рядки, списки, кортежі, типи даних 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 інший змінній – для цього назву функції вказують без круглих дужок.

Ми можемо зберігати це посилання в змінній і викликати функцію через неї.

Приклад 9. Збереження посилання на функцію в змінній

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

f = suma      # Зберігаємо посилання в змінній f
v = f(10, 20) # Викликаємо функцію через змінну f
print("10 + 20 =",v)
Результат роботи: 10 + 20 = 30
```

1. У цьому прикладі об'єкт типу `function` збережений у змінній `f` за допомогою інструкції:

```
f = suma
```

2. Змінну `f` можемо використовувати для виклику функції `suma`:

```
v = f(10, 20)
```

Посилання на функцію може бути передане як параметр в іншу функцію

Функції, передані за посиланням, зазвичай називають *функціями зворотного виклику*

Приклад 10. Функції зворотного виклику

```
def suma (z, y):
    """Функція suma()"""
    return z + y

def func(f):
    """ Через змінну f буде доступне посилання на функцію suma() """
    a = 10
    b = 20
    return f(a, b) # Викликаємо функцію suma()
# Передаємо посилання на функцію як параметр
v = func(suma)
print("suma=", v)
Результат роботи: suma= 30
```

Атрибути об'єкта типу `function`

Звернутися до атрибутів можна, указавши атрибут після назви функції через крапку.

Наприклад.

Через атрибут `_name_` можна одержати назву функції у вигляді рядка.

```
print(suma.__name__)
print(func.__name__)
```

```
suma
```

Через змінну f буде доступне посилання на функцію suma()

Через атрибут `_doc_` – рядок документування і т. д.

```
print(suma.__doc__)
print(func.__doc __)
```

```
Функція suma()
```

Через змінну f буде доступне посилання на функцію suma()

Приклад 11. Атрибути функції `summa`

```
>>> def suma(x, y):
    """ Додавання двох чисел """
    return x + y
```

Приклади виклику атрибутів функції

```
# Одержуємо ім'я функції
>>> suma.__name__
suma
# Одержуємо рядок документування
>>> suma.__doc__
Додавання двох чисел
# Одержуємо параметри функції
print(suma.__code__.co_varnames)
  ('x', 'y')
# Довідаємося, в якому модулі розташована функція
print(suma.__module__)
__main__
```

Розташування визначення функцій

1. Усі інструкції в програмі виконуються послідовно зверху вниз.
2. Тому, перш ніж використовувати в програмі ідентифікатор, його необхідно попередньо визначити, присвоївши йому значення.
3. Отже, визначення функції повинно бути розташоване перед викликом функції.

Приклад 12.

Правильно:

```
def difference (x, y) :
    return x - y
v = difference(20, 9)# Викликаємо після визначення.
print("20 - 9 =",v)
Результат роботи: 20 - 9 = 11
```

Неправильно:

```
v = difference(20, 9)  # Ідентифікатор ще не визначений.
```

Це помилка!!!

```
def difference(x, y):
    return x - y
```

Результат роботи:

```
Traceback (most recent call last):
  File "C:/PYTHON/Samples.py", line 1, in
<module>
    v = difference(20, 9)
Nameerror: name 'difference' is not defined
```

Щоб уникнути помилки,

визначення функції розміщають на самому початку програми після підключення модулів або в окремому модулі

Кілька функцій з однією назвою

За допомогою оператора розгалуження if можна вибирати необхідне визначення функції з однаковою назвою, але різною реалізацією.

Приклад 13.

```
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
```

Результат роботи1:

```
Введіть 1 для виклику першої функції: 1
Ви ввели число 1
```

Результат роботи2:

```
Введіть 1 для виклику першої функції: 0
Альтернативна функція
```

Випадок перевизначення функції

Пом’ятайте, що інструкція **def** усього лише присвоює посилання на об'єкт функції ідентифікатору, розташованому після ключового слова **def**.

Якщо визначення однієї функції зустрічається в програмі кілька раз, то буде використовуватися функція, яка була визначена останньою.

Приклад 14.

```
def echo():
    print("Ви ввели число 1")
def echo():
    print("Альтернативна функція")
echo() # Завжди виводить "Альтернативна функція"
```

```
Результат роботи: Альтернативна функція
```

Необов'язкові параметри й зіставлення по ключах

1. Щоб зробити деякі параметри необов'язковими, слід у визначенні функції присвоїти цьому параметру початкове значення.
2. Необов'язкові параметри повинні слідувати після обов'язкових параметрів, інакше буде виведене повідомлення про помилку.

Приклад 15. Необов'язкові параметри

```
def suma (x, y, z=2): #z-необов'язковий параметр
    return = x + y + z
a = suma(5,10)
print("a =",a)
b = suma (10, 50, 40)
print("b =",b)
```

Результат роботи: a = 17
b = 100

Таким чином, якщо третій параметр не заданий, то його значення буде дорівнювати 2.

Що таке позиційна передача параметрів?

Позиційна передача параметрів – це присвоєння значень параметрам функції в порядку, у якому вони задані при виклику функції.

Приклад 16. Позиційна передача параметрів

```
def forcalc(x,y,z):
    print("x =",x, "y =",y, "z =",z)
    return x*y+z

print("m =", forcalc(3,5,7))
```

```
Результат роботи: x = 3 y = 5 z = 7
                  m = 22
```

Змінній `x` при зіставленні буде присвоєно значення 3, змінній `y` – значення 5, а змінній `z` = 7 .

Передача параметрів зіставленням по ключах

1. При виклику функції параметрам можуть присвоюватися значення.

2. Послідовність вказівки параметрів у цьому випадку може бути довільною.

Приклад 17. Зіставлення по ключах

```
def forcalc(x,y,z):
    print("x =",x, "y =",y, "z =",z)
    return x*y+z

print("m =", forcalc(z=7, x=3, y=5))
```

```
Результат роботи: x = 3 y = 5 z = 7
                  m = 22
```

Зіставлення по ключах і необов'язкові параметри

Зіставлення по ключах дуже зручно використовувати, якщо функція має кілька необов'язкових параметрів.

У цьому випадку не потрібно перераховувати всі значення, а достатньо присвоїти значення потрібному параметру.

Приклад 18.

```
def suma(a=2, b=3, c=4): # Усі параметри необов'язкові
    return a + b + c
print("case1",suma())#За замовчуванням
print("case2",suma(2, 3, 20))#Позиційне присвоювання
print("case3",suma(c = 15))#Зіставлення по ключах
```

```
Результат роботи: case1 9
                  case2 25
                  case3 20
```

Розпакування списку або кортежу

Якщо значення параметрів, які планується передати у функцію, містяться в кортежі або списку, то перед об'єктом слід указати символ * (*розпакувати*).

Розглянемо передачу значень із кортежу й списку.

Приклад 19.

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

t1, arr = (1, 2, 3), [6, 7, 8]
print("case1",suma(*t1))   # Розпакували кортеж
print("case2",suma(*arr)) # Розпакували список
t2 = (4, 5) # Розпакування на два параметри з 3
print("case3",suma(1, *t2)) # Комбінувати
print("case4",suma(t1,t2,tuple(arr)))#without
```

Результат роботи:

```
case1 6
case2 21
case3 10
case4 (1, 2, 3, 4, 5, 6, 7, 8)
```

Розпакування словника

Якщо значення параметрів містяться в словнику, то розпакувати словник можна, указавши перед ним дві зірочки: **(* *).**

Приклад 20.

```
def suma(a, b, c):
    return a + b + c
d1 = {"a": 1, "b": 2, "c": 3}
print("dict", suma(**d1)) # Розпаковуємо словник
t, d2 = (1, 2), {"c": 3}
print("comb",suma(*t, **d2))#Можна комбінувати значення
```

```
Результат роботи: dict 6
                  comb 6
```

Передача у функцію незмінюваних об'єктів

Об'єкти у функцію передають за посиланням.

Якщо об'єкт є об’єктом незмінюваного типу, то зміна значення усередині функції не торкнеться значення змінної поза функцією:

Приклад 21.

```
def func(a, b):
    a, b = 20, "str"
    print("a =",a, "b =",b)
x, s = 80, "test"
func(x, s)
print("x =",x, "s =",s)
```

Результат роботи:
```
a = 20 b = str
x = 80 s = test
```

У цьому прикладі значення в змінних `x` і `s` не змінилися.

Однак якщо об'єкт є об’єктом змінюваного типу, то ситуація буде іншою:

Передача у функцію змінюваних об'єктів

Приклад 22.

```
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 =",x, "y =",y)

x = ['str', 2, 3] y = {'b': 2, 'a': 800}
```

Як видно з прикладу, значення в змінних `x` та `y` змінилися, оскільки список і словник є об’єктами змінюваних типів.

Як уникнути зміни об'єкта?

Якщо необхідно уникнути зміни значень, усередині функції слід створити копію об'єкта.

Приклад 23. Передача змінюваного об'єкта у функцію

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

x = [ 1, 2, 3] # Список
y= {"a": 1, "b": 2} # Словник
func(x, y) # Значення залишаться колишніми
print("x =",x, "y =",y)
Результат роботи:
a = ['str', 2, 3] b = {'a': 800, 'b': 2}
x = [1, 2, 3] y = {'a': 1, 'b': 2}
```

Можна також відразу передавати копію об'єкта у виклику функції:

Приклад 24.

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

x = [ 1, 2, 3] # Список
y = {"a": 1, "b": 2} # Словник
func(x[:], y.copy()) # Значення залишаться колишніми
print("x =",x, "y =",y)
Результат роботи:
a = ['str', 2, 3] c = {'a': 800, 'b': 2}
x = [1, 2, 3] y = {'a': 1, 'b': 2}
```

Значення змінюваного типу за замовчуванням

Якщо вказати об'єкт, що має змінюваний тип, як значення за замовчуванням, то цей об'єкт буде зберігатися між викликами функції.

Приклад 25.

```
def func(a = []):
    a.append(2)
    return a
print(func()) # Виведе: [2]
print(func()) # Виведе: [2, 2]
print(func()) # Виведе: [2, 2, 2]
```

```
Результат роботи: [2]
                  [2, 2]
                  [2, 2, 2]
```

Як видно з прикладу, значення накопичуються усередині списку.

Як уникнути нагромадження

Якщо необхідно уникнути накопичення, то можна зробити в такий спосіб:

Приклад 26.

```
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([1,3])) # Виведе: [1, 2]
print(func()) # Виведе: [2]
```

Результат роботи:

```
[2]
[1, 2]
[1, 3, 2]
[2]
```