Лекція 17
Функції користувача (продовження)

Змінне число параметрів у функції

1. Якщо перед параметром у визначенні функції вказати символ (*), то функції можна буде передати довільну кількість параметрів.
2. Усі передані параметри будуть об'єднані в кортеж. Для прикладу напишемо функцію додавання довільної кількості чисел.

Приклад 1. Збереження переданих даних у кортежі

```
def suma(*t):
    """ Приймаємо довільну кількість параметрів """
    res = 0
    for i in t:
        res += i
    return res

print(suma(10, 20, 30, 40, 50))
```

Результат роботи: 150

Приклад 2. Збереження кортежів

```
def f(*t): print(t)
d=(1,2,3)
f(*d)
```

Результат роботи: (1, 2, 3)

Обов'язкові параметри зі значеннями за замовчуванням

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

Приклад 3.

```
def suma(x, y=5, *t): # Комбінація параметрів
    res = x + y
    for i in t:
        res += i
    return res

print(suma(10))
# Перебираємо кортеж з переданими параметрами
print(suma(10, 20, 30, 40, 50))
```

```
Результат роботи: 15
                  150
```

Збереження іменованих параметрів у словнику

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

Приклад 4. Збереження переданих даних у словнику

```
def func(**d):
    func(a=1, b=2, c=3)
Результат: {'c': 3, 'a': 1, 'b': 2}
def func(**d): print(d)
func(**m)
m={"a":1,"b":2}
Результат:{'b':2, 'a':1}}
```

Комбінування (*) і (**)

При комбінуванні параметрів параметр з двома зірочками вказується останнім.

Якщо у визначенні функції вказується комбінація параметрів з однією зірочкою й двома зірочками, то функція прийме будь-які передані їй параметри.

Приклад 5. Комбінування параметрів

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

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

Передача параметрів тільки по іменах

У визначенні функції можна вказати, що деякі параметри передаються тільки по іменах.

Для цього параметри повинні вказуватися після параметра з однією зірочкою, але перед параметром з двома зірочками.

Іменовані параметри можуть мати значення за замовчуванням.

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

Приклад 6. `a, b` – тільки по іменах

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

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

1. У цьому прикладі змінна `t` прийме будь-яку кількість значень, які будуть об'єднані в кортеж.
2. Змінні `a` й `b` повинні передаватися тільки по іменах, причому змінній `a` обов'язково потрібно передати значення при виклику функції.

3. Змінна `b` має значення за замовчуванням, тому при виклику допускається не передавати їй значення, але якщо значення передається, то воно повинно бути зазначене після назви параметра й оператора **=**.
4. Змінна `d` прийме будь-яку кількість іменованих параметрів і збереже їх у словнику.
5. Хоча змінні `a` й `b` є іменованими, вони не потраплять у цей словник.
6. Параметра із двома зірочками може не бути у визначенні функції
7. Параметр з однією зірочкою при вказівці параметрів, переданих тільки по іменах, повинен бути присутнім обов'язково.
8. Якщо функція не повинна приймати змінну кількість параметрів, але необхідно використовувати змінні, передані тільки по іменах, то можна вказати тільки зірочку без змінної:

Приклад 7.

```
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!
```

```
Результат роботи:
35 10 1 10
10 2 5 10
1 2 1 2
7 8 1 10
```

1. У цьому прикладі значення змінним `x` и `y` можна передавати як по позиціях, так і по іменах.

2. Оскільки змінні мають значення за замовчуванням, допускається взагалі не передавати їм значень.

3. Змінні `a` й `b` розташовані після параметра з однією зірочкою, тому передати значення при виклику можна тільки по іменах.

4. Оскільки змінна `b` має значення за замовчуванням, допускається не передавати їй значення при виклику, а змінна `a` обов'язково повинна одержати значення, причому тільки по імені.

Анонімні функції

Крім звичайних, мова Python дозволяє використовувати анонімні функції, які також називають лямбда-функціями. Анонімну функцію описують за допомогою ключового слова `lambda` за наступною схемою:

lambda [<Параметр1>[,...,<Параметрn>]]:<значення, що повертається>

Після ключового слова `lambda` можна вказати <параметри>. У якості параметра <значення, що повертається> вказують вираз, результат виконання якого буде повернутий функцією.

Анонімна функція не має імені

Визначення анонімної функції повертає посилання на об'єкт-функцію, яку можна зберегти в змінній або передати як параметр в іншу функцію.

Викликати анонімну функцію можна, як і звичайну, за допомогою круглих дужок, усередині яких розташовані передані параметри.

Розглянемо приклад використання анонімних функцій.

Приклад 8. Приклад використання анонімних функцій

Визначення lambda-функцій

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

Виклик lambda-функцій

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

Результат роботи:

```
30
15
45
```

Обов’язкові та необов’язкові параметри

Як і у звичайних функціях, деякі параметри анонімних функцій можуть бути необов'язковими.

Для цього параметрам у визначенні функції присвоюється значення за замовчуванням.

Приклад 9. Необов'язкові параметри в анонімних функціях

Визначення lambda-функції

```
f = lambda x, y = 2: x + y
```

Виклик lambda-функції

```
print (f(5))   # Виведе: 7
print(f(5, 6))   # Виведе: 11
```

```
Результат роботи:
7
11
```

Передача анонімної функції, як параметра

Найчастіше анонімну функцію не зберігають у змінній, а відразу передають як параметр в іншу функцію.

Наприклад, метод списків `sort()` дозволяє вказати функцію користувача в параметрі `key`.

Приклад 10. Сортування без врахування регістру символів

```
arr=["anaconda", "Africa", "bee", "Brazil"]
arr.sort(key=lambda s: s.lower())
for i in arr:
print(i, end=" ")
Результат роботи: Africa anaconda bee Brazil
```

Функції-генератори

Функцією-генератором називають функцію, яка може повертати одне значення з декількох значень на кожній ітерації.

Призупинити виконання функції й перетворити функцію в генератор дозволяє ключове слово `yield` (вироблення продукції).

Приклад 11. Піднесення елементів послідовності до степеня

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

for n in func(10, 2):
    print(n, end=" ")
```

Результат роботи:

```
1 4 9 16 25 36 49 64 81 100
```

```
for n in func(10, 3):
    print(n, end=" ")
```

Результат роботи:

```
1 8 27 64 125 216 343 512 729 1000
```

Метод `_next_()` та функції-генератори

Коли значення закінчуються, метод викликає виключення `Stopiteration`.

Виклик методу `_next_()` у циклі `for` проводиться непомітно для нас.

Приклад 12. Використання методу `_next_()`

```
def func (x, y) :
    for i in range(1, x+1):
        yield i ** y
i = func(3, 3)# Функція як ітератор
print(i.__next__()) # Виведе: 1   (1 **3)
print(i.__next__()) # Виведе: 8   (2 **3)
print(i.__next__()) # Виведе: 27   (3 **3)
print(i.__next__()) # Виключення
Stopiteration
Результат роботи: 1 8 27
```

Відмінність звичайної функції та функції-генератора

За допомогою звичайних функцій ми можемо повернути всі значення відразу у вигляді списку.

За допомогою функцій-генераторів – тільки одне значення за раз.

Перевага. Ця особливість дуже корисна при обробці великої кількості значень, тому що не потрібно завантажувати увесь список зі значеннями.

Функція – генератор виконує обробку даних «на льоту»

Виклик функції-генератора з функції генератора

Для цього застосовується розширений синтаксис ключового слова `yield`:

```
yield from <Викликувана функція-генератор>
```

Розглянемо наступний приклад.

Нехай у нас є список чисел b, і нам потрібно одержати інший список, що включає числа в діапазоні від 1 до кожного із чисел у першому списку.

b=[2,3,4]

d=[1,2,1,2,3,1,2,3,4]

Щоб створити такий список, розглянемо код на прикладі.

Приклад 13. Виклик однієї функції-генератора з іншої (простий випадок)

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

b = [ 5, 10]
for i in gen([5, 10]): print(i, end = " ")
Результат роботи:
1 2 3 4 5 1 2 3 4 5 6 7 8 9 10
```

Помножимо числа списку на 2. Код, що виконує цю задачу, показаний на прикладі.

Приклад 14. Виклик однієї функції-генератора з іншої (складний випадок)

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

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

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

Результат роботи:

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

Декоратори функцій

Декоратори дозволяють змінити поведінку звичайних функцій – наприклад, виконати які-небудь дії перед та після виконанням функції. Створимо декоратор вручну

Приклад 15

```
def my_decorator(function_to_decorate):
    # В середені декоратор визначає функцію-обгортку
    # Ця функція дає можливість виконувати код до та після функції
    def wrapper_function():
        print("Тут пишемо код, до виклику функції")
        function_to_decorate() # Сама функція
        print("Тут пишумо код, що працює після виклику")
       # Повертаємо цю функцію
return wrapper_function
def unchangeable_function():
   print("Це незмінна функція")
unchangeable_function() # До декоратора
unchangeable_function = my_decorator(unchangeable_function)
unchangeable_function() #Після декоратора
```

Попередній приклад з синтаксисом декораторів

Приклад 16

```
def my_decorator(function_to_decorate):
    # В середені декоратор визначає функцію-обгортку
    # Ця функція дає можливість виконувати код до та після функції
    def wrapper_function():
        print("Тут пишемо код, до виклику функції")
        function_to_decorate() # Сама функція
        print("Тут пишемо код, що працює після виклику")
    # Повертаємо цю функцію
    return wrapper_function


@my_decorator
def unchangeable_function():
    print("Це незмінна функція")
unchangeable_function()
```

Інструкцію `unchangeable_function = my_decorator(unchangeable_function)` замінено на простішою: `@my_decorator`

Приклад 17. Декоратори функцій

```
def deco(f): # Функція-декоратор
    print ("Викликана функція func ()")
    return f # Повертаємо посилання на функцію
@deco
def func(x) :
    return "x = {0}".format(x)
print (func (10))
Результат роботи:
Викликана функція func ()
x = 10
```

У цьому прикладі перед визначенням функції `func( )` вказується назва функції `deco()` із символом @: `@deco`

Таким чином, функція `deco()` стає декоратором функції `func()`.

Як параметр функція-декоратор приймає посилання на функцію, поведінку якої необхідно змінити, і повинна повертати посилання на ту ж функцію або яку-небудь іншу. Наш попередній приклад еквівалентний наступному коду:

Приклад 18.

```
def deco(f):
    print("Викликана функція func()")
    return f

def func(x):
    return "x = {0}".format(x)
# викликаємо функцію func() через функцію deco()
print (deco(func)(10))
Результат роботи:
Викликана функція func ()
x = 10
```

Кілька декораторів

Перед визначенням функції можна вказати відразу кілька функцій-декораторів. Для прикладу обернемо функцію `func()` у два декоратори: `deco1()` і `deco2()`.

Приклад 19.

```
def deco1(f):
    print("Викликана функція deco1()")
    return f
def deco2(f):
    print("Викликана функція deco2()")
    return f

@deco1
@deco2
def func(x):
   return "x = {0}".format(x)
print(func(10))
```

Результат роботи:

```
Викликана функція deco2()
Викликана функція deco1()
x = 10
```

Використання двох декораторів еквівалентно наступному коду:

```
func = deco1(deco2(func))
```

Тут спочатку буде викликана функція `deco2()`, а потім функція `deco1()`. Результат виконання буде присвоєний ідентифікатору `func`.

Ще один приклад використання декораторів – виконання функції тільки при правильно введеному паролі.

Приклад 20. Обмеження доступу за допомогою декоратора

```
ps = input("Уведіть пароль: ")
def test_passw(p):
    def deco (f):
        if p == "universe": # Порівнюємо паролі
            return f
        else:
            return lambda: "Доступ закритий"
    return deco #Повертаємо функцію-декоратор

@test_passw(ps)
def func():
    return "Доступ відкритий"
print(func())   # Викликаємо функцію
```

Результат роботи:

Уведіть пароль: 1 Доступ закритий	Уведіть пароль: universe Доступ відкритий

У цьому прикладі після символу @ зазначено не посилання на функцію, а вираз, який повертає декоратор.

```
@test_passw(ps)
```

Іншими словами, декоратором є не функція `test passw()`, а результат її виконання (функція `deco()`).

```
def deco (f):
        if p == "universe": # Порівнюємо паролі
            return f
        else:
            return lambda: "Доступ закритий"
```

Якщо введений пароль є правильним, то виконається функція `func()`:

```
def func():
    return "Доступ відкритий"
```

а якщо ні, то буде виведений напис "`доступ закритий`", який повертає анонімна функція.

Рекурсія. Обчислення факторіала

Рекурсія – це можливість функції викликати саму себе. Рекурсію зручно використовувати для перебору об'єкта, що має заздалегідь невідому структуру, або для виконання невизначеної кількості операцій. Як приклад розглянемо обчислення факторіала

Приклад 21.

```
def factor(n):
    if n == 0 or n == 1: return 1
    else:
        return n * factor(n - 1)
while True:
    z = input("Введіть число: ")
    if z.isdigit():
        b=int(z)
        break
    else:
        print("Ви ввели не число!")
print(b,factor(b))
```

Результат роботи: 6 720

Втім, простіше за все для обчислення факторіала скористатися функцією `factorial ()` з модуля `math`.

Приклад 22.

```
>>> import math
>>> math.factorial(5)
120

>>> math.factorial(6)
720
```

Глобальні й локальні змінні

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

Приклад 23. Глобальні змінні

```
def func (glob2):
    print("Значення глобальної змінної glob1 =", glob1)
    glob2 += 10
    print("Значення локальної змінної glob2 =", glob2)
glob1, glob2 = 10, 5
func(77) # Викликаємо фунцію
print("Значення глобальної змінної glob2 =", glob2)
```

Результат виконання:

```
Значення глобальної змінної glob1 = 10
Значення локальної змінної glob2 = 87
Значення глобальної змінної glob2 = 5
```

Змінній `glob2` усередині функції присвоюється значення, передане при виклику функції.

У результаті створюється нова змінна з тим же ім'ям: `glob2`, яка є локальною.

Усі зміни цієї змінної усередині функції не торкнуться значення однойменної глобальної змінної.

Отже, *локальні змінні* – це змінні, що оголошуються усередині функцій.

Якщо ім'я локальної змінної збігається з іменем глобальної змінної, то всі операції усередині функції здійснюються з локальною змінною, а значення глобальної змінної не змінюється.

Локальні змінні видимі тільки усередині тіла функції.

Приклад 24. Локальні змінні

```
def func():
    local1 = 77 # Локальна змінна
    glob1 = 25  # Локальна змінна
    print("Значення glob1 усередині функції =", glob1)
glob1 = 10   # Глобальна змінна
func()      # Викликаємо функцію
print("Значення glob1 поза функцією =", glob1)
try:
    print(local1) # Викличе виключення Nameerror
except Nameerror: # Обробляємо виключення
    print("Змінна local1 не видима поза функцією")
```

Результат виконання:

Значення glob1 усередині функції = 25

Значення glob1 поза функцією = 10

Змінна local1 не видна поза функцією

Як видно з прикладу, змінна `local1`, оголошена усередині функції `func()`, недоступна поза функцією.

Оголошення усередині функції локальної змінної `glob1` не змінило значення однойменної глобальної змінної.

Якщо доступ до змінної проводиться до присвоювання значення

(навіть якщо існує однойменна глобальна змінна),

то буде викликане виключення `Unboundlocalerror`

Приклад 25. Помилка при доступі до змінної до присвоєння значення

```
def func():
    #Локальная 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 усередині функції, необхідно оголосити змінну глобальною за допомогою ключового слова **global**. Продемонструємо це на прикладі.

Приклад 26.

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

Таким чином, пошук ідентифікатора, використовуваного усередині функції, буде проводитися в наступному порядку:

1. Пошук оголошення ідентифікатора усередині функції (у локальній області видимості).
2. Пошук оголошення ідентифікатора в глобальній області.
3. Пошук у вбудованій області видимості (вбудовані функції, класи і т. д.).

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

Приклад 27.

```
x = 5
# Зберігається посилання, а не значення змінної x!!!

func = lambda: x

x = 80 # Змінили значення

print(func()) # Виведе: 80, а не 5
Результат виконання:80
```

Якщо необхідно зберегти саме поточне значення змінної, то можна скористатися способом, наведеним у прикладі.

Приклад 28. Збереження значення змінної

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

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

Зберегти поточне значення змінної можна також, указавши глобальну змінну як значення параметра за замовчуванням у визначенні функції.

Приклад 29. Збереження значення за допомогою параметра за замовчуванням

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

Одержати всі ідентифікатори і їх значення дозволяють наступні функції:

Функція `globals()` – повертає словник з глобальними ідентифікаторами;

Функція `locals()` – повертає словник з локальними ідентифікаторами.

Приклад 30.

```
def func():
    local1: = 54
    glob2 = 2
    print("Локальні ідентифікатори усередині функції")
    print(sorted(locals().keys()))
glob1, glob2 = 10, 88
func()
print ("Глобальні ідентифікатори поза функцією")
print(sorted(globals().keys()))
```

Результат виконання:

Локальні ідентифікатори усередині функції

['glob2', 'local1']

Глобальні ідентифікатори поза функцією

['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'func', 'glob1', 'glob2']

Вкладені функції

Одну функцію можна вкласти в іншу функцію, причому рівень вкладеності не обмежений. При цьому вкладена функція одержує свою власну локальну область видимості, має доступ до змінних, оголошених всередині функції, у яку вона вкладена (функції-предка). Розглянемо вкладення функцій на прикладі.

Приклад 31.

```
def func1 (b) :
    def func2():
        print(b)
    return func2
f1 = func1(10)
f2 = func1(99)
 f1()# Виведе: 10
 f2()# Виведе: 99
Результат виконання: 10 99
```

Розглянемо приклад

Тут ми визначили функцію `func1()`, що приймає один параметр `b`.

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

Усередині функції `func1()` визначена вкладена функція `func2()`.

```
def func2():
    print(b)
```

Результатом виконання функції `func1()` буде посилання на цю вкладену функцію.

```
return func2
```

Усередині функції `func2()` ми виконуємо вивід значення змінної `b`, яка є локальною у функції `func1()`.

```
print(b)
```

Розглядаємо області видимості

1. Глобальна область видимості
2. Локальна область видимості.
3. Вбудована область видимості.
4. Вкладена область видимості.

Порядок пошуку індентифікаторів:

1. Пошук всередині вкладеної функції
2. Пошук всередині функції-предка,
3. Пошук у функціях більш високого рівня
4. Пошук у глобальній і вбудованих областях видимості.

У нашому прикладі змінна `b` буде знайдена в області видимості функції `func1()`.

Слід враховувати, що зберігаються лише посилання на змінні, а не їхні значення в момент визначення функції.

Наприклад, якщо після визначення функції `func2()` надати нове значення змінній `b`, то буде використовуватися це нове значення:

Приклад 32.

```
def func1(b):
    def func2():
        print(b)
    b = 30
    return func2
f1 = func1(10)
f2 = func1(99)
f1 ()# Виведе: 30
f2 ()# Виведе: 30
Результат виконання: 30 30
```

Для збереження значення змінної при визначенні вкладеної функції, слід передати значення як значення за замовчуванням:

Приклад 33.

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

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

Зміна значення змінної `b`, оголошеної в функції `func1()` із вкладеної функції `func2()`.

Якщо в функції `func2()` присвоїти значення змінній `b`, то буде створена нова локальна змінна з таким же ім'ям.

Якщо усередині функції `func2()` оголосити змінну як глобальну й присвоїти їй значення, то зміниться значення глобальної змінної, а не значення змінної `b` усередині функції `func1()`.

Отже, жоден з вивчених раніше способів не дозволяє із вкладеної функції змінити значення змінної, оголошеної усередині функції-предка. Щоб розв'язати цю задачу, слід оголосити необхідні змінні за допомогою ключового слова `nonlocal`.

Приклад 34.

```
def func1(a):
    b = a
    def func2(c):
        nonlocal b  # Оголошуємо змінну як nonlocal
        print(b)  # Можемо змінити значення x в func1()
        b = c
    return func2
f = func1(10)
f(5) # Виведе: 10
f(12) # Виведе: 5
f(3) # Виведе: 12
```

Результат виконання: 10 5 12

При використанні ключового слова `nonlocal` слід пам'ятати, що змінна обов'язково повинна існувати усередині функції-предка. В протилежному випадку буде виведене повідомлення про помилку.

Анотації функцій

В Python 3 функції можуть містити анотації, які вводять новий спосіб документування.

Тепер у заголовку функції допускається:

-вказувати призначення кожного параметра,

-дані якого типу він може приймати,

-тип значення, що повертається функцією.

Анотації мають наступний формат:

```
def <Ім'я функції>(
[<Параметр1>[: <Вираз>] [= <Значення за замовчуванням>]
[,...<Параметрn>[: <Вираз>][= <Значення за замовчуванням>]]]
) -> < значення, що повертається>:
<Тіло функції>
```

У параметрах

<Вираз> і < значення, що повертається > можна вказати будь-який припустимий вираз мови Python.

Цей вираз буде виконано при створенні функції.

Приклад 35. Вказівка анотацій

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

Для змінної `a` створений опис `"Параметр1"`.

Для змінної `b` вираз `10 + 5` є описом.

Число `3` – значення параметра за замовчуванням.

Після закриваючої круглої дужки зазначений тип значення, що повертається функцією: `None`.

Атрибут __annotations__

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:

Приклад 36.

```
>>> def func(a: "Параметр1", b: 10 + 5 = 3)
-> None: pass
>>> func.__annotations__
{'b': 15, 'a': 'Параметр1', 'return': None}
```