Ключевые и позиционные аргументы

Ранее мы узнали, что функции имеют два типа аргументов:

- Ключевые аргументы (Keyword args)
- Позиционные аргументы (Position args)

kwargs - сокращенно keyword args. **args** - позиционные аргументы. Кортеж `args` и список `kwargs` хранят остаточные аргументы - те, которые не имеются в определении оригинальной функции и упакуются посредством `args` или `kwargs`. В коде ниже остаточным аргументом будет являться `arg3`

In [1]:
```
call_number = 0

def get_self_args(arg1: int, arg2: int, *args, **kwargs) -> str:
    global call_number
    call_number += 1
    return f'Call {call_number}. Arg1: {arg1}, Arg2: {arg2}, args: {args}, kwargs:


print(get_self_args(arg1=1, arg2=2, arg3=3))
print(get_self_args(1, 2, 3))
```

```
Call 1. Arg1: 1, Arg2: 2, args: (), kwargs: {'arg3': 3}
Call 2. Arg1: 1, Arg2: 2, args: (3,), kwargs: {}
```

Как мы видим, любой аргумент можно передать и по позиции и по имени. В определённых случаях мы хотим, чтобы некоторые аргументы могли быть переданы только по позиции, а какие-то только по имени. Для этого существуют операторы `/` и `*`. Вот как это работает:

- Параметры, указанные перед `/`, являются исключительно позиционными. Это значит, что их можно указывать только по порядку, не используя имена параметров при вызове функции.
- Параметры, указанные после `*`, являются только ключевыми. Это значит, что их можно передать в функцию только по имени, но не по позиции.

In [2]:
```
def function(arg1, arg2, /, *, kwarg1, kwarg2):
    print(f"arg1 = {arg1}, arg2 = {arg2}, kwarg1 = {kwarg1}, kwarg2 = {kwarg2}")

# Правильный вызов функции:
function(1, 2, kwarg1='a', kwarg2='b')  # arg1 и arg2 переданы позиционно, kwarg1 и
```

```
arg1 = 1, arg2 = 2, kwarg1 = a, kwarg2 = b
```

In [3]:
```
# Неправильные вызовы функции:
function(arg1=1, arg2=2, kwarg1='a', kwarg2='b')  # Ошибка, arg1 и arg2 должны быть
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 2
      1 # Неправильные вызовы функции:
----> 2 function(arg1=1, arg2=2, kwarg1='a', kwarg2='b')

TypeError: function() got some positional-only arguments passed as keyword argument
s: 'arg1, arg2'
```

In [ ]:
```
function(1, 2, 'a', 'b')  # Ошибка, kwarg1 и kwarg2 должны быть переданы как ключев
```

Теперь может возникнуть вопрос: синтаксис есть, а какое применение?

Функции-декораторы

Декоратор — это мощный и полезный инструмент, позволяющий модифицировать поведение функций или классов без изменения их кода. Декораторы предоставляют простой способ применить обертку (wrapper) вокруг функции или метода, что позволяет выполнять дополнительный код до и после основной логики функции, не затрагивая её непосредственно.

Также декоратор — это функция, которая принимает другую функцию в качестве аргумента и возвращает новую функцию, обогащённую дополнительным функционалом. В данном случае наш декоратор после выполнения каждой функции выводит время, за которое она была выполнена.

In [4]:
```
import time
from typing import Callable
from functools import wraps

def time_execution(func: Callable) -> Callable:
    @wraps(func)
    def wrapper(*args, **kwargs): # *args и **kwargs упаковывают все аргументы пере
        start_time = time.time()
        result = func(*args, **kwargs) # Далее упакованные аргументы мы распакуем в
        end_time = time.time()
        print(f"Функция {func.__name__} выполнялась {end_time - start_time} секунд.
        return result # После выполненных действий мы возвращаем результат выполнен
    return wrapper

def sum_sequence(x: int) -> int:
    return sum(i for i in range(x))

def square_sequence(x: int) -> list[int]:
    return [i ** 2 for i in range(x)]

sum_sequence = time_execution(sum_sequence)
square_sequence = time_execution(square_sequence)

sum_sequence(100000)
square_sequence(100000);
```

```
Функция sum_sequence выполнялась 0.0218808650970459 секунд.
Функция square_sequence выполнялась 0.007247209548950195 секунд.
```

Декораторы используются для того, чтобы уменьшить повторяющийся код в множестве схожих функций. Ведь без использования декораторов в каждой из функций нам пришлось повторять идентичный код перед её выполнением и после, чтобы получить желаемый результат.

Вместо того, чтобы декорировать функции прямым вызовом:

```
sum_sequence(100000)
square_sequence(100000);
```

В Python существует синтаксис для применения декораторов сразу при объявлении функции - `@<function_name>` . Пример:

In [5]:
```
@time_execution
def sum_sequence(x: int) -> int:
    """
    Возвращает сумму чисел от 1 до x
    :param x: конечное число
    :return: Сумма чисел
    """
    return sum(i for i in range(x))

@time_execution
def square_sequence(x: int) -> list[int]:
    """
    Возвращает список квадратов чисел от 1 до x
    :param x: конечное число
    :return: Список квадратов чисел от 1 до x
    """
    return [i ** 2 for i in range(x)]

sum_sequence(100000)
square_sequence(100000);
```

```
Функция sum_sequence выполнялась 0.0211789608001709 секунд.
Функция square_sequence выполнялась 0.00619196891784668 секунд.
```

Обратим внимание на `@wraps(func)` в:

```
def time_execution(func: Callable) -> Callable:
    @wraps(func)
    def wrapper(*args, **kwargs): # *args и **kwargs упаковывают все
аргументы передаваемые в декорируемую функцию
        start_time = time.time()
        result = func(*args, **kwargs) # Далее упакованные аргументы мы
распакуем в объект оригинальной(декорируемой) функции
        end_time = time.time()
        print(f"Функция {func.__name__} выполнялась {end_time -
start_time} секунд.")
        return result # После выполненных действий мы возвращаем
результат выполнения функции
    return wrapper
```

Эта строчка позволяет сохранить имя. документацию и другие магические аттрибуты оригинальной функции. Иначе, так как нам возвращается новый объект функции - wrapper, результирующая функция будем иметь отличное имя и строки документации.

In [6]:
```
def time_execution(func: Callable) -> Callable:
    def wrapper(*args, **kwargs): # *args и **kwargs упаковывают все аргументы пере
        start_time = time.time()
        result = func(*args, **kwargs) # Далее упакованные аргументы мы распакуем в
        end_time = time.time()
        print(f"Функция {func.__name__} выполнялась {end_time - start_time} секунд.
        return result # После выполненных действий мы возвращаем результат выполнен
    return wrapper

sum_sequence = time_execution(sum_sequence)
print(sum_sequence.__name__, end=' : ') # wrapper - хотя ожидается sum_sequence
print(sum_sequence.__doc__) # None, хотя ожидается: "Возвращает сумму чисел от 1 до

def time_execution(func: Callable) -> Callable:
    @wraps(func)
    def wrapper(*args, **kwargs): # *args и **kwargs упаковывают все аргументы пере
        start_time = time.time()
        result = func(*args, **kwargs) # Далее упакованные аргументы мы распакуем в
        end_time = time.time()
        print(f"Функция {func.__name__} выполнялась {end_time - start_time} секунд.
        return result # После выполненных действий мы возвращаем результат выполнен
    return wrapper

square_sequence = time_execution(square_sequence)
print(square_sequence.__name__, end=' : ')
print(square_sequence.__doc__)
```

```
wrapper : None
square_sequence :
    Возвращает список квадратов чисел от 1 до x
    :param x: конечное число
    :return: Список квадратов чисел от 1 до x
```

Как же связано применение декораторов и операторов `/` , `*` ? К примеру у нас есть те же функции: `square_sequence` и `sum_sequence` . Мы также хотим выводить время их выполнения, но не всегда. Только если аргумент нашей функции - `print_time` будет равен `True` . Иначе выводить не будем. Внутри декоратора нам нужно будет получать значение этого аргумента и выполнять его код в зависимости от значения.

Чтобы узнать аргументы декорируемой функции мы можем перебрать кортеж `*args` или список `**kwargs` , в зависимости от того каким образом был передан аргумент - как остаточный(тот, который упакуется в `*args` или `**kwargs` и не имеется в определении самой функции) или как обычный. То есть нам придется перебрать кортеж и список из-за отсутствия знаний о том, каким образом пользователь данный аргумент передал.

In [7]:
```
def time_execution(func: Callable) -> Callable:
    @wraps(func)
    def wrapper(*args, **kwargs): # *args и **kwargs упаковывают все аргументы пере
        if kwargs.get('print_time') is None: # Если аргумент print_time в ключевых
            print_time = args[1]
        else:  # Если найден, то берём его значение по ключу
            print_time = kwargs['print_time']

        if print_time:
            start_time = time.time()

        result = func(*args, **kwargs) # Далее упакованные аргументы мы распакуем в

        if print_time:
            end_time = time.time()
            print(f"Функция {func.__name__} выполнялась {end_time - start_time} сек
        return result # После выполненных действий мы возвращаем результат выполнен
    return wrapper

@time_execution
def sum_sequence(x: int, **kwargs) -> int:
    return sum(i for i in range(x))

@time_execution
def square_sequence(x: int, print_time: bool = False) -> list[int]:
    return [i ** 2 for i in range(x)]

sum_sequence(10000, print_time=True)
square_sequence(10000, False);
```

```
Функция sum_sequence выполнялась 0.0010738372802734375 секунд.
```

Это неверный подход именно из-за этой строчки

```
print_time = args[1]
```

Если мы создадим функцию, которая принимает два позиционных аргумента и попытаемся передать `print_time` как третий позиционный, то за основу возьмётся булево значение второго аргумента, а не `print_time` .

In [8]:
```
@time_execution
def pow_sequence(x: int, y: int, print_time: bool = False) -> list[int]:
    return [i ** y for i in range(x)]

pow_sequence(2, 3, False)
```

```
Функция pow_sequence выполнялась 0.0 секунд.
```
Out[8]:
```
[0, 1]
```

`bool(3) == True` следственно код не выполнился как ожидается. Чтобы предотвратить такие ситуации, мы можем "вынудить" пользователя использовать `print_time` только как ключевой аргумент(оператор `*`) или же только как позиционный, стоящий на третьей позиции(оператор `\`). В данном случае удобнее использовать первый вариант.

In [9]:
```
def time_execution(func: Callable) -> Callable:
    @wraps(func)
    def wrapper(*args, **kwargs): # *args и **kwargs упаковывают все аргументы пере
        print_time = kwargs.get('print_time') # Значение аргумента print_time тепер

        if print_time:
            start_time = time.time()

        result = func(*args, **kwargs) # Далее упакованные аргументы мы распакуем в

        if print_time:
            end_time = time.time()
            print(f"Функция {func.__name__} выполнялась {end_time - start_time} сек
        return result # После выполненных действий мы возвращаем результат выполнен
    return wrapper

@time_execution
def pow_sequence(x: int, y: int, *, print_time: bool = False) -> list[int]:
    return [i ** y for i in range(x)]
```

In [10]:
```
print('Call 1', pow_sequence(2, 3, print_time=False))
```

```
Call 1 [0, 1]
```

In [11]:
```
print('Call 2', pow_sequence(2, 3, print_time=True))
```

```
Функция pow_sequence выполнялась 0.0 секунд.
Call 2 [0, 1]
```

In [12]:
```
print('Call 3', pow_sequence(2, 3, True))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 print('Call 3', pow_sequence(2, 3, True))

Cell In[9], line 9, in time_execution.<locals>.wrapper(*args, **kwargs)
      6 if print_time:
      7     start_time = time.time()
----> 9 result = func(*args, **kwargs) # Далее упакованные аргументы мы распакуем в
объект оригинальной(декорируемой) функции
     11 if print_time:
     12     end_time = time.time()

TypeError: pow_sequence() takes 2 positional arguments but 3 were given
```