Аргументы функции

В Python функции могут принимать аргументы, которые передаются при вызове функции. Аргументы могут быть обязательными или необязательными, позиционными или именными. В этой лекции мы рассмотрим подробнее различные способы передачи аргументов в функцию и как использовать их в своем коде.

1. Обязательные и необязательные аргументы

Аргументы в функции могут быть обязательными или необязательными. Обязательные аргументы должны быть переданы при вызове функции, в противном случае будет выдана ошибка. Необязательные аргументы имеют значение по умолчанию, которое будет использовано, если аргумент не будет передан при вызове функции.

Например, вот функция с одним обязательным аргументом `x` и одним необязательным аргументом `y` со значением по умолчанию `1`:

```
def multiply(x, y=1):
    return x * y
```

При вызове функции `multiply(5)` будет возвращено значение `5`, так как значение по умолчанию `1` будет использовано для аргумента `y`. При вызове функции `multiply(5, 3)` будет возвращено значение `15`, так как аргумент `y` будет иметь значение `3`.

2. Позиционные и именные аргументы

Аргументы в функции могут быть позиционными или именными. Позиционные аргументы передаются в функцию в том же порядке, в котором они определены в определении функции. Именные аргументы передаются в функцию с указанием их имени и значения.

Например, вот функция с двумя позиционными аргументами `x` и `y`:

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

При вызове функции `add(2, 3)` будет возвращено значение `5`, так как первый аргумент `2` будет соответствовать `x`, а второй аргумент `3` будет соответствовать `y`.

Вот тот же пример с использованием именных аргументов:

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

При вызове функции `add(y=3, x=2)` будет возвращено значение `5`, так как аргумент `x` будет иметь значение `2`, а аргумент `y` будет иметь значение `3`.

3. Произвольное количество аргументов

В Python можно определить функцию, которая принимает произвольное количество аргументов. Для этого используется специальный синтаксис `*args` для позиционных аргументов и `**kwargs` для именных аргументов.

Например, вот функция, которая принимает произвольное количество позиционных аргументов:

```
def sum_numbers(*args):
    return sum(args)
```

При вызове функции `sum_numbers(1, 2, 3, 4, 5)` будет возвращено значение `15`, так как все аргументы будут переданы в функцию `sum` в виде кортежа `(1, 2, 3, 4, 5)`.

Вот функция, которая принимает произвольное количество именных аргументов:

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

При вызове функции `print_info(name="Alice", age=25, city="New York")` будет выведено:

```
name: Alice
age: 25
city: New York
```

Так как все аргументы будут переданы в функцию `print_info` в виде словаря `{"name": "Alice", "age": 25, "city": "New York"}`.

4. Распаковка аргументов

В Python можно использовать распаковку аргументов, чтобы передать список или кортеж аргументов в функцию как отдельные аргументы. Для этого используется синтаксис `*` перед именем списка или кортежа.

Например, вот функция, которая принимает два аргумента:

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

При вызове функции `add(*[2, 3])` будет возвращено значение `5`, так как список `[2, 3]` будет распакован в отдельные аргументы `2` и `3`.

То же самое можно сделать с именными аргументами, используя синтаксис `**` перед именем словаря.

Например, вот функция, которая принимает два именных аргумента:

```
def print_info(name, age):
    print(f"Name: {name}")
    print(f"Age: {age}")
```

При вызове функции `print_info(**{"name": "Alice", "age": 25})` будет выведено:

```
Name: Alice
Age: 25
```

Так как словарь `{"name": "Alice", "age": 25}` будет распакован в отдельные именные аргументы `name="Alice"` и `age=25`.

5. Замыкания

Замыкание в Python — это функция, определенная внутри другой функции, которая имеет доступ к переменным внешней функции. Замыкания можно использовать для создания функций с сохраненным состоянием.

Например, вот функция, которая возвращает другую функцию, которая увеличивает входное значение на некоторое сохраненное значение:

```
def make_adder(n):
    def adder(x):
        return x + n
    return adder
```

При вызове функции `make_adder(5)` будет возвращена функция `adder`, которая увеличивает входное значение на `5`.
При вызове функции `make_adder(10)` будет возвращена функция `adder`, которая увеличивает входное значение на `10`.

Например:

```
add_five = make_adder(5)
add_ten = make_adder(10)
print(add_five(3))  # Выведет: 8
print(add_ten(3))  # Выведет: 13
```

Так как функция `make_adder` возвращает замыкание `adder`, которое сохраняет значение `n` из внешней функции `make_adder`.

Выводы

В этой лекции мы рассмотрели различные способы передачи аргументов в функцию в Python. Мы узнали о обязательных и необязательных аргументах, позиционных и именных аргументах, произвольном количестве аргументов, распаковке аргументов и замыканиях. Эти концепции являются фундаментальными для понимания работы функций в Python и их использования в реальных проектах.

Функция может возвращать несколько значений в Python с помощью кортежа. Просто включите значения, которые вы хотите вернуть, в круглые скобки, и они будут объединены в кортеж. Затем вы можете назначить этот кортеж переменным, чтобы распаковать значения.

Вот пример функции, которая возвращает два значения:

```
def divide(x, y):
    quotient = x / y
    remainder = x % y
    return quotient, remainder
```

Эта функция принимает два аргумента `x` и `y`, вычисляет частное и остаток от деления `x` на `y`, и возвращает их в виде кортежа `(quotient, remainder)`.

Вы можете вызвать эту функцию и распаковать возвращенные значения в переменные, используя следующий синтаксис:

```
quotient, remainder = divide(10, 3)
print(quotient)  # Выведет: 3.3333333333333335
print(remainder)  # Выведет: 1
```

Вы также можете назначить возвращенный кортеж целиком переменной, если вам не нужно распаковывать значения:

```
result = divide(10, 3)
print(result)  # Выведет: (3.3333333333333335, 1)
```

Если функция возвращает несколько значений, вы можете использовать их в выражениях или в качестве аргументов для других функций, распаковывая их с помощью синтаксиса `*`:

```
quotient, remainder = divide(10, 3)
print(sum((quotient, remainder)))  # Выведет: 4.333333333333333
```

Вы также можете использовать распаковку аргументов, чтобы передать несколько возвращенных значений в качестве аргументов для другой функции:

```
def add_quotient_and_remainder(x, y):
    quotient, remainder = divide(x, y)
    return quotient + remainder

result = add_quotient_and_remainder(10, 3)
print(result)  # Выведет: 4.333333333333333
```

В этом примере мы определили функцию `add_quotient_and_remainder`, которая вызывает функцию `divide` и распаковывает возвращенные значения в переменные `quotient` и `remainder`. Затем она возвращает сумму этих

значений. Мы вызываем функцию `add_quotient_and_remainder` с аргументами `10` и `3`, и она возвращает `4.333333333333333`.

Выводы

В Python функция может возвращать несколько значений, объединив их в кортеж. Вы можете распаковать возвращенные значения в переменные, используя синтаксис распаковки, или использовать их в выражениях или в качестве аргументов для других функций, распаковывая их с помощью синтаксиса `*`. Это полезная особенность, которая позволяет вам возвращать несколько результатов из функции и использовать их в вашем коде.

В Python, вы можете разделить обязательные именованные атрибуты в функции от необязательных позиционных атрибутов, используя символ `*` как разделитель. Все аргументы до символа `*` будут считаться обязательными позиционными аргументами, а аргументы после символа `*` будут считаться обязательными именованными аргументами. Необязательные позиционные аргументы должны иметь значения по умолчанию.

Вот пример функции с обязательными именованными атрибутами и необязательными позиционными атрибутами:

```
def example_function(pos_arg1, pos_arg2, *, named_arg1, named_arg2, optional_pos_arg=None):
    print(f"pos_arg1: {pos_arg1}")
    print(f"pos_arg2: {pos_arg2}")
    print(f"named_arg1: {named_arg1}")
    print(f"named_arg2: {named_arg2}")
    print(f"optional_pos_arg: {optional_pos_arg}")
```

В этом примере `pos_arg1` и `pos_arg2` являются обязательными позиционными аргументами, `named_arg1` и `named_arg2` являются обязательными именованными аргументами, а `optional_pos_arg` является необязательным позиционным аргументом с значением по умолчанию `None`.

Вы можете вызвать эту функцию следующим образом:

```
example_function(1, 2, named_arg1="value1", named_arg2="value2", optional_pos_arg="optional_value")
```

В этом вызове `1` и `2` передаются как обязательные позиционные аргументы, `named_arg1="value1"` и `named_arg2="value2"` передаются как обязательные именованные аргументы, а `optional_pos_arg="optional_value"` передается как необязательный позиционный аргумент.

Если вы попробуете вызвать функцию без обязательных именованных аргументов, то получите ошибку:

```
example_function(1, 2)
```

Вывод:

```
TypeError: example_function() missing 2 required keyword-only arguments: 'named_arg1' and 'named_arg2'
```

Если вы попробуете вызвать функцию с необязательным позиционным аргументом, но без значения, то значение по умолчанию будет использовано:

```
example_function(1, 2, named_arg1="value1", named_arg2="value2")
```

Вывод:

```
pos_arg1: 1
pos_arg2: 2
named_arg1: value1
named_arg2: value2
optional_pos_arg: None
```

Практика:

Задача 1:

Напишите функцию `info_kwargs`, которая принимает произвольное количество именованных аргументов.

Функция `info_kwargs` должна распечатать именованные аргументы в каждой новой строке в виде пары <Ключ> = <Значения>, причем ключи должны следовать в алфавитном порядке. Пример работы смотрите ниже

```
info_kwargs(first_name="John", last_name="Doe", age=33) """ данный вызов печатает следующие строки age = 33 first_name = John last_name = Doe """
```

Вам необходимо написать только определение функции

Решение:

```
def info_kwargs(**kwargs):
    for k, v in sorted(kwargs.items()):
        print(f'{k} = {v}')
```

Задача 2:

Давайте теперь создадим функцию `print_goods`, которая печатает список покупок. На вход она будет принимать произвольное количество значений, а товаром мы будем считать любые непустые строки. То есть числа, списки, словари и другие нестроковые объекты вам нужно будет проигнорировать. Функция `print_goods` должна печатать список товаров в виде: <Порядковый номер товара>. <Название товара> (см. пример ниже). В случае, если в переданных значениях не встретится ни одного товара, необходимо распечатать текст "Нет товаров"

```
print_goods('apple', 'banana', 'orange')
# Программа должна вывести следующее:
1. apple
2. banana
3. orange
```

```
print_goods(1, True, 'Грушечка', '', 'Pineapple')
# Программа должна вывести следующее:
1. Грушечка
2. Pineapple
```

```
print_goods([], {}, 1, 2)
# Программа должна вывести следующее:
Нет товаров
```

Вам необходимо написать только определение функции `print_goods`

Решение:

```
def print_goods(*words):
    i = 0
    for word in words:
        if isinstance(word, str) and len(word) != 0:
            print(f'{i + 1}. {word}')
            i += 1
    if i == 0:
        print('Нет товаров')
```

Задача 3:

Напишите функцию `create_actor`, которая принимает произвольное количество именованных аргументов и возвращает словарь с характеристиками актера. Если функции `create_actor` не передавать никаких аргументов, то она должна возвращать базовый словарь с ключами `name`, `surname`, `age`. Вот так он выглядит:

```
create_actor() → {
        'name': 'Райан',
        'surname': 'Рейнольдс',
        'age': 46,
    }
```

Если передавать именованные параметры, которые отсутствуют в базовом словаре, они дополняются к этому словарю

```
create_actor(height=190, movies=['Дедпул', 'Главный герой']) ⇒ {
    'name': 'Райан',
    'surname': 'Рейнольдс',
    'age': 46,
    'height': 190,
    'movies': ['Дедпул', 'Главный герой']
}
```

Если передавать именованные параметры, которые совпадают с ключами базового словаря, то значения в словаре должны заменяться переданными значениями:

```
create_actor(name='Jack', age=20) → {
        'name': 'Jack',
        'surname': 'Рейнольдс',
        'age': 20,
    }
```

Вам необходимо написать только определение функции `create_actor`

Решение:

```python
def create_actor(**kwargs):
    base_dict = {
        'name': 'Райан',
        'surname': 'Рейнольдс',
        'age': 46,
    }
    base_dict.update(kwargs)
    return base_dict
```