Тема: Python. Функции

Что такое функции?

- **Функции** - это удобный способ разделить код на полезные блоки, позволяя его упорядочить и сделать более читабельным, повторно использовать его и сэкономить некоторое время.

- Функции в Python определяются с помощью ключевого слова "`def`", за которым следует имя функции в качестве имени блока.

Правила для создания функций

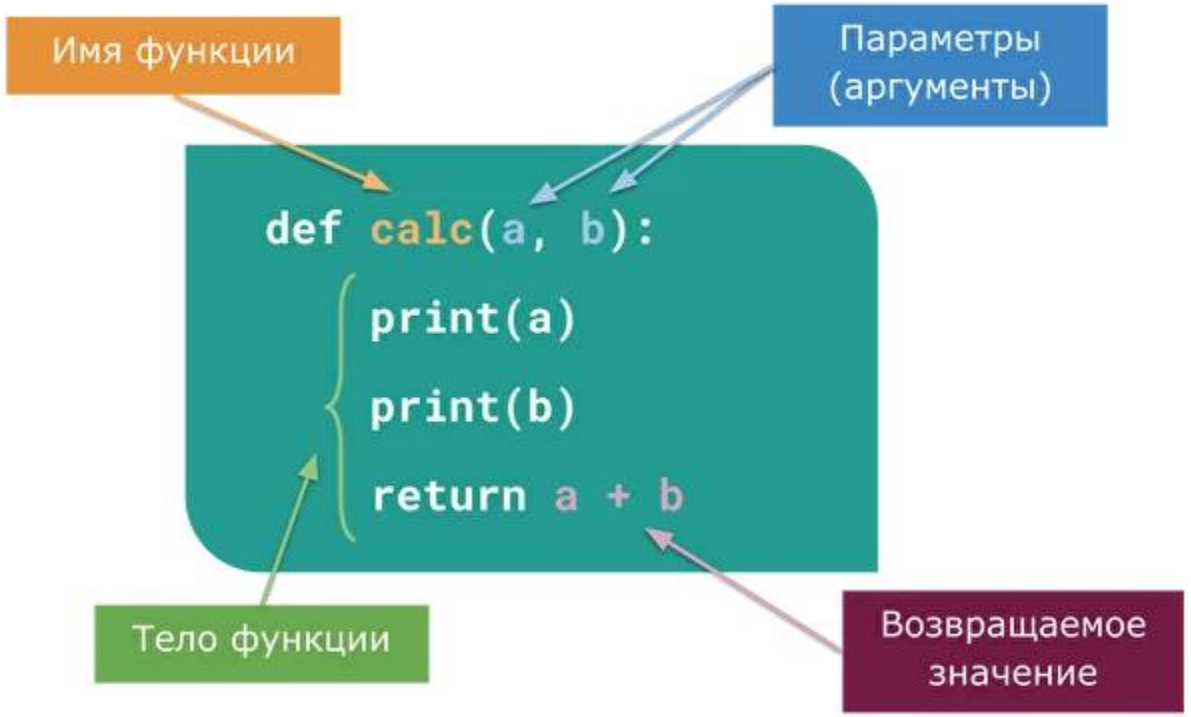
Существуют некоторые правила для создания функций в Python.

1. Блок функции начинается с ключевого слова **def**, после которого следуют название функции и круглые скобки **()**.
2. Любые аргументы, которые принимает функция, должны находиться внутри этих скобок.
3. После скобок идет двоеточие **:** и с новой строки с отступом начинается тело функции.

Т. е. функция определяется следующим образом:

```
def <имя_функции>(<аргументы функции>) :
```

Функция в Python. Синтаксис



Функции

Пример простой функции выводящий текст

```
def my_function():
    print("Это новая функция!")

# обращение к функции в программе
my_function()
```

Функции

Пример простой функции с параметрами

```
def sum_two_numbers(a, b):
    return a + b

# значению x передаем результат работы функции
x = sum_two_numbers(1,2)
```

Пример простой функции сложения двух чисел

```
def calc(a, b):
    print(a)
    print(b)
    return a + b

calc(5, 15)

sum = calc(50, 50)
print(sum)
```

```
5
15
50
50
100
```

Функции. Нахождение максимального значения двух чисел

```
def m_max(a, b):
    if a > b:
        return a
    else:
        return b

print(m_max(3, 5))
print(m_max(8, 3))
```

Вывод

5
8

Аргументы функции

Аргументы функции

- Вызывая функцию, мы можем передавать ей следующие типы аргументов:
  1. **Обязательные аргументы** (Required arguments)
  2. **Аргументы-ключевые слова** (Keyword arguments)
  3. **Аргументы по-умолчанию** (Default arguments)
  4. **Аргументы произвольной длины** (Variable-length argumens)

1. Обязательные аргументы

- **1. Обязательные аргументы**
- Если при создании функции мы указали количество передаваемых ей аргументов и их порядок, то и вызывать ее мы должны с тем же количеством аргументов, заданных в нужном порядке.

1. Обязательные аргументы. Пример

```
# Определим функцию hour_to_sec
# Она переводит часы в секунды
def hour_to_sec(hour, min, sec):
    return hour * 60 * 60 + min * 60 + sec

# Вызовем функцию. Количество и порядок аргументов очень важны!
# Иначе результат вычислений будет неверным
hour_to_sec(0, 5, 50)
```

350

2. Аргументы-ключевые слова

- **2. Аргументы-ключевые слова**
- Аргументы-ключевые слова используются при вызове функции. Благодаря ключевым аргументам, вы можете задавать произвольный (то есть не такой, каким он описан при создании функции) порядок аргументов.

2. Аргументы-ключевые слова. Пример

```
# Используем ту же самую функцию
def hour_to_sec(hour, min, sec):
    return hour * 60 * 60 + min * 60 + sec

# Хотя в определении первым параметром идут часы,
# мы можем передать секунды в качестве первого аргумента.
# В таком случае мы обязаны указать имя параметра
hour_to_sec(sec=50, hour=0, min=5)
```

3. Аргументы по-умолчанию

- **3. Аргументы по-умолчанию**
- Аргумент по умолчанию, это аргумент, значение для которого задано изначально, при создании функции.
- Если при вызове функции вы не будете передавать данный аргумент, то функция возьмет его значение по-умолчанию.

3. Аргументы по-умолчанию. Пример

```
# Функция принимает два параметра:
имя и возраст
# Параметр age имеет значение по-умолчанию
def person(name, age=25):
    print(name, '-', age, ' лет ')

# Передадим функции оба параметра и посмотрим
результат
person('Иван', 19)

# Теперь передадим функции только 1 параметр
# Параметр age примет свое значение
по-умолчанию
person('Петр')
```

```
Иван - 19 лет
Петр - 25 лет
```

4. Аргументы произвольной длины

- **4. Аргументы произвольной длины**
- Иногда возникает ситуация, когда вы заранее не знаете, какое количество аргументов будет необходимо принять функции.
- В этом случае следует использовать аргументы произвольной длины.
- Они задаются произвольным именем переменной, перед которой ставится звездочка (*).

4. Аргументы произвольной длины. Пример

```
# Определим функцию с произвольным
количеством параметров
# Данная функция выводит переданные ей
аргументы в консоль
def print_args(*args):
    print(args)

# Вызовем функцию без аргументов
print_args()

# Вызовем функцию с 1 аргументом
print_args('Строка')

# Вызовем функцию с 5ю аргументами
print_args(1, 'Строка', 'Еще строка', 38, 4)
```

```
()
('Строка',)
(1, 'Строка', 'Еще строка', 38, 4)
```

Локальные и глобальные переменные в функциях

Глобальные переменные

Внутри функции можно использовать переменные, объявленные вне этой функции

```
def f():
    print(a)

a = 1
f()
```

- Здесь переменной `a` присваивается значение `1`, и функция `f()` печатает это значение, несмотря на то, что до объявления функции `f` эта переменная не инициализируется. В момент вызова функции `f()` переменной `a` уже присвоено значение, поэтому функция `f()` может вывести его на экран.
- Такие переменные (объявленные вне функции, но доступные внутри функции) называются **глобальными**.

Локальные переменные

- Но если инициализировать какую-то переменную внутри функции, использовать эту переменную вне функции не удастся. Например:

```
def f():
    a = 1

f()
print(a)
```

- **Получим ошибку NameError: name 'a' is not defined.**
  Такие переменные, объявленные внутри функции, называются **локальными**.
- Эти переменные становятся недоступными после выхода из функции.

«Защита» глобальных переменных

- Интересным получится результат, если попробовать изменить значение глобальной переменной внутри функции:

```
def f():
    a = 1
    print(a)

a = 0
f()
print(a)
```

```
1
0
```

- Будут выведены числа 1 и 0. Несмотря на то, что значение переменной a изменилось внутри функции, вне функции оно осталось прежним!
- Это сделано в целях “защиты” глобальных переменных от случайного изменения из функции.

Пример

- **Интерпретатор Python считает переменную локальной для данной функции, если в её коде есть хотя бы одна инструкция, модифицирующая значение переменной, то эта переменная считается локальной и не может быть использована до инициализации.**
- Инструкция, модифицирующая значение переменной — это операторы =, +=, а также использование переменной в качестве параметра цикла for.
- При этом даже если инструкция, модицифицирующая переменную никогда не будет выполнена, интерпретатор это проверить не может, и переменная все равно считается локальной. Пример:

```
def f():
    print(a)
    if False:
        a = 0

a = 1
f()
```

- **Возникает ошибка: UnboundLocalError: local variable 'a' referenced before assignment.**
- В функции `f()` идентификатор a становится локальной переменной, т.к. в функции есть команда, модифицирующая переменную a, пусть даже никогда и не выполняющийся (но интерпретатор не может это отследить).
- Поэтому вывод переменной a приводит к обращению к неинициализированной локальной переменной.

Глобальные переменные - global

- Чтобы функция могла изменить значение глобальной переменной, необходимо объявить эту переменную внутри функции, как глобальную, при помощи ключевого слова **global**:

```
def f():
    global a
    a = 1
    print(a)

a = 0
f()
print(a)
```

```
1
1
```

- В этом примере на экран будет выведено 1 1, так как переменная a объявлена, как глобальная, и ее изменение внутри функции приводит к тому, что и вне функции переменная будет доступна.
- **Тем не менее, лучше не изменять значения глобальных переменных внутри функции.** Если ваша функция должна поменять какую-то переменную, пусть лучше она вернёт это значением, и вы сами при вызове функции явно присвоите в переменную это значение.
- Если следовать этим правилам, то функции получаются независимыми от кода, и их можно легко копировать из одной программы в другую.

Пример

```
# начало куска кода, который можно
копировать из программы в программу
def factorial(n):
    res = 1
    for i in range(2, n + 1):
        res *= i
    return res
# конец куска кода

n = int(input())
f = factorial(n)
print(f)
# дальше всякие действия с переменной f
```

Рекурсия

Рекурсия

```
def short_story():
    print("У попа была собака, он ее любил.")
    print("Она съела кусок мяса, он ее убил,")
    print("В землю закопал и надпись написал:")
    short_story()
```

Рекурсия. Вычисление факториала

- **Рассмотрим это на примере функции вычисления факториала.**
- Хорошо известно, что 0!=1, 1!=1. А как вычислить величину n! для большого n?
- Если бы мы могли вычислить величину (n-1)!, то тогда мы легко вычислим n!, поскольку n!=n·(n-1)!. Но как вычислить (n-1)!? Если бы мы вычислили (n-2)!, то мы сможем вычисли и (n-1)!=(n-1)·(n-2)!. А как вычислить (n-2)!? Если бы...
- В конце концов, мы дойдем до величины 0!, которая равна 1.
- Таким образом, для вычисления факториала мы можем использовать значение факториала для меньшего числа.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

- **Подобный прием (вызов функцией самой себя) называется рекурсией, а сама функция называется рекурсивной.**

Рекурсия

- Рекурсивные функции являются мощным механизмом в программировании. К сожалению, они не всегда эффективны.
- Также часто использование рекурсии приводит к ошибкам.
- Наиболее распространенная из таких ошибок – бесконечная рекурсия, когда цепочка вызовов функций никогда не завершается и продолжается, пока не кончится свободная память в компьютере.
- Две наиболее распространенные причины для бесконечной рекурсии:

1. Неправильное оформление выхода из рекурсии. Например, если мы в программе вычисления факториала забудем поставить проверку if n == 0, то factorial(0) вызовет factorial(-1), тот вызовет factorial(-2) и т. д.
2. Рекурсивный вызов с неправильными параметрами. Например, если функция factorial(n) будет вызывать factorial(n), то также получится бесконечная цепочка.

- Поэтому при разработке рекурсивной функции необходимо прежде всего оформлять условия завершения рекурсии и думать, почему рекурсия когда-либо завершит работу.