IT CAREER HUB

Урок 20.

Функции. Области видимости

IT CAREER HUB

Функция

Функция — это именованный блок кода, предназначенный для выполнения определённой задачи. Функции позволяют переиспользовать код, делая программы более организованными и читаемыми.

Ранее мы использовали только готовые функции. На этом занятии мы научимся создавать функции самостоятельно.

Зачем нужны функции?

- **Повторное использование кода**: Один раз написав функцию, её можно вызывать многократно.
- **Читаемость**: Функции помогают разбивать программу на логические части.
- **Модульность**: Удобно структурировать код, разбивая его на независимые части.
- **Упрощение отладки**: Изменения в функции автоматически применяются во всех местах её вызова.

Синтаксис:

Python
```
def function_name(parameters):

    # Тело функции: код, выполняющийся при вызове

    return result  # Возвращаемое значение (опционально)
```

- **def**: ключевое слово, обозначающее создание функции.
- **function_name**: имя функции, используемое для её вызова.
- **parameters**: входные данные (аргументы), которые передаются в функцию.
- **return**: возвращает результат выполнения функции (необязательно).

Пример:

Python
```
def greet(name):

    print(f"Привет, {name}!")
```

```
greet("Алиса")
```

IT CAREER HUB

Ключевое слово pass

Ключевое слово **pass** используется как заглушка. Оно позволяет определить пустой блок кода, который не выполняет никаких действий. Это полезно, когда вы планируете добавить код позже, но хотите сохранить корректный синтаксис.

Особенности:

- **Ничего не делает**: `pass` ничего не выполняет, он просто позволяет избежать ошибок синтаксиса в местах, где блок кода обязателен.
- **Часто используется для заглушек**: Позволяет временно оставить место под код, который будет добавлен позже.
- **Применяется в конструкциях с обязательным блоком кода**: Например, в циклах, функциях, условиях.

Примеры:

Python

```python
# Условие

if True:

    pass  # Блок кода будет добавлен позже

else:

    print("False")



# Цикл

for i in range(5):

    pass  # Цикл ничего не делает, но синтаксически корректен



# Функция

def validate_data():
```

```
    pass  # Функция пока не реализована
```

IT CAREER HUB

Ключевое слово def

def — это ключевое слово, которое используется для определения пользовательской функции. С его помощью можно создавать именованные блоки кода, которые выполняют определённые задачи и могут быть вызваны в программе многократно.

Имя функции — это уникальный идентификатор, используемый для её вызова. Оно должно быть информативным и описывать, что делает функция. В Python для именования функций применяются определённые правила и рекомендации, следование которым помогает улучшить читаемость и поддерживаемость кода.

Функция — это именованный блок кода, предназначенный для выполнения определённой задачи. Функции позволяют переиспользовать код, делая программы более организованными и читаемыми.
Ранее мы использовали только готовые функции. На этом занятии мы научимся создавать функции самостоятельно.

Зачем нужны функции?

- **Повторное использование кода**: Один раз написав функцию, её можно вызывать многократно.
- **Читаемость**: Функции помогают разбивать программу на логические части.
- **Модульность**: Удобно структурировать код, разбивая его на независимые части.
- **Упрощение отладки**: Изменения в функции автоматически применяются во всех местах её вызова.

Синтаксис:

Python

```
def function_name(parameters):

    # Тело функции: код, выполняющийся при вызове

    return result  # Возвращаемое значение (опционально)
```

- `def`: ключевое слово, обозначающее создание функции.
- `function_name`: имя функции, используемое для её вызова.
- `parameters`: входные данные (аргументы), которые передаются в функцию.

- `return`: возвращает результат выполнения функции (необязательно).

Пример:

```python
def greet(name):

    print(f"Привет, {name}!")



greet("Алиса")
```

IT CAREER HUB

Вызов функции

Вызов функции — это процесс выполнения ранее определённого блока кода (функции) с помощью её имени. При вызове можно передать функции необходимые аргументы и получить результат её работы.

Примеры:

Python

```python
def greet():

    print("Hello!")



# Вызов функции без аргументов

greet()
```

Python

```python
def greet_person(name):

    print(f"Hello, {name}!")



# Вызов функции с аргументами

greet_person("Alice")
```

IT CAREER HUB

Аргументы функций

Аргументы функции — это значения, которые передаются функции при её вызове. С их помощью можно передавать данные, которые функция использует для выполнения своих задач.

Типы аргументов функций

Позиционные аргументы

- Передаются функции в порядке, указанном в определении.
- **Порядок важен**: каждое значение будет присвоено соответствующему параметру.
- **Количество ожидаемых аргументов должно совпадать с количеством переданных аргументов.**

Пример:

```python
def greet(name, age):  # Ожидаемые аргументы

   print(f"My name is {name} and I am {age} years old.")



greet("Alice", 25)  # Переданные аргументы

greet("Bob", 30)  # Переданные аргументы
```

Значения `"Alice"` и `25` передаются в переменные `name` и `age` соответственно. Теперь переменные `name` и `age` можно использовать внутри функции. При каждом вызове переменные принимают новые значения.

TypeError

TypeError — это исключение, которое возникает, когда операция или функция применяется к объекту неподходящего типа. Эта ошибка сигнализирует, что программа пытается выполнить действие, которое не поддерживается данным типом данных.

Например, TypeError возникает если количество переданных аргументов не совпадает с количеством ожидаемых аргументов.

Примеры:

Передано меньше аргументов, чем ожидается

```python
def greet(name, age):  # Ожидаемые аргументы
    print(f"My name is {name} and I am {age} years old.")


greet("Alice")  # Ошибка: меньше чем ожидается
```

Передано больше аргументов, чем ожидается Если передать больше аргументов, чем указано в определении функции, Python сообщит об избыточных аргументах.

```python
def greet(name, age):  # Ожидаемые аргументы
    print(f"My name is {name} and I am {age} years old.")


greet("Alice", 25, "Minsk")  # Ошибка: больше чем ожидается
```

Именованные аргументы:

Передаются с указанием имени параметра, что делает вызов функции более понятным. Порядок следования не имеет значения.

Пример:

```python
def greet(name, age):  # Ожидаемые аргументы
    print(f"My name is {name} and I am {age} years old.")
```

```
greet(age=30, name="Bob")  # Именованные аргументы
```

Аргументы по умолчанию:

При определении функции можно указать значения по умолчанию для аргументов. Если аргумент не передан, используется значение по умолчанию. Сначала указываются обязательные аргументы (без значения по умолчанию), а затем — аргументы со значениями по умолчанию. Нарушение этого порядка приведёт к ошибке синтаксиса.

Пример:

Python
```
def greet(name, greeting="Hello"):
   print(f"{greeting}, {name}!")


greet("Alice")  # Приветствие не передано, будет использовано "Hello"
greet("Bob", "Hi")  # Вывод: Hi, Bob!
```

Упаковка аргументов в функции

Упаковка аргументов — это процесс, при котором передаваемые в функцию аргументы объединяются в одну коллекцию:

- кортеж для позиционных аргументов (с помощью символа `*`)
- словарь для именованных аргументов (с помощью символов `**`).

Параметры *args и **kwargs

`*args` и `**kwargs` — это специальные параметры, которые позволяют передавать в функцию переменное количество аргументов.

*args

Позволяет передавать любое количество позиционных аргументов (в том числе ни одного). Аргументы упаковываются в кортеж.

Пример:

```python
def calculate_sum(*args):

    print("Аргументы:", args)

    print("Сумма:", sum(args))



calculate_sum(1, 2, 3)

calculate_sum()
```

**kwargs

Позволяет передавать любое количество именованных аргументов. Аргументы упаковываются в словарь.

Пример:

```python
def print_user_info(**kwargs):

    print("Информация о пользователе:")

    for key, value in kwargs.items():

        print(f"\t{key}: {value}")



print_user_info(name="Alice", age=25, city="New York")

print_user_info()
```

IT CAREER HUB

Комбинация различных типов аргументов

В одной функции можно использовать разные типы аргументов. При этом важно соблюдать их порядок:

1. Позиционные аргументы
2. `*args`
3. Аргументы по умолчанию
4. `**kwargs`

Пример:

Python

```python
def show_full_info(name, *args, age=25, **kwargs):

   print(f"Name: {name}")

   print(f"Other details: {args}")

   print(f"Age: {age}")

   print(f"Additional info: {kwargs}")



show_full_info("Alice", "Developer", age=30, city="New York", hobby="Reading")
```

Ключевое слово return

Ключевое слово `return` используется для возврата значения из функции в место её вызова. Оно завершает выполнение функции и возвращает указанное значение (или `None`, если значение не указано).

Особенности:

- **Возврат значения:**
  `return` позволяет передать результат работы функции обратно к вызывающему коду.
- **Завершение функции:**
  После выполнения инструкции `return` функция завершает свою работу, даже если после неё есть другие строки кода.

- **Отсутствие значения:**
  Если `return` вызывается без указания значения, функция возвращает `None`.
- **Отсутствие return:**
  Даже если `return` отсутствует или не достигнут, функция выполняет код до конца и возвращает `None`.
- **Несколько return:**
  Функция может содержать несколько `return`. В этом случае выполнится первый достигнутый `return`.

Синтаксис:

```python
def function_name(parameters):

    # тело функции

    return value
```

- `return`: возвращает значение, указанное после него

Примеры:

1. Возврат значения:

```python
def add(a, b):

    return a + b



result = add(3, 5)

print(result)
```

Функция `add` возвращает сумму чисел, которая сохраняется в переменной `result`.

2. Возврат одного из значений:

```python
def check_positive(number):
    if number > 0:
        return "Положительное число"
    return "Отрицательное или ноль"

print(check_positive(10))
print(check_positive(-5))
```

Если условие выполняется, функция завершается после первого `return`, и код после него не исполняется.

3. Возврат `None`:

```python
def say_hello():
    print("Hello, World!")

result = say_hello()
print(result)
```

Функция `say_hello` ничего не возвращает, поэтому её результат равен `None`.

4. Множественный возврат значений:

```python
def calculate(a, b):
    return a + b, a - b
```

```
result = calculate(10, 5)
print(result)
```

`return` может возвращать несколько значений, которые упаковываются в кортеж.

5. Пустой `return`:

Python

```
def calculate_factorial(n):
    if n < 0:
        return  # Завершаем функцию без вычислений
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

num1 = -5
print(f"Факториал числа {num1}: {calculate_factorial(num1)}")
num2 = 5
print(f"Факториал числа {num2}: {calculate_factorial(num2)}")
```

В этом примере `return` используется для завершения функции, если число отрицательное, чтобы избежать дальнейших вычислений.

Задания для закрепления

1. Какой результат будет выведен при выполнении следующего кода?

```python
def example():

    pass



print(example())
```

a. None
b. Ошибка
c. pass
d. Ничего не будет выведено

2. Какой результат будет выведен при выполнении следующего кода?

```python
def func(a, b, c=10):

    return a + b + c



print(func(2, 3))
```

a) Ошибка
b) 15
c) 5
d) None

3. Какой результат будет выведен при выполнении следующего кода?

```python
def check_number(n):
    if n > 0:
        return "Positive"
    return "Non-positive"

print(check_number(-1))
```

a) Ошибка из-за отсутствия `else`
b) `"Positive"`
c) `"Non-positive"`
d) `None`

4. Какой результат будет выведен при выполнении следующего кода?

```python
def info(**kwargs):
    return kwargs

print(info(name="Alice", age=30))
```

a) Ошибка
b) `{"name": "Alice", "age": 30}`
c) `["name", "Alice", "age", 30]`
d) `("name", "Alice", "age", 30)`

IT CAREER HUB

Области видимости в Python

Область видимости — это контекст (границы), в котором переменные доступны для использования. Она определяет, где можно обращаться к данным и какие переменные доступны в разных частях программы.

Python использует систему областей видимости, определяемую **правилом LEGB**:

1. **Local** (Локальная область)
2. **Enclosing** (Область окружающих функций, подробнее при изучении замыканий)
3. **Global** (Глобальная область)
4. **Built-in** (Встроенная область)

1. Local (Локальная область)

- **Область действия:** ограничена функцией, в которой переменная была объявлена.
- **Срок жизни:** существует только во время выполнения функции.
- **Определение:** локальная переменная создаётся внутри функции и доступна только в её пределах.

Пример:

Python

```python
def my_function():

    local_var = 10  # Локальная переменная

    print(f"Локальная переменная: {local_var}")



my_function()

# print(local_var)  # Ошибка: local_var недоступна за пределами функции
```

2. Global (Глобальная область)

- **Область действия:** доступна во всей программе, включая функции.

- **Срок жизни:** существует, пока выполняется программа.
- **Определение:** объявляется вне функций или других блоков и доступна во всей программе.

Пример:

```python
global_var = 20  # Глобальная переменная



def show_global():

    print(f"Глобальная переменная: {global_var}")



show_global()

print(global_var)  # Глобальная переменная доступна в любом месте
```

3. Built-in (Встроенные объекты)

- Эта область содержит встроенные функции и объекты Python (например, `len()`, `print()`, `int()`).
- Они доступны в любом месте программы, если не переопределены.

Пример:

```python
print(len("Hello"))  # Вызов встроенных функций len и print
```

IT CAREER HUB

Правило LEGB (поиск переменных в Python)

Python ищет переменные в следующем порядке:

1. **Local** — в локальной области функции.
2. **Enclosing** — в области окружающих функций (замыкания).
3. **Global** — среди глобальных переменных.
4. **Built-in** — среди встроенных объектов.

Примеры работы LEGB

Пример 1:

```python
# x = 10  # Глобальная переменная


def function():
    # x = 5  # Локальная переменная
    print(x)


function()
```

Если переменная отсутствует во всех четырёх областях, **Python выбрасывает исключение `NameError`**.

Пример 2:

```python
x = 10  # Глобальная переменная


def my_function():
```

```
    x = 5  # Локальная переменная с тем же именем

    print(f"Локальная переменная: {x}")



my_function()

print(f"Глобальная переменная: {x}")     # Глобальная переменная остаётся неизменной
```

Вывод:
Если внутри функции объявляется переменная с тем же именем, что и глобальная, **локальная переменная перекрывает доступ к глобальной**.

IT CAREER HUB

Ключевое слово global

global — это ключевое слово в Python, которое позволяет изменять значение глобальной переменной внутри функции. Без него присваивание значения переменной внутри функции создаст новую локальную переменную, не затрагивая глобальную.

Синтаксис:

```python
global variable_name
```

variable_name — имя глобальной переменной, к которой требуется доступ.

Пример без global:

```python
count = 0  # Глобальная переменная



def increment_counter():

     count = count + 1   # Ошибка, так как `count` внутри функции считается локальной

    print(count)



increment_counter()
```

Код вызовет **UnboundLocalError**, так как Python воспринимает `count` как локальную переменную, но мы пытаемся использовать её до присваивания.

Пример с global:

```python
count = 0  # Глобальная переменная
```

```
def increment_counter():
    global count  # Указываем, что работаем с глобальной переменной
    count += 1

increment_counter()
print(count)
increment_counter()
print(count)
```

Теперь `count` изменяется в глобальной области видимости, и код работает без ошибок.

Особенности использования global:

- Изменяет глобальную переменную, а не создаёт новую локальную.
- Не требуется при **чтении** глобальной переменной внутри функции, но нужен при **изменении**.
- Чрезмерное использование `global` может привести к трудноотслеживаемым ошибкам, поэтому его следует применять с осторожностью.

Зачем передавать аргументы в функцию

В программировании рекомендуется передавать значения в функцию через параметры, а не использовать глобальные переменные внутри функции. Это связано с несколькими важными принципами и проблемами, которые могут возникнуть при работе с глобальными переменными.

- **Повышение читаемости и предсказуемости кода:** Когда функция получает значения через параметры, она становится независимой от внешних переменных. Это облегчает понимание кода, так как все входные данные функции видны прямо в её объявлении.

- **Предотвращение неожиданных ошибок:** Использование глобальных переменных внутри функций может привести к неожиданным результатам, если глобальная переменная будет изменена в другом месте программы.
- **Упрощение тестирования и переиспользования кода:** Функции, которые зависят от глобальных переменных, трудно тестировать, так как их поведение меняется в зависимости от состояния программы.
- **Избегание конфликтов с именами переменных:** Глобальные переменные могут случайно перезаписаться в разных частях программы, что приведёт к трудноуловимым ошибкам.

Пример с аргументами (рекомендуемый вариант):

Python

```python
# Все значения передаются в функцию явно, что делает код понятным

def calculate_area(width, height):

    return width * height



result = calculate_area(5, 10)

print(result)
```

Пример с глобальными переменными (плохая практика):

Python

```python
width = 5

height = 10



def calculate_area():

   # Непонятно, откуда берутся width и height, если смотреть только на функцию

   return width * height  # Использует глобальные переменные
```

```
result = calculate_area()

print(result)
```

Использование global в Python

Когда НЕ стоит использовать `global`

- **Для передачи данных между функциями** — лучше передавать параметры.
- **В больших программах** — сложно отслеживать, где изменяется переменная.
- **Для временных значений** — лучше использовать локальные переменные.

Когда `global` все же оправдан

- **Для изменения переменных на уровне модуля.** Например, в настройках программы для хранения конфигураций.
- **В небольших скриптах**, где глобальные переменные помогают быстро решить задачу и масштабируемость не важна.

Задания для закрепления

1. Какой результат будет выведен при выполнении следующего кода?

```python
x = 10  # Глобальная переменная



def my_function():

    x = 5  # Локальная переменная с тем же именем

    print(f"Локальная переменная: {x}")



my_function()

print(f"Глобальная переменная: {x}")
```

Варианты ответов:

a. a) Локальная переменная: 10
   Глобальная переменная: 10
b. b) Локальная переменная: 5
   Глобальная переменная: 10
c. c) Локальная переменная: 5
   Глобальная переменная: 5
d. d) Ошибка

2. Какой результат будет выведен при выполнении следующего кода?

```python
y = 20



def change_global():
```

```
    global y

    y += 5



change_global()

print(y)
```

Варианты ответов:

a) Ошибка
b) 20
c) 25
d) 5

3. Какой результат будет выведен при выполнении следующего кода?

Python

```
counter = 0



def increment():

    counter += 1



increment()

print(counter)
```

Варианты ответов:

a) Ошибка
b) 1
c) 0
d) None

IT CAREER HUB

Практические задания

1. Конвертер температуры

Напишите функцию, которая конвертирует температуру из градусов Цельсия в Фаренгейты и наоборот.

Формулы для конвертации температур:

- **Из градусов Цельсия в Фаренгейты:** $F=C \times \frac{9}{5} + 32$
- **Из градусов Фаренгейта в Цельсия:** $C=(F-32) \times \frac{5}{9}$

Данные:

```python
temp = 100

scale = "C"
```

Пример вывода:

```
100C = 212.0F
```

Решение:

```python
def convert_temperature(temp, scale):
    if scale.upper() == "C":
        return f"{temp}C = {temp * 9/5 + 32}F"
    elif scale.upper() == "F":
        return f"{temp}F = {(temp - 32) * 5/9}C"
```

```
temp = 100

scale = "C"

print(convert_temperature(temp, scale))
```

2. Фильтрация списка по длине

Создайте функцию `filter_strings`, которая принимает целое число `n` и любое количество строк (по отдельности, а не как коллекция).
Функция должна возвращать список строк, длина которых больше `n`.

Данные:

Python

```
strings = ["apple", "banana", "cherry", "date", "fig"]

n = 5
```

Пример вывода:

Unset

```
['banana', 'cherry']
```

Решение:

Python

```
def filter_strings(min_len, *words):

    return [string for string in words if len(string) > min_len]



strings = ["apple", "banana", "cherry", "date", "fig"]

n = 5
```

```
print(filter_strings(n, *strings))
```

3. Проверка знака числа

Напишите функцию, которая принимает число и возвращает, является ли оно **положительным, отрицательным или нулём**.

Данные:

Python

```
num = -3
```

Пример вывода:

Unset

```
Число отрицательное
```

Решение:

Python

```
def check_number(num):
    if num > 0:
        return "Число положительное"
    elif num < 0:
        return "Число отрицательное"
    else:
        return "Число равно нулю"


num = -3
```

```
print(check_number(num))
```