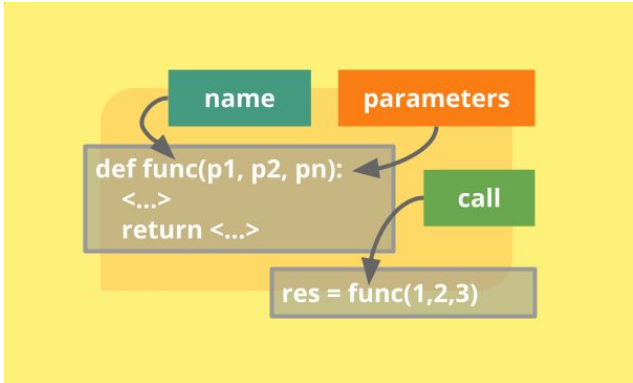
Python-6
Функции в Python.
Глобальные переменные.

Что такое функция?

Функции в Python представляют собой блоки кода, которые могут быть многократно вызваны для выполнения определенной задачи. Они служат для организации кода, делая его более читаемым, поддерживаемым и модульным. Функции также позволяют избежать дублирования кода и способствуют повторному использованию.



Как определить функцию?

Функция в Python определяется с использованием ключевого слова def, за которым следует имя функции, список параметров в круглых скобках и двоеточие. Тело функции выделяется отступами (обычно используется четыре пробела в соответствии с правилами PEP8).

Пример (определения функции)

ключевое слово def

имя функции

передаваемые параметры функции (список параметров)

```
def foo(p1, p2):

    # Тело функции

    result = p1 + p2

    return result
```

тело функции (или то что происходит внутри функции, в данном случае сумма двух переменных p1 и p2 с записью в переменную result)

оператор return - для возврата какого либо значения из функции (в данном примере возвращаем значение переменной result)

*оператор return используется только тогда когда это необходимо!

Вызов функции. Пример вызова.

Функцию можно вызвать, указав ее имя, передав необходимые аргументы в скобках. Если функцию не принимает никаких аргументов (параметров), тогда круглые скобки будут пустые.

Простой пример:

```
result = foo(10, 20)
print(result) # 10 + 20 = 30
```

Возвращаемые значение, использование return.

Функции могут возвращать значения с использованием ключевого слова return (оператор return). Если return отсутствует или возвращает None, функция автоматически возвращает None. (в целом об этом я уже сказал ранее в примерах).

Пример (сумма двух чисел a и b):

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

sum_result = add(3, 5)
print(sum_result)  # Выведет: 8
```

Типы передаваемых параметров в функцию.

- Обязательные параметры - те параметры, которые должны быть переданы функции при вызове.
- Параметры по умолчанию - те параметры, которые имеют значения по умолчанию и могут быть опущены при вызове.
- Произвольное количество параметров - позволяют передавать переменное количество аргументов в функцию. (*это достигается с использованием * для произвольных позиционных аргументов и ** для произвольных именованных аргументов.*)

Пример (обязательные параметры ф-ии).

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

# Вызов функции с обязательным параметром
greet("Alice")  # Выведет: Hello, Alice!

# Вызов функции без обязательного параметра (приведет к ошибке)
# greet()  # Ошибка: TypeError
```

Пример (параметры по-умолчанию).

```
def greet_with_default(name="Guest"):
    print(f"Hello, {name}!")

# Вызов функции с параметром
greet_with_default("Bob")  # Выведет: Hello, Bob!

# Вызов функции без параметра (используется значение по умолчанию)
greet_with_default()  # Выведет: Hello, Guest!
```

Пример (произвольное кол-во параметров). (*)

```
def add_numbers(*args):
    sum_result = sum(args)
    print(f"Sum: {sum_result}")

# Вызов функции с разным количеством аргументов
add_numbers(1, 2, 3)        # Выведет: Sum: 6
add_numbers(1, 2, 3, 4, 5)  # Выведет: Sum: 15
```

Пример (произвольное кол-во параметров). (**)

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

# Вызов функции с произвольными именованными аргументами
print_details(name="John", age=30, city="New York")
# Выведет:
# name: John
# age: 30
# city: New York
```

TypeHints

В Python введены type hints (подсказки типов) для улучшения читаемости кода и статической проверки типов. Type hints предоставляют информацию о типах переменных, аргументов функций и возвращаемых значений. Одним из важных аспектов type hints является указание типа возвращаемого значения функции.

Пример:

типы данных аргументов функции

тип возвращаемого элемента (по return определяется)

```
def add_numbers(a: int, b: int) -> int:
    return a + b
```

т.к int + int = int значит тип возвращаемого элемента будет int

Лямбда-функции.

Лямбда-функции (или анонимные функции) в Python - это специальный вид функций, которые определяются с использованием ключевого слова lambda. Они предназначены для создания коротких, однострочных функций, которые могут быть использованы там, где требуется функция, но нет необходимости явно определять функцию с помощью def.

Синтаксис лямбда-фукнций. Пример.

```
lambda arguments: expression
```

- lambda: ключевое слово, указывающее на создание лямбда-функции.
- arguments: список параметров, так же как в обычной функции.
- expression: выражение, которое выполняется при вызове лямбда-функции.

```
# Пример 1: Лямбда-функция с одним аргументом
square = lambda x: x ** 2
print(square(5))  # Выведет: 25
```

Лямбда функции примеры.

```
# Пример 2: Лямбда-функция с двумя аргументами
add = lambda a, b: a + b
print(add(3, 4))  # Выведет: 7

# Пример 3: Лямбда-функция без аргументов
greeting = lambda: "Hello, World!"
print(greeting())  # Выведет: Hello, World!

# Пример 4: Использование лямбда-функции внутри функции высшего порядка (например, функции map)
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)  # Выведет: [1, 4, 9, 16, 25]
```

Лямбда-функции удобны, когда требуется краткая функция для одноразового использования. Однако, они имеют ограничения: они ограничены одним выражением и не могут содержать множество инструкций.

Оператор PASS.

Оператор PASS - используется как пустая инструкция программного кода на языке программирования Python. когда вам нужно написать синтаксически корректный блок кода в Python, но вам не нужно в нем выполнять какие-либо конкретные действия. Простыми словами, это своего рода заполнитель, который позволяет вам создавать пустые блоки кода, чтобы ваш код соответствовал правильному синтаксису Python.

Пример: если у вас есть заглушка для функции, которую вы планируете реализовать позже, вы можете использовать pass, чтобы создать корректный синтаксический блок:

```
def my_function():
   pass  # Пока нет реализации, но синтаксически корректно
```

Функции @ДЕКОРАТОРЫ

Декораторы - это способ модифицировать или обернуть функцию другой функцией. Они предоставляют удобный синтаксис для изменения поведения функций без изменения их кода.

- Декораторы могут использоваться для записи информации о вызовах функции, времени их выполнения, переданных аргументах и так далее. (logging)
- Декораторы могут использоваться для кеширования результатов выполнения функций, чтобы избежать повторных вычислений. (from functools import lru_cache; @lru_cache(maxsize=None) def)
- Декораторы могут применяться для проверки прав доступа пользователя к выполнению определенной функции.
- Декораторы могут использоваться для измерения времени выполнения функции.
- Декораторы могут использоваться в ООП.

Пример (декоратора)

```python
def decore_hello(func):
    def foo(name):
        new_name = name.capitalize()
        res = func(new_name)
        return res
    return foo

@decore_hello
def hello(имя):
    return f"Привет, {имя}!"
print(hello("анна"))
```

```
Привет, Анна!
```

В этом фрагменте кода определен декоратор `decore_hello`. Этот декоратор принимает функцию `func` в качестве аргумента и возвращает новую функцию `foo`. Функция `foo` принимает аргумент `name`, преобразует его капитализацией, затем вызывает исходную функцию `func` с измененным аргументом и возвращает результат.

Строка `@decore_hello` является синтаксическим свойством для конструкции `hello = decore_hello(hello);`

Таким образом, функция `hello` теперь ссылается на результат вызова декоратора `decore_hello` с функцией `hello` в качестве аргумента. Теперь `hello` в действительности является функцией `foo`, возвращаемой декоратором.

При вызове `hello("анна")` происходит следующее:

- *Аргумент "анна" капитализируется в "Анна". (capitalize())*
- *Вызывается функция `foo` (теперь `hello`) с аргументом "Анна".*
- *Внутри `foo` вызывается исходная функция `func` (в данном случае, исходная `hello`) с аргументом "Анна".*
- *Результат этого вызова возвращается и выводится на экран.*

Перегрузка функций.

Перегрузка функций в Python представляет собой возможность определения нескольких функций с одинаковым именем, но различающихся параметрами. Это позволяет создавать более гибкий и удобный код, обрабатывающий различные типы данных или количество аргументов. Важно отметить, что Python не поддерживает строгую типизацию, поэтому перегрузка функций осуществляется на основе количества и типов параметров.

Например (1-ошибочный, 2-Normal):

```
def add(a: int, b: int) -> int:
    return a + b

def add(a: float, b: float) -> float:
    return a + b

def add(a: int, b: int, c: int) -> int:
    return a + b + c

print(add(1, 2)) # add(<int> a, <int> b) Error
```

```
def add(a: int, b: int) -> int:
    return a + b

def add_float(a: float, b: float) -> float:
    return a + b

def add_three(a: int, b: int, c: int) -> int:
    return a + b + c

print(add(1, 2)) # 3
```

Python, полезные функции.

Полезная функция v1/ZIP().

Функция **zip()** в Python используется для создания итератора, который комбинирует элементы из двух или более итерируемых объектов. Она объединяет элементы на соответствующих позициях, создавая кортежи.

```
zip(arg1, arg2, ... , argN)
```

Например есть у меня два списка **list1**, и **list2** и нужно допустим получить (создать) словарь где list1 это ключи, а list2 соответствующие значения. На помощь приходит функция **zip**.

```
list1 = ["Alex", "Tom", "Jack"]
list2 = [40, 50, 60]
dictionary = dict(zip(list1, list2))
print(dictionary) # {'Alex': 40, 'Tom': 50, 'Jack': 60}
```

Полезная функция v2/SORTED().

Функция **sorted()** в Python используется для сортировки итерируемых объектов, таких как списки, кортежи и строки.

```
sorted(iterable, key=None, reverse=False)
```

- **iterable** - Обязательный аргумент, представляющий объект, который вы хотите отсортировать (список, кортеж, строку и т. д.).
- **key** - Необязательный аргумент, представляющий функцию, которая вызывается для каждого элемента перед сравнением во время сортировки. По умолчанию None, что означает использование стандартного порядка сортировки.
- **reverse** - Необязательный аргумент, булевого типа. Если True, сортировка производится в порядке убывания, если False (по умолчанию), в порядке возрастания.

Пример sorted dict.

```
d = {"Alice": 25, "Bob": 30, "Charlie": 22}
# Сортировка по значениям в порядке убывания
sorted_dict = dict(sorted(d.items(), key=lambda x: x[1], reverse=True))
print(sorted_dict)
```

Полезная функция v3/ENUMERATE().

Функция **enumerate()** в Python используется для перебира элементов итерируемого объекта вместе с их индексами.

```
enumerate(iterable, start=0)
```

- **iterable** - Обязательный аргумент, представляющий объект, который вы хотите отсортировать (список, кортеж, строку и т. д.).
- **start** - начальная задаваемая позиция перебора элементов итерируемого объекта.

Пример (enumerate):

```
iterable = ['apple', 'banana', 'cherry']
for index, value in enumerate(iterable, start=1):
    print(f"Index {index}: {value}") #Index 1: apple; Index 2: banana; Index 3: cherry
```

*В данном примере **enumerate** возвращает пары (index, value), начиная с 1, так как мы указали start=1.*

Полезная функция v4/MAP().

Функция **map()** в Python применяет указанную функцию ко всем элементам итерируемого объекта.

```
map(function, iterable, ...)
```

- **function** - указанная функция.
- **iterable** - Обязательный аргумент, представляющий объект, который вы хотите отсортировать (список, кортеж, строку и т. д.).

Пример (map):

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
print(list(squared)) # [1, 4, 9, 16, 25]
```

*Функция **map** применяет переданную функцию (в данном случае, lambda-функцию) к каждому элементу **numbers** и возвращает объект, который нужно преобразовать в список.*

Полезная функция v5/FILTER().

Функция **filter()** в Python фильтрует элементы итерируемого объекта на основе заданной функции.

```
filter(function, iterable)
```

- **function** - указанная функция.
- **iterable** - Обязательный аргумент, представляющий объект, который вы хотите отсортировать (список, кортеж, строку и т. д.).

Пример (filter):

```
numbers = [1, 2, 3, 4, 5]
even = filter(lambda x: x % 2 == 0, numbers)
print(list(even)) # [2, 4]
```

filter оставляет только те элементы из **numbers**, для которых функция (в данном случае, проверка на четность) возвращает True.

Полезная функция v6/DECODE/ENCODE().

Функция **decode(), encode()** в Python используются для преобразования строк в байтовые объекты и обратно.

```
encode(encoding='UTF-8', errors='strict')
decode(encoding='UTF-8', errors='strict')
```

- **encoding** - Опциональный параметр, указывающий кодировку (по умолчанию UTF-8).
- **errors** - Опциональный параметр, указывающий, как обрабатывать ошибки кодирования (по умолчанию 'strict', что означает вызов исключения).

Пример (encode):

```
text = "Hello, World!"
encoded_text = text.encode('utf-8')
print(encoded_text) # b'Hello, World!'
```

encode() - преобразует строку в байтовый объект, используя указанную кодировку.

Пример decode().

```
byte_data = b'Hello, World!'
decoded_text = byte_data.decode('utf-8')
print(decoded_text) # Hello, World!
```

decode() обратная операция, функция преобразует байтовый объект в строку, используя указанную кодировку.

Python, глобальные переменные.

Глобальные переменные в Python.

Глобальные переменные в Python - это переменные, которые определены вне функций и имеют глобальную область видимости, что означает, что они могут быть доступны из любой части программы, включая функции. Однако, чтобы изменить значение глобальной переменной внутри функции, вы должны использовать ключевое слово global.

Пример (глобальные переменные) (чтение условно)

```
# Глобальная переменная
global_variable = 10

def my_function():
    # Использование глобальной переменной внутри функции
    print("Inside the function:", global_variable)

# Вызов функции
my_function()  # Выведет: Inside the function: 10
```

Пример (глобальные переменные) (изменение значения)

```
# Глобальная переменная
global_variable = 10

# Изменение глобальной переменной внутри функции
def modify_global():
    global global_variable
    global_variable = 20

# Вызов функции для изменения глобальной переменной
modify_global()

# Проверка изменения значения глобальной переменной
print("After modification:", global_variable)  # Выведет: After modification: 20
```

Важно использовать ключевое слово global перед именем переменной, чтобы указать, что вы хотите изменить глобальную переменную, а не создать новую локальную переменную внутри функции.

Предостережение по использованию глобальных переменных.

- Избегайте чрезмерного использования: Использование глобальных переменных может сделать код менее читаемым и управляемым.
- Предотвращение неожиданных изменений: Изменение глобальных переменных внутри функций может привести к неожиданным побочным эффектам. Будьте осторожны и понимайте, какие части программы могут изменять глобальные переменные.
- Разделение ответственности: Часто лучше избегать зависимости функций от глобальных переменных, чтобы функции были более автономными и переиспользуемыми.