IT CAREER HUB

Урок 23.

Документация. Аннотации типов

IT CAREER HUB

Документация

Документация — это описание кода, которое помогает разработчикам понимать, как работают функции и другие компоненты программы. Она делает код более понятным, облегчает его поддержку и расширение.

Docstrings (Документирующие строки) — это строки документации, заключённые в тройные кавычки (""" или '''), которые используются для описания функций и других компонентов. В отличие от комментариев, docstrings являются частью функции и могут быть получены через `help()`.

Синтаксис:

Python
```python
def function_name(param1, param2):
    """
    Описание функции.

    :param param1: Описание первого параметра.
    :param param2: Описание второго параметра.
    :return: Описание возвращаемого значения.
    """
    pass
```

Пример использования:

Python
```python
def greet(name):
    """
    Функция принимает имя и возвращает строку приветствия.
```

```
    :param name: Имя пользователя.
    :return: Приветственное сообщение.
    """
    return f"Hello, {name}!"
```

Автогенерация docstrings в средах разработки

Во многих средах разработки (например, PyCharm, VS Code) базовый шаблон docstrings создаётся автоматически при вводе тройных кавычек (""" или ''') и переноса строки между ними, сразу после объявления функции.

Пример:

```python
def greet(name):
    """



    :param name:
    :return:
    """
    return f"Hello, {name}!"
```

IT CAREER HUB

Функция help

Функция `help()` используется для просмотра встроенной и пользовательской документации объектов, например функций. Она выводит docstrings объекта, если они определены.

Синтаксис:

```python
help(object)
```

Параметры:

- `object` — любой объект Python, для которого нужно получить справку.

Если объект не указан, откроется интерактивный справочный режим.

Примеры использования:

1. **Получение справки по встроенной функции:**

```python
help(sum)
```

2. **Просмотр документации пользовательской функции:**

```python
def greet(name):
    """
    Функция принимает имя и возвращает строку приветствия.

    :param name: Имя пользователя.
    :return: Приветственное сообщение.
```

```
    """

    return f"Hello, {name}!"
```

3. **Получение списка методов объекта:** Функция `help()` также показывает методы и атрибуты объектов.

```python
# Выведет полное описание всех методов строк в Python

help(str)
```

4. **Вызов справки без аргументов:** Если вызвать `help()` без аргумента, откроется интерактивный режим справки:

```python
help()
```

После этого можно вводить имена объектов для получения информации.

IT CAREER HUB

Задания для закрепления

Какой результат будет выведен при выполнении следующего кода?

```
help(print)
```

a. Выведется документация по функции `print()`
b. Напечатается инструкция по использованию `help()`
c. Ошибка, нет скобок у `print`

Можно ли использовать `help()` без аргументов?

Ответ: Да, это откроет интерактивную справку.

IT CAREER HUB

Аннотации типов

Аннотации типов — это механизм, позволяющий указывать ожидаемые типы аргументов и возвращаемого значения функции, а также типы переменных. Они помогают сделать код понятнее, повысить читаемость и упростить отладку, но не накладывают жёстких ограничений на типы данных.
Python остаётся динамически типизированным языком, и аннотации типов не заставляют интерпретатор проверять соответствие типов во время выполнения.

Зачем нужны аннотации типов?

- **Упрощение понимания кода** — легче понять, какие данные ожидаются на входе и выходе.
- **Помощь в отладке** — статический анализ (проверка кода без выполнения) может обнаружить потенциальные ошибки.
- **Автодокументирование** — IDE могут использовать аннотации для подсказок.

Синтаксис:

Python

```
def function_name(param1: type1, param2: type2, ...) -> return_type:
    # тело функции
    return value


variable: type = value
```

- `param1, param2` — имена параметров функции.
- `type1, type2, type` — ожидаемые типы данных для соответствующих параметров или переменных.
- `-> return_type` — аннотация типа возвращаемого значения.

Пример аннотации типов:

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

    return a + b



num: int = 10
```

- `a: int` — аргумент `a` должен быть целым числом.
- `b: int` — аргумент `b` должен быть целым числом.
- `-> int` — функция должна возвращать целое число.
- `num: int` — в переменной можно хранить только целое число.

Аннотации не влияют на выполнение кода:

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

    return a + b



print(add(3, 5))  # не приведёт к ошибке, хотя переданы строки

print(add("3", "5"))
```

Аннотации для базовых типов

В большинстве случаев название аннотации совпадает с названием типа данных, который она обозначает. Это делает код интуитивно понятным и легко читаемым.

1. **Аннотация целых чисел (int):**

```python
def factorial(n: int) -> int:

    """Возвращает факториал числа."""
```

```
    result = 1

    for i in range(1, n + 1):

        result *= i

    return result
```

2. **Аннотация чисел с плавающей точкой (float):**

Python
```
def convert_to_celsius(fahrenheit: float) -> float:

    """Конвертирует температуру из градусов Фаренгейта в Цельсий."""

    return (fahrenheit - 32) * 5 / 9
```

3. **Аннотация строк (str):**

Python
```
def greet(name: str) -> str:

    """Возвращает приветственное сообщение."""

    return f"Hello, {name}!"
```

4. **Аннотация логических значений (bool):**

Python
```
def is_even(number: int) -> bool:

    """Определяет, является ли число чётным."""

    return number % 2 == 0
```

5. **Аннотация None (функция без возврата):**

```python
def log_message(message: str) -> None:

    """Выводит сообщение в консоль, но не возвращает значения."""

    print(f"LOG: {message}")
```

Аннотации для структур данных

В Python 3.9+ можно аннотировать списки, кортежи, множества и словари просто указывая встроенные типы. Однако важно понимать, как именно указывать тип содержимого и какие особенности у разных структур.

1. **Списки (list):**
   Синтаксис: `list[element_type]`
   В скобках указывается тип элементов внутри списка. Рекомендуется указывать тип содержимого.

```python
def process_numbers(numbers: list[int]) -> list[int]:

    # Список чисел

    return [n ** 2 for n in numbers]
```

2. **Кортежи (tuple):**
   Синтаксис:

- Если кортеж содержит фиксированное число элементов: `tuple[type1, type2]`
- Если длина кортежа не фиксирована: `tuple[element_type, ...]`

Примеры:

```python
def get_info() -> tuple[str, float]:

    # Первый элемент str, второй элемент float
```

```
    return "Bob", 4.91


def variable_tuple() -> tuple[int, ...]:
    # Кортеж произвольной длины, но только целые числа
    return 5, 8, 2
```

3. **Множества (set):**
   Синтаксис: `set[element_type]`
   В скобках указывается тип всех элементов множества. Рекомендуется указывать тип содержимого.

```python
def unique_chars(text: str) -> set[str]:
    # Множество уникальных символов
    return set(text)
```

4. **Замороженные множества (frozenset):**
   Синтаксис: `frozenset[element_type]`
   Работает аналогично `set`, но создаёт неизменяемое множество. Рекомендуется указывать тип содержимого.

```python
def frozen_example() -> frozenset[int]:
    # Множество уникальных чисел
    return frozenset([1, 2, 3])
```

5. **Словари (dict):**
   Синтаксис: `dict[key_type, value_type]`
   В скобках указываются типы ключей и значений. Рекомендуется указывать типы содержимого.

```python
def count_words(text: str) -> dict[str, int]:

    """Принимает строку и возвращает словарь с подсчётом каждого слова."""

    words = text.split()

    # Словарь, где ключи — строки, значения — целые числа

    return {word: words.count(word) for word in words}
```

Коллекции без указания типов

Если тип элементов в коллекции не важен или вы хотите создать коллекцию с элементами разных типов, типы можно не указывать.

Пример:

```python
# Тип элементов не указан

def process_data(data: list) -> list:

    return [d for d in data if d]



# Элементы должны быть числами

def process_numbers(numbers: list[int]) -> list[int]:

    return [n ** 2 for n in numbers]
```

Аннотация в старых версиях

В версиях Python до 3.9 для аннотации структур данных использовались специальные классы из модуля `typing`:

- `List` вместо `list`
- `Tuple` вместо `tuple`
- `Set` вместо `set`

- `FrozenSet` вместо `frozenset`
- `Dict` вместо `dict`

Пример различий в аннотациях:

Python

```python
# В Python 3.9+ (рекомендуется)

def process_numbers(numbers: list[int]) -> list[int]:

    return [n ** 2 for n in numbers]



# В Python <3.9 (старый стиль)

from typing import List



def process_numbers_old(numbers: List[int]) -> List[int]:

    return [n ** 2 for n in numbers]
```

IT CAREER HUB

Задания для закрепления

Найдите ошибку в коде и исправьте её

```python
def greet(name: str) -> str:

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



result = greet("Alice")

print(result.upper())
```

Исправленный вариант:

```python
def greet(name: str) -> str:

    return f"Hello, {name}!"  # Добавлен return вместо print



result = greet("Alice")

print(result.upper())
```

Что произойдёт, если вызвать функцию с неправильным типом аргумента?

Ответ: Аннотации типов не проверяются во время выполнения, поэтому код выполнится, но может вызвать неожиданное поведение.

В чём разница между `tuple[str, int]` и `list[str]` в аннотациях?

Ответ: `tuple[str, int]` означает кортеж фиксированной длины, а `list[str]` — список произвольной длины, содержащий строки.

Аннотации Any, Union и Optional

При аннотировании функций и переменных в Python бывают случаи, когда необходимо указывать разные возможные типы. Для этого используются `Any`, `Union` и `Optional` из модуля `typing`.

`Any` — любой тип

`Any` означает, что переменная или аргумент могут быть любого типа.

Пример:

```python
from typing import Any


def process_data(data: Any) -> str:
    """Принимает данные любого типа и возвращает строку с их представлением."""
    return f"Данные: {data}"


print(process_data(42))
print(process_data("Hello"))
print(process_data([1, 2, 3]))
```

`Union` — несколько возможных типов

`Union` позволяет указать, что переменная может содержать один из нескольких типов.

Пример:

```python
from typing import Union
```

```python
def calculate(value: Union[int, float]) -> float:
    """Принимает число (целое или дробное) и возвращает его квадрат."""
    return value ** 2


print(calculate(5))
print(calculate(2.5))
```

`Optional` — значение может быть `None`

`Optional[T]` — сокращённая запись для `Union[T, None]`, означающая, что значение может быть `None`.

Пример:

Python

```python
from typing import Optional


def get_user_name(user_id: int) -> Optional[str]:
    """Возвращает имя пользователя или None, если пользователь не найден."""
    users = {1: "Alice", 2: "Bob"}
    return users.get(user_id)  # Может вернуть None


print(get_user_name(1))
print(get_user_name(3))
```

Когда использовать `Optional`?

- Если функция может не возвращать значение.
- Если переменная может содержать объект или None.
- Когда значение может отсутствовать (например, поиск в базе данных).

`Union` с оператором `|`

В Python 3.10 появился более краткий синтаксис для аннотаций типов — вместо `Union` можно использовать оператор `|`.

Пример:

Python

```python
def calculate(value: int | float) -> float:
    """Принимает число (целое или дробное) и возвращает его квадрат."""
    return value ** 2


print(calculate(5))
print(calculate(2.5))
```

Аннотация Callable Callable используется для аннотирования аргументов или возвращаемого значения, если параметр должен быть функцией. Пример:

Python

```python
from typing import Callable


def add(x: int, y: int) -> int:
    return x + y


def multiply(x: int, y: int) -> int:
    return x * y
```

```
def execute_function(func: Callable[[int, int], int], nums1: list[int], nums2:
list[int]) -> list[int]:

    return [func(a, b) for a, b in zip(nums1, nums2)]



print(execute_function(add, [1, 2, 3], [4, 5, 6]))

print(execute_function(multiply, [1, 2, 3], [4, 5, 6]))
```

IT CAREER HUB

Задания для закрепления

Какой модуль нужен для использования Any, Union и Optional?
Ответ: `typing`.

Как записать аннотацию, если функция возвращает либо строку, либо None?
Ответ: `-> Optional[str]` (или `-> Union[str, None]`).

Передача неизменяемых и изменяемых объектов

При передаче аргументов в функцию важно учитывать, являются ли они изменяемыми или неизменяемыми. Это влияет на то, будут ли изменения, внесённые в аргументы внутри функции, затрагивать оригинальные объекты.

Передача по значению и передача по ссылке

В Python объекты передаются в функции по ссылке, но важно понимать, как это работает с изменяемыми и неизменяемыми типами.

- **Передача по значению (копия)** – это передача ссылки на объект, который нельзя изменить. Если объект изменяется внутри функции, создаётся новый объект, а оригинал остаётся неизменным. *Применяется к неизменяемым (int, float, bool, str, tuple, frozenset).*

- **Передача по ссылке (оригинал изменяется)** – это передача ссылки на объект, который можно изменить. Изменение внутри функции затрагивает сам объект. *Применяется к изменяемым (list, dict, set).*

Пример передачи по значению (копия, неизменяемый объект):

Python

```python
def modify_value(n: int) -> None:
    """Изменяет значение переменной внутри функции (не влияет на оригинал)."""
    print(f"До изменения в функции: {n}, id: {id(n)}")
    n += 1  # Создаётся новый объект
    print(f"После изменения в функции: {n}, id: {id(n)}")

num = 10
modify_value(num)
```

```
print(f"Вне функции: {num}, id: {id(num)}")
```

Пример передачи по ссылке (оригинал изменяется, изменяемый объект):

```python
def modify_list(lst: list) -> None:
    """Добавляет элемент в список (изменяет оригинальный объект)."""
    print(f"До изменения в функции: {lst}, id: {id(lst)}")
    lst.append(99)  # Изменение оригинального списка
    print(f"После изменения в функции: {lst}, id: {id(lst)}")


my_list = [1, 2, 3]
modify_list(my_list)
print(f"Вне функции: {my_list}, id: {id(my_list)}")
```

Как избежать нежелательных изменений?

Если необходимо передавать изменяемый объект, но не изменять его внутри функции, можно использовать:

- Создание копии (`copy()`) для поверхностного копирования.
- Использование `deepcopy()` для полного копирования вложенных структур.

Пример с `copy()`:

```python
def safe_modify_list(lst: list) -> list:
    """Работает с копией списка, оставляя оригинальный неизменным."""
    copy_lst = lst.copy()
```

```
    copy_lst.append(99)

    return copy_lst


original_list = [1, 2, 3]

new_list = safe_modify_list(original_list)

print(f"Оригинал: {original_list}")

print(f"Копия: {new_list}")
```

IT CAREER HUB

Практические задания

Что произойдёт с переданным списком, если изменить его внутри функции?
Ответ: Изменения затронут оригинальный список, так как он передаётся по ссылке.

Что делает метод `.copy()` у списка?
Ответ: Создаёт поверхностную копию списка, которая не влияет на оригинальный список.

Какой результат будет выведен при выполнении следующего кода?

```python
def modify_string(text: str) -> str:

    text += "!"

    return text



original = "Hello"

modify_string(original)

print(original)
```

a) HelloHello!
b) Hello!Hello
c) Hello! Hello
d) Hello

Ответ: d

Практические задания

1. Аннотация структур данных Напишите функцию, которая принимает список строк и возвращает словарь, где ключи — строки, а значения — длина этих строк. Добавьте документацию и аннотации типов для всех параметров и возвращаемого значения.

Данные: words = ["apple", "banana", "cherry"]

Пример вывода: {'apple': 5, 'banana': 6, 'cherry': 6}

Решение:

```python
def get_word_lengths(words: list[str]) -> dict[str, int]:
    """
    Возвращает словарь, где ключи — строки, а значения — их длины.

    :param words: Список строк.
    :return: Словарь с длинами строк.
    """
    return {word: len(word) for word in words}


words = ["apple", "banana", "cherry"]
print(get_word_lengths(words))
```

2. Генерация отчёта Напишите функцию, которая принимает имя пользователя и необязательный список его достижений. Если список пуст, возвращается сообщение "Нет достижений". Если список не пуст, возвращается строка с перечислением достижений. Добавьте документацию и аннотации типов для всех параметров и возвращаемого значения.

Данные: name = "Alice" achievements = ["Won chess tournament", "Completed marathon", "Published a book"]

Пример вывода: Alice: Won chess tournament, Completed marathon, Published a book

Решение:

```python
from typing import Optional
```

```
def generate_report(name: str, achievements: Optional[list[str]] = None) ->
str:
    """
    Генерирует отчёт о достижениях пользователя.

    :param name: Имя пользователя.
    :param achievements: Список достижений пользователя (необязательный).
    :return: Строка с отчётом.
    """
    if not achievements:
        return f"{name}: Нет достижений"
    return f"{name}: {', '.join(achievements)}"

name = "Alice"
achiev = ["Won chess tournament", "Completed marathon", "Published a book"]
print(generate_report(name, achiev))
print(generate_report("Bob"))
```

Обработка элементов списка функцией

Напишите функцию, которая принимает другую функцию и список произвольных элементов. Примените переданную функцию ко всем элементам списка и верните новый список. Добавьте документацию и аннотации типов для всех параметров и возвращаемого значения.

Данные: numbers = [1, 2, 3, 4, 5]

Пример вывода: [2, 4, 6, 8, 10]

Решение:

Python

```python
from typing import Callable, Any


def apply_to_all(func: Callable[[Any], Any], elements: list[Any]) -> list[Any]:
    """
    Применяет переданную функцию ко всем элементам списка.

        :param func: Функция, принимающая элемент списка и возвращающая обработанное значение.
    :param elements: Список произвольных элементов.
    :return: Новый список с результатами применения функции.
    """
    return [func(num) for num in elements]


numbers = [1, 2, 3, 4, 5]
result = apply_to_all(lambda x: x * 2, numbers)
print(result)
```