Документация. Файлы. Модули. Пакеты

Наумов Д.А., доц. каф. КТ

Основы программной инженерии, 2020

Содержание лекции

PEP8 - стиль кода в языке Python

Документирование кода в python – достаточно важный аспект, ведь от нее порой зависит читаемость и быстрота понимания вашего кода, как другими людьми, так и вами через полгода.

PEP 257 описывает соглашения, связанные со строками документации *python*, рассказывает о том, как нужно документировать python код.

- цель этого PEP – стандартизировать структуру строк документации: что они должны в себя включать, и как это написать (не касаясь вопроса синтаксиса строк документации).
- PEP описывает соглашения, а не правила или синтаксис.
- При нарушении этих соглашений, самое худшее, чего можно ожидать – некоторых неодобрительных взглядов.
- Некоторые программы (например, docutils), знают о соглашениях.

Строки документации

Строки документации – строковые литералы, которые являются первым оператором в модуле, функции, классе или определении метода.

Такая строка документации становится специальным атрибутом __doc__ этого объекта.

- Все модули должны, как правило, иметь строки документации, и все функции и классы, экспортируемые модулем также должны иметь строки документации.
- Публичные методы (в том числе __init__) также должны иметь строки документации.
- Пакет модулей может быть документирован в __init__.py.
- Для согласованности, всегда используйте """double quotes"""для строк документации.
- Используйте r"""raw double quotes""", если вы будете использовать обратную косую черту в строке документации.

Однострочные строки документации

```
def kos_root():
        """Return the pathname of the KOS root directory."""
        global _kos_root
        if _kos_root:
                return _kos_root
```

- Используйте тройные кавычки, даже если документация умещается на одной строке.
- Не добавляйте пустых строк перед или после документации.

Однострочная строка документации не должна быть «подписью» параметров функции / метода:

```
def function(a, b):
        """function(a, b) -> list"""
```

Многострочные строки документаци

Многострочные строки документации состоят из однострочной строки документации с последующей пустой строкой, а затем более подробным описанием.

```
def complex(real=0.0, imag=0.0):
        """Form a complex number.

        Keyword arguments:
        real -- the real part (default 0.0)
        imag -- the imaginary part (default 0.0)
        """

        if imag == 0.0 and real == 0.0:
                return complex_zero
```

Первая строка может быть использована автоматическими средствами индексации, поэтому важно, чтобы она находилась на одной строке и была отделена от остальной документации пустой строкой.

Строки документации скрипта

- должны быть доступны в качестве «сообщения по использованию», напечатанной, когда программа вызывается с некорректными или отсутствующими аргументами (или, возможно, с опцией -h, для помощи).
- такая строка документации должна документировать функции программы и синтаксис командной строки, переменные окружения и файлы.
- сообщение по использованию может быть довольно сложным (несколько экранов) и должно быть достаточным для нового пользователя для использования программы должным образом, а также полный справочник со всеми вариантами и аргументами для искушенного пользователя.

Строки документации модуля

Строки документации модуля должны, как правило, перечислять:

- классы,
- исключения,
- функции (и любые другие объекты),

которые экспортируются модулем, с краткими пояснениями (в одну строчку) каждого из них.

Строки документации пакета модулей (т.е. строка документации в __init__.py) также должны включать модули и подпакеты.

Строки документации функции/метода

Строки документации функции или метода должны и документировать:

- аргументы,
- возвращаемые значения
- побочные эффекты,
- исключения,
- дополнительные аргументы,
- именованные аргументы,
- ограничения на вызов функции.

Строки документации класса

Строки документации класса обобщают его поведение и перечисляют:

- открытые методы
- переменные экземпляра.

Если класс предназначен для подклассов, и имеет дополнительный интерфейс для подклассов, этот интерфейс должен быть указан отдельно (в строке документации).

Конструктор класса должен быть задокументирован в документации метода `__init__`.

Отдельные методы должны иметь свои строки документации.

Содержание лекции

Назначение аннотаций

Аннотации нужны для того чтобы повысить информативность исходного кода, и иметь возможность с помощью специальных инструментов производить его анализ.

Согласно PEP 3107 могут быть следующие варианты использования аннотаций:

- проверка типов;
- расширение функционала IDE в части предоставления информации об ожидаемых типах аргументов и типе возвращаемого значения у функций;
- перегрузка функций и работа с дженериками;
- взаимодействие с другими языками;
- использование в предикатных логических функциях;
- маппинг запросов в базах данных;
- маршалинг параметров в RPC (удаленный вызов процедур).

Контроль типов

Один из возможных вариантов (наверное самый логичный) решения данной задачи – это использование комментариев, составленных определенным образом.

```
name = "John" # type: str
```

Мы создали переменную с именем name и предполагаем, что ее тип - str. Естественно, для самого интерпретатора Python это не имеет значения, мы, без труда можем продолжить нашу мини программу таким образом:

```
name = "John" # type: str
print(name)
name = 10
print(name)
```

И это будет корректно с точки зрения Python.

Контроль типов

Один из возможных вариантов (наверное самый логичный) решения данной задачи – это использование комментариев, составленных определенным образом.

```
name = "John" # type: str
```

Мы создали переменную с именем name и предполагаем, что ее тип - str. Естественно, для самого интерпретатора Python это не имеет значения, мы, без труда можем продолжить нашу мини программу таким образом:

```
name = "John" # type: str
print(name)
name = 10
print(name)
```

И это будет корректно с точки зрения Python.

Контроль типов

Если необходимо проконтролировать, что переменной name будут присваиваться значения только строкового типа, мы должны:

1. указать в комментарии о нашем намерение
2. использовать специальный инструмент, который выполнит соответствующую проверку.

Установка модуля с помощью pip:

```
python -m pip install mypy
```

Если мы сохраним приведенный выше код в файле type_tester.py и выполним следующую команду:

```
python -m mypy test_type.py
```

Получим такое сообщение:

```
test_type.py:3: error: Incompatible types in assignment
(expression has type "int", variable has type "str")
```

Оно говорит о том, что обнаружено несоответствие типов в операции присваивание: переменная имеет тип str, а ей присвоено значение типа int.

Обзор PEP'ов регламентирующий работу с аннотациями

Работу с аннотациями регламентируют четыре ключевых документа:

1. PEP 3107 — Function Annotations
   - описывается синтаксис использования аннотаций в функциях Python
   - аннотации не имеют никакого семантического значения для интерпретатора Python и предназначены только для анализа сторонними приложениями
   - аннотировать можно аргументы функции и возвращаемое ей значение.
2. PEP 484 — Type Hints
3. PEP 526 — Syntax for Variable Annotations
4. PEP 563 — Postponed Evaluation of Annotations

Обзор PEP'ов регламентирующий работу с аннотациями

Работу с аннотациями регламентируют четыре ключевых документа:

1. PEP 3107 — Function Annotations
2. PEP 484 — Type Hints
   - представлены рекомендации по использованию аннотаций типов
   - аннотация типов упрощает статический анализ кода, рефакторинг, контроль типов в рантайме и кодогенерацию, использующую информацию о типах
   - определены следующие варианты работы с аннотациями:
     - использование аннотаций в функциях согласно PEP 3107,
     - аннотация типов переменных через комментарии в формате # type: type_name
     - использование stub-файлов.
3. PEP 526 — Syntax for Variable Annotations
4. PEP 563 — Postponed Evaluation of Annotations

Обзор PEP'ов регламентирующий работу с аннотациями

Работу с аннотациями регламентируют четыре ключевых документа:

1. PEP 3107 — Function Annotations
2. PEP 484 — Type Hints
3. PEP 526 — Syntax for Variable Annotations
   - приводится описание синтаксиса для аннотации типов переменных (базируется на PEP 484), использующего языковые конструкции, встроенные в Python.
4. PEP 563 — Postponed Evaluation of Annotations
   - предлагается использовать отложенную обработку аннотаций, это позволяет определять переменные до получения информации об их типах и ускоряет выполнение программы.

Использование аннотаций в функциях

В функциях мы можем аннотировать аргументы и возвращаемое значение.

```
def repeater(s: str, n: int) -> str:
        return s * n
```

- аннотация для аргумента определяется через двоеточие после его имени;
- аннотация, определяющая тип возвращаемого функцией значения, указывается после ее имени с использованием символов -> ;
- доступ к аннотациям функции можно получить через атрибут __annotations__;
- аннотации представлены в виде словаря, где ключами являются атрибуты, а значениями - аннотации;
- возвращаемое функцией значение хранится в записи с ключом return;
- для lambda-функций аннотации не поддерживаются.

Создание аннотированных переменных

Можно использовать один из трех способов создания аннотированных переменных:

```
var = value # type: annotation
var: annotation; var = value
var: annotation = value
```

Рассмотрим это на примере работы со строковой переменной с именем name:

```
name = 'John' # type: str
name: str; name = 'John'
name: str = 'John'
```

Создание аннотированных переменных

```
from typing import List, Tuple
# список
scores: List[int] = [1 ]
# кортеж
pack: Tuple[int, int, int] = (1 , 2, 3)
# логическая переменная
flag: bool
flag = True
# класс
class Point:
        x: int
        y: int
        def __init__(self, x: int, y: int):
                self.x = x
                self.y = y
```

Контроль типов с использованием аннотаций

Для проверки можно использовать уже знакомый нам инструмент mypy. Напишем вот такой код:

```
a: int = 10
b: int = 15
def sq_sum(v1: int, v2: int) -> int:
        return v1 ** 2 + v2 ** 2
print(sq_sum(a, b))
```

Сохраним его в файле с именем work.py и запустим mypy для анализа:

> python -m mypy work.py

Если не указывать дополнительные ключи, то окно консоли будет чистым, т. к. mypy не найдет никаких ошибок в вашем коде.

Контроль типов с использованием аннотаций

Но если заменить первую строку:

```
a: int = 10
```

на такую:

```
a: int = 10.3
```

и вновь запустить mypy, то увидим вот такое сообщение:

```
work.py:7: error: Argument 1 to 'sq_sum' has incompatible type
```

При этом, естественно, код будет выполняться без всяких проблем, потому что интерпретатор Python в данном случае не обращает внимание на аннотации.

Содержание лекции

Инструкция import

Модулем в Python называется любой файл с программой.

- Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам.
- При импорте модуля происходит его выполнение (интерпретация).

Подключить модуль можно с помощью инструкции **import**.

```
import os
os.getcwd()
```

Одной инструкцией можно подключить несколько модулей, хотя этого не рекомендуется делать, так как это снижает читаемость кода.

```
import time, random
print(time.time())
print(random.random())
```

После импортирования модуля его название становится переменной, через которую можно получить доступ к атрибутам модуля.

```
import math
print(math.e)
```

Если указанный атрибут модуля не будет найден, возбудится исключение AttributeError. А если не удастся найти модуль для импортирования, то ImportError.

```
import notexist
Traceback (most recent call last):
File "", line 1, in import notexist
ImportError: No module named 'notexist'
import math
print(math.Ë)
Traceback (most recent call last):
File "", line 1, in math.Ë
AttributeError: 'module' object has no attribute 'Ë'
```

Инструкция from

Подключить определенные атрибуты модуля можно с помощью инструкции from. Она имеет несколько форматов:

```
from <Название модуля> import <Атрибут 1> [ as <Псевдоним 1> ]
from <Название модуля> import *
```

Первый формат позволяет подключить из модуля только указанные вами атрибуты. Для длинных имен также можно назначить псевдоним, указав его после ключевого слова as.

```
from math import e, ceil as c
print(e)
print(c(4.6))
```

Импортируемые атрибуты можно разместить на нескольких строках, если их много, для лучшей читаемости кода:

```
from math import (sin, cos,
                  tan, atan)
```

Второй формат инструкции from позволяет подключить (почти) все переменные из модуля:

```
from sys import *
print(version)
```

Для длинных имен также можно назначить псевдоним, указав его после ключевого слова as.

```
from math import e, ceil as c
print(e)
print(c(4.6))
```

- если в модуле определена переменная __all__, то будут подключены только атрибуты из этого списка.
- если переменная __all__ не определена, то будут подключены все атрибуты, не начинающиеся с нижнего подчёркивания.
- импортирование всех атрибутов из модуля может нарушить пространство имен главной программы.

Создание своего модуля

Создадим файл mymodule.py, в которой определим какие-нибудь функции:

```
def hello():
        print('Hello, world!')

def fib(n):
        a = b = 1
        for i in range(n - 2):
                a, b = b, a + b
        return b
```

Теперь в этой же папке создадим другой файл, например, main.py:

```
        import mymodule

        mymodule.hello()
        print(mymodule.fib(10))
```

При импортировании модуля его код выполняется полностью, то есть, если программа что-то печатает, то при её импортировании это будет напечатано. Этого можно избежать, если проверять, запущен ли скрипт как программа, или импортирован.

```
def hello():
        print('Hello, world!')

def fib(n):
        a = b = 1
        for i in range(n - 2):
                a, b = b, a + b
        return b

if __name__ == "__main__":
        hello()
        for i in range(10):
                print(fib(i))
```

Содержание лекции

Открытие файла

Прежде, чем работать с файлом, его надо открыть.

```
f = open('text.txt', 'r')
```

У функции open много параметров, рассмотрим первые два:

- первый – имя файла. Путь к файлу может быть относительным или абсолютным.
- второй – режим, в котором будет открыт файл.

Режим	Обозначение
'r'	открытие на чтение (является значением по умолчанию).
'w'	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый.
'x'	открытие на запись, если файла не существует, иначе исключение.
'a'	открытие на дозапись, информация добавляется в конец файла.
'b'	открытие в двоичном режиме.
't'	открытие в текстовом режиме (является значением по умолчанию).
'+'	открытие на чтение и запись

Чтение из файла

Метод read читает весь файл целиком, если был вызван без аргументов, и n символов, если был вызван с аргументом (целым числом n).

```
> f = open('text.txt')
> f.read(1)
'H'
> f.read()
'ello world!\nThe end.\n\n'
```

Ещё один способ сделать это - прочитать файл построчно, воспользовавшись циклом for:

```
f = open('text.txt')
for line in f:
        line
```

Запись в файл

```
> l = [str(i)+str(i-1) for i in range(20)]
> l
['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98',
'1110', '1211', '1312', '1413', '1514', '1615', '1716', '1817',
```

Откроем файл на запись:

```
> f = open('text.txt', 'w')
```

Запись в файл осуществляется с помощью метода write:

```
> for index in l:
        f.write(index + '\n')
4
3
3
3
3
```

Запись в файл

```
> l = [str(i)+str(i-1) for i in range(20)]
> l
['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98',
'1110', '1211', '1312', '1413', '1514', '1615', '1716', '1817'
```

Откроем файл на запись:

```
> f = open('text.txt', 'w')
```

Запись в файл осуществляется с помощью метода write:

```
> for index in l:
        f.write(index + '\n')
```

После окончания работы с файлом его необходимо закрыть:

```
> f.close()
```

Менеджеры контекста

Конструкция with ... as используется для оборачивания выполнения блока инструкций менеджером контекста.

Синтаксис конструкции with:

```
with expression [as target] (, expression [as target])* :
        suite
```

Выполнение менеджера контекста:

1. Выполняется выражение в конструкции with ... as.
2. Загружается специальный метод __exit__ для дальнейшего использования.
3. Выполняется метод __enter__. Если конструкция with включает в себя слово as, то возвращаемое методом __enter__ значение записывается в переменную.
4. Выполняется suite.
5. Вызывается метод __exit__. В этот метод передаются параметры исключения, если оно произошло, или во всех аргументах значение None, если исключения не было.

Менеджеры контекста

Самый распространённый пример использования этой конструкции – открытие файлов. Конструкция with гарантирует закрытие файла в любом случае:

```
with open('newfile.txt', 'w', encoding='utf-8') as g:
        d = int(input())
        print('1 / {} = {} '.format(d, 1 / d), file=g)
```

Файл будет закрыт вне зависимости от того, что введёт пользователь.