ЛАБОРАТОРНАЯ РАБОТА

ДОКУМЕНТИРОВАНИЕ КОДА НА PYTHON

Прежде чем мы перейдем к документированию вашего кода на Python, нам нужно отличать документирование от комментирования.

В общем случае **комментирование** - это описание вашего кода для разработчиков. Предполагаемая основная аудитория - разработчики кода на Python и специалисты, которые будет сопровождать код. В сочетании с хорошо написанным кодом комментарии помогают читателю лучше понять ваш код, его назначение и дизайн (архитектуру). **Документирование** кода - это описание его использования и функциональности для ваших пользователей. Хотя это может быть полезно в процессе разработки, основной целевой аудиторией являются пользователи (программисты также могут быть пользователями).

1. Основы комментирования кода

Комментарии создаются на Python с использованием знака # и должны быть краткими описанием, не длиннее нескольких предложений. Вот простой пример:

```
def hello_world():
    # Простой комментарий, предшествующий простому оператору печати
    print("Hello World")
```

Согласно ***PEP 8***, комментарии должны иметь максимальную длину 72 символа. Это условие справедливо даже если в вашем проекте максимальная длина строки будет больше рекомендуемых 80 символов. Если комментарий будет больше, чем ограничение на символ комментария, уместно использовать несколько строк для комментария:

```
def hello_long_world():
    # Очень длинное заявление, которое просто продолжается и продолжается,
    # и продолжается, и продолжается, и никогда не заканчивается, пока
    # не будет достигнуто ограничение в 80 символов
    print("Hellooooooooooooooooooooooooooooooooooooooooooooooooooooooo World")
```

Комментирование вашего кода служит нескольким целям, в том числе:

- **Планирование и проверка**: Когда вы разрабатываете новые части своего кода, может быть целесообразно сначала использовать комментарии как способ планирования или описания этого раздела кода. Не забудьте удалить эти комментарии, как только фактическое кодирование будет реализовано и проверено / протестировано

```
# First step
# Second step
# Third step
```

- **Описание кода**: Комментарии могут использоваться для объяснения цели определенных разделов кода:

```
# Попытка подключения на основе предыдущих настроек. В случае неудачи
# запросите у пользователя новые настройки.
```

- **Описание алгоритмов**: Когда используются алгоритмы, особенно сложные, может быть полезно объяснить, как работает алгоритм или как он реализован в вашем коде. Также может быть уместно описать, почему конкретный алгоритм был выбран вместо другого.

```
# Использование быстрой сортировки для повышения производительности
```

- **Маркировка**: использование тегов может использоваться для обозначения определенных разделов кода, где находятся известные проблемы или области улучшения. Вот некоторые примеры: BUG, FIXME, и TODO.

```
# TODO: Добавить условие, когда val равно None
```

Комментарии к вашему коду должны быть краткими и целенаправленными. По возможности избегайте использования длинных комментариев. Кроме того, вы должны использовать следующие четыре основных правила, предложенные Джеффом Этвудом:

1. Держите комментарии как можно ближе к описываемому коду. Комментарии, которые не соответствуют описываемому ими коду, расстраивают читателя и легко пропускаются при внесении обновлений.
2. Не используйте сложное форматирование (например, таблицы или цифры ASCII). Сложное форматирование приводит к отвлекающему контенту, и со временем его может быть трудно поддерживать.
3. Не включайте избыточную информацию. Предположим, что читатель кода имеет базовое представление о принципах программирования и синтаксисе языка.
4. Разработайте свой код так, чтобы он комментировал сам себя. Самый простой способ понять код - это прочитать его. Когда вы разрабатываете свой код, используя четкие, простые для понимания концепции, читатель сможет быстро концептуализировать ваше намерение.

Помните, что комментарии предназначены для читателя, включая вас, чтобы помочь им понять назначение и дизайн программного обеспечения.

2. Комментирование кода с помощью подсказки типа (Python 3.5+)

Подсказка типа была добавлена в Python 3.5 и является дополнительной формой, помогающей читателям вашего кода. Фактически, это переводит четвертое предложение Джеффа сверху на следующий уровень. Это позволяет разработчику разрабатывать и объяснять части своего кода без комментариев. Вот краткий пример:

```
def hello_name(name: str) -> str:
    return(f"Hello {name}")
```

Изучив подсказку типа, вы можете сразу сказать, что функция ожидает, что входные данные в для аргумента ***name*** будут иметь тип ***str*** (строка). Вы также можете сказать, что ожидаемый результат функции также будет иметь тип ***str***. Хотя указание типа помогает уменьшить количество комментариев, примите во внимание, что это может также потребовать дополнительной работы при создании или обновлении документации проекта.

3. Документирование вашего кода Python с помощью docstrings

Документирование вашего кода на Python полностью сосредоточено на docstrings . Это встроенные строки, которые при правильной настройке могут помочь вашим пользователям и вам самим в работе с документацией вашего проекта. Наряду с docstrings, Python также имеет встроенную функцию `help()`, которая выводит строку docstring объектов на консоль. Вот краткий пример, где мы выводим документацию к объекту строки с помощью функции `help()`:

```
>>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors are specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 # Truncated for readability
```

Можно манипулировать любым другим пользовательским объектом:

```
def say_hello(name):
    print(f"Hello {name}, is it me you're looking for?")

say_hello.__doc__ = "A simple function that says hello... Richie style"
```

```
>>> help(say_hello)
Help on function say_hello in module __main__:

say_hello(name)
    A simple function that says hello... Richie style
```

В Python есть еще одна функция, упрощающая создание строк документации. Вместо прямого управления свойством __doc__ можно разместить блок документации непосредственно под объектом, что автоматически установит значение свойства __doc__. Вот что происходит с тем же самым примером, что и выше:

```
def say_hello(name):
    """A simple function that says hello... Richie style"""  ← вот этот блок в тройных кавычках
    print(f"Hello {name}, is it me you're looking for?")
```

```
>>> help(say_hello)
Help on function say_hello in module __main__:

say_hello(name)
    A simple function that says hello... Richie style
```

3.1. Типы строк документации

Соглашения о строках документации описаны в PEP 257. Их цель - предоставить вашим пользователям краткий обзор объекта. Они должны быть достаточно краткими, чтобы их было легко поддерживать, но при этом достаточно сложными, чтобы новые пользователи могли понять их назначение и как использовать документируемый объект.

Во всех случаях в строках документов следует использовать тройные двойные кавычки (""""). Это следует делать независимо от того, является ли строка документа многострочной или нет. Как минимум, строка документа должна представлять собой краткое изложение того, что вы описываете, и должна содержаться в одной строке:

```
"""Это краткая сводная строка, используемая в качестве описания объекта."""
```

Многострочные строки документации используются для дальнейшей детализации объекта за пределами сводки. Все многострочные строки документации состоят из следующих частей:

- Однострочная сводная строка
- Пустая строка, продолжающая резюме
- Любая дальнейшая разработка для строки документа
- Еще одна пустая строка

```
"""Это сводная строка

Это дальнейшая детализация блока документации. В этом разделе вы можете более
подробно остановиться на деталях, соответствующих ситуации. Обратите внимание,
что сводка и уточнение разделены пустой новой строкой.
"""

# Обратите внимание на пустую строку выше. Код должен продолжаться на этой строке.
```

Все строки документации должны иметь ту же максимальную длину символов, что и комментарии (72 символа). Строки документации можно разделить на три основные категории:

- Строки документации класса: Класс и методы класса
- Строки документации пакетов и модулей: Пакет, модули и функции
- Строки документации скрипта: Сценарий и функции

СТРОКИ ДОКУМЕНТАЦИИ КЛАССА

Строки документации класса создаются для самого класса, а также для любых методов класса. Строки документации размещаются сразу после класса или метода класса с отступом на один уровень:

```
class SimpleClass:
    """Строки документации класса находятся здесь."""

    def say_hello(self, name: str):
        """Строки документации метода класса находятся здесь."""

        print(f'Hello {name}')
```

Строки документации класса должны содержать следующую информацию:

- Краткое описание его назначения и поведения
- Любые общедоступные методы вместе с кратким описанием
- Любые свойства (атрибуты) класса
- Все, что связано с интерфейс для подклассов, если класс предназначен для подклассов

Давайте рассмотрим простой пример класса данных, который представляет Animal . Этот класс будет содержать несколько свойств класса, свойств экземпляра, __init__и один метод экземпляра:

```python
class Animal:
    """
    Класс, используемый для представления животного

    ...какое-то подробное описание...

    Attributes
    ----------
    says_str : str
        форматированная строка для вывода того, что говорит животное
    name : str
        имя животного
    sound : str
        звук, который издает животное
    num_legs : int
        количество ног у животного (по умолчанию 4)

    Methods
    -------
    says(sound=None)
        Печатает имя животного и звук, который оно издает
    """

    says_str = "A {name} says {sound}"

    def __init__(self, name, sound, num_legs=4):
        """
        Parameters
        ----------
        name : str
            Имя животного
        sound : str
            Звук, который издает животное
        num_legs : int, optional
            Количество ног у животного (по умолчанию 4)
        """

        self.name = name
        self.sound = sound
        self.num_legs = num_legs

    def says(self, sound=None):
        """Выводит название животного и звук, который оно издает.

        Если аргумент `звук` не передан, используется звук животного по
        умолчанию.

        Parameters
        ----------
        sound : str, optional
            Звук, который издает животное (по умолчанию нет)

        Raises
        ------
        NotImplementedError
            Если звук животного не задан или передан в качестве параметра.
        """

        if self.sound is None and sound is None:
            raise NotImplementedError("Silent Animals are not supported!")

        out_sound = self.sound if sound is None else sound
        print(self.says_str.format(name=self.name, sound=out_sound))
```

СТРОКИ ДОКУМЕНТАЦИИ СКРИПТА

Скрипты считаются исполняемыми файлами с одним файлом, запускаемыми с консоли. Строки документации для скриптов размещаются в верхней части файла и должны быть задокументированы достаточно хорошо, чтобы пользователи могли иметь достаточное представление о том, как использовать скрипт.

```
"""Печать столбцов таблицы

Этот сценарий позволяет пользователю выводить на консоль все столбцы
электронной таблицы. Предполагается, что первая строка электронной таблицы
является расположением столбцов.

Этот инструмент принимает файлы значений, разделенных запятыми (.csv), а также
файлы Excel (.xls, .xlsx).

Этот скрипт требует, чтобы в среде Python, в которой вы запускаете этот
скрипт, была установлена библиотека Pandas.

Этот файл также может быть импортирован как модуль и содержит следующие функции:

    * get_spreadsheet_cols - возвращает заголовки столбцов файла
    * main - основная функция скрипта
"""

import argparse

import pandas as pd


def get_spreadsheet_cols(file_loc, print_cols=False):
    """
    ...
    """

    file_data = pd.read_excel(file_loc)
    col_headers = list(file_data.columns.values)

    if print_cols:
        print("\n".join(col_headers))

    return col_headers


def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument(
        'input_file',
        type=str,
        help="The spreadsheet file to pring the columns of"
    )
    args = parser.parse_args()
    get_spreadsheet_cols(args.input_file, print_cols=True)


if __name__ == "__main__":
    main()
```

3.2. Форматы Docstring

Возможно, вы заметили, что во всех примерах, приведенных в этом руководстве, присутствует специфическое форматирование с общими элементами: `Arguments`, `Returns`, и `Attributes`. Существуют определенные форматы строк документации, которые можно использовать, чтобы помочь анализаторам строк документации и пользователям иметь знакомый и известный формат. Форматирование, используемое в примерах в этом руководстве, представляет собой строки документации в стиле NumPy / SciPy. Некоторые из наиболее распространенных форматов следующие:

Тип форматирования	Описание	Поддерживается Sphynx	Формальная спецификация
Строки документации Google	Рекомендуемая форма документации Google	ДА	НЕТ
reStructuredText	Официальный стандарт документации на Python; Не подходит для начинающих, но многофункциональен	ДА	ДА
Строки документации NumPy/SciPy	Комбинация NumPy из reStructuredText и Google Docstrings	ДА	ДА
Epytext	Адаптация Epydoc на Python; Отлично подходит для разработчиков Java	Не официально	ДА

Выбор формата docstring зависит от вас, но вы должны придерживаться одного и того же формата во всем вашем документе / проекте. Ниже приведены примеры каждого типа, чтобы дать вам представление о том, как выглядит каждый формат документации.

ПРИМЕР GOOGLE DOCSTRINGS

```
"""Gets and prints the spreadsheet's header columns

Args:
    file_loc (str): The file location of the spreadsheet
    print_cols (bool): A flag used to print the columns to the console
        (default is False)

Returns:
    list: a list of strings representing the header columns
"""
```

RESTRUCTUREDTEXT EXAMPLE

```
"""Gets and prints the spreadsheet's header columns

:param file_loc: The file location of the spreadsheet
:type file_loc: str
:param print_cols: A flag used to print the columns to the console
    (default is False)
:type print_cols: bool
:returns: a list of strings representing the header columns
:rtype: list
"""
```

NUMPY/SCIPY DOCSTRINGS EXAMPLE

```
"""Gets and prints the spreadsheet's header columns

Parameters
----------
file_loc : str
    The file location of the spreadsheet
print_cols : bool, optional
    A flag used to print the columns to the console (default is False)

Returns
-------
list
    a list of strings representing the header columns
"""
```

EPYTEXT EXAMPLE

```
"""Gets and prints the spreadsheet's header columns

@type file_loc: str
@param file_loc: The file location of the spreadsheet
@type print_cols: bool
@param print_cols: A flag used to print the columns to the console
    (default is False)
@rtype: list
@returns: a list of strings representing the header columns
"""
```