Курс “Научный Питон”

Лекция 11
Документирование, тестирование, профилирование и версионирование кода

Документирование

Что такое документирование?

- Документирование кода (классы, функции, интерфейсы, ...)
- Детальное описание используемых алгоритмов, подходов
- Структурное и функциональное описание
- Выполнение требований стандартов (ГОСТ-19, ГОСТ-34)
- Понятие программного продукта (ГОСТ 19) – код + сопроводительная документация.

Далее под документированием понимаем документирование программного кода.

Зачем документировать?

- Доводы за документирование:
  - Для других разработчиков
  - Для себя, чтобы вспомнить через некоторое время
  - Требуется при публикации проекта в интернете (оформление проекта)
  - Для улучшения понимания (детальное описание алгоритмов)
- Доводы против документирования:
  - Простая программа
  - Нет смысла описывать очевидные вещи
  - Захламление кода
  - Кодом больше никто не будет пользоваться
  - Изменение алгоритма требует изменения документации

Требования к документированию

- Простота написания документации
- Поближе к коду
- Возможность использовать формулы, картинки, таблицы, ссылки и др
- Возможность экспортировать в различные форматы (html, pdf и др)
- Поддержка русского языка

Подходы к документированию в Python

- Комментарии # (внутрифайловая документация)
- Функция dir (списки атрибутов)
- Строки документации __doc__
- PyDoc: функция help
- PyDoc: отчеты HTML
- Публикация документации на сайтах в internet
- Печатные издания

Общие советы

- Код как документация
  - Правильно выбирайте названия функций и переменных
  - Разбивайте сложные функции на более простые
  - Декомпозируйте функциональность (модули, пакеты)
- Добавляйте лаконичные комментарии, если кода недостаточно
- Используйте встроенные средства Python для документирования
- В случае сложных проектов можно использовать внешние инструменты
  - Word
  - Latex
  - DocBook

Встраивание документации в код

```
"""
Документация модуля...
"""


def fsingle():
    "Однострочная документация функции"
    pass

def fmultiple():
    '''
    Для многострочных комментариев используются тройные кавычки, однострочные
    фрагменты в одинарных или двойных кавычках вполне хороши, но они не
    разрешают записывать многострочный текст.
    '''
    pass
```

Встраивание документации в код

```
class A:
    '''
    Приведенный под шапкой определения функции или класса текст рассматривается
    как комментарий и будет сохранен в переменной __doc__.
    '''

    def msingle():
        'Многострочный комментарий (первая строка)'
        'Вторая строка не попадет в документацию'
        pass
```

Функция help

- help(obj) – выводит описание существующего объекта (obj.__doc__)
- help("objname") – выводит описание объекта по имени
- help() – интерактивная справка

- help("".replace)
- help(float)
- Help("+")
- help(None)
- help(help(help)))

PyDoc

- python -m pydoc -g, pydoc
- pydoc module (вывод описания в консоли)
- pydoc -b module (запуск web сервера с документацией)
- pydoc

Тестирование

Цели тестирования

- Для проверки соответствия требованиям (качество)
- Для обнаружение проблем на более ранних этапах разработки
- Обнаружение вариантов использования, которые не были предусмотрены
- Тесты как требования (TDD - Test Driven Development)
- Тесты как примеры использования (обучение)

Не тестированный код не работает!!!

Виды тестирования

- Проверки в коде (asserts, __debug__, python -O)
- Unit-тестирование
- Регрессионные тесты
- Интеграционные тесты
- Нагрузочное тестирование
- Стрессовое тестирование
- Тестирование надежности
- Usability тестирование
- ...

Unit-тесты

- Тестируют каждый "кирпичик" независимо
- Автоматические тесты
- Можно написать свой тест для каждой функции, класса, метода
- Тестовый код лучше помещать в отдельный файл

Средства для написания unit-тестов

- Питон модуль unittest
- Питон модуль coverage
- python -m unittest discover
- python -m coverage report

```
import unittest

class MyTestCase(unittest.Testcase):
    def test_name1():
        …
    def test_name2():
        …
```

Проверки

- assertEqual
- assertAlmostEqual
- assertTrue
- assertFalse
- assertListEqual
- assertDictEqual
- assertIs
- assertIsNone
- assertIsNot

```
class
PointTestCase(unittest.TestCase):
    def test_create(self):
        pt = Point(10.1, 20.2)
        self.assertEqual(pt[0], 10.1)
        self.assertEqual(pt[1], 20.2)
```

Практический пример

Написание unit-тестов для классов типов проекта sphereplot

Профилирование

Профилирование

- Профилирование – это сбор характеристик работы программы
  - Время выполнения
  - Количество вызовов функций
  - Количество используемой памяти
  -
- Цели профилирования:
  - Поиск проблемных мест в плане производительности и используемой памяти
  -

Пакет timeit

- Способы использования
  - Внутри кода
  - В командной строке

```
import timeit
tm = timeit.timeit("for x in range(100)",
                   number = 100)
```

```
python -m timeit -n 10 -r 5 -s \
    "sum = 0" "for x in range(100): sum += x"
```

Пакет cProfile

- Запуск из командной строки
  - `python -m cProfile main.py`
- Запуск в коде
  - `import cprofile`
  - `if __name__ == '__main__': cProfile.run('main()')`
- Практический пример: ex.2

Версионирование

GIT

- Файл
- Коммит (идентифицируется хэшом коммита)
- Репозиторий (локальный, удаленный)
- Ветка (branch)
- Тэг (tag)
- Индекс (staging area)

Команды GIT

- Конфигурация
  - git config
- Создание репозитория
  - git init
  - git clone
- Текущее состояние репозитория
  - git status
  - git diff
- Внесение изменений:
  - git add <file>
  - git commit
- Просмотр истории коммитов
  - git log
- Загрузка коммитов из удал. реп.
  - git pull

https://training.github.com/downloads/ru/github-git-cheat-sheet/

github.com

- Платформа для коллективной разработки:
  - Общий репозиторий кода и документации проекта
  - Bug и Issue треккинг
  - Wiki проекта
  - Средства Continuous Integration (GitHub Actions)
  - Средства для публикации сайта проекта в Интернет (GitHub Pages)
  - Средства для редактирования кода online
  -