Урок 7

PEP-8, подготовка документации

PEP-8. Документирование кода. Генератор документации Sphinx.

Введение

На основе курсов второго уровня по Python мы реализовали полноценную клиент-серверную систему обмена сообщениями. Теперь обратим внимание на оформление кода в соответствии с рекомендациями **PEP-8**. Еще рассмотрим подготовку документации для приложения и особенности генератора документации **Sphinx**.

PEP-8. Внешний вид кода

В Python есть руководство по написанию легко читаемого кода — **PEP-8**. PEP расшифровывается как Python Enhancement Proposals — предложение по развитию Python. Документ под номером 8 — это правила написания кода. Рассмотрим их кратко.

Отступы

Для отступа используйте 4 пробела или табуляцию. Нельзя смешивать символы табуляции и пробелы. Последние предпочтительнее.

Максимальная длина строки

Ограничьте максимальную длину строки 79 символами.

Пустые строки

Отделяйте функции верхнего уровня (не функции внутри функций) и определения классов двумя пустыми строками. Определения методов внутри класса — одной. Дополнительные отступы строками можно изредка использовать, чтобы выделить группы логически связанных функций. Пустые строки можно пропускать между несколькими выражениями, записанными в одну строку — например, «заглушки» функций.

Кодировки (PEP 263)

Начиная с версии Python 3.0 предпочтительна кодировка UTF-8.

import-секции

Импортирование разных модулей должно быть на разных строках.

Правильно:

```
import os
import sys
```

Неправильно:

```
import os,sys
```

Но можно писать так:

```
from subprocess import Popen, PIPE
```

Импортирование всегда нужно делать сразу после комментариев к модулю и строк документации, перед объявлением глобальных переменных и констант. Группируйте импорты в следующем порядке:

- импорты стандартной библиотеки;
- импорты сторонних библиотек;
- импорты модулей текущего проекта.

Вставляйте пустую строку между каждой группой импортов. Относительные импорты крайне не рекомендуются — всегда указывайте абсолютный путь к модулю.

Пробелы в выражениях и инструкциях

Избегайте пробелов в следующих ситуациях:

1. Сразу после скобок или перед ними (обычными, фигурными и квадратными).

Можно:

```
spam(ham[1], {eggs: 2})
```

Нельзя:

```
spam( ham[ 1 ], { eggs: 2 })
```

2. Сразу перед запятой, точкой с запятой, двоеточием.

Можно:

```
if x == 4: print x, y; x, y = y, x
```

Нельзя:

```
if x == 4 : print x , y ; x , y = y , x
```

3. Сразу перед открывающей скобкой, после которой начинается список аргументов при вызове функции.

Можно:

```
spam(1)
```

Нельзя:

```
spam (1)
```

4. Сразу перед открывающей скобкой, после которой следует индекс или срез.

Можно:

```
dict['key'] = list[index]
```

Нельзя:

```
dict ['key'] = list [index]
```

5. Использование более одного пробела вокруг оператора присваивания (или любого другого), чтобы выровнять его с оператором на соседней строке.

Можно:

```
x = 1
y = 2
long_variable = 3
```

Нельзя:

```
x             = 1
y             = 2
long_variable = 3
```

Комментарии

Комментарии, которые противоречат коду, — хуже, чем их полное отсутствие. Всегда исправляйте комментарии, если меняете код. Они должны быть законченными предложениями. Если комментарий — фраза или предложение, первое слово должно быть написано с заглавной буквы, если только это не имя переменной, которая начинается со строчной. Кстати, никогда не отступайте от этого правила для имен переменных. Если комментарий короткий, можно опустить точку в конце предложения. Блок комментариев обычно состоит из одного или более абзацев, составленных из полноценных предложений, поэтому каждое из них должно завершаться точкой.

Блок комментариев

Блок комментариев обычно объясняет код (весь или часть), идущий после него, и должен иметь тот же отступ, что и сам код. Каждая строчка такого блока должна начинаться с символа # и одного пробела после него (если только сам текст комментария не имеет отступа). Абзацы внутри блока комментариев лучше отделять строкой, состоящей из одного символа #.

Комментарии в строке с кодом

Старайтесь реже использовать подобные комментарии.

Соглашению по именованию

Важно правильно именовать переменные, функции и классы.

Стили имен

Никогда не используйте символы l (строчная латинская буква «эль»), O (заглавная латинская буква «о») или I (заглавная латинская буква «ай») как однобуквенные идентификаторы. В некоторых шрифтах эти символы неотличимы от цифр 1 и 0. Если очень нужно использовать l-имена, пишите заглавную L.

Имена модулей и пакетов

Модули должны иметь короткие имена, состоящие из строчных букв. Можно использовать и символы подчеркивания, если это улучшает читаемость. Это, за исключением символов подчеркивания, относится и к именам пакетов.

Имена классов

Все имена классов должны следовать соглашению **CapWords** почти без исключений. Классы внутреннего использования могут начинаться с символа подчеркивания.

Имена исключений (exceptions)

Исключения являются классами, поэтому к ним применяется стиль именования классов. Можно добавить **Error** в конце имени (если исключение действительно является ошибкой).

Имена функций

Имена функций должны состоять из строчных букв, а слова — разделяться символами подчеркивания, чтобы код было легче читать.

Аргументы функций и методов

Всегда используйте **self** в качестве первого аргумента метода экземпляра объекта, а **cls** — первого аргумента метода класса.

Если имя аргумента конфликтует с зарезервированным ключевым словом **python**, лучше добавить в конец имени символ подчеркивания, чем исказить написание слова или использовать аббревиатуру. Таким образом, **print_** лучше, чем **prnt**. Вариант — подобрать синоним.

Имена методов и переменные экземпляров классов

Используйте тот же стиль, что и для имен функций: имена должны состоять из строчных букв, а слова разделяться символами подчеркивания. Чтобы избежать конфликта имен с подклассами, добавьте два символа подчеркивания, чтобы включить механизм изменения имен. Если класс **Foo** имеет атрибут с именем **__foo**, к нему нельзя обратиться, написав **Foo.__a** (хотя настойчивый пользователь может получить доступ, написав **Foo._Foo__a**). Двойное подчеркивание в именах должно использоваться, чтобы избежать конфликта имен с атрибутами классов, спроектированных так, чтобы от них наследовали подклассы.

Константы

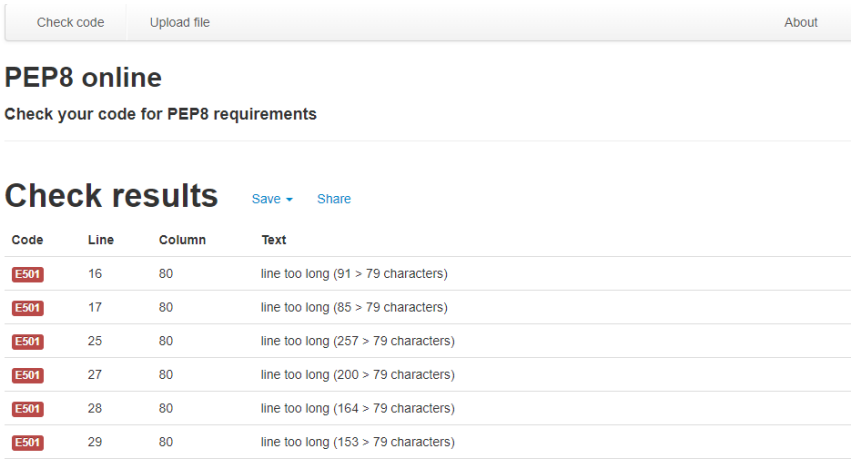
Константы обычно объявляются на уровне модуля и записываются только заглавными буквами, а слова разделяются символами подчеркивания: **MAX_OVERFLOW, TOTAL**.

Инструменты

Оформление кода в соответствии с положениями **PEP-8** — нетривиальная задача для начинающих разработчиков. В помощь программистам предлагается ряд вспомогательных инструментов, позволяющих автоматически определить ошибки.

Онлайн-сканеры

Для поиска ошибок стилизации кода есть онлайн-сервисы — например, **pep8online.com**, где через специальную форму можно ввести программный код и запустить проверку. Пример результата сканирования:



Консольные утилиты

Для контроля качества кода широкое распространение получила утилита **Flake8**. Она сканирует код проекта и ищет в нем стилистические ошибки и нарушения рекомендаций.

Для работы с **Flake8** необходимо установить утилиту:

```
pip install flake8
```

Запуск:

```
flake8 <project_name>
```

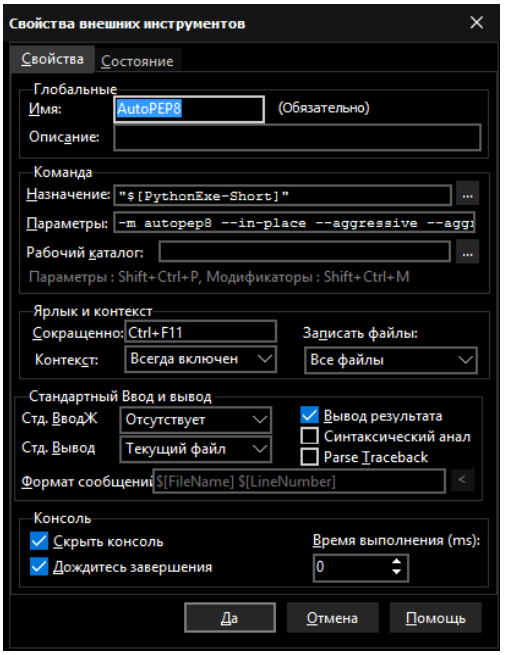
Пример результатов:

```
fQMD_window.py:2314:80: E501 line too long (83 > 79 characters)
fQMD_window.py:2318:5: E265 block comment should start with '# '
```

Средства преобразования кода

Для автоматизации преобразования кода в соответствии с **PEP-8** применяют утилиту **autopep8**, которую можно интегрировать в среду разработки на Python, — например, в свободно распространяемый продукт **PyScripter**.

Для интеграции утилиты в список инструментов **PyScripter** необходимо выполнить ряд шагов: «Инструменты — Настройка инструментов — Добавить». И заполнить форму добавления нового инструмента по шаблону:



«Главное поле формы — ‘Параметры’ содержит значение: **-m autopep8 --in-place --aggressive --aggressive $[ActiveDoc-Short]**».

Подготовка документации

Хорошая документация помогает человеку, работающему с программным обеспечением, понять его особенности и функции. Это может быть спецификация для программистов и тестировщиков, технический документ для внутренних пользователей или программное руководство и файлы справки для конечных пользователей. Документация должна быть специфичной, краткой, актуальной и предоставлять исчерпывающую информацию.

Из-за нехватки времени в процессе разработки отдельные программисты и менеджеры проектов игнорируют написание документации. Впоследствии это делается на скорую руку. Хуже может быть только обновление такой документации. Гораздо проще приступать к документированию в самом начале проекта и расценивать документацию как часть кода.

Преимущества ведения документации в течение всего жизненного цикла продукта:

1. Документация обеспечивает «общее пространство» проекта. Любой участник когда угодно может получить необходимую информацию как по конкретной задаче, так и по общему направлению работы.

2. Команда говорит на одном языке — ведь гораздо проще понять человека, сообщающего об «ошибке в функции, описанной в Use Case №12», чем о «непонятном баге в той функции, которую Вася месяц назад делал».

3. Документирование позволяет разграничить зоны ответственности между участниками проекта.

4. Только тщательно описанные требования могут быть проверены на полноту и непротиворечивость. Наколенные записки — прямой и быстрый путь к тому, что границы проекта расширятся, а функционал не будет отражать пожелания заказчика.

Стоит отметить несколько подходов, позволяющих упростить ведение технической документации:

- Писать в два этапа: сначала идеи, потом вычитывание и форматирование;
- Определить аудиторию: кто будет читать документ?
- Простой стиль. Сохранять простоту и прямолинейность, следить за правописанием:
    - Используйте простые предложения, не длиннее 2 строк текста;
    - Составляйте параграф не более чем из 3–4 предложений, доносящих одну мысль;
    - Не делайте много повторений;
    - Не используйте разные времена — в большинстве случаев можно обойтись настоящим временем;
    - Не используйте шутки в тексте, но в примерах кода они могут быть уместны.
- Ограничивать подачу информации: описывать только один аспект в разделе текста;
- Приводить реалистичные примеры кода: избегать абстрактных «ФуБаров» (**foo/bar**);
- Простота, но достаточность: информация должна раскрывать суть (API, настройка, использование);
- Использовать шаблоны: это поможет читателям привыкнуть к документу и быстро находить информацию.

Существуют утилиты — **генераторы документации**, позволяющие составлять документацию автоматически на основе комментариев кода. Это **doxygen**, **Epydoc**, **Sphinx** — его рассмотрим сегодня.

Генератор документации Sphinx

Sphinx — это инструмент, позволяющий создавать текстовые документы и преобразовывать их в различные форматы. Это удобно при использовании систем управления версиями, предназначенных для отслеживания изменений. Документация в обычном текстовом формате подходит для совместного использования несколькими соавторами, работающими в различных системах. Обычный текст является одним из наиболее переносимых форматов, доступных на сегодняшний день.

Хотя **Sphinx** написан на Python и изначально был разработан для создания документации по этому языку, вовсе не обязательно использовать его только для документирования языков программирования или создания документов для программистов. Sphinx подходит для различных задач, включая написание целых книг.

Sphinx можно рассматривать как платформу для создания документов: он избавляет от рутинных действий и предлагает автоматическую функциональность для решения типовых проблем — например, индексирования заголовков и специального выделения кода (при включении в документ его фрагментов) с соответствующим выделением синтаксиса.

Для работы в **Sphinx** в основном используется интерфейс командной строки, но это не должно вызвать неудобств. В системе должен быть установлен Python.

Синтаксис

Для управления документами **Sphinx** использует синтаксис разметки текста **reStructuredText** (с некоторыми дополнениями). Это вариант текстовой разметки (форматирование текста оформляется прямо в нем).

Разметка позволяет определять и структурировать текст для его вывода в надлежащем формате. Прежде чем начать работу, взгляните на листинг ниже, в котором содержится небольшой пример синтаксиса разметки **reStructuredText**:

```
Это заголовок
=============
Заголовок содержит главную тему и отделяется символами '='.
Их количество должно быть не меньше, чем количество символов
в заголовке.

Подзаголовок
------------
Подзаголовки отделяются символами '-'. Их количество должно
быть тем же, что и количество символов в подзаголовке
(как и в случае с заголовками).

Списки могут быть маркированными:

 * Элемент Foo
 * Элемент Bar

Или же автоматически пронумерованными:

 #. Элемент 1
 #. Элемент 2

Внутренняя разметка
-------------------
Слова можно выделять *наклонным* или **полужирным** шрифтами.
Фрагменты кода (например, примеры команд) можно заключать в обратные кавычки,
например:
команда ``sudo`` дает вам привилегии суперпользователя!
```

Как видим, синтаксис обычного текста вполне удобочитаемый. При создании документа в определенном формате (например, HTML), шрифт заголовка будет большего размера, чем у подзаголовка, а элементы списков будут отображаться соответствующим образом. Добавление дополнительных элементов или изменение их порядка в списках не влияет на нумерацию, а размеры и важность заголовков можно варьировать, изменяя символы подчеркивания под ними.

Установка и настройка

Установка выполняется в командной строке:

```
pip install sphinx
```

Для хранения исходных (в обычном текстовом формате) и конечных (сгенерированных в том или ином формате) файлов в Sphinx используются различные директории. Например, если мы создаем PDF-файл из исходного текстового, PDF будет сохранен в директорию **build**. Эту конфигурацию можно изменить, но в примерах будут использоваться настройки по умолчанию.

Приступим к созданию нового проекта — в консоли выполним команду **sphinx-quickstart**. В процессе вам предложат ответить на несколько вопросов. Нажимайте клавишу **Enter**, чтобы принять все значения по умолчанию:

```
sphinx-quickstart

Welcome to the Sphinx 1.5.3 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Enter the root path for documentation.
> Root path for the documentation [.]:
```

В качестве имени проекта было указано Project Lemon. Вы можете выбрать другое имя.

После выполнения команды **sphinx-quickstart** в рабочей директории должны появиться файлы, аналогичные представленным ниже:

```
|   make.bat
|   Makefile
|
├───build
└───source
    |   conf.py
    |   index.rst
    |
    ├───_static
    └───_templates
```

- **Makefile** — этот файл содержит инструкции для генерации результирующего документа командой make (в UNIX). Разработчики, которым приходилось компилировать код, должны быть знакомы с этим файлом;
- **make.bat** — make-файл для Windows;
- **build** — это директория, в которую будут помещены файлы в определенном формате после того, как будет запустится их генерация;
- **_static** — в эту директорию помещаются все файлы, не являющиеся исходным кодом (например, изображения). Позже создаются связи этих файлов с директорией **build**;
- **_templates** — здесь располагаются файлы шаблонов для генерации выходных форматов;
- **conf.py** — это файл Python, содержащий конфигурационные параметры Sphinx, включая те, которые были выбраны при запуске **sphinx-quickstart** в окне терминала;

- **index.rst** — это корневой файл проекта. Он соединяет документацию воедино, если она разделена на несколько файлов.

Начало работы

После установки Sphinx и рассмотрения базового синтаксиса не стоит сразу переходить к написанию документов. Не разобравшись со схемой и выходными форматами, можно запутаться и потерять время.

В файле **index.rst** много информации, а также дополнительный сложный синтаксис. Чтобы упростить некоторые вещи, включим в проект новый файл, указав его в главном разделе.

После главного заголовка в файле **index.rst** находится оглавление. Оно содержит директиву **toctree**. Это элемент, объединяющий все документы проекта. Если проект содержит дополнительные файлы, которые не перечислены в этой директиве, они не будут включены в состав проекта при его компиляции.

Добавим в проект новый файл под именем **example.rst**. Этот файл необходимо указать в директиве **toctree**. Для правильной работы этого списка надо придерживаться определенных правил разметки, а расширение файла указывать не нужно (в нашем случае это **.rst**). В листинге ниже показано, как должен выглядеть этот список. Имя файла отделено четырьмя пробелами от левого поля, а после опции **caption** вставлена одна пустая строка. Содержимое файла **index.rst**:

```
.. toctree::
   :maxdepth: 2
   :caption: Содержание:

   example
```

Сейчас вам достаточно знать о том, что существует индексный файл, в котором перечислены все остальные файлы документов проекта, и что необходимо придерживаться правильной разметки при работе с ним.

Скопируйте содержимое первого листинга в файл **example.rst** и сохраните его. Теперь все готово к созданию выходного документа.

Выполните команду **make** и укажите в качестве конечного формата HTML. Сгенерированный вывод можно непосредственно размещать на сайте, поскольку результат будет содержать все необходимое, включая файлы JavaScript и CSS. Вывод команды **make html**:

```
> make html
Running Sphinx v1.5.3
loading translations [ru]... done
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 1 source files that are out of date
updating environment: 0 added, 1 changed, 0 removed
reading sources... [100%] example
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] index
generating indices... genindex
writing additional pages... search
copying static files... done
copying extra files... done
dumping search index in Russian (code: ru) ... done
dumping object inventory... done
build succeeded.
Build finished. The HTML pages are in build\html.
```

Ошибки синтаксиса

Если при генерации документа возникли проблемы разметки в исходном файле, **Sphinx** непременно сообщит об этом:

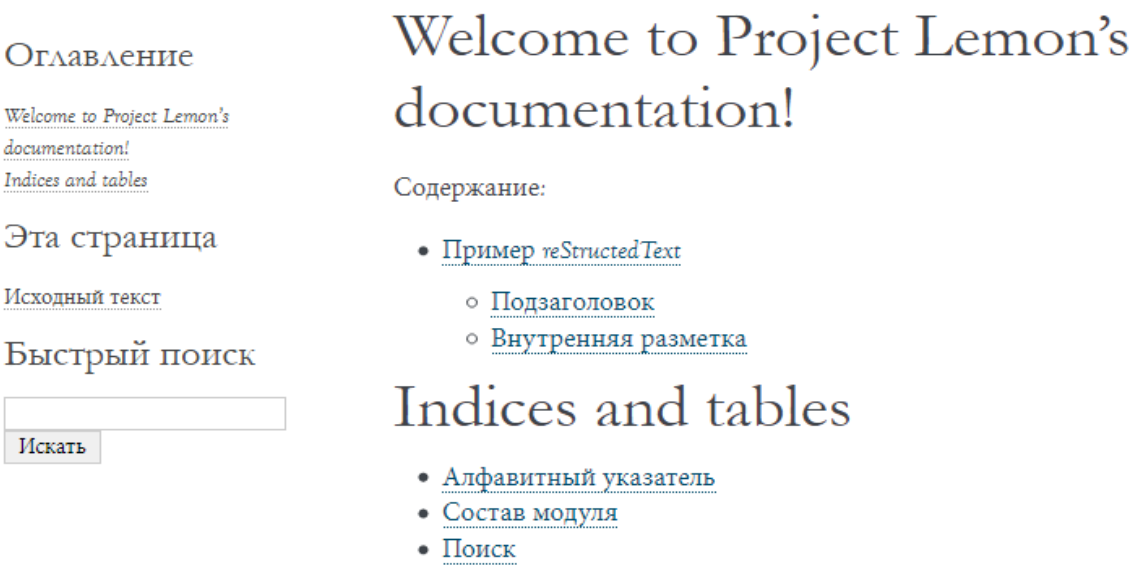
```
reading sources... [100%] index
.\project_lemon\doc\source\example_2.rst:29: WARNING: Inline emphasis
start-string without end-string.
.\project_lemon\doc\source\example_2.rst:33: WARNING: Block quote ends without a
blank line; unexpected unindent.
```

В этом случае внесите необходимые изменения в исходный файл и снова запустите команду **make** для создания готового проекта.

Результат генерации

За один прием был сгенерирован HTML-код из двух файлов — получился полнофункциональный (статический) веб-сайт.

В директории **build** должны появиться две новых директории: **doctrees** и **html**. Результирующие файлы документации находятся в директории **html**. Если открыть файл **index.html** в браузере, отобразится похожая страница:



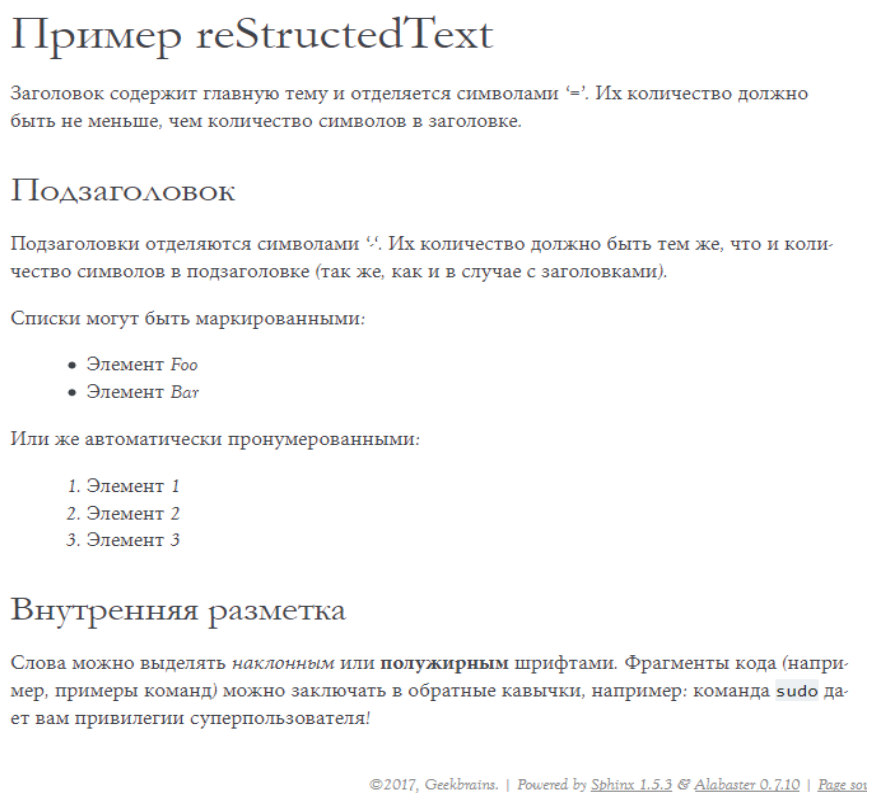
Получив совсем немного исходных данных, **Sphinx** сумел создать несложную компоновку, содержащую информацию о документации проекта, раздел поиска, содержание, заметки об авторских правах, включая имя и дату, а также нумерацию страниц.

В разделе поиска **Sphinx** проиндексировал все файлы и с помощью JavaScript создал статический сайт, на котором можно искать нужную информацию.

Поскольку во втором листинге в директиве **toctree** был указан отдельный файл с именем **example**, теперь на главной странице видно, что основной заголовок отображается в содержании в виде списка первого уровня, а подзаголовок — второго уровня. **Sphinx** позаботился о правильной структуре.

Все ссылки указывают на нужные места документа, а заголовки и подзаголовки содержат якори, на которые можно непосредственно ссылаться. Например, якорь раздела «Подзаголовок» в браузере выглядит как **../example.html#id2**. Как уже упоминалось, автору документа не нужно заботиться об этих мелочах.

На рисунке ниже показано, как выглядит файл **example.rst** в виде статического веб-сайта HTML:



Пример reStructedText

Заголовок содержит главную тему и отделяется символами '='. Их количество должно быть не меньше, чем количество символов в заголовке.

Подзаголовок

Подзаголовки отделяются символами '-'. Их количество должно быть тем же, что и количество символов в подзаголовке (так же, как и в случае с заголовками).

Списки могут быть маркированными:

- Элемент *Foo*
- Элемент *Bar*

Или же автоматически пронумерованными:

1. Элемент *1*
2. Элемент *2*
3. Элемент *3*

Внутренняя разметка

Слова можно выделять *наклонным* или **полужирным** шрифтами. Фрагменты кода (например, примеры команд) можно заключать в обратные кавычки, например: команда `sudo` дает вам привилегии суперпользователя!

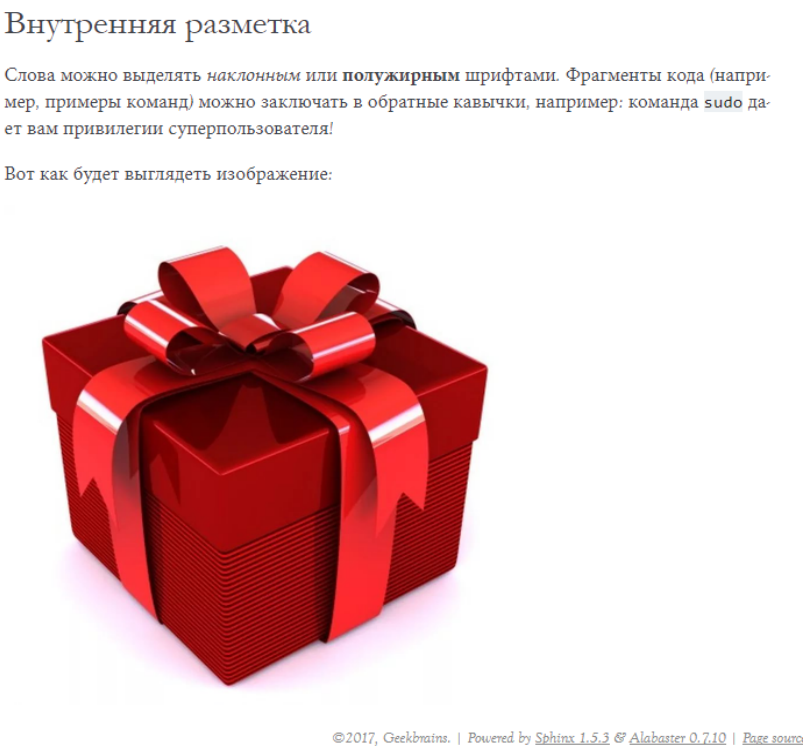
Добавление изображений

Sphinx позволяет добавлять все графические элементы в качестве статических файлов. Синтаксис легко запомнить. Изображения необходимо поместить в директорию **_static**, которая была добавлена при создании схемы проекта, после этого можно в тексте ссылаться на них. Можно ссылаться и на любые другие пути —- при генерации документа изображения будут скопированы в директорию **_static**). В листинге 4 показано, как должна выглядеть такая ссылка в разметке **reStructuredText**.

Листинг 4. Добавление статического изображения:

```
.. image:: ../img/gift.png
```

Результат добавления изображения в созданном HTML-документе:



Внутренняя разметка

Слова можно выделять *наклонным* или **полужирным** шрифтами. Фрагменты кода (например, примеры команд) можно заключать в обратные кавычки, например: команда `sudo` дает вам привилегии суперпользователя!

Вот как будет выглядеть изображение:

Строки документации в коде

Указания по составлению строк документации приведены в документах PEP 257 – Docstring Conventions (соглашения по строкам документации), а также отдельные моменты в PEP 258 – Docutils Design Specification (спецификация инструмента Docutils). Это лишь указания, но не правила и синтаксис языка.

Рассмотрим базовые аспекты этих указаний (некоторые уже знакомы вам по указаниям PEP-8).

Строка документации (docstring) — это строковый литерал, который является первым выражением в модуле, функции, классе или методе. Такая строка заносится в специальный атрибут **__doc__** объекта, в котором она расположена.

Строки документации должны быть:

- в модулях;
- в функциях и классах, экспортируемых модулем;
- в публичных методах классах (включая метод **__init__**);
- пакет может быть документирован в файле **__init__.py** в директории пакета;

Строковые литералы, расположенные в других местах Python-кода, тоже могут быть документацией. Они не обрабатываются компилятором байт-кода Python и недоступны во время выполнения (в отличие от атрибута **__doc__**). Однако два типа дополнительных строк документации могут быть извлечены утилитами:

- строковые литералы, идущие сразу после простого присваивания на верхнем уровне модуля, класса или метода **__init__**, называются строками документации атрибута;

- строковые литералы, расположенные сразу после других строк документации, называются дополнительными строками документации.

Строки документации нужно заключать в """утроенные двойные кавычки""". Если строка содержит символ обратного слэша "\", используют r"""сырые \ строки""".

Пример документирования атрибутов (см. **листинг 1**):

```
g = 'module attribute (module-global variable)'
"""Это документация переменной g."""

class AClass:
    c = 'class attribute'
    """Это документация атрибута AClass.c."""

    def __init__(self):
        """Документация метода ``__init__``"""
        self.i = 'instance attribute'
        """Документация атрибута self.i."""

def f(x):
    """Документация функции f."""
    return x**2

f.a = 1
"""Документация атрибута функции f."""
```

Пример дополнительных строк документации:

```
def function(arg):
    """Это документации функции; будет помещена в атрибут __doc__."""
    """
    Это дополнительная строка документации;
    игнорируется компилятором байт-кода
    но извлекается инструментом Docutils.
    """
    pass
```

Строки документации делятся на:

- однострочные;
- многострочные.

Однострочные Docstrings

Однострочная документация применяется для очевидных случаев и должна умещаться на одной строке:

```
def kos_root():
    """Вернуть полный путь директории KOS."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

Замечания к коду:

- Тройные кавычки применяются даже в случае однострочной документации;
- Закрывающие кавычки располагаются на той же строке, что и открывающие;
- Ни перед, ни после строки документации нет пустых строк;
- Строка документации завершается точкой и описывает функцию/метод как команду («Сделать это», «Вернуть то»), но не как описание (не следует писать в стиле «Возвращает полный путь...»);
- Однострочная документация не должна быть простой сигнатурой функции, повторяющей аргументы функции/метода (это может быть получено интроспекцией — как информация об объекте во время выполнения). Не стоит делать так:

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

Такая строка документации приемлема только для функций на языке C, для которых невозможна интроспекция. Однако тип возвращаемого значения не может быть получен интроспекцией, и его стоит описать. Приемлемой формой для такой строки будет:

```
def function(a, b):
    """Выполнить что-то и вернуть список."""
```

Многострочные Docstrings

Многострочная документация содержит обобщающую строку, как в однострочном варианте. Далее следует пустая строка, после которой идет подробное описание. Строка-обобщение может использоваться автоматическими инструментами индексирования; важно, чтобы она умещалась на одной строке и была отделена от остальной части пустой строкой. Обобщающая строка может быть на той же строке, что и открывающие кавычки, или на следующей. Все содержимое строки документации имеет такой же отступ, как и кавычки на первой строке.

Общие рекомендации

После однострочной/многострочной документации класса необходимо добавлять пустую строку, чтобы отделить документацию класса от описания методов.

Строку документации Python-скрипта (как отдельной программы) следует использовать как usage-сообщение (подсказка по использованию скрипта), когда скрипт запускается с неверными

параметрами или вообще без них. Такая строка должна описывать назначение скрипта и синтаксис командной строки, переменные окружения и файлы. Usage-сообщение может быть достаточно сложным и должно содержать информацию для нового пользователя по правильному запуску скрипта. Для опытного пользователя это сообщение может быть содержательной шпаргалкой.

В строках документации модуля должны быть перечислены классы, исключения и функции (и любые другие объекты), которые экспортируются модулем. Перечисление — в виде краткого однострочного описания каждого объекта (короче, чем обобщающая строка в документации класса/метода). Строка документации пакета должна также перечислять модули и внутренние пакеты, экспортируемые данным пакетом.

Документация для функции/метода должна обобщать ее поведение и описывать аргументы, возвращаемое значение, побочные эффекты, возбуждаемые исключения и ограничения, при которых функция может быть вызвана. Необязательные аргументы должны быть отмечены, как и именованные, которые являются частью интерфейса функции.

Документация для класса должна обобщать его поведение, перечислять публичные методы и переменные экземпляра. Конструктор класса должен быть документирован в строке документации метода **__init__**. У каждого метода должны быть свои строки документации.

Если класс-наследник имеет незначительные дополнения относительно базового класса, это должно быть отмечено в строках документации. Следует указать отличия класса-наследника от базового. Используйте слово «перегрузка» (**override**), чтобы отметить случаи, когда класс-наследник полностью переопределяет базовый метод (без обращения к методу базового класса). Словом «расширение» (**extend**) отмечайте, что класс-наследник обращается к методу базового класса и дополняет его своим функционалом.

Создание документация из Python-кода

У Sphinx есть средства для формирования документов на основании строк документации в Python-коде. Для этого необходимо, чтобы Sphinx «знал» расположение файлов с кодом — необходимо добавить директорию с кодом проекта в файл конфигурации Sphinx-проекта **conf.py** (создается в директории **source** после инициализации командой **sphinx-quickstart**):

```
sys.path.insert(0, "C:\\workspace\\PyPlotStatistics\\src")
```

Для использования относительного пути следует предварительно воспользоваться функцией **os.path.abspath**:

```
path = os.path.abspath('mydir/myfile.txt')
```

В файл **index.rst** надо добавить имена модулей и классов, которые существуют в директории с исходными кодами. Файл **index.rst** будет иметь содержимое:

```
Contents:

.. toctree::
   :maxdepth: 2

.. automodule:: SomeModuleName

.. autoclass:: SomeClassName
    :members:

Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Выполните команду **make html**, чтобы обновить документы. **Sphinx** должен автоматически идентифицировать добавленный Python-модуль и добавить ссылки на модуль **SomeModuleName** и класс **SomeClassName**. Пункт: **members:** указывает, что все атрибуты класса будут документированы — нет необходимости прописывать каждую функцию вручную.

Возможные проблемы

- ***Python-модули и файлы не появляются в документации***

Решения:

1. Sphinx-документация активно использует отступы. Убедитесь, что они расставлены так же, как в примере выше — все элементы выровнены соответствующим образом.
2. Проверьте путь в переменной **sys.path**.
3. Проверьте имя модуля/класса.

Класс отображается в документации, но не отображаются все или часть функций.

Решение: документация в функциях должна быть аналогична примеру и корректно выровнена, иначе Sphinx не распознает ее:

```
"""
Created on 29 July 2017
@author: Lisa Simpson
"""

class DatabaseManager(object):
    """ Создание и управление базой данных sqlite. """

    def connect(self):
        """ Создание подключения к базе данных """
        pass
```

- ***Добавлять вручную файлы в содержание — утомительно.***

Решение: использовать **APIdoc**.

Использование APIdoc

APIdoc — это инструмент для автоматического получения документации из программного кода проекта. Идет в комплекте со Sphinx. Он автоматически создает **.rst**-файлы для каждого модуля в проекте.

Запуск **APIdoc** осуществляется командой:

```
sphinx-apidoc [options] -o <outputdir> <sourcedir> [pathnames ...]
```

Параметры **options** и **pathnames** — не обязательны, их подробное описание можно найти в документации APIdoc. В качестве выходной директории **outputdir** удобно указать директорию с rst-файлами документации.

После запуска **APIdoc** в директории вывода будут созданы rst-файлы Python-модулей, а также файл **modules.rs**t, объединяющий ссылки на все обработанные модули. Например, если директория с исходными кодами содержит два модуля **simple_module.py** и **usefull_module.py**, автоматически созданный файл **modules.rst** будет иметь вид:

```
src
===

.. toctree::
   :maxdepth: 4

   simple_module
   usefull_module
```

Первая строка здесь содержит имя исходной директории с Python-модулями и оформлена как заголовок в разметке **reStructuredText**. Есть смысл заменить эту строку на понятное название, например:

```
API проекта Lemon
=================
```

Есть файл **modules.rst**, объединяющий документацию на программный код проекта. Поэтому эта часть присоединится к общей документации проекта простым добавлением имени **modules** в файл **index.rst**:

```
.. toctree::
   :maxdepth: 2
   :caption: Содержание:

   example
   example_2
   example_3
   modules
```

В результате в содержание будут добавлены ссылки на документацию к модулям:

Таким образом, документирование программного кода будет заключаться в написании простых строк документации для классов и функций.

Примеры документирования кода

В Приложении к уроку расположены **листинг 2** и **листинг 3** — примеры оформления строк документации в Google Style Python Docstrings и NumPy Style Python Docstrings соответственно. Для автоматической генерации документации для файла NumPy-стиля необходимо установить Sphinx-расширение numpydoc, так как этот стиль документирования имеет некоторые особенности (например, использование заголовков разделов в строках документации).

reStructedText

reStructuredText — это обычный текст, в котором применяются простые и понятные конструкции для формирования структуры документа. Их легко читать как в исходном, так и в обработанном виде. Примеры конструкций **reStructuredText** приведены в файле **02_reStructedText_basic.rst** (**листинг 4**). Парсер **reStructuredText** является частью **Docutils** (opensource-система для обработки текста, написанная на Python).

Простая косвенная разметка используется для выделения специальных конструкций — заголовков, списков, выделения текста. Разметка минимальная и ненавязчивая, насколько это возможно. Небольшая часть конструкций и расширений базовой разметки может иметь явный и более сложный формат.

Разметка **reStructuredText** может применяться к документу любого размера. Рассмотрим базовые возможности разметки **reStructedText** (подробная документация: reStructuredText Markup Specification).

Абзацы и внутренняя разметка

```
Абзацы содержат текст и могут содержать внутреннюю разметку:
*курсивное выделение*, **выделение жирным**, `интерпретируемый текст`,
``внутренние литералы``, отдельные гиперссылки (http://www.python.org),
внешние ссылки (Python_), внутренние перекрестные ссылки (example_),
сноски ([1]_), ссылки на цитаты ([CIT2002]_), подстановочные ссылки (|pi|).
```

Абзацы разделяются пустыми строками и имеют выравнивание по левому краю.

Списки

Поддерживаются пять разновидностей списков.

Маркированный список:

```
- Это первый элемент списка
- В качестве маркеров могут быть символы "*", "+", или "-"
```

Нумерованный список:

```
#. Это нумерованный список.
#. В качестве номеров могут использоваться арабские/римские числа, буквы или знак
# для автонумерации.
```

Список определений (терминов):

```
Что
    Список определений ассоциирует значение и его термин (определение)
Как
    Термин - однострочная фраза; значение - блок текста с отступом, связанным с
термином.
```

Список полей:

```
:что: Список полей ассоциирует имена полей с их содержимым, как записи в базе
данных
:как: Поле определяется последовательностью двоеточие, имя поля и снова
двоеточие.

      Содержимое поля может состоять из одного или нескольких элементов с
отступом.
```

Список опций (для перечисления опций командной строки):

```
-a            опция "a"
-b file       опции могут иметь аргументы
              и длинные описания
--long        опции тоже могут иметь длинные имена
--input=file  опции с длинными именами тоже могут
              иметь аргументы
/V            опции в стиле DOS/VMS тоже могут
              иметь место
```

Между опцией и описанием должно быть по крайней мере два пробела.

Литеральные блоки

Для создания литерального блока нужно поставить два двоеточия (`::`) в предшествующем абзаце, а сам блок оформить отступами.

```
Некоторый код::

    if literal_block:
        text = 'is left as-is'
        spaces_and_linebreaks = 'are preserved'
        markup_processing = None
```

Цитаты

Блок цитаты состоит из элементов с отступами:

```
 Ближайшая к действию цель лучше дальней.

 А.В. Суворов
```

Блок doctest

```
>>> print 'Python-specific usage examples; begun with ">>>"'
Python-specific usage examples; begun with ">>>"
>>> print '(cut and pasted from interactive Python sessions)'
(cut and pasted from interactive Python sessions)
```

Таблицы

Таблицы бывают двух типов.

Полноценные таблицы, но сложные и подробные:

```
+------------------------+------------+----------+
| Header row, column 1   | Header 2   | Header 3 |
+========================+============+==========+
| body row 1, column 1   | column 2   | column 3 |
+------------------------+------------+----------+
| body row 2             | Cells may span        |
+------------------------+-----------------------+
```

Простые таблицы — компактные, но ограниченные:

```
====================  ==========  ==========
Header row, column 1  Header 2    Header 3
====================  ==========  ==========
body row 1, column 1  column 2    column 3
body row 2            Cells may span columns
====================  ======================
```

Подробная разметка

Блоки с подробной разметкой начинаются с двух точек и пробела.

Сноски

```
.. [1] Сноска, имеющая внутренние элементы, должна иметь
   по крайней мере 3 пробела перед этими элементами
```

Цитаты

```
.. [CIT2002] Оформляются так же, как и цитаты,
   но метка может быть текстовой
```

Гиперссылки

```
.. _Python: http://www.python.org

.. _example:
Ссылка "_example" приводит к текущему абзацу.
```

Директивы

Например, изображение:

```
.. image:: img/ok2.png
```

Подстановки

```
.. |pi| image:: img/pi.png
```

Комментарии

```
.. Комментарии начинаются с двух точек и пробела. Здесь может быть что угодно,
   кроме цитат, сносок, гиперссылок, директив и подстановок.
```

Пустые строки

Пустые строки используются для разделения абзацев и других элементов. Несколько пустых строк — то же самое, что одна пустая, за исключением литеральных блоков (пустые строки сохраняются).

Отступы

Отступы используются только для выделения цитат, определений (в их списках) и вложенного содержимого (многострочные элементы списков, вложенные списки; литеральные блоки; маркированные блоки).

Резюме

Документация — важная часть проекта. Генераторы документации помогают сделать процесс документирования максимально продуктивным.

Возможности генератора документации Sphinx обширны:

- экспорт документации в различные форматы (PDF, epub, man (man-страницы UNIX) и LaTeX). Необходимо устанавливать дополнительные библиотеки и программное обеспечение;
- для работы со сложными диаграммами имеется подключаемый модуль, который позволяет добавлять в проекты диаграммы **Graphviz**;
- существует множество других подключаемых модулей для Sphinx (расширений). Некоторые из них (например, **interSphinx**, позволяющий связывать несколько отельных проектов) входят в комплект самого Sphinx;
- стиль документа можно менять, используя одну из многочисленных тем Sphinx.

Всю работу по составлению документации можно выполнить одним инструментом — **текстовым редактором**. Текстовый формат позволяет использовать системы контроля версий не только для программного кода, но и для документации проекта.

Примечание. Методические указания к курсу «Python. Уровень 2.2» составлялись в формате **reStructedText** для последующей генерации документов системой Sphinx. Исходные коды методических указаний располагаются в директории **examples/course**.

Практическое задание

1. Для проекта «Мессенджер» подготовить документацию с использованием **sphinx-doc**.
2. Проверить программный код практических заданий текущего курса и курса Python-1 на соответствие положениям **PEP-8**. При необходимости выполнить преобразования.

Дополнительные материалы

1. Awesome Sphinx (Python Documentation Generator).
2. Официальная документация Sphinx.
3. Documenting Your Project With Sphinx.
4. Example Google Style Python Docstrings.
5. Example NumPy Style Python Docstrings.
6. Sphinx Autodoc Tutorial for Dummies.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. David Beazley, Brian K. Jones. Python Cookbook. Third Edition (каталог Дополнительные материалы).
2. Michael Driscoll. Python 101 (каталог Дополнительные материалы).

3. Michał Jaworski, Tarek Ziadé. Expert Python Programming, Second Edition (каталог Дополнительные материалы).
4. Бизли Дэвид. Python. Подробный справочник. 4-е издание (каталог Дополнительные материалы).
5. Стильный код на Python, или учимся использовать Flake8.
6. Форматирование Python-кода.
7. Онлайн-конвертер в PEP-8.
8. Простое и удобное создание документации в Sphinx

Приложение

Листинг 1

```
# ====================== Распространение приложений
============================
# ------------------------- Документирование кода
-----------------------------


# ---------------------- Документирование атрибутов
---------------------------

g = 'module attribute (module-global variable)'
"""Это документация переменной g."""

module_level_variable2 = 98765
"""int: Переменная модуля.

Можно дополнительно указывать тип переменной,
отделяя его двоеточием от описания
"""

class AClass:

    c = 'class attribute'
    """Это документация атрибута AClass.c."""

    def __init__(self):
        """Документация метода ``__init__``"""

        self.i = 'instance attribute'
        """Документация атрибута self.i."""


def f(x):
    """Документация функции f."""
    return x**2
```

```
f.a = 1
"""Документация атрибута фукнции f."""

# ---------------- Дополнительные строки документации
-------------------------

def function(arg):
    """Это документации функции; будет помещена в атрибут __doc__."""
    """
    Это дополнительная строка документации;
    игнорируется компилятором байт-кода
    но извлекается инструментом Docutils.
    """
    pass


def function_with_types_in_docstring(param1, param2):
    """ Пример функции с указанием типов в строках документации.

    Поддерживается аннотация типов согласно `PEP 484`_ .
    Если атрибут, параметр и возвращаемое значение имеют аннотацию по `PEP
484`_,
    то из не нужно включать в строки документации.

    Args:
        param1 (int): Первый параметр.
        param2 (str): Второй параметр.

    Returns:
        bool: Возвращаемое значение. True - успешное завершение, False - ошибка.

    .. _`PEP 484`: https://www.python.org/dev/peps/pep-0484/

    """
    pass


def function_with_pep484_type_annotations(param1: int, param2: str) -> bool:
    """ Пример функции с аннотацией типов по PEP 484.

    Args:
        param1: Первый параметр.
        param2: Второй параметр.

    Returns:
        Возвращаемое значение. True - успешное завершение, False - ошибка.

    """
    pass
```

Листинг 2

```
# -*- coding: utf-8 -*-
"""Example Google style docstrings.

This module demonstrates documentation as specified by the `Google Python
Style Guide`_. Docstrings may extend over multiple lines. Sections are created
with a section header and a colon followed by a block of indented text.

Example:
    Examples can be given using either the ``Example`` or ``Examples``
    sections. Sections support any reStructuredText formatting, including
    literal blocks::

        $ python example_google.py

Section breaks are created by resuming unindented text. Section breaks
are also implicitly created anytime a new section starts.

Attributes:
    module_level_variable1 (int): Module level variables may be documented in
        either the ``Attributes`` section of the module docstring, or in an
        inline docstring immediately following the variable.

        Either form is acceptable, but the two should not be mixed. Choose
        one convention to document module level variables and be consistent
        with it.

Todo:
    * For module TODOs
    * You have to also use ``sphinx.ext.todo`` extension

.. _Google Python Style Guide:
   http://google.github.io/styleguide/pyguide.html

"""

module_level_variable1 = 12345

module_level_variable2 = 98765
"""int: Module level variable documented inline.

The docstring may span multiple lines. The type may optionally be specified
on the first line, separated by a colon.
"""


def function_with_types_in_docstring(param1, param2):
    """Example function with types documented in the docstring.

    `PEP 484`_ type annotations are supported. If attribute, parameter, and
    return types are annotated according to `PEP 484`_, they do not need to be
    included in the docstring:
```

```
    Args:
        param1 (int): The first parameter.
        param2 (str): The second parameter.

    Returns:
        bool: The return value. True for success, False otherwise.

    .. _PEP 484:
        https://www.python.org/dev/peps/pep-0484/

    """


def function_with_pep484_type_annotations(param1: int, param2: str) -> bool:
    """Example function with PEP 484 type annotations.

    Args:
        param1: The first parameter.
        param2: The second parameter.

    Returns:
        The return value. True for success, False otherwise.

    """


def module_level_function(param1, param2=None, *args, **kwargs):
    """This is an example of a module level function.

    Function parameters should be documented in the ``Args`` section. The name
    of each parameter is required. The type and description of each parameter
    is optional, but should be included if not obvious.

    If ``*args`` or ``**kwargs`` are accepted,
    they should be listed as ``*args`` and ``**kwargs``.

    The format for a parameter is::

        name (type): description
            The description may span multiple lines. Following
            lines should be indented. The "(type)" is optional.

            Multiple paragraphs are supported in parameter
            descriptions.

    Args:

        :param1 (int): The first parameter.
        :param2 (`str`, optional): The second parameter. Defaults to None.

                                   Second line of description should be
```

```
indented.

        :`*args`: Variable length argument list.
        :`**kwargs`: Arbitrary keyword arguments.

    Returns:

        bool: True if successful, False otherwise.

        The return type is optional and may be specified at the beginning of
        the ``Returns`` section followed by a colon.

        The ``Returns`` section may span multiple lines and paragraphs.
        Following lines should be indented to match the first line.

        The ``Returns`` section supports any reStructuredText formatting,
        including literal blocks::

            {
                'param1': param1,
                'param2': param2
            }

    Raises:

        AttributeError: The ``Raises`` section is a list of all exceptions
                        that are relevant to the interface.

        ValueError: If `param2` is equal to `param1`.

    """
    if param1 == param2:
        raise ValueError('param1 may not be equal to param2')
    return True


def example_generator(n):
    """Generators have a ``Yields`` section instead of a ``Returns`` section.

    Args:
        n (int): The upper limit of the range to generate, from 0 to `n` - 1.

    Yields:
        int: The next number in the range of 0 to `n` - 1.

    Examples:
        Examples should be written in doctest format, and should illustrate how
        to use the function.

        >>> print([i for i in example_generator(4)])
        [0, 1, 2, 3]
```

```
    """
    for i in range(n):
        yield i


class ExampleError(Exception):
    """Exceptions are documented in the same way as classes.

    The __init__ method may be documented in either the class level
    docstring, or as a docstring on the __init__ method itself.

    Either form is acceptable, but the two should not be mixed. Choose one
    convention to document the __init__ method and be consistent with it.

    Note:
        Do not include the `self` parameter in the ``Args`` section.

    Args:
        msg (str): Human readable string describing the exception.
        code (:obj:`int`, optional): Error code.

    Attributes:
        msg (str): Human readable string describing the exception.
        code (int): Exception error code.

    """

    def __init__(self, msg, code):
        self.msg = msg
        self.code = code


class ExampleClass(object):
    """The summary line for a class docstring should fit on one line.

    If the class has public attributes, they may be documented here
    in an ``Attributes`` section and follow the same formatting as a
    function's ``Args`` section. Alternatively, attributes may be documented
    inline with the attribute's declaration (see __init__ method below).

    Properties created with the ``@property`` decorator should be documented
    in the property's getter method.

    Attributes:
        attr1 (str): Description of `attr1`.
        attr2 (:obj:`int`, optional): Description of `attr2`.

    """

    def __init__(self, param1, param2, param3):
        """Example of docstring on the __init__ method.
```

```
        The __init__ method may be documented in either the class level
        docstring, or as a docstring on the __init__ method itself.

        Either form is acceptable, but the two should not be mixed. Choose one
        convention to document the __init__ method and be consistent with it.

        Note:
            Do not include the `self` parameter in the ``Args`` section.

        Args:
            param1 (str): Description of `param1`.
            param2 (:obj:`int`, optional): Description of `param2`. Multiple
                lines are supported.
            param3 (list(str)): Description of `param3`.

        """
        self.attr1 = param1
        self.attr2 = param2
        self.attr3 = param3  #: Doc comment *inline* with attribute

        #: list(str): Doc comment *before* attribute, with type specified
        self.attr4 = ['attr4']

        self.attr5 = None
        """str: Docstring *after* attribute, with type specified."""

    @property
    def readonly_property(self):
        """str: Properties should be documented in their getter method."""
        return 'readonly_property'

    @property
    def readwrite_property(self):
        """list(str): Properties with both a getter and setter
        should only be documented in their getter method.

        If the setter method contains notable behavior, it should be
        mentioned here.
        """
        return ['readwrite_property']

    @readwrite_property.setter
    def readwrite_property(self, value):
        value

    def example_method(self, param1, param2):
        """Class methods are similar to regular functions.

        Note:
            Do not include the `self` parameter in the ``Args`` section.

        Args:
```

```
            param1: The first parameter.
            param2: The second parameter.

        Returns:
            True if successful, False otherwise.

        """
        return True

    def __special__(self):
        """By default special members with docstrings are not included.

        Special members are any methods or attributes that start with and
        end with a double underscore. Any special member with a docstring
        will be included in the output, if
        ``napoleon_include_special_with_doc`` is set to True.

        This behavior can be enabled by changing the following setting in
        Sphinx's conf.py::

            napoleon_include_special_with_doc = True

        """
        pass

    def __special_without_docstring__(self):
        pass

    def _private(self):
        """By default private members are not included.

        Private members are any methods or attributes that start with an
        underscore and are *not* special. By default they are not included
        in the output.

        This behavior can be changed such that private members *are* included
        by changing the following setting in Sphinx's conf.py::

            napoleon_include_private_with_doc = True

        """
        pass

    def _private_without_docstring(self):
        pass
```

Листинг 3

```
# -*- coding: utf-8 -*-
"""Example NumPy style docstrings.
```

```
This module demonstrates documentation as specified by the `NumPy
Documentation HOWTO`_. Docstrings may extend over multiple lines. Sections
are created with a section header followed by an underline of equal length.

Example
-------
Examples can be given using either the ``Example`` or ``Examples``
sections. Sections support any reStructuredText formatting, including
literal blocks::

    $ python example_numpy.py


Section breaks are created with two blank lines. Section breaks are also
implicitly created anytime a new section starts. Section bodies *may* be
indented:

Notes
-----
    This is an example of an indented section. It's like any other section,
    but the body is indented to help it stand out from surrounding text.

If a section is indented, then a section break is created by
resuming unindented text.

Attributes
----------
module_level_variable1 : int
    Module level variables may be documented in either the ``Attributes``
    section of the module docstring, or in an inline docstring immediately
    following the variable.

    Either form is acceptable, but the two should not be mixed. Choose
    one convention to document module level variables and be consistent
    with it.


.. _NumPy Documentation HOWTO:
   https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt

"""

module_level_variable1 = 12345

module_level_variable2 = 98765
"""int: Module level variable documented inline.

The docstring may span multiple lines. The type may optionally be specified
on the first line, separated by a colon.
"""
```

```
def function_with_types_in_docstring(param1, param2):
    """Example function with types documented in the docstring.

    `PEP 484`_ type annotations are supported. If attribute, parameter, and
    return types are annotated according to `PEP 484`_, they do not need to be
    included in the docstring:

    Parameters
    ----------
    param1 : int
        The first parameter.
    param2 : str
        The second parameter.

    Returns
    -------
    bool
        True if successful, False otherwise.

    .. _PEP 484:
        https://www.python.org/dev/peps/pep-0484/

    """


def function_with_pep484_type_annotations(param1: int, param2: str) -> bool:
    """Example function with PEP 484 type annotations.

    The return type must be duplicated in the docstring to comply
    with the NumPy docstring style.

    Parameters
    ----------
    param1
        The first parameter.
    param2
        The second parameter.

    Returns
    -------
    bool
        True if successful, False otherwise.

    """


def module_level_function(param1, param2=None, *args, **kwargs):
    """This is an example of a module level function.

    Function parameters should be documented in the ``Parameters`` section.
    The name of each parameter is required. The type and description of each
    parameter is optional, but should be included if not obvious.
```

```
If ``*args`` or ``**kwargs`` are accepted,
they should be listed as ``*args`` and ``**kwargs``.

The format for a parameter is::

    name : type
        description

        The description may span multiple lines. Following lines
        should be indented to match the first line of the description.
        The ": type" is optional.

        Multiple paragraphs are supported in parameter
        descriptions.

Parameters
----------
param1 : int
    The first parameter.
param2 : :obj:`str`, optional
    The second parameter.
`*args`
    Variable length argument list.
`**kwargs`
    Arbitrary keyword arguments.

Returns
-------
bool
    True if successful, False otherwise.

    The return type is not optional. The ``Returns`` section may span
    multiple lines and paragraphs. Following lines should be indented to
    match the first line of the description.

    The ``Returns`` section supports any reStructuredText formatting,
    including literal blocks::

        {
            'param1': param1,
            'param2': param2
        }

Raises
------
AttributeError
    The ``Raises`` section is a list of all exceptions
    that are relevant to the interface.
ValueError
    If `param2` is equal to `param1`.
```

```
    """
    if param1 == param2:
        raise ValueError('param1 may not be equal to param2')
    return True


def example_generator(n):
    """Generators have a ``Yields`` section instead of a ``Returns`` section.

    Parameters
    ----------
    n : int
        The upper limit of the range to generate, from 0 to `n` - 1.

    Yields
    ------
    int
        The next number in the range of 0 to `n` - 1.

    Examples
    --------
    Examples should be written in doctest format, and should illustrate how
    to use the function.

    >>> print([i for i in example_generator(4)])
    [0, 1, 2, 3]

    """
    for i in range(n):
        yield i


class ExampleError(Exception):
    """Exceptions are documented in the same way as classes.

    The __init__ method may be documented in either the class level
    docstring, or as a docstring on the __init__ method itself.

    Either form is acceptable, but the two should not be mixed. Choose one
    convention to document the __init__ method and be consistent with it.

    Note
    ----
    Do not include the `self` parameter in the ``Parameters`` section.

    Parameters
    ----------
    msg : str
        Human readable string describing the exception.
    code : :obj:`int`, optional
        Numeric error code.
```

```
    Attributes
    ----------
    msg : str
        Human readable string describing the exception.
    code : int
        Numeric error code.

    """

    def __init__(self, msg, code):
        self.msg = msg
        self.code = code


class ExampleClass(object):
    """The summary line for a class docstring should fit on one line.

    If the class has public attributes, they may be documented here
    in an ``Attributes`` section and follow the same formatting as a
    function's ``Args`` section. Alternatively, attributes may be documented
    inline with the attribute's declaration (see __init__ method below).

    Properties created with the ``@property`` decorator should be documented
    in the property's getter method.

    Attributes
    ----------
    attr1 : str
        Description of `attr1`.
    attr2 : :obj:`int`, optional
        Description of `attr2`.

    """

    def __init__(self, param1, param2, param3):
        """Example of docstring on the __init__ method.

        The __init__ method may be documented in either the class level
        docstring, or as a docstring on the __init__ method itself.

        Either form is acceptable, but the two should not be mixed. Choose one
        convention to document the __init__ method and be consistent with it.

        Note
        ----
        Do not include the `self` parameter in the ``Parameters`` section.

        Parameters
        ----------
        param1 : str
            Description of `param1`.
        param2 : list(str)
```

```
            Description of `param2`. Multiple
            lines are supported.
        param3 : :obj:`int`, optional
            Description of `param3`.

        """
        self.attr1 = param1
        self.attr2 = param2
        self.attr3 = param3  #: Doc comment *inline* with attribute

        #: list(str): Doc comment *before* attribute, with type specified
        self.attr4 = ["attr4"]

        self.attr5 = None
        """str: Docstring *after* attribute, with type specified."""

    @property
    def readonly_property(self):
        """str: Properties should be documented in their getter method."""
        return "readonly_property"

    @property
    def readwrite_property(self):
        """list(str): Properties with both a getter and setter
        should only be documented in their getter method.

        If the setter method contains notable behavior, it should be
        mentioned here.
        """
        return ["readwrite_property"]

    @readwrite_property.setter
    def readwrite_property(self, value):
        value

    def example_method(self, param1, param2):
        """Class methods are similar to regular functions.

        Note
        ----
        Do not include the `self` parameter in the ``Parameters`` section.

        Parameters
        ----------
        param1
            The first parameter.
        param2
            The second parameter.

        Returns
        -------
        bool
```

```
            True if successful, False otherwise.

        """
        return True

    def __special__(self):
        """By default special members with docstrings are not included.

        Special members are any methods or attributes that start with and
        end with a double underscore. Any special member with a docstring
        will be included in the output, if
        ``napoleon_include_special_with_doc`` is set to True.

        This behavior can be enabled by changing the following setting in
        Sphinx's conf.py::

            napoleon_include_special_with_doc = True

        """
        pass

    def __special_without_docstring__(self):
        pass

    def _private(self):
        """By default private members are not included.

        Private members are any methods or attributes that start with an
        underscore and are *not* special. By default they are not included
        in the output.

        This behavior can be changed such that private members *are* included
        by changing the following setting in Sphinx's conf.py::

            napoleon_include_private_with_doc = True

        """
        pass

    def _private_without_docstring(self):
        pass
```

Листинг 4

```
Это заголовок
=============
Заголовок содержит главную тему и отделяется символами '='.
Их количество должно быть не меньше, чем количество символов
в заголовке.

Подзаголовок
```

```
------------
Подзаголовки отделяются символами '-'. Их количество должно
быть тем же, что и количество символов в подзаголовке
(так же, как и в случае с заголовками).

Списки могут быть маркированными:

 * Элемент Foo
 * Элемент Bar

Или же автоматически пронумерованными:

 #. Элемент 1
 #. Элемент 2

Внутренняя разметка
-------------------
Слова можно выделять *наклонным* или **полужирным** шрифтами.
Фрагменты кода (например, примеры команд) можно заключать в обратные кавычки,
например:
команда ``sudo`` дает вам привилегии суперпользователя!
```