Hyperiondev

TASK

Software Documentation

Visit our website

Introduction

Reliable documentation is crucial for software engineers. It enhances code clarity, maintainability, and knowledge transfer, ultimately improving software quality. This lesson explores both external and internal documentation.

DOCUMENTATION IS KEY

> *Code as if whoever maintains your program is a violent psychopath who knows where you live.*
> *—Anonymous*

We have all, at some point or another, tried to put together a toy, an appliance, or even cook a meal without following the instructions. Unless you have a natural flair for mechanics or cooking, you tend to run into problems. In programming, documentation is the manual, or recipe, you write so that the next person looking at your code is able to follow it and understand it. This is done on two different levels: broadly, how to use the software, and more specifically, how the software itself works.

The first level is known as **external documentation**. This can include a separate document that explains to the user how to use the software, known as **user documentation**. It could also use folders, such as Unit Development Folders (UDFs), which document the developer's notes during creation, such as particular design decisions that have been taken and tools that have been used. This is known as **library documentation**. In the UDF, there could be a Detailed-Design Document (DDD): a low-level document that shows design decisions on a class level, as well as what possibilities were considered and why the final decision was taken. Ironically, not all external documentation is outside the code. A DDD could actually be in the code itself.

In comparison, **internal documentation**, the most detailed form of documentation, is always within the code. It is the documentation that you have been encouraged to include in your code thus far: comments and programming style. Because you are already familiar with these, we will first look at internal documentation, and then discuss how we go about generating external documentation.

INTERNAL DOCUMENTATION

Internal documentation ensures code clarity, maintainability, and efficient collaboration among developers, leading to better software quality and faster development cycles. You have already been implementing best practices by following PEP8 guidelines. Let's recap good programming style, comments and docstrings.

Good programming style

Have a look at the code below. Can you figure out what's going on?

```
        a=45; b=None
if a>=80:
    b='A'
elif          (a>=70 and a<80):
         b='B'
elif (a>=60 and a<70):
            b='C'
elif (a>=50 and a<60):
                b='D'
else:
                      b='F'
print(b)
```

I'm sure you can see it has something to do with determining if something is A, B, C, D or F, so you may correctly deduce it has something to do with grades, but have a look at how much easier it is to understand in the code below:

```
# Determines a student's academic symbol based on their grade (%)
grade = 85

if grade >= 80:
    symbol = 'A'

elif grade >= 70 and grade < 80:
    symbol = 'B'

elif grade >= 60 and grade < 70:
    symbol = 'C'

elif grade >= 50 and grade < 60:
    symbol = 'D'

else:
    symbol = 'F'

print(symbol)
```

Now that we have refactored the code using descriptive variable names, the correct spacing and indentation, and a summary comment at the top, understanding the code takes very little effort. While this is a simple example, it illustrates how vital good programming style is as a form of internal documentation, especially when the code becomes more complicated. Let us now recap the important roles of that comments play :

1. Explaining the code
2. Summarising the code
3. Describing what the code is meant to be doing
4. Separating logical sections of code
5. Commenting out code for debugging
6. Commenting out code while trying different approaches
   Pproviding additional information such as copyright, confidentiality, etc.

Python provides three kinds of comments including block comments, inline comments, and documentation strings.

```
# This is a block comment,
# it is used to describe a logical block of code
greeting = "Hello, World!"
print(greeting)
```

```
greeting = "Hello, World!"
print(greeting)  # This is an inline comment it describes a line of code
```

A documentation string (or **docstring**) is a string literal placed at the beginning of a code block, such as a function or class definition. It provides a concise description of the code's purpose, usage, and parameters. Docstrings are used to automatically generate documentation for your code, making it easier to understand and maintain. Unlike comments, docstrings are not ignored by the Python interpreter but are stored as metadata associated with the code object.

Now let's take a closer look at external documentation in Python.

EXTERNAL DOCUMENTATION

As you can imagine, writing out an entirely separate document for your code could be a gruelling task. Fortunately, though, there are some tools that can help. In Python, we can use **Sphinx**, a documentation generator. It can be used to create documentation in HTML format. What is particularly helpful about Sphinx is that you write the documentation in the code using docstrings.

When the first statement in a function (or class or module) is a string literal, it is referred to as a docstring and is stored in your project as such.

Generally, a Sphinx docstring has the following format:

```
"""[Summary]

:param [ParamName]: [ParamDescription], defaults to [DefaultParamVal]
:type [ParamName]: [ParamType](, optional)
...
:raises [ErrorType]: [ErrorDescription]
...
:return: [ReturnDescription]
:rtype: [ReturnType]
"""
```

A pair of `:param:` and :`type:` directive options must be used for each parameter we wish to document. The `:raises:` option is used to describe any errors that are raised by the code, while the `:return:` and `:rtype:` options are used to describe any values returned by our code. A more thorough explanation of the Sphinx docstring format can be found in this **docstring writing tutorial** from the Sphinx documentation.

Note that the '`...`' notation has been used above to indicate repetition and should not be used when generating actual docstrings, as can be seen by the example presented below.

Look at the following example taken from the Sphinx documentation to see what typical documentation using docstrings may look like for a class and the methods of that class.

```
class SimpleBleDevice(object):
    """This is a conceptual class representation of a simple BLE device
    (GATT Server). It is essentially an extended combination of the
    :class:`bluepy.btle.Peripheral` and :class:`bluepy.btle.ScanEntry`classes

    :param client: A handle to the :class:`simpleble.SimpleBleClient` client
        object that detected the device
    :type client: class:`simpleble.SimpleBleClient`
    :param addr: Device MAC address, defaults to None
    :type addr: str, optional
    :param addrType: Device address type - one of ADDR_TYPE_PUBLIC or
        ADDR_TYPE_RANDOM, defaults to ADDR_TYPE_PUBLIC
    :type addrType: str, optional
    :param iface: Bluetooth interface number (0 = /dev/hci0) used for the
        connection, defaults to 0
    :type iface: int, optional
    :param data: A list of tuples (adtype, description, value) containing
        the AD type code, human-readable description and value for all available
        advertising data items, defaults to None
    :type data: list, optional
    :param rssi: Received Signal Strength Indication for the last received
        broadcast from the device. This is an integer value measured in dB,
        where 0 dB is the maximum (theoretical) signal strength, and more
        negative numbers indicate a weaker signal, defaults to 0
    :type rssi: int, optional
    :param connectable: `True` if the device supports connections, and `False`
        otherwise (typically used for advertising 'beacons').,
        defaults to `False`
    :type connectable: bool, optional
    :param updateCount: Integer count of the number of advertising packets
        received from the device so far, defaults to 0
    :type updateCount: int, optional
    """

    def __init__(self, client, addr=None, addrType=None, iface=0,
                 data=None, rssi=0, connectable=False, updateCount=0):
        """Constructor method
        """
        super().__init__(deviceAddr=None, addrType=addrType, iface=iface)
        self.addr = addr
        self.addrType = addrType
```

```
        self.iface = iface
        self.rssi = rssi
        self.connectable = connectable
        self.updateCount = updateCount
        self.data = data
        self._connected = False
        self._services = []
        self._characteristics = []
        self._client = client

    def getServices(self, uuids=None):
        """Returns a list of :class:`bluepy.blte.Service` objects representing
        the services offered by the device. This will perform Bluetooth service
        discovery if this has not already been done; otherwise it will return a
        cached list of services immediately..

        :param uuids: A list of string service UUIDs to be discovered,
            defaults to None
        :type uuids: list, optional
        :return: A list of the discovered :class:`bluepy.blte.Service` objects,
            which match the provided ``uuids``
        :rtype: list On Python 3.x, this returns a dictionary view object,
            not a list
        """
        self._services = []
        if(uuids is not None):
            for uuid in uuids:
                try:
                    service = self.getServiceByUUID(uuid)
                    self.services.append(service)
                except BTLEException:
                    pass
        else:
            self._services = super().getServices()
        return self._services

    def setNotificationCallback(self, callback):
        """Set the callback function to be executed when the device sends a
        notification to the client.

        :param callback: A function handle of the form
            ``callback(client, characteristic, data)``, where ``client`` is a
            handle to the :class:`simpleble.SimpleBleClient` that invoked the
            callback, ``characteristic`` is the notified
            :class:`bluepy.blte.Characteristic` object and data is a
            `bytearray` containing the updated value. Defaults to None
```

```
        :type callback: function, optional
        """
        self.withDelegate(
            SimpleBleNotificationDelegate(
                callback,
                client=self._client
            )
        )

    def getCharacteristics(self, startHnd=1, endHnd=0xFFFF, uuids=None):
        """Returns a list containing :class:`bluepy.btle.Characteristic`
        objects for the peripheral. If no arguments are given, will return all
        characteristics. If startHnd and/or endHnd are given, the list is
        restricted to characteristics whose handles are within the given range.

        :param startHnd: Start index, defaults to 1
        :type startHnd: int, optional
        :param endHnd: End index, defaults to 0xFFFF
        :type endHnd: int, optional
        :param uuids: a list of UUID strings, defaults to None
        :type uuids: list, optional
        :return: List of returned :class:`bluepy.btle.Characteristic` objects
        :rtype: list
        """
        self._characteristics = []
        if(uuids is not None):
            for uuid in uuids:
                try:
                    characteristic = super().getCharacteristics(
                        startHnd, endHnd, uuid)[0]
                    self._characteristics.append(characteristic)
                except BTLEException:
                    pass
        else:
            self._characteristics = super().getCharacteristics(startHnd,
                                                               endHnd)
        return self._characteristics

    def connect(self):
        """Attempts to initiate a connection with the device.

        :return: `True` if connection was successful, `False` otherwise
        :rtype: bool
        """
        try:
            super().connect(self.addr,
```

```
                            addrType=self.addrType,
                            iface=self.iface)
        except BTLEException as ex:
            self._connected = False
            return (False, ex)
        self._connected = True
        return True

    def disconnect(self):
        """Drops existing connection to device
        """
        super().disconnect()
        self._connected = False

    def isConnected(self):
        """Checks to see if device is connected

        :return: `True` if connected, `False` otherwise
        :rtype: bool
        """
        return self._connected

    def printInfo(self):
        """Print info about device
        """
        print("Device %s (%s), RSSI=%d dB" %
              (self.addr, self.addrType, self.rssi))
        for (adtype, desc, value) in self.data:
            print("  %s = %s" % (desc, value))
```

To add comments for Sphinx to include in the documentation, we enclose the text within triple quotes (""").

```
"""Some documentation here
"""
```

Sphinx uses what are called “directives” to document things like return types, information about parameters and their types, etc. They are used like tags with a colon prefixing the directive almost like tags and also suffixing the directive.

Some of the most used directives are: `:param:`, `:type:`, `:return:` and `:rtype:` (the return

Let's look at the example below:

```
def add_nums(num1, num2):
    """This method will be used to add two numbers

        :param int num1: The first number
        :param int num2: The second number

        :returns: The sum of two numbers

        :rtype: int
    """
    answer = num1 + num2
    return answer
```

The first thing you see in the docstring is a description of what the function is used for. After this, we see the two `:param:` directives used to describe the parameters that the function takes. Note that a colon is used to close the directive as well before the information about that directive is given. The param directive contains within it the type and the name of the param before closing it i.e. the type is `int` and the names are `num1` and `num2`, respectively.

Following this, we also have a `:return:` directive explaining what is being returned and the return type (`:rtype:`) directive explaining what datatype the function returns.

Sphinx actually uses a markup language called RTP (reStructuredText) for document production. That means that there is flexibility in how we can structure and format the documentation within the docstring. We suggest looking at the **Sphinx documentation** to see how we can creatively use this simple plaintext markup language.

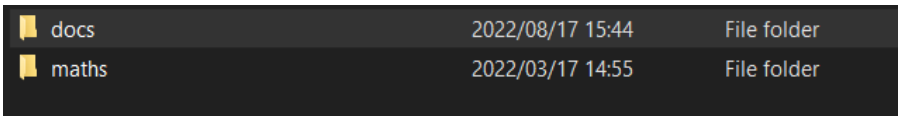
Instructions

Before you start with the practical tasks, please work through the **Setting Up a Virtual Environment** additional reading to ensure you are familiar with creating and activating virtual environments. You will need to install Sphinx within a virtual environment to manage dependencies effectively. If you try to generate documentation using Sphinx outside of a virtual environment the commands may not work as expected.

Practical task 1

The following instructions will guide you through the steps needed to generate documentation using Sphinx for the code in the sphinx_maths folder.

1. First, download the **sphinx_maths** folder which can be found in the folder for this task. Ensure you download the folder onto your local drive outside any folders associated with a cloud storage provider such as OneDrive or Dropbox.
2. Within the downloaded **sphinx_maths** folder, create a new folder named **docs**. This **docs** folder will hold all the generated documentation files from Sphinx.
    a. After creating the **docs** folder, you should now have both a **maths** and a **docs** folder within the **sphinx_maths** folder, similar to the image below:



3. The next step is to create and activate a virtual environment. This isolates your project's dependencies and ensures that installing Sphinx won't affect other projects or system-wide packages. If you're unsure about how to activate the virtual environment, please refer to the additional reading for guidance.
4. Before proceeding with installing the packages required for this task, make sure the virtual environment is still activated. Once confirmed, you can continue with the installation.
    a. The first package we will install is the Sphinx package, which can be installed using `pip` command below:

    ```
    pip install -U sphinx
    ```

    For additional information regarding installing the Sphinx package, have a look at the **Sphinx Documentation**.

    b. Next, you may install a theme designed to enhance the reader's experience. Using a theme is optional for this task. If you choose to install a theme, we recommend using **sphinx-rtd-theme**. Install the theme using the following command :

    ```
    pip install sphinx-rtd-theme
    ```

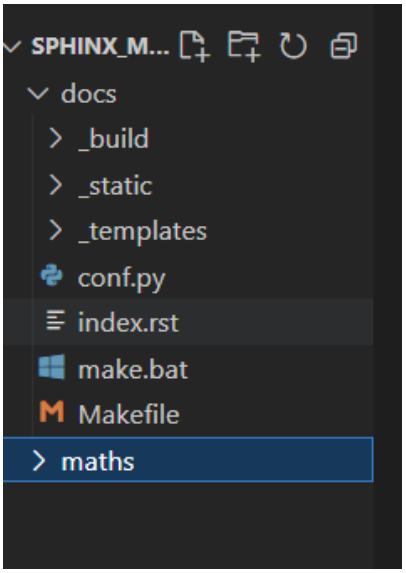
    You can learn more about the **sphinx-rtd-theme** on the PyPI website.

    After installing the packages, remember that you need to activate the virtual environment each time you want to use these Sphinx-related packages.

5. We can now proceed to run a command that sets up the necessary files for Sphinx to generate documentation. Before running the command, make sure to change the directory in the command line to the **docs** folder that we created. Once you are in the **docs** folder, execute the following command:

    ```
    sphinx-quickstart
    ```

6. You will be prompted with questions to configure the settings:
    a. When prompted if you want to separate **source** and **build** directories, press the enter key to accept the default, which is **no**.
    b. Following this, when asked to provide the project name, call the project **Maths**.
    c. When prompted to provide the author's name, enter your name.
    d. For the project release, use the following versioning format: **00.00.01**.
    e. When asked to enter the language, press Enter to accept the default option, which is English.
7. If you open the **docs** folder, you should see a file and folder structure similar to the example below:



8. Before continuing with the next steps, open the sphinx_maths folder within your code editor. We will need to configure a few files to ensure that the documentation is generated correctly.
    a. First, we need to configure Sphinx, by editing the **conf.py** file, which is the build configuration file. You can find this file in the **docs** folder. In the **conf.py** file, locate the section with the empty **extensions** list and modify it as follows:

```
extensions = [
    "sphinx.ext.autodoc",
    "sphinx.ext.viewcode",
    "sphinx.ext.napoleon"
]
```

    b. At the top of the **conf.py** file, add the following code to ensure that Sphinx reads from the root folder of the project:

```
import os
import sys
sys.path.insert(0, os.path.abspath(".."))
```

    c. Lastly, if you have installed the optional **sphinx-rtd-theme**, locate the **html_theme** variable in the **conf.py** file and update it as follows:

```
html_theme = 'sphinx_rtd_theme'
```

9. Next, we need to generate **.rst** files (reStructuredText markup files) for the Python files in the sphinx_maths folder. These **.rst** files will help Sphinx process and convert the docstrings in our Python files into HTML.

   To do this, first change the directory to the parent folder, **sphinx_maths**. Before running the following command, double check that you are in the **sphinx_maths** folder:

   ```
   sphinx-apidoc -o docs maths/
   ```

   If the command runs successfully, you will find the **maths.rst** file in the **docs** folder.

10. Open your code editor and locate the **index.rst** file. Update this file to include the **modules.rst** file by adding it below the **:caption:** directive, as shown in the example below:

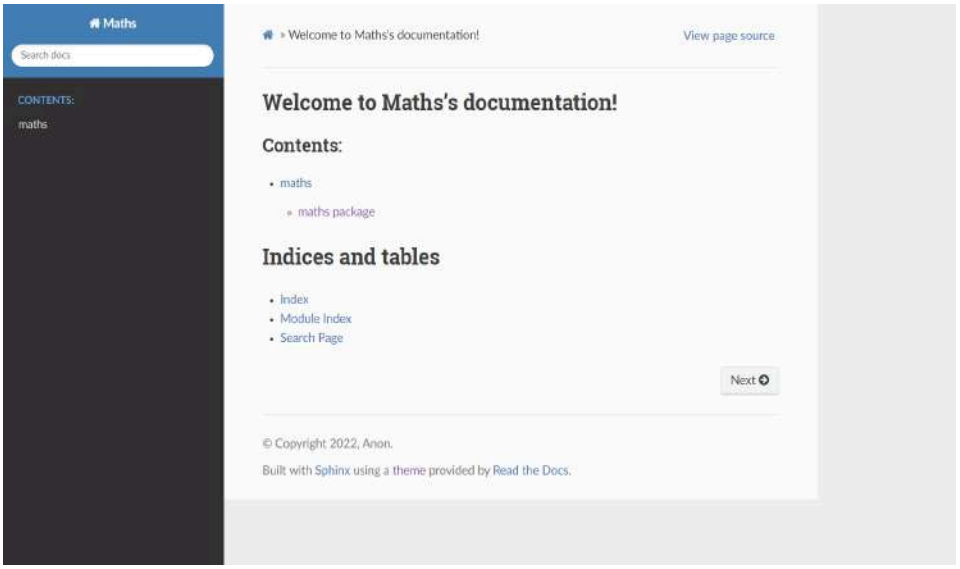
    ```
    Maths documentation
    ===================

    .. toctree::
       :maxdepth: 2
       :caption: Contents:

       modules
    ```

    Note the layout of the **index.rst** file may differ slightly from the example.

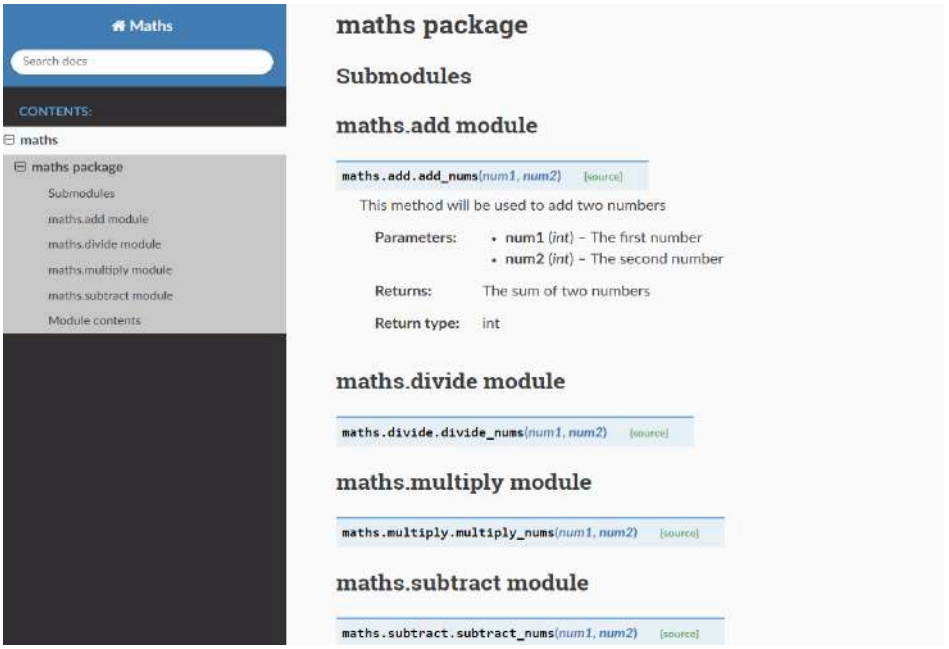
11. After modifying the **index.rst** file, in your command line change directories once again to the **docs** folder. Then, run the following command to generate the HTML documentation:

    ```
    make html
    ```

12. If the command executes successfully, you should find the generated HTML files in the **_build/html** folder. Open the **index.html** file in a web browser, and voilà! You will see your beautifully formatted documentation. After opening **index.html**, you should see a page similar to the following:



13. Navigate to the maths package in the documentation to explore the various sections. You will see that only the **add module** has documentation available. Don't worry if the other modules don't have documentation yet as this will be completed in **Practical task 2**.



Practical task 2

1. Complete the documentation for the **divide.py**, **multiply.py**, and **subtract.py** files. Open each file in your code editor and use the **add.py** file as a guide. The **add.py** file provides examples of how the documentation should be formatted. Refer to it to ensure that your documentation is consistent across all files.

2. After adding documentation to each of the files, generate the updated HTML documentation to reflect the changes. To do this, navigate to the **docs** folder in your command line and follow these instructions.

    a. Before we can generate the updated HTML documentation files, we will first need to remove the previously generated HTML documentation files. This can be done by running the following command:

    ```
    make clean
    ```

    b. After running the `make clean` command we can then proceed with regenerating the HTML documentation files using the following command:

    ```
    make html
    ```

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.