TASK

Software Documentation

Visit our website

Introduction

Reliable documentation is essential for any Software Engineer. Documentation helps keep track of all aspects of an application and it improves the quality of a software product. Successful documentation will make information easily accessible, it helps transfer knowledge to other developers, simplifies the product, and helps with maintenance. In this task, we discuss 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've 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

Good programming style

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

```python
        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:

```python
# 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 by using descriptive variable names, the correct spacing and indentation, and a summary comment at the top, it takes very little effort to understand the code. 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 look at using comments in more detail.

Comments

When you have a good programming style, comments that describe each line of code (e.g. `print(symbol) #... this prints the determined symbol`) is not necessary. However, comments have some important functions, including:

1. **Explaining the code**: for a particularly tricky piece of code, this is intended to help someone reading it to follow your logic and understand the process.

2. **Summarising the code**: as with the example above, this is a simple summary of the overall process of the code.

3. **Describing what the code is meant to be doing**: this indicates to the reader that a piece of the code is not doing what it should be, and possibly needs to be reworked.

4. **Code markers**: this is used to show where code still needs to be completed, and is usually indicated by a string of ##################, **************** or !!!!!!!!!!!!!!!!!!!!!!!!!!!. Code markers can also be used to separate logical sections of code.

5. **Commented out code for debugging**: this is code that is commented out until it is time to debug. For example, where a variable is usually assigned a value from user input, that line could be commented out and the value could be hard-coded for the sake of debugging. Once the debugging process is completed, these comments will usually be deleted. See below:

```
# number = int(input("Please enter a number"))
number = 6
print(number)
```

6. **Commented out code that is not meant to be run (at this time)**: if a programmer is working on a piece of code, they may be trying different approaches to find the most efficient or understandable way forward. At this time there might be pieces of code that are commented out. Once they have decided on the best way, the unused code will be deleted.

7. **Information that needs to be in the code, but cannot be expressed by the code itself**: these include comments about copyright, confidentiality, etc.

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

```
#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 is a string literal that you put as the first lines in a code block, for example, a function.

Typically, you use documentation strings to automatically generate the code documentation. Documentation strings are called **docstrings**.

Technically speaking, docstrings are not the comments. They create anonymous variables that reference the strings. Also, they're not ignored by the Python interpreter.

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 **here**.

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

```python
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 = []
```

```python
        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
```

```python
                :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
```

```python
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))
```

As you can see, each time you want to add a comment for Sphinx to add as documentation, we use the triple quotes before and after that which we are documenting.

```
"""
    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 type).

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 doctstring is a description of what the function is used for. After this, we see the 2 **: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 name is 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 so much 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.
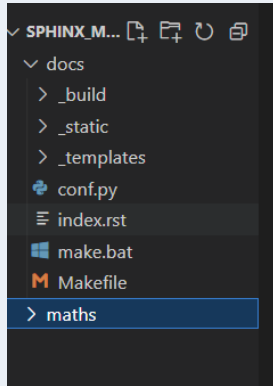
Compulsory Task 1

Follow these steps as we set up Sphinx and document the code in the **sphinx_maths** folder in this task folder:

- Firstly, let's create a new folder in the **sphinx_maths folder** named **"docs"**.

- You should now have 2 folders as follows:

docs	2022/08/17 15:44	File folder
maths	2022/03/17 14:55	File folder

- Install Sphinx using the following command in the command prompt for Windows:
  pip install -U sphinx

- For other operating systems **look here**.

- Next we want to install a theme that makes the documentation look good. Use the following command in the command prompt (this step is not compulsory):
  pip install sphinx-rtd-theme

- Please take note! Change the directory to the docs folder that you created and run the following command:
  sphinx-quickstart

- You will be prompted if you want to separate **source** and **build**, just press the enter key to accept the default which is 'no'.

- You are now prompted to enter a project name (call it "Maths").

  - Also, an Author name (use your name).

  - And a project release (you can use the following format 00.00.01).

- Lastly you will be asked for a language. Just press enter to default to English.

- If you open the project folder, you will see the following file/folder structure (we recommend using VS Code for Python development):

```
∨ SPHINX_M...
  ∨ docs
    > _build
    > _static
    > _templates
    ⚙ conf.py
    ≡ index.rst
    🪟 make.bat
    M Makefile
  > maths
```

- We will now need to edit the conf.py file to configure our documentation generator. The file can be found in the **docs/source** folder. Locate the section in the file that has an empty extensions array and edit as follows:

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

- Also add the following at the top of the **conf.py** file to ensure that Sphinx reads from the root folder of the project.

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

- Lastly find the string variable named html_theme and change the theme to **sphinx_rtd_theme** if you have installed the theme.

```python
html_theme = 'sphinx_rtd_theme'
```

- We will first need to generate the .rst files for the markup before we can then generate HTML documents from the .rst files. We will need to create a .rst file for the python files in the maths folder. First change the directory to the parent folder, **sphinx_maths**. Now run the following command:

  sphinx-apidoc -o docs maths/

  You will find **maths.rst** file in the docs folder if this runs successfully.

- Now include the **modules.rst** file in the **index.rst** file by adding it below the :caption: directive.

```
Welcome to Maths's documentation!
=================================

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

   modules
```
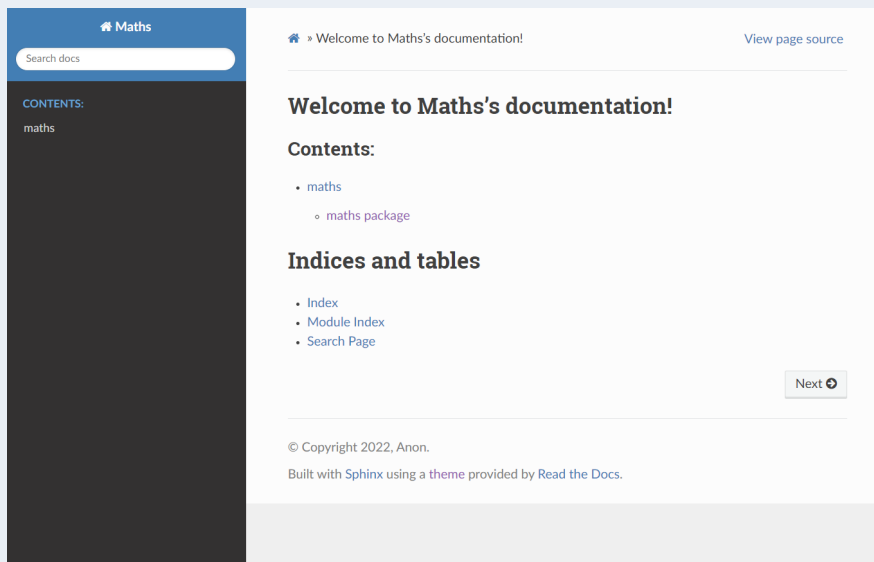
- Change directories once again to the docs folder and run the following command to generate the HTML documentation:
  make html

- You can find the HTML files in the **_build/html** folder. Open the index.html file in a browser and: *voila*! You now have beautiful documentation!

- Navigate to the maths package to see the documentation. You will see that only the add module has documentation, don't worry. You will complete the rest in Compulsory Task 2.

Compulsory Task 2

For this task follow the following steps:

- Please complete the documentation in the divide.py, multiply.py, and subtract.py files.

- Generate the HTML documentation for all of the modules. **Please note** that **before** you run the **make html** command again, you will need to navigate to the docs folder and run the **make clean** command first to remove the old documentation.

Rate us
Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.