



Python documentation

Why Documenting Your Code Is So Important-advance Python

“Code is more often read than written.”

— Guido van Rossum

- When you write code, you write it for two primary audiences: your users and your developers (including yourself). Both audiences are equally important.
- **“What in the world was I thinking?”** If you’re having a problem reading your own code, imagine what your users or other developers are experiencing when they’re trying to use or contribute to your code.

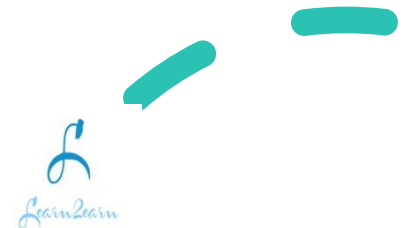


Why Documenting Your Code Is So Important-advance Python

- Many important Libraries just failed because they have no good documentation nor example how to use their own code!

“It doesn’t matter how good your software is, because **if the documentation is not good enough, people will not use it.**”

— *Daniele Procida*

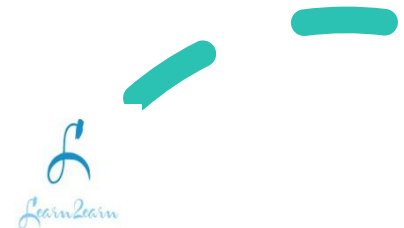


Commenting vs Documenting Code-advance Python

- In general, commenting is describing your code to/for developers.
- with well-written code, comments help to guide the reader to better understand your code and its purpose and design:

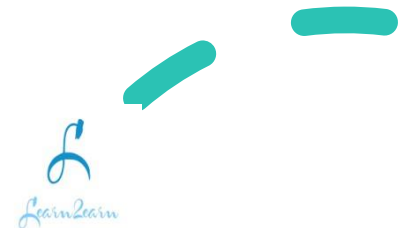
“Code tells you how; Comments tell you why.”

— *Jeff Atwood (aka Coding Horror)*



Commenting vs Documenting Code-advance Python

- Documenting code is **describing its use** and **functionality** to **your users**. While it may be helpful in the development process, the main intended audience is the users.
- Basics of Commenting Code:
 - According to PEP 8, comments should have a maximum length of 72 characters.
 - Commenting your code serves multiple purposes, including:
 - Planning and Reviewing:
 - Code Description
 - Algorithmic Description
 - Tagging.

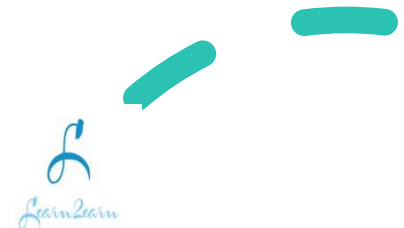


Commenting -Planning and Reviewing-advance Python

- When you are developing new portions of your code, it may be appropriate to first use comments as a way of planning or outlining that section of code. Remember to remove these comments once the actual coding has been implemented and reviewed/tested:

Python

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



Commenting - Code Description - advance Python

- Comments can be used to explain the intent of specific sections of code:

Python

```
# Attempt a connection based on previous settings. If unsuccessful,  
# prompt user for new settings.
```

Commenting - Algorithmic Description - advance Python

- When algorithms are used, especially complicated ones, it can be useful to explain how the algorithm works or how it's implemented within your code. It may also be appropriate to describe why a specific algorithm was selected over another.

Python

```
# Using quick sort for performance gains
```


Commenting – Tagging -advance Python

- The use of tagging can be used to label specific sections of code where known issues or areas of improvement are located. Some examples are: BUG, FIXME, and TODO.

Python

```
# TODO: Add condition for when val is None
```

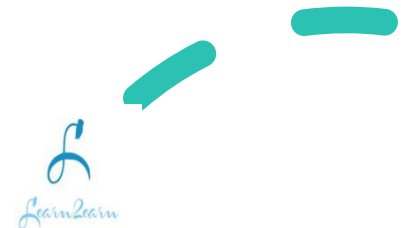
Commenting – Essential rules -advance Python

- Comments to your code should be kept brief and focused. Avoid using long comments when possible. Additionally, you should use the following four essential rules as suggested by Jeff Atwood:
 1. Keep **comments as close to the code** being described as possible. Comments that aren't near their describing code are frustrating to the reader and easily missed when updates are made.
 2. **Don't use complex formatting** (such as tables or — ASCII figures). Complex formatting leads to distracting content and can be difficult to maintain over time.



Commenting – Essential rules -advance Python

3. Don't include **redundant information**. Assume the reader of the code has a basic understanding of programming principles and language syntax.
4. Design your code to comment itself. The easiest way to understand code is by reading it. When you design your code using clear, easy-to-understand concepts, the reader will be able to quickly conceptualize your intent.



Commenting – Commenting Code via Type Hinting Python 3.5+ -advance Python

- It allows the developer to design and explain portions of their code without commenting. Here's a quick example:

Python

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

- From examining the type hinting, you can **immediately tell** that the function expects the input name to be of a type `str`, or `string`. You can also tell that the expected output of the function will be of a type `str`, or `string`, as well. While type hinting helps **reduce comments**, take into consideration that doing so may also make extra work when you are creating or updating your project documentation.

Commenting – Commenting Code via Type Hinting Python 3.5+ -advance Python

- It allows the developer to design and explain portions of their code without commenting. Here's a quick example:

Python

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

- From examining the type hinting, you can **immediately tell** that the function expects the input name to be of a type `str`, or `string`. You can also tell that the expected output of the function will be of a type `str`, or `string`, as well. While type hinting helps **reduce comments**, take into consideration that doing so may also make extra work when you are creating or updating your project documentation.

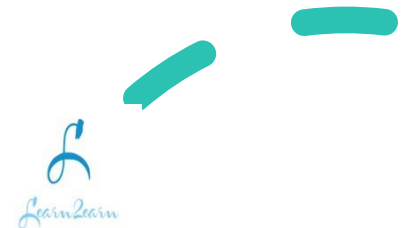
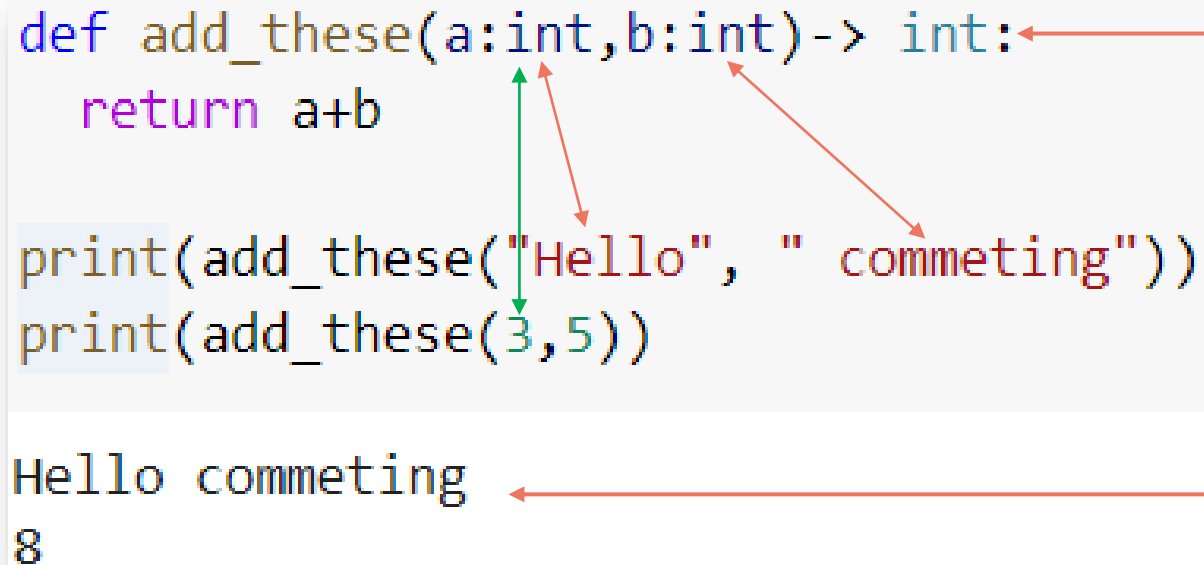
Commenting – Commenting Code via Type Hinting Python 3.5+ -advance Python

- Commenting never affect the execution :

```
def add_these(a:int,b:int)-> int:
    return a+b

print(add_these("Hello", " commeting"))
print(add_these(3,5))
```

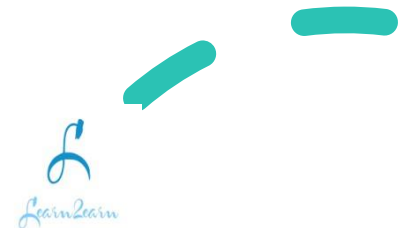
Hello commeting
8



Documenting Your Python Code Base Using Docstrings

-advance Python

- let's take a deep dive into documenting a Python code base.
- **Docstrings Background:**
 - Documenting your Python code is all centered on docstrings. These are built-in strings that, when configured correctly, can help your users and yourself with your project's documentation.
 - Along with docstrings, Python also has the built-in function **help()** that ***prints out the objects docstring to the console***. Here's a quick example:



Documenting Your Python Code Base Using Docstrings

-advance Python

Python

>>>

```
>>> help(str)
```

```
Help on class str in module builtins:
```

```
class str(object)
```

```
| str(object='') -> str
```

```
| str(bytes_or_buffer[, encoding[, errors]]) -> str
```

```
|
```

```
| Create a new string object from the given object. If encoding or  
| errors are specified, then the object must expose a data buffer  
| that will be decoded using the given encoding and error handler.  
| Otherwise, returns the result of object.__str__() (if defined)
```

```
| or repr(object).
```

```
| encoding defaults to sys.getdefaultencoding().
```

```
| errors defaults to 'strict'.
```

```
# Truncated for readability
```


Documenting Your Python Code Base Using Docstrings

-advance Python

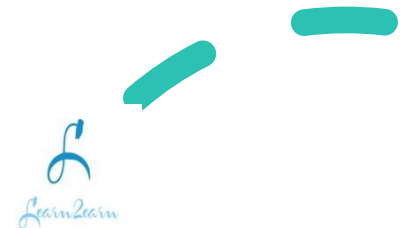
- built-in in Python a **magic property** there is `__doc__` for every type as :

Python

>>>

```
>>> print(str.__doc__)
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str
```

Create a new string object from the given object. If encoding or errors are specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.



Documenting Your Python Code Base Using Docstrings

-advance Python

- More and More we can edit/manipulate it, However, there are restrictions for built-ins:

```
Python >>>

>>> str.__doc__ = "I'm a little string doc! Short and stout; here is my input and pri
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/extension type 'str'
```

Documenting Your Python Code Base Using Docstrings

-advance Python

- Any other custom object can be manipulated:

Python

```
def say_hello(name):  
    print(f"Hello {name}, is it me you're looking for?")  
  
say_hello.__doc__ = "A simple function that says hello... Richie style"
```

Python

>>>

```
>>> help(say_hello)  
Help on function say_hello in module __main__:  
  
say_hello(name)  
    A simple function that says hello... Richie style
```

Documenting Your Python Code Base Using Docstrings

-advance Python

- Python has one more feature that simplifies docstring creation. Instead of **directly manipulating the `__doc__`** property, the strategic placement of the string literal **directly below the object** will automatically set the `__doc__` value. Here's what happens with the same example as above:

Python

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

Python

>>>

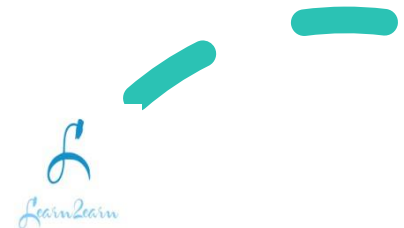
```
>>> help(say_hello)  
Help on function say_hello in module __main__:  
  
say_hello(name)  
    A simple function that says hello... Richie style
```

Documenting Docstring Types -advance Python

- The docstrings should use the **triple-double quote (""")** string format. This should be done whether the docstring is **multi-lined** or not. At a bare minimum, a docstring should be a quick summary of whatever is it you're describing and should be contained within a single line:

Python

```
"""This is a quick summary line used as a description of the object."""
```



Documenting Docstring Types -advance Python

- **Multi-lined docstrings** are used to further elaborate on the object beyond the summary. All multi-lined docstrings have the following parts:
- A one-line summary line
- A blank line proceeding the summary
- Any further elaboration for the docstring
- Another blank line

```
Python
"""This is the summary line
"""
This is the further elaboration of the docstring. Within this section,
you can elaborate further on details as appropriate for the situation.
Notice that the summary and the elaboration is separated by a blank new
line.
"""
# Notice the blank line above. Code should continue on this line.
```

Documenting Docstring Types -advance Python

- All docstrings should have the same max character length as comments **(72 characters)**. Docstrings can be further broken up into three major categories:
 - **Class Docstrings:** Class and class methods
 - **Package and Module Docstrings:** Package, modules, and functions
 - **Script Docstrings:** Script and functions

Documenting Class Docstrings-advance Python

- Class Docstrings are created for the class itself, as well as any class methods. The docstrings are placed immediately following the class or class method indented by one level:

Python

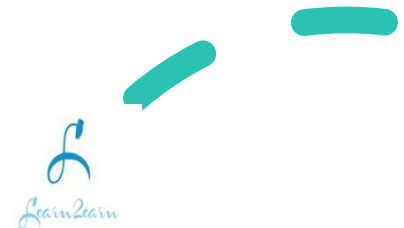
```
class SimpleClass:
    """Class docstrings go here."""

    def say_hello(self, name: str):
        """Class method docstrings go here."""

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


Documenting Class Docstrings-advance Python

- Class docstrings should contain the following information:
 - A brief summary of its purpose and behavior
 - Any public methods, along with a brief description
 - Any class properties (attributes)
 - Anything related to the interface for subclassers, if the class is intended to be subclassed



Documenting Class Docstrings-advance Python

- **The class constructor** parameters **should be documented within the `__init__` class method** docstring. Individual methods should be documented using their individual docstrings. Class method docstrings should contain the following:
 - A brief description of what the method is and what it's used for
 - Any arguments (both required and optional) that are passed including keyword arguments
 - Label any arguments that are considered optional or have a default value
 - Any side effects that occur when executing the method
 - Any exceptions that are raised
 - Any restrictions on when the method can be called

Documenting Class Docstrings-advance Python

- Let's take a simple example of a data class that represents an Animal. This class will contain a few class properties, instance properties, a `__init__`, and a single instance method:

Python

```
class Animal:
    """
    A class used to represent an Animal

    ...

    Attributes
    -----
    says_str : str
        a formatted string to print out what the animal says
    name : str
        the name of the animal
    sound : str
        the sound that the animal makes
    num_legs : int
        the number of legs the animal has (default 4)

    Methods
    -----
    says(sound=None)
        Prints the animals name and what sound it makes
    """
```

Documenting Class Docstrings-advance Python

- Let's take a simple example of a data class that represents an Animal. This class will contain a few class properties, instance properties, a `__init__`, and a single instance method:

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

def __init__(self, name, sound, num_legs=4):
    """
    Parameters
    -----
    name : str
        The name of the animal
    sound : str
        The sound the animal makes
    num_legs : int, optional
        The number of legs the animal (default is 4)
    """

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

Documenting Class Docstrings-advance Python

- Let's take a simple example of a data class that represents an Animal. This class will contain a few class properties, instance properties, a `__init__`, and a single instance method:

```
def says(self, sound=None):
    """Prints what the animals name is and what sound it makes.

    If the argument `sound` isn't passed in, the default Animal
    sound is used.

    Parameters
    -----
    sound : str, optional
        The sound the animal makes (default is None)

    Raises
    -----
    NotImplementedError
        If no sound is set for the animal or passed in as a
        parameter.
    """

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

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

Documenting Script Docstrings-advance Python

- **Scripts** are considered to be single file executables run from the console.
- Docstrings for scripts **are placed at the top of the file** and should be documented well enough for users to be able to have a sufficient understanding of how to use the script.
- It should be usable for its **“usage” message**, when the user incorrectly passes in a parameter or uses the -h option.
- Any custom or third-party imports should be listed within the docstrings to allow users to know which packages may be required for running the script. Here’s an example of a script that is used to simply print out the column headers of a spreadsheet:

Documenting Script Docstrings-advance Python

Python

```
"""Spreadsheet Column Printer
```

```
This script allows the user to print to the console all columns in the  
spreadsheet. It is assumed that the first row of the spreadsheet is the  
location of the columns.
```

```
This tool accepts comma separated value files (.csv) as well as excel  
(.xls, .xlsx) files.
```

```
This script requires that `pandas` be installed within the Python  
environment you are running this script in.
```

```
This file can also be imported as a module and contains the following  
functions:
```

```
    * get_spreadsheet_cols - returns the column headers of the file  
    * main - the main function of the script  
"""
```

Documenting Script Docstrings-advance Python

```
def get_spreadsheet_cols(file_loc, print_cols=False):
    """Gets and prints the spreadsheet's header columns

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

    Returns
    -----
    list
        a list of strings used that are the header columns
    """

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

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

    return col_headers
```


Documenting Script Docstrings-advance Python

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

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

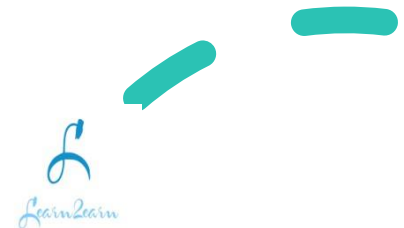
Documenting Package and Module Docstrings -advance Python

- Package docstrings should be placed at the top of the package's `__init__.py` file.
- This docstring should list the modules and sub-packages that are exported by the package.
- Module docstrings are similar to class docstrings. Instead of classes and class methods being documented, it's now the module and any functions found within. Module docstrings are placed at the top of the file even before any imports. Module docstrings should include the following:
 - A brief description of the module and its purpose
 - A list of any classes, exception, functions, and any other objects exported by the module



Documenting Package and Module Docstrings -advance Python

- The docstring for a module function should include the same items as a class method:
 - A brief description of what the function is and what it's used for
 - Any arguments (both required and optional) that are passed including keyword arguments
 - Label any arguments that are considered optional
 - Any side effects that occur when executing the function
 - Any exceptions that are raised
 - Any restrictions on when the function can be called



Documenting Docstring Formats - advance Python

- You may have noticed that, throughout the examples given in this tutorial, there has been specific formatting with common elements: **Arguments, Returns, and Attributes.**
- There are specific docstrings formats that can be used to help docstring parsers and users have a familiar and known format.

Formatting Type	Description	Supported by Sphinx	Formal Specification
Google docstrings	Google's recommended form of documentation	Yes	No
reStructuredText	Official Python documentation standard; Not beginner friendly but feature rich	Yes	Yes
NumPy/SciPy docstrings	NumPy's combination of reStructuredText and Google Docstrings	Yes	Yes
Epytext	A Python adaptation of Epydoc; Great for Java developers	Not officially	Yes

Documenting Docstring Formats - advance Python

- The **selection** of the docstring format is **up to you**, but you **should stick** with the same **format** throughout your document/project.
- The following are examples of each type to give you an idea of how each documentation format looks.

Google Docstrings Example

Python

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

Args:

```
    file_loc (str): The file location of the spreadsheet
```

```
    print_cols (bool): A flag used to print the columns to the console  
                        (default is False)
```

Returns:

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

```
"""
```

Documenting Docstring Formats - advance Python

- The **selection** of the docstring format is **up to you**, but you **should stick** with the same **format** throughout your document/project.
- The following are examples of each type to give you an idea of how each documentation format looks.

NumPy/SciPy Docstrings Example

Python

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

Parameters

```
-----
```

```
file_loc : str
```

```
    The file location of the spreadsheet
```

```
print_cols : bool, optional
```

```
    A flag used to print the columns to the console (default is False)
```

Returns

```
-----
```

```
list
```

```
    a list of strings representing the header columns
```

```
"""
```

Documenting : Docstring Formats -advance Python

- The **selection** of the docstring format is **up to you**, but you **should stick** with the same **format** throughout your document/project.
- The following are examples of each type to give you an idea of how each documentation format looks.

Epytext Example

Python

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

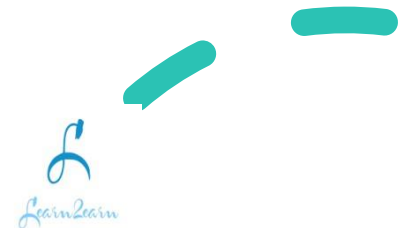
@param file_loc: str
@param file_loc: The file location of the spreadsheet
@param print_cols: bool
@param print_cols: A flag used to print the columns to the console
                    (default is False)
@rtype: list
@returns: a list of strings representing the header columns
"""
```

Documenting Your Python Projects

-advance Python

- Python projects come in all sorts of shapes, sizes, and purposes.
- The way you document your project should suit your specific situation.
- Keep in mind **who the users of your project** are going to be and adapt to their needs. Depending on the project type, certain aspects of documentation are recommended. The general layout of the project and its documentation should be as follows:

```
project_root/  
|  
├─ project/  # Project source code  
├─ docs/  
├─ README  
├─ HOW_TO_CONTRIBUTE  
├─ CODE_OF_CONDUCT  
└─ examples.py
```



Documenting Your Python Projects

-advance Python

Private Projects :

- Private projects are projects intended for **personal use** only and generally aren't shared with other users or developers.
- Documentation can be **pretty light on these types of projects**. There are some recommended parts to add as needed:
 - **Readme:** A brief summary of the project and its purpose. Include any special requirements for installation or operating the project.
 - **examples.py:** A Python script file that gives simple examples of how to use the project.
- Remember, even though private projects are intended for you personally, you are also considered a user. Think about anything that **may** be confusing to you down the road and make sure to capture those in either comments, docstrings, or the readme.

Documenting Your Python Projects

-advance Python

Shared Projects:

- Shared projects are projects in which you **collaborate** with a few other people in the development and/or use of the project. The “customer” or user of the project continues to be yourself and those limited few that use the project as well.
- Documentation should be a **little more rigorous** than it needs to be for a private project, mainly to help onboard new members to the project or alert contributors/users of new changes to the project. Some of the recommended parts to add to the project are the following:
 - **Readme:** A brief summary of the project and its purpose. Include any special requirements for installing or operating the project. ***Additionally, add any major changes since the previous version.***
 - **examples.py:** A Python script file that gives simple examples of how to use the projects.
 - **How to Contribute:** This should include how new contributors to the project can start contributing.



Documenting Your Python Projects

-advance Python

Public and Open Source Projects:

- Public and Open Source projects are projects that are intended to be shared with a **large group of users** and **can involve large development teams**. These projects should place as high of a priority on project documentation as the actual development of the project itself. Some of the recommended parts to add to the project are the following:
 - **Readme:** A brief summary of the project and its purpose. Include any special requirements for installing or operating the projects. Additionally, add any major changes since the previous version. ***Finally, add links to further documentation, bug reporting, and any other important information for the project.***
 - **How to Contribute:** This should include how new contributors to the project can help. ***This includes developing new features, fixing known issues, adding documentation, adding new tests, or reporting issues.***

Documenting Your Python Projects

-advance Python

Public and Open Source Projects:

- **Code of Conduct:** Defines how other contributors ***should treat each other*** when developing or using your software. ***This also states what will happen if this code is broken.*** If you're using Github, a Code of Conduct template can be generated with recommended wording. For Open Source projects especially, consider adding this.
- **License:** A plaintext file that ***describes the license*** your project is using. For Open Source projects especially, consider adding this.
- **docs:** A folder that contains further documentation.

Documenting Your Python Projects

-advance Python

The Four Main Sections of the docs Folder:

projects should have the following four major sections to help you focus your work:

- **Tutorials:** Lessons that take the reader by the hand through a series of steps to complete a project (or meaningful exercise). Geared towards the user's learning.
- **How-To Guides:** Guides that take the reader through the steps required to solve a common problem (problem-oriented recipes).
- **References:** Explanations that clarify and illuminate a particular topic. Geared towards understanding.
- **Explanations:** Technical descriptions of the machinery and how to operate it (key classes, functions, APIs, and so forth). Think Encyclopedia article.



Documenting Your Python Projects

-advance Python

The Four Main Sections of the docs Folder:

The following table shows how all of these sections relates to each other as well as their overall purpose:

	Most Useful When We're Studying	Most Useful When We're Coding
Practical Step	<i>Tutorials</i>	<i>How-To Guides</i>
Theoretical Knowledge	<i>Explanation</i>	<i>Reference</i>

Documenting Documentation Tools -**advance Python**

Documenting your code, especially large projects, can be daunting. Thankfully there are **some tools** out and references to get you started:

Tool	Description
Sphinx	A collection of tools to auto-generate documentation in multiple formats
Epydoc	A tool for generating API documentation for Python modules based on their docstrings
Read The Docs	Automatic building, versioning, and hosting of your docs for you
Doxygen	A tool for generating documentation that supports Python as well as multiple other languages
MkDocs	A static site generator to help build project documentation using the Markdown language. Check out Build Your Python Project Documentation With MkDocs to learn more.
pycco	A "quick and dirty" documentation generator that displays code and documentation side by side. Check out our tutorial on how to use it for more info.



Documenting Where Do I Start? -advance Python

The documentation of projects have a simple progression:

1. No Documentation
2. Some Documentation
3. Complete Documentation
4. Good Documentation
5. Great Documentation