COMMENTING AND DOCUMENTATION

DR IAN CORNELIUS

HELLO

- Learning Objectives:
    1. Understand the reasons as to why we should comment our code for both developers and end-users
    2. Understand the purpose of documentation and documenting your code
    3. Demonstrate the ability to use comments and creating documentation

WHY IS COMMENTING AND DOCUMENTATION IMPORTANT?

- Code is written for two audiences:
  - yourself (and other developers)
  - the end-user
- Both of these audiences are important
- Looking at code by other developers can be daunting, especially something that is quite advanced

COMMENTING VS. DOCUMENTING

COMMENTING

- Commenting involves describing your code for developers
- The intended audience is for the maintainers and developers of the code
- Comments are useful to guide the reader to better understand your code and its purpose

DOCUMENTING

- Documenting code describes its use and functionality to end-users
- This can be helpful in the development process; but its main audience is intended for the users

COMMENTING CODE

- Comments are created in Python using the hash/pound symbol (#)
- They are a brief statement that are no longer than a few sentences
  - i.e. a maximum length of 72 characters

```
</> def hello():
        # This is a simple comment that is preceding the print statement
        print("Hello 4061CEM")
```

- If a comment is going to be longer than 72 characters, using multiple lines is more appropriate

```
</> def hello():
        # This is a simple comment that is preceding the print statement.
        # But if we went over 72 characters, we need to go onto a new line.
        print("Hello 4061CEM")
```

REASONS TO COMMENT YOUR CODE

- Commenting your code can serve several purposes:
  - **planning and reviewing**: when in the first stages of development, comments can plan or outline sections of your code
  - **code description**: comments are useful for explaining the intent of a specific sections of your code
  - **algorithmic description**: some algorithms can be complicated, so it is useful to explain how the algorithm works or its implementation in your code; you may also want to include reasoning as to why you chose this particular algorithm
  - **tagging**: useful for label specific sections of code where known issues or areas of improvement are located, i.e. TODO or BUG

RULES OF COMMENTING

- When it comes to commenting, there are four essential rules:
  1. keep the comments close to the section of code they are describing
  2. do not use complex formatting, as this can lead to distracting content
  3. do not include redundant information, assume that the reader of the code has a basic understanding of programming
  4. design the code in a way that it comments itself; when designing code use clear and easy-to-understand concepts
- Comments should be kept brief and focused, they are designed for the reader
  - this will guide them into understanding the purpose of your code

TYPE HINTING

- Type hinting allows developers to design and explain portions of their code without commenting
- Whilst it can reduce comments, it creates extra work when creating or updating the projects documentation

```
def hello_person(name: str) -> str:
    return f"Hello {name}, welcome to 4061CEM!"
```

DOCUMENTING CODE

- Documenting your code can be achieved using docstrings
- Docstrings are stored inside an object in a property called `__doc__`
  - you are unable to edit the docstrings of built-in data types
- Calling a docstring can be performed using the `help()` function

```
def hello_person(name: str) -> str:
    return f"Hello {name}, welcome to 4061CEM!"
hello_person.__doc__ = """A function that says "Hello" to the user
                          and welcomes them to the module."""
```

```
help(hello_person)
```

```
Help on function hello_person in module __main__:
hello_person(name: str) -> str A function that says “Hello” to the user and welcomes them to the module.
```

DOCSTRING TYPES (1)

- The purpose of a docstring is to provide users with a brief overview of the object
- They are to be kept concise and easy to maintain, but still elaborate enough for users to understand their purpose
- Docstrings should always use the triple-double quote (""")
  - this should be done, even if the docstring is multi-line or not
- The bare minimum documentation should be a quick summary of the code and contained on a single line

DOCSTRING TYPES (2)

MULTI-LINE DOCSTRINGS

- These are used to further elaborate on an object beyond a summary
- Multi-lined docstrings should contain the following parts:
  - a single-line summary
  - a blank line proceeding the summary
  - any further elaboration required
  - a further blank line proceeding the elaboration

```
</> """This is a single line explaining some code.

    We then go into more detail of the docstring. In this section you can discuss
    more details about the algorithm you have chosen. Note, that the summary and
    elaboration is separated by a single, blank new line.
    """


    def function_header(parameter):
```

DOCSTRINGS TYPES (3)

- All docstrings must have the same max character length as comments
  - that is 72 characters
- There are three major categories of docstrings:
  1. class docstring
  2. package and module docstrings
  3. script docstrings

CLASS DOCSTRING

- These are docstrings that are created for the class itself, alongside any class functions also
- They are placed immediately following the class or class function indented by one level
- Class docstrings should contain the following information:
  - brief summary of its purpose and behaviour
  - any public functions, along with a brief description
  - any class properties (variables)
  - anything related to the interface for subclasses
    - if the class is intended to be a subclass

CLASS FUNCTION DOCSTRING

- Any class constructor parameters should be documented within the `__init__()` function docstring
- Individual functions should be documented using their individual docstrings
- Class function docstrings should contain the following:
  - brief description of what the function is and what it is used for
  - any arguments (both required and optional) that are passed, including the keyword arguments
  - any side effects that occur when executing the function
  - any exceptions that may be raised
  - any restrictions on when the function can be called

EXAMPLE OF A CLASS DOCSTRING

```
class Dog:
        """
        A class used to represent a dog.


        ...


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

        Methods
        -------
        says(sound=None)
            Prints the dogs name and the sound it makes
        """

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

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

PACKAGE AND MODULE DOCSTRINGS

- Package docstrings should be placed at the top of the packages `__init__.py` file
  - should list the modules and sub-packages that are exported by the package
- Module docstrings are similar to a class docstring
  - instead of class and class methods being documented, it is now the modules and their functions
  - these docstrings are placed at the top of the file, before any imports
- Module docstrings should include:
  - brief description of the module and its purpose
  - list of any classes, exceptions, functions and other objects exported by the module
- The docstring for a modules function should include the same items as a class function:
  - brief description of what the function is and what it is used for
  - any arguments (both required and optional) that are passed, including the keyword arguments
  - any side effects that occur when executing the method
  - any exceptions that may be raised
  - any restrictions on when the method can be called

SCRIPT DOCSTRINGS

- Scripts are a single file executable that runs from the console
- Docstrings for a script are placed at the top of the file
  - enough documentation should be provided that the user has a sufficient understanding of the script
  - should provide a useful message when the user incorrectly passes in a parameter or uses the `-h` option
- Third-party imports should be listed within the docstring
  - enables users to know which packages they may require to run the script

DOCSTRING FORMATS

- There are different formatting rules for docstrings, so there is no universally accepted method
- Some of the most common formatting types are:
  - Google Docstrings
  - reStructured Text
  - NumPy/SciPy Docstrings
  - Epytext
- The examples provided in this lecture are NumPy/SciPy docstrings

GOOGLE DOCSTRINGS EXAMPLE

```
</> """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
    """
```

RESTRUCTURED DOCSTRINGS EXAMPLE

```
</> """Gets and prints the spreadsheet's header columns

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

NUMPY/SCIPY DOCSTRINGS EXAMPLE

```
</> """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
    """
```

EPYTEXT DOCSTRINGS EXAMPLE

```
</> """Gets and prints the spreadsheet's header columns

    @type file_loc: str
    @param file_loc: The file location of the spreadsheet
    @type 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

- When it comes to documenting your work, you should consider the following:
  - **README**:
    - provide a brief summary on the project and its purpose
    - are there any special requirements for installing or operating the project?
    - add any major changes between versions
    - provide links to further documentation, bug reports
  - **example files/scripts**:
    - provide a script that gives examples of how your project works

GOODBYE

- Questions?
  - Post them in the **Community Page** on Aula
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk