King Arthur: ‘What does it say?’

Maynard: ‘It reads, ‘Here may be found the last words of Joseph of

Arimathea. He who is valiant and pure of spirit may find the Holy Grail

in the Castle of aaarrrrggh’.’

King Arthur: ‘What?’

Maynard: ‘…The Castle of aaarrrrggh.’

Bedevere: ‘What is that?’

Maynard: ‘He must have died while carving it.’

Programming Style

Code should be written in a style that makes it easier to comprehend. How you structure and layout your code is called programming style and it is a seemingly trivial but important part of ensuring your code is easy to read. (This is called making the code ‘readable’.) In this course we will following the Google Python Style Guide when writing code. Please ensure that you read and follow these rules. In the style guide the first section, Python Language Rules, relates to how to use particular language features. The second section, Python Style Rules, relates to how to structure your code. The style rules are most immediately applicable to the code you write in the early stages of this course. Some of the language rules will become more applicable later in the course.

Commenting

Writing Comments

Documentation is a software engineering concept that provides the “explanation” of the code. Comments should describe what the code is supposed to do, how to use it properly, and anything else that would be useful to know. A comment is a piece of syntax in a programming language which lets us describe the program code. We can add comments without affecting what our code actually does. Programming languages do this to give us a way of easily documenting our code.

Documenting code is very important — it aids communication between developers and is a great help when it comes to maintenance! Just as an agreed upon layout style is critical, the same holds for commenting style. In Python there are two types of comments, each serving a different purpose.

The first style of comment uses the # character followed by the comment until the end of the line. These comments are useful for describing complicated lines of code, or as a reminder for what a particular line of code is used for. These lines are ignored by the Python interpreter and are mainly notes to the writer of the code.

The second, more important style of comments, are called triple-quote comments or *docstrings*. These are written using triple quotes (three quote marks: `"""`). Docstrings are meant to explain what use a function serves, without going into the details of “how it does what it does”. In Python, docstrings, are composed of two important parts. The first is a brief explanation of what the function does. This is followed by a detailed explanation of how to use the function. This detailed explanation should include a description of the parameters; any preconditions, which are any requirements that need to be met before the function is called in order for the

function to perform correctly; a description of what the function returns; and possibly some examples of usage. We can think of these comments as a kind of contract between the writer and user of the function. The writer is promising that, if the user calls the function with arguments of the correct type that satisfy the precondition (if there is one), the function will behave as described in the comment. If the user calls the function with arguments of the wrong type, or that do not satisfy the precondition, the writer is not responsible for the function's behaviour.

Function Docstrings

Below is our `discuss` function updated with some helpful comments, which is in the `interaction.py` file.

```
def discuss(topic) :
    """Discuss a topic with the user and return their response.

    Ask if the user likes the topic and why.

    Parameters:
        topic (str): The topic under discussion.

    Return:
        str: Response to the question of why they like the topic.
    """
    like = input("Do you like " + topic + "? ")
    response = input("Why do you think that? ")
    return response
```

In this course, we will use the following triple quote commenting style.

The opening triple quote should be the first line of the body, indented as part of the body. The first line should give a short summary of the function's behaviour. If necessary, this line is followed by a blank line, and then by a more detailed explanation of the function's behaviour. This is followed by a blank line followed by a description of each parameter. The parameter description should indicate the type the function expects for that parameter, the type is in parenthesis after the function name, and then should provide a description of what the function expects to be passed to that parameter. If the function does not have any parameters this section is omitted. Following the parameter description is another blank line and then a description of what the function returns. The return description should indicate the type of value being returned and then describe what is being returned. In this case, the `discuss` function has one parameter `topic` of type `str` (short for "string") that is the topic used for the discussion in the function's body. The function returns a `str` that is the response to the question of why the user likes the topic or not. We can also add a precondition to the comments. Preconditions state what must be done (be logically true) before the function is called. Often this is a constraint on the values of some parameters, over and above the type constraint. (i.e. The code calling the function must ensure that the value of the parameters meets the precondition constraints.) The precondition may also be something that needs to be done before the function is called (e.g. some other function must be called before this function is called). The prime numbers example below shows a function with preconditions. Examples of usage demonstrate what the caller can expect when the use the function. These examples serve two purposes. Firstly, they demonstrate what result will be returned with particular parameters. This is useful for functions with complex logic. Secondly, the example of usage provides test cases, as the function can be executed and the results checked against the results indicated in the example of usage. The examples of usage should be formatted to look like an interactive Python session in the interpreter. This allows the Python doctest tool to automatically test the function to

ensure it produces the expected results. (**Note**: Initially do not worry about understanding how to use doctest or getting the examples of usage perfectly formatted. It is more important to get the idea of writing comments that help other programmers, than it is to worry about understanding automatic testing. Automated testing concepts will be explored in detail in later courses.) At the end is a line containing the terminating triple quotes.

```
def is_prime(num) :
    """Returns True iff 'num' is prime.

    Parameters:
        num (int): Integer value to be tested to see if it is prime.

    Return:
        bool: True if 'num' is prime. False otherwise.

    Preconditions:
        num > 1

    Examples:
        >>> is_prime(2)
        True
        >>> is_prime(3)
        True
        >>> is_prime(4)
        False
        >>> is_prime(5)
        True
        >>> is_prime(9)
        False
    """
```

The # style comments are completely ignored by the interpreter. On the other hand the triple quotes comments become part of the function. If you load `interaction.py` into the interpreter and then start typing a function call, you will see that the first line of the comment appears in the little pop-up window, as shown below.

Further, some python tools, like `pydoc`, extract this documentation to, for example, generate documentation. Also, below is an example using the `help` function which displays the docstring comments of the function.

```
>>> help(discuss)
Help on function discuss in module __main__:

discuss(topic)
    Discuss a topic with the user and return their response.

    Ask if the user likes the topic and why.

    Parameters:
        topic (str): The topic under discussion.

    Return:
        str: Response to the question of why they like the topic.

>>>
```

Classes and Methods Docstrings

Commenting classes and their methods is slightly different to commenting functions. The class itself requires a comment describing what the class does. Again not explaining how it does it. The methods are commented similarly to functions but with a slight difference to the type declaration. The other difference is that the `__init__` method of the class has a **Constructor** in place of the type declaration. Below is the Point class from the Class Design notes.

```
class Point(object) :
    """A 2D point ADT using Cartesian coordinates."""

    def __init__(self, x, y) :
        """Construct a point object based on (x, y) coordinates.

        Parameters:
            x (float): x coordinate in a 2D cartesian grid.
            y (float): y coordinate in a 2D cartesian grid.
        """
        self._x = x
        self._y = y

    def x(self) :
        """(float) Return the x coordinate of the point."""
        return self._x

    def y(self) :
        """(float) Return the y coordinate of the point."""
        return self._y

    def move(self, dx, dy) :
        """Move the point by (dx, dy).

        Parameters:
            dx (float): Amount to move in the x direction.
            dy (float): Amount to move in the y direction.
        """
        self._x += dx
        self._y += dy
```

Note that the comment for the `__init__` method does not have a Return: comment. This is due to a class constructor not returning a value, it creates an object of the class type instead of returning a value.

Also notice that the `self` parameter is never described in the Parameters: comment. Every method that operates on an object must have a `self` parameter, and it refers to the object on which the method operates. Consequently its type and value are always known. When calling a method on an object you do not pass the object as a parameter, it is implicitly passed as part of the method call. The `move` method of `Point`, would be called like: `point_object.move(1.0, 2.5)`. In this case the `self` parameter refers to the `point_object`.