Assignment 2 (15 Marks)

Due: Friday 3 May 2019 at 8:30pm

Introduction

The second assignment is a simple weather prediction and event planning system. You will provide some information about how the weather could affect an event. The system will predict the weather and indicate if conducting the event is advisable.

The purpose of this assignment is to help you to understand how to make use of classes to define an object's structure. And, to help you understand how to implement program functionality by having several objects interacting with each other, by sending messages to each other.

Overview

You are provided with three files for this assignment. They are **`weather_data.py`**, **`prediction.py`** and **`event_decision.py`**. Each file defines a set of classes, which are described below. You will need to implement parts, or all, of the classes in the **`prediction.py`** and **`event_decision.py`** files.

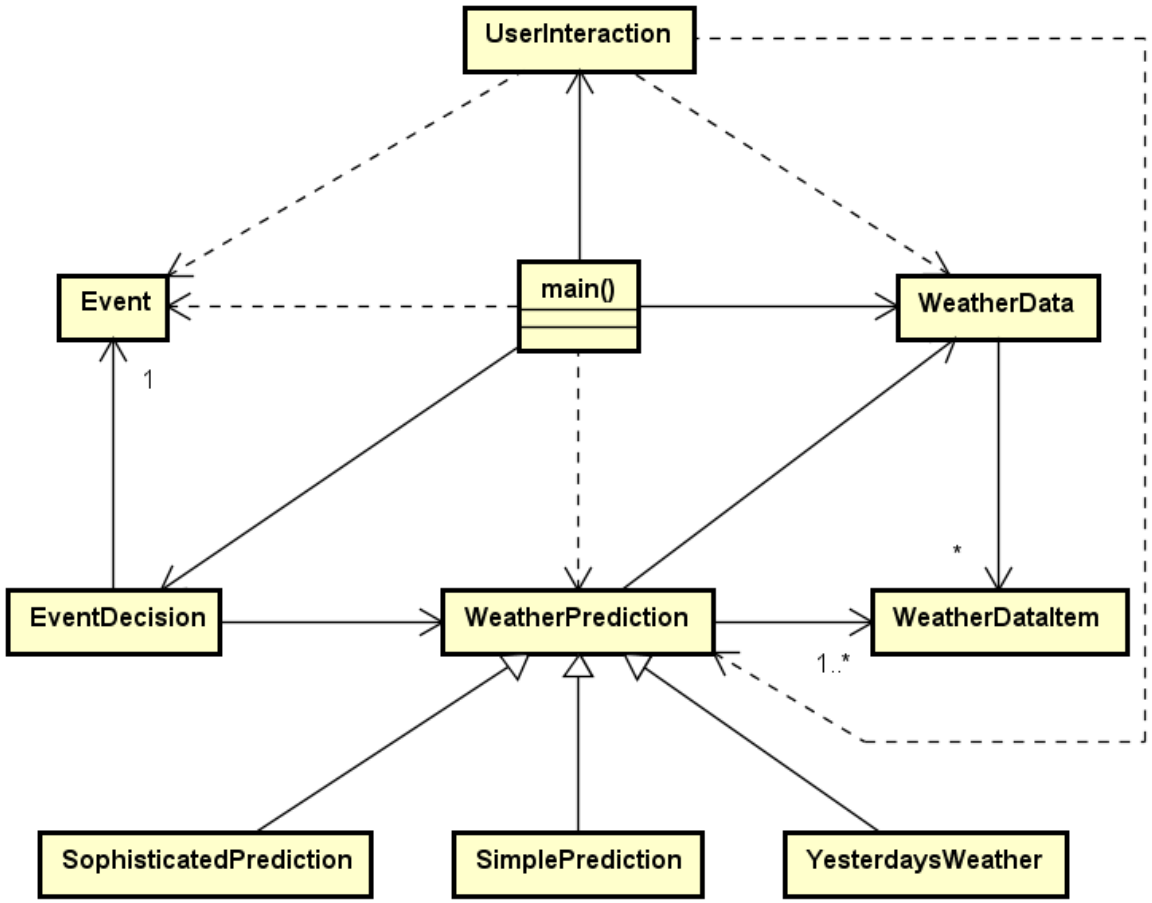
The file, **`weather_data.py`**, contains the **WeatherData** and **WeatherDataItem** classes. **WeatherData** represents all the weather data that is loaded from the data file. **WeatherDataItem** represents weather data for a single day. It provides methods to access this data.

The **`prediction.py`** file defines the super class **WeatherPrediction**. It defines a set of methods that you need to override in subclasses that you implement for the assignment. An example of doing this is provided by the **YesterdaysWeather** subclass. You need to implement the **SimplePrediction** and **SophisticatedPredication** subclasses.

The file, **`event_decision.py`**, provides a description of the **EventDecision** class. It uses data about the event to determine if aspects of the weather would impact on the event. You will need to implement the **__init__** and **advisability** methods. The **advisability** method determines how advisable it is to continue with the event, based on the weather prediction. Two other classes defined in this file are **Event** and **UserInteraction**. The **Event** class represents the data about an event and defines a set of methods that you will need to implement. The **UserInteraction** class provides methods that allow the program to interact with the user. It defines a set of methods that you will need to implement. Two of these methods, **get_prediction_model** and **output_advisability**, are partially implemented with some example ideas that you may use in your solution.

Design

The following diagram provides an overview of the classes in this design and their relationships.



The **WeatherDataItem** class defines a type of object that holds weather data for one day. It stores the amount of rainfall, the maximum and minimum temperatures, the number of hours of sunshine, average relative humidity, average and maximum wind speeds, predominant direction of the wind, extent of cloud cover and the mean sea level air pressure. This is the data that will be used to predict the next day's weather. The **WeatherDataItem** class has methods that provide access to the data.

An object of the **WeatherData** class loads the weather data from a file and creates a list of **WeatherDataItem** objects to hold the data. The **load** method takes a string as a parameter that is the name of the file from which the file is to be loaded. The **get_data** method takes an integer parameter that is the number of days of data to return. The data is returned as a list of **WeatherDataItem** objects. The **size** method returns the number of **WeatherDataItem**s being stored in the **WeatherData** object.

The **WeatherData** and **WeatherDataItem** classes are implemented for you.

The **WeatherPrediction** class defines the abstract interface for any weather prediction model. Each different prediction model will be a subclass of **WeatherPrediction**. The **WeatherPrediction** class has a reference to a **WeatherData** object, which is used to access the data to make a prediction. Each of the methods in **WeatherPrediction**, aside from **__init__**, raise the **NotImplementedError**. This is because these methods need to be implemented in the subclasses. An example of doing this is provided by the **YesterdaysWeather** class. An object of this class makes a prediction by accessing yesterday's **WeatherDataItem** object and using its data to indicate that tomorrow's weather will be pretty much the same as yesterday's. You need to implement subclasses for two other prediction models. These are called **SimplePrediction** and **SophisticatedPrediction**.

An object of the **EventDecision** class uses the predictions made by one of the subclasses of **WeatherPrediction** and the details from an object of the **Event** class to indicate how advisable it is to continue with an event.

An object of the **UserInteraction** class provides the user interface that allows a user to interact with the program. It gets the details of an event from the user and creates an **Event** object to hold this data. It also allows a user to select which weather prediction model to use and creates the appropriate object of a subclass of **WeatherPrediction**. When it does this it passes the **WeatherData** object to the **WeatherPrediction** object. After the **EventDecision** object has determined the advisability of continuing with the event, the **UserInteraction** object is given the advisability rating and outputs these details to the user. The **UserInteraction** object also provides a method to check to see if the user wants to try checking the advisability using a different prediction model.

The **main** function starts the execution of the program. It creates a **WeatherData** object and gets it to load the data from a file. It creates a **UserInteraction** object to provide the user interface. The main function gets the user interface to ask the user for event details and to choose the prediction model. Once this is done, the main function then gets the **EventDecision** object to determine the advisability of continuing with the event and then has the user interface output the result.

Example Usage

```
Let's determine how suitable your event is for the predicted weather.
What is the name of the event? My Event
Is the event outdoors? Y
Is there covered shelter? Y
What time is the event? 18
Select the weather prediction model you wish to use:
  1) Yesterday's weather.
  2) Simple prediction.
  3) Sophisticated prediction.
> 1
Based on the YesterdaysWeather model, the advisability of holding My Event is
2.8600000000000003.

Would you like to check again? Y
Select the weather prediction model you wish to use:
  1) Yesterday's weather.
  2) Simple prediction.
  3) Sophisticated prediction.
> 2
Enter how many days of data you wish to use for making the prediction: 4
Based on the SimplePrediction model, the advisability of holding My Event is 5.

Would you like to check again? Y
Select the weather prediction model you wish to use:
  1) Yesterday's weather.
  2) Simple prediction.
  3) Sophisticated prediction.
> 3
Enter how many days of data you wish to use for making the prediction: 12
Based on the SophisticatedPrediction model, the advisability of holding My Event
is 5.

Would you like to check again? N
```

Tasks

You need to implement the **Event**, **UserInteraction**, **EventDecision**, **SimplePrediction** and **SophisticatedPrediction** classes. Method definitions are provided for some classes in the **`prediction.py`** and **`event_decision.py`** files. You may **not** change the interfaces (names, parameters or return type) of any of these methods. You may add other private instance variables and methods to these classes to implement your logic and to structure your code.

Event defines an object that holds data about a single event and provides access to that data. The Event constructor will take as parameters:

- name: a string representing the name of the event,
- outdoors: a Boolean value representing whether the event is outdoors,
- cover_available: a Boolean value representing whether there is cover available,
- time: an integer from 0 up to, but not including, 24, indicating the closest hour to the starting time of the event.

You need to implement the following methods:

- __init__ – Stores local references to the given parameters
- get_name – Returns the Event name
- get_time – Returns the integer time value
- get_outdoors – Returns the Boolean outdoors value
- get_cover_available – Returns the Boolean cover_available value
- __str__ – Returns a string representation of the Event in the following format:
  'Event(*name* @ *time*, *outdoors*, *cover_available*)'
  One example might look like: 'Event(Party @ 12, True, False)'

UserInteraction defines an object that is the program's user interface. Unlike the first assignment, you have some flexibility in how you display your prompts and results to the user. You need to implement the following methods:

- get_event_details – Prompt the user to enter the data required to create an **Event** object, and return that object. The prompts need to be in the same order as shown in the example usage (event name, outdoors, shelter, time). The event name can be any string. If the event is outdoors of if there is shelter should be indicated by the user entering either 'Y' or 'Yes', in any combination of lower or upper case characters. The **UserInteraction** object should store a reference to the **Event** object, to be able to refer to it in other methods (e.g. output_advisability).
- get_prediction_model – An initial version of this method is provided. You need to extend it when you implement each new subclass of **WeatherPrediction**. The user needs to enter the digit corresponding to their choice of prediction model. If the prediction model is the simple or sophisticated prediction model, the user needs to then enter how many days of data should be used by the model. The **UserInteraction** object should store a reference to the selected **WeatherPrediction** object, to be able to refer to it in other methods (e.g. output_advisability).
- output_advisability – Display the result of determining how advisable it is to continue with the event.
- another_check – Ask the user if they would like to try using a different weather prediction model. The user needs to enter 'Y' or 'Yes' to indicate they want to perform another check. The user enters 'N' or 'No' to indicate they do not want to perform any more checks. This method only asks if they want to try again and returns a bool value indicating their response. It does not call other methods to restart the prediction.

EventDecision defines an object that uses the predicted weather to determine the impact it will have on an event. You need to implement the following methods:

- __init__ – Stores local references to the **Event** and **WeatherPrediction** objects passed as parameters.
- _temperature_factor – Returns the temperature factor which is determined by the following steps:
  1. The predicted temperatures may be adjusted based on the humidity. If the humidity is greater than 70% a humidity factor is calculated by dividing the humidity value by 20. The humidity factor should be added to the high and low temperatures, if they are positive; or subtracted from any temperature if it is negative.
  2. An initial temperature factor is calculated based on the following rules:
     a) If the time is between 6 and 19 inclusive and the event is taking place outdoors, and the adjusted high temperature is greater than or equal to 30, then the temperature factor is the adjusted high temperature divided by -5 and then 6 is added to the result.
     b) If the adjusted high temperature is greater than or equal to 45, regardless of time or if the event is indoors or outdoors, the temperature factor is calculated using the same formula (high temp ÷ -5 + 6).
     c) If the time is between 0 and 5 inclusive or 20 and 23 inclusive and the adjusted low temperature is less than 5, and the adjusted high temperature is less than 45, then the temperature factor is the adjusted low temperature divided by 5 and then 1.1 is subtracted from the result (low temp ÷ 5 - 1.1).
     d) If the adjusted low temperature is greater than 15 and the adjusted high temperature is less than 30, the temperature factor is the low temperature subtracted from the high temperature and the result divided by 5 ((high - low) ÷ 5).
     e) In all other cases, the initial temperature factor is 0.
  3. After calculating the initial temperature factor, if it is negative because of high temperature (rules a or b) add 1 to the temperature factor for each of the following conditions:
     - There is cover available for the event.
     - The wind speed is greater than 3 and less than 10.
     - The cloud cover is greater than 4.

- _rain_factor – Returns the rain factor which is determined by the following steps:
  1. An initial rain factor is calculated based on the following rules.
     a) If the chance of rain is less than 20%, then the rain factor is the chance of rain divided by -5 and then 4 is added to the result (chance of rain ÷ -5 + 4).
     b) If the chance of rain is greater than 50%, then the rain factor is the chance of rain divided by -20 and then 1 is added to the result (chance of rain ÷ -20 + 1).
     c) In all other cases, the initial rain factor is 0.
  2. After calculating the initial rain factor:
     - If the event is outdoors and cover is available, and the wind speed is less than 5, add 1 to the rain factor.
     - If the rain factor is less than 2 and the wind speed is greater than 15, then divide the wind speed by -15 and add this result to the rain factor (rain factor + (wind speed ÷ -15)). If the rain factor is less than -9 then set it to be -9.

- advisability – Returns the advisability ranking, which is calculated by the following:
  1. Add the final temperature factor and rain factor together. If the result is less than -5, set it to be -5. If the result is greater than 5, set it to be 5. Return this result as the advisability score.

SimplePrediction defines an object that predicts the weather based on the average of the past *n* days' worth of weather data, where *n* is a parameter given to the class at construction. You need to override the methods defined in the **WeatherPrediction** class.

- __init__ – Retrieves and stores references to the past *n* days' weather data. If *n* is greater than the number of days of weather data that is available, all of the available data is stored and used, rather than *n* days.
- get_number_days – Returns the number of days of data being used
- chance_of_rain – Calculate the average rainfall for the past *n* days. Multiply this average by 9. If the resulting value is greater than 100, set it to be 100. Return this result.
- high_temperature, low_temperature – Each should return the highest/lowest temperature recorded in the past *n* days
- humidity and cloud_cover – Each should calculate the average of their data from the past *n* days and return this result.
- wind_speed – Use only the average wind speed for the past *n* days, calculate the average value and return this result.

SophisticatedPrediction defines an object that predicts the weather based on the past *n* days' worth of weather data, where *n* is a parameter given to the class at construction. You need to override the methods defined in the **WeatherPrediction** class.

- __init__ – Retrieves and stores references to the past *n* days' weather data. If *n* is greater than the number of days of weather data that is available, all of the available data is stored and used, rather than *n* days.
- get_number_days – Returns *n*, the number of days of data being used
- chance_of_rain – Calculate the average rainfall for the past *n* days. If yesterday's air pressure was lower than the average air pressure for the past *n* days, multiply the average rainfall by 10. If yesterday's air pressure was higher than the average air pressure for the past *n* days, multiply the average rainfall by 7. After taking into account the air pressure, if yesterday's wind direction was from any easterly point (NNE, NE, ENE, E, ESE, SE or SSE) multiply the average rainfall by 1.2. If the resulting value is greater than 100, set it to be 100. Return this result.
- high_temperature – Calculate the average high temperature for the past *n* days. If yesterday's air pressure was higher than the average air pressure for the past *n* days, add 2 to the calculated average high temperature. Return this result.
- low_temperature – Calculate the average low temperature for the past *n* days. If yesterday's air pressure was lower than the average air pressure for the past *n* days, subtract 2 from the calculated average low temperature. Return this result.
- humidity – Calculate the average humidity for the past *n* days. If yesterday's air pressure was lower than the average air pressure for the past *n* days, add 15 to the calculated average humidity. If yesterday's air pressure was higher than the average air pressure for the past *n* days, subtract 15 from the calculated average humidity. If the resulting value is less than 0 or greater than 100, set it to be 0 or 100. Return this result.
- cloud_cover – Calculate the average cloud cover for the past *n* days. If yesterday's air pressure was lower than the average air pressure for the past *n* days, add 2 to the calculated average cloud cover. If the resulting value is greater than 9, set it to be 9. Return this result.
- wind_speed – Calculate the average of the average wind speed for the past *n* days. If yesterday's maximum wind speed was greater than four times the calculated average wind speed, multiply the calculated average wind speed by 1.2. Return this result.

When an object of any of the **WeatherPrediction** subclasses is created, it will store the weather data that it will use to make its predictions. Loading new data into the **WeatherData** object will not update the data used by the **WeatherPrediction** object.

Hints

The focus of this assignment is on demonstrating your ability to implement a simple object-oriented program using objects, classes and inheritance. It is possible to pass the assignment if you only implement some of the specified functionality. You should design and implement your program in stages, so that after the first stage you will always have a working previous partial implementation.

It is recommended that you start by implementing the **Event** class and then the **EventDecision** class. You can then implement the **UserInteraction** class. This will allow you to start using the program. The provided **YesterdaysWeather** class will give you an initial weather prediction model. You could implement the advisability method in **EventDecision** to initially return a random value without implement the _temperature_factor and _rain_factor methods. This allows you to run the program and create an **Event** object, even though the logic does not do anything useful. After initially testing **Event**, you should then implement advisability so that it uses _temperature_factor and _rain_factor.

You can then implement the **SimplePrediction** class as your own weather prediction model. The last stage of development should be to implement the **SophisticatedPrediction** class.

Submission

You must submit your assignment electronically through Blackboard. You need to submit both your **`prediction.py`** and **`event_decision.py`** files (use these names – all lower case). You may submit your assignment multiple times before the deadline – only the **last** submission before the deadline will be marked.

Late submissions of the assignment will **not** be accepted. Do not wait until the last minute to submit your assignment, as the time to upload it may make it late. In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension. See the course profile for details of how to apply for an extension:
https://www.courses.uq.edu.au/student_section_loader.php?section=5&profileId=98649

Requests for extensions **must** be made no later than 48 hours prior to the submission deadline. The expectation is that with less than 48 hours before an assignment is due it should be substantially completed and submittable. Applications for extension, and any supporting documentation (e.g. medical certificate), **must** be submitted via my.UQ. You **must** retain the original documentation for a minimum period of six months to provide as verification should you be requested to do so.

Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. apply basic object-oriented concepts such as classes, instances and methods,
3. read and analyse code written by others,
4. analyse a problem and design an algorithmic solution to the problem,
5. read and analyse a design and be able to translate the design into a working program,
6. apply techniques for testing and debugging

Criteria	Mark
Programming Constructs • Program is well structured and readable • Identifier names are meaningful and informative • Algorithmic logic is appropriate and uses appropriate constructs • Methods are well-designed, simple cohesive blocks of logic • Object usage demonstrates understanding of differences between classes and instances • Class implementation demonstrates understanding of encapsulation and inheritance	 1 1 1 1 1 1
Sub-Total	6
Functionality • Event implemented correctly. • EventDecision implemented correctly. • SimplePrediction implemented correctly. • SophisticatedPrediction implemented correctly.	 0.5 2 1.5 2.5
Sub-Total	6.5
Documentation • All modules, classes, methods and functions have informative docstring comments • Comments are clear and concise, without excessive or extraneous text • Significant blocks of program logic are clearly explained by comments	 1.5 0.5 0.5
Sub-Total	2.5
Total	**/ 15**

Your total mark will be constrained if your program does not implement key functionality. The following table indicates a multiplier that will be applied to your programming constructs and documentation marks based on how much of the functionality your program correctly implements. The table assumes you followed the advice provided in the hints section of this document.

Functionality Correctly Implemented	Multiplier
The program does not pass any tests	0%
Only Event passes tests	40%
EventDecision._rain_factor or EventDecision._temperature_factor passes tests	60%
All EventDecision methods pass tests	80%
SimplePrediction passes tests	100%

It is your responsibility to ensure that you have adequately tested your program to ensure that it is working correctly.

In addition to providing a working solution to the assignment problem, the assessment will involve discussing your code submission with a tutor. This discussion will take place in week 10, in the practical session to which you have signed up. You **must** attend your allocated practical session, swapping to another session is not possible.

In preparation for your discussion with a tutor you may wish to consider:

- any parts of the assignment that you found particularly difficult, and how you overcame them to arrive at a solution; or, if you did not overcome the difficulty, what you would like to ask the tutor about the problem;
- whether you considered any alternative ways of implementing a given function;
- where you have known errors in your code, their cause and possible solutions (if known).

It is important that you can explain to the tutor the details of, and rationale for, your implementation.

In the interview you must demonstrate understanding of your code. If you cannot demonstrate understanding of your code, this may affect the final mark you achieve for the assignment. A technically correct solution may not achieve a pass mark unless you can demonstrate that you understand its operation.

A partial solution will be marked. If your partial solution causes problems in the Python interpreter please comment out the code causing the issue and we will mark the working code. Python 3.7.2 will be used to test your program. If your program works correctly with another version of Python but does not work correctly with Python 3.7.2, you will lose **at least** all the marks for the functionality criteria.

Please read the section in the course profile about plagiarism. Submitted assignments will be electronically checked for potential plagiarism.

Detailed Marking Criteria

Criteria	Mark		
Programming Constructs	**1**	**0.5**	**0**
• Program is well structured and readable	Code structure highlights logical blocks and is easy to understand. Code does not employ global variables. Constants clarify code meaning.	Code structure corresponds to some logical intent and does not make the code too difficult to read. Code does not employ global variables.	Code structure makes the code difficult to read.
• Identifier names are meaningful and informative	All identifier names are informative, clearly describing their purpose, and aiding code readability.	Most identifier names are informative, aiding code readability to some extent.	Some identifier names are not informative, detracting from code readability.
• Algorithmic logic is appropriate and uses appropriate constructs	Algorithm design is simple, appropriate, and has no logical errors. Control structures are good choices to implement the expected logic.	Algorithm design is not too complex or has minor logical errors. Most control structures are good choices to implement the expected logic, but a few may be a little convoluted.	Algorithm design is overly complex or has significant errors. Some control structures are used in a convoluted manner (e.g. unnecessary nesting, multiple looping, …).
• Methods are well-designed, simple cohesive blocks of logic	Methods have a single logical purpose. Parameters and return values are used well and do not break class encapsulation.	Most methods have a single logical purpose. Parameters and return values are appropriate and rarely break class encapsulation.	Some methods perform multiple functional tasks or are overly complex. Parameters and return values are not used appropriately or break class encapsulation.
• Object usage demonstrates understanding of differences between classes and instances	Methods, attributes and comments indicate each class is treated as a definition of a type and objects are used well in implementation. System behaviour is implemented in terms of objects sending messages to each other.	Methods and attributes are based on provided design and objects are mostly used appropriately in implementation. Most behaviour is implemented in terms of objects sending messages to each other.	Classes are treated as modules with functions, not as self-contained entities. Method implementations depend on external logic or variables.
• Class implementation demonstrates understanding of encapsulation and inheritance	Provided class interfaces have not been modified, aside from adding new attributes and methods that complement the design. Attributes are only used inside of their defining class. Subclasses of WeatherPrediction conform to provided definition and are well designed, cohesive, specialised models.	Provided class interfaces have not been modified, aside from adding new attributes and methods. Attributes are only used inside of their defining class or subclasses. Subclasses of WeatherPrediction conform to provided definition.	Provided class interfaces have been modified in ways detrimental to the design. Attributes are accessed directly from outside of their defining class. Subclasses of WeatherPrediction do not conform to provided definition.

Documentation	**1.5**	**1**	**0.5**	**0**
• All modules, classes, methods and functions have informative docstring comments	All docstrings are accurate, complete & unambiguous descriptions of how item is to be used. All parameters and return types, and expected values, are described clearly.	Almost all docstrings are accurate and reasonably clear descriptions of how item is to be used. Almost all parameters and return types are described clearly.	Almost all docstrings are accurate descriptions of how item is to be used. Most parameters and return types are described clearly.	Some docstrings are inaccurate or unclear, or there are some items without docstrings. Some parameters and return types, or expected values, are not clear.
• Comments are clear and concise, without excessive or extraneous text			Most comments provide useful information that clarifies the intent of the code, making it easier to understand. Comments rarely repeat logic already clear in code.	Some comments are irrelevant, not providing any detail beyond what is obvious in the code. Comment style, or excessive length, obscures some program logic.
• Significant blocks of program logic are clearly explained by comments			Important or complex blocks of logic (e.g. significant loops or conditionals) are clearly explained and summarised (i.e. stating a loop iterates over a list is not a useful explanation or summary).	Some important or complex blocks of logic are explained poorly, or are not summarised. Some unimportant blocks of logic are described in too much detail.