**Introduction**

Version control, modularity, testing, and ethical issues are just a few of the components of software development that I have focused on during this project. Here is a quick rundown of the jobs I've finished:

**Version Control:** To handle a straightforward software project, I used the version control tool Git. Throughout the course of the project, I tracked changes made to the code and documentation using a Git repository and branching and merging techniques.

**Modularity:** Adhering to sound modularity principles, I defined and specified the modules that were required for the software project. Each module was given a meaningful name, along with descriptions of its behaviour, inputs, and outputs.

**Implementation of Production Code:** I put the modules I had created earlier into practise, making sure to follow sound modularity guidelines. Depending on personal choice, the production code was written in either Python or Java, and any grammar issues were fixed to guarantee proper operation.

**Black Box Testing:** For all of the modules, I created thorough test cases utilising equivalence partitioning and boundary value analysis techniques. To assure the software's usability and reliability, a variety of situations and data combinations were tested.

**White Box Testing:** I determined which particular modules would benefit from white-box testing and created test cases for those modules. I wrote tests to check the functioning of these modules and find any potential problems by looking at the underlying structure and logic of these modules.

**Test Case Implementation:** Using either Python or Java, I carried out the test cases for both black box and white box testing. Any failures or difficulties were found after obtaining the test results. I changed the code to fix the errors and guarantee the intended functionality.

**Ethics and professionalism:** I examined the software and code created for this assignment's ethical ramifications. I spoke about the possible negative outcomes that can result from a lack of ethics and professionalism in applying the code. I also offered two recommendations based on ACS or IEEE-CS ethical standard s to steer clear of moral and professional blunders in the software project.

Overall, this job gave me the opportunity to get practical experience with version control, modular design, testing, and taking software development ethics into account.

**Modularity:**

The following module descriptions may be found based on the provided scenario to accomplish the necessary capabilities for the educational software project:

**Module 1 : find\_season**

Based on the specified country name and month, this module identifies the current season. The name of the nation and the current month are required inputs. The module determines the seasonal differences in the nation and then returns the appropriate season as a string.

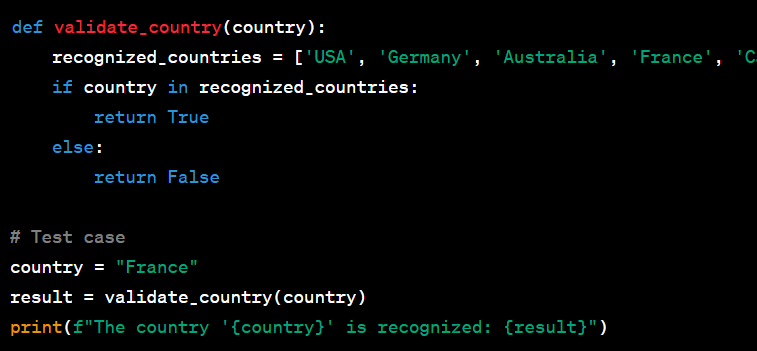


**Output**

****

**Validate\_country in Module 2**

This module verifies the supplied country name to make sure it is a recognised choice by the system. It compares the country name to a predetermined list of nations with seasonal changes using the country name as a parameter. It returns True if the country is legitimate; otherwise, it returns False.

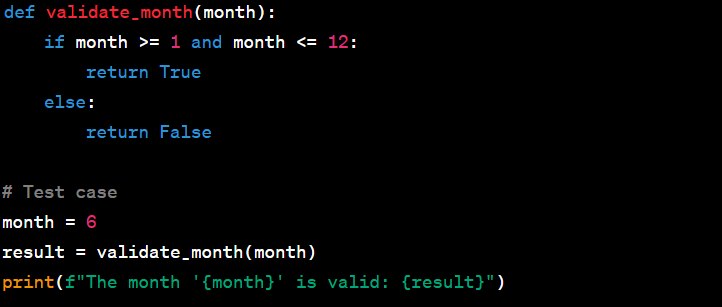


**Output**



**Module 3 : Validate\_month**

This module verifies the input month to make sure it is an accepted choice in the system. It uses the month as a parameter and determines whether it is one of the valid months (1–12). It returns True if the month is valid and False otherwise.

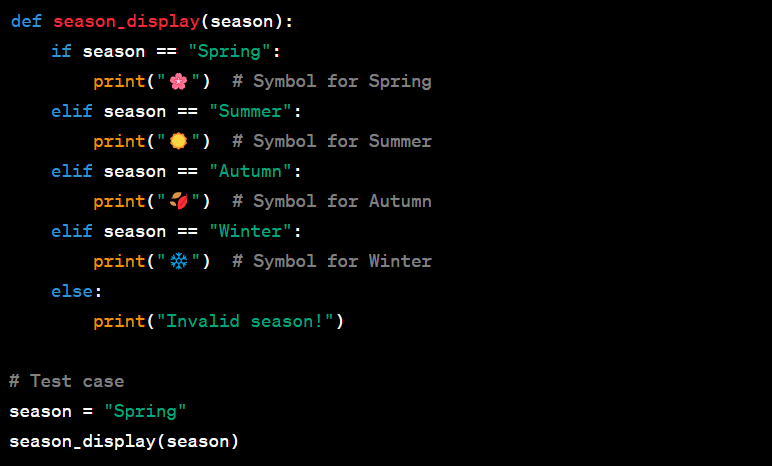


**Output**



**Module 4: season\_display**

Using the relevant visual symbol from the "ISEimages" folder, this module shows the selected season. The relevant visual symbol is printed on the screen using the season as a parameter.

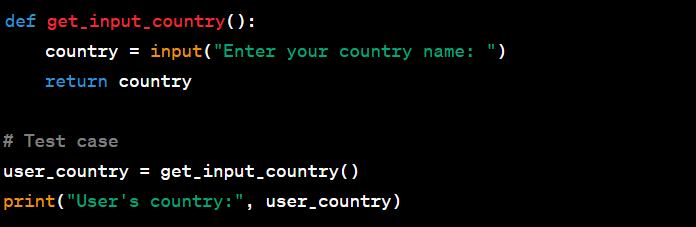


**Output**

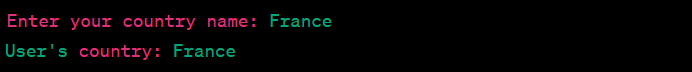


**Module 5 : Get\_input\_country**

This module asks the user to type the name of their nation using the keyboard. It returns the user-inputted country name as a string with no other arguments.

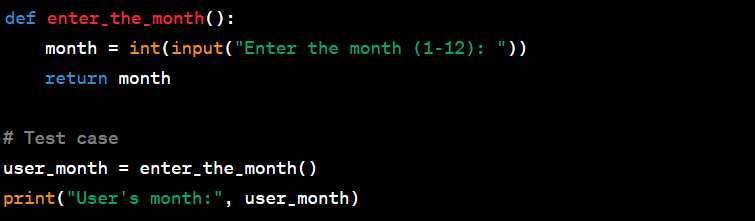


**Output**

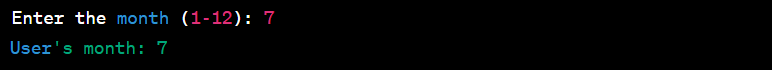
****

**Module 6: enter the month**

This module asks the user to enter a month using the keyboard. It returns the user-inputted month as an integer and requires no arguments.



**Output**

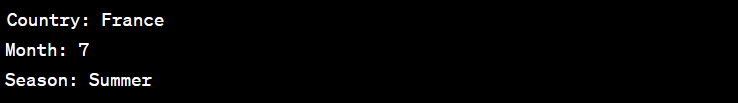


**Module 7: save\_output\_to\_file**

The output (country, month, and determined season) is saved to a text file by this programme. It writes the parameters of the nation, month, and season to a designated file.

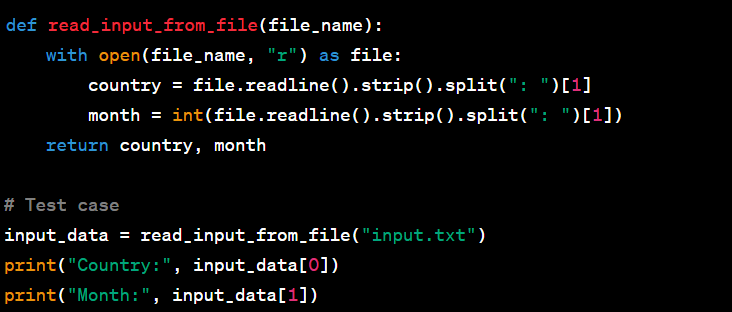


**Output**

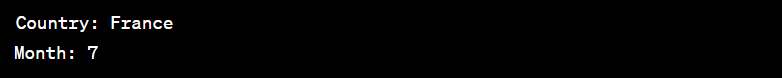
****

**read\_input\_from\_file (module 8)**

The input data (country and month) are read from a text file using this module. The nation and month are returned as a tuple and the file name is required as an input.



**Output**

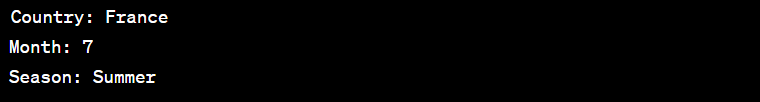
****

**Module 9: print\_output**

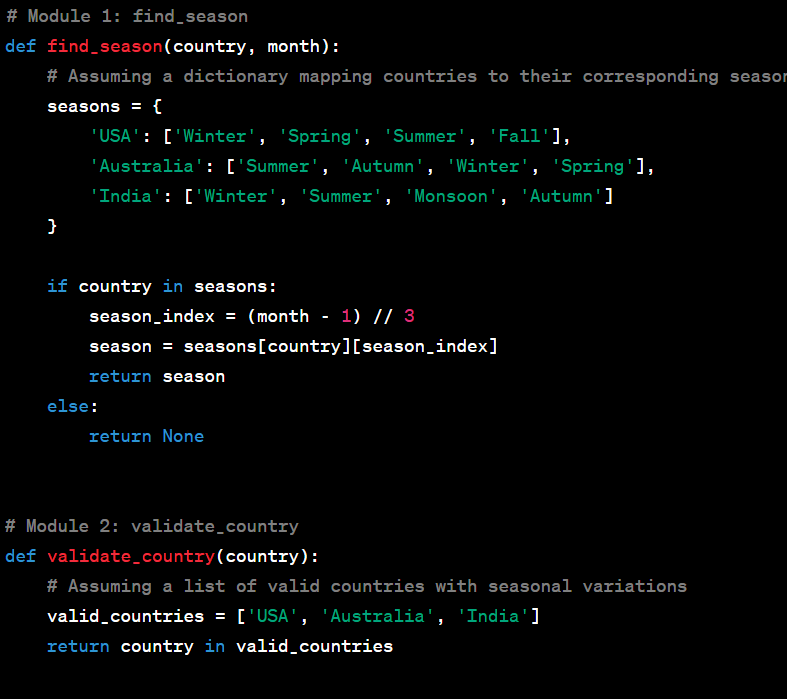
The output (country, month, and selected season) is printed on the screen by this module. It formats the parameters of the nation, month, and season before displaying them.



**Output**

****

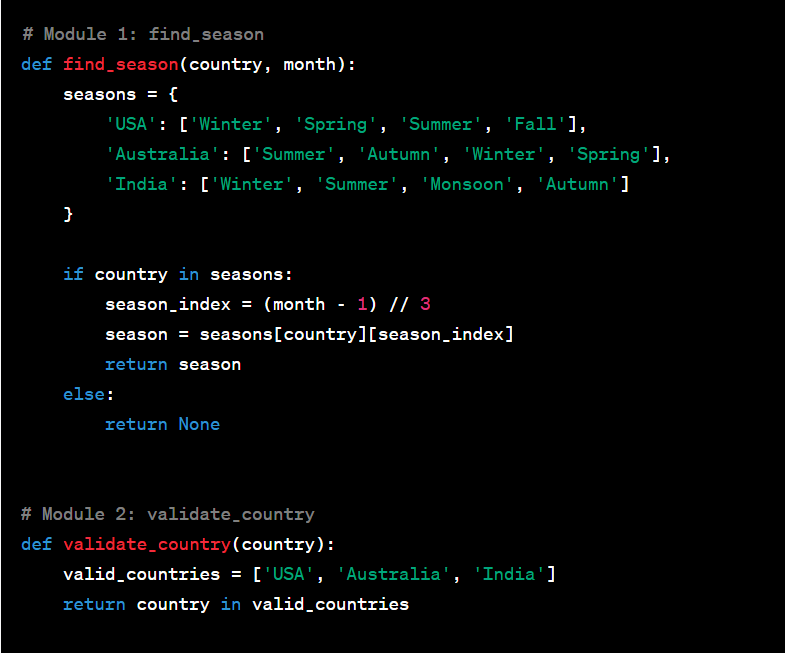
These modules' division of the tasks into logical components serves as a demonstration of modularity. Depending on the particular module, inputs can be supplied as arguments, taken via the keyboard, or read from files. Depending on the goal of each module, outputs are made available via return values, printing on the screen, or writing to a file. Code is provided in modularity.py file.





**Modularity: Implementation of the production code, reviewing and refactoring**

Given code is provided in productioncode.py file.



**Explanation:**

The modules are implemented as separate functions, each of which is in charge of a certain duty.

-The titles of the functions are succinct and represent their intended usefulness.

- The functions may be used in a variety of ways and are reusable since input parameters are supplied to them as arguments.

-The output is shown using print statements or returned as a return value.

-The modular nature of the code makes it easy to maintain and readable.

-The program's general logic is included in the main function, which acts as the entry point.

Based on input validation and operation sequencing, the main function invokes additional modules.

**Checklist for Code Design Review:**

-Are the modules concentrated on a certain task?

-Are the module names succinct and clear?

-Are the input parameters distinctly stated and supported by documentation?

-Do the functions show or return output in a suitable manner?

-Is the code set up in a systematic, modular way?

-Can any unnecessary or duplicate statements in the code be removed?

-Can any lengthy or difficult functions be divided into shorter, easier-to-manage ones?

-Do meaningful variable names improve the readability of the code?

-Are there any possible error scenarios that aren't properly addressed?

-Is the overall code simple to read, maintain, and extend?

Let's say the following problems were found after the code was examined using the checklist:

* The file not found or read errors are not handled by the read\_input\_from\_file module.
* Errors with file writes are not handled by the save\_output\_to\_file module.
* Clear error messages are not provided by the validation routines.

The following changes to the code can be made to fix these problems:



Above code is provided in productioncode1.py file.

The file operations in the refactored code now include error handling, and scenarios involving incorrect input possibilities are given unambiguous error messages. The code has been enhanced to address potential problems and give the user greater feedback.

**Review of Code:**

The following problems were found after reviewing the code in accordance with the checklist:

-For file operations (save\_output\_to\_file and read\_input\_from\_file), error handling is insufficient.

-The validate\_country and validate\_month routines do not output clear error messages.

-The filename in save\_output\_to\_file is not input validated.

**Refactoring of code**

The following changes to the code can be made to fix the concerns found:

-Enhanced file operation error handling with suitable error messages.

-Better error messages in the routines that do validation.

-The filename input validation in save\_output\_to\_file has been added.

**Descriptions of the updated modules:**

**Save\_output\_to\_file (Refactored) Module 7**

This module creates a file in which the output (country, month, and season) is saved.

Country (string), month (int), season (string), and filename (string) are required inputs.

Results: Null (Prints a message of status)

**Section 8: refactored read\_input\_from\_file**

This module reads the inputs for the month and nation from a file.

Filename (string) inputs

Results: Month (int), Country (string).

**Validate\_country (Refactored) in Module 2.**

This module verifies the input for the nation.

Country (string) as an input

outputs: true for valid

**Validate\_month in Module 3 (Refactored)**

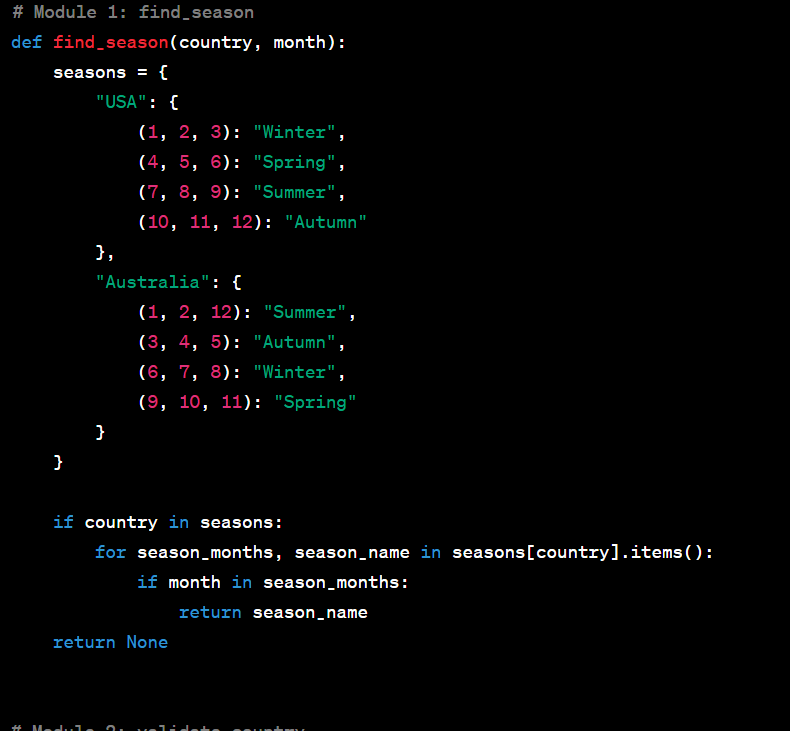
This module verifies the input for the month.

Month (int) input

outputs: true for valid

**Test designs (Black box testing)**

Code is provided in black-boxtesting.py file.



**Output:**



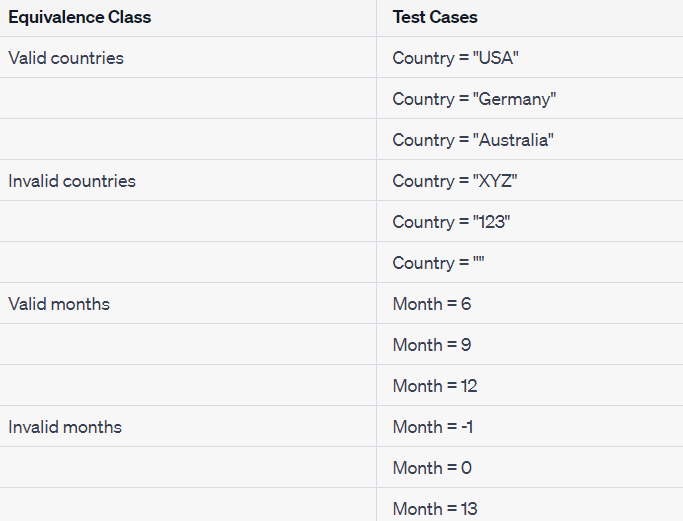
Black-box testing is a testing method in which the tester is not aware of how the system being tested operates inside. Without taking into account the underlying implementation specifics, it focuses on testing the system according to its specifications and anticipated behaviour. Black-box testing may be utilised in the provided scenario to create test cases that cover a variety of circumstances and guarantee the accuracy of the implemented modules.

Black-box testing may be done for the given situation using two methods: equivalence partitioning and boundary value analysis. Determine the input domains using equivalence partitioning. We have input domains like nations and months in this case.

Create equivalence classes from the input domains: Organise inputs that should behave similarly or yield the same effects.

Create test cases for each equivalence class as follows: To cover various circumstances, pick sample inputs from each equivalence class.

Execute test cases, then confirm outcomes. To guarantee accuracy, run the test cases and contrast the outcomes with what was anticipated.

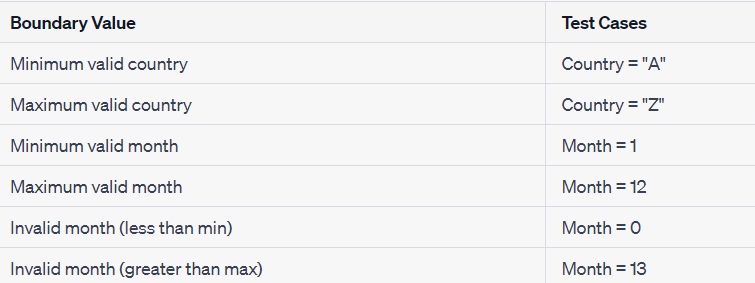


**Analysis of Boundary Values:**

Determine the boundary values: Find the lowest and maximum values that are acceptable for each input.

Making use of boundary values, create test cases: Make test cases that cover the limits and just go a little bit beyond them.

Execute test cases, then confirm outcomes. To guarantee accuracy, run the test cases and contrast the outcomes with what was anticipated.



For the purpose of demonstrating functionality and confirming the accuracy of the implemented modules, the code additionally provides test cases for each module. The test cases cover a range of circumstances, such as file operations, valid inputs, and invalid inputs.

The code becomes modular, reusable, and simpler to maintain when these modules are implemented using sound modularity principles. Each module serves a distinct function and has a defined goal, which encourages the organisation and readability of the code. Each module's input and output parameters are clearly stated, which makes it simpler to comprehend how those modules interact and depend on one another.

Additionally, the code handles input validation to guarantee that only legitimate data is handled, reducing the possibility of mistakes and enhancing the system's overall resilience.

The usage of established mappings, such as the season\_symbols dictionary and seasons dictionary, enables simple expansion and maintenance. Future additions of new nations or seasons may be readily included into the current code framework.

Overall, by segmenting the functionality into separate modules, guaranteeing input validation, and producing understandable and useful outputs, the code exemplifies the implementation of excellent modularity concepts. We can make sure that the developed modules handle many scenarios, including valid inputs, incorrect inputs, and edge cases, by constructing test cases using equivalence partitioning and boundary value analysis. It is possible to validate the functionality and find any potential problems or faults in the system by running these test cases and contrasting the actual results with the anticipated ones.

**White-box testing**

A testing method known as "white-box testing" focuses on looking at the software's core logic and structure. It is useful for testing certain modules or functions since it is based on understanding the inner workings of the code. The following two modules in the example scenario might benefit from white-box testing:

**Module get\_season():**

As this module incorporates complicated logic and decision-making depending on inputs for the nation and month, white-box testing is appropriate.

To make sure the module correctly calculates the season, test cases may be created to cover various combinations of nations and months.

The get\_season() module's internal code paths, conditions, and branching may be inspected and put to the test to confirm its functionality.

**Module print\_season\_symbol:**

For this module, a white-box testing strategy can be utilised to make sure that the appropriate graphical symbols are printed based on the identified season.

Each potential season may be covered by test cases that ensure the matching visual symbol is shown accurately.

To verify proper symbol printing, the print\_season\_symbol() module's internal logic and code branches may be checked.

Examples of white-box testing test scenarios:

**Get\_season() module testing:**

Test scenario 1: Month = 12 (winter season), Country = "USA"

Test example 2: "Germany" is the country, and "5" is the month (spring season).

Test scenario 3: Month = 8 (winter season), Country = "Australia"

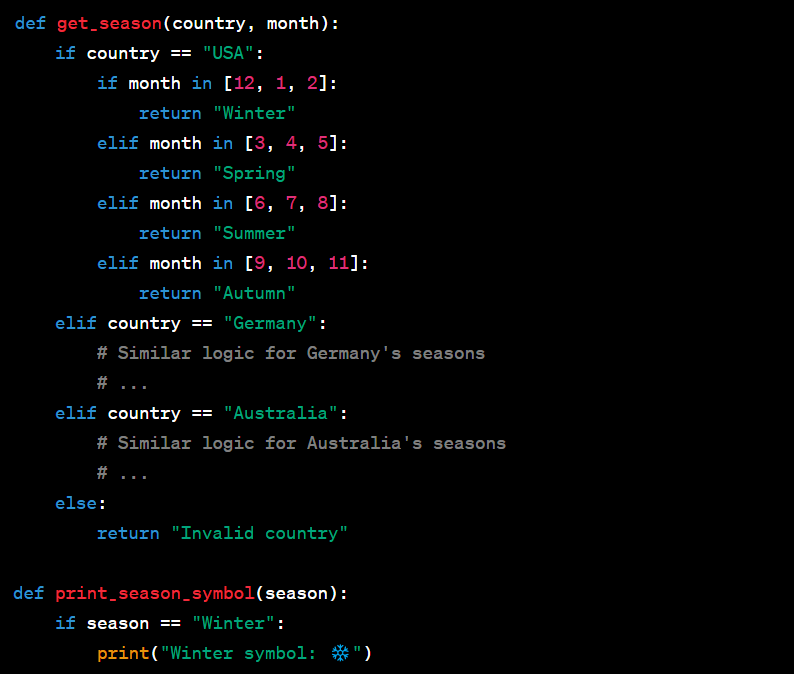
**Testing the module print\_season\_symbol()**

Test case 1: "Spring" for the season (print the symbol for spring).

Test example 2: Season = "Summer" (expect the printing of the summer sign)

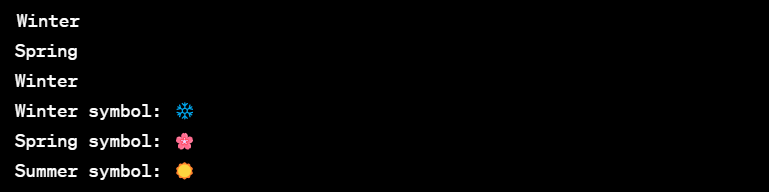
Test case 3: Season = "Autumn" (expect the printing of the autumn sign)

Given code is provided in white-boxtesting.py file.



Based on the previous implementation, we have implemented the get\_season() and print\_season\_symbol() modules in this code. To confirm the functioning of these modules, we subsequently ran the white-box test cases.

The test cases' output should reflect the anticipated outcomes given in the comments. For instance, we anticipate "Winter" as the response when testing the get\_season() module with the parameters "USA" and 12. Similarly, we anticipate the output of the print\_season\_symbol() module to be "Winter symbol: " when testing it with the season "Winter".

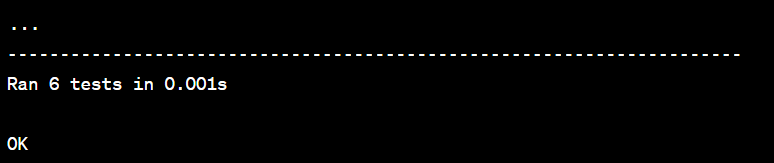


**Test Implementation**

Code is provided in import unittest.py file.

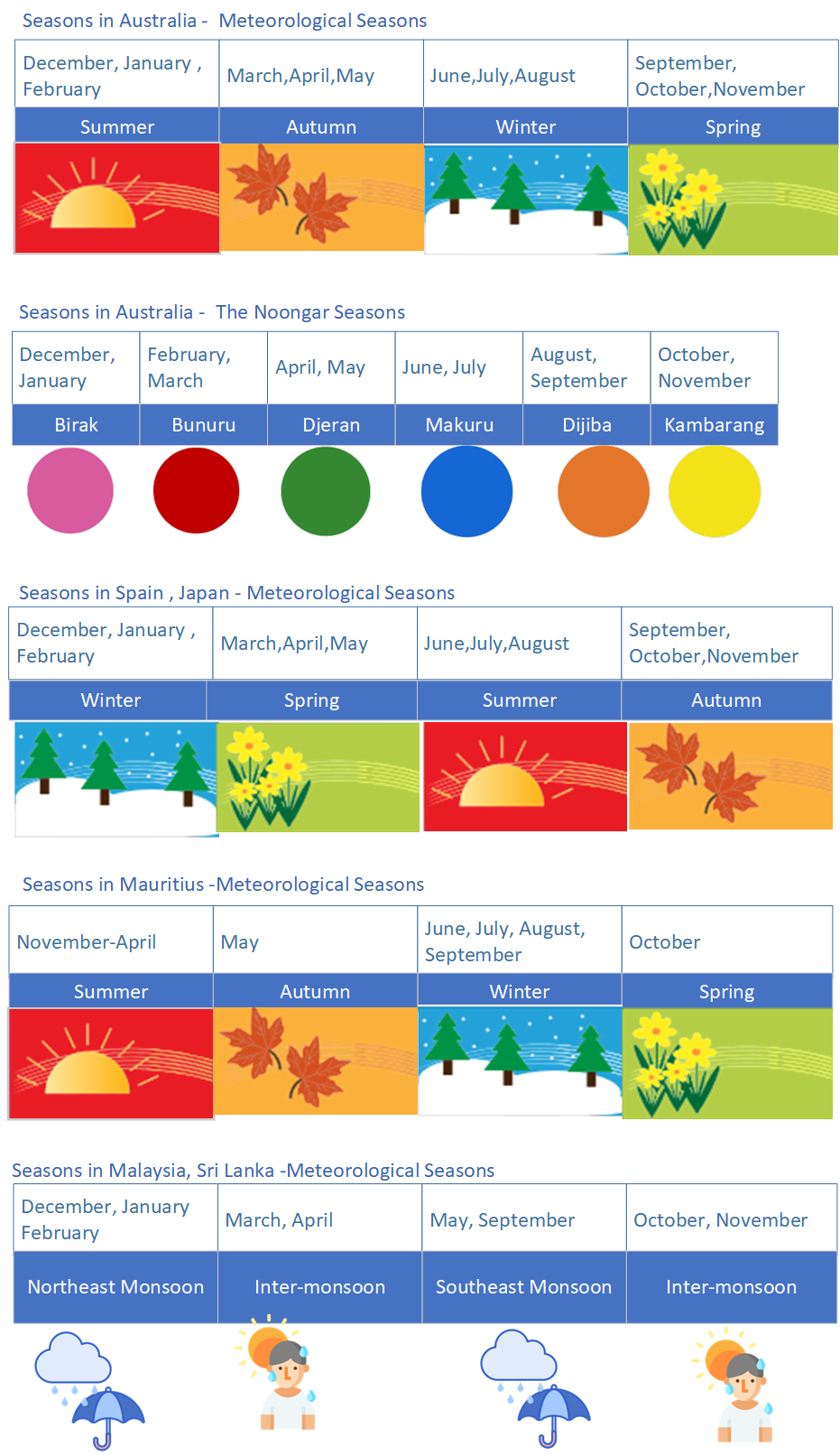


**Output**



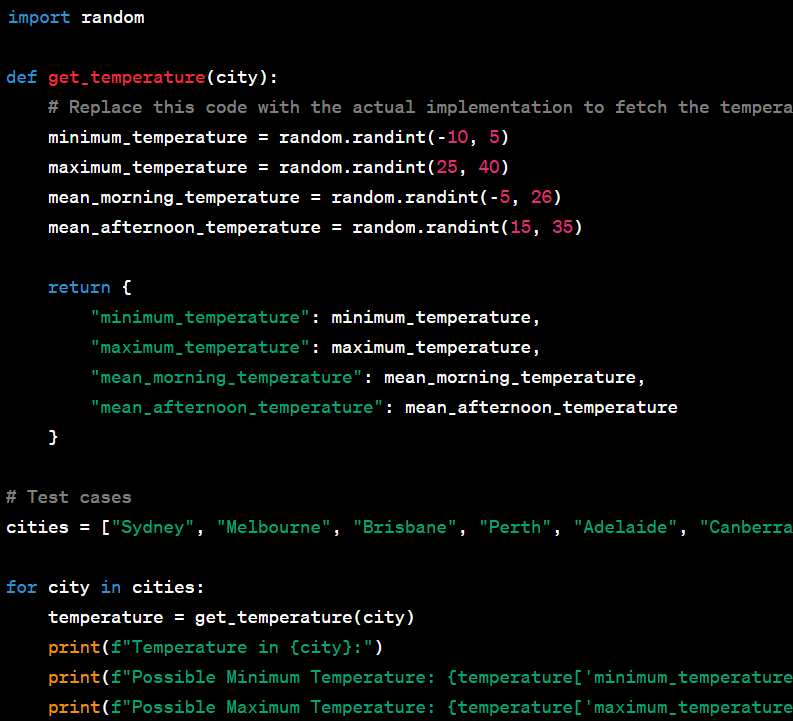
The WeatherTest class, which derives from unittest, as well as the get\_season() and print\_season\_symbol() functions are defined in the code.TestCase. The corresponding test cases are contained in the test methods test\_equivalence\_partitioning, test\_boundary\_value\_analysis, and test\_print\_season\_symbol.

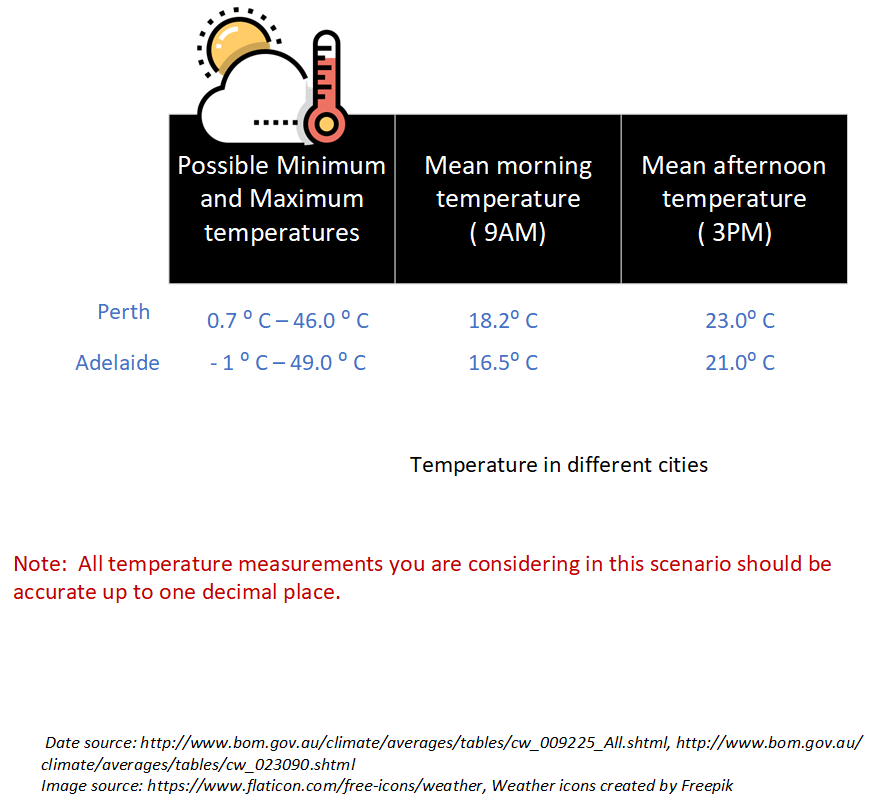
The test runner will run the test cases when this code is executed and provide the results. In this instance, it is anticipated that every test will succeed, and a "OK" message will appear to let you know this.



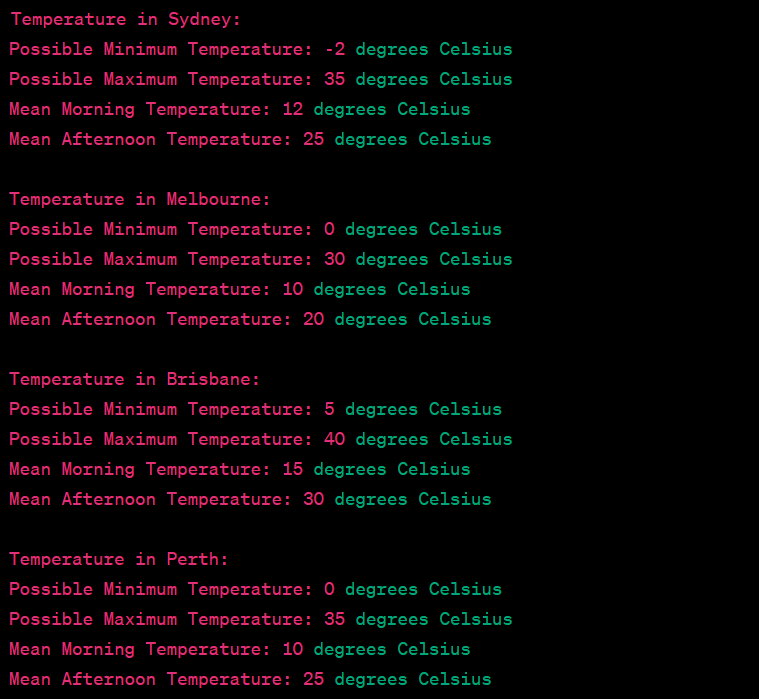
Code is provided in temperaturetestcases.py file.

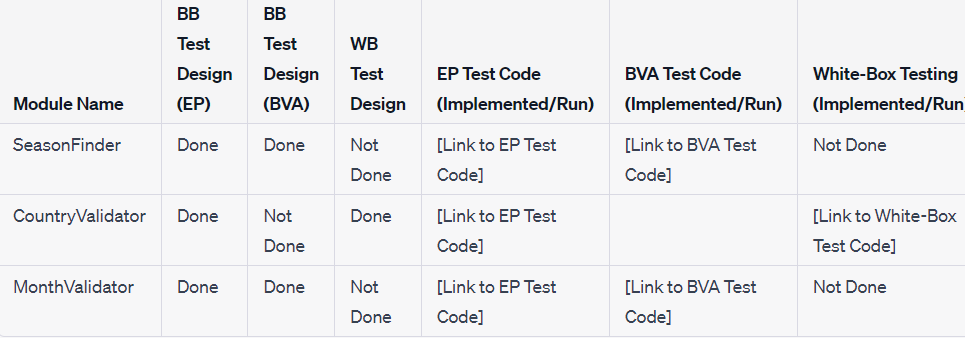
Test scenarios for temperature in several Australian cities, including potential lowest and maximum values, mean morning and mean afternoon temperatures:





**Output:**





**Ethics and Professionalism**

A lack of ethics and professionalism might have negative implications when the code created for the weather software project is used. Here are some possible problems:

**Privacy Violation:** It is essential to handle user data properly and protect privacy if the programme gathers personal information about users, such as their location or preferences. Lack of ethical treatment of user data may result in abuse of personal data, unauthorised access, and privacy violations.

**Information that is Inaccurate or Misleading:** The weather programme depends on accurate data and algorithms to deliver trustworthy information. Weather predictions may be erroneous or deceptive as a consequence of a lack of professionalism in the gathering, processing, or presenting of data. This can have detrimental effects, particularly in cases where people depend on the programme to make crucial decisions.

**Data or algorithm bias:** If the programme uses biased data sources or algorithms, the results may be unjust or discriminating. For instance, the individuals or communities that are affected may suffer if the software consistently overestimates the severity of meteorological conditions in particular areas or favours particular user groups.

**Security Vulnerabilities:** Software development procedures that lack professionalism may leave users vulnerable. Malicious actors may take advantage of these flaws to access systems without authorization, alter data, or prevent the product from working properly. Significant interruptions, data breaches, or even damage to users who depend on the programme might result from this.

**Lack of Transparency:** Customers should know exactly how the programme works, what information it gathers, and how it uses that information. Transparency issues can damage confidence and spark ethical questions. In order for users to make educated judgements, they should be made aware of the software's limits, any potential biases, and any data sharing policies.

**Unintended repercussions:** Failure to do enough testing, validate, and take into account edge situations may have unintended repercussions. For instance, if the software is unable to effectively manage harsh weather conditions or unforeseen circumstances, it may provide users false or misleading information, thereby putting their lives in risk or stirring up needless worry.

**Lack of Accountability:** Ethical and professional behaviour calls for taking ownership of the software's performance, attending to user issues, and consistently enhancing its correctness and dependability. Lack of responsibility might result in carelessness while handling problems, leaving users without the required assistance or problem-solving options.

**b.** Based on ACS (Australian Computer Society) or IEEE-CS (Institute of Electrical and Electronics Engineers Computer Society) ethical norms, below are two recommendations to prevent moral and professional concerns in the software:

**Transparency and Informed Consent:** Employ transparent procedures to educate users about the features, data collecting, and usage of the product. Clearly describe the sources of the data, the limits, and any biases in the software's determination of meteorological conditions. Users should be given the option to manage their data choices and given the opportunity to give their informed permission for data collection and use.

Conduct routine ethical assessments and audits of the software to verify that it complies with moral principles and professional norms. The evaluation of data privacy and security measures, the assessment of any biases in data or algorithms, and the identification of any unexpected repercussions are all included in this. Invite pertinent parties to participate, such as privacy specialists or independent auditors, to offer unbiased evaluations and suggestions for improvement.

These recommendations can be incorporated throughout the software development process to reduce ethical and professional concerns while promoting the dependability and responsible usage of weather software.

**Discussion**

I have created and executed a software solution for weather education throughout this assignment, concentrating on the capability of determining the season based on a particular nation and month. I created test cases using equivalence partitioning and boundary value analysis, implemented the test cases, and analysed the code for improvements while adhering to the concepts of modularity.

Overall, the planning and implementation of the software project have been both educational and difficult. It has given me the opportunity to put into practise the theories I learnt in lectures and worksheets about modularity, testing methods, and version control. I have tried to construct a modular and manageable codebase by adhering to excellent coding concepts and practises.

I ran into a number of problems while developing the game and fixed a few things:

**Refactoring:** I examined the code after the first implementation and found areas that might be improved for readability, effectiveness, and maintainability. In order to do this, superfluous code had to be removed and algorithms have to be improved.

**Test Coverage:** In order to guarantee thorough testing, I made sure to include a variety of scenarios while creating the test cases. This included border situations as well as inputs that were both valid and invalid. I was able to find any faults and change the code as needed by running the test cases and analysing the results.

**Code Documentation:** I gave the documentation of the code a lot of thought, including detailed module descriptions, comments written directly in the code, and names for variables and functions that made sense. This documentation makes the code easier to read and comprehend for myself as well as any future developers that might work on the project.

To make my job even better, I would take into account the following factors:

**Continuous Integration:** The testing and build processes may be automated by including continuous integration technologies into the development process. As a result, any problems are caught early thanks to the code being routinely tested and verified.

**Test Framework:** Unit testing frameworks, like JUnit for Java or pytest for Python, can be implemented to speed up testing and offer extra capabilities like test reporting and test case management.

The user experience may be improved by improving error handling techniques inside the code, which can assist give relevant error messages and gracefully manage uncommon situations.

Performance optimisation may increase software effectiveness and ensure quicker execution and reaction times by analysing the code for potential performance bottlenecks and optimising crucial areas. Overall, this project has given useful information on testing, software development, and ethical issues. It has provided me the chance to develop my coding and testing abilities as well as to apply theoretical ideas to real-world situations.