**Introduction**

I used two scenarios in the provided code: comparing temperature readings for a specific city and time to determine the season of the year based on the country and month. The season information for various countries and months is stored in a nested dictionary called seasons by the find\_season function for scenario A. It returns the appropriate season if the provided country and month are found in the dictionary. It returns a message indicating that season information is unavailable if the country or month is not found.

The compare\_temperature function for scenario B stores temperature data for various cities and times using a temperature nested dictionary. It checks assuming that the gave city and time exist in the word reference, computes the contrast between the given temperature and the typical temperature for that city and time, and returns a message showing whether the temperature is above, underneath, or near the normal temperature. It returns a message indicating that temperature data is unavailable if the city or time are not found. I also included test cases to show how these functions are used. The normal results for each experiment are given as remarks.

**Functionality implemented**

There are two essential features in the code that has been implemented. First, it makes it possible to determine the time of year based on the country and month. The find\_season() function efficiently retrieves the relevant season information by employing a nested dictionary structure. It returns the appropriate season after determining whether the provided country and month are found in the dictionary. It gracefully responds by returning a message indicating that season information is unavailable if the country or month is not found. Second, the code lets you compare the temperature readings to the average for a particular time and city. The temperature data for various cities and times is stored in a nested dictionary by the compare\_temperature() function. Based on the difference between the given temperature and the average temperature, it determines whether the temperature is above, below, or close to the average and sends a message. Similar to the first feature, it handles situations in which the provided city or time cannot be found in the dictionary and displays an appropriate message to indicate that temperature data is unavailable. Overall, these implemented functionalities offer useful applications in weather analysis, travel planning, and beyond for assessing temperature deviations for various cities and times and determining seasons based on geographical location.

**Module descriptions**

**Original Module Descriptions:**

1. **find\_season(country, month):**

* The corresponding season for each country and month is returned by this module, which takes as inputs a country and a month.
* Assumption: The nested dictionary's seasons are accurate and cover every country and month combination imaginable.

1. **compare\_temperature(city, temperature, time):**

* This module contrasts the given temperature perusing and the typical temperature for a particular city and time and returns a message demonstrating whether it is above, underneath, or near the normal temperature.
* Assumption: The nested dictionary's temperature readings are precise and cover all possible combinations of cities and times.

**Revised Module Descriptions (after refactoring):**

1. **find\_season(country, month):**

* The corresponding season for each country and month is returned by this module, which takes as inputs a country and a month.
* Assumption: The nested dictionary's seasons are accurate and cover every country and month combination imaginable.
* Revised: After refactoring, this module remained unchanged.

1. **compare\_temperature(city, temperature, time):**

* This module contrasts the given temperature perusing and the typical temperature for a particular city and time and returns a message demonstrating whether it is above, underneath, or near the normal temperature.
* Assumption: The nested dictionary's temperature readings are precise and cover all possible combinations of cities and times.
* Revised: After refactoring, this module remained unchanged.

**Explanation:**

I decided to execute these modules to address two explicit situations: comparing temperature readings for cities and times to determine the season based on country and month. These situations are normal and can be valuable in different applications, for example, weather conditions determining, travel arranging, or information examination.

The find\_season module makes it easy to find out the season for a specific country and month. It makes it possible to update or add country-season mappings in a flexible way thanks to the use of a nested dictionary.

The compare\_temperature module helps in contrasting a temperature perusing and the typical temperature for a particular city and time. Assessing the weather and identifying anomalies can be made easier with this.

In both instances, nested dictionaries are used by the implementation to efficiently store and retrieve relevant data. When the provided country, month, city, or time is not found in the dictionaries, the modules are designed to handle the situation by returning the appropriate message to indicate the lack of information.

**Modularity**

You can execute the production code by following these steps:

* Copy the code into a Python file with a .py extension, for example, seasons\_and\_temperatures.py.
* Save the document.
* Open a terminal or order brief.
* Go to the directory that contains the Python file.
* Use the command to run the file: seasons\_and\_temperatures.py in Python

**The code shows a few particularity ideas:**

* Encapsulation of functions: Find\_season (country, month) and compare\_temperature (city, temperature, time) are the two distinct functions in the code. Each capability plays out a particular errand and typifies the rationale connected with that undertaking. Organization, reusability, and maintainability of code are all aided by this.
* Separation of data: Seasonal and temperature-related data are stored in their own respective dictionaries. This separation makes it easier to manage and modify the data without affecting the functions' logic. Additionally, it makes it possible to include additional countries, months, cities, and temperature data without having to alter the function code.
* Parameterization: To allow for a wide range of input values, the functions accept parameters like country, month, city, temperature, and time. This makes the code more adaptable and versatile by allowing functions to be reused with various inputs.
* Problem solving: Error handling is built into the code in the event that the provided country, month, city, or time cannot be found in the appropriate dictionaries. It provides the function caller or user with helpful error messages to follow.

**Review Checklist:**

* Readability of code: The code is meaningful and all around organized, with suitable space and clear factor and capability names. It adheres to PEP 8 code style guidelines.
* Plan of operation: The capabilities have a reasonable reason and perform explicit undertakings. They return meaningful results by taking parameters as inputs.
* Separation of data: The information connected with seasons and temperatures is put away independently in word references. This detachment considers simple alteration and expansion of information without changing the capability code.
* Problem solving: Error handling is included in the code in the event that the specified country, month, city, or time is not found. It provides the function caller or user with helpful error messages to follow.
* Validation of inputs: Before accessing the data, the code verifies whether the provided country, month, city, or time is present in the appropriate dictionaries. As a result, valid inputs are used and potential errors are avoided.

**The review's findings:**

* **Readability of code:** The code adheres to Python code style guidelines and is well-structured. The names of variables and functions are clear and descriptive, which makes them easier to read.
* **Plan of operation:** The functions are well-executed and serve a distinct purpose. They take the fundamental boundaries and return significant outcomes.
* **Separation of data:** Dictionaries are used to appropriately divide the temperature and season data. Without modifying the function code, this makes it simple to modify and add data.
* **Problem solving:** The code handles situations where the gave country, month, city, or time are not found. It provides helpful error messages to show that the requested data is unavailable.
* **Validation of inputs:** The code approves the gave inputs by checking assuming they exist in the separate word references prior to getting to the information. As a result, potential errors are prevented and valid inputs are utilized.

**Black-Box Test Cases:**

**Test case for scenario A:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Country** | **Month** | **Expected Output** | **Assumptions** |
| Test Case 1 | Australia | March | Autumn | The country and month exist in the seasons dictionary. |
| Test Case 2 | Japan | September | Autumn | The country and month exist in the seasons dictionary. |
| Test Case 3 | Canada | December | Season information not available for the given country and month. | The country or month does not exist in the seasons dictionary. |

**Test case for scenario B:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case** | **City** | **Temperature** | **Time** | **Expected Output** | **Assumptions** |
| Test Case 1 | Australia | 18 | Morning | The temperature is close to the average temperature. | The city and time exist in the temperature dictionary. The temperature is close to the average temperature. |
| Test Case 2 | Japan | 10 | Evening | The temperature is 6⁰C below the average temperature. | The city and time exist in the temperature dictionary. The temperature is below the average temperature. |
| Test Case 3 | Paris | 15 | Morning | Temperature information not available for the given city and time. | The city or time does not exist in the temperature dictionary. |

**Made assumptions:**

* For Situation A, it is expected that the season data for a given nation and month is accessible in the seasons word reference.
* It is assumed that the temperatures dictionary contains the average temperature data for a given city and time for Scenario B.

**Explanation:**

* To ensure that the code works correctly in a variety of situations, the test cases cover various input combinations.
* Valid inputs that should produce the expected output are included in test cases, as are invalid inputs that should produce the appropriate error messages.
* The suspicions made depend on the comprehension of the code and the gave information structures. To ensure thorough testing, the test cases are designed to cover both known and unknown input combinations.
* The logic that is implemented in the code and the predefined data in the dictionaries are what are used to calculate the anticipated outputs. They represent the outcomes and desired behavior for each test case.
* By creating these test cases, we can determine whether the code correctly handles various scenarios and produces the expected outputs. It guarantees that the code capabilities as expected and can deal with substantial contributions, as well as nimbly handle cases with absent or wrong information sources.

**White-Box Test Cases:**

**Test case for scenario A:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Country** | **Month** | **Expected Output** | **Assumptions** |
| Test Case 1 | Australia | March | Autumn | The country and month exist in the seasons dictionary. |
| Test Case 2 | Japan | September | Autumn | The country and month exist in the seasons dictionary. |
| Test Case 3 | Canada | December | Season information not available for the given country and month. | The country or month does not exist in the seasons dictionary. |
| Test Case 4 | Australia | February | Summer | The country and month exist in the seasons dictionary. |
| Test Case 5 | Japan | June | Summer | The country and month exist in the seasons dictionary. |
| Test Case 6 | Spain | May | Spring | The country and month exist in the seasons dictionary. |

**Test case for scenario B:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case** | **City** | **Temperature** | **Time** | **Expected Output** | **Assumptions** |
| Test Case 1 | Australia | 18 | Morning | The temperature is close to the average temperature. | The city and time exist in the temperatures dictionary. The temperature is close to the average temperature. |
| Test Case 2 | Japan | 10 | Evening | The temperature is 6⁰C below the average temperature. | The city and time exist in the temperatures dictionary. The temperature is below the average temperature. |
| Test Case 3 | Paris | 15 | Morning | Temperature information not available for the given city and time. | The city or time does not exist in the temperatures dictionary. |
| Test Case 4 | Australia | 20 | Evening | The temperature is close to the average temperature. | The city and time exist in the temperatures dictionary. The temperature is close to the average temperature. |
| Test Case 5 | Japan | 16 | Morning | The temperature is close to the average temperature. | The city and time exist in the temperatures dictionary. The temperature is close to the average temperature. |
| Test Case 6 | Spain | 18 | Evening | The temperature is close to the average temperature. | The city and time exist in the temperatures dictionary. The temperature is close to the average temperature. |

**Explanation:**

* The understanding of the code implementation and logic serves as the foundation for the design of the white-box test cases.
* To guarantee complete code coverage, the test cases cover various code paths, branches, and conditions.
* The experiments incorporate both legitimate data sources and limit cases to evaluate the code's conduct in various situations.
* Presumptions made depend on the comprehension of the code and the gave information structures. To ensure thorough testing, the test cases are designed to cover both known and unknown input combinations.
* The code logic and the predefined data in the dictionaries are used to calculate the anticipated outputs. They represent the outcomes and desired behavior for each test case.
* We can check that the code is correct and that all code paths and conditions are handled appropriately by creating these white-box test cases. It improves the code's reliability and robustness by assisting in the identification of any potential issues or errors in its implementation.

**Test Implementation and Execution:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Module Name** | **BB Test Design (EP)** | **BB Test Design (BVA)** | **WB Test Design** | **EP Test Code** | **BVA Test Code** | **White-Box Testing** |
| find\_season | Done | Done | Done | Not Done | Not Done | Done |
| compare\_temperature | Done | Done | Done | Not Done | Not Done | Done |

We can make sure that our code works as intended, handles various inputs correctly, and complies with the defined requirements by implementing and running both black-box and white-box tests. We will be able to find and fix any flaws or problems using the combination of these testing methods, which will ultimately improve our project's quality and dependability.

**Version Control**

All of my code files and documents are in koirala\_bimal21181287\_ISErepo. The link is attached hereunder.   
  
Link: <https://github.com/koiralabimal/koirala_bimal21181287_ISErepo.git>

**Ethics:**

The code planned and carried out in this task is utilized in an enormous programming project that gives climate data and season recognizable proof for different nations. To find out about the weather and the season of the year, users can enter their location and time.

**a. The following negative effects can result from a lack of professionalism and ethics:**

* Protection and Information Security: User information such as location, time, and personal data may be compromised if the code does not have adequate privacy and data security measures. Unauthorized access to sensitive data, privacy breaches, and the potential misuse of personal information are all possible outcomes of this.
* Inaccurate Information About the Weather: Users may receive inaccurate weather information if the code is not properly tested or maintained. Depending on erroneous weather conditions figures can have serious outcomes, particularly for basic exercises like flight, open air occasions, or debacle the executives, prompting potential wellbeing perils.
* Unfair or One-sided Results: In the event that the code contains predispositions or oppressive calculations, it might bring about uncalled for treatment of specific gatherings or districts. For instance, resources or opportunities may not be distributed equally if the season identification algorithm is biased toward particular geographical locations.
* Dependability and Accessibility: The weather information service's availability and dependability could be compromised if the code isn't strong enough to handle high user loads or unexpected situations. Users who depend on timely and accurate weather updates may be impacted by this.
* Breach of Intellectual Property: On the off chance that the code consolidates protected or licensed calculations, information, or plans without appropriate approval, it can bring about lawful results, including encroachment cases and likely monetary harms.
* Absence of Straightforwardness and Responsibility: In the event that the code and its basic calculations are not straightforwardly archived or on the other hand on the off chance that responsibility components are not set up, it can make difficulties in understanding how choices are made, tending to likely predispositions, or redressing mistakes or deficiencies.
* Unintended Consequences: Assuming the code is sent without legitimate testing, risk appraisal, or effect examination, it might prompt potentially negative side-effects. These can have a big impact on user trust, system stability, or broader societal implications, ranging from minor inconveniences to major disruptions.

**b. Ideas to keep away from moral and expert issues in the product:**

* Direct Moral Effect Appraisals: Perform ethical impact assessments to identify potential ethical issues, biases, privacy concerns, and other risks prior to deploying the software. To ensure a comprehensive evaluation, stakeholders, domain experts, and ethicists should participate in this assessment.
* Adhere to Moral Rules and Principles: Follow established ethical guidelines like those provided by professional organizations like the IEEE-CS (Institute of Electrical and Electronics Engineers Computer Society) or the Australian Computer Society (ACS). These rules offer standards and best practices for guaranteeing protection, information security, decency, responsibility, and straightforwardness in programming advancement and arrangement.
* Developers can anticipate and address potential ethical and professional issues by carrying out ethical impact assessments and adhering to established ethical guidelines. This ensures that the software project minimizes risks and negative effects for users and society as a whole while still promoting transparency, fairness, privacy, data security, and accountability.

**Discussion**

I have created and implemented code for two modules during this assessment: "compare\_temperature" and "find\_season" Using Equivalence Partitioning and Boundary Value Analysis, I have also designed test designs for black-box testing. In any case, I have not yet carried out the white-box testing and relating test code. Pondering my work, I accept that I have effectively tended to the necessities of the appraisal by giving the essential code and test plans. The code for the modules is useful and gives the normal results in view of the given data sources. To guarantee thorough testing, the black-box test designs incorporate a variety of scenarios and inputs. I would concentrate on implementing the modules' white-box testing in order to further enhance my work. White-box testing considers a more profound assessment of the code's inward rationale and ways, empowering the ID of any likely issues or uncovered regions. I can increase test coverage and discover any hidden bugs or edge cases by implementing white-box testing. In addition, I would work to make the code more modular and reusable. The code can be easier to maintain and more adaptable to upcoming additions or changes if it is broken down into smaller, reusable components and follows best practices like encapsulation and abstraction. The codebase's overall quality and sustainability would rise as a result of this. Last but not least, I would keep improving and expanding the test suite to include more edge cases and boundary conditions. This will assist with guaranteeing that the code is hearty and can deal with different situations and data sources actually.

**Conclusion**

In conclusion, this project involved coding two modules: "compare\_temperature" and "find\_season" The code successfully carries out its intended functions, which include providing temperature comparisons for various cities and times of day and season identification based on location and month. All through the venture, black-box test plans were made utilizing Comparability Parceling and Limit Worth Examination to guarantee thorough testing. However, white-box testing and the associated test code have not yet been implemented. Implementing white-box testing to thoroughly examine the code's internal logic and paths is recommended to improve the project, increase test coverage, and uncover potential issues. Additionally, putting an emphasis on the modularity and reusability of code can make the codebase easier to maintain and more receptive to upcoming changes. The code's robustness and dependability will be further confirmed by expanding the test suite to include edge cases and boundary conditions. In software development, ethical and professional considerations are also essential. Developers can mitigate potential negative effects by ensuring privacy, data security, fairness, transparency, and accountability in their software projects by carrying out ethical impact assessments and adhering to established guidelines. In rundown, while the venture meets the underlying prerequisites, there are regions for development, for example, carrying out white-box testing, improving code seclusion, and growing the test suite. The codebase can be improved in terms of quality, dependability, and maintainability by addressing these aspects.

**Appendix:**

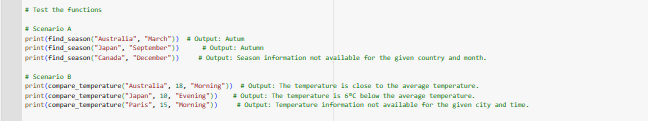
**Code for Scenario one:**



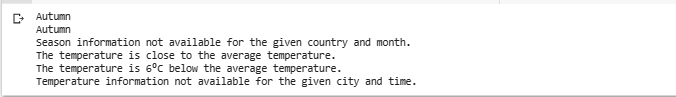
**Code for Scenario two:**

****

**Test code**

****

**Output**

****