**Modularity**

**Description of code:**

* **Name**: **calculate\_and\_display\_results**
* **Purpose**: This function was designed to handle several tasks within a single operation. It collects user inputs, validates the year, calculates the Life Path Number, identifies the Lucky Colour, determines if the Life Path Number is a Master Number, identifies the generation of the person, and finally, displays all these results to the user.
* **Inputs**: The function directly takes no parameters; instead, it prompts the user to enter the day, month, and year of their birth through the console.
* **Outputs**: There are no return values from this function. All outputs, including the Life Path Number, whether it's a Master Number, the corresponding Lucky Colour, and the generation classification, are printed directly to the console.

**Responsibilities Handled by the Function:**

1. **Input Collection**: The function prompts the user to input their day, month, and year of birth.
2. **Input Validation**: It checks if the provided year is within the acceptable range (1901 to 2024).
3. **Numerology Calculation**: Calculates the Life Path Number using a numerological method that sums up the digits of the day, month, and year, reducing them to a single digit unless it's a Master Number (11, 22, 33).
4. **Master Number and Colour Determination**: Determines if the Life Path Number is one of the special Master Numbers and assigns a lucky colour based on the final number.
5. **Generation Identification**: Identifies the generational category based on the year provided (e.g., Silent Generation, Baby Boomers, Generation X, Millennials, Generation Z, Generation Alpha).
6. **Output Presentation**: Displays the results (Life Path Number, Master Number status, Lucky Colour, and Generation) to the user through the console.

s

**Input Methods Used:**

* **Keyboard Entry in Main**: The **main** function handles all user interactions, including input via the keyboard.
* **Parameter Passing to Functions**: After gathering input, the **main** function passes these values as parameters to other functions.

**Output Methods Used:**

* **Direct Output from Main**: The **main** function is responsible for all direct outputs to the console, using results returned from the other functions.

In designing my python program for numerology and generation identification, I aimed to make the code easy to understand and maintain. Here are the key design decisions and how they support good modularity principles:

**1. Separation of Concerns**

* **Purpose**: Each module (function) has a specific job. This makes the code easier to understand and test.
* **Example**: The function **calculate\_life\_path\_number** only calculates the Life Path Number and doesn't mix this task with input from the user or determine the color.

**2. Reusability**

* **Purpose**: Functions are designed to be used many times without changes.
* **Example**: The **sum\_digits** function is used in multiple places to add up the numbers. This avoids rewriting the same code again and again.

**3. Simplicity in Interaction**

* **Purpose**: Functions communicate with each other using simple inputs and outputs. This makes it easy to trace what the program is doing.
* **Example**: Functions like **reduce\_to\_single\_digit** and **calculate\_life\_path\_number** pass results back and forth using simple numbers.

**4. Encapsulation**

* **Purpose**: Keeping details inside the functions and not exposing unnecessary information. This helps in making changes without affecting other parts of the program.
* **Example**: The **get\_lucky\_colour** function decides the color inside itself based on the number it gets. The rest of the program doesn't need to know how the color is chosen.

**Modularity Review Checklist:**

1. **Coupling**: Are different parts of the code too dependent on each other?
   * Can one part of the code be changed without needing to change other parts?
2. **Cohesion**: Does each part of the code focus on a single task?
   * Are tasks mixed together in confusing ways?
3. **Redundancy**: Is the same code written more than once?
   * Can we write this code just once and use it many times?

Using this checklist, let's review the **calculate\_and\_display\_results** function in the initial code.

1. **Coupling**:
   * **High Coupling**: The function manages input, calculation, and output all together. If you change how inputs are taken or how results are displayed, you might also need to adjust the calculation part, which isn't ideal.
2. **Cohesion**:
   * **Low Cohesion**: The function does many things: it checks inputs, calculates numbers, decides colors, determines if a number is a master number, finds out which generation someone belongs to, and shows all these results. This mix makes it hard to understand and maintain.
3. **Redundancy**:
   * **Significant Redundancy**: The method for adding up digits is used multiple times. We could make this a separate function to avoid repeating the same code.

**Refactoring the Code:**

To solve these issues, we can split the big function into smaller, more focused functions. Each function will do one thing well, which makes the whole program easier to understand and change.

**1. sum\_digits Module**

* **Name**: **sum\_digits**
* **Purpose**: This module sums the digits of a given integer.
* **Inputs**: The module takes a single integer **number** as input.
* **Outputs**: It returns the sum of the digits of the input number as an integer. This output is utilized by other functions in the program that require a breakdown of a number into its constituent digits and need the sum of these digits.

**2. reduce\_to\_single\_digit Module**

* **Name**: **reduce\_to\_single\_digit**
* **Purpose**: To reduce any given integer to a single digit or to a Master Number (11, 22, 33) following numerology rules.
* **Inputs**: Receives an integer **number** that represents either a day, month, or year total, or an accumulation of these values.
* **Outputs**: Outputs a single digit integer or a Master Number. This function calls **sum\_digits** to compute the sum of digits repeatedly until a single digit or Master Number is achieved. This output is critical for calculating the Life Path Number and is also used in determining the lucky colour.

**3. calculate\_life\_path\_number Module**

* **Name**: **calculate\_life\_path\_number**
* **Purpose**: Calculates the Life Path Number based on a person's birth date, using numerological reduction.
* **Inputs**: Three integers representing the day, month, and year of a person's birthdate.
* **Outputs**: Returns a single digit or a Master Number that represents the Life Path Number. This number is then used to determine the corresponding lucky colour and to print the final numerology report.

**4. get\_lucky\_colour Module**

* **Name**: **get\_lucky\_colour**
* **Purpose**: Maps the Life Path Number to its corresponding lucky colour based on predefined numerological associations.
* **Inputs**: A single integer, the Life Path Number, computed by **calculate\_life\_path\_number**.
* **Outputs**: Returns a string that represents the lucky colour associated with the given Life Path Number. This information is included in the final output to the user.

**5. find\_generation Module**

* **Name**: **find\_generation**
* **Purpose**: Determines the generational cohort a person belongs to based on their birth year.
* **Inputs**: A single integer representing the year of birth.
* **Outputs**: Returns a string that names the generational cohort (e.g., Generation X, Millennials). This helps contextualize the user’s birth year in terms of broader sociocultural categories and is included in the final output to the user.

**6. main Module**

* **Name**: **main**
* **Purpose**: Serves as the entry point for the program, orchestrating the flow of data among other modules, handling user input, and presenting results.
* **Inputs**: User inputs gathered via command-line prompts for day, month, and year.
* **Outputs**: Does not return data but outputs the Life Path Number, Master Number status, lucky colour, and generation directly to the console for the user’s information.

**Results and Actions After Refactoring:**

* **Reduced Coupling**: Each function now works more independently. Changes in one function are less likely to require changes in others.
* **Increased Cohesion**: Each function has a clear, single purpose, making the code easier to understand and test.
* **Reduced Redundancy**: Repeated logic is now encapsulated in specific functions, reducing code duplication.

**Test Design**

**1.Black Box testing**

**Equivalence Partitioning**

This method divides the input data into partitions that can be considered equivalent. We will create partitions for the **year** input as it has clear validation requirements. The test cases will check whether the app correctly accepts or rejects years based on the specified range (1901 to 2024).

**Equivalence Classes for Year Input:**

1. **Valid Year Class**: Years within the range (1901 to 2024).
2. **Invalid Year Class (Below Range)**: Years below the minimum year (1900 and earlier).
3. **Invalid Year Class (Above Range)**: Years above the maximum year (2025 and later).

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Year Input** | **Expected Result** | **Description** |
| TC1 | 2000 | Pass | Year within the valid range. |
| TC2 | 1901 | Pass | Boundary year at the start of the range. |
| TC3 | 2024 | Pass | Boundary year at the end of the range. |
| TC4 | 1899 | Fail (Invalid Year Entered) | Year below the valid range. |
| TC5 | 2025 | Fail (Invalid Year Entered) | Year above the valid range. |

**Boundary Value Analysis**

This method focuses on the values at the edges of each equivalence class.

**Boundary Values for Year Input:**

* **Boundary at Lower End**: 1901 (minimum valid year), 1900 (just below minimum).
* **Boundary at Upper End**: 2024 (maximum valid year), 2025 (just above maximum).

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Year Input** | **Expected Result** | **Description** |
| TC6 | 1901 | Pass | Exact boundary at the lower end. |
| TC7 | 1900 | Fail (Invalid Year Entered) | Just below the valid range. |
| TC8 | 2024 | Pass | Exact boundary at the upper end. |
| TC9 | 2025 | Fail (Invalid Year Entered) | Just above the valid range. |

For white-box testing, we'll select two modules from the refactored code that represent different constructs: the **calculate\_life\_path\_number** method which includes loop and conditional logic, and the **determine\_master\_and\_color** method, which primarily uses decision-based constructs (conditional statements).

**Selected Modules for White-Box Testing:**

1. **Module: calculate\_life\_path\_number**
   * **Constructs Tested**: Loop and Conditional Logic
   * **Purpose**: This method calculates the Life Path Number using the inputs of day, month, and year. It includes a loop to sum digits until a single digit or a Master Number is obtained.
2. **Module: determine\_master\_and\_color**
   * **Constructs Tested**: Conditional Statements
   * **Purpose**: This method determines whether the Life Path Number is a Master Number and assigns a lucky color based on this number.

**2.White Box testing**

For white-box testing, we'll select two modules from the refactored code that represent different constructs: the **calculate\_life\_path\_number** method which includes loop and conditional logic, and the **determine\_master\_and\_color** method, which primarily uses decision-based conditional statements.

**Selected Modules for White-Box Testing:**

1. **Module: calculate\_life\_path\_number**
   * **Constructs Tested**: Loop and Conditional Logic
   * **Purpose**: This method calculates the Life Path Number using the inputs of day, month, and year. It includes a loop to sum digits until a single digit or a Master Number is obtained.
2. **Module: determine\_master\_and\_color**
   * **Constructs Tested**: Conditional Statements
   * **Purpose**: This method determines whether the Life Path Number is a Master Number and assigns a lucky color based on this number.

**Test Cases for calculate\_life\_path\_number:**

**Test Strategy**: Verify the loop and conditional logic by providing inputs that lead to different paths through the method.

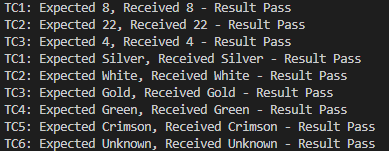
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Day** | **Month** | **Year** | **Expected Life Path Number** | **Description** |
| TC1 | 15 | 8 | 1992 | 8 | Single digit result without loop |
| TC2 | 29 | 12 | 1988 | 11 | Path resulting in a Master Number without further reduction |
| TC3 | 25 | 10 | 1994 | 6 | Requires multiple loops to reduce to a single digit |

**Test Cases for determine\_master\_and\_color:**

**Test Strategy**: Ensure each branch of the conditional logic is tested, especially the boundary and error cases.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case ID** | **Life Path Number** | **Expected Master Status** | **Expected Colour** | **Description** |
| TC1 | 11 | Yes | Silver | Master Number with specific color |
| TC2 | 22 | Yes | White | Another Master Number with specific color |
| TC3 | 9 | No | Gold | Non-Master Number with associated color |
| TC4 | 4 | No | Green | Regular path with correct color mapping |
| TC5 | 33 | Yes | Crimson | Highest Master Number with unique color |

Following is evidence of the executed tests:



**Summary Table**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Module Name** | **Test Design (Black-Box)** | **Test Design (White-Box)** | **Data Types** | **Input/Output Form** | **Equivalence Partitioning (EP)** | **Boundary Value Analysis (BVA)** | **White-Box** |
| **calculate\_life\_path\_number** | Done | Not Done | Integer (day, month, year) | Numeric inputs/Single numeric output | Done | Done | Not Done |
| **get\_lucky\_colour** | Done | Not Done | Integer (life path number) | Numeric input/String output | Done | Done | Not Done |
| **find\_generation** | Done | Not Done | Integer (year) | Numeric input/String output | Done | Done | Not Done |

**Discussion of My Work**

**Achievements:**

* **Comprehensive Testing**: I successfully designed and implemented test cases using both black-box and white-box testing methods. This included equivalence partitioning and boundary value analysis to ensure thorough testing of the application's functions like **calculate\_life\_path\_number** and **get\_lucky\_colour**.
* **Modular Code Refactoring**: The code was refactored to improve modularity. Functions were clearly defined to handle specific tasks, making the code easier to understand and maintain.
* **Detailed Output in Testing**: Enhanced the test cases to include detailed outputs, which helps in quickly identifying the test results and understanding why a particular test case failed or passed.

**Challenges Faced:**

* **Understanding Testing Requirements**: Initially aligning the test cases with the specific requirements of the application was challenging, especially ensuring that the boundary values and equivalence classes covered all possible scenarios.
* **Correct Implementation of Test Cases**: Ensuring that the test cases correctly reflected the logic within the code required careful consideration, particularly in managing data types and expected outcomes.

**Limitations:**

* **Lack of Automated Testing Frameworks**: All tests were conducted manually without the use of automated testing frameworks like PyTest. This limited the efficiency and scalability of the testing process.
* **Limited Scope of White-Box Testing**: White-box testing was not applied as comprehensively as it could have been, which might leave some internal logic paths insufficiently tested.

**Ways to Improve:**

* **Implementing Automated Testing**: Utilizing an automated testing framework would streamline the testing process, allow for more complex test scenarios, and facilitate continuous testing as the code evolves.
* **Expanding White-Box Testing**: More extensive white-box testing could be implemented to ensure all logical paths are tested, which would improve the robustness of the application.
* **User Interface Testing**: Adding tests for a user interface, if applicable, could be considered to ensure that user interactions are as expected and the UI displays the correct outputs based on various inputs.