

Intruduction

This Python codebase is designed to findout life path numbers, lucky colors of people based on their birthdate using object-oriented programming. It includes functionality for parsing different birthday formats, determining generation, and validating user input. We can also findout if two person are from same life path number and whether that number is master number or not.

We applied modularity concept along with object-oriented programming to reduce coupling and to improve cohesiveness of the program. Here I also applied blackbox testing and whitebox testing depending which one is more appropriate for that module.

Module Description

First I thought about having a Class (Person) for the person we are working on, in that we will have properties and method for determining life path number, lucky color and generation. Then the other functions we might need for implementation will be declared globally. This way we will have one sub module for one well defined task which is the ultimate goal we want to achieve in modularity.

Person Class This class represents a person with attributes such as name, birthday, generation, and life path number. It includes methods to determine a person's generation, calculate their life path number, and identify their lucky color based on their life path number.

Attributes:

- name: The name of the person.
- birthday: The birthday of the person in DDMMYYYY format.
- generation: The generation of the person based on their birth year.
- life_path_num: The life path number calculated from the birthday.

Methods:

- `_init_(self, name, birthday)`: Initializes the person with a name and birthday, and computes the generation and life path number.
- `get_lucky_color(self)`: Returns the lucky color based on the life path number.
- `get_life_path_number(self)`: Calculates and returns the life path number.
- `get_generation(self)`: Determines and returns the generation based on the birth year.

Design Decisions and Assumptions

Design Decisions:

Modularity and Separation of Concerns:

The design separates the main functionalities in Person class. This adheres to the principle of separation of concerns, where Person handles the attributes and behaviors of an individual, and global modules provides utility functions to support these behaviors.

By separating these concerns, the code is more maintainable, easier to understand, and allows for independent testing of each class.

Assumptions:

- The birthday input is assumed to be either in DD-MM-YYYY or DD Month YYYY format. This is handled by the `get_valid_birthday_input` method, which converts the input to a consistent DDMMYYYY format.
- I assumed the property name for the Person class and passed it as hard coded peremeter in the constructor.

Modularity

While implementing the initial design using modularity concept I got better ideas of improving maintainability and readability of the code.

- I moved my global methods to Helper class which opened rooms for future improvement, easier to understand and allows for independent testing of each class.
- I used static methods in the Helper class because these methods do not rely on instance-specific data. They perform general tasks that can be utilized without needing an instance of the class, making them suitable for utility functions.
- For unit testing I made two separate class as well. TestPerson and TestHelper for the respected classes.

Now the final design separates the functionalities into two classes: Person and Helper. This adheres to the principle of separation of concerns, where Person handles the attributes and behaviors of an individual, and Helper provides utility functions to support these behaviors.

Modularity Concepts Applied in the Code

Separation of Concerns

- **Person Class:** Manages attributes and behaviors related to an individual, such as:
 - Name
 - Birthday
 - Generation
 - Life path number
- **Helper Class:** Provides utility functions that support the Person class, including:
 - Summing digits
 - Checking master numbers
 - Comparing life path numbers
 - Validating input

Encapsulation

- **Person Class:** Encapsulates the logic for determining generation and life path number. Users of the class only need to know how to use the methods, not the implementation details.
-
- **Helper Class:** Encapsulates utility functions, allowing them to be reused without exposing their internal logic to the rest of the program.

Reusability

- The static methods in the Helper class are designed to be reusable across different parts of the application without modification.
- Methods in the Person class are designed to handle specific tasks related to the individual's data, making them reusable for any Person object.

Single Responsibility Principle (SRP)

- Each class and method has a single responsibility. For example:
 - `get_lucky_color` is responsible solely for determining the lucky color based on the life path number.
 - `get_generation` solely determines the generation based on the birth year.

Review Checklist

Code Clarity

1. Are the class and method names descriptive?
2. Is there any redundant code?

Functionality

1. Do methods perform their intended function correctly?
2. Are edge cases handled properly?

Modularity

1. Are the responsibilities of the classes and methods well-defined and separated?
2. Is the code organized into logical sections or modules?

Reusability

1. Are utility functions designed to be reusable?
2. Is there any duplicated code that could be refactored into reusable functions?

Test Coverage

1. Are there unit tests for all major functions and edge cases?
2. Do tests cover both typical and atypical usage scenarios?

Error Handling

1. Are potential errors and exceptions handled gracefully?
2. Is input validation performed to prevent invalid data?

Review Results and Refactoring Decisions

Review Checklist Results

Code Clarity

- The class and method names are descriptive and clear.
- There is no redundant code that is worth mentioning

Functionality

- Methods perform their functions correctly.
- Edge cases are generally well-handled, particularly in the input validation method.

Modularity

- Responsibilities are well-defined and separated between Person and Helper classes.
- The code is logically organized, making it easy to understand and maintain.

Reusability

- Utility functions in the Helper class are designed to be reusable.
- There are no significant instances of duplicated code.

Test Coverage

- Unit tests cover the major functionalities and edge cases effectively.
- Additional edge case tests, such as unusual but valid dates, would enhance coverage.

Refactoring Decisions

Enhance Input Validation

- Add more validation in the Person class constructor to ensure the birthday is in the correct format before proceeding with calculations.

Refactor global Methods

Add them in a new class as static method. Example:

Before:

```
def sum_digits(num):
    total = 0
    while num > 0:
        digit = num%10
        total = total + digit
        num = num//10
    return total

def is_master_number(num):
    master_number = [11,22,33]
    return num in master_number

def lifePathCompare(person1, person2):
    return person1.life_path_num == person2.life_path_num
```

After

```
class Helper:
    @staticmethod
    def sum_digits(num):
        total = 0
        while num > 0:
            digit = num%10
            total = total + digit
            num = num//10
        return total

    @staticmethod
    def is_master_number(num):
        master_number = [11,22,33]
        return num in master_number

    @staticmethod
    def lifePathCompare(person1, person2):
        return person1.life_path_num == person2.life_path_num
```

Refactor parse_birthday() and get_valid_birthday_input() functions:

These two modules had strong coupling which made it very difficult to design test code for them. To eliminate coupling I merged them. Example:

Before

```
def parse_birthday(birthday):
    try:
        return datetime.strptime(birthday, "%d-%m-%Y").strftime("%d%m%Y")
    except ValueError:
        try:
            return datetime.strptime(birthday, "%d %B %Y").strftime("%d%m%Y")
        except ValueError:
            return False

def get_valid_birthday_input():
    while True:
        birthdate = input("Enter birthday (DD-MM-YYYY or DD Month YYYY): ")
        birthdate = parse_birthday(birthdate)
        if birthdate:
            return birthdate
        else:
            print("Invalid birthday format. Please use either DD-MM-YYYY or DD Month YYYY.")
```

After

```
@staticmethod
def get_valid_birthday_input():
    while True:
        birthdate = input("Enter birthday (DD-MM-YYYY or DD Month YYYY): ")
        try:
            formatted_date = datetime.strptime(birthdate, "%d-%m-%Y").strftime("%d%m%Y")
            return formatted_date
        except ValueError:
            try:
                formatted_date = datetime.strptime(birthdate, "%d %B %Y").strftime("%d%m%Y")
                return formatted_date
            except ValueError:
                print("Invalid birthday format. Please use either DD-MM-YYYY or DD Month YYYY.\n")
```

Refactor life_path_compare method:

Refactored the method to do only one well defined job which is compare the life paths. printing is then handled by main() function. By doing so, it was much easier to design test code for this module.

Before

```
@staticmethod
def lifePathCompare(person1, person2):
    print(f"{person1.name} life path number is {person1.life_path_num}\n{person2.name} life path number is {person2.life_path_num}")
    if(person1.life_path_num == person2.life_path_num):
        print("life path number is same")
    else:
        print("life path number is not same")
```

After

```
# in class Helper
@staticmethod
def lifePathCompare(person1, person2):
    return person1.life_path_num == person2.life_path_num

#then in main()
....
if(Helper.lifePathCompare(person, person2)):
    print("Their life path number is same")
else:
    print("Their life path number is not same")
```

Combine Common Logic in Helper Methods

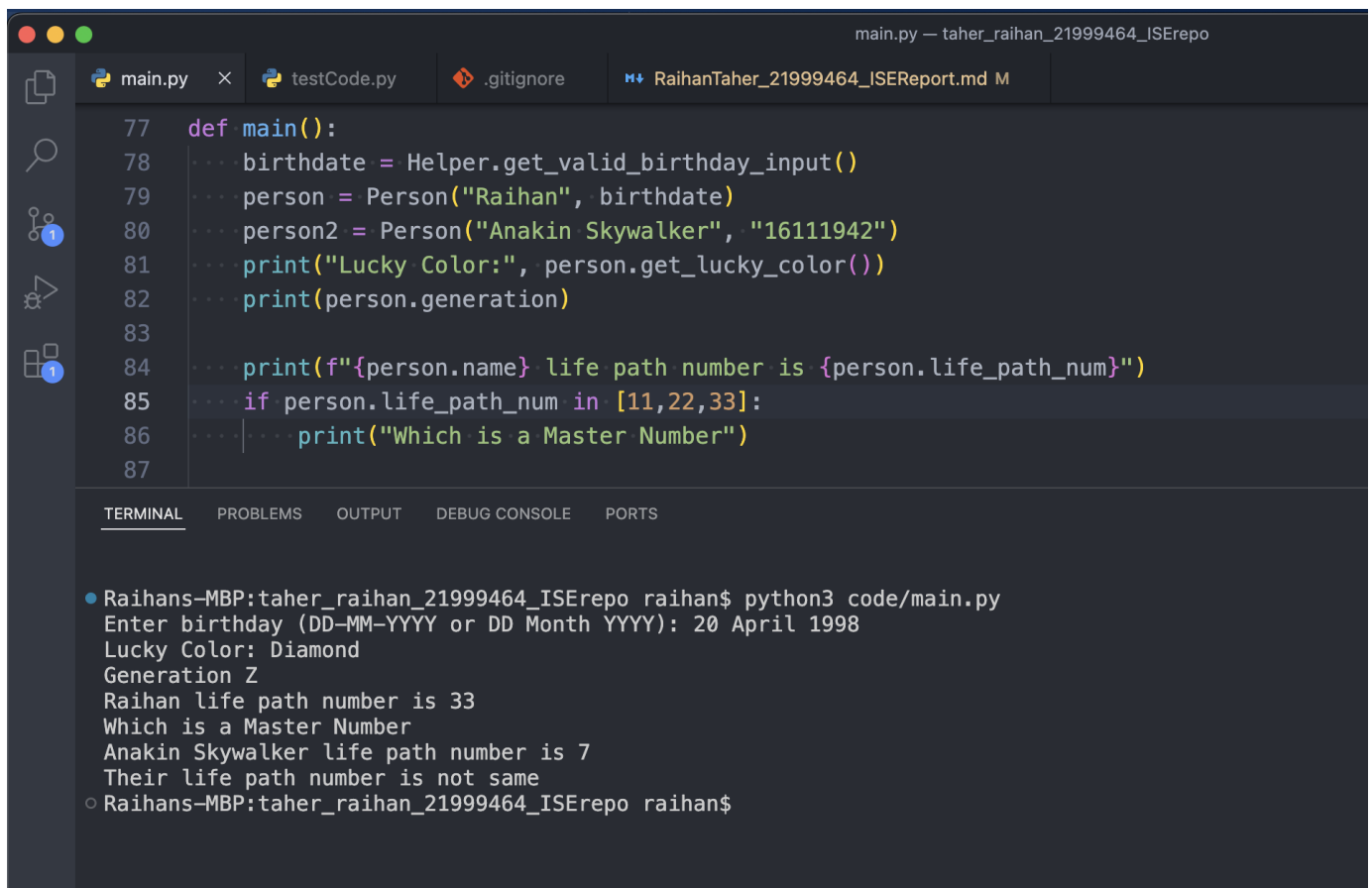
- Ensure that any repeated logic in Helper methods is combined into a single method to avoid redundancy.

Execute the Program

To run the program just execute the main.py file

```
python3 ./code/main.py
```

Then Enter a valid date with appropriate format (Please, Check the Screenshot)



The screenshot shows a code editor with a dark theme. The top bar indicates the file is 'main.py' in the repository 'taiher_raihan_21999464_ISERepo'. The editor displays the following Python code:

```
77 def main():
78     birthdate = Helper.get_valid_birthday_input()
79     person = Person("Raihan", birthdate)
80     person2 = Person("Anakin Skywalker", "16111942")
81     print("Lucky Color:", person.get_lucky_color())
82     print(person.generation)
83
84     print(f"{person.name} life path number is {person.life_path_num}")
85     if person.life_path_num in [11, 22, 33]:
86         print("Which is a Master Number")
87
```

Below the code editor is a terminal window with the following output:

```
• Raihans-MBP:taiher_raihan_21999464_ISERepo raihan$ python3 code/main.py
Enter birthday (DD-MM-YYYY or DD Month YYYY): 20 April 1998
Lucky Color: Diamond
Generation Z
Raihan life path number is 33
Which is a Master Number
Anakin Skywalker life path number is 7
Their life path number is not same
○ Raihans-MBP:taiher_raihan_21999464_ISERepo raihan$
```

Black Box Testing

Equivalence Partitioning Test Cases

We will apply equivalence partitioning to the following modules and methods:

- 1. `Person.get_life_path_number`
- 2. `Helper.sum_digits`
- 3. `Helper.is_master_number`
- 4. `Helper.lifePathCompare`
- 5. `Helper.get_valid_birthday_input`

Here is the detailed approach and the test cases for each method:

1. `Person.get_life_path_number`

Submodule: `get_life_path_number`
Imports: `self` (object)
Exports: `result` (int)
This method calculates the life path number based on the birthday. We consider the birthday as an integer in `DDMMYYYY` format.

Category	Test Data	Expected Result
Not Master Number	"01011901"	4
Master Number	"20041998"	33
Minimum valid birthday.	"20012004"	9
Maximum valid birthday	"31122024"	6

2. `Helper.sum_digits`

Submodule: `get_life_path_number`
Imports: `num` (int)
Exports: `result` (int)
This method calculates the sum of the digits of a given number.

Category	Test Data	Expected Result
one digit number	5	5
Multiple digit number	123	6
Large number	987654321	45
Zero	0	0

3. `Helper.is_master_number`

Submodule: `get_life_path_number`
Imports: `num` (int)
Exports: `result` (boolean)
This method checks if a number is a master number (11, 22, 33).

Category	Test Data	Expected Result
Master Number	33	True
Not a Master Number	10	False
Invalid Data type	"Taher"	False
Invalid Input	None	False
Invalid Data type	[11]	False

4. **Helper.lifePathCompare**

```
Submodule: lifePathCompare
Imports: person1, person2 (objects)
Exports: result (boolean)
This method compares the life path numbers of two persons.
```

Category	Input	Expected Result
Same life path number	5, 5	True
Different life path number	1, 9	False
Invalid input (string)	"9464", 5	False
Invalid input (None)	None, 5	False

5. **Helper.get_valid_birthday_input** (Mock input for testing)

```
Submodule: get_valid_birthday_input
Imports:
Exports: birthday(string)
This method prompts the user for a birthday and validates the input format.
```

Category	Test Data	Expected Output
Valid input (DD-MM-YYYY)	"01-01-2000\n"	"01012000"
Valid input (DD Month YYYY)	"15 June 1995\n"	"15061995"
Multiple invalid inputs then by input	"TAHER 9464\n22 Feb 2022\n05 May 2023\n"	"05052023"

Summary

Equivalence partitioning helps us ensure that all different input scenarios are covered with a minimal number of test cases. The above tables summarize how we can test each module effectively using this approach. These test cases should be implemented in our **unittest** framework to ensure comprehensive testing of the **Person** and **Helper** classes.

Boundary Value Analysis

We will apply Boundary Value Analysis to the following module:

- 1. **Person.get_generation**


```
Submodule: get_generation
Imports: self (object)
Exports: generation (string)
This method calculates determines generation name based on the
```

year from birthday.

Boundary	Test Data	Expected Result
Just below Silent Generation	1900	"Unknown Generation"
Lower boundary of Silent Generation	1901	"Silent Generation"
Upper boundary of Silent Generation	1945	"Silent Generation"
Lower boundary of Baby Boomers	1946	"Baby Boomers"
Upper boundary of Baby Boomers	1964	"Baby Boomers"
Lower boundary of Generation X	1965	"Generation X"
Upper boundary of Generation X	1979	"Generation X"
Lower boundary of Millennials	1980	"Millennials"
Upper boundary of Millennials	1994	"Millennials"
Lower boundary of Generation Z	1995	"Generation Z"
Upper boundary of Generation Z	2009	"Generation Z"
Lower boundary of Generation Alpha	2010	"Generation Alpha"
Upper boundary of Generation Alpha	2024	"Generation Alpha"
Just above Generation Alpha	2025	"Unknown Generation"

White Box Testing

These are the unit test modules we have here.

1. `TestPerson.test_life_path_number`
2. `TestPerson.test_get_generation`
3. `TestHelper.test_valid_birthday`
4. `TestHelper.test_sum_digits`
5. `TestHelper.test_is_master_number`
6. `TestHelper.test_life_path_compare`

1.Test design for `TestPerson.test_life_path_number`

Path	Test Data	Expected Result
Enter the If	20041998	33
Do not Enter the If	01012004	8

Note: Here we are assuming there will be no such case where this code doesn't enter the loop as we are validating the input from another module so its safe to make that assumption.

2.Test design for `TestPerson.test_get_generation`

Path	Test Data	Expected Result
Enter First If (1901-1945)	23041901	Silent Generation
Else If (1946 - 1964)	23041946	Baby Boomers
Else If (1965 - 1979)	23041965	Generation X
Else If (1980 - 1994)	23041980	Millennials
Else If (1995 - 2009)	23041995	Generation Z
Else If (2010 - 2024)	23042010	Generation Alpha
Else	21999464	Unknown

3.Test design for `TestHelper.test_valid_birthday`

Path	Test Data	Expected Result
Enter First try	20-04-1998	20041998
Enter Second try	22 May 2005	22052005
Enter last except	24/05/24	invalid Message - prompts again for input

4.Test design for `TestPerson.test_sum_digits`

Path	Test Data	Expected Result
Enter the While	20041998	33
Do not enter the while	5	5

5.Test design for `TestPerson.test_is_master_number`

Path	Test Data	Expected Result
Condition met	22	True
Doesn't meet condition	5	False

6.Test design for `TestPerson.test_life_path_compare`

Path	Test Data	Expected Result
is Equal	7,7	True
isn't Equal	5,11	False

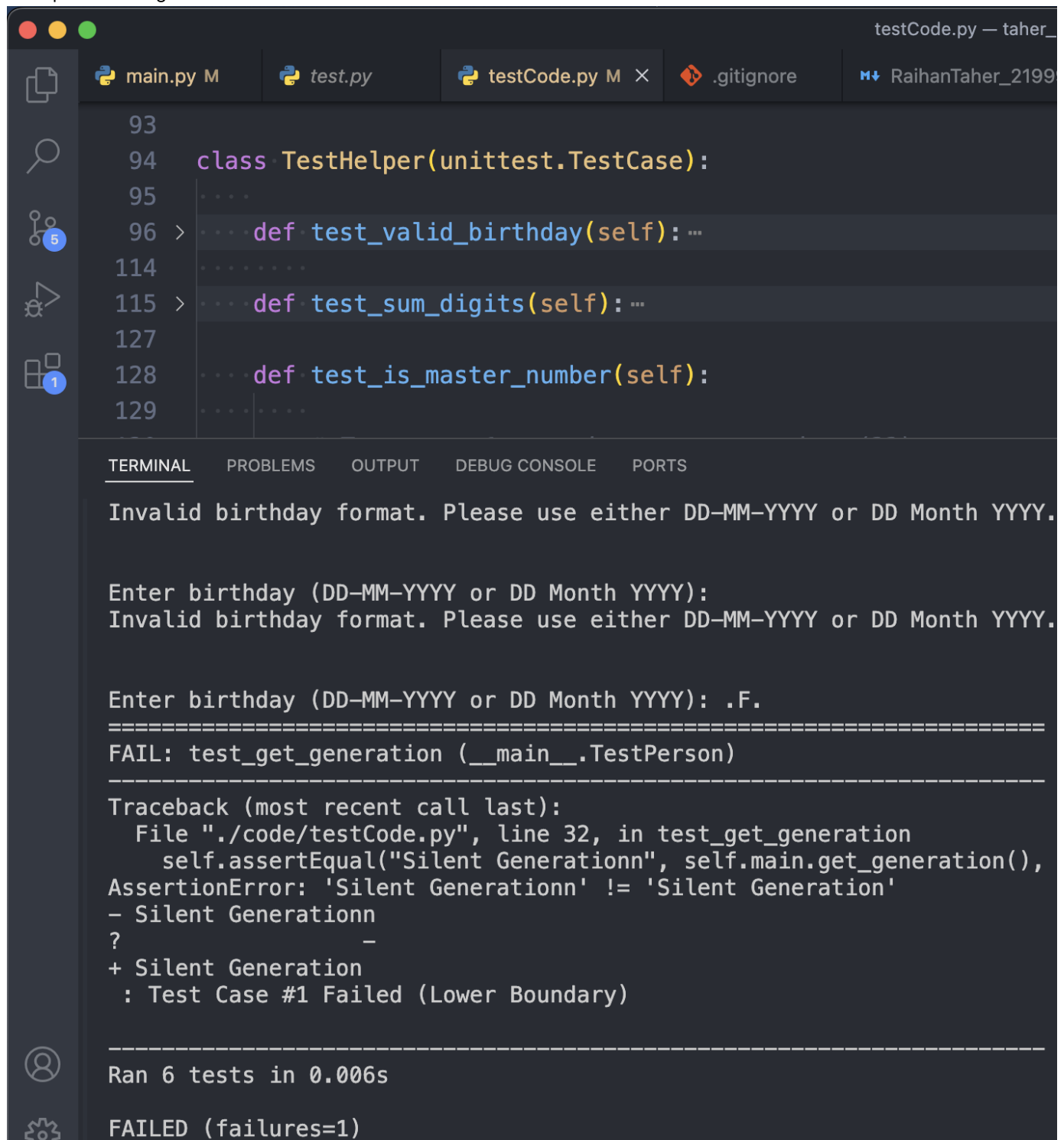
Exaplanation: Here in all the test methods we did follow all possible paths our program could go as we are shown in our workshops.

Test implementation and test execution

After opening the repository directory in the terminal run

```
python3 ./code/testCode.py
```

Example of running test modules with one failed case



The screenshot shows a code editor with a dark theme. The top bar displays the file name 'testCode.py' and the user 'taher_'. The editor has several tabs: 'main.py M', 'test.py', 'testCode.py M', '.gitignore', and 'RaihanTaher_2199'. The code in 'testCode.py' defines a 'TestHelper' class with three test methods: 'test_valid_birthday', 'test_sum_digits', and 'test_is_master_number'. The terminal output shows the execution of these tests. It starts with a prompt for a birthday, followed by two successful tests. The third test, 'test_get_generation', fails with an 'AssertionError' because the generated name 'Silent Generationnn' does not match the expected 'Silent Generation'. The terminal also shows the total number of tests run (6) and the overall result (FAILED with 1 failure).

```
93
94 class TestHelper(unittest.TestCase):
95     ...
96 > ... def test_valid_birthday(self): ...
114     ...
115 > ... def test_sum_digits(self): ...
127     ...
128     ... def test_is_master_number(self):
129     ...
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE PORTS

Invalid birthday format. Please use either DD-MM-YYYY or DD Month YYYY.

Enter birthday (DD-MM-YYYY or DD Month YYYY):

Invalid birthday format. Please use either DD-MM-YYYY or DD Month YYYY.

Enter birthday (DD-MM-YYYY or DD Month YYYY): .F.

=====

FAIL: test_get_generation (__main__.TestPerson)

Traceback (most recent call last):

File "./code/testCode.py", line 32, in test_get_generation

self.assertEqual("Silent Generationnn", self.main.get_generation(),

AssertionError: 'Silent Generationnn' != 'Silent Generation'

- Silent Generationnn

? -

+ Silent Generation

: Test Case #1 Failed (Lower Boundary)

Ran 6 tests in 0.006s

FAILED (failures=1)

Example of running all test modules successfully

testCode.py — taher_raihan_21999464_ISErepo

main.py Mtest.pytestCode.py M X.gitignoreRaihanTaher_21999464_ISEReport.md

```
94 class TestHelper(unittest.TestCase):
114     ...
115 > ... def test_sum_digits(self): ...
127
128 > ... def test_is_master_number(self): ...
144
145 > ... def test_life_path_compare(self): ...
169
170 if __name__ == '__main__':
171     unittest.main()
172
```

TERMINALPROBLEMSOUTPUTDEBUG CONSOLEPORTS

- Raihans-MBP:taher_raihan_21999464_ISErepo raihan\$ python3 ./code/testCode.py

```
...
Enter birthday (DD-MM-YYYY or DD Month YYYY):
Enter birthday (DD-MM-YYYY or DD Month YYYY):
Enter birthday (DD-MM-YYYY or DD Month YYYY):
Invalid birthday format. Please use either DD-MM-YYYY or DD Month YYYY.

Enter birthday (DD-MM-YYYY or DD Month YYYY):
Invalid birthday format. Please use either DD-MM-YYYY or DD Month YYYY.

Enter birthday (DD-MM-YYYY or DD Month YYYY):
Invalid birthday format. Please use either DD-MM-YYYY or DD Month YYYY.

Enter birthday (DD-MM-YYYY or DD Month YYYY): ...
-----
Ran 6 tests in 0.006s

OK
o Raihans-MBP:taher_raihan_21999464_ISErepo raihan$ $|
```

As I mentioned in the refactoring section that perse_birthday() & input_validation() modules have been improved as it was much harder design test code for them with such strong coupling.

Summary

Here is the summary table for the things covered for the program

	Design	of	test	cases		Test code	implementation	and execution
Module name	BB (EP)	BB (BVA)	WB	Data type/s	Form of Input/output	EP	BVA	White-Box
Person.get_lucky_color	not done	not done	not done	int	parameter	not done	not done	not done
Person.get_life_path_number	done	not done	done	string	file input	done	not done	done
Person.get_generation	not done	done	done	string	parameter	not done	done	done

	Design	of	test	cases		Test code	implementation	and execution
Helper.sum_digits	done	not done	done	int	parameter	done	not done	done
Helper.is_master_number	done	not done	done	int	parameter	done	not done	done
Helper.lifePathCompare	done	not done	done	object	parameter	done	not done	done
Helper.get_valid_birthday_input	done	not done	done	string	keyboard input	done	not done	done

Version Control

Git

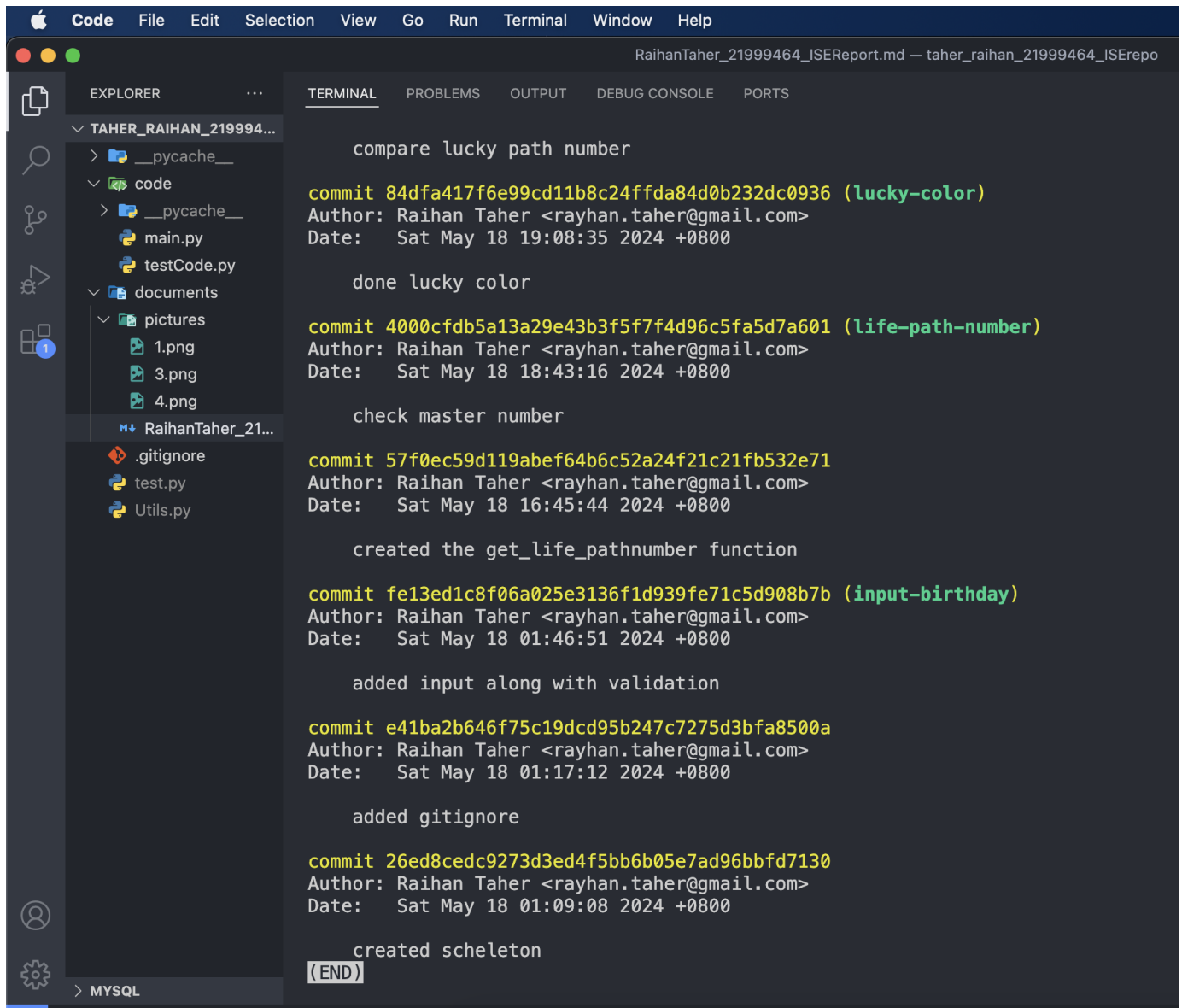
At the beginning, We were thinking about after doing initial setup in main branch, we'll do feature branch for all the distinct modules and then merge all of them in main branch. Ideally in software development its best practice to have a development branch and merge your feature branch there. Finally, when everything up and running fully tested, ready for release then it goes into main branch. But here as its a very simple program and I'm the only developer here so for simplicity we are not using the development branch for this program.

Branch used

```
- main
- input-birthday
- life-path-compare
- life-path-number
- lucky-color
- test-code
```

Their purpose is self explanatory, after each job is done and tested to will be merged in the main branch. Thats the initial plan, which might slightly change while developing.

Log of the use of Version Control:



```
compare lucky path number

commit 84dfa417f6e99cd11b8c24ffda84d0b232dc0936 (lucky-color)
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 19:08:35 2024 +0800

done lucky color

commit 4000cfdb5a13a29e43b3f5f7f4d96c5fa5d7a601 (life-path-number)
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 18:43:16 2024 +0800

check master number

commit 57f0ec59d119abef64b6c52a24f21c21fb532e71
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 16:45:44 2024 +0800

created the get_life_pathnumber function

commit fe13ed1c8f06a025e3136f1d939fe71c5d908b7b (input-birthday)
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 01:46:51 2024 +0800

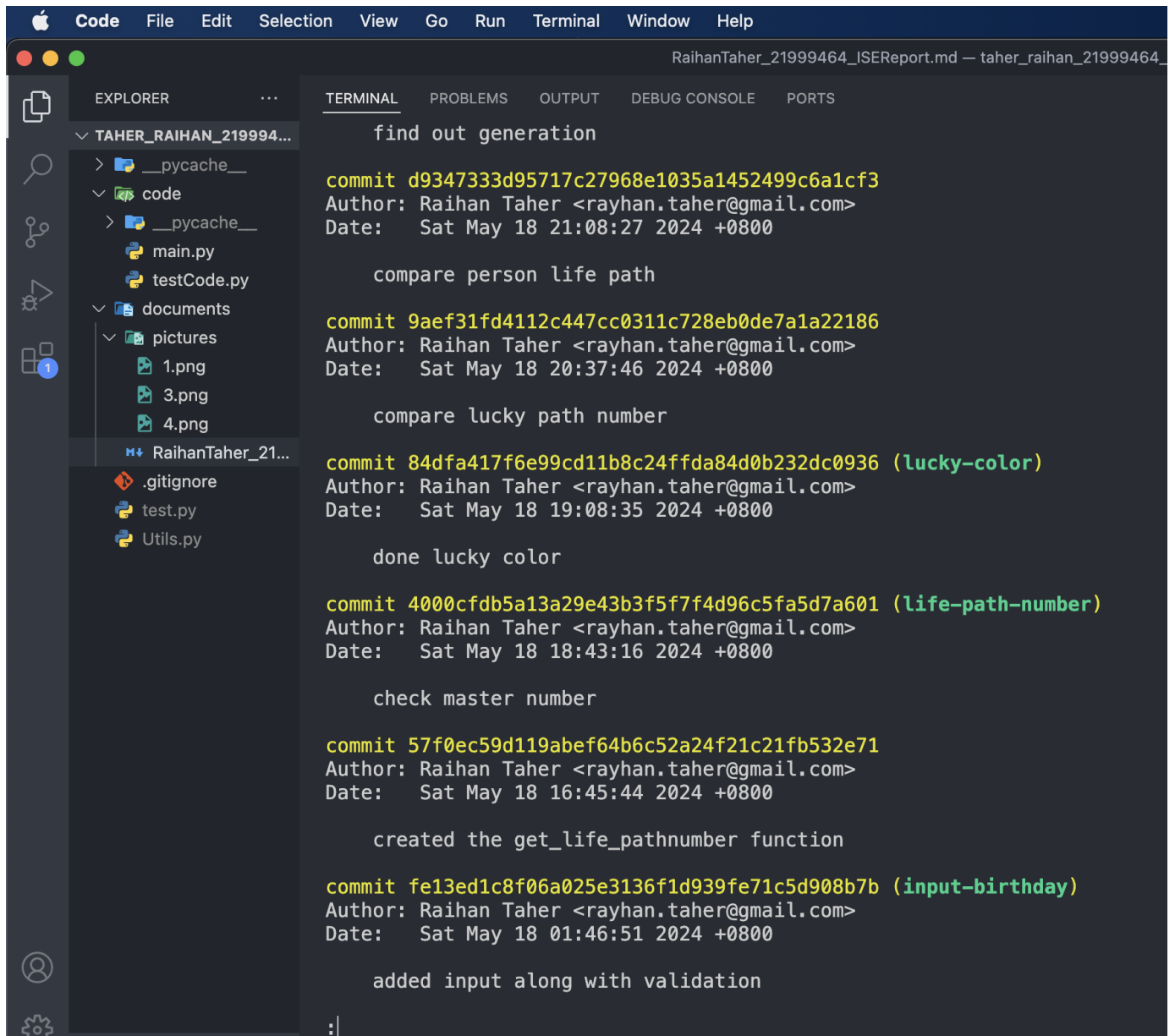
added input along with validation

commit e41ba2b646f75c19dcd95b247c7275d3bfa8500a
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 01:17:12 2024 +0800

added gitignore

commit 26ed8cedc9273d3ed4f5bb6b05e7ad96bbfd7130
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 01:09:08 2024 +0800

created scheleton
(END)
```



The screenshot shows a VS Code editor interface. On the left is the Explorer sidebar showing a file tree for a project named 'TAHER_RAIHAN_219994...'. The tree includes folders like '__pycache__', 'code', and 'documents', and files like 'main.py', 'testCode.py', '1.png', '3.png', '4.png', '.gitignore', 'test.py', and 'Utils.py'. The main area is the Terminal, which displays a series of git commit messages. The messages are as follows:

```
find out generation
commit d9347333d95717c27968e1035a1452499c6a1cf3
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 21:08:27 2024 +0800

compare person life path
commit 9aef31fd4112c447cc0311c728eb0de7a1a22186
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 20:37:46 2024 +0800

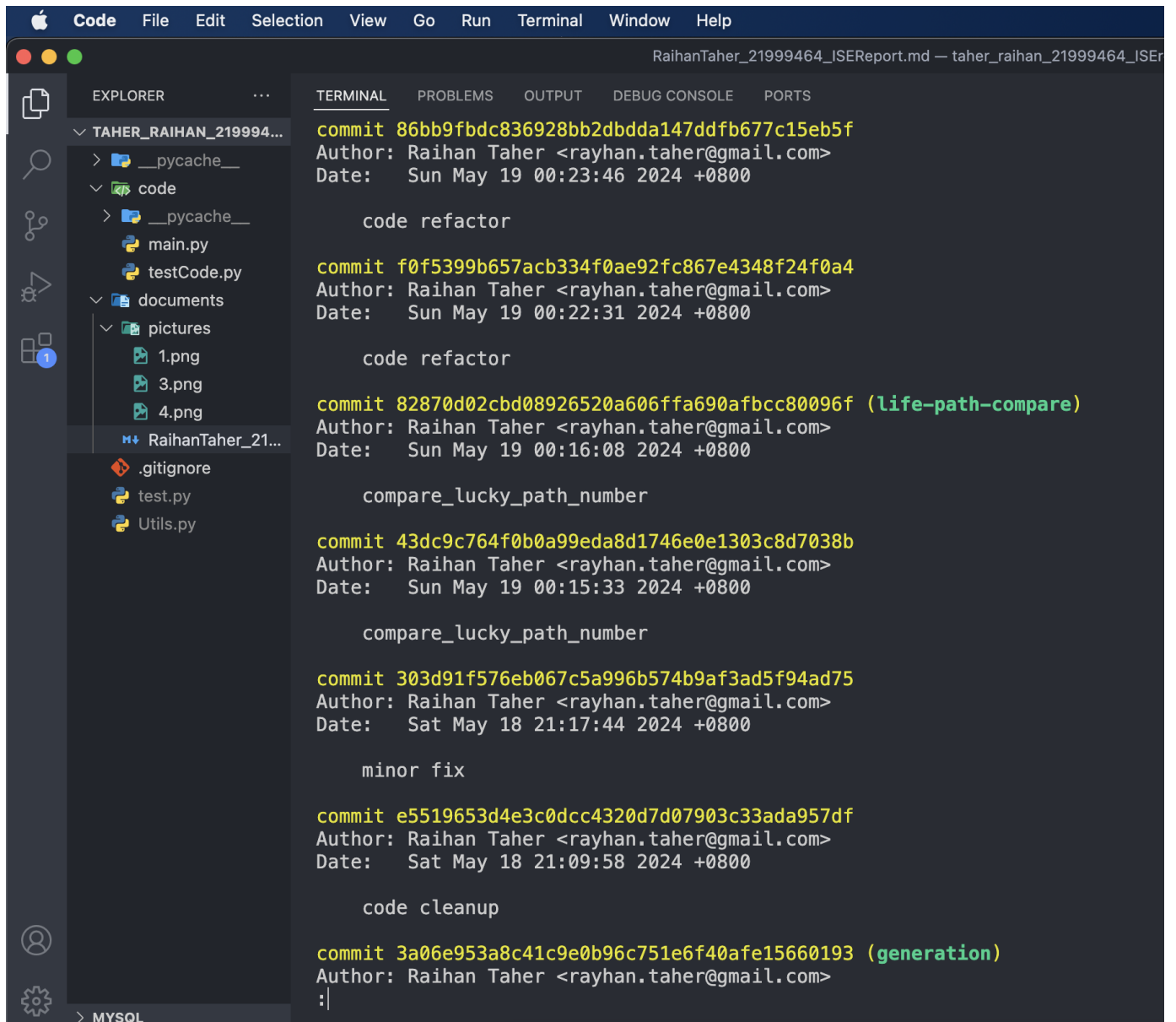
compare lucky path number
commit 84dfa417f6e99cd11b8c24ffda84d0b232dc0936 (lucky-color)
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 19:08:35 2024 +0800

done lucky color
commit 4000cfdb5a13a29e43b3f5f7f4d96c5fa5d7a601 (life-path-number)
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 18:43:16 2024 +0800

check master number
commit 57f0ec59d119abef64b6c52a24f21c21fb532e71
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 16:45:44 2024 +0800

created the get_life_pathnumber function
commit fe13ed1c8f06a025e3136f1d939fe71c5d908b7b (input-birthday)
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 01:46:51 2024 +0800

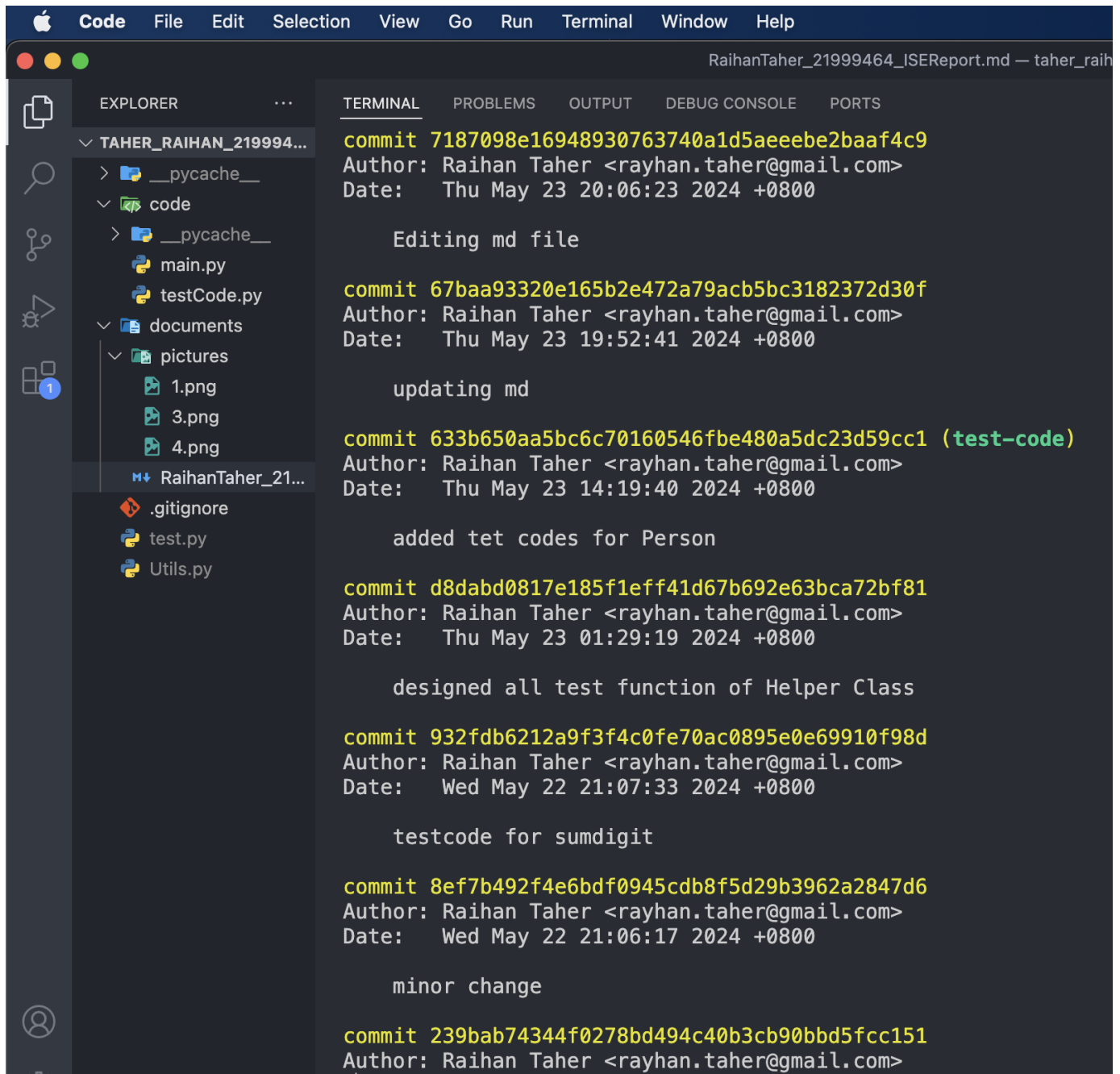
added input along with validation
:
```



The screenshot shows a Visual Studio Code window with a dark theme. The Explorer panel on the left shows a project structure for 'TAHER_RAIHAN_219994...'. The file tree includes folders like '__pycache__', 'code', and 'documents', and files like 'main.py', 'testCode.py', '1.png', '3.png', '4.png', '.gitignore', 'test.py', and 'Utils.py'. The Terminal panel on the right displays a series of git commit messages and hashes. The commits are as follows:

- `commit 86bb9fbdc836928bb2dbdda147ddf677c15eb5f`
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sun May 19 00:23:46 2024 +0800
code refactor
- `commit f0f5399b657acb334f0ae92fc867e4348f24f0a4`
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sun May 19 00:22:31 2024 +0800
code refactor
- `commit 82870d02cbd08926520a606ffa690afbcc80096f (life-path-compare)`
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sun May 19 00:16:08 2024 +0800
compare_lucky_path_number
- `commit 43dc9c764f0b0a99eda8d1746e0e1303c8d7038b`
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sun May 19 00:15:33 2024 +0800
compare_lucky_path_number
- `commit 303d91f576eb067c5a996b574b9af3ad5f94ad75`
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 21:17:44 2024 +0800
minor fix
- `commit e5519653d4e3c0dcc4320d7d07903c33ada957df`
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Sat May 18 21:09:58 2024 +0800
code cleanup
- `commit 3a06e953a8c41c9e0b96c751e6f40afe15660193 (generation)`
Author: Raihan Taher <rayhan.taher@gmail.com>
:|

The status bar at the bottom indicates the current file is 'Mysql'.



The screenshot shows a VS Code editor interface. On the left is the Explorer sidebar showing a project structure with folders like `__pycache__`, `code`, `documents`, and `pictures`, and files like `main.py`, `testCode.py`, `1.png`, `3.png`, `4.png`, `.gitignore`, `test.py`, and `Utils.py`. The main area is the Terminal, which displays a series of git commit messages. The messages are as follows:

```
commit 7187098e16948930763740a1d5aeebe2baaf4c9
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Thu May 23 20:06:23 2024 +0800

    Editing md file

commit 67baa93320e165b2e472a79acb5bc3182372d30f
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Thu May 23 19:52:41 2024 +0800

    updating md

commit 633b650aa5bc6c70160546fbe480a5dc23d59cc1 (test-code)
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Thu May 23 14:19:40 2024 +0800

    added tet codes for Person

commit d8dabd0817e185f1eff41d67b692e63bca72bf81
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Thu May 23 01:29:19 2024 +0800

    designed all test function of Helper Class

commit 932fdb6212a9f3f4c0fe70ac0895e0e69910f98d
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Wed May 22 21:07:33 2024 +0800

    testcode for sumdigit

commit 8ef7b492f4e6bdf0945cdb8f5d29b3962a2847d6
Author: Raihan Taher <rayhan.taher@gmail.com>
Date: Wed May 22 21:06:17 2024 +0800

    minor change

commit 239bab74344f0278bd494c40b3cb90bbd5fcc151
Author: Raihan Taher <rayhan.taher@gmail.com>
```

Discussion

Achievements

Code Implementation

- Developed the **Person** class with methods to calculate the life path number and determine the generation based on birth year.
- Implemented the **Helper** class with utility methods for summing digits, checking master numbers, and comparing life path numbers.
- Created a function to validate and format birthday inputs.

Testing

- Designed test cases using Equivalence Partitioning and Boundary Value Analysis for the appropriate methods.
- Implemented unit tests for the **Person** and **Helper** classes covering various scenarios and edge cases.
- Verified the functionality and robustness of the code through extensive testing.

Documentation

- Provided detailed explanations of the modularity concepts applied in the code.
- Created a markdown file to document everything we have done including test design and implementation facilitating clear understanding and review.

Challenges Faced

Handling Date Inputs

- Ensuring accurate parsing and validation of various date formats was challenging.
- Addressing potential user input errors required robust error handling.

Test Case Design

- Designing comprehensive test cases that cover all possible input scenarios.
- Ensuring the tests accurately reflect real-world inputs and edge cases.

Limitations

Generation Ranges

- The generation ranges are hardcoded and might not cover future generations or changes in generational definitions.
- The "Unknown Generation" category might need more nuanced handling for future years beyond 2024.

Date Handling

- The current implementation only supports specific date formats (`DD-MM-YYYY` and `DD Month YYYY`).
- There is no support for alternative date formats or localization.

User Input

- The `get_valid_birthday_input` method relies on console input, which may not be ideal for all applications.

Ways to Improve

Dynamic Generation Handling

- Implement a more flexible approach to handle future generational ranges dynamically.
- Allow updates to the generation ranges without modifying the core logic.

Enhanced Date Parsing

- Expand support for additional date formats and localization to cater to a wider user base.
- Implement more sophisticated date validation to handle edge cases and invalid dates gracefully.

User Interface

- Develop a graphical user interface (GUI) or web-based interface to improve user experience.
- Provide clearer error messages and guidance for user inputs.

Additional Information

- The project demonstrates the application of software engineering principles, including modularity, encapsulation, and thorough testing.
- The code is designed to be easily extensible, allowing for future enhancements and modifications.
- Detailed documentation and test cases ensure maintainability and ease of understanding for future developers.

By addressing the identified limitations and implementing the suggested improvements, the project can be further refined to provide a more robust and user-friendly experience.

