

Half-Semester Project CIS613

Abstract

I developed a website for my half-semester project that allows users to enter their birthdate or any other date and receive a variety of numerical values and a horoscope in return. This project ([GitHub Link](#)) presents a comprehensive web application designed to offer users insightful information based on their birth date. It calculates the user's age in years, months, weeks, and days, identifies their generation, and provides interesting numerological insights, such as life path numbers. Additionally, it delves into astrology, offering zodiac signs, horoscopes, and compatibility reports with other zodiac signs. The application seamlessly integrates various disciplines, such as astrology, numerology, and demographic studies, to provide a personalized and engaging user experience.

Introduction

In the age of personalized content, individuals seek more than just generic information; they look for insights that resonate on a personal level. This project taps into this desire by combining numerology and astrology to provide users with unique insights based on their birth date. It bridges the gap between complex astrological and numerological calculations and the end-user by presenting this information in an accessible and engaging web application format.



Fig. Demo of the Project

Methodology

How the Project Works

The project consists of a frontend web application and a backend server. The frontend, developed with Vue.js, provides a user-friendly interface where users can input their birth date and a comparison date (usually the current date) to calculate their age. Once the calculation is requested, the frontend sends this data to the backend.

The backend, built with Flask, processes the request by calculating the user's age, determining their zodiac sign, generating a life path number, and fetching their horoscope and zodiac compatibility. It then sends this information back to the frontend, where it is displayed to the user.

Key functionalities include:

- Age Calculation: Computes how old the user is in years, months, weeks, and days.
- Zodiac Sign Identification: Determines the user's zodiac sign based on their birth date.
- Life Path Number Calculation: Calculates the user's life path number, offering insights into their personality and life's journey.
- Horoscope and Zodiac Compatibility: Provides daily horoscopes and compatibility scores with other zodiac signs.

What the Project Does

The project offers a multifaceted approach to understanding oneself through various lenses:

- Astrological Insights: By providing the user's zodiac sign and daily horoscope, the application offers a glimpse into the astrological influences on their day-to-day life and personality.
- Numerological Analysis: The life path number calculation reveals key personality traits and potential life paths, adding another layer of personal insight.
- Demographic Context: The generation identification offers a broader demographic context, connecting the user with broader societal cohorts.
- Compatibility Reports: The zodiac compatibility feature allows users to explore their relationships with others through the astrological compatibility scores, adding a fun and interactive element to the experience.

What are some useful functions, and what do they do?

1. `calculate_age()` : The snippet of code is part of a Flask application's backend route that handles

```
@app.route('/calculate', methods=['POST'])
@cross_origin(origin="localhost:8080")
def calculate_age():
    if not request.is_json:
        print("Request is not JSON!")
        return jsonify({"error": "Missing JSON in request"}), 400

    data = request.get_json()
    print("Received data:", data)
    from_date = datetime.strptime(data['fromDate'], '%Y-%m-%d')
    to_date = datetime.strptime(data['toDate'], '%Y-%m-%d')
    delta = to_date - from_date

    age_years = to_date.year - from_date.year - ((to_date.month, to_date.day) < (from_date.month, from_date.day))
    age_months = age_years * 12 + to_date.month - from_date.month
    age_weeks = delta.days // 7
    age_days = delta.days
    leap_years = count_leap_years(from_date.year, to_date.year)

    zodiac_info = get_zodiac_info(from_date)

    extra_months = to_date.month - from_date.month - (to_date.day < from_date.day)
    extra_days = to_date.day - from_date.day

    if extra_days < 0:
        days_in_prev_month = (to_date - timedelta(days=to_date.day)).day
        extra_days += days_in_prev_month

    if extra_months < 0:
        extra_months += 12

    generation = "Unknown"
    if from_date.year < 1946:
        generation = "The Silent Generation"
    elif from_date.year < 1965:
        generation = "Baby Boomer"
    elif from_date.year < 1981:
        generation = "Generation X"
    elif from_date.year < 1997:
        generation = "Millennials"
    elif from_date.year < 2013:
        generation = "Generation Z"

    birth_month = from_date.month
    birthstone_name, birthstone_url = birthstones[birth_month]

    return jsonify({
        'age': age_years,
        'months': age_months,
        'weeks': age_weeks,
        'days': age_days,
        'leapYears': leap_years,
        'generation': generation,
        'extra_months': extra_months,
        'extra_days': extra_days,
        'birthstone': {
            'name': birthstone_name,
```

the calculation of a user's age based on input dates. When the '/calculate' route receives a POST request with JSON data, it retrieves the 'fromDate' and 'toDate' values, which represent the user's birthdate and the date to calculate the age until, respectively. The

`datetime.strptime` function converts these date strings into `datetime` objects. The age in years is calculated by subtracting the birth year from the current year and adjusting for whether the current date has surpassed the birth date in the current year. Age in months multiplies the years by 12 and adds the difference between the current and birth months. Age In weeks, divide the total days by 7. The `count_leap_years` function calculates the number of leap years experienced by the user, which could be used to further refine the age

calculation. The `get_zodiac_info` function presumably determines the zodiac sign based on the birth date and could provide additional astrological information such as a horoscope and birthstone. Additional code calculates 'extra_months' and 'extra_days' for a more precise age and determines the

user's generation based on the year of birth. Finally, the function returns a JSON object with all these calculated values, including the user's age in years, months, weeks, and days, the number of leap years faced, generation, birthstone name, and URL for more information about the birthstone. This data is likely to be used in the frontend to display it to the user.

Enter your birthday (From Date): 06/11/1998

To Date: 04/09/2024

Calculate

YOUR AGE IS 25 YEARS 9 MONTHS 29 DAYS

Age: 25 years

Age in Months: 298 months

Age in Weeks: 1347 weeks

Age in Days: 9434 days

Leap Years Faced: 7

Your Generation: Generation Z

Birthstone: Pearl

Life Path Number: 8

Fig. How the backend calculate the mathematical terms

2. ``get_zodiac_sign(month, day)``: This function determines a user's zodiac sign based on their birth date. It checks the provided month and day against a predefined range of dates corresponding to each zodiac sign.



Fig. Frontend showing Zodiac Sign, Horoscope and Interesting facts with a link to learn more

3. ``get_zodiac_info(birth_date)``: This function acts as an aggregator, gathering various pieces of zodiac-related information. It uses the user's birth date to determine their zodiac sign, life path number, and zodiac compatibility.

- ``calculate_life_path_number(month, day, year)``: Called within ``get_zodiac_info``, this function calculates the life path number based on the numerology of the user's birth date.
- ``get_zodiac_compatibility(zodiac_sign)``: Also called ``get_zodiac_info``, it retrieves compatibility scores or insights based on the user's zodiac sign.

If a zodiac sign is determined, the function fetches the horoscope, facts about the sign, and a URL for more information from predefined dictionaries (``horoscopes`` and ``facts_about_signs``). It then bundles this information into a JSON-like dictionary, including the life path number and compatibility information, which is likely sent back to the frontend for display. If the zodiac sign is unknown (e.g., if the birth date does not match any predefined zodiac date ranges), it defaults to returning a dictionary with values set to "Unknown" or "No horoscope available."

The frontend portion (not shown in the snippet but implied), would display this information appropriately, likely under a section titled "Zodiac Sign" or similar. This functionality enriches the web app by providing personalized zodiac and numerology information based on the user's inputted birth date.

```
def get_zodiac_sign(month, day):
    for zodiac, (start_date, end_date) in zodiac_dates.items():
        if (month == start_date[0] and day >= start_date[1]) or \
            (month == end_date[0] and day <= end_date[1]):
            return zodiac
    return "Unknown"

def get_zodiac_info(birth_date):
    month = birth_date.month
    day = birth_date.day
    year = birth_date.year
    zodiac_sign = get_zodiac_sign(month, day)
    life_path_number = calculate_life_path_number(month, day, year)
    compatibility_info = get_zodiac_compatibility(zodiac_sign)

    if zodiac_sign:
        horoscope = horoscopes.get(zodiac_sign, "No horoscope available.")
        facts = facts_about_signs.get(zodiac_sign, "No facts available.")
        more_info_url = f"https://en.wikipedia.org/wiki/{zodiac\_sign}\_\"\(astrology\)\""

        return {
            'sign': zodiac_sign,
            'horoscope': horoscope,
            'facts': facts,
            'more_info_url': more_info_url,
            'life_path_number': life_path_number,
            'compatibility': compatibility_info
        }
    else:
        return {
            'sign': "Unknown",
            'horoscope': "No horoscope available.",
            'facts': "No facts available.",
            'more_info_url': "https://en.wikipedia.org/wiki/Zodiac"
        }
```

Fig. Backend of determining Zodiac Sign and Zodiac Info

Tests Conducted

Unittest

Conducting unittests, especially for critical functions like ``calculate_life_path_number`` and ``count_leap_years``, is an essential practice in software development that ensures the reliability and accuracy of the application's functionality. In this project, the unittests serve multiple purposes and bring significant benefits, contributing to the overall quality and robustness of the web application.

The core appeal of this project lies in its ability to provide users with insightful and accurate information based on their birth dates. Unittests ensure that the calculations for life path numbers and leap years are performed correctly, guaranteeing the integrity of the insights provided to the user. By systematically testing each function with a variety of inputs, potential errors or edge cases that could lead to incorrect outputs are identified early. This proactive approach helps in preventing user-facing issues that could detract from the user experience. As the project evolves, new features may be added, or existing code may be refactored. Unittests serve as a safety net, ensuring that changes in the codebase do not inadvertently break existing functionality. Writing unittests encourages a modular and decoupled design, as functions need to be testable in isolation. This not only leads to cleaner code but also speeds up the development process by allowing for quick checks on the functionality of individual components.

The tests contribute directly to the reliability of the application by ensuring that key functions perform as expected under various scenarios. This reliability is critical for an application that users rely on for personal insights. The process of writing tests often leads to discovering better ways to structure the code or optimize algorithms, leading to overall improvements in code quality. By ensuring the accuracy of the calculations and predictions, unittests indirectly contribute to user trust and satisfaction. Users are more likely to engage with and recommend the application when they can rely on its outputs. As the project grows, the tests will serve as a foundation for safely integrating new features or making changes. They reduce the risk of introducing bugs that could undermine the user experience.

In conclusion, conducting unittests for the ``calculate_life_path_number`` and ``count_leap_years`` functions has played a pivotal role in enhancing the project's quality, reliability, and user trust. It sets a solid foundation for future development, ensuring that the application can evolve without compromising on the accuracy and integrity of the information it provides.

The screenshot shows the PyCharm IDE interface. The left sidebar displays the project structure for 'HOROSCOPE-BIRTHDAY', with the 'test.py' file selected under the 'Backend' directory. The main editor window shows the code for the 'TestApp' class, which inherits from 'unittest.TestCase'. It contains two test methods: 'test_calculate_life_path_number' and 'test_count_leap_years'. The 'test_calculate_life_path_number' method uses 'datetime' objects to test the 'calculate_life_path_number' function with various birth dates. The 'test_count_leap_years' method tests the 'count_leap_years' function with various year ranges. The bottom panel shows the 'TERMINAL' output, indicating that 2 tests were run successfully in 0.000s.

```

5 class TestApp(unittest.TestCase):
7     def test_calculate_life_path_number(self):
8         test_cases = [
9             (datetime(1990, 12, 25), 2), # 1+9+9+0+1+2+2+5 = 29 -> 2+9 = 11 -> 1+1 = 2
10            (datetime(2000, 1, 1), 4), # 2+0+0+0+1+1+1 = 4
11            (datetime(1985, 10, 23), 2), # 1+9+8+5+1+0+2+3 = 29 -> 2+9 = 11 -> 1+1 = 2
12            (datetime(2000, 1, 1), 4),
13            (datetime(1995, 5, 17), 1), # 1+9+9+5+5+1+7 = 37 -> 3+7 = 10 -> 1+0 = 1
14            (datetime(1988, 12, 29), 4), # 1+9+8+8+1+2+2+9 = 40 -> 4+0 = 4
15            (datetime(2002, 2, 2), 8), # 2+0+0+2+2+2 = 8
16            (datetime(1990, 6, 25), 5), # 1+9+9+0+6+2+5 = 32 -> 3+2 = 5
17            (datetime(2021, 3, 7), 6), # 2+0+2+1+3+7 = 15 -> 1+5 = 6
18            (datetime(1964, 7, 12), 3), # 1+9+6+4+7+1+2 = 30 -> 3+0 = 3
19            (datetime(1975, 11, 5), 2), # 1+9+7+5+1+1+5 = 29 -> 2+9 = 11 -> 1+1 = 2
20            (datetime(1942, 8, 23), 2), # 1+9+4+2+8+2+3 = 29 -> 2+9 = 11 -> 1+1 = 2
21            (datetime(1983, 4, 14), 3), # 1+9+8+3+4+1+4 = 30 -> 3+0 = 3
22            (datetime(2007, 9, 18), 9), # 2+0+0+7+9+1+8 = 27 -> 2+7 = 9
23            (datetime(2012, 12, 21), 2), # 2+0+1+2+1+2+2+1 = 11 -> 1+1 = 2
24            (datetime(1980, 3, 5), 8), # 1+9+8+0+3+5 = 26 -> 2+6 = 8
25            (datetime(1945, 5, 29), 8), # 1+9+4+5+5+2+9 = 35 -> 3+5 = 8
26            (datetime(1999, 1, 1), 3), # 1+9+9+9+1+1 = 30 -> 3+0 = 3
27        ]
28        for birth_date, expected in test_cases:
29            self.assertEqual(calculate_life_path_number(birth_date.month, birth_date.day), expected)
30
31    def test_count_leap_years(self):
32        test_cases = [
33            (2000, 2020, 6), # 2000, 2004, 2008, 2012, 2016, 2020
34            (1990, 2000, 3), # 1992, 1996, 2000
35            (1985, 1985, 0), # No leap year
36            (2000, 2020, 6),
37            (1990, 2000, 3),
38            (1985, 1985, 0),
39            (1960, 1969, 3),
40            (1900, 2000, 25),
41            (1800, 1900, 24),
42            (1600, 1700, 25), # 1600 is a leap year
43            (2001, 2021, 5),
44            (2020, 2020, 1), # Only one leap year
45            (2000, 2004, 2), # Two leap years, including a century leap year
46            (1996, 2000, 2),
47            (1988, 1992, 2),
48            (1952, 1960, 3),
49            (1904, 1912, 3),
50            (1888, 1896, 3),
51        ]
52        for year1, year2, expected in test_cases:
53            self.assertEqual(count_leap_years(year1, year2), expected)

```

TERMINAL

```

..
Ran 2 tests in 0.000s

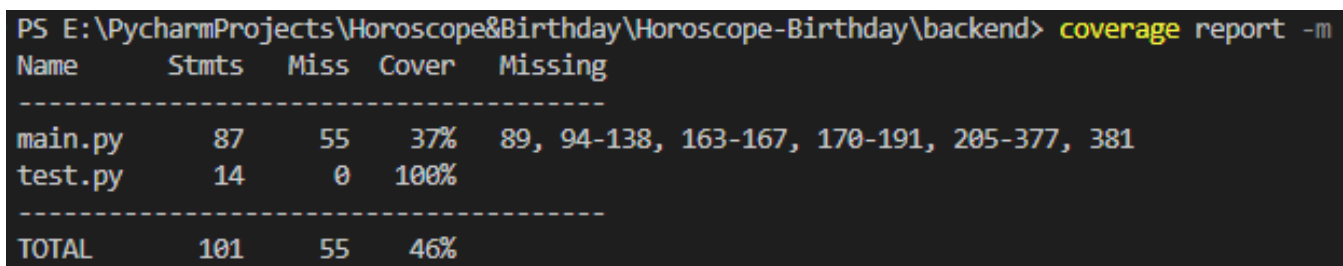
OK
PS E:\PycharmProjects\Horoscope&Birthday\Horoscope-Birthday\backend>

```

Fig. Unittest

Code Coverage

Code coverage reveals untested code areas, guiding test development to improve overall code quality and prevent production bugs. It ensures thorough testing, covering edge cases and failure paths, which reduces the cost and complexity of addressing bugs found later in production. High test coverage equates to stable code, boosting confidence in the application's reliability. As a form of documentation, it highlights tested areas and potential risk zones, helping teams focus their testing efforts effectively. By increasing test coverage, developers can safeguard new features and refactoring, enhance the application's robustness and user experience, and prevent the accrual of technical debt.



```
PS E:\PycharmProjects\Horoscope&Birthday\Horoscope-Birthday\backend> coverage report -m
```

Name	Stmts	Miss	Cover	Missing
main.py	87	55	37%	89, 94-138, 163-167, 170-191, 205-377, 381
test.py	14	0	100%	
TOTAL	101	55	46%	

Fig. Code Coverage Report

The coverage test reveals areas that the current tests do not cover. This could mean that more test cases need to be added to make sure these parts of the code are also checked. By confirming that every functional aspect of the application has been tested and verified to operate as intended, this can potentially result in improved output.

After enhancing the test suite with additional test cases for the ``get_zodiac_sign`` function, the code coverage metric has improved from 37% to 40%. This increase indicates that a greater proportion of the codebase has been executed during tests, reducing the likelihood of undetected bugs. A coverage of 40% is a significant indicator that the application is now better vetted, though there's still room for improvement to ensure even more comprehensive testing. It is essential to aim for higher coverage to enhance confidence in the application's stability and minimize the risk of regression issues. Adding tests for uncovered sections could further safeguard the application against potential defects, promoting a reliable and consistent user experience. The improved coverage also provides a more in-depth understanding of the application's test coverage state, which can guide future testing efforts.

```
PS E:\PycharmProjects\Horoscope&Birthday\Horoscope-Birthday\backend> coverage report -m
```

Name	Stmts	Miss	Cover	Missing
main.py	87	52	40%	89, 94-138, 167, 170-191, 205-377, 381
test.py	19	0	100%	
TOTAL	106	52	51%	

Fig. FINAL Code Coverage Report

Mutation Testing

Mutation testing is a method that helps us assess the quality of our test cases. It does this by making small changes to the code, called mutations, to see if the tests can detect the changes. A low mutation score indicates that the test suite may have flaws because many of these mutations went undetected by the current tests. Although it is true that achieving a 100% mutation score may not always be necessary for simple applications like a birthday reminder web application, where direct data retrieval and processing are involved. However, the risk of settling for a low mutation score is that certain edge cases or error handling paths could remain untested. This might not be a significant concern for non-critical applications, but it's still a risk that needs acknowledgment. Moreover, neglecting these surviving mutants might mean missing out on opportunities to improve the code's robustness against future changes. As the application grows in complexity, the gaps in the test coverage might become more problematic. Ensuring that important functionalities are thoroughly tested by addressing these surviving mutants can provide a stronger safety net for future development. Therefore, even though the current test cases seem sufficient for the primary functionalities of the project, I should still consider the risks of having a low mutation score. Reviewing and refining the test cases to improve the mutation score can lead to better test coverage, a more reliable application, and a higher quality user experience. It's about balancing the cost and benefits of additional testing with the potential risks of undetected issues in the application.

```
⌵ 704/708 🚀 19 🕒 0 😬 0 😬 685 🚫 0  
DevTools listening on ws://127.0.0.1:56536/devtools/browser/1cec4e26-49d2-4eba-88f9-97feb5753a73  
⌵ 705/708 🚀 19 🕒 0 😬 0 😬 686 🚫 0  
DevTools listening on ws://127.0.0.1:56620/devtools/browser/3b4fab54-9b9b-412c-8e71-5d4e755ddda9  
⌵ 706/708 🚀 19 🕒 0 😬 0 😬 687 🚫 0  
DevTools listening on ws://127.0.0.1:56690/devtools/browser/f16f628a-fd0e-49f4-9513-ee1f4b0514d4  
⌵ 707/708 🚀 19 🕒 0 😬 0 😬 688 🚫 0  
DevTools listening on ws://127.0.0.1:56759/devtools/browser/41bdcd0e-9169-421c-85ba-7d18d4111689  
⌵ 708/708 🚀 19 🕒 0 😬 0 😬 689 🚫 0
```

```
(.venv) PS E:\PycharmProjects\Horoscope&Birthday\horoscope-birthday\backend> mutmut results
To apply a mutant on disk:
  mutmut apply <id>

To show a mutant:
  mutmut show <id>

Survived 🤖 (689)

---- src\main.py (689) ----
```

Fig. Mutation Testing

Boundary Value Testing

Boundary value testing is particularly effective for some functions because it targets the edges of input ranges, where errors are more likely to occur. By testing the boundaries for `calculate_life_path_number`, `count_leap_years`, and `get_zodiac_sign`, I focused on the most crucial and vulnerable points in the input domain. This type of testing is efficient as it provides a high probability of finding errors with a minimal number of test cases. For `calculate_life_path_number`, which likely involves numerical calculations based on dates, boundary values help to ensure that the logic correctly handles edge cases like end-of-year or leap year dates. This function is fundamental to the app's core feature of numerology analysis, making it critical to validate thoroughly. `count_leap_years` is an essential function for accurately calculating age, and leap years are a classic example of a boundary condition in date calculations. By testing years around known leap years and century years, I ensured the function's reliability in adjusting for leap days. `get_zodiac_sign` determines astrological information based on birthdates, and since zodiac signs change on specific dates, boundary value testing ensures that the transition from one sign to the next is handled correctly. Testing the dates right at the cusp of zodiac transitions helps to validate that users are given the correct horoscope information. These tests are sufficient for the project as they cover the most significant risk areas in the input domain without requiring an exhaustive testing approach. By strategically choosing boundary values, I can efficiently detect flaws in the logic that could lead to incorrect outputs, thereby enhancing the reliability of the application.

```

5 class BoundaryValueTest(unittest.TestCase):
7     def test_calculate_life_path_number_boundaries(self):
9         test_cases = [
10             (datetime(1999, 12, 31), 8),
11             (datetime(2000, 1, 1), 4),
12             (datetime(2000, 1, 1), 4),
13             (datetime(2001, 1, 1), 5),
14             (datetime(1980, 1, 1), 2),
15             (datetime(1980, 1, 2), 3),
16             (datetime(1982, 12, 31), 9),
17             (datetime(1983, 1, 1), 5),
18             (datetime(1990, 4, 22), 9),
19             (datetime(1990, 4, 23), 1),
20             (datetime(2000, 12, 31), 9),
21             (datetime(2001, 1, 1), 5),
22             (datetime(2010, 5, 5), 4),
23             (datetime(2010, 5, 6), 5),
24         ]
25         for birth_date, expected in test_cases:
26             self.assertEqual(calculate_life_path_number(birth_date.month, birth_date.day, birth_date.year), expected)
27
28     def test_count_leap_years_boundaries(self):
29         test_cases = [
30             (1899, 1901, 0),
31             (2000, 2004, 2),
32             (1900, 1904, 1),
33             (2096, 2104, 2),
34             (1896, 1904, 2),
35             (1996, 2004, 3),
36             (1900, 1901, 0), # No leap year in this range
37             (1999, 2001, 1), # The year 2000 is a leap year
38             (2003, 2005, 1), # The year 2004 is a leap year
39             (2096, 2104, 2), # Century leap year not included
40             (2099, 2101, 0), # No leap year
41             (2100, 2105, 1), # Only 2104 is a leap year
42             (1992, 2000, 3), # 1992, 1996, and 2000 are leap years
43         ]
44         for year1, year2, expected in test_cases:
45             self.assertEqual(count_leap_years(year1, year2), expected)
46
47     def test_get_zodiac_sign_boundaries(self):
48         test_cases = [
49             ((3, 20), 'Pisces'), # The last day of Pisces
50             ((3, 21), 'Aries'), # The first day of Aries
51             ((4, 19), 'Aries'), # The last day of Aries
52             ((4, 20), 'Taurus'), # The first day of Taurus
53             ((5, 20), 'Taurus'), # The last day of Taurus
54             ((5, 21), 'Gemini'), # The first day of Gemini
55             ((6, 20), 'Gemini'), # The last day of Gemini
56             ((6, 21), 'Cancer'), # The first day of Cancer
57             ((7, 22), 'Cancer'), # The last day of Cancer
58             ((7, 23), 'Leo'), # The first day of Leo
59             ((8, 22), 'Leo'), # The last day of Leo
60             ((8, 23), 'Virgo'), # The first day of Virgo
61             ((9, 22), 'Virgo'), # The last day of Virgo
62             ((9, 23), 'Libra'), # The first day of Libra
63             ((10, 22), 'Libra'), # The last day of Libra
64             ((10, 23), 'Scorpio'), # The first day of Scorpio
65             ((11, 21), 'Scorpio'), # The last day of Scorpio
66             ((11, 22), 'Sagittarius'), # The first day of Sagittarius
67             ((12, 21), 'Sagittarius'), # The last day of Sagittarius
68             ((12, 22), 'Capricorn'), # The first day of Capricorn

```

PROBLEMS OUTPUT **TERMINAL** PORTS GITLENS DEBUG CONSOLE

```

(.venv) PS E:\PycharmProjects\Horoscope&Birthday\horoscope-birthday\backend> py boundarytest.py
...
Ran 3 tests in 0.000s
OK

```

Fig. Boundary Value Testing

Fig. Selenium

Why do these tests seem to be enough?

Here's how I came up with the tests I have conducted -

1. Understanding the project's logic
2. Determining the scoop of testing
3. Testing user interactions
4. Ensuring comprehensive coverage
5. Evaluating test effectiveness

Unittests are the foundation, ensuring each module of the backend operates correctly in isolation. They are especially vital in a deterministic system like mine, where functions are expected to yield consistent results for given inputs.

Code coverage analysis complements Unittest by highlighting which parts of the code are actually being executed during tests. This helps to identify untested paths, leading to a more thoroughly tested application. In my case, with a coverage exceeding 40%, it suggests that the code affects the test outcomes, indicative of a well-tested application core. The rest of the code is usually string based dictionary, and it can't be tested as it is constant.

Selenium tests ensure that the frontend, the user's main touchpoint, integrates seamlessly with the backend logic. These tests simulate real user behavior, validating that the user interface is responsive and functions correctly across different scenarios.

Boundary value testing is paramount for functions like ``calculate_life_path_number``, ``count_leap_years``, and `'get_zodiac_sign'`, which have well-defined input ranges and can be prone to errors at the edges. By focusing on these critical junctures, I have ensured that transitions between different zodiac signs and age calculations are accurate, which is crucial for a horoscope related application like mine.

Mutation testing gauges the effectiveness of the tests by introducing changes ('mutants') to the code and checking if the tests detect them. While a low mutation score can be alarming, in my project, it highlights the areas where tests could be further strengthened. Nevertheless, mutation testing has fulfilled its role by helping me identify potential weaknesses in the test suite.

Each testing method targets different aspects of the application, from functional correctness and comprehensive code assessment to real-world user interaction and edge-case robustness. While no testing strategy can guarantee absolute perfection, the combination I've employed covers the critical facets necessary for the application's integrity and user satisfaction. In the context of the project, which involves date manipulations and predefined outputs, these tests are everything that's needed because they validate that the core features—age calculation, zodiac determination, and horoscope provision—are functioning correctly. Any further testing, like exploratory or life cycle-based testing, might not yield additional significant insights given the straightforward nature of the web app's functionality and the thoroughness of the current testing strategy.

Why do some other tests not make sense?

Metamorphic testing is often used when it's difficult to define clear expected outcomes for certain tests or to verify non-functional aspects of the application, like performance under stress. My project doesn't seem to require this level of testing, as the expected behaviors are well-defined and predictable.

Life Cycle testing is critical for applications with complex stages of development and deployment. This project's backend and frontend components are relatively simple and are validated through unit, coverage, and frontend testing. The straightforward development cycle and deployment process do not have multiple stages that evolve over time, which life cycle based testing typically addresses.

Exploratory testing relies on the tester's knowledge, experience, creativity, and intuition. This project has been designed with a clear set of functional requirements and expected behaviors, which have been covered by automated tests. Exploratory testing is often used for applications where requirements are unclear or continuously evolving, or to uncover unusual or unexpected behaviors. Given the deterministic nature of the app's functionality, this form of testing provides little additional value.

Given the nature of my application, the current testing strategies are likely sufficient to ensure its reliability and correctness without needing the granular level of scrutiny provided by mutation testing. Overall, the testing methodologies I've chosen are apt for the scope and scale of my project, ensuring functionality, reliability, and a good user experience without the overhead of more complex testing strategies that might not yield proportional benefits for my use case.

Conclusion

This project successfully merges astrology, numerology, and demographic insights into a single, cohesive web application. It not only serves as an entertaining tool but also encourages users to explore the deeper aspects of their personalities and relationships. By providing personalized information in an accessible manner, it caters to the growing demand for customized content and self-exploration tools.

References

1. Selenium. (n.d.). Selenium documentation. <https://www.selenium.dev/documentation/>
2. Fowler, M. (n.d.). UnitTest. Martin Fowler. <https://martinfowler.com/bliki/UnitTest.html>
3. Mozilla Developer Network. (n.d.). Web technology for developers. MDN Web Docs. <https://developer.mozilla.org/>
4. Vue.js. (n.d.). Introduction. Vue.js. <https://vuejs.org/v2/guide/>
5. Astrology.com. (n.d.). Horoscopes. <https://www.astrology.com/horoscope/daily.html>
6. Cafe Astrology. (n.d.). Cafe Astrology .com. <https://cafeastrology.com/>
7. Decoz,H.(n.d.).Numerology.com. <https://www.numerology.com/numerology-news/life-path-number>