Code Review Challenge for Absent Students

(Roshankumar Loganathan rlogana@g.clemson.edu)

Introduction

The provided Python script is designed to generate a Fibonacci sequence up to a specified number of terms, identify prime numbers within this sequence, and display these primes to the user. The script is functional but presents opportunities for optimization and enhancement in terms of efficiency, readability, and maintainability.

Original Code Analysis

Functionality

The script has three main functions:

calculate_fibonacci_sequence(n):

- Generates the Fibonacci sequence up to the n-th term.

is_prime(num):

- Checks if a given number is prime.

filter_primes_in_fibonacci_sequence(sequence):

- Filters out prime numbers from a given Fibonacci sequence.

These functions are orchestrated in the main() function, which handles user input and displays the results.

Error Handling

- The script includes basic error handling for invalid (non-positive) inputs.

Efficiency

While the script is functional, there are inefficiencies:

1. The Fibonacci sequence generation is not optimized for large n.
2. The prime checking function is inefficient as it iterates through all numbers up to num.
3. Prime numbers are checked multiple times in the sequence without any caching mechanism.
4. Readability and Maintainability
5. The code is well-structured, and functions are descriptively named. Docstrings are provided, enhancing maintainability. However, there are opportunities to improve readability through more concise code constructs.

Refactoring Process

1. Optimizing Fibonacci Sequence Generation

The original function created the sequence in a straightforward manner. To optimize this:

- Used list slicing and a more concise loop to generate the sequence.
- Ensured that the function handles edge cases (like n <= 0) effectively.

2. Enhancing Prime Checking Efficiency

The is_prime function was sub-optimal as it checked all numbers up to num. Improvements:

- Implemented a check up to the square root of num, significantly reducing the number of iterations.
- Used math.sqrt for a more mathematically sound approach.

3. Implementing Caching for Prime Checks

To avoid redundant calculations:

- Introduced a dictionary (prime_cache) to store the results of prime checks.
- Modified filter_primes_in_fibonacci_sequence to utilize this cache, reducing the computational load.

4. Code Readability and Conciseness

- Applied list comprehensions where appropriate for more concise and readable code.
- Ensured variable names are descriptive, maintaining readability.

5. Maintaining Error Handling and Input Validation

- Preserved the existing error handling and input validation logic in the main function.
- Ensured that the refactored code continues to handle edge cases and invalid inputs effectively.

Conclusion

The refactored code maintains the original functionality while achieving greater efficiency, particularly in generating the Fibonacci sequence and identifying prime numbers. The introduction of a caching mechanism for prime checks significantly reduces redundant computations. These changes, along with a focus on concise and readable code, enhance the script's maintainability and performance.