

Prompts Lab

Automated Code Reviews:

Instruction: Analyze the following code snippet and provide suggestions for improvements. Focus on optimizing the performance, refactoring for better readability and maintainability, or identifying potential bugs.

Input	Output
<pre>def find_max(numbers): max_num = numbers[0] for num in numbers: if num > max_num: max_num = num return max_num</pre>	<p>Optimisation: You can use Python's built-in max() function to improve the efficiency of finding the maximum number:</p> <pre>def find_max(numbers): return max(numbers)</pre> <p>Bug Check: In the original function, if the input list is empty, it will raise an IndexError. Consider adding input validation:</p> <pre>def find_max(numbers): if not numbers: return None # or raise an appropriate exception return max(numbers)</pre>
<pre>function addNumbers(a, b) { if (a === undefined b === undefined) { return "Error"; } return a + b; }</pre>	<p>Refactoring: Instead of manually checking for undefined, you can provide default values for the parameters:</p> <pre>function addNumbers(a = 0, b = 0) { return a + b; }</pre> <p>Bug Check: If the input is not a number, consider adding type validation:</p> <pre>function addNumbers(a = 0, b = 0) { if (typeof a !== 'number' typeof b !== 'number') { return "Error: Both inputs must be numbers"; } return a + b; }</pre>

<pre> public class Calculator { public int multiply(int a, int b) { int result = 0; for (int i = 0; i < b; i++) { result += a; } return result; } } </pre>	<p>Optimization: You can simplify the multiplication by using the built-in multiplication operator:</p> <pre> public class Calculator { public int multiply(int a, int b) { return a * b; } } </pre> <p>Bug Check: Consider handling cases where b is negative to ensure correct behavior:</p> <pre> public class Calculator { public int multiply(int a, int b) { return a * b; // Handle edge cases for negative b if needed } } </pre>
---	---

Testing my Prompts:

Input	Output
<pre> def sum_numbers(numbers): total = 0 for num in numbers: total += num return total </pre>	<p>Refactoring: You can use the built-in sum() function to make the code more concise:</p> <pre> def sum_numbers(numbers): return sum(numbers) </pre> <p>Bug Check: If the input is not a list, consider adding type validation:</p> <pre> def sum_numbers(numbers): if not isinstance(numbers, list): raise TypeError("Input must be a list") return sum(numbers) </pre>
<pre> function isEven(number) { if (number % 2 == 0) { return true; } else { return false; } } </pre>	<p>Refactoring: You can simplify the function by using the modulus operator:</p> <pre> function isEven(number) { return number % 2 == 0; } </pre> <p>Bug Check: Consider handling negative numbers, zero, and non-integer inputs:</p> <pre> function isEven(number) { if (typeof number !== 'number') { return "Error: Input must be a number"; } return number % 2 == 0; } </pre>

<pre> public class Factorial { public int calculateFactorial(int n) { int result = 1; for (int i = 1; i <= n; i++) { result = result * i; } return result; } } </pre>	<p>Refactoring: Consider using a recursive approach to calculate the factorial:</p> <pre> public class Factorial { public int calculateFactorial(int n) { if (n == 0) { return 1; } return n * calculateFactorial(n - 1); } } </pre> <p>Bug Check: Consider handling negative input to ensure correct behavior:</p> <pre> public class Factorial { public int calculateFactorial(int n) { if (n < 0) { throw new IllegalArgumentException("Input must be a non-negative integer"); } // Recursive calculation } } </pre>
--	---

Code Document Generation

Instruction: Analyze the following function or class and generate detailed documentation. The documentation should include:

1. A clear description of what the function or class does.
2. The purpose and data types of the input arguments.
3. The return value and its data type.
4. Any exceptions or errors that could be raised.
5. An example of how to use the function or class in practice.
6. For classes, also document each method separately, including a brief explanation of the class itself.”

Example Inputs:

Input	Output
-------	--------

<pre>def divide_numbers(a, b): return a / b</pre>	<pre>def divide_numbers(a, b): """ Divide two numbers and return the result. Args: a (float): The dividend (numerator). b (float): The divisor (denominator), must be non- zero. Returns: float: The result of dividing a by b. Raises: ZeroDivisionError: If b is zero. Example: >>> divide_numbers(10, 2) 5.0 >>> divide_numbers(10, 0) ZeroDivisionError: division by zero """ return a / b</pre>
<pre>function findMax(arr) { let max = arr[0]; for (let i = 1; i < arr.length; i++) { if (arr[i] > max) { max = arr[i]; } } return max; }</pre>	<pre>/** * Find the maximum value in an array of numbers. * * @param {number[]} arr - An array of numbers to find the maximum value from. * @returns {number} The maximum value in the array. * * @throws {Error} If the array is empty. * * Example: * findMax([1, 3, 2, 5, 4]); // returns 5 * findMax([]); // throws Error: Array is empty */ function findMax(arr) { if (arr.length === 0) { throw new Error("Array is empty"); } let max = arr[0]; for (let i = 1; i < arr.length; i++) { if (arr[i] > max) { max = arr[i]; } } return max; }</pre>

```

class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount
        return self.balance

```

```

class BankAccount:
    """
    A class to represent a bank account.

    Attributes:
        owner (str): The name of the account owner.
        balance (float): The account balance, initialized to 0 by default.

    Methods:
        deposit(amount): Add funds to the account and return the new balance.
        withdraw(amount): Withdraw funds from the account and return the new
        balance.
    """

    def __init__(self, owner, balance=0):
        """
        Initialize the BankAccount with an owner and an optional initial balance.

        Args:
            owner (str): The name of the account owner.
            balance (float, optional): The initial account balance. Defaults to 0.

        Example:
            >>> account = BankAccount('John', 100)
            >>> account.owner
            'John'
            >>> account.balance
            100
        """
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        """
        Deposit the specified amount into the account and return the new balance.

        Args:
            amount (float): The amount to deposit.

        Returns:
            float: The updated account balance.

        Example:
            >>> account.deposit(50)
            150
        """
        self.balance += amount
        return self.balance

    def withdraw(self, amount):
        """
        Withdraw the specified amount from the account.

        Args:
            amount (float): The amount to withdraw.

        Returns:
            float: The updated account balance.

        Raises:
            ValueError: If the amount is greater than the available balance.

        Example:
            >>> account.withdraw(50)
            100
            >>> account.withdraw(200)
            ValueError: Insufficient funds
        """
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount
        return self.balance

```

```

public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }

    public double getCircumference() {
        return 2 * Math.PI * radius;
    }
}

```

```

/**
 * A class to represent a circle and calculate its area and
 * circumference.
 *
 * Attributes:
 *   radius (double): The radius of the circle.
 *
 * Methods:
 *   getArea(): Calculate and return the area of the circle.
 *   getCircumference(): Calculate and return the
 *   circumference of the circle.
 *
 * Example:
 *   Circle circle = new Circle(5);
 *   circle.getArea(); // returns 78.53981633974483
 *   circle.getCircumference(); // returns
 *   31.41592653589793
 */
public class Circle {
    private double radius;

    /**
     * Constructs a Circle object with the specified radius.
     *
     * @param radius The radius of the circle.
     */
    public Circle(double radius) {
        this.radius = radius;
    }

    /**
     * Calculate and return the area of the circle.
     *
     * @return The area of the circle.
     */
    public double getArea() {
        return Math.PI * radius * radius;
    }

    /**
     * Calculate and return the circumference of the circle.
     *
     * @return The circumference of the circle.
     */
    public double getCircumference() {
        return 2 * Math.PI * radius;
    }
}

```

API Integration Code Generater

Instruction: Generate boilerplate code for integrating with the specified IBM API or service. The generated code should:

1. Include necessary authentication and authorization steps.
2. Provide the basic structure for making API requests (GET, POST, PUT, DELETE, etc.).
3. Handle potential errors or exceptions, including response code checking.
4. Use the appropriate libraries (e.g., requests for Python, fetch for JavaScript).
5. Demonstrate how to send data with the request (e.g., in JSON format).
6. Include an example of how to parse and handle the response.
7. Provide detailed comments explaining each part of the integration process.
8. Make sure the code is functional and ready to use

Example Inputs and Outputs:

Input	Output
-------	--------

I'm integrating with IBM's Watson Text-to-Speech API to convert text into speech. Here mentioned are my basic requirements

Input:

API: IBM Watson Text-to-Speech

Method: POST

Authentication: API Key

Data format: JSON

Sample Text: "Hello, world!"

```
import requests
import json

def convert_text_to_speech(api_key, text):
    """
    Converts the provided text to speech using IBM Watson Text-to-Speech API.

    Args:
        api_key (str): The API key for authenticating with the IBM Watson API.
        text (str): The text to be converted to speech.

    Returns:
        Response object: The response containing the audio data or an error message.

    Example:
        api_key = "your_ibm_api_key_here"
        text = "Hello, world!"
        response = convert_text_to_speech(api_key, text)
        if response.status_code == 200:
            with open("output_audio.wav", "wb") as audio_file:
                audio_file.write(response.content)
        else:
            print(f"Error: {response.status_code} - {response.json()}")
    """

    # Endpoint for Watson Text-to-Speech
    url = "https://api.us-south.text-to-speech.watson.cloud.ibm.com/v1/synthesize"

    # Headers for the request
    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {api_key}"
    }

    # Data to be sent in JSON format
    data = json.dumps({
        "text": text,
        "voice": "en-US_AllisonV3Voice", # Specify the voice
        "accept": "audio/wav"
    })

    # Making the POST request
    response = requests.post(url, headers=headers, data=data)

    # Handle potential errors
    if response.status_code == 200:
        print("Request successful, writing audio to file...")
        return response
    else:
        print(f"Error: {response.status_code}")
        return response
```


I'm integrating with IBM Watson Assistant API to send user input and get a response. Generate a boilerplate with following requirements

Input:

API: IBM Watson Assistant

Method: POST

Authentication: API Key

Assistant ID: "your_assistant_id"

User Query: "What is the weather today?"

```
async function sendMessageToAssistant(apiKey, assistantId, userInput) {  
  /**  
   * Sends a user query to IBM Watson Assistant and receives a response.  
   *  
   * Args:  
   *   apiKey (string): The API key for authenticating with IBM Watson Assistant.  
   *   assistantId (string): The unique identifier of the Assistant.  
   *   userInput (string): The user query to be sent to the Assistant.  
   *  
   * Returns:  
   *   JSON object: The response from the Assistant or an error message.  
   *  
   * Example:  
   *   sendMessageToAssistant("your_api_key", "your_assistant_id", "What is the  
weather today?")  
   *   .then(response => console.log(response))  
   *   .catch(error => console.error("Error:", error));  
   */  
  
  const url = `https://api.us-south.assistant.watson.cloud.ibm.com/instances/  
{your_instance_id}/v2/assistants/${assistantId}/sessions`;  
  
  // Headers for the request  
  const headers = {  
    'Content-Type': 'application/json',  
    'Authorization': `Bearer ${apiKey}`  
  };  
  
  // Creating a session  
  let sessionResponse;  
  try {  
    sessionResponse = await fetch(url, {  
      method: 'POST',  
      headers: headers  
    });  
  
    if (!sessionResponse.ok) {  
      throw new Error(`Error creating session: ${sessionResponse.statusText}`);  
    }  
  
    const sessionData = await sessionResponse.json();  
    const sessionId = sessionData.session_id;  
  
    // Sending user input  
    const messageUrl = `${url}/${sessionId}/message`;   
    const requestBody = JSON.stringify({  
      input: {  
        message_type: 'text',  
        text: userInput  
      }  
    });  
  
    const response = await fetch(messageUrl, {  
      method: 'POST',  
      headers: headers,  
      body: requestBody  
    });  
  
    if (!response.ok) {  
      throw new Error(`Error sending message: ${response.statusText}`);  
    }  
  
    const responseData = await response.json();  
    return responseData;  
  
  } catch (error) {  
    console.error("Error:", error);  
  }  
}
```

I'm integrating with IBM Cloud Object Storage to upload a file. Generate a boilerplate for the below mentioned credentials

API: IBM Cloud Object Storage

Method: PUT

Authentication: API Key

Bucket Name: "your_bucket_name"

File Path: "path/to/your/file.txt"

import requests

```
def upload_file_to_cos(api_key, bucket_name, file_path):
```

```
    """
```

```
    Uploads a file to IBM Cloud Object Storage (COS).
```

```
    Args:
```

```
        api_key (str): The API key for authenticating with IBM COS.
```

```
        bucket_name (str): The name of the COS bucket where the file will be uploaded.
```

```
        file_path (str): The local path to the file that will be uploaded.
```

```
    Returns:
```

```
        Response: The response object from the server indicating success or failure.
```

```
    Example:
```

```
        upload_file_to_cos("your_ibm_api_key", "your_bucket_name", "path/to/your/file.txt")
```

```
    """
```

```
    # Endpoint for COS upload
```

```
    url = f"https://s3.us-south.cloud-object-storage.appdomain.cloud/{bucket_name}/{file_path.split('/')[-1]}"
```

```
    # Headers for authentication
```

```
    headers = {
```

```
        "Authorization": f"Bearer {api_key}",
```

```
        "Content-Type": "application/octet-stream"
```

```
    }
```

```
    # Reading file data
```

```
    with open(file_path, "rb") as file:
```

```
        file_data = file.read()
```

```
    # Making the PUT request to upload the file
```

```
    response = requests.put(url, headers=headers, data=file_data)
```

```
    # Check response status
```

```
    if response.status_code == 200:
```

```
        print(f"File {file_path.split('/')[-1]} successfully uploaded.")
```

```
    else:
```

```
        print(f"Error: {response.status_code} - {response.text}")
```

```
    return response
```

Test Input and Output:

Input	Output
<p>I'm Integrating with IBM Watson Language Translator API. Generate boilerplate for the below mentioned credentials</p> <p>API: IBM Watson Language Translator Method: POST Authentication: API Key Text to Translate: "Hello, how are you?" Source Language: English (en) Target Language: Spanish (es)</p>	<pre>import requests def translate_text(api_key, text, source_lang, target_lang): """ Translates the provided text from the source language to the target language using IBM Watson Language Translator API. Args: api_key (str): The API key for authenticating with the IBM Watson API. text (str): The text to be translated. source_lang (str): The source language of the text. target_lang (str): The target language for the translation. Returns: str: The translated text. Example: api_key = "your_ibm_api_key_here" text = "Hello, how are you?" source_lang = "en" target_lang = "es" translated_text = translate_text(api_key, text, source_lang, target_lang) print(translated_text) """ # Endpoint</pre>

<p>I'm Integrating with IBM Cloud Databases for PostgreSQL. Generate the boilerplate for the mentioned credentials</p> <p>API: IBM Cloud Databases for PostgreSQL Method: GET Authentication: API Key Query: SELECT * FROM users WHERE active = true; Database Name: user_db Region: us-south</p>	<pre>import psycopg2 def query_postgresql_db(api_key, database_name, region, query): """ Queries the specified PostgreSQL database and returns the result. Args: api_key (str): The API key for authenticating with IBM Cloud Databases for PostgreSQL. database_name (str): The name of the database to query. region (str): The region where the database is located. query (str): The SQL query to be executed. Returns: list: A list of tuples containing the query results. Example: query_postgresql_db("your_api_key", "user_db", "us-south", "SELECT * FROM users WHERE active = true;") """ # Connection details connection_details = { "dbname": database_name, "user": "your_username", # Replace with your actual username "password": "your_password",</pre>
---	--

I'm Integrating with IBM Visual Recognition API. Generate the Boilerplate, Here is my requirements

API: IBM Visual Recognition

Method: POST

Authentication: API Key

Image File Path: path/to/your/image.jpg

Model ID: default

Purpose: Classify objects in the image

```
import requests
```

```
import json
```

```
def classify_image_objects(api_key, image_path, model_id):
```

```
    """
```

```
    Classifies objects in an image using IBM Visual Recognition API.
```

```
    Args:
```

```
        api_key (str): The API key for authenticating with IBM Visual Recognition.
```

```
        image_path (str): The local path to the image file.
```

```
        model_id (str): The model ID to be used for object classification.
```

```
    Returns:
```

```
        JSON object: The response containing the classified objects or an error message.
```

```
    Example:
```

```
        api_key = "your_ibm_api_key_here"
```

```
        image_path = "path/to/your/image.jpg"
```

```
        model_id = "default"
```

```
        response = classify_image_objects(api_key, image_path, model_id)
```

```
        if response.status_code == 200:
```

```
            print(json.dumps(response.json))
```

Database Query Optimiser

Instruction: Take the following SQL or NoSQL query and optimize it for improved performance. Consider the following:

1. Optimize for speed and efficiency, especially for large datasets.
2. Use appropriate indexing or partitioning strategies if applicable.
3. Refactor the query to minimize unnecessary operations, such as joins or nested queries.
4. Suggest improvements for query structure, syntax, or usage of functions.
5. Provide a brief explanation of why the optimization improves performance.

Example Inputs and Outputs:

Input	Output
<pre>SELECT * FROM orders JOIN customers ON orders.customer_id = customers.customer_id WHERE customers.country = 'USA' AND orders.order_date > '2022-01-01';</pre>	<pre>SELECT orders.order_id, orders.order_date, customers.customer_name FROM orders JOIN customers ON orders.customer_id = customers.customer_id WHERE customers.country = 'USA' AND orders.order_date > '2022-01-01' AND customers.customer_id IN (SELECT customer_id FROM customers WHERE country = 'USA');</pre> <p>Explanation:</p> <ol style="list-style-type: none">1. The query now selects only the necessary columns (order_id, order_date, customer_name) rather than using SELECT *, which reduces data transfer and improves performance.2. The subquery IN condition ensures that only the relevant customer_id values are processed, potentially improving performance for large datasets if combined with appropriate indexing on customer_id and country.
<pre>db.orders.find({ "customer_id": 12345, "status": "shipped", "items": { \$elemMatch: { "category": "electronics" } } });</pre>	<pre>db.orders.createIndex({ "customer_id": 1, "status": 1, "items.category": 1 }); db.orders.find({ "customer_id": 12345, "status": "shipped", "items.category": "electronics" });</pre> <p>Explanation:</p> <ol style="list-style-type: none">1. An index is created on customer_id, status, and items.category to improve the performance of this query by speeding up the search on these fields, especially with large collections.2. The \$elemMatch operator was replaced by a direct field search (items.category), making the query simpler and more efficient, leveraging the new index.

SELECT department, COUNT(*), AVG(salary) FROM employees GROUP BY department HAVING AVG(salary) > 50000;	SELECT department, COUNT(*), AVG(salary) FROM employees WHERE salary > 50000 GROUP BY department; <p>Explanation:</p> <ol style="list-style-type: none"> 1. The HAVING clause is replaced by a WHERE clause. Since the condition (salary > 50000) applies to individual rows rather than aggregate values, using WHERE filters out unnecessary rows earlier in the process, reducing the workload of the GROUP BY operation. 2. This optimization improves performance by reducing the number of rows processed before aggregation.
--	---

Test Input and Output:

Input	Output
SELECT * FROM products JOIN categories ON products.category_id = categories.category_id WHERE categories.name = 'Electronics' AND products.price > 500 ORDER BY products.name;	SELECT * FROM products JOIN categories ON products.category_id = categories.category_id WHERE categories.name = 'Electronics' AND products.price > 500 ORDER BY products.name LIMIT 100; <p>Explanation:</p> <ol style="list-style-type: none"> 1. The query limits the number of returned rows to 100 using the LIMIT clause. This is a performance optimization to avoid returning a large number of rows that may not be necessary. 2. The query structure remains the same, but the LIMIT clause reduces the amount of data transferred, improving performance.

<pre>db.transactions.find({ "user_id": 98765, "date": { \$gte: new ISODate("2022-01-01"), \$lte: new ISODate("2022-12-31") }, "status": "completed" });</pre>	<pre>db.transactions.createIndex({ "user_id": 1, "date": 1, "status": 1 }); db.transactions.find({ "user_id": 98765, "date": { \$gte: new ISODate("2022-01-01"), \$lte: new ISODate("2022-12-31") }, "status": "completed" }).sort({ "date": 1 });</pre> <p>Explanation:</p> <ol style="list-style-type: none">1. An index is created on user_id, date, and status to improve the performance of this query by speeding up the search on these fields, especially with large collections.2. The sort operation (sort({ "date": 1 })) is added to ensure the documents are returned in the correct order, which may be necessary for further processing or reporting.
---	--