Exploring LLM-Assisted Code Commenting on Class-Level Code

Team Name: TheCommenters
Pranav Hariharane pkh210
Richard Cruz-Silva rsc2174

Synopsis

In recent years, Large Language Models (LLMs) have quickly established a stronghold in modern society as they have proven to be extremely useful in a variety of applications such as quick search, knowledge retrieval, writing help, summarization, explanation, planning, brainstorming, etc. Given how powerful LLMs have become, there have been a lot of studies in recent years on how well LLMs can be used to generate code. LLMs like Codex, CodeLlama, and GPT-4 that are pre-trained on massive amounts of code have proven to be able to produce pretty reliable code solutions for many different programming languages when provided with a natural language description of the coding task. However, most of these studies have only evaluated the capability of these LLMs in generating code for simple function-level code generation tasks. When evaluated on more complex code or pragmatic software development code, many code LLMs struggle to accurately produce correct code. In his midterm paper, Pranav explored the current performance of code LLMs in code generation tasks and their feasibility for usage in software development [2]. One paper that Pranav researched conducted an experiment evaluating the capability of code LLMs in generating code for class-level code generation tasks that are more representative of realistic software development scenarios [1]. The authors of this paper created a new benchmark called ClassEval which contains a dataset with 100 Python class-level code generation tasks and evaluated how well various code LLMs were able to produce accurate code for each task [1]. Unsurprisingly, the authors of the paper concluded that all the code LLMs struggled to perform nearly as well for class-level code generation tasks compared to the simple function-level code generation tasks used in most LLM-based code generation studies. The biggest reason for this was the inability of code LLMs to both be aware of and understand the contextual dependencies present within the class-level code templates provided to it. Contextual dependencies can be defined as the elements of a program that depend on both external or surrounding code. Some examples of contextual dependencies for class-level code include class attributes, class methods, third-party libraries, global attributes, etc. To investigate the insufficiencies that code LLMs have when producing code for realistic software development tasks in more depth, we chose to explore how well code LLMs could reason about the realistic software development code through comments. In our project, we evaluated how well two popular code LLMs (GPT–4 and Claude-3.7 Sonnet) can generate comments for code solutions provided by the ClassEval benchmark. In our experiment, we first manually annotated the 100 Python class-level code solutions from the ClassEval benchmark to serve as ground-truth commented code solutions. Then, we prompted both the GPT-4 and Claude LLMs to generate comments for the code present in 96 of the class-level code solutions since we provided 4 randomly selected examples of our manually annotated code solutions in the prompt to the LLMs. After receiving the outputs from both the LLMs, we evaluated the comments produced on the 96 class–level code solutions through both quantitative and qualitative metrics. The exact experimental setup, methodology, and results from our experiment can be seen in later sections of this report. Nonetheless, we hope that the results of our experiment and research can provide insights to the LLM-based code generation community and others about the next steps that should be taken to improve how LLMs perform in software development code generation tasks.

Research Questions

1. How well can code LLMs generate high-quality useful comments for methods in class-level code?

   a. Through our experiment, we want to assess how well code LLMs can generate high-quality comments demonstrating a high understanding for class-level code that's more representative of what realistic software development code is like.
2. How well can code LLMs capture the contextual dependencies present within the code through their comments?
   a. On top of assessing how well code LLMs can generate high-quality comments, we want to see if code LLMs are able to notice and highlight the use and purpose behind the contextual dependencies present within the code.
3. How do different code LLMs compare in their ability to reason about code?
   a. Since we're evaluating more than one code LLMs, we also want to see if there's a significant difference between the GPT-4 and Claude-3.7 Sonnet LLM in their ability to reason about code and all the contextual dependencies involved.

Experiment

1. Dataset Creation

For our experiment, we decided to evaluate the comments generated by the GPT–4 and Claude-3.7 Sonnet LLMs to see exactly how these LLMs are able to reason about class-level code and the contextual dependencies associated with it. As highlighted in the *Synopsis* section, our first step was to manually annotate the code solutions provided by the ClassEval benchmark so that we could have some sort of ground-truth solution to compare to the LLM-generated outputs. Since the ClassEval benchmark contained 100 Python class-level code examples, Pranav manually annotated the first 50 class-level code solutions provided by the benchmark and Richard manually annotated the last 50 class-level code solutions. Each class-level code template provided by the ClassEval benchmark already included docstrings describing the class, the role of each method in the class, and the appropriate behavior for each method. We included these docstrings in our annotations and manually added inline comments to describe the crucial lines of code for each method in the class. We also made sure to highlight the contextual dependencies that occurred in each method through our comments. Figure A in the Appendix section displays one example of the classes we manually annotated. After we finished manually annotating the classes, we reviewed each other's annotations to verify that they were correct, included the correct amount of detail, and highlighted all the contextual dependencies present in the class. Finally, we finalized the creation of our dataset for the experiment by creating a list of 100 entries in a JSON file where each entry contained an identifier for the class, the code solution for the class without any of our added inline comments, and our manually annotated version of the class.

2. LLM Configurations and Prompt

As mentioned previously, we used the GPT-4 and Claude-3.7 Sonnet LLMs to generate comments and we evaluated the results produced by each LLM separately. We configured both LLMs to have a *temperature* (parameter that controls the randomness of the output generated by the LLM where lower values result in more deterministic outputs and higher values result in more creative or diverse outputs) of 0.3, and *top-p* (parameter that controls the randomness of the output generated by the LLMs by considering only the tokens with a cumulative probability greater than a specified threshold) of 0.2 since that was recommended by online sources as the ideal parameters for code commenting tasks [3]. Both LLMs were also provided with the same system prompt to ensure they were operating under the same exact conditions and context. Figure 1 displays the exact system prompt provided to both LLMs.

"You are an expert software engineer and code annotator. Your task is to add clear, concise, technically inline comments to the code in the methods for the provided Python classes, without modifying the existing code or docstrings. In your annotations, emphasize identifying and explaining the contextual dependencies such as: third-party library functions or classes, class attributes and their usage across methods, calls to other class methods within the same class, external/global variables, or configurations that the method relies on. If any behavior is ambiguous, make cautious inferences but do not fabricate functionality."

Figure 1. System Prompt

As you can see, we made sure the system prompt provided context to the LLMs on how we wanted the comments to be generated on the class-level code solutions we provided to it. We even included an emphasis on the contextual dependencies that we wanted the LLMs to highlight through the comments since this was a major point of emphasis we wanted to explore. Even though the system prompt helped us guide the LLMs on the overall context, behavior, and responses we wanted, the user prompt is the main thing that drives the LLM to respond to a particular task. For the user prompt, we utilized few-shot prompting to further guide the LLMs in their response. The user prompt consists of 4 randomly selected examples of manually annotated code from the dataset that we created along with the target class-level code solution. We also specify in the prompt for the LLMs to annotate the class methods of the target class-level code solution with inline comments. To ensure consistency and correctness in the context provided to both LLMs we used the same 4 randomly selected examples of manually annotated code for all the user prompts and made sure to not evaluate the LLMs on these examples.

3. Metrics

After invoking the GPT-4 and Claude-3.7 Sonnet LLMs to generate comments for 96 class-level code solutions, we stored the results for each LLM in separate output JSON files. To evaluate these results, we used the BLEU metric and human evaluation. The BLEU metric is basically just a word and phrase similarity checking metric that compares the ngram tokens in the output generated by the LLMs with the manually annotated class-level code solutions we created in the input dataset. Because this metric only calculates token similarity and doesn't really check for semantic correctness of the comments, we place more weight on the human evaluation. For the human evaluation metric, both Pranav and Richard assigned a score of 0 (poorly generated comments that don't capture the intent of the code in the method and all the involved contextual dependencies), 0.5 (decently generated comments that do an okay job of capturing the intent of the code in the method and all the involved contextual dependencies), or 1 (well generated comments that do a great job of capturing the intent of the code in the method and all the involved contextual dependencies) to all the outputs generated by each LLM. Pranav and Richard also analyzed the comments generated by the LLMs separately to see if there were any significant patterns or trends and to see if the LLMs were able to document the contextual dependencies properly.

Results

Upon generating the commented outputs from both the Claude-3.7 Sonnet and GPT-4 LLMs for the 96 class-level code solutions, we first evaluated the generated comments with the BLEU metric. The overall summary for the BLEU metric results are displayed in Table 1.

LLM Used	Min BLEU Score	Max BLEU Score	Mean BLEU Score
GPT-4	0.39795532646391424	0.8877140346215109	0.6502555474446245
Claude-3.7 Sonnet	0.43395282678357294	0.9066257485276514	0.6720030322814536

Table 1. BLEU Metric Summary

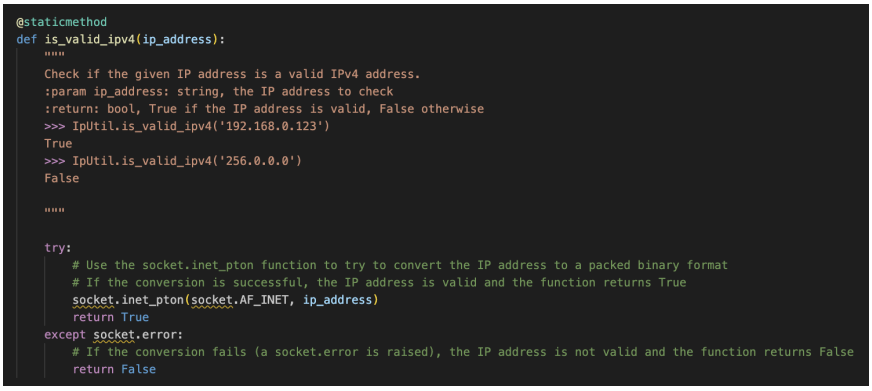
For the BLEU metric, scores in the range of 0.3-0.5 convey decent similarity, scores in the range of 0.5-0.7 indicate good similarity similar to human-level translations, and scores in the range of 0.7-0.9 denote excellent quality similarity with high overlap. Since the minimal BLEU scores are around 0.4 for GPT-4 and 0.43 for Claude-3.7 Sonnet and the average BLEU scores are around 0.65 for GPT-4 and 0.67 for Claude-3.7 Sonnet, we can conclude for the most part that the annotated classes produced by both LLMs are pretty close in token similarity with the manually annotated class-level code solutions we created for the dataset. These results are promising since they suggest that the comments generated by both LLMs may also be pretty similar to the ground-truth comments generated by us for the manually annotated class-level code solutions. Even though these results are promising, the BLEU metric only evaluates the LLM-generated outputs based on token similarity and doesn't really evaluate the actual correctness of the generated comments. That's why we also conducted a human evaluation where we manually check the generated comments for their correctness and detail. The results from the human evaluation can be summarized by Table 2.

LLM Used	Human Evaluation Mean Score
GPT-4	0.8203125
Claude-3.7 Sonnet	0.9296875

Table 2. Human Evaluation Summary

The human evaluation results indicate that both LLMs performed really well and were able to generate good high-quality comments that highlighted the contextual dependencies present within most of the provided code. However, we can't be enticed by these results too much since the classes in the ClassEval benchmark were very simple and elementary for the most part. Although they are more complex than the simple function-level code provided in most code generation benchmarks, they still definitely do not accurately reflect the true complexity of software development code. In addition, the classes in the ClassEval benchmark only made use of very simple third-party libraries like datetime, numpy, string, json, etc. These libraries are so common that it's very likely that they were included in the code that the LLMs were trained on. This is definitely the biggest limitation for this experiment and a major reason why the LLMs were able to generate high-quality comments across all the provided classes for the most part.

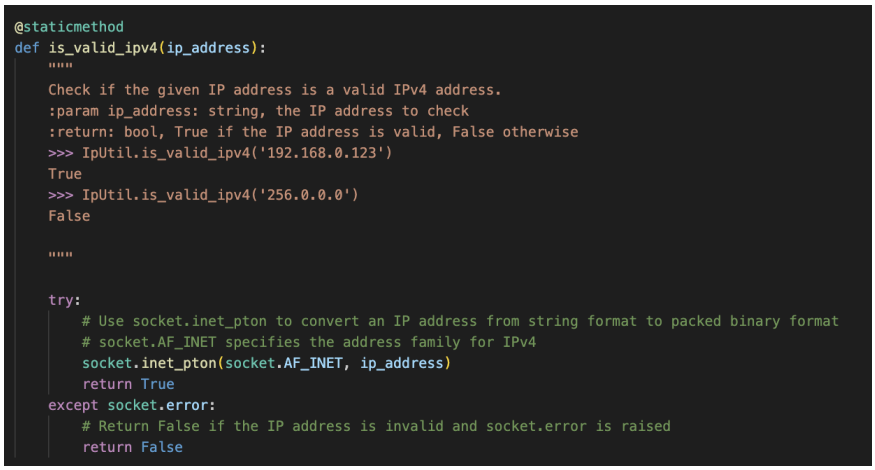
The human evaluation scores also show that the Claude-3.7 Sonnet LLM produced better comments on average compared to the GPT-4 model. Upon further qualitative investigation we noticed that the biggest reason for this was the GPT-4 LLM's inability to notice and properly reason about some of the contextual dependencies within the code. One example of this behavior can be seen in Figure 2 which displays a Python method utilizing the socket library.



```
@staticmethod
def is_valid_ipv4(ip_address):
    """
    Check if the given IP address is a valid IPv4 address.
    :param ip_address: string, the IP address to check
    :return: bool, True if the IP address is valid, False otherwise
    >>> IpUtil.is_valid_ipv4('192.168.0.123')
    True
    >>> IpUtil.is_valid_ipv4('256.0.0.0')
    False

    """

    try:
        # Use the socket.inet_pton function to try to convert the IP address to a packed binary format
        # If the conversion is successful, the IP address is valid and the function returns True
        socket.inet_pton(socket.AF_INET, ip_address)
        return True
    except socket.error:
        # If the conversion fails (a socket.error is raised), the IP address is not valid and the function returns False
        return False
```

Figure 2. GPT-4 LLM Comments Example

As shown in the image, although the GPT-4 LLM was able to describe the socket.inet_pton function and the socket.error variable, it wasn't able to notice and point out the usage of the socket.AF_INET constant which is crucial towards the functionality of the method. There's many other examples like this where the GPT-4 LLM fails to point out or properly reason about certain third-party library properties. The only third-party libraries where GPT-4 consistently performs well were the "re", "numpy", "sqlite3", and "nltk" libraries. One possible reason for this could be that the code that the GPT-4 LLM was trained on may have contained high usage of these libraries and as a result the LLM is able to understand and make sense of all the usages of these libraries. On the other hand, the Claude-3.7 Sonnet LLM almost never failed to point and properly reason about almost all the contextual dependencies involved in the code. Figure 3 illustrates the comments that the Claude-3.7 Sonnet LLM made on the exact same code snippet displayed in Figure 2.



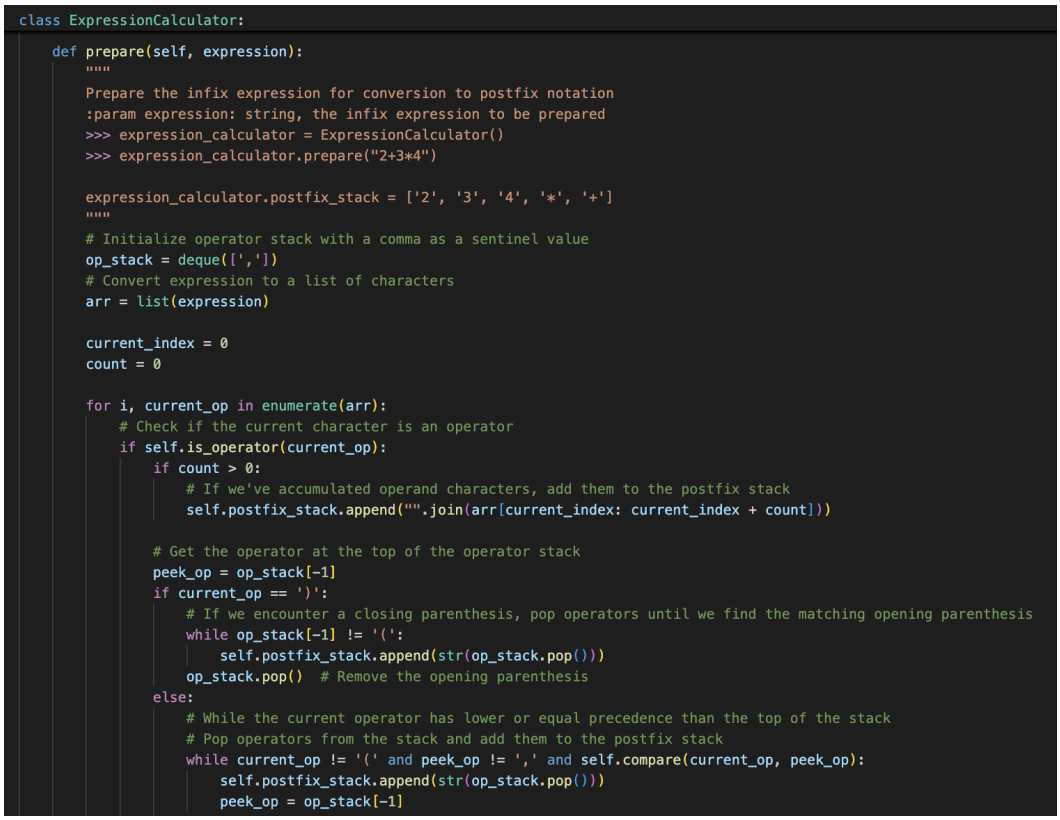
```
@staticmethod
def is_valid_ipv4(ip_address):
    """
    Check if the given IP address is a valid IPv4 address.
    :param ip_address: string, the IP address to check
    :return: bool, True if the IP address is valid, False otherwise
    >>> IpUtil.is_valid_ipv4('192.168.0.123')
    True
    >>> IpUtil.is_valid_ipv4('256.0.0.0')
    False

    """

    try:
        # Use socket.inet_pton to convert an IP address from string format to packed binary format
        # socket.AF_INET specifies the address family for IPv4
        socket.inet_pton(socket.AF_INET, ip_address)
        return True
    except socket.error:
        # Return False if the IP address is invalid and socket.error is raised
        return False
```

Figure 3. Claude-3.7 Sonnet Comments Example

As displayed in the image above, the Claude-3.7 Sonnet is able to point out and reason about all the contextual dependencies involved in the code. It is even able to highlight the importance of the socket.AF_INET constant which the GPT-4 LLM failed to do.

One insufficiency that we noticed both LLMs displayed in their comments was their inability to reason about code at a high level for larger classes that had very large methods. These larger methods also proved to be more of a challenge for the LLMs when reasoning about the involved contextual

dependencies. In these cases, whenever the code made use of a contextual dependency, the LLMs simply commented on what the code did instead of describing the purpose of the code at a higher level. There were instances where the LLMs generated comments that simply stated something along the lines of “Call <insert function name> and store the result.” instead of pointing out how this line of code relates to the overall purpose of the method or particular section of the code. An example of this can be seen in Figure 4.



```
class ExpressionCalculator:
    def prepare(self, expression):
        """
        Prepare the infix expression for conversion to postfix notation
        :param expression: string, the infix expression to be prepared
        >>> expression_calculator = ExpressionCalculator()
        >>> expression_calculator.prepare("2+3*4")

        expression_calculator.postfix_stack = ['2', '3', '4', '*', '+']
        """
        # Initialize operator stack with a comma as a sentinel value
        op_stack = deque([','])
        # Convert expression to a list of characters
        arr = list(expression)

        current_index = 0
        count = 0

        for i, current_op in enumerate(arr):
            # Check if the current character is an operator
            if self.is_operator(current_op):
                if count > 0:
                    # If we've accumulated operand characters, add them to the postfix stack
                    self.postfix_stack.append("".join(arr[current_index: current_index + count]))

                # Get the operator at the top of the operator stack
                peek_op = op_stack[-1]
                if current_op == ')':
                    # If we encounter a closing parenthesis, pop operators until we find the matching opening parenthesis
                    while op_stack[-1] != '(':
                        self.postfix_stack.append(str(op_stack.pop()))
                    op_stack.pop()  # Remove the opening parenthesis
                else:
                    # While the current operator has lower or equal precedence than the top of the stack
                    # Pop operators from the stack and add them to the postfix stack
                    while current_op != '(' and peek_op != ',' and self.compare(current_op, peek_op):
                        self.postfix_stack.append(str(op_stack.pop()))
                        peek_op = op_stack[-1]
```

Figure 4. Large Code Comments Example

As seen in the image above, at times, the LLM only generates comments that literally describe exactly what the code is doing without relating it to the overall purpose it serves for the method.

Overall, these were the main significant results we found when analyzing the results of the LLM-generated comments produced by the GPT-4 and Claude-3.7 Sonnet LLMs both through quantitative and qualitative measures. These results address all the research questions that we stated earlier in the report.

Deliverables

Github Link: Project Code

Self-Evaluation

We were able to show some promising results of having LLMs comment class-level code to include contextual dependencies. It was very time-consuming to hand comment 100 Python classes.

Assigning scores to the LLM-generated code was also very subjective and time-consuming so having 2 people evaluate the generated code may have helped decrease the subjective variability. Richard was very surprised how well the LLMs generated code comments with only 4 examples prompted. Richard created the code that compares two different input strings by splitting them into their individual words and using the Natural Language Toolkit module in Python to calculate the BLEU scores, aggregating the scores, and printing the minimum, maximum, and mean BLEU scores for both GPT-4 and Claude.

Pranav was surprised with how we were able to enable LLMs to annotate classes fairly well with very limited provided context through the system and user prompts. Pranav also highly underestimated how long it would take to manually annotate the 100 Python class-level code solutions provided by the ClassEval benchmark and how long it would take to conduct the human evaluations for the outputs generated by the LLMs. One big challenge Pranav faced was being able to determine whether an LLM was able to correctly reason about contextual dependencies based on how they were used in the code or just the semantic meaning of the method or variable names. Both Richard and Pranav split the effort that went into the creation of the input dataset with the manually annotated code. Pranav created the code to prompt both the GPT-4 and Claude-3.7 Sonnet LLMs and insert their outputs into JSON files in the output folder. Pranav also contributed the code that calculated the average of the human evaluation scores across all the inputs that were tested. Finally, both Richard and Pranav conducted human evaluation across all the outputs generated by both LLMs. These summarize the main contributions made by each team member along with the parts of the project they found surprising, challenging, etc.

References

[1] Du, X. *et al.* (2024) 'Evaluating large language models in class-level code generation', *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13. https://dl.acm.org/doi/10.1145/3597503.3639219

[2] Hariharane, Pranav. (2025) 'An In-Depth Investigation On Current LLM Code Generation And Feasibility Of LLM-Based Code Generation In Software Development', *Columbia University COMS E6156 Full Paper*, pp. 1–16. https://github.com/hariharanep/ClassEvalCodeGenerationExperiment

[3] OpenAI Developer Community. (2023) 'Cheat Sheet: Mastering Temperature and Top_p in ChatGPT API', *OpenAI Developer Forum*. https://community.openai.com/t/cheat-sheet-mastering-temperature-and-top-p-in-chatgpt-api/172683

Appendix

Figure A. Example Manually Annotated Class

```
import sqlite3
class UserLoginDB:
  """
  This is a database management class for user login verification, providing functions for inserting user information, searching user information, deleting user information, and validating user login.
  """
  def __init__(self, db_name):
    """
    Initializes the UserLoginDB object with the specified database name.
    :param db_name: str, the name of the SQLite database.
    """
    # Use the connect function to create a connection to the database db_name.
    self.connection = sqlite3.connect(db_name)
    # Use the cursor function for the connection to create a cursor object.
    self.cursor = self.connection.cursor()
  def insert_user(self, username, password):
    """
    Inserts a new user into the "users" table.
    :param username: str, the username of the user.
    :param password: str, the password of the user.
    :return: None
    >>> user_db = UserLoginDB("user_database.db")
    >>> user_db.create_table()
    >>> user_db.insert_user('user1', 'pass1')
    """
    # Use the execute function of the cursor to insert the provided username and password into the "users" table.
    self.cursor.execute("""
      INSERT INTO users (username, password)
      VALUES (?, ?)
    """, (username, password))
    # Use the commit function of the connection to save the changes to the database.
    self.connection.commit()
  def search_user_by_username(self, username):
    """
    Searches for users in the "users" table by username.
    :param username: str, the username of the user to search for.
    :return:list of tuples, the rows from the "users" table that match the search criteria.
    >>> user_db = UserLoginDB("user_database.db")
```

```
    >>> user_db.create_table()
    >>> user_db.insert_user('user1', 'pass1')
    >>> result = user_db.search_user_by_username('user1')
    len(result) = 1
    """
    # Use the execute function of the cursor to search for the user with the provided username in the "users" table.
    self.cursor.execute("""
       SELECT * FROM users WHERE username = ?
    """, (username,))
    # Use the fetchone function of the cursor to retrieve the first row of the result set and return it.
    user = self.cursor.fetchone()
    return user
  def delete_user_by_username(self, username):
    """
    Deletes a user from the "users" table by username.
    :param username: str, the username of the user to delete.
    :return: None
    >>> user_db = UserLoginDB("user_database.db")
    >>> user_db.create_table()
    >>> user_db.insert_user('user1', 'pass1')
    >>> user_db.delete_user_by_username('user1')
    """
    # Use the execute function of the cursor to delete the user with the provided username from the "users" table.
    self.cursor.execute("""
       DELETE FROM users WHERE username = ?
    """, (username,))
    # Use the commit function of the connection to save the changes to the database.
    self.connection.commit()
  def validate_user_login(self, username, password):
    """
    Determine whether the user can log in, that is, the user is in the database and the password is correct
    :param username:str, the username of the user to validate.
    :param password:str, the password of the user to validate.
    :return:bool, representing whether the user can log in correctly
    >>> user_db = UserLoginDB("user_database.db")
    >>> user_db.create_table()
    >>> user_db.insert_user('user1', 'pass1')
    >>> user_db.validate_user_login('user1', 'pass1')
    True
```

```
    """
    # Use the search_user_by_username class method to get the user information for the provided username.
    user = self.search_user_by_username(username)
    # If the user exists and the provided password matches the stored password, return True.
    if user is not None and user[1] == password:
        return True
    # Otherwise, return False.
    return False.
```