

ET574: Copilot, Prompt Engineering for Large Language Models

GitHub Copilot


Objective:

By the end of this section the student should be comfortable using GitHub Copilot for code completion, debugging and self-learning.

GitHub Copilot is an AI assistant specifically designed for code completion, explanation and debugging with contextual awareness. *GitHub Copilot* and *GitHub Copilot Chat* are available as Codespaces extensions. Copilot is free of charge with its default AI model but there will be a finite number of code completions and chats per GitHub user account which resets monthly.

Using Copilot for free within the quota limits requires a disciplined strategy. As a student it may be optimal to reserve code completion for writing unfamiliar code whereas a professional would probably use it to complete obvious or repetitive code. The external ChatGPT website is an alternative to Copilot Chat for explaining code, but Copilot Chat can be superior for debugging or modification because it has the context of the entire file and chat history.

Copilot Code Completion:

The most common use of GitHub Copilot is code completion in which context dependent code suggestions are displayed as code is edited. Suggested code can be accepted by hitting the *TAB* key. While useful, code completion can be intrusive. Fortunately, it is possible to configure this feature from the Copilot menu which is accessible by clicking on the  icon on the bottom status bar of Codespaces. When using code completion keep in mind that there are a finite number of free completions per month so if coding regularly one should be selective about when to use it. It is also worth noting that Code completion usage quota can be viewed by clicking on the menu icon.

Assignment Requirements:

The entire team must have access to a shared repository. Each teammate will create their own Codespace to connect to the team repository. Members will complete simple programming tasks in their respective Codespaces before committing changes to the team repository. This assignment is designed to be used with the Codespaces IDE Source Control interface but can also be accomplished using traditional Git command line instructions. Furthermore, students should already be familiar with the basics of using *Codespaces Explorer*, *Extension* and *Source Control* windows as well as how to write, run and debug a Python program located within a subdirectory of the local branch. Team members may divide the tasks and complete them individually, or they can collaborate and work together on the same task.

Create a repository for your work:

First, one team member should create a shared repository in their GitHub account. The repository name must follow this naming format exactly:

<Course Name>-<Section Name>-<HW3>-<Your Group Name>

For example, *ET574-K-HW3-Group1* (ET 574 Section K Homework 3 for Group 1)

After creating the repository, that team member must add all other group members as collaborators, and also add the instructor as a collaborator.

Task A: Copilot for code completion:

Modern development will rely upon AI coding assistants to expedite development. In this example we will write comments specifying what we want to do and use the AI assistant to fill in the code.

1. Create a new program named *l_copilot.py* and copy the following comments into the program.

list of three students named Jon, Kim and Lee

function to print 'Hi name' for each student in the list

call the function

2. Click at the end of the first comment and hit *return*. Copilot will suggest code to implement a list of the three students. Hit the *TAB* key to insert the suggested code.

3. Click at the end of the second comment and hit *return*. Copilot will suggest a context related function. Hit the *TAB* key to insert the suggested code.

4. Click at the end of the third comment and hit *return*. Copilot will suggest a context related function call. Hit the *TAB* key to insert the suggested code. Run the program and view the output.

Copilot Chat:

Another useful feature is Copilot Chat. Copilot Chat is a custom ChatGPT AI prompt interface which contains a variety of developer-oriented features for analyzing and debugging code. Chat can be accessed *inline* by typing *CTRL+I* (windows) or *CMD+I* (macOS) within a program. Alternatively, a sidebar Chat window can be toggled by typing *SHIFT+CTRL+I* (windows) or *SHIFT+CMD+I* (macOS). These features can be accessed by selecting code within a program and typing */command* in the Chat to enact a specific action such as:

/explain: explain the selected code
/fix: debug the selected code
/test: setup unit testing

Furthermore, the Copilot Chat supports prompt requests to enhance or modify selected code to support new features.

Task B: Copilot Chat for self-learning:

Use Copilot Chat to explain the new code completion additions to the program.

1. The code completed program contains structures we may not have seen before such as a *list* or a *for loop*.
2. Use the mouse to select the *list* code from *1_copilot.py*:

```
students = ['Jon', 'Kim', 'Lee'].
```

3. Open the sidebar Copilot Chat and type */explain* followed by the *TAB* key, then hit the *return* key. This should provide an explanation of what a Python *list* is.

4. Use the mouse to select the *for-loop* code from *1_copilot.py*:

```
for student in students:
    print(f'Hi {student}')
```

5. Open the sidebar Copilot Chat and type */explain* followed by the *TAB* key, then hit the *return* key. This should provide an explanation of what a *for loop* is and how it can *iterate through a list*.



If the explanation is unclear, it is possible to adjust how an explanation is presented by providing additional context. This will be further explained later in this assignment.

Task C: Copilot Chat for adding features:

Use Copilot Chat to improve or augment existing code with new or enhanced features.

1. Use the mouse to select all the code in `I_copilot.py`.
2. Type the following into Copilot Chat:

Add Sara and Miko to the list after the list is created. Modify the function to also print the total number of students.


3. Hover over the suggested code and click the  icon to view the possible change in the program. The changes will appear in the program as a temporary modification. Three options will be presented in the bottom right corner of the main window, *Keep*, *Undo* or the  icon to *Toggle the Diff editor for Chat Edits*. *Keep* will make the changes permanent, *Undo* will remove the modification, and *Toggle Diff* enables manual editing of the changes. Click *Keep* making the change permanent. Run the program to view the modified output.

Task D: Copilot Chat for debugging:

Use Copilot Chat to debug errors in our code.

1. Add the following code after the recently inserted code to append more students.

```
# change Jon to John
students[1] = 'John'
```

2. Run the program and examine the output. There is a logical error. The name *Jon* was not replaced with *John*, instead *Kim* was replaced with the name *John*.
3. Copilot can debug and correct this error based upon the code and the associated comment which gives it context. Select the new code then type `/fix` followed by hitting the `TAB` key and then the `return` key. Hover over the suggested code and click the  icon to view the possible change in the program. Click on *Keep* making the change permanent. Run the program to view the modified output.

Copilot should be reasonably proficient for detecting syntax errors but not necessarily logical errors unless additional *context* is provided. The comment '*change Jon to John*' provided a *context* for Copilot to recognize a logical error. Otherwise, Copilot would correctly assume that the intent was to replace *Kim* with *John* and nothing would change. This concept leads to the next section of the assignment where the focus is on designing successful prompts.

Copilot references the history of previous chats as *context* when answering new prompts. This can result in suggestions becoming oriented towards a singular path of development. To reduce the likelihood of this issue, click the + icon at the top of the Copilot Chat window to start a new chat conversation and start afresh.

Prompt Engineering

Objectives:

By the end of this assignment the student should be able to:

1. Explain basic artificial intelligence concepts relating to Large Language Models.
2. Explain the components of an effective AI model prompt.
3. Construct a well-formed prompt to process data and generate code using a Large Language Model.

Background Concepts:

Artificial Intelligence focuses on the development of intelligent systems that can perform tasks requiring properties of human intelligence such as perception, reasoning, learning, problem solving and decision making.

Machine Learning is a subset of AI for developing methods for machines to learn from data without relying upon explicit instructions.

Deep Learning is a subset of Machine Learning in which the biological concepts of neurons and synapses are used to implement an artificial neural network of nodes to analyze training data and make predictions.

Generative AI is a subset of Deep Learning that focuses upon generating new forms of data based upon patterns and structures learned from training data. This includes text, code, image, audio and video generation.

AI Models are computer programs trained on data to identify patterns and relationships and apply that knowledge to new data. Some well-known examples of AI models include GPT-4 for text, Dall-E for images, Claude for code, and Gemini for multimodal handling of text, images, audio and video.

A **Large Language Model** is a type of Generative AI Model trained on vast amounts of data for the purpose of understanding and generating human-like output. GPT-4, Claude, Gemini and Copilot are renown LLMs.

A **Prompt** is an input request, in the form of a statement, question, or instruction given to an AI model with the purpose of producing a specific response.

Prompt Engineering is the method of creating effective instructions, commonly referred to as prompts, to guide generative AI models to produce desired outputs.


Prompt Engineering for Large Language Models:

A well-designed prompt may include one or more *instructions*, a *context*, *input data*, *output indicators*, and *examples*.

Instruction:	a specific request you want the model to execute
Context:	additional background information to direct the model to an optimal response
Input Data:	the data to be processed by the instruction
Output Indicator:	the type or format of the output
Examples:	examples of similar input-output pairs for reference

The subsequent tasks will use <https://chatgpt.com/> to demonstrate the use of these concepts for successful prompt development. As mentioned previously, the quota limitations for free use of Copilot provide a reason to secondarily use an external AI assistant such as ChatGPT for significant projects. ChatGPT uses conversation history as additional training data and context which may steer the results on a certain path. This can be avoided by setting ChatGPT to Temporary Chat mode as specified in the first task below.

Task E: A simple prompt

1. Set ChatGPT to temporary mode by clicking the temporary  icon in the upper right corner.
2. Navigate to <https://chatgpt.com/>.
3. Enter the following prompt containing a simple *instruction*:

Write a program to store twenty-five students and their GPAs into two lists for analysis.

Note the length of the output, the complexity of code, the method of obtaining the data (console input which is tedious) and the assumptions of the type of analysis to be conducted. Does this prompt produce a solution that is suitable for an entry level programming student?

4. Click New Chat in the upper left-hand corner and try the prompt again. Is the response consistent with the previous response?

For subsequent tasks, test each prompt multiple times using temporary chat mode and new chat windows to evaluate the consistency of the output response for a given prompt.

Specificity:

When humans converse misunderstandings can occur, by mistake or intentionally, due to the phrasing of a statement or question. Often humans rely on additional cues such as body language or tone to mitigate confusion. The same problem may occur in human to machine interaction except that the machine must solely rely upon the phrasing of the request to respond appropriately. Therefore, a successful prompt must be clear and concise to reduce confusion yet specific enough to guide the scope of the request to the desired outcome. The use of direct and precise language in a prompt tends to improve the output.

Task F: Improving the simple prompt:

1. Test an updated prompt including detailed *instructions* and *output indicators* using dashes, bullets or other separators to separate the details of the request:

Write a Python program to store students and their GPAs into two lists.

- *Read student and GPA data from a file named students.txt.*
- *Compute the average GPA and print it out.*
- *Print all above average students.*
- *Predict which students will earn a scholarship.*

The complex prompt produces a more focused output, but the code is still beyond the knowledge of a beginner. In addition to the *output indicators*, we can provide *context* to produce a more appropriate solution for a beginner.

2. Test an updated prompt with *context* and more *instructions* specifying code complexity:

The solution should be understandable by a Python beginner.

Write a Python program to store students and their GPAs into two lists.

- *Read student and GPA data from a file named students.txt.*
- *Compute the average GPA and print it out.*
- *Print all above average students.*
- *Predict which students will earn a scholarship.*
- *Do not use syntax such as def, while, try, except.*

The do not use *output indicator* reduces the requisite knowledge to understand the solution. The additional *context* causes the model to produce simpler but more verbose code solving the problem in a stepwise manner that is easier for a beginner to understand. This may not be the optimal solution for a professional, but it is the better solution for a beginner.

3. Test the updated prompt with *input data* to process and more information about the GPA range:

The solution should be understandable by a Python beginner.

Write a Python program to store students and their GPAs into two lists.

- *GPA is a value between 0.0 and 4.0. Compute the average GPA and print it out.*
- *Print all above average students.*
- *Predict which students will earn a scholarship.*
- *Do not use syntax such as def, while, try, except.*
- *list data: Jon 3.25, Kim 2.25, Lee 2.30, Sara 4.00, Miko 1.90, Lin 2.10, Toby 2.89, Ben 2.75, Mark 2.34, Xia 3.53*

There is still the question of how the model predicts which students earn scholarships. Without data, examples or additional instructions the model is forced to rely upon its training and make an educated guess.

Zero-Shot, One-Shot, Few-Shot Prompting:

Prompts may or may not include examples of *input-output pairs* to guide the model. When working with LLMs the term ‘shots’ refers to the number of examples provided within the prompt. A Zero-Shot or One-Shot prompt (0 or 1 example(s)) can be effective using an LLM because of the large-scale training. Few-Shot prompts containing multiple examples may be more effective as the model benefits from in-context learning of the desired outcome.

Task G: Few-Shot prompting:

1. Test the updated prompt with *input-output pairs* as examples:

The solution should be understandable by a Python beginner.

Write a Python program to store students and their GPAs into two lists.

- GPA is a value between 0.0 and 4.0. Compute the average GPA and print it out.

- Print all above average students.

- Predict which students will earn a scholarship.

- Do not use syntax such as *def*, *while*, *try*, *except*.

list data: Jon 3.25, Kim 2.25, Lee 2.30, Sara 4.00, Miko 1.90, Lin 2.10, Toby 2.89, Ben 2.75, Mark 2.34, Xia 3.53

Jake 2.75 earned a scholarship.

Markus 2.90 earned a scholarship.

Teri 3.15 earned a scholarship.

Leigh 3.75 earned a scholarship.

The examples offer criteria for evaluating scholarships. This guides the model to predict that students with a 2.75 GPA or better will earn scholarships. However, the model may misinterpret information due to incorrect prompt parsing because of prompt format. The examples could be incorrectly evaluated as additional data with scholarship predictions based upon *Jon*, *Sara*, *Toby*, *Ben* and *Xia* not earning scholarships.

Prompt Injection:

It is possible for the human to sabotage a prompt by *hijacking* it with a *prompt injection*. Prompt injections can be malicious and intentional but may also occur inadvertently when prompt content is parsed incorrectly. This can be avoided by separating prompt sections into layers using *delimiters*. Delimiters such as ``###<> are characters that surround prompt text to separate the various sections of the prompt (instructions, data, examples etc.)

Task H: Avoiding parsing errors:

1. Test the updated prompt with delimiters to isolate data and scholarship prediction based upon examples:

The solution should be understandable by a Python beginner.

Write a Python program to store students and their GPAs into two lists.

- GPA is a value between 0.0 and 4.0. Compute the average GPA and print it out.

- Print all above average students.

- Do not use syntax such as *def*, *while*, *try*, *except*.

- Predict which students will earn a scholarship *based upon these examples*:

Jake 2.75 earned a scholarship.

Markus 2.90 earned a scholarship.

Teri 3.15 earned a scholarship.

Leigh 3.75 earned a scholarship.

###list data: Jon 3.25, Kim 2.25, Lee 2.30, Sara 4.00, Miko 1.90, Lin 2.10, Toby 2.89, Ben 2.75, Mark 2.34, Xia 3.53
###

Chain-of-Thought Prompting:

Complex reasoning tasks requiring multiple intermediary steps for resolution benefit from a stepwise problem-solving approach. CoT prompts include a list of steps on how to solve the problem or use language such as ‘think step by step’ to instruct the model to evaluate instructions in a stepwise manner.

Task I: Complex reasoning:

A third dimension, student major, was added and data is now stored in a *Dictionary*, which pairs a key (student name) with corresponding values (GPA, major). Updated examples should guide the model to assess Math students with a 3.5 or better GPA and Biology students with a 2.75 or better GPA as eligible for scholarships. **The goal is for the model, instead of the human, to generate the algorithm for why students are awarded scholarships by recognizing patterns in data.**

1. Test the below prompt multiple times in temporary mode with new chats. It is likely the model will produce inconsistent predictions for when students would earn scholarships:

The solution should be understandable by a Python beginner.

*Write a Python program to **store students, majors and their GPAs into a dictionary.***

- GPA is a value between 0.0 and 4.0.
- Compute the average GPA and print it out.
- Print all above average students.
- Do not use syntax such as *def*, *while*, *try*, *except*.
- Predict which students will earn a scholarship based upon these examples:

Teri 3.15 Math did not earn a scholarship.

Sue 2.75 Biology earned a scholarship.

Markus Biology had an above average GPA and earned a scholarship.

Anders Math had an above average GPA and did not earn a scholarship.

Leigh 3.57 Math earned a scholarship.

###list data: Jon 3.25 Math, Kim 2.25 Biology, Lee 2.30 Math, Sara 4.00 Math, Miko 1.90 Math, Lin 2.10 Biology, Toby 2.89 Biology, Ben 2.75 Math, Mark 2.34 Math, Xia 3.53 Biology

###

2. Adding an instruction to process the scholarship prediction in a step-by-step manner may yield better results:

The solution should be understandable by a Python beginner.

Write a Python program to store students, majors and their GPAs into a dictionary.

- GPA is a value between 0.0 and 4.0.
- Compute the average GPA and print it out.
- Print all above average students.
- Do not use syntax such as *def*, *while*, *try*, *except*.
- Predict which students will earn a scholarship based upon these examples:

Teri 3.15 Math did not earn a scholarship.

Sue 2.75 Biology earned a scholarship.

Markus Biology had an above average GPA and earned a scholarship.

Anders Math had an above average GPA and did not earn a scholarship.

Leigh 3.57 Math earned a scholarship.

- Think step-by-step to evaluate which students earn scholarships.

###list data: Jon 3.25 Math, Kim 2.25 Biology, Lee 2.30 Math, Sara 4.00 Math, Miko 1.90 Math, Lin 2.10 Biology, Toby 2.89 Biology, Ben 2.75 Math, Mark 2.34 Math, Xia 3.53 Biology

###

Task J: Using AI to test AI generated code:

AI models can be prompted to generate test data or code to test previously generated code.

1. Modify the previous prompt with instructions to a) generate test data in a common file format b) add code to read the data file and c) provide instructions on how to use the generated content to execute the test:

The solution should be understandable by a Python beginner.

Write a Python program to store students, majors and their GPAs into a dictionary.

- GPA is a value between 0.0 and 4.0.
- Compute the average GPA and print it out.
- Print all above average students.
- Do not use syntax such as `def`, `while`, `try`, `except`.
- Predict which students will earn a scholarship based upon these examples:

Teri GPA of 3.15 Math major did not earn a scholarship.

Sue GPA of 2.75 Biology major earned a scholarship.

Markus Biology major had an above average GPA and earned a scholarship.

Anders Math major had an above average GPA and did not earn a scholarship.

Leigh GPA of 3.57 Math major earned a scholarship.

Jon GPA of 3.67 Math major earned a scholarship.

- Think step-by-step to evaluate which students earn scholarships.
- Generate 20 entries of appropriate test data in CSV format.
- Add code to import this data from the CSV file.
- Output instructions on how to test the program.

2. Modern AI models reference the prompt and response chat history. New prompts can reference previous prompts to evaluate and debug code. Enter the following to evaluate the code generated from a previous prompt:

Reference the previous prompt. Output the result of testing the program with the generated CSV file.

Task K: Using AI to translate code:

LLMs can translate between spoken or programming languages. Enter a second follow-up prompt to rewrite the previous Python solution in the Java programming language.

Reference previous prompts. Rewrite the Python program in Java for a beginner programmer to understand.

Conclusion:

This assignment is limited but still offers a glimpse into what is possible using LLMs to purposefully generate content. Used responsibly AI assisted development can accelerate the student's ability to write complex programs to do real things. Nonetheless, the student should endeavor to learn how to code to retain maximum control and flexibility of the final product. The next step for the budding software developer or technically adept entrepreneur is to learn how to develop code automating prompt and response activities for building AI driven applications.

PDF Submission:

Submit a pdf file to BrightSpace which should include the following images of your work

1. Source Code

- Screenshots of the final source code for each task in Codespaces. Make sure to show all the comments you use to prompt AI.

2. Commit History

- Screenshot of the commit history tab in GitHub showing:
Multiple commits overtime and commit messages that match your work.

3. Program Output / Results

- Screenshot(s) of the program running successfully:
Terminal output with test results.

4. Issue Tracker

- Screenshot of the content of your issue tracker displayed in GitHub.
- Must clearly show entries of each task/issue being opened and closed (with commit ID).