

# Prompt engineering foundations and best practices

7 minutes

In this unit, we'll cover:

- What is prompt engineering?
- Foundations of prompt engineering
- Best practices in prompt engineering
- How Copilot learns from your prompts

## What is prompt engineering?

Prompt engineering is the process of crafting clear instructions to guide AI systems, like GitHub Copilot, to generate context-appropriate code tailored to your project's specific needs. This ensures the code is syntactically, functionally, and contextually correct.

Now that you know what prompt engineering is, let's learn about some of its principles.

## Principles of prompt engineering

Before we explore specific strategies, let's first understand the basic principles of prompt engineering, summed up in the **4 Ss** below. These core rules are the basis for creating effective prompts.

- **Single:** Always focus your prompt on a single, well-defined task or question. This clarity is crucial for eliciting accurate and useful responses from Copilot.
- **Specific:** Ensure that your instructions are explicit and detailed. Specificity leads to more applicable and precise code suggestions.
- **Short:** While being specific, keep prompts concise and to the point. This balance ensures clarity without overloading Copilot or complicating the interaction.
- **Surround:** Utilize descriptive filenames and keep related files open. This provides Copilot with rich context, leading to more tailored code suggestions.

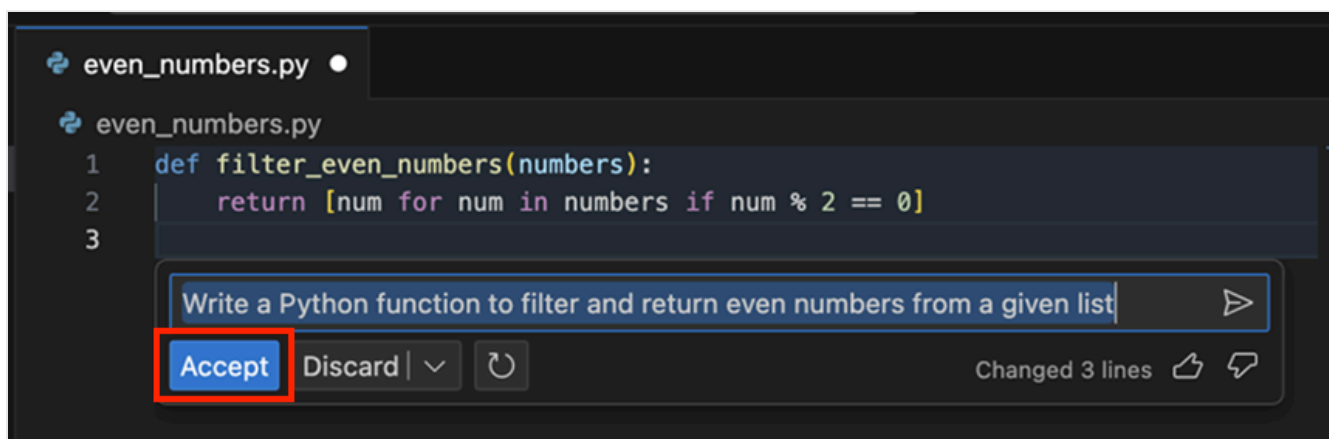
These core principles lay the foundation for crafting efficient and effective prompts. Keeping the 4 Ss in mind, let's dive deeper into advanced best practices that ensure each interaction with GitHub Copilot is optimized.

# Best practices in prompt engineering

The following advanced practices, based on the 4 Ss, refine and enhance your engagement with Copilot, ensuring that the generated code isn't only accurate but perfectly aligned with your project's specific needs and contexts.

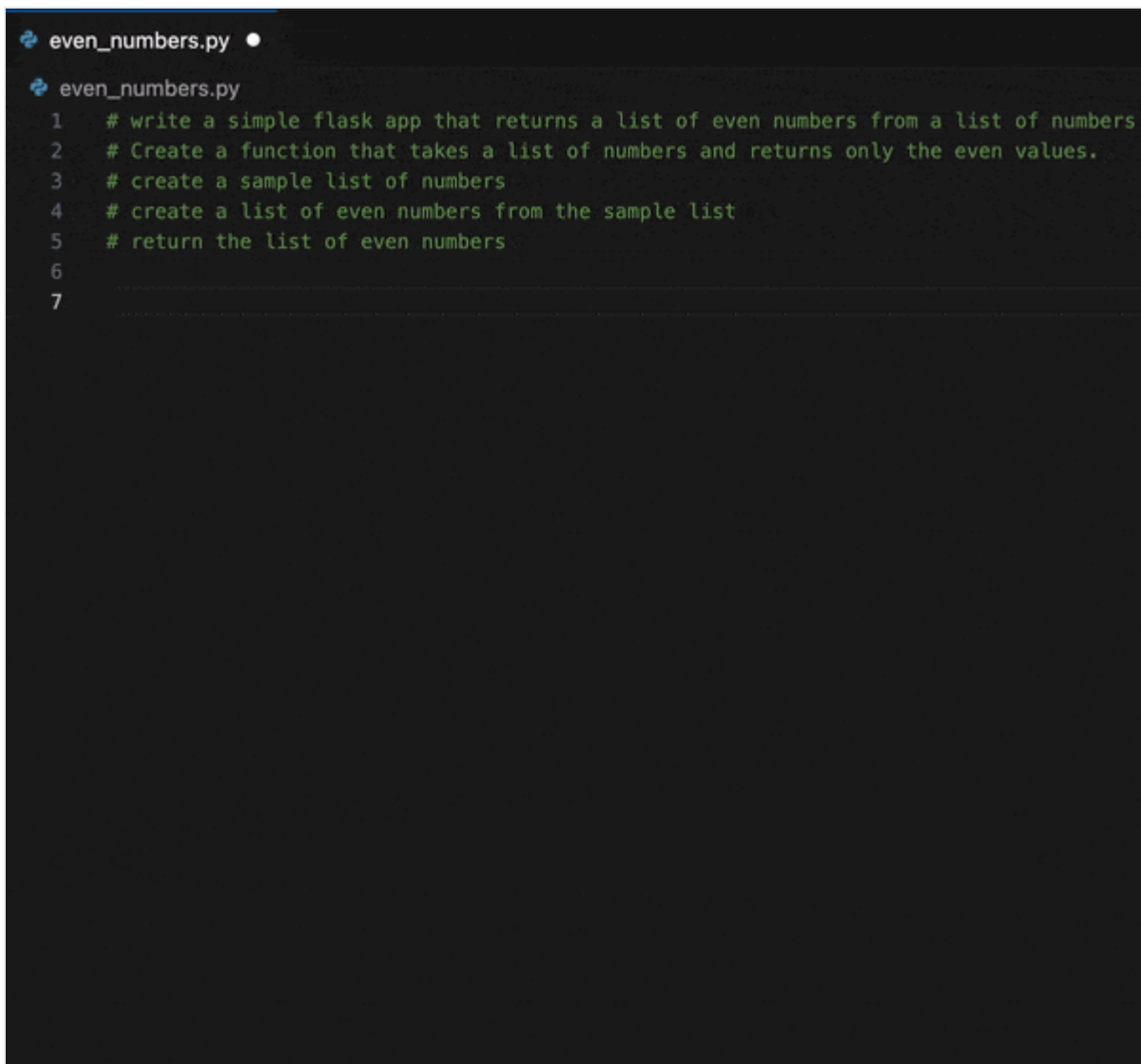
## Provide enough clarity

Building on the 'Single' and 'Specific' principles, always aim for explicitness in your prompts. For instance, a prompt like "Write a Python function to filter and return even numbers from a given list" is both single-focused and specific.



## Provide enough context with details

Enrich Copilot's understanding with context, following the 'Surround' principle. The more contextual information provided, the more fitting the generated code suggestions are. For example, by adding some comments at the top of your code to give more details to what you want, you can give more context to Copilot to understand your prompt, and provide better code suggestions.



```
even_numbers.py •  
even_numbers.py  
1  # write a simple flask app that returns a list of even numbers from a list of numbers  
2  # Create a function that takes a list of numbers and returns only the even values.  
3  # create a sample list of numbers  
4  # create a list of even numbers from the sample list  
5  # return the list of even numbers  
6  
7
```

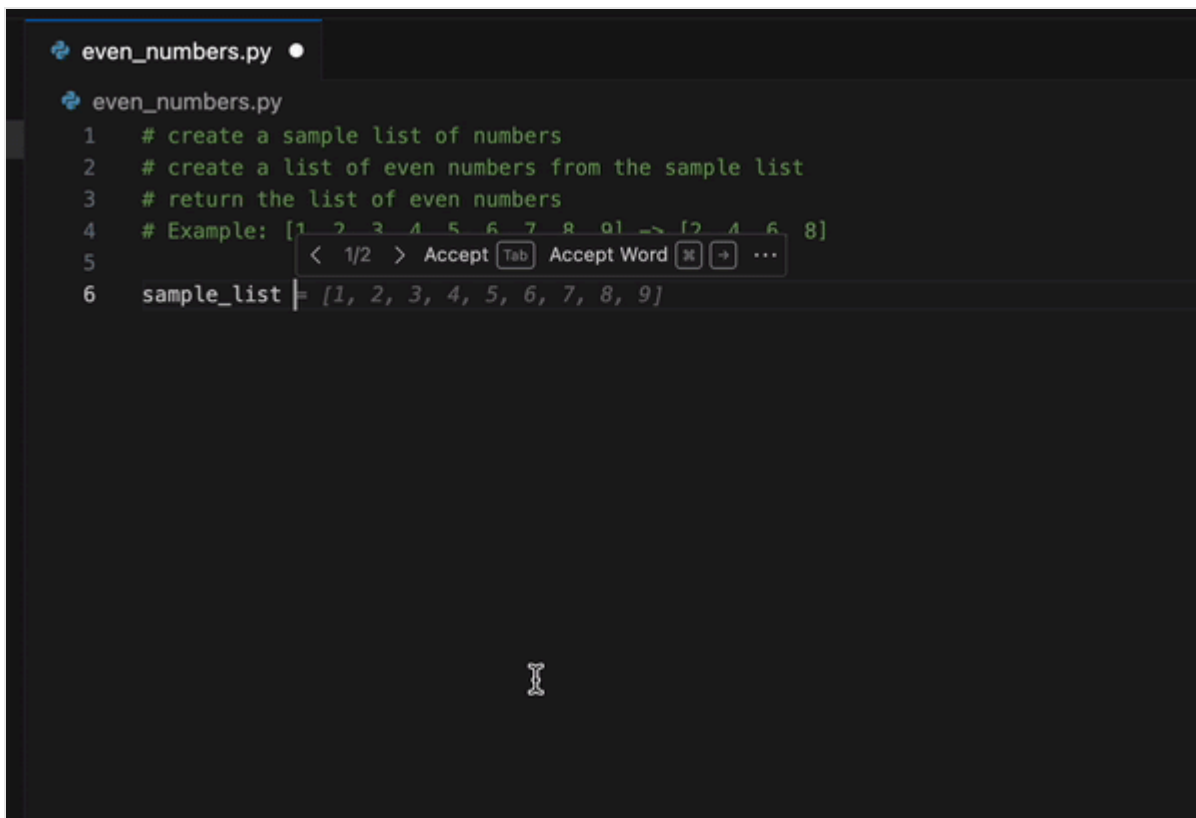
In the example above, we used steps to give more detail while keeping it short. This practice follows the 'Short' principle, balancing detail with conciseness to ensure clarity and precision in communication.

#### ⓘ Note

Copilot also uses parallel open tabs in your code editor to get more context on the requirements of your code.

## Provide examples for learning

Using examples can clarify your requirements and expectations, illustrating abstract concepts and making the prompts more tangible for Copilot.



```
even_numbers.py
1 # create a sample list of numbers
2 # create a list of even numbers from the sample list
3 # return the list of even numbers
4 # Example: [1, 2, 3, 4, 5, 6, 7, 8, 9] -> [2, 4, 6, 8]
5
6 sample_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Tooltip: < 1/2 > Accept Tab Accept Word % + ...

## Assert and iterate

One of the keys to unlocking GitHub Copilot's full potential is the practice of iteration. Your first prompt might not always yield the perfect code, and that's perfectly okay. If the first output isn't quite what you're looking for, treat it as a step in a dialogue. Erase the suggested code, enrich your initial comment with added details and examples, and prompt Copilot again.

Now that you learned best practices to improve your prompting skills, let's take a closer look at how you can provide examples Copilot can learn from.

## How Copilot learns from your prompts

GitHub Copilot operates based on AI models trained on vast amounts of data. To enhance its understanding of specific code contexts, engineers often provide it with examples. This practice, commonly found in machine learning, led to different training approaches such as:

### Zero-shot learning

Here, GitHub Copilot generates code without any specific examples, relying solely on its foundational training. For instance, suppose you want to create a function to convert temperatures between Celsius and Fahrenheit. You can start by only writing a comment describing what you want, and Copilot might be able to generate the code for you, based on its previous training, without any other examples.

```
1 # I need a function to convert Celsius to Fahrenheit
2
3 def celsius_to_fahrenheit(celsius):
4
5     fahrenheit = (celsius * 9/5) + 32
6     return fahrenheit
7
```

## One-shot learning

With this approach, a single example is given, aiding the model in generating a more context-aware response. Building upon the previous zero-shot example, you might provide an example of a temperature conversion function and then ask Copilot to create another similar function. Here's how it could look:

```
# Example: Function to convert Fahrenheit to Celsius
def fahrenheit_to_celsius(fahrenheit):
    celsius = (fahrenheit - 32) * 5/9
    return celsius

# Now I need a function to convert Celsius to Fahrenheit

def celsius_to_fahrenheit(celsius):
    fahrenheit = celsius * 9/5 + 32
    return fahrenheit
```

## Few-shot learning

In this method, Copilot is presented with several examples, which strike a balance between zero-shot unpredictability and the precision of fine-tuning. Let's say you want to generate code that sends you a greeting depending on the time of the day. Here's a few-shot version of that prompt:

```
greeting.py X
greeting.py
1  # Example 1: Greeting message for morning
2  # Input: 9 AM
3  # Output: "Good morning!"
4
5  # Example 2: Greeting message for afternoon
6  # Input: 2 PM
7  # Output: "Good afternoon!"
8
9  # Example 3: Greeting message for evening
10 # Input: 7 PM
11 # Output: "Good evening!"
12
13 # Now, generate a python code that takes the current time as input using the datetime module
14 # and returns the appropriate greeting message
15
16 # Solution:
17 # Import datetime module
18 import datetime
19
20 # Get current time
21 current_time = datetime.datetime.now()
22
23 # Get current hour
24 current_hour = current_time.hour
25
26 # Check if it is morning (before 12 PM)
27 if current_hour < 12:
28     print("Good morning!")
29
30 # Check if it is afternoon (between 12 PM and 4 PM)
31 elif current_hour < 16:
32     print("Good afternoon!")
33
34 # Check if it is evening (after 4 PM)
35 elif current_hour < 21:
36     print("Good evening!")
37
38 # Else it is night time
39 else:
40     print("Good night!")
41
```

Now that you know how Copilot uses your prompts to learn, let's take an in-depth look at how it actually uses your prompt to suggest code for you.

## Next unit: GitHub Copilot user prompt process flow

[< Previous](#)[Next >](#)