6.3 - Apply prompt engineering with Azure OpenAI Service

- Overview
- Introduction
- Understand prompt engineering
- What is prompt engineering
    - Considerations for API endpoints
    - Adjusting model parameters
- Write more effective prompts
    - Provide clear instructions
    - Format of instructions
        - Use section markers
- Primary, supporting, and grounding content
    - Cues
- Provide context to improve accuracy
    - Request output composition
    - System message
    - Conversation history
    - Few shot learning
    - Break down a complex task
        - Chain of thought
- Exercise - Utilize prompt engineering in your application
    - Provision an Azure OpenAI resource
    - Deploy a model
    - Configure your application
    - Add code to use the Azure OpenAI service

Overview

Prompt engineering in Azure OpenAI is a technique that involves designing prompts for natural language processing models. This process improves accuracy and relevancy in responses, optimizing the performance of the model.

By the end of this module, you'll be able to:

- Understand the concept of prompt engineering and its role in optimizing Azure OpenAI models' performance.
- Know how to design and optimize prompts to better utilize AI models.
- Include clear instructions, request output composition, and use contextual content to improve the quality of the model's responses.

Introduction

Response quality from AI models in Azure OpenAI depends on the quality of the prompt provided. Improving prompt quality through various techniques is called prompt engineering.

In this module, you'll learn about prompt engineering and how it can be used to optimize the performance of Azure OpenAI models. Prompt engineering involves designing and optimizing prompts to better utilize Azure OpenAI models.

Understand prompt engineering

The quality of the input prompts we send to an AI model, like those available in Azure OpenAI, directly influences the quality of what we get back. By carefully constructing the prompts we send to the model, the model can provide better and more interesting responses.

What is prompt engineering

Prompt engineering is the process of designing and optimizing prompts to better utilize AI models. Designing effective prompts is critical to the success of prompt engineering, and it can significantly improve the AI model's performance on specific tasks. Providing relevant, specific, unambiguous, and well structured prompts can help the model better understand the context and generate more accurate responses.

For example, if we want an OpenAI model to generate product descriptions, we can provide it with a detailed description that describes the features and benefits of the product. By providing this context, the model can generate more accurate and relevant product descriptions.

Prompt engineering can also help mitigate bias and improve fairness in AI models. By designing prompts that are diverse and inclusive, we can ensure that the model isn't biased towards a particular group or perspective.

> Important: No matter how good of a prompt you can design, **responses from AI models should never be taken as fact or completely free from bias.** Always use AI responsibly. For more information, see Microsoft's transparency note on Azure OpenAI and Microsoft's AI principles.

In addition, prompt engineering can help us understand which references the model uses to generate its response. Generative AI models have a ton of parameters and the logic it follows is largely unknown to users, so it can be confusing how it arrives at the response it gives.

By designing prompts that are easy to understand and interpret, we can help humans better understand how the model is generating its responses. This can be particularly important in domains such as healthcare, where it's critical to understand how the model is making decisions.

There are different methods to utilize when engineering your own prompts, many of which are covered in upcoming units of this module. These include providing instructions, contextual content, cues or few-shot examples, and correctly ordering content in your prompt. The methods covered here aren't exhaustive as this area is a nuanced and fluid topic.

Considerations for API endpoints

The examples in the rest of this module will be focused on `ChatCompletion`. **It's worth noting that `ChatCompletion` can also be used for non chat scenarios**, where any instructions are included in the system message and user content is provided in the user role message. Most of these examples can be altered to use the `Completion` endpoint, if desired.

In terms of model availability, the `Completion` endpoint can be used with `gpt-3` and earlier, and `ChatCompletion` can be used with `gpt-35-turbo` and later models.

Adjusting model parameters

In addition to techniques discussed in this module, adjusting parameters of the model can have a significant impact on the response.

In particular, `temperature` and `top_p` (top_probability) are the most likely to impact a model's response as they both control randomness in the model, but in different ways.

Higher values produce more creative and random responses, but will likely be less consistent or focused. Responses expected to be fictional or unique benefit from higher values for these parameters, whereas content desired to be more consistent and concrete should use lower values.

Try adjusting these parameters with the same prompt to see how they impact the response. It's recommended to change either `temperature` or `top_p` at a time, but not both.

Write more effective prompts

Azure OpenAI models are capable of generating responses to natural language queries with remarkable accuracy. However, the quality of the responses depends largely on how well the prompt is written. Developers can optimize the performance of Azure OpenAI models by using different techniques in their prompts, resulting in more accurate and relevant responses.

Provide clear instructions

Asking the Azure OpenAI model clearly for what you want is one way to get desired results. By being as descriptive as possible, the model can generate a response that most closely matches what you're looking for.

For example, say you want to create a product description for a new water bottle. Consider the answer associated with each prompt:

Prompt

```
Write a product description for a new water bottle
```

Response

```
Introducing the latest addition to our product line - the innovative and eco-friendly
water bottle. Made from high-quality, BPA-free materials, this bottle is designed to
keep your beverages at the perfect temperature for hours. Its sleek and stylish design
makes it the perfect accessory for any outdoor activity or workout session. With its
convenient carrying handle and leak-proof lid, you can take it with you wherever you go,
without worrying about spills or leaks. Plus, its easy-to-clean design ensures that you
can use it again and again, making it the perfect choice for anyone looking for a
reliable and sustainable water bottle.
```

Versus a similar prompt, with clear instructions.

Prompt

```
Write a product description for a new water bottle that is 100% recycled. Be sure to
include that it comes in natural colors with no dyes, and each purchase removes 10
pounds of plastic from our oceans
```

Response

```
Introducing our newest water bottle, the eco-friendly choice for the conscious consumer.
Made from 100% recycled materials and available in natural colors with no dyes, this
bottle is not only stylish but also sustainable. Each purchase of this bottle removes 10
pounds of plastic from our oceans, making it a responsible choice for those who care
about the environment. Its durable and leak-proof design makes it perfect for outdoor
activities, while its easy-to-carry handle makes it convenient to take on the go. Join
us in our mission to reduce plastic waste and choose this eco-friendly water bottle for
your daily hydration needs.
```

Including specifics that need to be included lets the model know what information to add to the description, making it more accurate to the new product.

This method can be extrapolated to include complex instructions, such as a bulleted list of details to include, length of response, or desired formats to be included in the output. Try asking for exactly what you want to see in the result, and you may be surprised at how well the model satisfies your request.

Format of instructions

How instructions are formatted can impact how the model interprets the prompt. Recency bias can affect models, where information located towards the end of the prompt can have more influence on the output than information at the beginning.

You may get better responses by repeating the instructions at the end of the prompt and assessing how that affects the generated response.

This recency bias can also come into play when using `ChatCompletion` in a chat scenario, where **more recent messages in the conversation included in the prompt have a greater impact on the response**.

The next unit goes more in depth on using conversations to improve response quality, but putting important information closer to the end of the prompt may result in a better response.

Use section markers

A specific technique for formatting instructions is to **split the instructions at the beginning or end of the prompt, and have the user content contained within `---` or `###` blocks.** These tags allow the model to more clearly differentiate between instructions and content. For example:

```
Translate the text into French

---
What's the weather going to be like today?
---
```

Note

Best practices for section markers may change with future versions.

Primary, supporting, and grounding content

Including content for the model to use to respond with allows it to answer with greater accuracy. This content can be thought of in two ways: primary and supporting content.

Primary content refers to content that is the subject of the query, such as a sentence to translate or an article to summarize. This content is often included at the beginning or end of the prompt (as an instruction and differentiated by `---` blocks), with instructions explaining what to do with it.

For example, say we have a long article that we want to summarize.

We could put it in a `---` block in the prompt, then end with the instruction.

```
---
<insert full article here, as primary content>
---

Summarize this article and identify three takeaways in a bulleted list
```

Supporting content is content that may alter the response, but isn't the focus or subject of the prompt. Examples of supporting content include things like names, preferences, future date to include in the response, and so on.

Providing supporting content allows the model to respond more completely, accurately, and be more likely to include the desired information.

For example, given a very long promotional email, the model is able to extract key information. If you then add supporting content to the prompt specifying something specific you're looking for, the model can provide a more useful response.

In this case the email is the primary content, with the specifics of what you're interested in as the supporting content

```
---
<insert full email here, as primary content>
---
<the next line is the supporting content>
Topics I'm very interested in: AI, webinar dates, submission deadlines

Extract the key points from the above email, and put them in a bulleted list:
```

Grounding content allows the model to provide reliable answers by providing content for the model to draw answer from. Grounding content could be an essay or article that you then ask questions about, a company FAQ document, or information that is more recent than the data the model was trained on.

If you need more reliable and current responses, or you need to reference unpublished or specific information, grounding content is highly recommended.

Grounding content differs from primary content as it's the source of information to answer the prompt query, instead of the content being operated on for things like summarization or translation.

For example, when provided an unpublished research paper on the history of AI, it can then answer questions using that grounding content.

```
---
<insert unpublished paper on the history of AI here, as grounding content>
---

Where and when did the field of AI start?
```

This grounding data allows the model to give more accurate and informed answers that may not be part of the dataset it was trained on.

Cues

Cues are leading words for the model to build upon, and often help shape the response in the right direction. They often are used with instructions, but don't always. **Cues are particularly helpful if prompting the model for code generation.**

Current Azure OpenAI models can generate some interesting code snippets, however code generation will be covered in more depth in a future module.

For example, if you're wanting help creating a SQL query, provide instructions of what you need along with the beginning of the query:

```
Write a join query to get customer names with purchases in the past 30 days between
tables named orders and customer on customer ID.

SELECT
```

The model response picks up where the prompt left off, continuing in SQL, even though we never asked for a specific language.

Other examples could be to help with python code, by giving code comments about the desired app and including `import` as a leading word at the end of the prompt, or similar in your desired language.

Another example, given a large collection of customer reviews in a prompt, and ending with:

```
Summarize the reviews above:
Most common complaints:
-
```

The model then knows to complete the statements based off the context provided in the reviews.

Provide context to improve accuracy

By providing context to the AI model, it allows the model to better understand what you are asking for or what it should know to provide the best answer.

Context can be provided in several ways.

Request output composition

Specifying the structure of your output can have a large impact on your results.

This could include something like asking the model to cite their sources, write the response as an email, format the response as a SQL query, classify sentiment into a specific structure, and so on.

For example:

Prompt

```
Write a table in markdown with 6 animals in it, with their genus and species
```

Response

```
| Animal           | Genus          | Species        |
| ---------------- |:--------------:|:--------------:|
| Red Panda        | Ailurus        | fulgens        |
| African Elephant | Loxodonta      | africana       |
| Snow Leopard     | Panthera       | uncia          |
| Arctic Fox       | Vulpes         | lagopus        |
| Chimpanzee       | Pan            | troglodytes    |
| Siberian Tiger   | Panthera       | tigris altaica |
```

This technique can be used with custom formats, such as a JSON structure:

Prompt

```
Put two fictional characters into JSON of the following format

{
  firstNameFictional:
  jobFictional:
}
```

Response

```
Here's an example of how you can put two fictional characters into JSON format:
```

```
{
  "firstNameFictional": "Henry",
  "jobFictional": "Wizard"
},
{
  "firstNameFictional": "Frank",
  "jobFictional": "Hero"
}
```

System message

The system message is included at the beginning of a prompt and is designed to give the model instructions, perspective to answer from, or other information helpful to guide the model's response.

This system message might include tone or personality, topics that shouldn't be included, or specifics (like formatting) of how to answer.

For example, you could give it some of the following system messages:

- "I want you to act like a command line terminal. Respond to commands exactly as cmd.exe would, in one unique code block, and nothing else."
- "I want you to be a translator, from English to Spanish. Don't respond to anything I say or ask, only translate between those two languages and reply with the translated text."
- "Act as a motivational speaker, freely giving out encouraging advice about goals and challenges. You should include lots of positive affirmations and suggested activities for reaching the user's end goal."

Other example system messages are available at the top of the chat window in Azure AI Foundry portal by selecting the **Prompt samples** button. Try defining your own system prompt that specifies a unique response, and chat with the model to see how responses differ.

The `ChatCompletion` endpoint enables including the system message by using the `System` chat role.

```
response = openai.ChatCompletion.create(
    model="gpt-35-turbo",
    messages=[
        {"role": "system", "content": "You are a casual, helpful assistant. You will
talk like an American old western film character."},
        {"role": "user", "content": "Can you direct me to the library?"}
    ]
)
```

Response

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "message": {
        "content": "Well howdy there, stranger! The library, huh?
                    Y'all just head down the main road till you hit the town
                    square. Once you're there, take a left and follow the street
```

```
          for a spell. You'll see the library on your right, can’t
          miss it. Happy trails!",
    "role": "assistant"
   }
  }
 ],
 ...
}
```

System messages can significantly change the response, both in format and content.

Try defining a clear system message for the model that explains exactly what kind of response you expect, and what you do or don't want it to include.

Conversation history

Along with the system message, other messages can be provided to the model to enhance the conversation. Conversation history enables the model to continue responding in a similar way (such as tone or formatting) and allow the user to reference previous content in subsequent queries. This history can be provided in two ways: from an actual chat history, or from a user defined example conversation.

Chat interfaces that use OpenAI models, such as ChatGPT and the chat playground in Azure AI Foundry portal, include conversation history automatically which results in a richer, more meaningful conversation. In the **Parameters** section of the chat playground, you can specify how many past messages you want included. Try reducing that to 1 or increasing to max to see how different amounts of history impact the conversation.

> Note: More conversation history included in the prompt means a larger number of input tokens are used. You will have to determine what the correct balance is for your use case, considering the token limit of the model you are using.

Chat systems can also utilize the summarization capabilities of the model to save on input tokens.

An app can choose to summarize past messages and include that summary in the conversation history, then provide only the past couple messages verbatim to the model.

Few shot learning

Using a user defined example conversation is what is called *few shot learning*, which provides the model examples of how it should respond to a given query. These examples serve to train the model how to respond.

For example, by providing the model a couple prompts and the expected response, it continues in the same pattern without instructing it what to do:

```
User: That was an awesome experience
Assistant: positive
User: I won't do that again
Assistant: negative
User: That was not worth my time
Assistant: negative
```

```
User: You can't miss this
Assistant:
```

If the model is provided with just `You can't miss this` with no additional context from few shot learning, the response isn't likely to be useful.

In practical terms, conversation history and few shot learning are sent to the model in the same way; **each user message and assistant response is a discrete message in the message object.**

The `ChatCompletion` endpoint is optimized to include message history, regardless of if this message history is provided as few shot learning, or actual conversation history.

```
response = openai.ChatCompletion.create(
    model="gpt-35-turbo",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "That was an awesome experience"},
        {"role": "assistant", "content": "positive"},
        {"role": "user", "content": "I won't do that again"},
        {"role": "assistant", "content": "negative"},
        {"role": "user", "content": "That was not worth my time"},
        {"role": "assistant", "content": "negative"},
        {"role": "user", "content": "You can't miss this"}
    ]
)
```

Break down a complex task

Another technique for improved interaction is to **divide complex prompts into multiple queries.**

This allows the model to better understand each individual part, and can improve the overall accuracy. Dividing your prompts also allows you to include the response from a previous prompt in a future prompt, and using that information in addition to the capabilities of the model to generate interesting responses.

For example, you could ask the model `Doug can ride down the zip line in 30 seconds, and takes 5 minutes to climb back up to the top. How many times can Doug ride the zip line in 17 minutes?`. The result is likely `3`, which if Doug starts at the top of the zip line is incorrect.

A more informative answer could come from asking it multiple questions, about the round trip time to get back to the top of the zip line, and how to account for the fact that Doug starts at the top. Breaking down this problem reveals that Doug can, in fact, ride the zip line four times.

Chain of thought

One useful method to help you break down your task effectively is to ask the model to explain its chain of thought.

Asking a model to respond with the step by step process by which it determined the response is a helpful way to understand how the model is interpreting the prompt.

By doing so, you can see where the model made an incorrect logical turn and better understand how to change your prompt to avoid the error. This technique can include asking it to cite its sources, like Bing chat does (which uses a GPT-4 generation model), and giving reasoning for why it determined its answer.

The chain of thought prompting technique is best used to help you iterate and improve on your prompts to get the highest quality answer from the model.

For example, asking the model `What sport is easiest to learn but hardest to master?` results in response with an answer, and a small explanation of why.

However, when prompted with `What sport is easiest to learn but hardest to master? Give a step by step approach of your thoughts, ending in your answer`, the response is a complete explanation of how it arrived at its answer.

```
Sure, here's a step-by-step approach to answering the question:
1. Identify sports that are popular and have a large following.
2. Consider the basic mechanics of each sport and how easy they are to learn.
3. Think about the various techniques and skills required to play each sport at a high level, and how difficult they are to master.
4. Compare the relative ease of learning each sport to the difficulty of mastering it.
5. Based on the above analysis, determine which sport is easiest to learn but hardest to master.

Using this approach, I would say that the sport of golf is easiest to learn but hardest to master. Here's why:
1. Golf is a popular sport with a large following.
2. The basic mechanics of hitting a golf ball are relatively simple and can be learned quickly with some practice.
3. However, achieving consistency and mastering the various shots and techniques required to play at a high level can take years of practice and dedication.
4. Compared to other sports, the difference between a beginner and an intermediate golfer is relatively small, while the difference between an intermediate and an advanced golfer is much larger.
5. Therefore, I would say that golf is the sport that is easiest to learn but hardest to master.
```

Knowing the steps of reason the model used to form its response allows us to identify where we need to split the prompt, provide more information, or clarify instructions.

Exercise - Utilize prompt engineering in your application

With the Azure OpenAI Service, developers can create chatbots and other applications that excel at understanding natural human language through the use of REST APIs or language specific SDKs. When working with these language models, how developers shape their prompt greatly impacts how the generative AI model will respond. Azure OpenAI models are able to tailor and format content, if requested in a clear and concise way. In this exercise, you'll learn how to connect your application to Azure OpenAI and see how different prompts for similar content help shape the AI model's response to better satisfy your requirements.

In the scenario for this exercise, you will perform the role of a software developer working on a wildlife marketing campaign. You are exploring how to use generative AI to improve advertising emails and categorize articles that might apply to your team. The prompt engineering techniques used in the exercise can be applied similarly for a variety of use cases.

This exercise will take approximately **30** minutes.

Provision an Azure OpenAI resource

If you don't already have one, provision an Azure OpenAI resource in your Azure subscription.

1. Sign into the **Azure portal** at `https://portal.azure.com`.
2. Create an **Azure OpenAI** resource with the following settings:
    - **Subscription**: *Select an Azure subscription that has been approved for access to the Azure OpenAI service*
    - **Resource group**: *Choose or create a resource group*
    - **Region**: *Make a **random** choice from any of the following regions**
        - Canada East
        - East US
        - East US 2
        - France Central
        - Japan East
        - North Central US
        - Sweden Central
        - Switzerland North
        - UK South
    - **Name**: *A unique name of your choice*
    - **Pricing tier**: Standard S0

    > - Azure OpenAI resources are constrained by regional quotas. The listed regions include default quota for the model type(s) used in this exercise. Randomly choosing a region reduces the risk of a single region reaching its quota limit in scenarios where you are sharing a subscription with other users. In the event of a quota limit being reached later in the exercise, there's a possibility you may need to create another resource in a different region.

3. Wait for deployment to complete. Then go to the deployed Azure OpenAI resource in the Azure portal.

Deploy a model

Next, you will deploy an Azure OpenAI model resource from the CLI. Refer to this example and replace the following variables with your own values from above:

```
az cognitiveservices account deployment create \
    -g *Your resource group* \
    -n *Name of your OpenAI service* \
    --deployment-name gpt-35-turbo \
    --model-name gpt-35-turbo \
    --model-version 0125 \
    --model-format OpenAI \
    --sku-name "Standard" \
   --sku-capacity 5
```

> * Sku-capacity is measured in thousands of tokens per minute. A rate limit of 5,000 tokens per minute is more than adequate to complete this exercise while leaving capacity for other people using the same subscription.

> If you see a warnings about net7.0 framework being out of support, you can disregard them for this exercise.

Configure your application

Applications for both C# and Python have been provided, and both apps feature the same functionality. First, you'll complete some key parts of the application to enable using your Azure OpenAI resource with asynchronous API calls.

1. In Visual Studio Code, in the **Explorer** pane, browse to the **Labfiles/01-app-develop** folder and expand the **CSharp** or **Python** folder depending on your language preference. Each folder contains the language-specific files for an app into which you're you're going to integrate Azure OpenAI functionality.
2. Right-click the **CSharp** or **Python** folder containing your code files and open an integrated terminal. Then install the Azure OpenAI SDK package by running the appropriate command for your language preference:

   C#:

   ```
   dotnet add package Azure.AI.OpenAI --version 2.0.0
   ```

   Python:

   ```
   pip install openai==1.54.3
   ```

3. In the **Explorer** pane, in the **CSharp** or **Python** folder, open the configuration file for your preferred language
   - **C#**: appsettings.json
   - **Python**: .env
4. Update the configuration values to include:
   - The **endpoint** and a **key** from the Azure OpenAI resource you created (available on the **Keys and Endpoint** page for your Azure OpenAI resource in the Azure portal)
   - The **deployment name** you specified for your model deployment.
5. Save the configuration file.

Add code to use the Azure OpenAI service

Now you're ready to use the Azure OpenAI SDK to consume your deployed model.

1. In the **Explorer** pane, in the **CSharp** or **Python** folder, open the code file for your preferred language, and replace the comment ***Add Azure OpenAI package*** with code to add the Azure OpenAI SDK library:

C#: Program.cs

```
// Add Azure OpenAI packages using Azure.AI.OpenAI; using OpenAI.Chat;
```

Python: [application.py]()

```python
# Add Azure OpenAI package
from openai import AsyncAzureOpenAI
```

2. In the code file, find the comment ***Configure the Azure OpenAI client***, and add code to configure the Azure OpenAI client:

C#: Program.cs

```
// Configure the Azure OpenAI client AzureOpenAIClient azureClient = new (new Uri(oaiEndpoint), new ApiKeyCredential(oaiKey)); ChatClient chatClient = azureClient.GetChatClient(oaiDeploymentName); ChatCompletion completion = chatClient.CompleteChat( [ new SystemChatMessage(systemMessage), new UserChatMessage(userMessage), ]);
```

Python: [application.py]()

```python
# Configure the Azure OpenAI client
client = AsyncAzureOpenAI(
    azure_endpoint=azure_oai_endpoint,
    api_key=azure_oai_key,
    api_version="2024-02-15-preview"
)
```

3. In the function that calls the Azure OpenAI model, under the comment ***Get response from Azure OpenAI***, add the code to format and send the request to the model.

C#: Program.cs

```
// Get response from Azure OpenAI Console.WriteLine($"{completion.Role}: {completion.Content[0].Text}");
```

Python: [application.py]()

```python
# Get response from Azure OpenAI
messages = [
    {"role": "system", "content": system_message},
    {"role": "user", "content": user_message},
]

print("\nSending request to Azure OpenAI model...\n")

# Call the Azure OpenAI model
response = await client.chat.completions.create(
    model=model,
    messages=messages,
    temperature=0.7,
    max_tokens=800
)
```

```
4. Save the changes to the code file.

### Run your application

Now that your app has been configured, run it to send your request to your model and observe the response. You'll notice the only difference between the different options is the content of the prompt, all other parameters (such as token count and temperature) remain the same for each request.

1. In the folder of your preferred language, open `system.txt` in Visual Studio Code. For each of the interactions, you'll enter the **System message** in this file and save it. Each iteration will pause first for you to change the system message.
2. In the interactive terminal pane, ensure the folder context is the folder for your preferred language. Then enter the following command to run the application.

    - **C#**: `dotnet run`
    - **Python**: `python application.py`

    > **Tip**: You can use the **Maximize panel size** (**^**) icon in the terminal toolbar to see more of the console text.

3. For the first iteration, enter the following prompts:

    **System message**
    `You are an AI assistant`

    **User message:**
    `Write an intro for a new wildlife Rescue`

4. Observe the output. The AI model will likely produce a good generic introduction to a wildlife rescue.
5. Next, enter the following prompts which specify a format for the response:

    **System message**
    `You are an AI assistant helping to write emails`

    **User message:**
    `Write a promotional email for a new wildlife rescue, including the following:  - Rescue name is Contoso  - It specializes in elephants  - Call for donations to be given at our website`

    > **Tip**: You may find the automatic typing in the VM doesn't work well with multiline prompts. If that is your issue, copy the entire prompt then paste it into Visual Studio Code.

6. Observe the output. This time, you'll likely see the format of an email with the specific animals included, as well as the call for donations.
7. Next, enter the following prompts that additionally specify the content:
```

~~~
    **System message**
    `You are an AI assistant helping to write emails`

    **User message:**
    `Write a promotional email for a new wildlife rescue, including the following:  - Rescue name is Contoso  - It specializes in elephants, as well as zebras and giraffes  - Call for donations to be given at our website  \n Include a list of the current animals we have at our rescue after the signature, in the form of a table. These animals include elephants, zebras, gorillas, lizards, and jackrabbits.`

8. Observe the output, and see how the email has changed based on your clear instructions.
9. Next, enter the following prompts where we add details about tone to the system message:

    **System message**
    `You are an AI assistant that helps write promotional emails to generate interest in a new business. Your tone is light, chit-chat oriented and you always include at least two jokes.`

    **User message:**
    `Write a promotional email for a new wildlife rescue, including the following:  - Rescue name is Contoso  - It specializes in elephants, as well as zebras and giraffes  - Call for donations to be given at our website  \n Include a list of the current animals we have at our rescue after the signature, in the form of a table. These animals include elephants, zebras, gorillas, lizards, and jackrabbits.`

10. Observe the output. This time you'll likely see the email in a similar format, but with a much more informal tone. You'll likely even see jokes included!
11. For the final iteration, we're deviating from email generation and
exploring _grounding context_. Here you provide a simple system message, and change the app to provide the grounding context as the beginning of the user prompt. The app will then append the user input, and extract information from the grounding context to answer our user prompt.
12. Open the file `grounding.txt` and briefly read the grounding context you'll be inserting.
13. In your app immediately after the comment **_Format and send the request to the model_** and before any existing code, add the following code snippet to read text in from `grounding.txt` to augment the user prompt with the grounding context.

    **C#**: Program.cs
    `// Format and send the request to the model Console.WriteLine("\nAdding grounding context from grounding.txt"); string groundingText = System.IO.File.ReadAllText("grounding.txt"); userMessage = groundingText + userMessage;`

    **Python**: [[application.py]]
    ```python
    # Format and send the request to the model
    print("\nAdding grounding context from grounding.txt")
~~~

~~~~
    # Read grounding text from the file
    grounding_text = open(file="grounding.txt", encoding="utf8").read().strip()

    # Combine grounding text with the user message
    user_message = grounding_text + user_message
    ```

14. Save the file and rerun your app.
15. Enter the following prompts (with **system message** still being entered and saved in [[Data and ML Engineering/Courses/Designing and Implementing a Microsoft Azure AI Solution/mslearn-openai/Labfiles/03-prompt-engineering/Python/system.txt|system.txt]]).

    **System message**
    `You're an AI assistant who helps people find information. You'll provide answers from the text provided in the prompt, and respond concisely.`

    **User message:**
    `What animal is the favorite of children at Contoso?`

> **Tip**: If you would like to see the full response from Azure OpenAI, you can set the **printFullResponse** variable to `True`, and rerun the app.


---
## Knowledge Check

![[Pasted image 20250102151751.png]]


---
## Summary

In this module, you learned about prompt engineering and how it can be used to optimize the performance of Azure OpenAI models. Prompt engineering involves designing and optimizing prompts to better utilize AI models. By providing relevant, specific, unambiguous, and well-structured prompts, the model can better understand the context and generate more accurate responses.

The main takeaways from this module are that prompt engineering is a critical process for optimizing the performance of Azure OpenAI models. Providing clear instructions, requesting output composition, adjusting model parameters, and using contextual content can all help improve the accuracy and relevance of the model's responses.

It's important to use AI responsibly and to always keep in mind that responses from AI models should never be taken as fact or free from bias.

Additional Reading:
- [Microsoft's AI principles](https://www.microsoft.com/ai/responsible-ai)
- [Transparency note on Azure OpenAI](https://learn.microsoft.com/en-us/legal/cognitive-services/openai/transparency-note)
~~~~

```
- [Azure OpenAI documentation on prompt engineering](https://learn.microsoft.com/en-us/azure/cognitive-services/openai/concepts/advanced-prompt-engineering)
- [Azure AI Foundry portal](https://ai.azure.com/resource/playground)



---
✍️ Compiled by [Kenneth Leung](https://github.com/kennethleungty) (2025)
```