Open in Colab
(https://colab.research.google.com/github/alexandrosanat/prompt-demo/blob/main/prompt_engineering.ipynb)

Practical Prompt Engineering

In this tutorial, you'll learn how to:

- Apply prompt engineering techniques to practical, real-world examples
- Tap into the power of roles in messages to go beyond using singular role prompts
- Use numbered steps, delimiters, few-shot prompting and other techniques to improve your results
- Understand and use chain-of-thought prompting to add more context

⚠️ You can follow along by opening this notebook on google colab.

First let's install and then import all the libraries we're going to use...

In [ ]:
```
!pip install --upgrade openai
!pip install jupyter-black
```

In [3]:
```
import os, json
import openai

from google.colab import userdata

%load_ext jupyter_black
```

and retrieve our OpenAI API key...

In [4]:
```
# openai.api_key = os.environ.get("OPENAI_API_KEY")
openai.api_key = userdata.get('OPENAI_API_KEY')
```

Chat Completions API

Chat models like GPT take a list of messages as input and return a model-generated message as output.

Chat completion API documentation (https://platform.openai.com/docs/guides/gpt):

In [6]:
```
client = openai.OpenAI(api_key=openai.api_key)
```

In [7]:
```
response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Who won the world series in 2020?"},
    ],
)
```

> **Tip:** The 3 types of roles you can use are:
>
> - **System role:** Allows you to specify the way the model answers questions. Typically we can use it to determine what **role** the AI should play and how it should behave generally.
> - **User role:** Equivalent to the queries made by the user
> - **Assistant role:** The model's responses

An example Chat Completions API response (https://platform.openai.com/docs/guides/gpt/chat-completions-response-format) looks as follows:

In [8]:
```
response = {
    "choices": [
        {
            "finish_reason": "stop",
            "index": 0,
            "message": {
                "content": "The Los Angeles Dodgers won the World Series in
                "role": "assistant",
            },
        }
    ],
    "created": 1677664795,
    "id": "chatcmpl-7QyqpwdfhqwajicIEznoc6Q47XAyW",
    "model": "gpt-3.5-turbo-0613",
    "object": "chat.completion",
    "usage": {"completion_tokens": 17, "prompt_tokens": 57, "total_tokens":
}
```

The assistant's reply can be extracted with:

In [9]:
```
content = response["choices"][0]["message"]["content"]
```

In [10]:
```
content
```

Out[10]:
```
'The Los Angeles Dodgers won the World Series in 2020.'
```

The next function allows us to send prompts to the model and get a response back.

We have specified the following model parameters:

- Model: **GPT-3.5-turbo**
- Temperature: **0**

The function takes as inputs prompt and a list of previous messages, and returns a new list of messsages that include the prompt and the response from the model:

In [11]:
```python
def get_completion_from_messages(
    prompt, messages, model="gpt-3.5-turbo", temperature=0
):
    if not isinstance(messages, list):
        messages = [messages]

    messages.append(prompt)

    response = client.chat.completions.create(
        model=model, messages=messages, temperature=temperature
    )

    message = response.choices[0].message

    print(message.content)

    messages.append({"role": message.role, "content": message.content})

    return messages
```

Let's test it!

In [12]:
```python
# fmt: off

messages = [
    {"role": "user", "content": "My name is Alex"},
    {"role": "assistant","content": "Nice to meet you, Alex! How can I ass
]

prompt = {"role": "user", "content": "What is my name?"}

response = get_completion_from_messages(prompt, messages)
```

```
Your name is Alex.
```

In [13]:
```python
response[-1]
```

Out[13]:
```
{'role': 'assistant', 'content': 'Your name is Alex.'}
```

Ways to Engineer your Prompts ⚙️

We will now look at a few different examples of how you can improve your promtps to become the utlimate prompt engineer!

Let's explore the following techniques:

- **Zero-shot prompting**
- **Role prompting**
- **Detail & Specificity**
- **Few-shot prompting**
- **Using delimiters**
- **Chain-of-thought prompting (CoT)**

1. Zero-shot prompting 💬

First, we have a list of reviews written below. Our task is to summarise them using ChatGPT.
Let's see what we can do.

In [14]:
```
reviews = [
    (
        "Date: December 15, 2021; Username: John123; Review: I am absolutel
    with this savings product! The interest rates are fantastic, and it has
    me grow my savings significantly. Highly recommend!"
    ),
    (
        "Date: November 28, 2021; Username: Sarah77; Review: I must say, I
    disappointed with this savings product. The promised returns were not a
    impressive as advertised, and the fees associated with it added up quic
    Not worth it."
    ),
    (
        "Date: January 5, 2022; Username: AlexSmith; Review: My opinion? I
    indifferent about this savings product. It's just like any other basic
    account out there. Nothing special, but it does the job."
    ),
]
```

In [15]:
```
content = f"""Summarise these 3 reviews:{reviews}"""

prompt = {
    "role": "user",
    "content": content,
}
```

In [16]:
```
messages = get_completion_from_messages(prompt, [])
```

```
Review 1: John123 is extremely satisfied with the savings product, praising t
he fantastic interest rates and significant growth of their savings. They hig
hly recommend it.

Review 2: Sarah77 expresses disappointment with the savings product, stating
that the promised returns were not as impressive as advertised and the associ
ated fees accumulated quickly. They believe it is not worth it.

Review 3: AlexSmith is indifferent about the savings product, considering it
to be similar to any other basic savings account. They find nothing special a
bout it but acknowledge that it fulfills its purpose.
```

Great! It provided a summary of each review.

What we just did is call **zero-shot prompting**, which is just a fancy way of saying that you're asking a normal question or simply describing a task.

But it's safe to say this is by no means useful enough yet to be used in a product.

Let's now dive into the other prompt engineering techniques to improve our results.

2. Role Prompting 🤖

In the next example we are setting the **role** of the model via the system message.

This is known as "Role Prompting", and is a general practice to help set the tone and context for the model's responses.

In [17]:
```
system_message = {
    "role": "system",
    "content": (
        """
        You are a Review Summariser.
        Your job is to process reviews from customers,
        and summarise them for analysis for the customer review team.
        """
    ),
}

messages = [system_message]
```

In [18]:
```
content = f"""Summarise these 3 reviews:{reviews}"""

prompt = {
    "role": "user",
    "content": content,
}

messages = get_completion_from_messages(prompt, messages)
```

```
Review 1: John123 is highly satisfied with the savings product. They are deli
ghted with the fantastic interest rates and mention that it has significantly
helped them grow their savings. They highly recommend it.

Review 2: Sarah77 is disappointed with the savings product. They mention that
the promised returns were not as impressive as advertised and the associated
fees added up quickly. They do not think it is worth it.

Review 3: AlexSmith is indifferent about the savings product. They state that
it is just like any other basic savings account and nothing special. However,
they mention that it does the job.

Overall, the reviews are mixed. One customer is highly satisfied, one is disa
ppointed, and one is indifferent about the savings product.
```

✅ By using a role prompt via the system message, the summary is more structured and much clearer to read for the user.

Examples of this could include:

- You are a financial advisor, qualified only to answer questions about finances
- You are a helpful assistant that answer queries about a user's transactions

3. Detail and Specificity 🔍

It is important that when you prompt, you are specific and as clear as possible.

Note that clear ≠ short: the more detail you add, the better the model will understand how you want it to behave.

In the next example, we'll try to add a category for each review: positive, neutral or negative:

In [19]:

```
system_message = {
    "role": "system",
    "content": """
    You are a Review Summariser. \
    Your job is to process reviews from customers, \
    and summarise them for analysis for the customer review team.
    Categorise each review as positive, neutral, or bad.
    """,
}
content = f"""Summarise these 3 reviews:{reviews}"""

prompt = {"role": "user", "content": content}

messages = [system_message]
messages = get_completion_from_messages(prompt, messages)
```

```
Review 1: Positive
Review 2: Bad
Review 3: Neutral
```

As you can see, the model has only output the categories, and we're now missing all the other information.

Let's be a bit more specific:

In [20]:

```
system_message = {
    "role": "system",
    "content": """
    You are a Review Summariser. Your job is to process reviews
    from customers and summarise them for analysis for the customer review

    For each review output the name and date.
    Also label each review as either 'Positive','Neutral', or 'Negative'.
    Then provide a summary of key points in a single sentence.
    """,
}

content = f"""Summarise these 3 reviews:{reviews}"""

prompt = {"role": "user", "content": content}

messages = [system_message]
messages = get_completion_from_messages(prompt, messages)
```

```
Review 1:
- Name: John123
- Date: December 15, 2021
- Sentiment: Positive
- Summary: John123 is delighted with the savings product, praising the fantas
tic interest rates and significant growth of savings.

Review 2:
- Name: Sarah77
- Date: November 28, 2021
- Sentiment: Negative
- Summary: Sarah77 is disappointed with the savings product, mentioning that
the promised returns were not as impressive as advertised and the associated
fees added up quickly.

Review 3:
- Name: AlexSmith
- Date: January 5, 2022
- Sentiment: Neutral
- Summary: AlexSmith is indifferent about the savings product, stating that i
t is similar to any other basic savings account and does the job.
```

Great! We now have all the information displayed appropriately for each review.

It's even structured the data for us in a list!

Let's see if we can further increase out specificity by numbering the steps as well as specifying the format that we want the model to use for our output:

In [21]:
```
system_message = {
    "role": "system",
    "content": """
    You are a Review Summariser. Your job is to process reviews
    from customers and summarise them for analysis for the customer review

    1. For each review output the name and date.
    2. Also label each review as either 'Positive','Neutral', or 'Negative
    3. Provide a summary of key points in a single sentence.

    Output: When outputting this information, use the following structure:
    - [Username]: name of customer who made review
    - [Date]: date of review, written as dd/mm/yyyy
    - [Sentiment]: review sentiment
    - [Key points]: key points of each review in one
    - [Summary]: a two sentense summary of the review
    """,
}
```

In [22]:
```
content = f"""Summarise these 3 reviews:{reviews} and return your response

prompt = {
    "role": "user",
    "content": content,
}

messages = [system_message]
messages = get_completion_from_messages(prompt, messages)
```

```
{
    "reviews": [
        {
            "Username": "John123",
            "Date": "15/12/2021",
            "Sentiment": "Positive",
            "Key points": "Delighted with the savings product, fantastic inte
rest rates, helped grow savings significantly",
            "Summary": "John123 is absolutely delighted with this savings pro
duct, praising the fantastic interest rates and significant growth of saving
s."
        },
        {
            "Username": "Sarah77",
            "Date": "28/11/2021",
            "Sentiment": "Negative",
            "Key points": "Disappointed with the savings product, promised re
turns not as impressive as advertised, fees added up quickly",
            "Summary": "Sarah77 is quite disappointed with this savings produ
```

As you can see, we get the output in JSON format.

This is quite powerful: when we structure our output as we have done in the previous section, the model can easily convert this into a structured format as shown.

We could now parse the model's output and use it in our product.

4. Few-shot prompting

Few-shot prompting is a prompt engineering technique where you provide example tasks and their expected outputs in your prompt.

So, instead of just describing the task you also provide examples:

In [23]:
```
system_message = {
    "role": "system",
    "content": """
    You are a Review Summariser. Your job is to process reviews
    from customers and summarise them for analysis for the customer review

    For each review output the name and date.
    Also label each review as either 'Positive','Neutral', or 'Negative'.
    Then provide a summary of key points in a single sentence.

    Example 1:
    "Date: April 1, 2022\nUsername: JaneDoe\nReview: I'm really not happy w
    The interest rates are lower than promised and the customer service is

    Output 1:
    - username: "JaneDoe"
    - review_date: "01/04/2022"
    - sentiment: "negative"
    - key_points: "Lower interest rates than promised, lacking customer ser
    - summary: JaneDoe is unhappy with the savings product, stating that th
    rates are lower than promised and the customer service is lacking.
    """,
}
```

In [24]:
```
content = f"""Summarise these 3 reviews:{reviews}"""

prompt = {"role": "user", "content": content}

messages = [system_message]
messages = get_completion_from_messages(prompt, messages)
```

```
- username: "John123"
- review_date: "15/12/2021"
- sentiment: "positive"
- key_points: "Fantastic interest rates, helped grow savings significantly"
- summary: John123 is delighted with the savings product, praising the fantas
tic interest rates and how it has significantly helped grow their savings.

- username: "Sarah77"
- review_date: "28/11/2021"
- sentiment: "negative"
- key_points: "Promised returns not as impressive as advertised, fees added u
p quickly"
- summary: Sarah77 is disappointed with the savings product, stating that the
promised returns were not as impressive as advertised and the fees associated
with it added up quickly.

- username: "AlexSmith"
- review_date: "05/01/2022"
- sentiment: "neutral"
- key_points: "Just like any other basic savings account, nothing special"
- summary: AlexSmith is indifferent about the savings product, describing it
as just like any other basic savings account out there, without anything spec
ial.
```

✅ Few-shot prompting can be really useful when we're asking the model to deal with input that it has never seen before. This is most powerful when we use a representative number of examples.

> For example, we might be dealing with a text categorisation scenario.

5. Using Delimiters

Delimiters can help to separate the content and examples from the task description. They can also make it possible to refer to specific parts of your prompt at a later point in the prompt. A delimiter can be any sequence of characters that usually wouldn't appear together, for example:

- >>>>>
- ====
- ####

The number of characters that you use doesn't matter too much, as long as you make sure that the sequence is relatively unique, otherwise this might confuse the model. Additionally, you can add labels just before or just after the delimiters:

- START CONTENT>>>>> content <<<<<END CONTENT
- ==== START content END ====
- #### START EXAMPLES examples #### END EXAMPLES

In [25]:

```
system_message = {
    "role": "system",
    "content": """
    You are a Review Summariser. Your job is to process reviews
    from customers and summarise them for analysis for the customer review

    For each review output the name and date.
    Also label each review as either 'Positive','Neutral', or 'Negative'.
    Then provide a summary of key points in a single sentence.

    #### START EXAMPLES ####

    ------ Example Inputs ------
    "Date: April 1, 2022\nUsername: JaneDoe\nReview: I'm really not happy w
    The interest rates are lower than promised and the customer service is

    ------ Example Outputs ------
    - username: "JaneDoe"
    - review_date: "01/04/2022"
    - sentiment: "negative"
    - key_points: "Lower interest rates than promised, lacking customer ser
    - summary: JaneDoe is unhappy with the savings product, stating that th
    rates are lower than promised and the customer service is lacking.

    #### END EXAMPLES ####

    """,
}
```

In [26]:
```python
content = f"""Summarise these 3 reviews:{reviews}"""

prompt = {"role": "user", "content": content}

messages = [system_message]
messages = get_completion_from_messages(prompt, messages)
```

```
- username: "John123"
  - review_date: "15/12/2021"
  - sentiment: "positive"
  - key_points: "Fantastic interest rates, helped grow savings significantly"
  - summary: John123 is delighted with the savings product, praising the fant
astic interest rates and how it has significantly grown their savings.

- username: "Sarah77"
  - review_date: "28/11/2021"
  - sentiment: "negative"
  - key_points: "Promised returns not as impressive as advertised, fees added
up quickly"
  - summary: Sarah77 is disappointed with the savings product, stating that t
he promised returns were not as impressive as advertised and the fees associa
ted with it added up quickly.

- username: "AlexSmith"
  - review_date: "05/01/2022"
  - sentiment: "neutral"
  - key_points: "Just like any other basic savings account, nothing special"
  - summary: AlexSmith is indifferent about the savings product, describing i
t as just like any other basic savings account with nothing special.
```

✅ Splitting the prompt into sections becomes increasingly important as your prompt grows and multiple instructinos are passed to the model!

Chain-of-thought prompting (CoT) 💭

To apply CoT, you prompt the model to generate intermediate results that then become part of the prompt in a second request. The increased context makes it more likely that the model will arrive at a useful output.

The smallest form of CoT prompting is **zero-shot CoT**, where you literally ask the model to *think step by step*.
This approach yields impressive results for mathematical tasks that LLMs otherwise often solve incorrectly.

More commonly, chain-of-thought operations are technically split into two stages:

- *Reasoning extraction*, where the model generates the increased context
- *Answer extraction*, where the model uses the increased context to generate the answer

Zero-shot CoT

In [28]:

```
content = """
I went to the market and bought 10 apples.
I gave 2 apples to the neighbor and 2 to the repairman.
I then went and bought 5 more apples and ate 1. How many apples did I remai
Let's think step by step.
"""

prompt = {"role": "user", "content": content}

messages = []
messages = get_completion_from_messages(prompt, messages)
```

```
Step 1: Bought 10 apples.
Step 2: Gave 2 apples to the neighbor. Remaining apples: 10 - 2 = 8 apples.
Step 3: Gave 2 apples to the repairman. Remaining apples: 8 - 2 = 6 apples.
Step 4: Bought 5 more apples. Total apples: 6 + 5 = 11 apples.
Step 5: Ate 1 apple. Remaining apples: 11 - 1 = 10 apples.

Therefore, you remained with 10 apples.
```

Summary 🔥

We have looked at all the main different techniques that you can use to engineer your prompts!

- **Zero-shot prompting:** Asking the language model a normal question without any additional context
- **Role prompting:** Specifying how the model will answer the questions
- **Detail & Specificity:** Breaking down a complex prompt into a series of small, specific steps
- **Few-shot prompting:** Conditioning the model on a few examples to boost its performance
- **Using delimiters:** Adding special tokens or phrases to provide structure and instructions to the model
- **Chain-of-thought prompting:** Prompt the model to generate intermediate results that then become part of the prompt in a second request

Prompt Engineering Tricks 🪄

- As the number of instructions increases, the model tends to forget some of them
- Repeating instructions might help
- Including instructions at the start vs at the end of the prompt will have a different effect
- Logically splitting the prompt into sections helps a lot
- Slightly tweaking the prompt might yield very different results
- Emphasising specific instructions works, eg. "ALWAYS include your thought process in your answer"
- Including context in the prompt helps reduce hallucinations
- Each use case is different, so you will need to iterate to understand what works for yours

Bonus: Chain-of-thought prompting in functions 💥

Chain of thought process is a powerful technique that has enabled LLMs to interact with functions and itegrate the provided context into their answers.

Let's try to demonstrate this by answering the following question:

What is the weather in Edinburgh?

First we need to modify our API call to enable function calling:

In [34]:
```python
def get_completion_from_messages_func(
    prompt, messages, tools, model="gpt-3.5-turbo-0613", temperature=0
):
    if not isinstance(messages, list):
        messages = [messages]

    messages.append(prompt)

    response = client.chat.completions.create(
        model=model,
        messages=messages,
        tools=tools,
        tool_choice="auto",
        temperature=temperature,
    )

    message = response.choices[0].message

    if message.tool_calls is not None:
        messages.append(
            {
                "role": message.role,
                "content": "null",
                "function_call": {
                    "name": message.tool_calls[0].function.name,
                    "arguments": message.tool_calls[0].function.arguments,
                },
            }
        )
        print(messages[-1])
    else:
        messages.append({"role": message.role, "content": message.content})
        print(messages[-1])

    return messages
```

Now let's define our python API that returns the weather using a location as input:

In [35]:
```python
def get_current_weather(location):
    if location == "Edinburgh":
        return {"temperature": 9, "unit": "celsius", "description": "Sunny"
    else:
        return None
```

Remember! LLMs can only understand *language* so the only way for our model to have knowledge of this function is if we describe it:

In [36]:
```
tools = [
    {
        "type": "function",
        "function": {
            "name": "get_current_weather",
            "description": "Get the current weather in a given location",
            "parameters": {
                "type": "object",
                "properties": {
                    "location": {
                        "type": "string",
                        "description": "The city and state, e.g. San Franci
                    },
                },
                "required": ["location"],
            },
        },
    }
]
```

Let's now send our question to the model:

In [37]:
```
content = """What is the weather in Edinburgh?"""

prompt = {"role": "user", "content": content}

messages = []

messages = get_completion_from_messages_func(prompt, messages, tools)
```

```
{'role': 'assistant', 'content': 'null', 'function_call': {'name': 'get_curre
nt_weather', 'arguments': '{\n  "location": "Edinburgh"\n}'}}
```

First we need to extract the function name and function arguments from the response:

In [38]:
```
messages
```

Out[38]:
```
[{'role': 'user', 'content': 'What is the weather in Edinburgh?'},
 {'role': 'assistant',
  'content': 'null',
  'function_call': {'name': 'get_current_weather',
   'arguments': '{\n  "location": "Edinburgh"\n}'}}]
```

In [39]:
```
function_name = messages[-1]["function_call"]["name"]
function_args = json.loads(messages[-1]["function_call"]["arguments"])
```

In [40]:
```
print("Function Name:", function_name)
print("Function Arguments:", function_args)
```

```
Function Name: get_current_weather
Function Arguments: {'location': 'Edinburgh'}
```

Let's now call our actual python function to get the response!

In [41]:
```
function_response = eval(function_name)(**function_args)

print(function_response)
```

```
{'temperature': 9, 'unit': 'celsius', 'description': 'Sunny'}
```

Note:

This: `eval(function_name)(**function_args)`

is equivalent to this: `get_current_weather(location='Edinburgh')`

> **Tip:** There is actually only more role that can be used with ONLY gpt-3.5-turbo-0613 and gpt-4-0613.
>
> - **Function role:** Allows you to specify the response of a function that the model can use to answer the original question.

In [42]:
```
new_message = {
    "role": "function",
    "name": function_name,
    "content": str(function_response),
}
```

Let's finally get the response from our model:

In [44]:
```
messages = get_completion_from_messages_func(new_message, messages, tools)
```

```
{'role': 'assistant', 'content': 'The current weather in Edinburgh is sunny w
ith a temperature of 9 degrees Celsius.'}
```