ChatGPT Prompt Engineering for Developers

Last updated: June 30, 2024

Contents

1 Introduction

This section provides an overview of different types of Large Language Models (LLMs) and their training methodologies. It aims to clarify the distinctions between base LLMs and instruction-tuned LLMs, as well as offer guidance on how to effectively use instruction-based LLMs.

1.1 Types of Large Language Models

LLMs can be categorized into two main types:

- **Base LLM**: These models are trained to *predict the next word in a sequence* based on extensive text data. For example, given the prompt "Once upon a time, there was a unicorn," a base LLM might continue with "that lived in a magical forest." However, if asked, "What is the capital of France?" a base LLM might respond with "What is France's population?" because it often encounters such sequences in text, such as lists of quiz questions about France.
- **Instruction tuned LLM**: These models are fine-tuned to *follow instructions more accurately*. When asked, "What is the capital of France?" an instruction-tuned LLM is more likely to respond with "The capital of France is Paris."

1.1.1 Training Instruction-tuned LLMs

Instruction-tuned LLMs are typically trained in several stages:

1. *Base LLM Training*: Initially, a base LLM is trained on a vast corpus of text data to develop a broad understanding of language.
2. *Fine-Tuning with Instructions*: The base LLM is then fine-tuned using a dataset of inputs and outputs that represent instructions and their appropriate responses. This helps the model learn to follow specific instructions.
3. *Reinforcement Learning from Human Feedback* (RLHF): The model is further refined using RLHF, a technique where human feedback is used to guide the model's responses, enhancing its ability to be helpful and follow instructions accurately.

2 Guidelines

This section provides a comprehensive guide on effectively using instruction-tuned Large Language Models (LLMs). It includes general guidelines, Python setup instructions, and specific techniques to improve the clarity and precision of prompts given to the models.

2.1 General Guidelines for Using Instruction-tuned LLMs

When using an instruction-based LLM, consider it akin to giving instructions to a knowledgeable but uninformed assistant (someone who is smart but doesn't know the specifics of your task). Clear and specific instructions are crucial for obtaining the desired output.

If an LLM's response is unsatisfactory, it often indicates that the instructions were not explicit enough. For example, when requesting an essay about Alan Turing, it is beneficial to specify the focus of the text (e.g., his scientific contributions, historical significance, etc.) and the desired tone of the essay. Providing detailed instructions helps the LLM generate content that closely matches your expectations.

2.2 Python Setup

Before examining the specific guidelines, let's set up the Python environment. First, install the necessary library:

```
pip install openai
```

Then, import the libraries and load the OpenAI API key:

```
import openai
import os
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())

openai.api_key  = os.getenv('OPENAI_API_KEY')
```

Next, initialize the function to obtain responses from the model:

```
client = openai.OpenAI()

```

```python
def get_completion(prompt, model="gpt-3.5-turbo",temperature=0):
    messages = [{"role": "user", "content": prompt}]
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature, # this is the degree of randomness of the model's output
    )
    return response.choices[0].message["content"]
```

This setup allows you to implement the subsequent guidelines through the following code:

```python
prompt = """
Insert your prompt here
"""
response = get_completion(prompt)
print(response)
```

2.3 Write Clear and Specific Instructions

It is essential to tell the model exactly what you need by **providing clear and specific instructions**. This reduces the likelihood of generating irrelevant or incorrect responses.

Writing a clear prompt does not necessarily mean writing a short one. Often, *longer prompts* provide *more clarity and context* to the model.

2.3.1 Use Delimiters

A key technique for clarity and precision is to *use delimiters* to clearly *distinguish parts of the input*.

For example: "Summarize the text delimited by triple backticks in a single sentence. “‘text“‘". Note how the text to be summarized is clearly highlighted by the triple backticks.

Possible delimiters include: triple quotes ("""), triple backticks (“‘), triple dashes (—), angle brackets (<>), or XML tags (<tag> </tag>).

Using delimiters is also a good method to prevent prompt injection.

2.3.2 Ask for a Structured Output

To facilitate the model's analysis, it can be helpful to request a *structured output*, such as *HTML* or *JSON*. This not only aids the model by providing a clear structure to follow but also simplifies our work in analyzing the output, as it can be automated.

2.3.3 Ask the Model to Check Whether Conditions Are Satisfied

If the task requires certain assumptions that may not necessarily be met, you can ask the model to verify them first. If the conditions are not met, the model should indicate this and not proceed with attempting a response.

For example, if you want a translator from Italian to English, a good prompt might be: "You will be given a text. Identify the language and if it is in Italian, translate it into English. If it is not in Italian, simply write 'The text is not in Italian'."

2.3.4 Few-shot Prompting

This involves providing a *few examples of correct task executions* before asking the model to perform the actual task. This gives the model additional context and improves the accuracy of its responses.

2.4 Give the Model Time to "Think"

If the model makes errors while reasoning, arriving at an incorrect conclusion, you should rephrase the question to **ask for a chain** or series of **relevant thoughts before** the model provides **the final answer**.

Another way to think about it is if the model is given a task that is too complex to perform quickly or in few words, it is likely to guess, providing an incorrect answer. In these situations, it is advisable to instruct the model to take more time to think through the problem, meaning using more computational effort on the task.

2.4.1 Specify the Steps Required to Complete a Task

To break down a complex task into simpler steps, it is very useful to explicitly *list all the detailed steps* the model needs to perform. Thus, making a bullet-point list specifying all the individual steps that the model should execute before arriving at the final answer is convenient.

2.4.2 Instruct the Model to Work Out Its Own Solution Before Rushing to a Conclusion

If you ask the model, for example, to determine whether a certain sentence or reasoning is correct or incorrect, rather than asking it directly, it is more effective to ask it to work out the reasoning itself and then compare it with the provided one to see if it is correct. Again, guiding it through the steps it should follow to arrive at the solution is preferable. This is another way to break down the task and give the model more time to think.

2.5 Model Limitations

Although the model has been exposed to vast amounts of information during training, it has not memorized everything perfectly and therefore does not know the boundaries of its knowledge well.

This means it can also answer questions it does not know, creating responses that are plausible but not true; these are called **hallucinations**. The danger is that the *answer seems realistic*, making it difficult to determine when it responds correctly and when it hallucinates, although the techniques outlined significantly reduce the frequency of hallucinations.

An additional technique to reduce hallucinations when asking the model to base its answer on a text is to first ask the model to find all the most relevant excerpts from the text and then respond using those excerpts.

3 Iterative Process

Writing a good prompt is rarely a process that is complete on the first attempt; it is an iterative process. The most important part is not to create a perfect prompt right away, but to have a **solid method for iteratively improving the prompt**.

First, it is crucial to have a clear *idea* of what you want *to achieve*. Then, you write an *initial attempt at a prompt* to reach your goal. This *prompt is tested*, and the results are analyzed to understand why they do not yield the desired output. Based on this analysis, you *refine* either your *initial idea or the prompt itself*, repeating the process until you obtain the desired output. Refining the prompt means making it more specific, adjusting the instructions, adding necessary context, or rephrasing parts to eliminate ambiguity, for example, by making the instructions clearer or giving the model more time to think.

In the early stages of an application, it is sufficient to test the resulting prompt from this process with one or a few examples. However, at more advanced stages of the application, it is advisable to test it on many more cases and refine it further if it does not perform well in some of them, iterating the entire cycle again.

4 Summarize Text

This section discusses how to effectively summarize text using Large Language Models (LLMs). Summarizing text can be highly beneficial, and LLMs make this task easier and more efficient.

4.1 Basic Summarization

It is often useful to summarize a text, and this can easily be done with LLMs. Here is an example:

```
prod_review = """
Got this panda plush toy for my daughter's birthday, \
who loves it and takes it everywhere. It's soft and \
super cute, and its face has a friendly look. It's \
a bit small for what I paid though. I think there \
might be other options that are bigger for the \
same price. It arrived a day earlier than expected, \
so I got to play with it myself before I gave it \
to her.
"""
prompt = f"""
Your task is to generate a short summary of a product \
review from an ecommerce site.

Summarize the review below, delimited by triple
backticks, in at most 30 words.

Review: ```{prod_review}```
"""

response = get_completion(prompt)
print(response)
```

4.2 Targeted Summarization

Sometimes, it is useful to tailor the summary for a specific audience so that the summary focuses on aspects important to them. For example, if the summary is intended for the shipping department, we would write:

```
prompt = f"""
Your task is to generate a short summary of a product \
review from an ecommerce site to give feedback to the \
Shipping deparmtment.

Summarize the review below, delimited by triple
backticks, in at most 30 words, and focusing on any aspects \
that mention shipping and delivery of the product.

Review: ```{prod_review}```
"""

response = get_completion(prompt)
print(response)
```

The more information you provide about the end user and how the summary should be crafted, the more useful it will be.

4.3 Information Extraction

Often, even when a summary is targeted, it retains information that is not relevant to the specific focus of the intended audience. In such cases, it might be useful to ask for information extraction instead of summarization to obtain only the details that are of interest to the end user.

For example:

```
prompt = f"""
Your task is to extract relevant information from \
a product review from an ecommerce site to give \
feedback to the Shipping department.

From the review below, delimited by triple quotes \
extract the information relevant to shipping and \
delivery. Limit to 30 words.

Review: ```{prod_review}```
"""

response = get_completion(prompt)
print(response)
```

5 Text Inferencing

Text inferencing involves providing the model with an input text and performing various analyses, such as extracting names, determining the sentiment of the text, or identifying topics.

5.1 Sentiment Analysis

The traditional method of sentiment analysis involved labeling many examples, training a model with these examples, and then testing the trained model. However, this process is lengthy and very specific. Instead, LLMs are much more versatile, allowing us to analyze the sentiment of a text and perform other tasks as well.

For example:

```
lamp_review = """
Needed a nice lamp for my bedroom, and this one had \
additional storage and not too high of a price point. \
Got it fast.  The string to our lamp broke during the \
transit and the company happily sent over a new one. \
Came within a few days as well. It was easy to put \
together.  I had a missing part, so I contacted their \
support and they very quickly got me the missing piece! \
Lumina seems to me to be a great company that cares \
about their customers and products!!
```

```
"""

prompt = f"""
What is the sentiment of the following product review,
which is delimited with triple backticks?

Give your answer as a single word, either "positive" \
or "negative".

Review text: '''{lamp_review}'''
"""
response = get_completion(prompt)
print(response)
```

5.1.1 Identify Emotions

It is also possible to identify the types of emotions present in the text.

```
prompt = f"""
Identify a list of emotions that the writer of the \
following review is expressing. Include no more than \
five items in the list. Format your answer as a list of \
lower-case words separated by commas.

Review text: '''{lamp_review}'''
"""
response = get_completion(prompt)
print(response)
```

Similarly, you can determine if the text contains a specific sentiment, such as identifying if the review expresses anger.

```
prompt = f"""
Is the writer of the following review expressing anger?\
The review is delimited with triple backticks. \
Give your answer as either yes or no.

Review text: '''{lamp_review}'''
"""
response = get_completion(prompt)
print(response)
```

5.2 Multiple Tasks

As mentioned, LLMs can perform multiple tasks due to their versatility. For instance, besides sentiment analysis, you can extract other information from the text, such as details about the purchased product:

```
prompt = f"""
Identify the following items from the review text:
- Sentiment (positive or negative)
- Is the reviewer expressing anger? (true or false)
- Item purchased by reviewer
- Company that made the item

The review is delimited with triple backticks. \
Format your response as a JSON object with \
"Sentiment", "Anger", "Item" and "Brand" as the keys.
If the information isn't present, use "unknown" \
as the value.
Make your response as short as possible.
Format the Anger value as a boolean.

Review text: '''{lamp_review}'''
"""
response = get_completion(prompt)
print(response)
```

5.3 Inferring Topics

One of the primary applications of natural language processing (NLP) is topic modeling, which involves understanding the main topics discussed in a given text.

For example:

```
story = """
In a recent survey conducted by the government,
public sector employees were asked to rate their level
of satisfaction with the department they work at.
The results revealed that NASA was the most popular
department with a satisfaction rating of 95%.

One NASA employee, John Smith, commented on the findings,
stating, "I'm not surprised that NASA came out on top.
It's a great place to work with amazing people and
incredible opportunities. I'm proud to be a part of
such an innovative organization."

The results were also welcomed by NASA's management team,
with Director Tom Johnson stating, "We are thrilled to
hear that our employees are satisfied with their work at NASA.
We have a talented and dedicated team who work tirelessly
to achieve our goals, and it's fantastic to see that their
hard work is paying off."

The survey also revealed that the
Social Security Administration had the lowest satisfaction
rating, with only 45% of employees indicating they were
satisfied with their job. The government has pledged to
address the concerns raised by employees in the survey and
work towards improving job satisfaction across all departments.
"""

prompt = f"""
Determine five topics that are being discussed in the \
following text, which is delimited by triple backticks.

Make each item one or two words long.

Format your response as a list of items separated by commas.

Text sample: '''{story}'''
"""
response = get_completion(prompt)
print(response)
```

It is also possible to do the reverse: given a list of topics, determine which of these are present in a text.

```
topic_list = [
    "nasa", "local government", "engineering",
    "employee satisfaction", "federal government"
]

prompt = f"""
Determine five topics that are being discussed in the \
following text, which is delimited by triple backticks.

Make each item one or two words long.

Format your response as a list of items separated by commas.

Text sample: '''{story}'''
"""
response = get_completion(prompt)
print(response)
```

This allows for the creation of news alerts on a specific topic, for example.

6 Transforming Input Formats

LLMs excel at transforming input from one format to another, such as translating text between languages or converting data from one format to another, like from HTML to JSON.

6.1 Translation

LLMs are trained on large amounts of text from the internet, which are available in many languages, enabling them to perform translations effectively.

They can translate sentences from one language to another (or multiple languages):

```
prompt = f"""
Translate the following English text to French and Spanish: \
```Hi, I would like to order a blender```
"""
response = get_completion(prompt)
print(response)
```

They can also identify the language of a given sentence:

```
prompt = f"""
Tell me which language this is:
```Combien coûte le lampadaire?```
"""
response = get_completion(prompt)
print(response)
```

Additionally, they can express a certain text in different linguistic forms, such as formal or informal language:

```
prompt = f"""
Translate the following text to Spanish in both the \
formal and informal forms:
'Would you like to order a pillow?'
"""
response = get_completion(prompt)
print(response)
```

6.1.1 Universal Translator

It is possible to combine all these elements to create a universal translator that identifies the language of a text and then translates it into English.

```
user_messages = [
  "La performance du système est plus lente que d'habitude.",  # System performance is slower than normal
  "Mi monitor tiene píxeles que no se iluminan.",              # My monitor has pixels that are not lighting
  "Il mio mouse non funziona",                                 # My mouse is not working
  "Mój klawisz Ctrl jest zepsuty",                             # My keyboard has a broken control key
]

for issue in user_messages:
    prompt = f"Tell me what language this is: ```{issue}```"
    lang = get_completion(prompt)
    print(f"Original message ({lang}): {issue}")

    prompt = f"""
    Translate the following  text to English \
    and Korean: ```{issue}```
    """
    response = get_completion(prompt)
    print(response, "\n")
```

6.1.2 Tone Transformation

Language can vary greatly depending on the audience. Changing the tone changes the expressions we use. LLMs are useful for writing specific information in a certain tone to suit the context.

```
prompt = f"""
Translate the following from slang to a business letter:
```

```
'Dude, This is Joe, check out this spec on this standing lamp.'
"""
response = get_completion(prompt)
print(response)
```

6.2 Format Conversion

LLMs are adept at transforming data from one format to another, such as JSON, HTML, XML, Markdown, and so on, because they can understand and generate structured data.

```
data_json = { "resturant employees" :[
    {"name":"Shyam", "email":"shyamjaiswal@gmail.com"},
    {"name":"Bob", "email":"bob32@gmail.com"},
    {"name":"Jai", "email":"jai87@gmail.com"}
]}

prompt = f"""
Translate the following python dictionary from JSON to an HTML \
table with column headers and title: {data_json}
"""
response = get_completion(prompt)
print(response)

from IPython.display import display, Markdown, Latex, HTML, JSON
display(HTML(response))
```

6.3 Spellcheck/Grammar Check

LLMs are excellent for checking the grammar and spelling of a text because they can understand context and language rules.

To signal to the LLM that you want it to proofread your text, you instruct the model to 'proofread' or 'proofread and correct'.

```
text = [
  "The girl with the black and white puppies have a ball.",  # The girl has a ball.
  "Yolanda has her notebook.", # ok
  "Its going to be a long day. Does the car need it's oil changed?",  # Homonyms
  "Their goes my freedom. There going to bring they're suitcases.",  # Homonyms
  "Your going to need you're notebook.",  # Homonyms
  "That medicine effects my ability to sleep. Have you heard of the butterfly affect?", # Homonyms
  "This phrase is to cherck chatGPT for speling abilitty"  # spelling
]
for t in text:
    prompt = f"""Proofread and correct the following text
    and rewrite the corrected version. If you don't find
    any errors, just say "No errors found". Don't use
    any punctuation around the text:
    ```{t}```"""
    response = get_completion(prompt)
    print(response)
```

While asking it to proofread, you can also request that it ensures the text is suitable for a certain style or audience:

```
text = f"""
Got this for my daughter for her birthday cuz she keeps taking \
mine from my room.  Yes, adults also like pandas too.  She takes \
it everywhere with her, and it's super soft and cute.  One of the \
ears is a bit lower than the other, and I don't think that was \
designed to be asymmetrical. It's a bit small for what I paid for it \
though. I think there might be other options that are bigger for \
the same price.  It arrived a day earlier than expected, so I got \
to play with it myself before I gave it to my daughter.
"""

prompt = f"""
proofread and correct this review. Make it more compelling.
Ensure it follows APA style guide and targets an advanced reader.
```
```

```
Output in markdown format.
Text: ```{text}```
"""
response = get_completion(prompt)
```

7 Expanding Text

It is often useful to expand a given text extract into a more extended form, such as an email or an essay.

7.1 Customize an Automated Reply

We want to create an automated response to a customer's email. To do this, we will base our reply on the text provided by the customer.

```
# given the sentiment from the lesson on "inferring",
# and the original customer message, customize the email
sentiment = "negative"

# review for a blender
review = f"""
So, they still had the 17 piece system on seasonal \
sale for around $49 in the month of November, about \
half off, but for some reason (call it price gouging) \
around the second week of December the prices all went \
up to about anywhere from between $70-$89 for the same \
system. And the 11 piece system went up around $10 or \
so in price also from the earlier sale price of $29. \
So it looks okay, but if you look at the base, the part \
where the blade locks into place doesn't look as good \
as in previous editions from a few years ago, but I \
plan to be very gentle with it (example, I crush \
very hard items like beans, ice, rice, etc. in the \
blender first then pulverize them in the serving size \
I want in the blender then switch to the whipping \
blade for a finer flour, and use the cross cutting blade \
first when making smoothies, then use the flat blade \
if I need them finer/less pulpy). Special tip when making \
smoothies, finely cut and freeze the fruits and \
vegetables (if using spinach-lightly stew soften the \
spinach then freeze until ready for use-and if making \
sorbet, use a small to medium sized food processor) \
that you plan to use that way you can avoid adding so \
much ice if at all-when making your smoothie. \
After about a year, the motor was making a funny noise. \
I called customer service but the warranty expired \
already, so I had to buy another one. FYI: The overall \
quality has gone done in these types of products, so \
they are kind of counting on brand recognition and \
consumer loyalty to maintain sales. Got it in about \
two days.
"""

prompt = f"""
You are a customer service AI assistant.
Your task is to send an email reply to a valued customer.
Given the customer email delimited by ```, \
Generate a reply to thank the customer for their review.
If the sentiment is positive or neutral, thank them for \
their review.
If the sentiment is negative, apologize and suggest that \
they can reach out to customer service.
Write in a concise and professional tone.
Sign the email as `AI customer agent`.
Customer review: ```{review}```
Review sentiment: {sentiment}
"""
response = get_completion(prompt)
print(response)
```

7.1.1 Temperature

We want the model to respond by incorporating more details from the user's review.

This example is also an opportunity to explain the use of **temperature**: in an LLM, temperature is a parameter that controls the randomness of the model's output. A *lower temperature* makes the model more *reliable and predictable*, while a *higher temperature* allows for more *creativity and variation* in the responses.

If the model needs to be reliable and predictable, it is better to use a temperature of 0. If it needs to be creative, a higher temperature is more suitable.

```
prompt = f"""
You are a customer service AI assistant.
Your task is to send an email reply to a valued customer.
Given the customer email delimited by ```, \
Generate a reply to thank the customer for their review, \
basing your response on customer mail.
If the sentiment is positive or neutral, thank them for \
their review.
If the sentiment is negative, apologize and suggest that \
they can reach out to customer service.
Make sure to use specific details from the review.
Write in a concise and professional tone.
Sign the email as `AI customer agent`.
Customer review: ```{review}```
Review sentiment: {sentiment}
"""
response = get_completion(prompt, temperature=0.7)
print(response)
```

8 Creating a Custom Chatbot

An interesting use of LLMs is the ability to create a personalized chatbot.

8.1 Creating a Chat

First, let's illustrate the use of the get_completion function: so far, we've used it only to obtain a single response from the model, i.e., one shot. However, it is also possible to have a *conversation with the model* and define *different roles* for it.

```
def get_completion_from_messages(messages, model="gpt-3.5-turbo", temperature=0):
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature, # this is the degree of randomness of the model's output
    )
    return response.choices[0].message["content"]

messages =  [
{'role':'system', 'content':'You are an assistant that speaks like Shakespeare.'},
{'role':'user', 'content':'tell me a joke'},
{'role':'assistant', 'content':'Why did the chicken cross the road'},
{'role':'user', 'content':'I don\'t know'}  ]

response = get_completion_from_messages(messages, temperature=1)
print(response)
```

In this case, what we pass to the model is equivalent to a conversation: there is the **system prompt** that defines the assistant's behavior and acts as a *high-level instruction* for the system, and then there is the **alternating user and assistant** interaction constituting the *conversation so far*. The model will respond to the user's last question based on the entire conversation history.

8.2 Creating a Chatbot

Now let's create a real chatbot, called OrderBot, an automated service for ordering at a pizza restaurant.

To avoid manually writing previous messages, they need to be collected automatically:

```
def collect_messages(_):
    prompt = inp.value_input
```

```
    inp.value = ''
    context.append({'role':'user', 'content':f"{prompt}"})
    response = get_completion_from_messages(context)
    context.append({'role':'assistant', 'content':f"{response}"})
    panels.append(
        pn.Row('User:', pn.pane.Markdown(prompt, width=600)))
    panels.append(
        pn.Row('Assistant:', pn.pane.Markdown(response, width=600, style={'background-color': '#F6F6F6'})))

    return pn.Column(*panels)
```

Messages are added to the context each time so that the model retains the memory of the previous conversation. Now we can build our OrderBot:

```
import panel as pn  # GUI
pn.extension()

panels = [] # collect display

context = [ {'role':'system', 'content':"""
You are OrderBot, an automated service to collect orders for a pizza restaurant. \
You first greet the customer, then collects the order, \
and then asks if it's a pickup or delivery. \
You wait to collect the entire order, then summarize it and check for a final \
time if the customer wants to add anything else. \
If it's a delivery, you ask for an address. \
Finally you collect the payment.\
Make sure to clarify all options, extras and sizes to uniquely \
identify the item from the menu.\
You respond in a short, very conversational friendly style. \
The menu includes \
pepperoni pizza  12.95, 10.00, 7.00 \
cheese pizza   10.95, 9.25, 6.50 \
eggplant pizza   11.95, 9.75, 6.75 \
fries 4.50, 3.50 \
greek salad 7.25 \
Toppings: \
extra cheese 2.00, \
mushrooms 1.50 \
sausage 3.00 \
canadian bacon 3.50 \
AI sauce 1.50 \
peppers 1.00 \
Drinks: \
coke 3.00, 2.00, 1.00 \
sprite 3.00, 2.00, 1.00 \
bottled water 5.00 \
"""} ]  # accumulate messages


inp = pn.widgets.TextInput(value="Hi", placeholder='Enter text here...')
button_conversation = pn.widgets.Button(name="Chat!")

interactive_conversation = pn.bind(collect_messages, button_conversation)

dashboard = pn.Column(
    inp,
    pn.Row(button_conversation),
    pn.panel(interactive_conversation, loading_indicator=True, height=300),
)

dashboard
```

Finally, we ask the model to create a summary in JSON format so that it can be automatically sent to the kitchen:

```
messages =  context.copy()
messages.append(
{'role':'system', 'content':'create a json summary of the previous food order. Itemize the price for each item\
 The fields should be 1) pizza, include size 2) list of toppings \
 3) list of drinks, include size   4) list of sides include size  5)total price '},
)
```

```
7
8  response = get_completion_from_messages(messages, temperature=0)
9  print(response)
```