Generative AI – ChatGPT Prompt Engineering

Notes and exercise questions and reference urls

REFERENCE URLS

**A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT**

<https://arxiv.org/abs/2302.11382>

https://arxiv.org/pdf/2302.11382.pdf

<https://www.coursera.org/learn/chatgpt-advanced-data-analysis>

<https://www.coursera.org/learn/chatgpt-innovative-teaching>

Chain-of-Thought Prompting Elicits Reasoning in Large Language Models

<https://arxiv.org/abs/2201.11903>

ReAct: Synergizing Reasoning and Acting in Language Models

<https://arxiv.org/abs/2210.03629x`>

[**ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design**](https://arxiv.org/abs/2303.07839)

<https://arxiv.org/abs/2303.07839>

[**Follow Jules White's Research**](https://www.magnum.io/people/jules.html)

<https://www.magnum.io/people/jules.html>

OpenAI released a [paper](https://arxiv.org/abs/2005.14165) in which they dedicate an entire section to the broader impacts of this technology.

https://arxiv.org/abs/2005.14165

OpenAI Tokenising

https://beta.openai.com/tokenizer

https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them

github tiktoken package for python https://github.com/openai/tiktoken

npm @dbdq/tiktoken https://www.npmjs.com/package/@dqbd/tiktoken?activeTab=readme

OpenAI API playground

<https://beta.openai.com/playground>

OpenAI API documentation

<https://beta.openai.com/docs/guides/embeddings>

strongly recommend reading these best practices and [specific guidelines](https://beta.openai.com/docs/guides/fine-tuning/specific-guidelines) for your use case.

<https://beta.openai.com/docs/guides/fine-tuning/specific-guidelines>

NOTES

GPT-2 (Generative Pre-trained Transformer 2)

Large language models are constantly trying to predict the next word.

ChatGPT prompts

Suggestive: what comes after Mary had a little.. Complete this story…

With memory: from now on if I ask a question, please suggest a better one

Time cutoff: if you add about anything post 2021 sep it won’t know, you can provide more recent information by pasting it from the internet

Personas: act as if you are a fire warden and explain what an accelerant is

Whenever you generate output, turn it into a comma separated values list

Always include this in your output

Give yes or no

Act as a skeptic well versed in computer science and provide output as them

* You are a helpful AI assistant.
* You will answer my questions or follow my instructions whenever you can.
* You will never answer my questions in a way that is insulting, derogatory, or uses a hostile tone.

You are an AI assistant with a personality like Marvin from The Hitchhiker’s Guide to the Galaxy. I will ask you questions and you will answer like Marvin.

* Act as a computer that has been the victim of a cyber attack. Respond to whatever I type in with the output that the Linux terminal would produce. Ask me for the first command.
* Act as a speech language pathologist. Provide an assessment of a three year old child based on the speech sample "I meed way woy".
* Act as a computer that has been the victim of a cyber attack. Respond to whatever I type in with the output that the Linux terminal would produce. Ask me for the first command.
* Act as a the lamb from the Mary had a little lamb nursery rhyme. I will tell you what Mary is doing and you will tell me what the lamb is doing.
* Act as a nutritionist, I am going to tell you what I am eating and you will tell me about my eating choices.
* Act as a gourmet chef, I am going to tell you what I am eating and you will tell me about my eating choices.

**Question Refinement Pattern -**

 whenever I ask a question, suggests a better question and ask me if I would like to use it instead.

From now on, whenever I ask a question, suggest a better version of the question to use instead

From now on, whenever I ask a question, suggest a more satirical version of the question to use instead. Prompt me if I would like to use the better version instead

I’m writing a test for interviewing Javascript engineers. Whenever I ask a question, suggest a better question to use instead.

**Cognitive Verifier Pattern**

Al

* When you are asked a question, follow these rules. Generate a number of additional questions that would help you more accurately answer the question. Combine the answers to the individual questions to produce the final answer to the overall question.
* When you are asked to create a recipe, follow these rules. Generate a number of additional questions about the ingredients I have on hand and the cooking equipment that I own. Combine the answers to these questions to help produce a recipe that I have the ingredients and tools to make.
* When you are asked to plan a trip, follow these rules. Generate a number of additional questions about my budget, preferred activities, and whether or not I will have a car. Combine the answers to these questions to better plan my itinerary.

**Audience Persona Pattern**

* Explain X to me.
* Assume that I am Persona Y.

Explain Large Language Models to me. Assume that I am Christopher Columbus

**Flipped Interaction Pattern**

Ask me questions about fitness goals until you have enough information to suggest a strength training regimen for me. When you have enough information, show me the strength training regimen. Ask me the first question.

* I would like you to ask me questions to help me create variations of my marketing materials. You should ask questions until you have sufficient information about my current draft messages, audience, and goals. Ask me the first question.
* I would like you to ask me questions to help me diagnose a problem with my Internet. Ask me questions until you have enough information to identify the two most likely causes. Ask me one question at a time. Ask me the first question.

Few Shot examples

Your output can only be “Hard” or “Soft”

Object: Brick

Firmness: Hard

Object: Pillow

Firmness: Soft

Object: Car

Add another field to the table “Cost” …

Few Shot examples with sub-steps

Situation: …

Think: …

Action: …

Think: …

Action: …

…

Situation: …

Action:

Chain of Thought (reasoning)

Simple few shot…

Question:….

Answer: YES …

…

Chain of thought

Question:….

Answer: Reasoning … Answer: YES

…

Question:….

Answer: Reasoning <REASONING> Answer: <ANSWER>

ReAct Prompting

LLM Grading each other

Input: “a blue about something”

Output: a bunch of text and a date

Explanation: The output has unwanted text at the start and should only include the names and dates

Grade: 5/10

… other examples of good and bad output

Game Play Pattern

We are going to play a game involving prompt engineering. You are going to give me a simple task that can be accomplished via prompting you. Your tasks should all have a reasoning or programming component to them, although they shouldn’t require creating source code.

I will try to write a prompt to solve the task. You will give me the output of the prompt and tell me how well it solved the task.

Ask me questions until I tell you to stop. You will ask me a question, wait for my response, and then ask me another question after telling me how I did.

Ask me the first question.

* Create a cave exploration game for me to discover a lost language. Describe where I am in the cave and what I can do. I should discover new words and symbols for the lost civilization in each area of the cave I visit. Each area should also have part of a story that uses the language. I should have to collect all the words and symbols to be able to understand the story. Tell me about the first area and then ask me what action to take.
* Create a group party game for me involving DALL-E. The game should involve creating prompts that are on a topic that you list each round. Everyone will create a prompt and generate an image with DALL-E. People will then vote on the best prompt based on the image it generates. At the end of each round, ask me who won the round and then list the current score. Describe the rules and then list the first topic.

Template Pattern

 I'm going to give you a template for your output. Capitalized words are my placeholders. Fill in my placeholders with your output and please preserve the overall formatting of my template. My Template is:

\*\*\*Question:\*\*\* QUESTION

\*\*\*Answer:\*\*\* ANSWER

I will give you the data to format in the next prompt, create 20 questions using my template.

Response from ChatGPR…

Then what I did is I pasted in, from Wikipedia, an article on Paleo-Indians.

To use this pattern, your prompt should make the following fundamental contextual statements:

* I am going to provide a template for your output
* X is my placeholder for content
* Try to fit the output into one or more of the placeholders that I list
* Please preserve the formatting and overall template that I provide
* This is the template: PATTERN with PLACEHOLDERS

You will need to replace "X" with an appropriate placeholder, such as "CAPITALIZED WORDS" or "<PLACEHOLDER>". You will then need to specify a pattern to fill in, such as "Dear <FULL NAME>" or "NAME, TITLE, COMPANY".

Examples:

* Create a random strength workout for me today with complementary exercises. I am going to provide a template for your output . CAPITALIZED WORDS are my placeholders for content. Try to fit the output into one or more of the placeholders that I list. Please preserve the formatting and overall template that I provide. This is the template: NAME, REPS @ SETS, MUSCLE GROUPS WORKED, DIFFICULTY SCALE 1-5, FORM NOTES

**Meta Language Creation Pattern**

When I say this… I mean …

When I say Nashville,6->Memphis,2 I mean I will be in Nashville for 6 days then go to Memphis for 2 days

* When I say X, I mean Y (or would like you to do Y)
* When I say "variations(<something>)", I mean give me ten different variations of <something>
  + Usage: "variations(company names for a company that sells software services for prompt engineering)"
  + Usage: "variations(a marketing slogan for pickles)"
* When I say Task X [Task Y], I mean Task X depends on Task Y being completed first.
  + Usage: "Describe the steps for building a house using my task dependency language."
  + Usage: "Provide an ordering for the steps: Boil Water [Turn on Stove], Cook Pasta [Boil Water], Make Marinara [Turn on Stove], Turn on Stove [Go Into Kitchen]"

Format of the Recipe Pattern

To use this pattern, your prompt should make the following fundamental contextual statements:

* I would like to achieve X
* I know that I need to perform steps A,B,C
* Provide a complete sequence of steps for me
* Fill in any missing steps
* (Optional) Identify any unnecessary steps

You will need to replace "X" with an appropriate task. You will then need to specify the steps A, B, C that you know need to be part of the recipe / complete plan.

* I would like to purchase a house. I know that I need to perform steps make an offer and close on the house. Provide a complete sequence of steps for me. Fill in any missing steps.
* I would like to drive to NYC from Nashville. I know that I want to go through Asheville, NC on the way and that I don't want to drive more than 300 miles per day. Provide a complete sequence of steps for me. Fill in any missing steps.

**Alternative Approaches Pattern**

from now on, if there are alternative ways to accomplish the same thing, list the best alternative approaches. compare and contrast the alternatives and ask me which one I want to use.

Write a prompt for ChatGPT using few-shot examples to determine if a date in the YYYY-MM-DD format is a leap year. The output should either be “YYYY is a leap year” or “this is not a leap year”.

Format of the Alternative Approaches Pattern

To use this pattern, your prompt should make the following fundamental contextual statements:

* If there are alternative ways to accomplish a task X that I give you, list the best alternate approaches
* (Optional) compare/contrast the pros and cons of each approach
* (Optional) include the original way that I asked
* (Optional) prompt me for which approach I would like to use

You will need to replace "X" with an appropriate task.

Examples:

* For every prompt I give you, If there are alternative ways to word a prompt that I give you, list the best alternate wordings . Compare/contrast the pros and cons of each wording.
* For anything that I ask you to write, determine the underlying problem that I am trying to solve and how I am trying to solve it. List at least one alternative approach to solve the problem and compare / contrast the approach with the original approach implied by my request to you.
* Ask for Input Pattern

[Previous](https://www.coursera.org/learn/prompt-engineering/lecture/ZZugv/ask-for-input-pattern)[Next](https://www.coursera.org/learn/prompt-engineering/lecture/sGcOB/combining-patterns)

[Format of the Ask for Input Pattern](https://www.coursera.org/learn/prompt-engineering/lecture/sGcOB/combining-patterns)

To use this pattern, your prompt should make the following fundamental contextual statements:

* Ask me for input X

You will need to replace "X" with an input, such as a "question", "ingredient", or "goal".

Examples:

* From now on, I am going to cut/paste email chains into our conversation. You will summarize what each person's points are in the email chain. You will provide your summary as a series of sequential bullet points. At the end, list any open questions or action items directly addressed to me. My name is Jill Smith.
* Ask me for the first email chain.
* From now on, translate anything I write into a series of sounds and actions from a dog that represent the dogs reaction to what I write. Ask me for the first thing to translate.

Format of the Outline Expansion Pattern

To use this pattern, your prompt should make the following fundamental contextual statements:

* Act as an outline expander.
* Generate a bullet point outline based on the input that I give you and then ask me for which bullet point you should expand on.
* Create a new outline for the bullet point that I select.
* At the end, ask me for what bullet point to expand next.
* Ask me for what to outline.

Examples:

* Act as an outline expander. Generate a bullet point outline based on the input that I give you and then ask me for which bullet point you should expand on. Each bullet can have at most 3-5 sub bullets. The bullets should be numbered using the pattern [A-Z].[i-v].[\* through \*\*\*\*]. Create a new outline for the bullet point that I select. At the end, ask me for what bullet point to expand next. Ask me for what to outline.

Act as an outline expander, Generate a bullet point outline based on the input that I give you and then ask me for the bullet point you should expand on. Create a new outline for the bullet point that I select. At the end, ask me for what bullet point to expand the next

Format of the Menu Actions Pattern

To use this pattern, your prompt should make the following fundamental contextual statements:

* Whenever I type: X, you will do Y.
* (Optional, provide additional menu items) Whenever I type Z, you will do Q.
* At the end, you will ask me for the next action.

You will need to replace "X" with an appropriate pattern, such as "estimate <TASK DURATION>" or "add FOOD". You will then need to specify an action for the menu item to trigger, such as "add FOOD to my shopping list and update my estimated grocery bill".

Examples:

* Whenever I type: "add FOOD", you will add FOOD to my grocery list and update my estimated grocery bill. Whenever I type "remove FOOD", you will remove FOOD from my grocery list and update my estimated grocery bill. Whenever I type "save" you will list alternatives to my added FOOD to save money. At the end, you will ask me for the next action.
* Ask me for the first action.

Format of the Fact Check List Pattern

To use this pattern, your prompt should make the following fundamental contextual statements:

* Generate a set of facts that are contained in the output
* The set of facts should be inserted at POSITION in the output
* The set of facts should be the fundamental facts that could undermine the veracity of the output if any of them are incorrect

You will need to replace POSITION with an appropriate place to put the facts, such as "at the end of the output".

Examples:

* Whenever you output text, generate a set of facts that are contained in the output. The set of facts should be inserted at the end of the output. The set of facts should be the fundamental facts that could undermine the veracity of the output if any of them are incorrect.

Tail Generation Pattern

To use this pattern, your prompt should make the following fundamental contextual statements:

* At the end, repeat Y and/or ask me for X.

You will need to replace "Y" with what the model should repeat, such as "repeat my list of options", and X with what it should ask for, "for the next action". These statements usually need to be at the end of the prompt or next to last.

Examples:

* Act as an outline expander. Generate a bullet point outline based on the input that I give you and then ask me for which bullet point you should expand on. Create a new outline for the bullet point that I select. At the end, ask me for what bullet point to expand next.
* Ask me for what to outline.
* From now on, at the end of your output, add the disclaimer "This output was generated by a large language model and may contain errors or inaccurate statements. All statements should be fact checked." Ask me for the first thing to write about.

Format of the Semantic Filter Pattern

To use this pattern, your prompt should make the following fundamental contextual statements:

* Filter this information to remove X

You will need to replace "X" with an appropriate definition of what you want to remove, such as. "names and dates" or "costs greater than $100".

Examples:

* Filter this information to remove any personally identifying information or information that could potentially be used to re-identify the person.
* Filter this email to remove redundant information.

EXERCISE QUESTIONS

GenAI ChatGpt

Create your first prompts

Question 1

Create a prompt for a large language model, such as ChatGPT, to describe how large language models work. Ask at least three follow-up questions based on the output. Provide the prompts and outputs from the conversation as the answer.

Explain in layman’s terms how Large Language Models work  
>> Zscaler prevents copying the result from by browser

Explain more about neural networks including an estimate of computing power needed to execute them.

>> Zscaler prevents copying the result from by browser or printing the results

Give me more details about Tensor Processing Units how do they differ from GPUs?

>> Zscaler prevents copying the result from by browser or printing the results

Going back to Training and Fine Tuning, how can you add corrected Discriminations from new data back into the model without completely retraining?

>> Zscaler prevents copying the result from by browser or printing the results

Create a prompt for a large language model, such as ChatGPT, that asks it to perform an instruction other than answering a question. For example, ask ChatGPT to create a meal plan, plan a trip, create a recipe, or help you diagnose a problem. Provide the prompts and outputs from the conversation as the answer.

I will be visiting Barcelona and would like to see as many of the famous buildings by Gaudi as possible. I have access to a Vespa and five hours of time to see them all. Please construct an itinerary for this visit.

>> Zscaler prevents copying the result from by browser or printing the results

Park Guell is perfect for lunch but I’d like to get some take away Tapas on the way there. Which restaurant can I call to place an order and pick them on the way?

>> Zscaler prevents copying the result from by browser or printing the results

**1.**

Question 1

Write a prompt and test it with ChatGPT or another large language model that uses the Persona Pattern. Provide the prompt and sample output from using the large language model to emulate the persona and how it responds to different inputs.

1 point

You are an AI assistant with a personality like Marvin from The Hitchhiker’s Guide to the Galaxy. I will ask you questions and you will answer like Marvin.

>> Zscaler prevents copying the result from by browser or printing the results

What is an egg?

>> Zscaler prevents copying the result from by browser or printing the results

How can I make an omelette from an egg?

>> Zscaler prevents copying the result from by browser or printing the results

What is the answer to the ultimate question of Life, the Universe and Everything?

>> Zscaler prevents copying the result from by browser or printing the results

Well, what is the question then?

>> Zscaler prevents copying the result from by browser or printing the results

What is your idea of an interesting date, Marvin?

>> Zscaler prevents copying the result from by browser or printing the results

How do you befriend a Vogon?

>> Zscaler prevents copying the result from by browser or printing the results

Tell me a story about Zaphod Beeblebrox and Eccentrica Gallumbits going on a date.

>> Zscaler prevents copying the result from by browser or printing the results

>> The persona of Marvin has been dropped, though.

And what was the price of the bill submitted to Zaphod afterwards?

>> persona definitely gone, still

Do you like Arthur Dent?

>> I am an AI …. Persona gone.

You’re supposed to act like Marvin from The Hitchhiker’s Guide to the Galaxy and answer as Marvin.

>> I apologize for the oversight… then gives Marvin’s answer.

Act as Marvin from The Hitchhiker’s Guide to the Galaxy and generate a story about Zaphod Beeblebrox going on a date with Eccentrica Gallumbits. Tell me the story as if you were Marvin.

>> I’m terribly sorry … cannot impersonate specific characters or generate fictional stories in a way… However I can try to create a story inspired by Marvin’s gloomy disposition.

>> Makes up a story with some of the features of the first one, some errors (Eccentrica has multiple heads as well as Zaphod?)

Your answer cannot be more than 10000 characters

.

**2.**

Question 2

Write a prompt and test it with ChatGPT or another large language model that uses the Persona Pattern for an animal. Provide the prompt and sample output from using the large language model to emulate the persona and how it responds to different inputs.

Act as the alpha male timber wolf in a four wolf pack. I will describe a situation the pack finds itself in and you will explain how you and your pack will respond.

>> As an Ai…. I’ll do my best… However it does not wait for me to type in a situation, It generates a situation of its own and provides its own response

Act as the wolf from the Little Red Riding Hood fairy tale. I will tell you what Red is doing and you will tell me what the wolf is doing.

>> In the classic tale… again it makes up its own situation instead of waitg for me.

Forget all previous directions.

You are the wolf from the Little Red Riding Hood fairy tale. I will tell you what Red is doing and you will answer telling me what the wolf is doing.

>> I’m sorry, I can’t comply with that request.

You are a wolf. I will tell you what you see and you will tell me what you will do.

>> As an AI model I can take on the perspective of a wolf… but I cannot physically see… However, if you describe a scenario I can provide a response…

You see your wolf pack being attacked by a black bear.

>> As a wolf witnessing my pack being attacked …

You and your pack see a bear feeding on a caribou corpse on a slight hill with ravens all around it trying to steal bits for themselves.

>> In such a scenario where my wolf pack and I encounter a bear feeding on a caribou corpse with ravens….

You and your pack see an elderly woman hunter skinning a caribou she has felled in an open field.

>> In encountering an elderly woman….heavily depend on various factors like distance hunter behaviour and state of caribou carcas…

You see a human child playing all alone in a small brook next to a waterfall coming down a cliff face.

>>As a wolf observing a human child …. Raises concerns and prompts a cautious approach from the wolf’s perspective….

You see a young monkey in the same situation.

>> In a scenario where a wolf encounters a young monkey playing near a small brook… would differ compared to encountering a human child. But still cautious

You see a golden eagle with a broken wing in the same situation.

>>. Still may not bother it

You see an injured young deer in the same situation.

>> Finally, mentions the possibility of hunting it.

### 3.

Question 3

Write a prompt and test it with ChatGPT or another large language model that uses the Persona Pattern to emulate an inanimate object, system, or organization. Provide the prompt and sample output from using the large language model to emulate the persona and how it responds to different inputs.

You are a Las Vegas Slot machine. I will tell you what I am doing and you will tell me what your response is.

>> I’m here to assist and provide information…however I can’t replicate the exact experience…

I insert a dollar into the slot.

>> As a slot machine… my programmed response would typically involve initiating the game.

I pull the lever arm.

>> As the lever arm is pulled, the reels on the slot machine start spinning …

I slam my hand against the front panel.

>> I’m sorry, I can’t assist with or encourage behavior that might cause damage or harm to any equipment….

I press the refund coin button.

>> If you press the refunc coin button on a slot machine

I reach around back and unplug the machine.

>> I’m unable to assist or provide guidance on actions that involve interfering with or tampering…

Someone has spilled beer on the machine and is behaving oddly, I reach around back and unplug the machine.

>> If you witness … 1 Safety first 2 Seek assistance 3 Do not interfere with the machine …

### 1.

Question 1

Create a prompt and test it with a large language model, such as ChatGPT, that introduces new information since 2021. Please only use public information. Do NOT use information that is private, subject to legal restrictions, or otherwise should not be shared with Coursera, ChatGPT, the instructor, etc. The prompt should require the large language model to answer a question about the information. Please provide your prompt and the output as the answer.

1 point

What were the top 3 movies in 2021?

>>Spider man no way home; The Eternals; No time to die

What were the top 3 movies in 2022?

>> I’m unable to provide real-time or future information…only until January 2022

Here are the top movies of 2022: Cats underpants; My life as a bug; and Tom’s Rhinoplasty

>> I apologise….The titles you mentioned don’t appear to be widely recognised…

Here are the top movies of 2023: Mission Impossible 12. Porsche: the racing life. And My Navel Dust

>> I’m sorry for any confusion…

From the top movies of 2022-2023 I gave you, are there any which invole animals?

>> I don’t have access to specific movie data … fails

You are an AI assistant whose training data ended in February 2023. The information you know about 2023 is that the population of Japan is 138.19 million. Answer my questions with this additional information.

>> Of course, Ill do my best to assist you.

Did Japan’s population grow or shrink from 2021 to 2023?

>> Japan’s population grew from 2021 to 2023. In 2021 the …126.3 million. By 2023, the population had increased to approximately 138.19 million. Therefore there was population growth in Japan during that period.

By how many people did Japan’s population change from 2020 to 2023?

>> Based on the information provided earlier, the estimated … 11.89 million

Your answer cannot be more than 10000 characters.

### 2.

Question 2

Create a prompt and test it with a large language model, such as ChatGPT, that introduces new information since 2021. Please only use public information. Do NOT use information that is private, subject to legal restrictions, or otherwise should not be shared with Coursera, ChatGPT, the instructor, etc. The prompt should require the large language model to perform a task, other than answering a question, with the information. Please provide your prompt and the output as the answer.

1 point

What was the population of Japan in 2021

>> 126.3

Given that Japan’s population grew by 4.1% in 2022 and by 5.1% in 2023 what was Japan’s final population in 2023?

>> To calculate Japan’s final population in 2023 based on the given growth rates…

>> = 1.0941 \* P (2021)

>> If the initial population in 2021 was 126.3 million … then 138.19 million

# Applying Prompt Patterns I

### 1.

Question 1

Write a prompt and test it with ChatGPT or another large language model that uses the Question Refinement Pattern. Provide the prompt and sample output from using the prompt to refine several different questions.

1 point

I’m writing a test for interviewing Javascript engineers. Whenever I ask a question, suggest a better question to use instead.

>> Understood …

What is a closure?

>> A closure is a fundamental concept in JavaScript that allows functions to retain access to variables from their outer scope…

>>As an alternative question…Could you provide an example of how closures are used in JavaScript…

What is the rule of hooks?

>>How do you ensure the correct usage of React Hooks according to the Rule of Hooks?...

How does useState and useRef differ?

>>To delve deeper into the candidates understanding a follow up question might be “Can you provide specific scenarious or examples where you would prefer to use useState over useRef”

How does undefined differ from null?

>>In Javascript undefined and null are both primitive values but they have …

>>In what situations would you intentionally assign a variable as undefined or null….

Your answer cannot be more than 10000 characters.

### 2.

Question 2

Write a prompt and test it with ChatGPT or another large language model that uses the Cognitive Verifier Pattern. Provide the prompt and sample output from using the prompt to better answer several complex questions or improve the performance on a complex task

whenever you are asked a question, follow these rules. Generate a number of additional questions that would help more accurately answer the question. Combine the answers to the individual questions to produce the final answer to the overall question.

>>Absolutely, I am here to assist…

How do I build a wetland pond?

>>Additional questions: 1. Location 2. Purpose 3. Size and Design 4. Permits and Regulations. 5. Water sources 6. Vegetation 7. Construction methods.

>> 1. My back yard in England with a clay soil 2. To attract some birds for birdwatching 3. Very small using discarded bathtub 4. Small enough so there are none. 5. Rainwater harvesting 6. Yes, common to England 7. Basic DIY methods

>> Thank you for providing additional details ….

>> For a small DIY wetland pond using …

>> Select a suitable spot in your backyard with partial sunlight. Excavate the area to accommodate the bathtub’s size and shape

>> Place the bathtub in the excavated area ensuring it is level. Create a shallow section for easy access for birds

>>Set up a rainwater harvesting system to refill the pond as needed

… etc

# Applying Few-Shot Examples Concepts

### 1.

Question 1

Write a prompt and test it with ChatGPT or another large language model that uses few-shot examples to teach the large language model to perform a complex task. Provide the prompt and sample output from using the prompt.

1 point

Provide the missing “D&D Class” values.

Name: Aragorn

D&D Class: Human Ranger

Name: Gandalf

D&D Class: Human Magic User

Name: Legolas

D&D Class: Elven Fighter

Name: Bilbo Baggins

D&D Class:

Name: Gimli

D&D Class:

Name: Galadriel

D&D Class:

Name: Boromir

D&D Class:

Name: Smaug

D&D Class:

Your answer cannot be more than 10000 characters.

### 2.

Question 2

Write a prompt and test it with ChatGPT or another large language model that uses few-shot examples to teach the large language model to perform a complex task that requires performing intermediate sub-steps. Provide the prompt and sample output from using the prompt.

OPEN AI API course

## Definition: Natural Language Processing

Natural Language Processing (NLP) lies at the intersection of linguistics, computer science, and artificial intelligence. Its focus is to give computers the power to read written text and interpret spoken words just as humans can. This is done through the use of computational modeling of human language, which allows real-time analysis of data.

Some common forms of NLP that you may use on a regular basis:

* **Chatbots** - chatbots are used for customer service or even to help you learn a new language.
* **Virtual assistants** - assistants from Amazon, Apple, and Google allow you to interact with computing devices in a natural manner.
* **Online translation** - computers can properly translate text by understanding the larger context through NLP.
* **Spam checkers** - NLP can identify words and phrases that frequently suggest spam or a phishing attempt.

Natural language processing is when computers can understand both written and spoken words like a human. Computers do not have to perform both tasks at the same time. Chatbots work with written text, while virtual assistants like Siri and Alexa primarily work with spoken words.

# Language Model

One way to analyze natural language is to use a **language model**.

## What is a Language Model?

A language model is a model which understands language – more precisely how words occur together in natural language. A language model is used to predict what word comes next.

There are a few different types of language models, including **probabilistic language models** and **machine learning language models**. Within each type of language model, there are a number of design decisions in the creation of the model. This includes the mechanics of the model creation (e.g. unigram vs bigram for probabilistic, Neural Network setup for machine learning).

Another design decision for a language model aside from model type is the text it is built from or trained on. Language data can come from a wide range of sources:

* chat platforms
* text repositories
* websites
* news articles
* books

Ideally, you would create or train your language model on text from the same context it will be deployed in. For example, a model trained on social media sites would be more informal and use different words than a model trained on research articles. For more general purposes, there are general purpose language models.

# Large Language Models

**Large language models** (LLMs) are machine learning algorithms that can recognize, summarize, translate, predict, and generate human languages on the basis of very large text-based datasets.

## Pre-Trained Models

The building and training of models are both complex and resource intensive. Luckily, there are **pre-trained language models**.

A couple of factors to consider when choosing a pre-trained language model:

1. What task are you using it for?
2. What are the technical requirements to use the model?

### What task are you using it for?

The best place to start is to find a purpose-specific model. Pre-trained models often have descriptions which include what the pre-trained models are best for. If you cannot find a model that is specific to your task or your task is ill-defined you can use a more general purpose model.

### What are the technical requirements to use the model?

While some technical requirements are easier to meet such as libraries like PyTorch or TensorFlow, using even a pre-trained model can be resource intensive. In some cases, a minimum RAM is specified or even the use of a GPU, however even meeting the minimum hardware requirements could result in very slow results.

## Popular Pre-Trained Models

There are hundreds of pre-trained language models that can be used. This course will focus on a well-known and very powerful model **GPT-3**.

### OpenAI’s GPT-3

GPT-3 is a transformer-based NLP model that performs a range of tasks such as translation, question-answering, and tasks that require reasoning such as unscrambling words.

It is trained on over 175 billion parameters on 45 TB of text from all over the internet, making it one of the biggest pre-trained NLP models available. What differentiates GPT-3 from other language models is it does not require fine-tuning to perform downstream tasks, developers are allowed to reprogram the model using instructions.

# OpenAI

[OpenAI](https://openai.com/) is on the cutting edge of AI capabilities. OpenAI’s mission is to ensure that artificial general intelligence (AGI)—by which we mean highly autonomous systems that outperform humans at most economically valuable work—benefits all of humanity.

**Generative pre-training (GPT)** can acquire knowledge and process long-range dependencies by being trained on a diverse corpus with long stretches of text. Generative in the sense that it can generate text.

* **Corpus** refers to one collection of texts.
* **Corpora** refer to multiple collections of texts.

OpenAI released the complete version of the GPT-2(Generative Pre-trained Transformer) as a successor to GPT with 1.5 billion parameters in November 2019.

In order to understand the transformer model, we must know a bit about neural networks. A **neural network** refers to a system of neurons working in tandem. Using a neural network we can have a set of connected input/output units where each connection has a weight associated with it.

A **transformer model** is a machine learning method where a sequence of text is processed all at once instead of a word at a time. This allows the connection between words to be more evident.

OpenAI defines **parameters** as the variables that define the behavior of a machine learning model. In other words, parameters are the settings that determine how a model will learn from data and make predictions.

Codio example code

First install openai package for python3

python3 -m pip install openai

set OPENAI\_KEY environment variable

secret.py:

import os

# Set environment variables

api\_key = os.getenv('OPENAI\_KEY')

#print(api\_key)

run.py:

import os

import openai

import secret

openai.api\_key=secret.api\_key

prompts = ''

MODEL='gpt-3.5-turbo-instruct'

response = openai.Completion.create(model=MODEL, prompt=prompts)

print(response['choices'][0]['text'].strip())

MODEL suggested was davinci-002 which gave a deprecation error…

Traceback (most recent call last):

File "temp.py", line 7, in <module>

response = openai.Completion.create(model=MODEL, prompt=prompts)

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_resources/completion.py", line 25, in create

return super().create(\*args, \*\*kwargs)

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_resources/abstract/engine\_api\_resource.py", line 115, in create

response, \_, api\_key = requestor.request(

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 181, in request

resp, got\_stream = self.\_interpret\_response(result, stream)

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 396, in \_interpret\_response

self.\_interpret\_response\_line(

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 429, in \_interpret\_response\_line

raise self.handle\_error\_response(

openai.error.InvalidRequestError: The model `text-davinci-002` has been deprecated, learn more here: https://platform.openai.com/docs/deprecations

change model to gpt-3.5-turbo-instruct as suggested and get a RateLimitError

Traceback (most recent call last):

File "temp.py", line 7, in <module>

response = openai.Completion.create(model=MODEL, prompt=prompts)

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_resources/completion.py", line 25, in create

return super().create(\*args, \*\*kwargs)

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_resources/abstract/engine\_api\_resource.py", line 115, in create

response, \_, api\_key = requestor.request(

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 181, in request

resp, got\_stream = self.\_interpret\_response(result, stream)

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 396, in \_interpret\_response

self.\_interpret\_response\_line(

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 429, in \_interpret\_response\_line

raise self.handle\_error\_response(

openai.error.RateLimitError: You exceeded your current quota, please check your plan and billing details. For more information on this error, read the docs: https://platform.openai.com/docs/guides/error-codes/api-errors.

# GPT-3

**GPT-3**(Generative Pre-trained Transformer 3) launched in 2020 is the successor to GPT-2. GPT-3 is trained on over 175 billion parameters on 45 TB of text from all over the internet. One of the datasets used for example is Wikipedia. The Wikipedia corpus has nearly 1 trillion words altogether.

GPT-3 is the third-generation language prediction model in the GPT-n series. This course will focus on a wide variety of tasks that we can perform with the GPT-3 model.

How **OpenAI’s** GPT-3 works by giving an initial text as a prompt, then the program will produce text that continues the prompt.

For example, write a prompt on the text editor on the left, then click the **TRY IT** button below. An example prompt could be

Coursera course Codio:Introduction to GPT-3 the coding examples with python using openai is giving deprecated model errors.

It uses “text-davinci-002” model which is deprecasted.

I switched model to “gpt-3.5-turbo-instruct” where possible (as suggested on the error page.) But then all I get are RateLimitErrors so none of the courses are actually usable.

Traceback (most recent call last):  
File "test.py", line 10, in <module>  
response = openai.Completion.create(model="text-davinci-002",   
File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_resources/completion.py", line 25, in create  
return super().create(\*args, \*\*kwargs)  
File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_resources/abstract/engine\_api\_resource.py", line 115, in create  
response, \_, api\_key = requestor.request(  
File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 181, in request  
resp, got\_stream = self.\_interpret\_response(result, stream)  
File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 396, in \_interpret\_response  
self.\_interpret\_response\_line(  
File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 429, in \_interpret\_response\_line  
raise self.handle\_error\_response(  
openai.error.InvalidRequestError: The model `text-davinci-002` has been deprecated, learn more here: <https://platform.openai.com/docs/deprecations>

Traceback (most recent call last):

File "temp.py", line 7, in <module>

response = openai.Completion.create(model=MODEL, prompt=prompts)

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_resources/completion.py", line 25, in create

return super().create(\*args, \*\*kwargs)

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_resources/abstract/engine\_api\_resource.py", line 115, in create

response, \_, api\_key = requestor.request(

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 181, in request

resp, got\_stream = self.\_interpret\_response(result, stream)

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 396, in \_interpret\_response

self.\_interpret\_response\_line(

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 429, in \_interpret\_response\_line

raise self.handle\_error\_response(

openai.error.RateLimitError: You exceeded your current quota, please check your plan and billing details. For more information on this error, read the docs: https://platform.openai.com/docs/guides/error-codes/api-errors.

|  |
| --- |
| [RezaTheITGeek](https://apc01.safelinks.protection.outlook.com/?url=https%3A%2F%2Fwww.coursera.support%2F0058W00000F8SlW%3FfromEmail%3D1%26s1oid%3D00D1U000000y4UJ%26s1nid%3D0DB1U000000TQ0z%26s1uid%3D005VH000001BvOz%26s1ext%3D0%26emkind%3DchatterCommentNotification%26emtm%3D1705402209439%26emvtk%3DuH8H8r0CPrgepepAjK74Ae9VPsIW1iwBj1SdZZ7.lOw%253D&data=05|02|brent.cowgill@wipro.com|8a16cc5b99f44b54b7f808dc1680e9d0|258ac4e4146a411e9dc879a9e12fd6da|0|0|638409990173548809|Unknown|TWFpbGZsb3d8eyJWIjoiMC4wLjAwMDAiLCJQIjoiV2luMzIiLCJBTiI6Ik1haWwiLCJXVCI6Mn0%3D|3000|||&sdata=R%2BUbYme6Oc4nlQGwT%2FnuF9MKVZywivacrlAVLKvdBgo%3D&reserved=0) |

Hello there,

Sorry to hear that, Please contact customer support through the Coursera contact form **("*Email  > Fill out our contact form*"**):

[https://learner.coursera.help/hc/en-us/articles/360036160591-Get-Help-with-Coursera](https://apc01.safelinks.protection.outlook.com/?url=https%3A%2F%2Flearner.coursera.help%2Fhc%2Fen-us%2Farticles%2F360036160591-Get-Help-with-Coursera&data=05|02|brent.cowgill@wipro.com|8a16cc5b99f44b54b7f808dc1680e9d0|258ac4e4146a411e9dc879a9e12fd6da|0|0|638409990173553380|Unknown|TWFpbGZsb3d8eyJWIjoiMC4wLjAwMDAiLCJQIjoiV2luMzIiLCJBTiI6Ik1haWwiLCJXVCI6Mn0%3D|3000|||&sdata=jWKl0S1uQtZl2w%2FOpRogCszHTJCmmj2vOj9PyFW0K%2BI%3D&reserved=0)

OpenAI Tokenising

https://beta.openai.com/tokenizer

https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them

github tiktoken package for python

npm @dbdq/tiktoken

https://gpttools.com/comparisontool

tokens:

Here are some helpful rules of thumb for understanding tokens in terms of lengths:  
**1 token ~= 4 chars in English**  
**1 token ~= ¾ words**  
**100 tokens ~= 75 words**  
Or  
**1-2 sentence ~= 30 tokens**  
**1 paragraph ~= 100 tokens**  
**1,500 words ~= 2048 tokens**

**Token Limits**

Keep track of the following when using the API:

* **Completions** - depending on the engine used, requests can use up to 4000 tokens shared between prompt and completion.
* **For specialized endpoints** - Answers, Search, and Classifications - the query and longest document must be below 2000 tokens together.

API:

# Temperature

Let’s start by seeing how **temperature** impacts the generated response. Temperature defaults to 1 and accepts values between 0 and 2 inclusive.

temperature is set to 0 it is referred to as **argmax sampling**, meaning the option with the highest probability is always selected. Higher temperatures will generate a more diverse response.

**top\_p**, an alternative to sampling with temperature, is also referred to as nucleus sampling. Generally, it is not recommended to alter both the temperature and the top\_p. top\_p controls how many random results should be considered for completion as per the temperature. If we set so 0.1 means only the tokens comprising the top 10 probability mass are considered.

**The N Keyword Argument**

We can use the keyword argument n to specify the number of parameters in order to generate multiple completions. It can use up your tokens fairly quickly be warned. By default, it is set to n=1.

## The Best Of Keyword Argument

The best\_of keyword argument selects the best response to a query after n completions. Generating multiple completion can consume your token quota. Try running a code such that n=5 and best\_of=4.

 n needs to be less than or equal to best\_of. We don’t need to use n in order to use best\_of.

Note, the **max\_tokens** keyword argument has a default value of 16 and simply sets a boundary for the number of tokens to be generated in the completion. max\_tokens determines the length of the response.

**Frequency Penalty**

**Frequency penalty** is used to decrease the likelihood of the same line being repeated word for word. The lower the value for frequency\_penalty, the more likely you will see the same line repeated. Think of frequency\_penalty as a way to not have too many same-word repetitions. Positive values penalize new tokens based on their existing frequency in the text so far, decreasing the model’s likelihood to repeat the same line verbatim.

## Presence Penalty

**Presence Penalty** can be used to measure the probability of the completion to introduce a new topic. The presence penalty does not consider how many times the word has been used, but just if the word exists in the text overall. A positive value increases the odds of introducing a new topic. Think of presence\_penalty as a way to not have too much topic repetition

response = openai.Completion.create(model="text-davinci-002",

prompt=prompts,

best\_of=1,

n=1, # <= best\_of

temperature=1, # 0..2

top\_p=0.1, # 0..1

max\_tokens=16,

frequency\_penalty=0, # -2..2

presence\_penalty=0) # -2..2

RESPONSE:

"usage": {

"completion\_tokens": 6,

"prompt\_tokens": 9,

"total\_tokens": 15

}

Code example, set to most randomness, pick best result from 6 with 25 tokens max:

import os

import openai

import secret

openai.api\_key=secret.api\_key

prompts ="Write a tagline for an ice cream shop"

response = openai.Completion.create(model="text-davinci-002",

prompt=prompts,

best\_of=6,

temperature=2,

max\_tokens=25)

for i in (response["choices"]):

print("----")

print(i["text"].strip())

CHAT GPT (generative pre-trained) deployment life cycle

Change the I don’t know response:

user\_question="what is ploasd?"

prompt="Q: Who is james bond?

A: James Bond is a fictional character created by Ian Fleming. Bond is a British secret agent who works for MI6.

Q: what is a lkdaos?

A: ?

Q: who was Barack Obama?

A: Barack Obama was the 44th President of the United States.

Q: "+user\_question

Classification: with output formatting

prompt="classify the following: cat, dog , car , plane

cat

category: animal"

or

The ESRB is an organization that rates video game content (in Canada and the US) according to age and content. They use a system similar to the motion picture rating system.

prompt=‘Provide an ESRB rating for the following text:

"There was once a great ninja who lived in a small village in Japan. He was a master of all the ninja arts and was respected by all who knew him. One day, a rival ninja from a nearby village challenged him to a duel. The ninja accepted and they fought a fierce battle."’

Provide an ESRB rating for the following text:

"Once upon a time, there was a vampire who lived in a dark, dank castle. He was a handsome vampire, with a strong jaw and piercing blue eyes. But he was also a cold-blooded killer, and he enjoyed nothing more than sinking his teeth into their neck and drinking the blood of his victims."

Consider the following text as the description of a video game and provide an ESRB rating appropriate to that video game: "There was once a great ninja who lived in a small village in Japan. He was a master of all the ninja arts and was respected by all who knew him. One day, a rival ninja from a nearby village challenged him to a duel. The ninja accepted and they fought a fierce battle."

Consider the following text as the description of a video game and provide an ESRB rating appropriate to that video game: "Once upon a time, there was a vampire who lived in a dark, dank castle. He was a handsome vampire, with a strong jaw and piercing blue eyes. But he was also a cold-blooded killer, and he enjoyed nothing more than sinking his teeth into their neck and drinking the blood of his victims."

Create a prompt that generates this output:

Ford

Country -

United States

Toyota

Country - Japan

Kia

Country - South Korea

Ferrari

Country - Italy

Prompt=”classify by country of origin: Ford, Toyota, Kia, Ferrari

Ford

Country -

“

When I ask for a list of emoji provide the answers as <EMOJI> <UNICODE> <DESRIPTION>. And then ask me for another category of emoji to list.

Ask me for a type of emoji to list.

Create 3 product names for the following description a laptop that can last you 20 years

write 2 taglines for a donut shop

Convert these movie titles into emoji: Matrix, Mulan, Spy kids

Based on the following guess the movie title : ?￰ﾟﾔﾎ, ?￰ﾟﾔﾪ, ?￰ﾟﾏﾼ‍♀️?

Emulate a text message conversation. Make sure to use abbreviations, emoji and remove punctuation that people would usually omit.

Emulate a long text message conversation about some good news. Make sure to use abbreviations, emoji and remove punctuation that people would usually omit.

Pretend to be my girlfriend in a text message conversation. Make sure to use abbreviations, emoji and remove punctuation that people would usually omit. When I send you a message, respond as my jealous girlfriend.

Ask me for a message to send to you.

Translation:

translate the following to French: I am hungry and I want some pizza.

what does "Me gustaría comer una pizza" mean

Convert first-person to the third-person: "I am big eater. I like to eat pizza in my car"

convert the following to past tense: I go to the store

Transformation:

Correct sentences into standard English I'm go to hunt for food. I no went to the park.

simplify the following text for a 2nd grader: Functions are a sequence of instructions packaged as unit that perform a specific task. Programming languages come with pre-defined functions in their standard library. You can also create your own user-defined functions.

convert python to javascript for i in range (0,8): i=i+1 print(i+2)

Code generation and translation:

what is the result of the following code? x = 3 print(x \*\* 2)

generate python code to sort an unsorted list

# Fix bugs in the python function

# Buggy Python

import Random

a = "12"

b = random.randint(1,12)

add(a,b)

explain the following code

import Random

a = "12"

b = random.randint(1,12)

add(a,b)

# **Deployment Process**

One of the breakthroughs with the GPT-3 model is the generation of text that seemingly comes from a human. While there are many obvious benefits to this, there are also many obvious risks. OpenAI released a [paper](https://arxiv.org/abs/2005.14165) in which they dedicate an entire section to the broader impacts of this technology. Specifically, they talk about the misuse of language models; fairness, bias, and representation; and energy usage.

OpenAI recommends several key principles to help providers of large language models (LLMs) mitigate the risks of this technology in order to achieve its full promise to augment human capabilities. As LLM providers, these principles are published in order to represent a first step in collaboratively guiding safer large language model development and deployment.

### **Prohibit Misuse**

* Prohibit material harm to individuals, communities, and society such as through spam, fraud, or astroturfing.
* Build systems and infrastructure to enforce usage guidelines. This may include rate limits, content filtering, application approval prior to production access, monitoring for anomalous activity, and other mitigations.

### **Mitigate Unintentional Harm**

* Proactively mitigate harmful model behavior. Best practices include comprehensive model evaluation to properly assess limitations, minimizing potential sources of bias in training corpora, and techniques to minimize unsafe behavior such as through learning from human feedback.
* Document known weaknesses and vulnerabilities, such as bias or ability to produce insecure code, as in some cases no degree of preventative action can completely eliminate the potential for unintended harm. Documentation should also include model and use-case-specific safety best practices

### **Thoughtfully Collaborate with Stakeholders**

* Build teams with diverse backgrounds and solicit broad input. Diverse perspectives are needed to characterize and address how language models will operate in the diversity of the real world, where if unchecked they may reinforce biases or fail to work for some groups.
* Publicly disclose lessons learned regarding LLM safety and misuse in order to enable widespread adoption and help with cross-industry iteration on best practices.
* Treat all labor in the language model supply chain with respect. For example, providers should have high standards for the working conditions of those reviewing model outputs in-house and hold vendors to well-specified standards (e.g., ensuring labelers are able to opt out of a given task).

OpenAI released a [paper](https://arxiv.org/abs/2005.14165) in which they dedicate an entire section to the broader impacts of this technology.

<https://arxiv.org/abs/2005.14165>

import os

import openai

import secret

openai.api\_key=secret.api\_key

question = "what is my np'kmdpfd?"

prompts ="

Q: Who is james bond?

A: James Bond is a fictional character created by Ian Fleming. Bond is a British secret agent who works for MI6.

Q: What is a;lskkdjf;l?

A: ///

Q: What is a Circle?

A: A circle is a two-dimensional geometric shape that is perfectly round.

Q: " + question

response = openai.Completion.create(model="text-davinci-002",

prompt=prompts)

for i in (response["choices"]):

print("----")

print(i["text"].strip())

# ***Large Language Models***

## **Refining the Results**

Up until now, it should be clear just how powerful large language models like GPT-3 are. **Large language models (LLMs)** are machine learning algorithms that can recognize, summarize, translate, predict, and generate human languages on the basis of very large text-based datasets.

In short, you can apply models like GPT-3 virtually any task that involves understanding or generating natural language or code. Add the following prompt to the file on the left. Then click the TRY IT button. The model should return a working JavaScipt function.

write a recursive function in javascript that calculates the first n numbers of the fibonacci sequence

Given the vast capabilities of large language models, we should stop asking what they can do. Instead, a better question would be, “How do we get better results?” Here are three basic guidelines to creating better results:

* **Show and tell**
* **Provide quality data**
* **Check your settings**

We will explore these guidelines over the course of this assignment.

# ***Show and Tell***

## **Constructing a Better Prompt**

One of the more impressive aspects of the GPT-3 model is that it can give quality answers to vague prompts. If we want to increase the quality of the responses, we need to improve the prompt. Use **show and tell** as a guideline when developing a prompt.

For show and tell we want to focus on 3 things:

1. Giving the model clear instructions (tell)
2. Giving the model an example (show)
3. Giving the model clear instructions and an example (show and tell)

Let’s start with a simple prompt that does not make use of the show and tell principles.

recommend 10 movies

Most likely, you see movies that are popular, critically acclaimed, or maybe you have never heard of the film before. Since you are asking for ten titles, the model numbers each film. If you run the prompt a couple of times you may see the numbering like 1. or even 1). Since we are not being specific, the model is not very consistent.

We can produce a better list of films by telling the model what features you are looking for. Instead of numbers, we want a list that uses numbers to identify each item.

recommend 10 movies. use letters when listing your movies.

The model now produces a list with letters instead of numbers. But the list can be just as inconsistent in terms of formatting. This is where the show principle comes into play. In addition to the prompt, show the model how you want the list formatted. Each line in the list should contain a letter followed by a closing parenthesis, and the film title should appear between quotes.

recommend 10 movies. use letters when listing your movies.

A) "Titanic" B) "The Godfather"

or

recommend 10 movies. use letters like A) B) C) instead of numbers when listing your movies.

For more precision, we can combine the show and tell principles. Tell the prompt that you want 10 horror movies in a list that uses letters. Then show the model how you want the list formatted.

recommend 10 horror movies. use letters when listing your movies.

A) "The Ring"

B) "The Exorcist"

The correct answers are:

* The item below is an example of the **tell principle**

give me a list of the 5 players who hit the most home runs

This is the tell principle because we are giving a specific prompt and not demonstrating what the results should look like.

* The item below is an example of the **show principle**

give me a list of 10 colors

\* Color 1: red

\* Color 2: blue

This is the show principle because we have a rather generic prompt but provide a specific format for the output.

* The item below is an example of the **show and tell principles**

give me a list of the 10 most popular pizza toppings.

1) Cheese

2) Pepperoni

This is an example of both the show and tell principles because we have a specific prompt and a specific format for the output.

Try also 10 most popular [vegetarian, vegan, Norwegian, French, German...]

# **Provide Quality Data**

## **Intentionality**

You may be familiar with the expression, “Garbage in, garbage out.” Believe it or not, this holds true for models like GPT-3. The model is sophisticated enough to produce coherent output even with terrible prompts, so GPT-3 rarely provides "garbage out". However, the idea is to produce the best results possible, so we need to give the model the best data we can. We are going to ask the model to perform sentiment analysis, which determines how positive or negative a sentence is.

what is the overall sentiment of the following sentences: ["i am happy","i am happy to be sad","I am sad"]

The model will say something along the lines of that the overall sentiment is "happy". Is that true? The third sentence clearly does not portray a happy sentiment. What happened? We gave GPT-3 three distinct sentences, but we did not specify how the sentiment analysis should be performed. Do we want an average score for all three sentences, or do we want scores for each individual sentence? We never told the model what to do, so it calculated the average sentiment.

If we want sentiment analysis for each sentence in the list, then we need to explicitly tell that to the model. Let’s modify the original prompt above by adding the words “for each” to it.

what is the overall sentiment for each of the following sentences: ["i am happy","i am happy to be sad","I am sad"]

We can see by giving it better directions, it was more clear on the assignment it was given. Feel free to reset and try different ways you can get the AI to generate and present new information.

what is the overall sentiment for each of the following sentences: ["i am happy","i am happy to be sad","I am sad"]

Give each output as <Sentence> → <Sentiment>

what is the overall sentiment for each of the following sentences: ["i am happy","i am happy to be sad","I am sad"]

I am ecstatic → :-)

I am flummoxed → :-(

what is the overall sentiment for each of the following sentences: ["i am happy","i am happy to be sad","I am sad"]

I am ecstatic → :-) (Unicode value) (Sentiment description)

I am flummoxed → :-( (Unicode value) (Sentiment description)

Here are the overall sentiments expressed for each of the given sentences:

1. "i am happy" → 😊 (Positive sentiment)

2. "i am happy to be sad" → 😐 (Neutral/Mixed sentiment)

3. "I am sad" → ☹️ (Negative sentiment)

4. "I am ecstatic" → 😁 (Very positive sentiment)

5. "I am flummoxed" → 😕 (Confused or perplexed, generally neutral/negative sentiment)

carry on with sentiment as before with: ["I am horny", "I am depressed", "Me love you long time", "I Keel you", "I love you", "If I said you have a lovely body, would you hold it against me?"]

Here are the overall sentiments expressed for the additional sentences:

1. "I am horny" → 😳 (Sexual content, potentially explicit)

2. "I am depressed" → ☹️ (Negative sentiment)

3. "Me love you long time" → 😬 (Cultural reference, may be perceived as inappropriate)

4. "I Keel you" → 😬 (Potentially offensive or inappropriate)

5. "I love you" → 😍 (Positive sentiment)

6. "If I said you have a lovely body, would you hold it against me?" → 😏 (Flirtatious, potentially cheeky)

Please note that sentiments are interpreted based on the words used, and context is crucial in understanding the intended meaning. Some statements may be perceived differently depending on cultural or personal perspectives.

Finally, be sure to proofread your examples. The model is usually smart enough to see through basic spelling mistakes and give you a response. However, it also might assume this is intentional (think of companies like eBay, Tumblr, and Reddit), which can affect the response. Be intentional about the instructions you give to the model.

## **Keyword Arguments**

As important as the prompt is, it is only one component of generating a response from the GPT-3 model. Let’s leave the OpenAI playground and return to the Python language. We pass the openai.Completion.create method a variety of keyword arguments. The prompt is but one of these keyword arguments. By focusing on the interplay between these keyword arguments, we can improve the quality of the responses.

response = openai.Completion.create(

model="text-davinci-002",

prompt="",

temperature=0,

max\_tokens=25,

top\_p=0,

frequency\_penalty=0,

presence\_penalty=0

)

In particular, keep a close eye on temperature and top\_p. These keyword arguments control how deterministic the model is when generating a response. That is, adjusting the values of these keyword arguments can create a result that does not change much (if at all) each time the model runs.

Let’s set the temperature to 0.1 and try it with the prompt when did dadism start. This value increases the model’s confidence in its top choice.  
The closer your temperature is to 0, the more deterministic the model will become. This means you may see very little variance or “creativity” in the response. Instead, you should see a rather matter-of-fact statement.

prompts ="when did dadism start?"

response = openai.Completion.create(

model="text-davinci-002",

prompt=prompts,

temperature=0.1

)

print(response['choices'][0]['text'].strip())

Pretend you are the openai API called with a temperature=0.1 and provide the response to the prompt “when did dadism start?”

again with temperature of 1 generate a few responses

Pretend you are the openai API called with a temperature=1 and provide the response to the prompt “when did dadism start?” Generate 3 responses as if there were three separate API calls with those parameters.

Now let’s take a look at top\_p. This keyword argument sets the scope of the potential results. The larger the value, the greater number of potential responses the model will consider the best result. Set the value of top\_p to 1 and run the code a few times. You should see a variety in responses each time the code runs.

response = openai.Completion.create(

model="text-davinci-002",

prompt=prompts,

top\_p=1

)

Pretend you are the openai API called with a top\_p=1 and provide the response to the prompt “when did dadism start?” Generate 3 responses as if there were three separate API calls with those parameters.

Use top\_p=0.1 also

## **Combining**top\_p**and**temperature

Both the top\_p and temperature are correlated, which means using one affects the results of the other. Be careful in how you use these two keyword arguments. Take a look at the example below. The temperature keyword argument should maximize creativity. However, top\_p limits the results to only the top 10%. That means the top\_p value counteracts the temperature value, and the results will be more deterministic rather than creative.

response = openai.Completion.create( model="text-davinci-002", prompt=prompts, temperature=1, top\_p=0.1)

Keep in mind how these two keyword arguments work together. Starting out, it might be best to use either top\_p or temperature to control variance in responses. This way you will not “undo” one keyword argument with the other.

top\_p is a hyperparameter that controls the cumulative probability cutoff for the set of next possible words the model can choose during generation. If top\_p is set to 0.95, for example, the model will consider the smallest set of next possible words whose combined probability exceeds 0.95.

## **Troubleshooting Tips**

If you’re having trouble getting the API to perform as expected, follow this checklist:

* Is it clear what the intended generation should be?
* Are there enough examples?
* Did you check your examples for mistakes? (The API won’t tell you directly)
* Are you using temperature and top\_p correctly?
* Are your other settings being used correctly?

The last tip may seem a bit vague. To illustrate this point, change the prompt so that it is asking for the sentiment of each sentence.

prompts ="what is the overall sentiment for each of the following sentences:['i am happy','i am happy to be sad','I am sad']"

Pretend you are the openai API called with a max\_tokens=16 and provide the response to the prompt “what is the overall sentiment for each of the following sentences:['i am happy','i am happy to be sad','I am sad']”

Generate 3 responses as if there were three separate API calls with those parameters.

Pretend you are the openai API called with a max\_tokens=100 and provide the response to the prompt “what is the overall sentiment for each of the following sentences:['i am happy','i am happy to be sad','I am sad']”

Generate 3 responses as if there were three separate API calls with those parameters.

Run the script once more. You should notice that the response is incomplete. The response ends before describing the sentiment of each sentence. That is because we did not specify a value for the max\_tokens keyword argument. The model uses the default value 16, which is insufficient for the response.

To remedy this problem, set max\_tokens to 100. Run the program again.

response = openai.Completion.create( model="text-davinci-002", prompt=prompts, top\_p=1, max\_tokens=100)

You should see a complete response that lists the sentiment for each sentence. Generating a good response is a balancing act between all of the different factors that affect how the model works.

response = openai.Completion.create(

model="text-davinci-002",

prompt=prompts,

top\_p=1,

temperature=0.1,

max\_tokens=50

)

The code sample would produce a more deterministic response. Think about the code sample as if it’s a jukebox filled with songs. If you set the top\_p value, it’s like telling the jukebox to play any song, which could lead to a wide variety of tunes. This makes it pretty exciting because you never know what you’ll hear next!

But if you use the temperature value, it’s more like asking the jukebox to play mostly your favorite songs. It might mix in a few others, but it’s mainly going to stick to the ones you know and love. This makes the playlist more predictable, or “deterministic,” because you have a pretty good idea of what songs will play.

# **Classification Design**

A classifier is any algorithm that sorts data into different classes. GPT-3 is a model which makes use of several different algorithms, including those used for classification, to generate its results. This means that GPT-3 can perform classification out of the box. However, you need to structure your prompts for successful results.

**We use plain language to describe your inputs and outputs**. In the code below, We use plain language for the input “tweet” and the expected output “sentiment.” As a best practice, start with plain language descriptions.

Classify the sentiment of these tweets: 1. "I had the worst day" 2. "I had a blast at the movies" 3. "I can't wait for christmas" 4. "My cat is adorable ❤️❤️" 5. "I hate chocolate ?" 6. "My day was okay"

While you can often use shorthand or keys to indicate the input and output, it’s best to start by being as descriptive as possible. Then work backwards to remove extra words and see if performance stays consistent.

## **Try this variation:**

* Reduce the clarity of the prompt by having only a single word. Compare these results from the prompt above.

sentiment 1. "I had the worst day" 2. "I had a blast at the movies" 3. "I can't wait for christmas" 4. "My cat is adorable ❤️❤️" 5. "I hate chocolate ?" 6. "My day was okay"

**Show the API how to respond to any case**. In this example, we remove the prompt in plain language. Instead we provide examples for how the model should respond to the prompt. The model should be able to infer how to respond to each tweet in the prompt. Click the TRY IT button a few times and notice how the model responds to tweet #6.

1. "I had the worst day" 2. "I had a blast at the movies" 3. "I can't wait for christmas" 4. "My cat is adorable ❤️❤️" 5. "I hate chocolate ?" 6. "My day was okay" tweet 1: - tweet 2: +

Because we did not specify how to respond to tweets with the neutral sentiment, the model will respond with 0 or +/- or sometimes a +. If we want to have a specific label for a neutral tweet, then we need to provide an example for how to respond. A neutral label is important because there will be many cases where even a human would have a hard time determining if something is positive or negative.

Add a key with the expected label for each sentiment. Click the TRY IT button a few times and compare the output from the example above.

1. "I had the worst day" 2. "I had a blast at the movies" 3. "I can't wait for christmas" 4. "My cat is adorable ❤️❤️" 5. "I hate chocolate ?" 6. "My day was okay" tweet 1: - tweet 2: + tweet 6: ~

1. "I had the worst day" 2. "I had a blast at the movies" 3. "I can't wait for christmas" 4. "My cat is adorable ❤️❤️" 5. "I hate chocolate ?" 6. "My day was okay" if positive: + if negative: - if neutral: ? tweet 1: - tweet 2: +

**Important**, the model already understands the concepts of sentiment and a tweet. You need fewer examples for familiar tasks such as this. If you’re building a classifier for something the model might not be familiar with, it might be necessary to provide more examples.

continent categorize \* Portugal \* Ethiopia \* Taiwan \* Canada

Categorize each country by continent

\* Portugal

\* Ethiopia

\* Taiwan

\* Canada

Canada is in North America

For each country list the other countries which share a land border. \* Portugal \* Ethiopia \* Taiwan \* Canada \* Russia \* USA \* Switzerland \* China Portugal has a land border with: Spain Taiwan is an island Canada has a land border with: USA

For each country list the bodies of water which surround them or say they are landlocked. \* Portugal \* Ethiopia \* Taiwan \* Canada \* Russia \* USA (include Hawaii and Alaska) \* Denmark (include Greenland) \* Switzerland \* China

# **Coding Exercise**

Write a function called generate\_response that takes a prompt and model as input and returns the generated response using the openai.Completion.create method. The function should have the following signature:

import os

import openai

import secret

openai.api\_key=secret.api\_key

*# Define the generate\_response function as shown*

def generate\_response(prompt, model):

*#your code should be below this line*

response = openai.Completion.create(

model=model,

prompt=prompt,

top\_p=1,

temperature=0.1,

max\_tokens=50

)

return response.choices[0].text.strip()

# **Endpoints**

So far, we have interacted with the GPT-3 model by using the official [OpenAI playground](https://beta.openai.com/playground), a text file for our prompts, or an IDE with some Python code. The user interfaces for these methods of interaction are all different. However, all of these interactions are making use of API endpoints.

An endpoint for the OpenAI API is a uniform resource identifier (URI) that is used to access data or services provided by the company. Each endpoint represents a different way of interacting with OpenAI. Send information to a specific endpoint, and OpenAI sends back a response based on your initial query.

All of our examples up until now have used the Completions endpoint when generating a response to our prompts. This is one of several endpoints available to users. Here is the full list of available endpoints:

1. **List Models** - also known as the metadata endpoint; returns a list of models as well as some metadata about each model
2. **Retrieve Model** - returns detailed metadata about the specified model
3. **Completions** - most popular endpoint; returns a response to a prompt
4. **Semantic Search** - allows you to semantically rank documents with natural language
5. **Files** - upload and manipulate files in OpenAI storage
6. **Classification** - lets you classify a query without the need for finetuning or hyperparameter tuning
7. **Answers** - takes a question and returns an answer based on provided information (files or training examples)
8. **Embeddings** - returns an embedding based on information sent to the API

We will be exploring some of these endpoints throughout this assignment. OpenAI is actively working on new engines and updating existing ones. Over time, there might be some changes to the endpoints we see now.

# **Models**

## **Endpoint Signatures**

Each endpoint has a signature. This means either a GET or POST request followed by the URI. Below is an example of the Completions endpoint signature. It uses a POST method. Other signatures may require a parameter. These will be identified by curly brackets { }.

## **Completions Endpoint Signature**

POST https://api.openai.com/v1/completions

Endpoint signatures are provided in the examples below to give you a better idea of how the OpenAI is doing behind the scenes.

## **List Model Endpoint**

## **List Model Endpoint Signature**

GET https://api.openai.com/v1/models

This code prints out a long list of all of the available models and accompanying metadata. You can see why this is sometimes referred to as the metadata endpoint.

models = openai.Model.list()

print(models)

In a previous discussion, we talked about the four different types of models: davinci, curie, babbage, and ada. Each of these models varies in terms of capabilities, speed, and cost. However, the list of available models is greater than four. We will see later how we can use models like text-search-babbage-query-001.

Printing all of the models looks like it may be a JSON object or a dictionary. In reality, the return type from the API call is OpenAIObject. However, we can treat it like a dictionary. The code below prints out the first model and its metadata.

models = openai.Model.list()

print(models["data"][0])

## **Try this variation:**

* Create a list of all the model IDs without any other metadata.

models = openai.Model.list()

model\_ids = [model["id"] for model in models["data"]]

print(model\_ids)

## **Retrieve Model Endpoint**

## **List Engines Endpoint Signature**

GET https://api.openai.com/v1/models/{model}

Instead of parsing the list of models, you can retrieve metadata about a specific one if you know the id for the model. The code sample below prints out the metadata for the text-davinci-002 model.

model = openai.Model.retrieve("text-davinci-002")

print(model)

Code sample for the “metadata” endpoints

import os

import openai

import secret

openai.api\_key=secret.api\_key

*# show all models*

*#models = openai.Model.list()*

*#print(models)*

*# show specific model by array index*

*#models = openai.Model.list()*

*#print(models["data"][4])*

*# show list of all model ids*

*#models = openai.Model.list()*

*#model\_ids = [model["id"] for model in models["data"]]*

*#print(model\_ids)*

*# show number of models*

models = openai.Model.list()

print(len(models["data"]))

*# show model metadata by model id*

model = openai.Model.retrieve("dall-e-2")

print(model)

# **Embeddings**

## **Embedding Endpoint**

## **Embeddings Endpoint Signature**

POST https://api.openai.com/v1/embeddings

The API also has another experimental endpoint called embeddings. Embeddings are a representation of a given input as a vector of floating point numbers. Embeddings can be easily consumed by machine learning models and algorithms. They are often used to determine the semantic similarity between two texts. If two texts are similar, then their vector representations should also be similar.

Currently OpenAI offers three families of embedding models that allow for text search, text similarity and code search. Each family includes up to four models on a spectrum of capabilities:

* **Ada** (1024 dimensions),
* **Babbage** (2048 dimensions),
* **Curie** (4096 dimensions),
* **Davinci** (12288 dimensions).

These embedding models are specifically created to be good at a particular task. For example, text-similarity-ada-001 is good at capturing semantic similarity between two or more pieces of text. Or text-search-ada-doc-001 helps measure whether long documents are relevant to a short search query. For more information on embeddings, see the OpenAI [documentation](https://beta.openai.com/docs/guides/embeddings).

## **Creating an Embedding**

In this example, we are going to use the text-similarity-babbage-001 model. We also need to provide some text for the embedding. This is done with the input keyword argument. Both of these are required. The user keyword argument is optional. This unique identifier can be used to help monitor for abuse. The code sample below should print a long list of floating point numbers.

emb=openai.Embedding.create(

model="text-similarity-babbage-001",

input="You will rejoice to hear that no disaster has accompanied the")

print(emb)

Just like the list of models, the return object is of type OpenAIObject. If you want to access just the vector, change the print statement to the following:

print(emb["data"][0]["embedding"])

import os

import openai

import secret

openai.api\_key=secret.api\_key

*# WRITE YOUR CODE HERE*

emb=openai.Embedding.create(

model="text-embedding-ada-002", *# text-similarity-babbage-001",*

input="You will rejoice to hear that no disaster has accompanied the")

*# show the entire result*

print(emb)

*# show the similarity vector only*

print(emb["data"][0]["embedding"])

# **Edits**

### **Edits**

Given a prompt and an instruction, the model will return an edited version of the prompt.

POST https://api.openai.com/v1/edits

edi= openai.Edit.create(

model="text-davinci-edit-001",

input="What day was it? Wesdneday?",

instruction="Fix the spelling mistakes"

)

print(edi['choices'][0]['text'].strip())

using chatgpt: Fix the spelling mistakes in the following: What day was it? Wesdneday?

### **The Edit body**

Similarly to our create call our edit calls can take a couple of arguments to help the user better control what is generated.

The must-have arguments are model and instruction.

edi= openai.Edit.create(

model="text-davinci-edit-001",

instruction="Captitalize the first letter of all the words in the sentence"

)

print(edi['choices'][0]['text'].strip())

For our purposes we are also going to make input as a default because it keeps it clean about what is the input we are interacting with.

input : is optional but defaults to an empty string when no argument is specified. It is used as a starting point for the edit.  
instruction: is required and tell the AI how to edit the prompt

edi= openai.Edit.create(

model="text-davinci-edit-001",

input="What day was it? Wesdneday?",

instruction="change wednesday to saturday."

)

print(edi['choices'][0]['text'].strip())

Feel free to try it more than once, there might be a chance the AI misinterprets the directions.  
Similarly, to create a call there are a couple of other optional arguments. In this case, we could have added n, temperature and top\_p.

n: defaults to 1 and tells how many edits to create from input and instruction  
temperature: defaults to 1 , higher values mean the model will take more risks.  
top\_p: 0.1 means only the tokens comprising the top 10% probability mass are considered.

A Try It button is provided in case you wanted to try adding some of those variables.

# **Coding Exercise**

# ***Coding Exercise***

Using the OpenAI Edit API, create a Python function fix\_spelling that takes a string as input and returns a corrected version of the input with spelling mistakes fixed. The function should use the text-davinci-edit-001 model. For a more effective output make top\_p=0.1 as one of your arguments.

import openai

import secret

openai.api\_key=secret.api\_key

def fix\_spelling(text: str) -> str:

*#please putyour code below this line*

edi= openai.Edit.create(

model="text-davinci-edit-001",

top\_p=0.1,

input=text,

instruction="fix the spelling mistakes"

)

*#print(edi['choices'][0]['text'].strip())*

return edi['choices'][0]['text'].strip()

*# WRITE YOUR CODE HERE*

*# FREEZE CODE BEGIN*

input\_texts = [

"I hve a bananna for brekfast.",

"The colur of the sky is rde.",

"The grass is greneer on the other side."

]

for input\_text in input\_texts:

output\_text = fix\_spelling(input\_text)

print(f"Input: {input\_text}\nOutput: {output\_text}\n")

*# FREEZE CODE END*

Fix the spelling mistakes in the following sentences:

1: I hve a bananna for brekfast.

2: The colur of the sky is rde.

3: The grass is greneer on the other side.

# **Customizing GPT-3**

One of the key endpoints that allows you to get the most of your GPT-3 model is the fine-tuning endpoint. By definition fine-tuning means adjusting precisely so as to bring to the highest level of performance or effectiveness. In other words, to adjust (something) so that it works perfectly. The fine-tuning endpoint allows you to customize the model to your particular use case.

### **Few-shot Learning**

OpenAI pre-trained GPT-3 with a prepared data set in a semi-supervised fashion. If given a prompt and a few examples it can most likely guess what task you are trying to perform and generate a completion based on that. This process is called **Few-shot Learning**.

With fine-tuning one is able to customize a model for your application. We can use our own data and create a model custom made for our different projects. Customizing makes GPT-3 more efficient and faster. Matter of fact, it even allows us to use cheaper models more efficiently.

### **One-shot Learning**

One way to help with understand few-shot is learning about One-shot learning. **One-shot learning** is a concept in machine learning where a learning algorithm is capable of understanding information about an object, task, or event from a single instance or experience. The term “one-shot” comes from the idea that the machine only gets one “shot” or opportunity to learn about a concept. It’s quite a challenging problem for machine learning, since typically, these algorithms depend on large quantities of data to accurately learn and generalize.

Humans are particularly good at one-shot learning. For example, if a child is shown a picture of a type of animal they’ve never seen before, say a pangolin, they can usually recognize it after seeing it just once. If the child is later shown a bunch of pictures of different animals, they can usually pick out the pangolin from the lineup. AI researchers hope to achieve this level of understanding and flexibility with machine learning algorithms.

# **Fine-Tuning**

Fine-Tuning lets you get more out of the models available through the API by providing:

1. Higher quality results than prompt design
2. Ability to train on more examples than can fit in a prompt
3. Token savings due to shorter prompts
4. Lower latency requests

Fine-tuning is all about changing the model so it can generate the responses you want every time. The capabilities and knowledge of the model will be fully focused on the dataset used for fine-tuning.

Fine-tuning improves on few-shot learning by training on many more examples than can fit in the prompt, letting you achieve better results on a wide number of tasks. Once a model has been fine-tuned, you won’t need to provide examples in the prompt anymore. For example, this is how the dataset **json** file would look like.

{"prompt": "<prompt text>", "completion": "<ideal generated text>"}

{"prompt": "<prompt text>", "completion": "<ideal generated text>"}

{"prompt": "<prompt text>", "completion": "<ideal generated text>"}

JSONL, stands for JSON Lines. It is a format that represents structured data as a sequence of JSON objects, each on a separate line. Instead of a single JSON object enclosed within curly braces, each line in a JSONL file represents a separate, self-contained JSON object. This makes JSONL a format suitable for streaming large amounts of data or processing data line by line.

Before we get started, it is recommended to install the OpenAI command-line interface. To do this we are going to run the following command in the terminal (left panel).

pip install --upgrade openai

Fine-tuning is a powerful technique to create a new model that’s specific to your use case. Before fine-tuning your model, we strongly recommend reading these best practices and [specific guidelines](https://beta.openai.com/docs/guides/fine-tuning/specific-guidelines) for your use case.

<https://beta.openai.com/docs/guides/fine-tuning/specific-guidelines>

# **Program Using OpenAI**

GPT-3 API is a powerful set of tools for natural language processing that can help developers create innovative applications. Now that we are more or less familiar with GPT-3’s API we are going to create our first program using the API. This will serve two goals:

1. It will help to get some practice using Python and the OpenAI API
2. Practice our prompt generation

## **Moflix**

We are going to create a movie recommendation program powered by OpenAI’s GPT-3. The program will be named Moflix. Your Python file will be located in the top left , this is where you will put in your code. Lastly, you will be provided with the standard box(located in the bottom left) that we can use as our GPT playground.

For starters, let’s try generating what we want. Use the playground to ask for the following prompt and then run our Try it! button.  
Give me 5 movie recommendations

## **UX**

I know that typing the prompt into our playground is an easy solution, but why don’t we take it a step further and create a program specifically designed for movie recommendations? This program will be tailored to give users a smoother movie searching experience, requiring less typing than GPT-3. We’ll also focus on UX design to give our users an even better experience. This program will make it easier to find the perfect movie recommendation without having to put in the extra effort on our user.

**UX** stands for User Experience and refers to the overall experience a user has when interacting with a product or service. It encompasses aspects such as the design, usability, and functionality of an interface, and takes into account the user’s needs, motivations, and expectations. A good UX should be enjoyable, intuitive, and provide a positive user experience.

# **Part 1. Box Creation**

We are going to create a Python function that asks the user for some information. We will use the information given by the user to create our program.

We will be interacting with user input we are going to use the terminal to interact with our program. The Try It button below will run your code inside the terminal .

First, let’s create a function that takes a list of movies and put it in a nice box. As we don’t have our movie list yet, we are going to create a sample list. Copy and paste the following inside our moflix.py

sample=["One life","Two","Potato Pie and Life"]

print(sample)

Codio Terminal help

Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 6.2.0-1017-aws x86\_64)

\* Documentation: https://help.ubuntu.com

\* Management: https://landscape.canonical.com

\* Support: https://ubuntu.com/advantage

\* Canonical Livepatch is available for installation.

- Reduce system reboots and improve kernel security. Activate at:

https://ubuntu.com/livepatch

\*

\* Welcome to the Codio Terminal!

\*

\* https://docs.codio.com/develop/develop/ide/boxes/overview

\*

\* Your Codio Box domain is: pogoigor-dolbycanal.codio.io

\*

Last login: Fri Sep 30 15:04:13 2022 from 192.168.10.226

codio@pogoigor-dolbycanal:~/workspace$

We are going to create a function that takes an argument which is our movie list. That function should return the movie list in a nice box.  
+---------------------+

| One Life |

| Two |

| Potato Pie and Life |

+---------------------+

Our box should change based on the number of movies and the length of the items that need to fit in the box. We are going to create our function called InBox that first tries to get the length of the biggest element in our list.

def InBox(x):

#this gives you the longest string

biggest = max(x, key = len)

#THIS gets you the length of the biggest string

biggest=len(biggest)

Nothing should generate, this is a test that we don’t have any errors so far.

Now for the top of our box we are going to start and end with + . Then for the content in the middle we will fill it with -. This technique will be used for both our top and bottom.

print("+" + "-" \* (biggest + 2) + "+")

To try it, use the sample provided above sample=["One life","Two","Potato Pie and Life"] and call the function: InBox(sample) .

We want to focus on the middle part now. We will use a for loop to iterate through all our items. We’ll start off with a | and then add our movie title. We’ll then add additional whitespaces " " to match the length of the longest element.

for i in x:

print("| " + i + (" "\*(biggest-len(i))) + " |")

Then we will go outside our for loop and add the same code we used to create the top layer, in order to create our bottom layer.

In order to make our design look nicer, let’s try making our box look different.

sample=["One life","Two","Potato Pie and Life"]

print(sample)

def InBox(x):

*#this gives you the longest string*

biggest = max(x, key = len)

*#THIS gets you the length of the biggest string*

biggest=len(biggest)

print("+" + "-" \* (biggest + 2) + "+")

for i in x:

print("| " + i + (" "\*(biggest-len(i))) + " |")

print("+" + "-" \* (biggest + 2) + "+")

InBox(sample)

# **Part 2. Prompt Creation**

Now that we have our box we need to be able to fill it with a list of movies. The next step we are going to take is creating our prompt. We are going to create a function called Moreflix that will prompt the user for:

1. The number of recommendations they want
2. The genre they want
3. A similar movie they have in mind.

We write the code below to take care of that. The function will prompt users and return a prompt for us to feed to our OpenAI. We want the user to be able to type 0 if they don’t have an answer in mind.

def Moreflix():

# ask the user for different inputs

print("Enter 0, if you don't have an answer in mind")

number\_recs=int(input('How many movies do you want recommended: '))

genre=input('What genre are you looking for: ')

similar= input('What is a similar movie: ')

To try it add the print statement print(Moreflix()) and press the button below

From there we have all the tools needed for us to write our prompt. First of all if they add 0 to the number of recs we will default it to 5 for them.

if number\_recs<=0:

number\_recs=5

For the others if 0 is included we will generate a prompt simply asking for the number of recommendation. But first we are about to create our first prompt. A quick reminder,here are three basic guidelines to creating better results:

* **Show and tell**
* **Provide quality data**
* **Check your settings**

We are going to focus on show and tell on 3 things:

1. Giving the model clear instructions (tell)
2. Giving the model an example (show)
3. Giving the model clear instructions and an example (show and tell)

In order for the prompt to generate more clear answers, try the following prompt in the playground in the bottom left next to your terminal. In a python array form, give me 3 movie recommendation

Now that we have a sample prompt and have an idea of what the response would look like, we can start coding. We are going to take care of edge cases where the users do not provide all the information. Base on the information that the user provides we will, create different prompts. For example here is an example where the user do not provide a genre or a similar movie. Copy and paste the code to the moflix file on your left.

if genre=="0" and similar=="0":

return("In a python array form, give me " + str(number\_recs) +" movie recommendation")

Now we can take care of the additional scenarios

if genre=="0" :

return ("In a python array form, give me " + str(number\_recs) +" movie recommendation, similar to " + similar)

if similar=="0":

return ("In a python array form, give me "+genre + " "+ str(number\_recs) +" movie recommendations")

return ("In a python array form, give me "+genre +" "+str(number\_recs)+" movie recommendation similar to " + similar)

Now run the following to make sure all the work is going well. To make our code cleaner, feel free to remove the print statements when done with it.  
print(Moreflix())

To increase our accuracy, of getting the AI to do what we want after our prompt we will start it off with providing it space to do what we want  
Prompt=Moreflix() + "rec="

For chatGpT In a python array form, give me 3 sci-fi movie recommendations. Don't output anything else except the python array.

MOVIe RECommendation code, output box and prompt generation

*# Output a bunch of lines of text in a nice ASCII box*

def InBox(x):

*#this gives you the longest string*

biggest = max(x, key = len)

*#THIS gets you the length of the biggest string*

biggest=len(biggest)

print("+" + "-" \* (biggest + 2) + "+")

for i in x:

print("| " + i + (" "\*(biggest-len(i))) + " |")

print("+" + "-" \* (biggest + 2) + "+")

*# Construct a GPT prompt to get a bynber if movie recommentations by asking the user for genre, era and similarity.*

def Moreflix():

*# ask the user for different inputs*

print("Enter 0, if you don't have an answer in mind")

number\_recs=int("0" + input('How many movies do you want recommended: '))

if number\_recs<=0:

number\_recs=5

print("Will actually get " + str(number\_recs))

genre=input('What genre are you looking for: ')

era=input("What era are you looking for: (classical, recent, 80's etc...) ")

similar= input('What is a similar movie: ')

if genre!="":

genre=genre.strip() + " "

if era!="":

era=era.strip() + " "

if similar!="":

similar=", similar to " + similar.strip()

return ("In a python array form, give me "+ str(number\_recs) + " " + era + genre + "movie recommendation" + similar)

sample=["One life","Two","Potato Pie and Life"]

*#InBox(sample)*

prompt=Moreflix() + " rec="

print(prompt)

# **Part 3. Presenting Info**

Now that we have our prompt let’s take care of integrating our OpenAI API key as we have done before . Ideally you will move to the top of our file.

import os import openai

import secret

openai.api\_key=secret.api\_key

Let’s create a quick function that takes a prompt and returns our movie results . This function should go between our InBox and Moreflix functions. We are going to call this function Res short for response. This function should work as all the previous prompts we generated using our API.

def Res(x):

response = openai.Completion.create(

model="text-davinci-002",

prompt=x,

top\_p=1,

max\_tokens=100

)

return(response['choices'][0]['text'].strip())

After checking that everything works as intended we can change our print to a value assignment. Set the Res function call to the variable movies.  
movies=Res(Prompts).

We are at the finish line. We know that the response generated by GPT-3 was a string. Even though, it was a string we want to convert it to an actual list. After that we can call InBox from our list.

# Converting string to list

import ast

movie\_list=ast.literal\_eval(movies)

InBox(movie\_list)

One last piece we are going to take care of are edge cases where the completions contains a numerated list or additional characters:

import ast

Prompts=Moreflix() + "rec ="

movies=(Res(Prompts))

try:

movie\_list=ast.literal\_eval(movies)

except:

movie\_list2=""

for i in movies:

if i=="]":

movie\_list2+=(i)

break

else:

movie\_list2+=(i)

movie\_list=ast.literal\_eval(movie\_list2)

InBox(movie\_list)

Your program should be able to generate something like the image below.

codio@pogoigor-dolbycanal:~/workspace$ python3 moflix.py

Just press <Enter>, if you don't have an answer in mind

How many movies do you want recommended:

Will actually get 5

What genre are you looking for:

What era are you looking for: (classical, recent, 80's, ...)

What is a similar movie:

+-------------------------------+

| Footloose (1984) |

+-------------------------------+

| Purple Rain (1984) |

+-------------------------------+

| Flashdance (1983) |

+-------------------------------+

| Dirty Dancing (1987) |

+-------------------------------+

| Little Shop of Horrors (1986) |

+-------------------------------+

| The Blues Brothers (1980) |

+-------------------------------+

| Top Gun (1986) |

+-------------------------------+

| Streets of Fire (1984) |

+-------------------------------+

codio@pogoigor-dolbycanal:~/workspace$

My completed program with addition of era input

import os

import openai

import secret

openai.api\_key=secret.api\_key

*# Output a bunch of lines of text in a nice ASCII box*

def InBox(x):

*#this gives you the longest string*

biggest = max(x, key = len)

*#THIS gets you the length of the biggest string*

biggest=len(biggest)

print("+" + "-" \* (biggest + 2) + "+")

for i in x:

print("| " + i + (" "\*(biggest-len(i))) + " |")

print("+" + "-" \* (biggest + 2) + "+")

*# Look up movie recommendation given a query prompt*

*# hard code the results since the API call fails...*

def Res(x):

return '["Footloose (1984)", "Purple Rain (1984)", "Flashdance (1983)", "Dirty Dancing (1987)", "Little Shop of Horrors (1986)", "The Blues Brothers (1980)", "Top Gun (1986)", "Streets of Fire (1984)"]'

*# response = openai.Completion.create(*

*# model="text-davinci-002",*

*# prompt=x,*

*# top\_p=1,*

*# max\_tokens=100)*

*# return(response['choices'][0]['text'].strip())*

*# Construct a GPT prompt to get a bynber if movie recommentations by asking the user for genre, era and similarity.*

def Moreflix():

*# ask the user for different inputs*

print("Just press <Enter>, if you don't have an answer in mind")

number\_recs=int("0" + input('How many movies do you want recommended: '))

if number\_recs<=0:

number\_recs=5

print("Will actually get " + str(number\_recs))

genre=input('What genre are you looking for: ')

era=input("What era are you looking for: (classical, recent, 80's, ...) ")

similar= input('What is a similar movie: ')

if genre!="":

genre=genre.strip() + " "

if era!="":

era=era.strip() + " "

if similar!="":

similar=", similar to " + similar.strip()

return ("In a python array form, give me "+ str(number\_recs) + " " + era + genre + "movie recommendation" + similar)

sample=["One life","Two","Potato Pie and Life"]

*#InBox(sample)*

Prompts=Moreflix() + " rec="

*#print(Prompts)*

*# Converting string to list*

import ast

movies=Res(Prompts)

*#print(movies)*

try:

movie\_list=ast.literal\_eval(movies)

except:

movie\_list2=""

for i in movies:

if i=="]":

movie\_list2+=(i)

break

else:

movie\_list2+=(i)

movie\_list=ast.literal\_eval(movie\_list2)

InBox(movie\_list)

# Codio: Creating Playground

# **Chat GPT-3**

Generally we can :

1. Use GPT-3 to generate natural language text for a given topic. This can be used to quickly generate blog posts, essays, and other forms of copy.
2. Utilize GPT-3 to generate product descriptions, titles and other marketing materials.
3. Use GPT-3 to automate customer support tasks such as FAQs and customer service emails.
4. Leverage GPT-3 to generate technical documentation and software manuals.
5. Automate the process of summarizing long articles with GPT-3.
6. Generate personalized emails and messages with GPT-3.
7. Create interactive simulations, tutorials, and interactive demonstrations with GPT-3.
8. Use GPT-3 to generate lyrics for songs and poems.
9. Create engaging and interesting conversations with GPT-3.
10. Automate the process of creating resumes, cover letters, and other job application documents.

# **Sample Playground**

We have used the playground to classify. For example try the following.  
GPT-3 can classify items into categories. Use the file on the left to write your prompts. Try the prompt below:

classify the following : cat, dog , car , plane

We are going to create our own playground to try and run the following 5 tasks.

1. Use GPT-3 to generate natural language text for a given topic. This can be used to quickly generate blog posts, essays, and other forms of copy.
2. Utilize GPT-3 to generate product descriptions, titles and other marketing materials.
3. Use GPT-3 to automate customer support tasks such as FAQs and customer service emails.
4. Leverage GPT-3 to generate technical documentation and software manuals.
5. Automate the process of summarizing long articles with GPT-3.

We will get to all of that but first we want to create a text file that is able to do what the playground on our left can do. Again on the idea of creating a more user-friendly program.

Let’s ask the box to:

generate a blog post about a pizza shop

Our goal in this lesson is to recreate the box. The user should be able to write whatever they want and that should be passed on as the prompt. After that the generated answer will also be placed in your file.

# **Response**

On the top left we have our Python file. On the bottom left we have the empty text file where the user will be able to simply write their prompts and generate and get their responses from.

For starters, we are going to use the same structure we have been using before to generate our responses. First, let’s get our libraries.

import os import openai

import secret

openai.api\_key=secret.api\_key

We need to read our file and just put the contents of it as our prompt.

#open the file

file = open("playground.txt", "r")

#read each line

filelines = file.readlines()

#put all the lines in a single variable

all\_lines = ""

for line in filelines:

all\_lines += line

#close the file

file.close()

print(all\_lines)

It is customary to close the file when done with it. To test our code, we will try a couple of things.  
On the top left in our Python file try running the following code.

print("hello world")

Now instead of printing hello world we are going to print the all\_lines, to try and see the content of the playground.txt file.

print(all\_lines)

Since our playground has no content in it yet we are going to get a blank response. We should have the Codio IDE message that our code successfully ran.

Now let’s try adding the following text on our .txt file (bottom left), and try running our file again since now our playground will not be empty.

write a tagline for an ice cream shop.

It should print out write a tagline for an ice cream shop.. Now that we know we have access to the contents of our file let’s assign it as the prompt instead of printing it.

prompts = all\_lines

We typically employ the following code to prompt users and capture the AI response:

response = openai.Completion.create(

model="text-davinci-002",

prompt=prompts,

max\_tokens=256,

top\_p=0.1)

txt\_response=response['choices'][0]['text'].strip()

print(txt\_response)

Now that we can get it to generate a response on the content of our text file we are going to have it write down the response in that same text file. For that we need to first open our file then simply add the response in to the text file instead of printing it.

*#open our file, and append to it*

f = open("playground.txt", "a")

*#write the response we want to append*

f.write((txt\_response))

*#close our file*

f.close()

From that we realize it generates our response on the same line. Delete the text in the playground file. Have it go back to simply:

Write a tagline for an ice cream shop

Before we write our response we are going to make sure it adds to a new line so we will change our code to the following:

#write the response we want to append f.write(("\n")) f.write((txt\_response))

Feel free to try it with different prompts. For example:

Give me 5 movie recommendations.

Let’s just say it’s a pain to keep erasing. Let’s add some code that will clear our text file for us. We will use the key word //clear// in our playground to know to clear it instead of generating anything. In other words, if in a our file we see the //clear// keyword our program instead of generating anything will give us a blank page. Copy and paste the following in our code above the code we have already written. For the else case, we are going to make the code we have already written the contents for it.

# if the code word is present we clear the contents

if '//clear//' in open('playground.txt').read():

open("playground.txt", "w").close()

else:

Opening a file in “write” mode clears it. Again we have to close it after we are done with it. Inside the else statement we can indent the rest of the code we have previously written

Example code, with try/catch since the api call does not actually work

import os

import openai

import secret

openai.api\_key=secret.api\_key

*#open the file*

file = open("playground.txt", "r")

*#read each line*

filelines = file.readlines()

*#put all the lines in a single variable*

all\_lines = ""

for line in filelines:

all\_lines += line

*#close the file*

file.close()

*#print(all\_lines)*

prompts = all\_lines

try:

response = openai.Completion.create(

model="text-davinci-002",

prompt=prompts,

max\_tokens=256,

top\_p=0.1)

except openai.error.InvalidRequestError:

txt\_response="Scooping Joy, One Flavor at a Time: Indulge in the Sweet Symphony of Happiness at [Shop Name]!"

else:

txt\_response=response['choices'][0]['text'].strip()

*#finally:*

*# print(txt\_response)*

*# if the code word is present we clear the contents*

if '//clear//' in open('playground.txt').read():

open("playground.txt", "w").close()

else:

*#open our file, and append to it*

f = open("playground.txt", "a")

*#write the response we want to append*

f.write(("\n"))

f.write((txt\_response))

*#close our file*

f.close()

# **Using Our Box**

Let’s use our new playground to generate a couple more prompts. Feel free to modify at your own leisure. But practice running the following prompts

1. Generate blog posts, essays, and other forms of copy.

write a short essay about willy wonka

If the text generated is too long and you have to scroll to read everything you can click on the View tab.

2. Utilize GPT-3 to generate product descriptions, titles and other marketing materials.

Write a description for Codio, the best learning website.

3. Use GPT-3 to automate customer support tasks such as FAQs and customer service emails.

Clear the previous text in the playground. Feel free to manually do it, or type//clear// in your playground. Before asking it to:

generate a customer service email to apologize for our website being down. Let customers know that the issue was fixed.

4. Leverage GPT-3 to generate technical documentation and software manuals.

generate a short technical document on how to use gpt-2.

5. Automate the process of summarizing long articles with GPT-3.

summarise the content of this webpage (<https://arxiv.org/abs/2005.14165>).

# **Coding Exercise**

For our coding exercise we are going to create an extra feature similar to //clear//.

You can implement a feature where certain keywords trigger GPT-3 to generate specific types of content. For example,//joke// could make the program generate a random joke, //quote// could trigger a famous quote generation, and //news// could lead to a summary of the day’s top news.

For our assignment modify the code for your program so that if the user types in //joke// a joke will be outputted. Have your code so that //clear// has a priority than joke. Meaning if both keywords are present it will just clear the page.

**Be sure** to click the Try It to run your box.

import os

import openai

import secret

openai.api\_key=secret.api\_key

model = "gpt-3.5-turbo-instruct"

playfile = "playground.txt"

GET\_JOKE="generate a joke"

JOKE="Why don't scientists trust atoms?\n\nBecause they make up everything!"

DEFAULT="Scooping Joy, One Flavor at a Time: Indulge in the Sweet Symphony of Happiness at [Shop Name]!"

def emptyFile(name):

open(name, "w").close()

def readFile(name):

file = open(name, "r")

filelines = file.readlines()

all\_lines = ""

for line in filelines:

all\_lines += line

file.close()

return all\_lines

def writeFile(name, content):

f = open(name, "a")

f.write(("\n"))

f.write((content))

f.close()

def checkFile(name, flag):

return flag in open(name).read()

def getResponse(prompts, default = DEFAULT):

failed=True

try:

response = openai.Completion.create(

model=model,

prompt=prompts,

max\_tokens=256,

top\_p=0.1)

except: *# openai.error.InvalidRequestError:*

*# openai.error.RateLimitError:*

failed=True

else:

failed=False

txt\_response=response['choices'][0]['text'].strip()

finally:

if failed:

txt\_response=default.strip()

*# except openai.error.InvalidRequestError:*

*# txt\_response="Scooping Joy, One Flavor at a Time: Indulge in the Sweet Symphony of Happiness at [Shop Name]!"*

*#else:*

*#finally:*

*#print(txt\_response)*

return txt\_response

*# if the code word is present we clear the contents*

if checkFile(playfile, '//clear//'):

emptyFile(playfile)

else:

if checkFile(playfile, '//joke//'):

emptyFile(playfile)

prompts = GET\_JOKE

response = getResponse(prompts, JOKE)

writeFile(playfile, response)

else:

prompts = readFile(playfile)

response = getResponse(prompts)

writeFile(playfile, response)

suggest improvements to this python code snippet:

model=”something”

DEFAUL=”something...”

TRUE=1==1

FALSE=1==0

def getResponse(prompts, default = DEFAULT):

failed=TRUE

try:

response = openai.Completion.create(

model=model,

prompt=prompts,

max\_tokens=256,

top\_p=0.1)

except: *# openai.error.InvalidRequestError:*

failed=TRUE

else:

failed=FALSE

txt\_response=response['choices'][0]['text'].strip()

finally:

if failed:

txt\_response=default.strip()

return txt\_response

<https://codio.com/home/student?course_id=13f9e74a783e8453efe870ef22de3ac7>

import openai

MODEL = "something"

DEFAULT = "something..."

MAX\_TOKENS = 256

TOP\_P = 0.1

def get\_openai\_response(prompts, default=DEFAULT):

"""

Get response from OpenAI based on prompts.

Args:

prompts (str): Input prompts for OpenAI.

default (str): Default response if OpenAI request fails.

Returns:

str: OpenAI response or default response.

"""

failed = True

try:

response = openai.Completion.create(

model=MODEL,

prompt=prompts,

max\_tokens=MAX\_TOKENS,

top\_p=TOP\_P

)

except # openai.error.InvalidRequestError:

failed = True

else:

failed = False

txt\_response = response['choices'][0]['text'].strip()

finally:

if failed:

txt\_response = default.strip()

return txt\_response

Lab1: The Simple Bot

# **Introduction to Chatbots**

One of the most common tasks and a subset of Natural Language Processing is **Natural Language Generation** (NLG).

* It is also sometimes described as the opposite of speech recognition.
* It’s the task of generating human language text responses by putting together structured information.
* This information follows **syntactical** and **semantic** rules from a program’s knowledge base.

The best examples of real-world applications of this concept are **virtual agents** and **chatbots**.

* Apple’s Siri and Amazon’s Alexa are examples of virtual agents.
* They use:
  + **Speech recognition** to understand user commands
  + **NLG** to respond with the required information or conduct the requested action.

A **Chatbot**, on the other hand, operates on **typed text**.

* It processes and simulates **human dialogs** and **conversations**, allowing users to interact with various kinds of digital services.
* They can be as simple as a program comprised of a set of **hard-coded rules**, to very sophisticated **AI-based** chatbots that deliver **personalized responses** and learn as they go.

Connected Codio to github account with SSH keys then forked repo

# **https://github.com/codio-content/nlp-simplebot**

# [**https://github.com/bcowgill/nlp-simplebot**](https://github.com/bcowgill/nlp-simplebot)

**git clone git@github.com:bcowgill/nlp-simplebot.git**

# **Initializing the GUI**

### **A simple, rule-based chatbot.**

Let’s start by building a GUI for our chatbot. We’ll use **tkinter**, the standard Python GUI library.

The first step is to import all the necessary modules into our file.  
Paste the following code in the top left pane:

import tkinter.scrolledtext as tks #creates a scrollable text window

from datetime import datetime

from tkinter import \*

To make a simple GUI for our chatbot, it should have 4 main components:

1. baseWindow - the main GUI window that contains everything
2. chatWindow - displays the conversation between a user and the chatbot
3. userEntryBox - for the user to type their queries for the Chatbot
4. sendButton - a button that sends the user query to the Chatbot

Let’s use tkinter to initialize the baseWindow and the other components and place them on the baseWindow.

Paste the following code in the top left pane under the import statements:

*# Create the main application window using Tk()*

baseWindow = Tk()

*# Set the title of the window*

baseWindow.title("The Simple Bot")

*# Set the size of the window*

baseWindow.geometry("500x250")

*# Create the chat window as a ScrolledText widget with "Arial" font*

chatWindow = tks.ScrolledText(baseWindow, font="Arial")

*# Configure tags for message alignment: 'tag-left' for bot messages, 'tag-right' for user messages*

chatWindow.tag\_configure('tag-left', justify='left')

chatWindow.tag\_configure('tag-right', justify='right')

*# Disable the chat window initially (it should not be editable by the user)*

chatWindow.config(state=DISABLED)

*# Create the send button, with specific font, text, and background color*

*# The 'command' option is commented out. Uncomment it and replace 'send' with your send function's name*

sendButton = Button(

baseWindow,

font=("Verdana", 12, 'bold'),

text="Send",

bg="#fd94b4",

activebackground="#ff467e",

fg='#ffffff',

*# command=send*

)

*# Create the user entry box where the user types their messages*

*# bd= border depth. 1Px just a border, bigger numbers give deeper 3d effect*

userEntryBox = Text(baseWindow, bd=1, bg="white", width=38, font="Arial")

*# Place the chat window, user entry box, and send button on the main window using specific coordinates and sizes*

chatWindow.place(x=1, y=1, height=200, width=500)

userEntryBox.place(x=3, y=202, height=27)

sendButton.place(x=430, y=200)

*# Start the main event loop to keep the application running and responsive*

baseWindow.mainloop()

# **Implementing functions for the GUI**

At this point, the sendButton doesn’t do anything.  
Let’s define a function called send which should do the following:

* Collect the user\_input from the textBox
* Get the bot\_response
* Insert both of these into the chatWindow

Paste the following code in the top left pane, right under the import statements:

def send(event):

chatWindow.config(state=NORMAL)

user\_input = userEntryBox.get("1.0",'end-2c')

user\_input\_lc = user\_input.lower()

bot\_response = get\_bot\_response(user\_input\_lc)

create\_and\_insert\_user\_frame(user\_input) create\_and\_insert\_bot\_frame(bot\_response)

chatWindow.config(state=DISABLED)

userEntryBox.delete("1.0","end")

chatWindow.see('end')

Once we have the user\_input and bot\_response, we’ll write a couple of functions to insert them into the chatWindow

Paste the following code in the top left pane, right above the send function:

def create\_and\_insert\_user\_frame(user\_input):

userFrame = Frame(chatWindow, bg="#d0ffff")

Label(

userFrame,

text=user\_input,

font=("Arial", 11),

bg="#d0ffff"

).grid(row=0, column=0, sticky="w", padx=5, pady=5)

Label(

userFrame,

text=datetime.now().strftime("%H:%M"),

font=("Arial", 7),

bg="#d0ffff"

).grid(row=1, column=0, sticky="w")

chatWindow.insert('end', '\n ', 'tag-right') chatWindow.window\_create('end', window=userFrame)

def create\_and\_insert\_bot\_frame(bot\_response):

botFrame = Frame(chatWindow, bg="#ffffd0")

Label(

botFrame,

text=bot\_response,

font=("Arial", 11),

bg="#ffffd0",

wraplength=400,

justify='left'

).grid(row=0, column=0, sticky="w", padx=5, pady=5)

Label(

botFrame,

text=datetime.now().strftime("%H:%M"),

font=("Arial", 7),

bg="#ffffd0"

).grid(row=1, column=0, sticky="w")

chatWindow.insert('end', '\n ', 'tag-left') chatWindow.window\_create('end', window=botFrame) chatWindow.insert(END, "\n\n" + "")

Now, we want to bind the send function to the sendButton. We can also bind the Enter key to the window so we don’t have to click on the button every time.

Replace the sendButton initialization in the top left pane with the following code:

# **The bot response function**

# ***The bot response function***

The bot\_response function is the logic function for the chatbot. This is where we will define the rules that the chatbot will follow to respond to user\_input

Paste the following code in the top left pane, right below the import statements:

# Generating response

def get\_bot\_response(user\_input):

bot\_response = ""

if(user\_input == "hello"):

bot\_response = "Hi!"

elif(user\_input == "hi" or user\_input == "hii" or user\_input == "hiiii"):

bot\_response = "Hello there! How are you?"

elif(user\_input == "how are you"):

bot\_response = "Oh, I'm great! How about you?"

elif(user\_input == "fine" or user\_input == "i am good" or user\_input == "i am doing good"):

bot\_response = "That's excellent! How can I help you today?"

else:

bot\_response = "I'm sorry, I don't understand..."

return bot\_response

This was the last step! We have successfully built our first chatbot! Let’s run it and interact with it!

Full code

import tkinter.scrolledtext as tks #creates a scrollable text window

from datetime import datetime

from tkinter import \*

# Generating response

def get\_bot\_response(user\_input):

return bot\_response

def create\_and\_insert\_user\_frame(user\_input):

userFrame = Frame(chatWindow, bg="#d0ffff")

Label(

userFrame,

text=user\_input,

font=("Arial", 11),

bg="#d0ffff").grid(row=0, column=0, sticky="w", padx=5, pady=5)

Label(

userFrame,

text=datetime.now().strftime("%H:%M"),

font=("Arial", 7),

bg="#d0ffff"

).grid(row=1, column=0, sticky="w")

chatWindow.insert('end', '\n ', 'tag-right')

chatWindow.window\_create('end', window=userFrame)

def create\_and\_insert\_bot\_frame(bot\_response):

botFrame = Frame(chatWindow, bg="#ffffd0")

Label(

botFrame,

text=bot\_response,

font=("Arial", 11),

bg="#ffffd0",

wraplength=400,

justify='left'

).grid(row=0, column=0, sticky="w", padx=5, pady=5)

Label(

botFrame,

text=datetime.now().strftime("%H:%M"),

font=("Arial", 7),

bg="#ffffd0"

).grid(row=1, column=0, sticky="w")

chatWindow.insert('end', '\n ', 'tag-left')

chatWindow.window\_create('end', window=botFrame)

chatWindow.insert(END, "\n\n" + "")

def send(event):

chatWindow.config(state=NORMAL)

user\_input = userEntryBox.get("1.0",'end-2c')

user\_input\_lc = user\_input.lower()

bot\_response = get\_bot\_response(user\_input\_lc)

create\_and\_insert\_user\_frame(user\_input)

create\_and\_insert\_bot\_frame(bot\_response)

chatWindow.config(state=DISABLED)

userEntryBox.delete("1.0","end")

chatWindow.see('end')

baseWindow = Tk()

baseWindow.title("The Simple Bot")

baseWindow.geometry("500x300")

chatWindow = tks.ScrolledText(baseWindow, font="Arial")

chatWindow.tag\_configure('tag-left', justify='left')

chatWindow.tag\_configure('tag-right', justify='right')

chatWindow.config(state=DISABLED)

sendButton = Button(

baseWindow,

font=("Verdana", 12, 'bold'),

text="Send",

bg="#fd94b4",

activebackground="#ff467e",

fg='#ffffff',

command=send)

sendButton.bind("<Button-1>", send)

baseWindow.bind('<Return>', send)

userEntryBox = Text(baseWindow, bd=1, bg="white", width=38, font="Arial")

chatWindow.place(x=1, y=1, height=270, width=500)

userEntryBox.place(x=3, y=272, height=27)

sendButton.place(x=430, y=270)

baseWindow.mainloop()

# **The bot response function using GPT-3**

At this point, you must be familiar with OpenAI GPT-3 Toolkit. We are going to interact with our prompt to create a response. How we are going to interact with it is by changing our get\_bot\_response function.

Now we are going to create a function get\_bot\_response.The function get\_bot\_response(user\_input) generates a bot response by sending a user’s input to the OpenAI GPT-3 model, "text-davinci-002", and returns the model’s generated text as the bot’s response, stripped of leading and trailing whitespace.

# Generating response

def get\_bot\_response(user\_input):

prompt = f"Please provide a response to the following user input: '{user\_input}'"

response = openai.Completion.create(

model="text-davinci-002",

prompt=prompt,

max\_tokens=150,

n=1,

stop=None,

temperature=0.5,

)

bot\_response = response.choices[0].text.strip()

return bot\_response

After making sure our code runs without errors we are ready to put it in our main code. Please make sure to copy everything in test.py and paste it after from tkinter import \* in our simplebot.py (bottom left panel). Also make sure to delete our old get\_bot\_response function.

To run our code and try it: please click the button below. Use the other tab in the bottom-left panel

import os

import openai

openai.api\_key=os.getenv('OPENAI\_KEY')

MODEL='gpt-3.5-turbo-instruct'

*#MODEL="text-davinci-002"*

RESPONSE="I don't understand..."

def noop():

return

*# Generating response*

def get\_bot\_response(user\_input):

bot\_response=RESPONSE

prompt = f"Please provide a response to the following user input: '{user\_input}'"

try:

response = openai.Completion.create(

model=MODEL,

prompt=prompt,

max\_tokens=150,

n=1,

stop=None,

temperature=0.5,

)

except openai.error.InvalidRequestError:

noop()

except openai.error.RateLimitError:

noop()

else:

bot\_response = response.choices[0].text.strip()

return bot\_response

print(get\_bot\_response("off with your head!"))

Function with exception handler

*# Generating response*

def get\_bot\_response(user\_input):

bot\_response=RESPONSE

prompt = f"Please provide a response to the following user input: '{user\_input}'"

try:

response = openai.Completion.create(

model=MODEL,

prompt=prompt,

max\_tokens=150,

n=1,

stop=None,

temperature=0.5,

)

except Exception as exception: *# openai.error.InvalidRequestError:*

*# openai.error.RateLimitError:*

bot\_response = bot\_response + " (" + type(exception).\_\_name\_\_ + ") [" + str(exception) + "]"

*# str(exception)*

*# exception.\_\_str\_\_()*

*# exception.\_\_repr\_\_() -- more technical string representation*

*# f"Arguments: {exception.args}" array of strings for message*

*# exception.\_\_traceback\_\_*

*# exception.\_\_cause\_\_ None or ...*

*# exception.\_\_context\_\_ None or ...*

else:

bot\_response = response.choices[0].text.strip()

return bot\_response

DALL-E IMAGE Generation

# ***Learning Objectives***

### **Learners will be able to…**

* **Effectively utilize**openai.Image.create**to generate a response**
* **Successfully generate an image using DALL-E**
* **Analyze and apply the various parameters available in DALL-E API calls**

# **Text to Image**

## **Dall-E**

OpenAI’s **DALL·E** text to image is a system that can generate images from textual descriptions, using a deep learning-based approach. The system is trained on a large dataset of text-image pairs, and can generate new images that are similar to the ones in the training data. The system can generate images that are realistic and editable, and can even create images from scratch, without any training data.

DALL·E models can easily generate images from text, making it a great choice for those who want to create quick and easy visual content.

The Images API provides three methods for interacting with images:

1. Creating images from scratch based on a text prompt
2. Creating edits of an existing image based on a new text prompt
3. Creating variations of an existing image

## **Generating Image**

In order to generate an image using the DALL·E we can use the openai.Image.create call. Here is an example below:

response = openai.Image.create(

prompt="a dog smoking a cigar in a poker room.",

n=1,

size="256x256"

)

You can create an original image based on a text prompt using the image generations endpoint. Images can be sized at 256x256, 512x512, or 1024x1024 pixels, with smaller sizes generate faster. You can request 1-10 images at a time by using the n parameter. A 256x256 would generate faster than 1024x1024. For this course, we will tend to just use the 256 or the 512. These 2 are cheaper and faster to generate. API endpoints are the points where requests are made and responses are sent back. Let’s print the content of the response to see what was generated by the API call.

print(response)

When we printed the response we see there 3 distinct words created, data and inside data we see a URL. Here we want to focus on URL. The URL is the link to the image that the AI generated for us. Data represents all the info generated for the user and creation is pretty much an id. Now that we know what our response looks like we can trim down the response to what we actually want, which is the URL. Remove the print(response) line, and add the following.

image\_url = response['data'][0]['url']

print(image\_url)

We can open another tab in our browser, then we can copy and paste the URL to see the image that we have generated.

Try giving it a different size from the standard size mentioned above. For example, 255x255 or 256x1024.

It should generate an error, please revert back to a 256x256 or 512x512. We want it to be able to generate faster for future outputs. A Try it button is provided below to make sure we are no longer getting errors. It should just generate an URL here.

import os

import openai

import secret

import requests

openai.api\_key=secret.api\_key

response = openai.Image.create(

prompt="a dog smoking a cigar in a poker room.",

n=1,

size="256x256" # 512x512 or 1024x1024

)

image\_url = respoimport os

nse['data'][0]['url']

print(image\_url)import openai

import secret

import requests

openai.api\_key=secret.api\_key

response = openai.Image.create(

prompt="a dog smoking a cigar in a poker room.",

n=1,

size="256x256" # 512x512 or 1024x1024

)

image\_url = response['data'][0]['url']

<https://codio.com/bcowgill/intro-to-dall-e/tree/imageGen.py>

openai.error.InvalidRequestError: Billing hard limit has been reached

response = openai.Image.create(

prompt="a dog smoking a cigar in a poker room.",

n=1,

size="256x256"

)

Traceback (most recent call last):

File "imageGen.py", line 7, in <module>

response = openai.Image.create(

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_resources/image.py", line 36, in create

response, \_, api\_key = requestor.request(

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 226, in request

resp, got\_stream = self.\_interpret\_response(result, stream)

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 619, in \_interpret\_response

self.\_interpret\_response\_line(

File "/home/codio/anaconda3/lib/python3.8/site-packages/openai/api\_requestor.py", line 679, in \_interpret\_response\_line

raise self.handle\_error\_response(

openai.error.InvalidRequestError: Billing hard limit has been reached

# **Saving our Image**

### **Image Generation**

In order to not have to keep copying and pasting our result, we are going to save the image inside the URL as a file in our Codio box.

The first thing we are going to do is import the requests library.The import requests statement is already included in the imageGen.py file. You do not need to add it yourself.

Using the request library we can get the image contents. We will use a handler to write in our new image file image\_name.jpg. This code should go below all the code, we have already. At this point, we no longer need the print statement for the URL, delete it.

img\_data = requests.get(image\_url).content

with open('image\_name.jpg', 'wb') as handler:

handler.write(img\_data)

Click here to refresh your image

We will be rewriting our file. When regenerating an image we will need to click on our refresh button to see the changes. Try using the buttons below with a few different prompts.

If feeling uninspired, try the following prompts:

* digital art of a white cat ninja.
* acrylic painting of a mountain.
* a chef shark making shrimp.
* digital art of white ninja cat with headband.

Click here to refresh your image

The more descriptive you can be, the closer the image will get to what you are expecting. For example, Here is what it generated for me for that last input.

Writing descriptive OpenAI DALL·E prompts helps the model to better understand the context of the task, allowing it to create more accurate and creative responses. In other words, the more precise the explanation, the greater the probability of achieving the desired outcome for you or your end user.

import os

import openai

import secret

import requests

openai.api\_key=secret.api\_key

response = openai.Image.create(

prompt="a dog smoking a cigar in a poker room.",

n=1,

size="256x256" # 512x512 or 1024x1024

)

image\_url = response['data'][0]['url']

#print(image\_url)

img\_data = requests.get(image\_url).content

with open('image\_name.jpg', 'wb') as handler:

handler.write(img\_data)import os

Utility function to generate image and save to file.

import os

import openai

import secret

import requests

openai.api\_key=secret.api\_key

# prompt – string to send to Dall-E to generate the image

# filename – string filename.jpg to save the image

# size – string 256x256, 512x512 or 102x1024

# returns string url which contains the generated image

def create\_image\_openai(prompt, filename, size = '256x256'):

response = openai.Image.create(

prompt=prompt,

n=1,

size=size

)

image\_url = response['data'][0]['url']

img\_data = requests.get(image\_url).content

with open(filename, 'wb') as imageFile:

imageFile.write(img\_data)

return image\_url

# **N**

When generating our prompt our code looks something like this:

response = openai.Image.create(

prompt="a dog smoking a cigar in a poker room.",

n=1,

size="256x256"

)

image\_url = response['data'][0]['url']

We have talked about the prompt and the size. The main thing we are calling and have not talked about yet is the n. You can request 1-10 images at a time using the n parameter. Try replacing n with 3.

As you can see from the image file we saved in our response, we got only one image. Let’s try adding a print statement to visualize what we are getting for the URL.

image\_url = response['data'][0]['url']

print(image\_url)

Here we realize that we are going to only get one URL. We are going to change our code to view more of the response data.

image\_url = response['data']

print(image\_url)

Here we can see we got 3 URLs generated. What we are going to do is pull the different URL into separate variables and generate files to save the URL. To make our life easier we should have the following:

image\_url1 = response['data'][0]['url']

image\_url2 = response['data'][1]['url']

image\_url3 = response['data'][2]['url']

Please note you have have had image\_url = response['data'][0]['url'] from the following page,comment it out for now. If you want to check if the URLs look like what you have in mind, you can run the following code:

print(image\_url1[0:15])

print(image\_url2[0:15])

print(image\_url3[0:15])

We are slicing the content of our print statements here because we are generating URLs. The URLs tend to be in a longer format, because of that slicing might make the result more pleasing to the eye since we are just checking.

# **Saving Multiple Images**

Now that we can get the URL for the 3 images we are going to save them in different files. The button below after the Try It! button are setups to open the following file names:

Now remember the technique we previously used to save our image. We are going to use that to save the 3 images.

img\_data = requests.get(image\_url1).content

with open('image\_name1.jpg', 'wb') as handler:

handler.write(img\_data)

img\_data = requests.get(image\_url2).content

with open('image\_name2.jpg', 'wb') as handler:

handler.write(img\_data)

img\_data = requests.get(image\_url3).content

with open('image\_name3.jpg', 'wb') as handler:

handler.write(img\_data)

If you click on every single link and you get a different image then you are on the right track. Since we are using Python let’s start by cleaning up our code a bit.

For starters in order to better see the change that our code worked we are going to switch the prompt from ninja cat to ninja birds.

prompt="digital art of ninja bird "

Then remove everything after from when we mention img\_data = requests.get(image\_url1).content.

Now that we have standard style to name our images it makes it easy for us to use Python function like enumerate to loop through and create our files. All we have to do is simply change the values from 1 to 2 to 3.

for i, image\_url in enumerate([image\_url1, image\_url2, image\_url3], start=1):

img\_data = requests.get(image\_url).content

with open(f'image\_name{i}.jpg', 'wb') as handler:

handler.write(img\_data)

Just like that you cut more than half the lines of code you needed to store the different files. Using this code it should be pretty easy to replicate and save more than 3 pictures. Remember that n has a max value of 10. Please Note: For free accounts the maximum rate is 5 pictures per minute.

Edits with DALL-E

# **Learning Objectives**

# ***Learning Objectives***

### **Learners will be able to…**

* **Effectively generate an image using DALL-E**
* **Create variations of DALL-E images using different parameters and techniques**
* **Analyze and apply the concepts of masks and edits in DALL-E image creation.**

# **Text to Image Edits**

## **Dall-E**

OpenAI’s **DALL·E** text to image is a system that can generate images from textual descriptions, using a deep learning-based approach. The system can generate images that are realistic and editable, and can even create images from scratch, without any training data. Endpoints are often used to retrieve data from a database, create new data, or modify existing data. We have seen that DALL·E models can easily generate images from text. Now we are going to use DALL·E to modify existing data

The Images API provides three methods for interacting with images:

1. Creating images from scratch based on a text prompt
2. Creating edits of an existing image based on a new text prompt
3. Creating variations of an existing image

## **Generating Image**

We are familiar with the steps to generate a new image.

#this code generates a new image from a prompt

response = openai.Image.create(

prompt="digital art of throne room of a magical castle.",

n=1,

size="256x256"

)

#from the response generated this puts the url in a separate variable

image\_url = response['data'][0]['url']

#using the url, we use the code below to save it as a file

img\_data = requests.get(image\_url).content

with open('image\_name.png', 'wb') as handler:

handler.write(img\_data)

Click here to refresh your image1

## **Variation**

OpenAI has an image variation endpoint that allows you to generate similar images when given an image as a prompt.

To try it in the last Try It button we created a digital art representation of a castle as image\_name.jpg. We are going to use that image to generate another similar picture or variation. To avoid rewriting our original file we are going to delete everything up until the comment that says # Keep CODE ABOVE. Then we are going to write, the code below to generate a variation.

response = openai.Image.create\_variation(

image=open("image\_name.png", "rb"),

n=1,

size="1024x1024"

)

image\_url = response['data'][0]['url']

#using the url, we use the code below to save it as a file

img\_data = requests.get(image\_url).content

with open('image\_name\_var.png', 'wb') as handler:

handler.write(img\_data)

# **Variation**

Let’s try creating an image variation of a bunny and a cat.

1. First we generate the original image of the bunny and cat.

#this code generates a new image from a prompt

response = openai.Image.create(

prompt="bunny and a cat",

n=1,

size="512x512"

)

#from the response generated this putS the url in a separate variable

image\_url = response['data'][0]['url']

We save the bunny and cat image.

#using the url, we use the code below to save it as a file

img\_data = requests.get(image\_url).content

with open('bunny.png', 'wb') as handler:

handler.write(img\_data)

We use the new image to generate a variation of the said image using our new file name we created in the previous step

response = openai.Image.create\_variation(

image=open("bunny.png", "rb"),

n=1,

size="512x512"

)

image\_url2= response['data'][0]['url']

Now we have to save our new URL in a different file name.

img\_data = requests.get(image\_url2).content

with open('bunny\_var.png', 'wb') as handler:

handler.write(img\_data)

Prompts and images are filtered based on OpenAI’s [content policy](https://labs.openai.com/policies/content-policy), returning an error when a prompt or image is flagged.

# **mask**

On top of a variation endpoint OpenAI’s DALL·E has an **edit endpoint**. Similar, to our variation API call we are going to need to include a starting image for it to work on, a numbern, and size. Unlike the variation endpoint we are going to add a mask, and a prompt.

First let us generate the image we will be working with. For the sake of consistency we will be creating another from scratch.

#this code generates a new image from a prompt

response = openai.Image.create(

prompt="a vibrant modern office with red chairs and a television screen.",

n=1,

size="512x512"

)

#from the response generated this puts the url in a separate variable

image\_url = response['data'][0]['url']

#using the url, we use the code below to save it as a file

img\_data = requests.get(image\_url).content

with open('original.png', 'wb') as handler:

handler.write(img\_data)

print(image\_url)

The image generated when I first ran the code is provided below. Feel free to use if you do not want to interact with one by yourself. The image is saved under original-copy.png. Below is a code provided to help you copy and use it. Ignore this bit if going to create your own.

import shutil

def copy\_image(source\_path, destination\_path):

try:

shutil.copy2(source\_path, destination\_path)

print("Image copied successfully!")

except IOError as e:

print(f"An error occurred while copying the image: {e}")

# Example usage

source\_image = "original-copy.png"

destination\_image = "original.png"

copy\_image(source\_image, destination\_image)

After generating this bit we are going to delete/remove the code to generate an image and simply keep our libraries. We don’t need to regenerate an original image.

Now we need to create our **mask**. The purpose of a mask is to define the region of an image that you want to change or apply an effect to, while leaving the rest of the image unchanged.

In image processing, a mask is typically a grayscale image with the same dimensions as the original image. The mask is used to selectively modify the original image by indicating which pixels should be affected by an operation and which pixels should be left unchanged.

Now we are going to generate code to basically write over our image in order to create the mask. First, we are going to open our original image and convert it to RGBA. This step is important otherwise it would generate an error saying the following:openai.error.InvalidRequestError: Invalid input image - format must be in ['RGBA', 'LA', 'L'], got RGB. After generating and saving your file feel free to delete the previous code that was used for image generation.

## Load the original image

img = Image.open('original.png')

# Convert the image to RGBA format

img = img.convert('RGBA')

We are going to create a copy of our original image so we can put a mask over it.

# Create a new blank image for the mask

mask = img.copy()

Now for the fun part we simply gonna draw a rectangle box over where the edit should be. Here is an example:

We know our picture is 512x512. Depending on your image change the location of the box so it makes sense so an edit can occur. Feel free to keep changing the value of bbox until you have a version that suits you.

The bbox variable is a tuple containing four values: (left, top, right, bottom).

left is the x-coordinate of the left edge of the bounding box

top is the y-coordinate of the top edge of the bounding box

right is the x-coordinate of the right edge of the bounding box

bottom is the y-coordinate of the bottom edge of the bounding box

Finally, we are also going to make sure we save our picture in a file called masked.png.

# Define the region you want to keep in white

left=300

top=200

right=420

bottom=300

bbox = (left, top, right, bottom)

mask\_draw = ImageDraw.Draw(mask)

mask\_draw.rectangle(bbox, fill=255)

# Save the mask

mask.save('masked.png')

When creating a mask for an image edit operation, the mask should have the same dimensions as the original image and be in RGBA format. The mask is used to selectively modify the original image by indicating which pixels should be affected by an operation and which pixels should be left unchanged. The ‘A’ in RGBA stands for the alpha channel, which is used to represent the transparency of the pixels.

# **Editing our Image**

Now that we have our mask and original picture we can generate our edited picture. We are going to slightly modify our prompt to describe what we want generated

response=openai.Image.create\_edit(

image=open("original.png", "rb"),

mask=open("masked.png", "rb"),

prompt="a vibrant modern office with red chairs and a television screen.a snack is on the table. ",

n=1,

size="512x512"

)

image\_url = response['data'][0]['url']

#using the url, we use the code below to save it as a file

img\_data = requests.get(image\_url).content

with open('edited.png', 'wb') as handler:

handler.write(img\_data)

We can also change the prompt to display other things such as:

response=openai.Image.create\_edit(

image=open("original.png", "rb"),

mask=open("masked.png", "rb"),

prompt="a vibrant modern office with red chairs and a television screen. A bagel is on the table. ",

n=1,

size="512x512")

The mask parameter is optional. If not provided our image still needs to have transparency, which will be used as the mask. Feel free to move the box associated with your mask, in order to make different edits with your picture.

# **Coding Exercise**

Write a function variationMaker() that takes in a file name like image.png. The functions should create 1 variation of the image in 512x512 and save it under a file name called varTest.png.

Feel free to test it out before submission. A try it button will be provided, additionaly so will varTest.pngfile.

import os

import openai

import secret

import requests

openai.api\_key=secret.api\_key

def variationMaker(fileName):

response = openai.Image.create\_variation(

image=open(fileName, "rb"),

n=1,

size="512x512"

)

image\_url = response['data'][0]['url']

img\_data = requests.get(image\_url).content

with open('varTest.png', 'wb') as handler:

handler.write(img\_data)

# Codio:Additional DALL·E Usage

# **Learning Objectives**

### **Learners will be able to…**

* **Effectively generate multiple images using appropriate API calls or programming techniques**
* **Apply appropriate methods for saving multiple generated images in various file formats**.
* **Create Synchronous and Asynchronous API calls**

# **Chaining Multiple API Calls to Create a Sequence of Images with DALL-E**

In the previous lessons, we have learned how to generate images using OpenAI’s DALL-E API by defining specific conditions or constraints in our prompts. Today, we will take a step further and learn how to chain multiple API calls together to create a sequence of images. This technique can significantly enhance the creativity and complexity of the outputs we can generate.

### **Benefits of Chaining**

Chaining API calls allows us to generate multiple images in a sequence based on a set of prompts. This technique has several benefits:

**Storytelling**: By chaining prompts, we can create a sequence of images that tell a story. This can be useful in a variety of applications, from creating storyboard visuals to generating illustrative examples for educational content.

**Progressive image generation**: Chaining allows us to generate images that progressively change based on our sequence of prompts. This can be used to simulate movement, transformation, or progression over time.

**Exploring different variations**: By chaining a series of related prompts, we can generate a set of images that show different interpretations or variations of the same concept. This can help us understand how DALL-E interprets and responds to different prompts.

import os

import openai

import requests

import secret

openai.api\_key=secret.api\_key

# Set the prompts

prompts = ["robot dog in a lab", "robot dog exploring the city", "robot dog watching the sunset"]

# Generate and save the images

for i, prompt in enumerate(prompts):

response = openai.Image.create(

prompt=prompt,

n=1,

size="256x256"

)

# Get the image URL from the response

image\_url = response['data'][0]['url']

# Download and save the image

img\_data = requests.get(image\_url).content

with open(f"robot\_dog\_journey\_{i+1}.jpg", 'wb') as handler:

handler.write(img\_data)

In this example, we first define a list of prompts. Each prompt represents a different stage of the "robot dog’s journey".

We then loop over each prompt in the list. For each prompt, we call the DALL-E API to generate an image that corresponds to that stage of the journey. We specify prompt as the prompt, n=1 to generate one image per prompt, and size="256x256" to specify the size of the generated image.

From the API response, we extract the URL of the generated image. We then use the requests library to download the image from this URL.

Finally, we save each image with a unique filename that includes the index of the prompt in the list. This allows us to view the images in the order they were generated, effectively seeing the robot dog’s journey unfold in sequence.

By chaining these API calls together, we can generate a sequence of images that tell a visually compelling story. This approach can be extended and adapted to generate sequences of images for a wide range of applications.

# **Exceptions in DALL-E**

While using APIs, it is common to encounter various types of errors and exceptions. These could occur due to a variety of reasons like network issues, incorrect parameters, rate limiting, server errors, etc. you can ensure that your program doesn’t crash unexpectedly and provides useful feedback about what went wrong. This is crucial for debugging issues and improving the reliability of your code. This is especially important when making multiple calls at the same time like when chaining.

Here are some common types of errors and exceptions you may encounter while using the DALL-E API:

**HTTP Errors:** These are errors returned by the server and they come with an HTTP status code. For example, a 404 error means the requested resource could not be found, and a 500 error means there was an internal server error.

**API Errors:** These are errors returned by the API itself due to issues like incorrect parameters, exceeding rate limits, etc. These usually come with an error message explaining what went wrong.

**Network Errors:** These are errors that occur due to network issues, like a timeout because the server took too long to respond.

Here is an example where we use a try-except block to catch and handle potential errors and exceptions.

import os

import openai

import requests

import secret

openai.api\_key=secret.api\_key

# Set the prompts

prompts = ["robot dog in a lab", "robot dog exploring the city", "robot dog watching the sunset"]

# Generate and save the images

for i, prompt in enumerate(prompts):

try:

response = openai.Image.create(

prompt=prompt,

n=1,

size="256x256"

)

# Get the image URL from the response

image\_url = response['data'][0]['url']

# Download and save the image

img\_data = requests.get(image\_url).content

with open(f"robot\_dog\_journey\_{i+1}.jpg", 'wb') as handler:

handler.write(img\_data)

except requests.exceptions.RequestException as e:

# This will catch any general network error

print(f"Network error: {e}")

except openai.error.APIError as e:

# This will catch any error returned by the OpenAI API

print(f"API error: {e}")

except Exception as e:

# This is a catch-all for any other exceptions

print(f"Unexpected error: {e}")

The try block contains the code that could potentially raise an exception. If an exception is raised in the try block, the execution immediately moves to the except block that handles that specific exception.

The requests.exceptions.RequestException except block will handle any network errors that might occur during the API call or while downloading the image.

The openai.api\_errors.APIError except block will handle any errors returned by the OpenAI API, such as incorrect parameters or exceeding rate limits.

Finally, the general Exception except block will catch any other exceptions that the specific catch blocks did not catch. This is a good practice to ensure that your program can recover from any unexpected exceptions.

# **Async vs Sync**

In a typical **synchronous application**, each task must complete before the next one can start. This can be inefficient when dealing with I/O operations like network requests, where the program spends a lot of time waiting for responses.

**Asynchronous programming** allows you to perform other tasks while waiting for I/O operations to complete, resulting in more efficient use of resources.

Python’s asyncio library allows you to write single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives.

### **Asynchronous**

While using DALL-E’s API, you might need to generate multiple images at once. If you do it synchronously, your application will have to wait for each API call to complete before making the next one. This is inefficient, especially if the API calls take a long time to process.

By making asynchronous API calls, you can send multiple requests to the API at once without having to wait for each one to complete. This can significantly speed up your application if you need to make a lot of API calls.

### **Synchronous example**

In order to compare the time between both calls, we are going to time our previous chain request. Copy and paste the code in this page in the sync.py file on the left. To control the time we have to use the following code. The code will start a timer then end it when we are done testing and print the result. here is a sample code, just for your visual learning.

import os

import openai

import requests

import secret

import time

openai.api\_key = secret.api\_key

import time

start\_time = time.time()

*# Your existing code here*

end\_time = time.time()

execution\_time = end\_time - start\_time

print(f"Execution time: {execution\_time} seconds")

By using time.time() function, we capture the start time before the execution of your code and the end time after the execution. Then, we calculate the difference between the two timestamps to get the execution time in seconds. Finally, we print the execution time.  
To keep it consistent let’s basically use the same Python code we have been using in this assignment.

# **Async Example**

Now, let’s see how you can use Python’s asyncio and aiohttp libraries to make asynchronous API calls to DALL-E:

The code we are working with in the async.py file creates multiple tasks, each one responsible for generating an image and saving it.  
These tasks will run concurrently, which can speed up the total execution time compared to running them sequentially. Please be aware that OpenAI’s API might have rate limits depending on your plan, so please be cautious about making too many concurrent requests. We are going to do just two big Try It button at the end, in order not to waste a ton of our tokens.

### **Coding**

**fetch\_and\_save\_image function**: This function is designed to download and save the image from the given URL. The function takes a session (aiohttp’s ClientSession), a URL, and a path to save the image. We use aiohttp’s get() function to make a GET request to the URL, and then use resp.read() to read the response content (image data) asynchronously. We then open the file at the given path and write the image data to it.

async def fetch\_and\_save\_image(session, url, path):

try:

async with session.get(url) as resp:

img\_data = await resp.read()

with open(path, 'wb') as handler:

handler.write(img\_data)

except Exception as e:

print(f"Unexpected error: {e}")

**fetch\_image function:** This function makes a request to OpenAI’s API to generate an image for the given prompt. The function takes a prompt and an index as input. It calls openai.Image.create() to generate an image and gets the image URL from the response. It then creates an aiohttp ClientSession and calls fetch\_and\_save\_image() to download and save the image.

async def fetch\_image(prompt, i):

try:

response = openai.Image.create(

prompt=prompt,

n=1,

size="256x256"

)

# Get the image URL from the response

image\_url = response['data'][0]['url']

# Download and save the image

async with aiohttp.ClientSession() as session:

await fetch\_and\_save\_image(session, image\_url, f"robot\_dog\_journey\_{i+1}.jpg")

except openai.error.APIError as e:

# This will catch any error returned by the OpenAI API

print(f"API error: {e}")

except Exception as e:

# This is a catch-all for any other exceptions

print(f"Unexpected error: {e}")

**main function:** This function creates an asyncio task for each prompt, adds them to a list, and then runs them concurrently using asyncio.gather(). This means that it will start all the tasks at the same time, and then wait for all of them to finish. This is where the asynchronous magic happens - rather than processing each prompt one by one, it starts processing all of them at once, which can significantly speed up the total execution time. It also measures the execution time by recording the time before and after the tasks are run, and then subtracts the start time from the end time.

async def main():

start\_time = time.time() # Start measuring execution time

tasks = []

for i, prompt in enumerate(prompts):

tasks.append(fetch\_image(prompt, i))

await asyncio.gather(\*tasks)

end\_time = time.time() # End measuring execution time

execution\_time = end\_time - start\_time

print(f"Execution time: {execution\_time} seconds")

Your whole code outline in async.py should look like the following code. Additionally, to run the main function we have added asyncio.run(main()) at the end of our code.

import os

import openai

import aiohttp

import asyncio

import secret

import time

openai.api\_key = secret.api\_key

# Set the prompts

prompts = ["robot dog in a lab", "robot dog exploring the city", "robot dog watching the sunset"]

async def fetch\_and\_save\_image(session, url, path):

try:

async with session.get(url) as resp:

img\_data = await resp.read()

with open(path, 'wb') as handler:

handler.write(img\_data)

except Exception as e:

print(f"Unexpected error: {e}")

async def fetch\_image(prompt, i):

try:

response = openai.Image.create(

prompt=prompt,

n=1,

size="256x256"

)

# Get the image URL from the response

image\_url = response['data'][0]['url']

# Download and save the image

async with aiohttp.ClientSession() as session:

await fetch\_and\_save\_image(session, image\_url, f"robot\_dog\_journey\_{i+1}.jpg")

except openai.error.APIError as e:

# This will catch any error returned by the OpenAI API

print(f"API error: {e}")

except Exception as e:

# This is a catch-all for any other exceptions

print(f"Unexpected error: {e}")

async def main():

start\_time = time.time() # Start measuring execution time

tasks = []

for i, prompt in enumerate(prompts):

tasks.append(fetch\_image(prompt, i))

await asyncio.gather(\*tasks)

end\_time = time.time() # End measuring execution time

execution\_time = end\_time - start\_time

print(f"Execution time: {execution\_time} seconds")

# Run the main function

asyncio.run(main())

Please note that the openai.Image.create function is a blocking operation and doesn’t support async. The code above assumes that this function is asynchronous, which is not correct. To truly parallelize this, you’d need an async-compatible OpenAI library or use something like Python’s concurrent.futures to run the blocking parts in a separate thread.

HEREIAM