OpenAI API with Python Bootcamp Notes:

Notes taken for "OpenAI API with Python Bootcamp: ChatGPT API, GPT-4, DALL-E" by Andrei Dumitrescu and Crystal Mind Academy.

From course description: "*Learn ChatGPT API with Python. Integrate OpenAI Models like GPT-4 with your Python applications. Project-based course*."

Notes taken by Travis Rillos. These notes and all app code and practice code organized in the following Github repository: https://github.com/xroadtraveler/open-ai-api-python-bootcamp.

Project List:

- **Project 1:** Zero-Shot sentiment analysis using ChatGPT.
- **Project 2:** Building a ChatGPT clone from scratch (Chat-bot).
- **Project 3:** Building a healthy daily meal plan.
- **Project 4:** YouTube videos summary generator.
- **Project 5:** Program like a pro with Gpt-4.
- **Project 6:** Boost your Linux Sysadmin capabilities with ChatGPT (ShellGPT).
- **Project 7:** GPT-4 text embeddings.

Table of Contents

Section 1 – Getting Started:

Course Introduction:

- Just an overview.
- This course requires some basic Python programming knowledge (check).
- There are **11 Sections**.

Setting Up the Environment: Jupyter Notebook and Google Colab:

- Hmm, looks like we'll be using **Jupyter Notebook** for this course rather than something like VSCode.
    - The instructor noted that Jupyter Notebooks is probably the most popular Python development environment for **mac**hine **lea**rning, **de**ep **lea**rning, **da**ta **sci**ence, and **da**ta **vis**ualization.
- In this lecture, he shows how to install Jupyter Notebook. However, I already have it, having used it on a previous Python course.
- The instructor also noted that if you ***prefer*** to use any other IDE such as **PyCharm** or **VSCode**, you are free to do so.
    - At the end of the day, it doesn't matter where you write your code, as long as it works.
    - I may try to simultaneously run things in both Jupyter Notebook and VSCode like I did for several course projects in the past. This allows me to get practice in both, and I might also get to test out the **CodeGPT extension** for VSCode in some real use-cases.
- The name "Jupyter" is an indirect acronym of the *three core languages* it was designed for: **Julia**, **Python**, and **R** (I did not know that).
    - It's a web application that runs in the browser, that can run live Python code, visualizations, etc.
-
- After installing Jupyter, he created a directory to store Jupyter files for the course. I took this opportunity to create my own directory as well as a Github repo for course files.
-
- Once we had our directory created, we opened Jupyter Notebook in said directory using the command line. Once it was open in our default browser, we created a new notebook and gave it the name "**learn_python**".
    - From here he just showed some of Jupyter's basic functionality. Pretty basic stuff.
    - He started by showing that you can use Markdown in Jupyter by selecting it from the dropdown menu (by default, this is set to "Code").
        - The four options in the dropdown menu are:
            - Code
            - Markdown
            - Raw NBConvert
            - Heading

- He ended the lecture by saying that you can use **Google Colab** exactly like Jupyter Notebook, especially in cases where you're unable to download/use Jupyter Notebook, or perhaps don't have the right permissions.
- Google Colab has many commonly used Python libraries already installed.

Instructor's Note: Join My Private Community:

"I have created a private and exclusive Online Discord Community to provide you with improved, faster, and better support for your course-related questions.

Moreover, you are going to use this Community to better interact with your course colleagues and to help out others whenever I'm not around.

CLICK HERE TO JOIN THE COMMUNITY NOW!

Lots of companies around the world use Discord to communicate across teams therefore it's a valuable tool for you as you advance to becoming a valuable Engineer.

And don't forget to subscribe to my Channel for video tutorials on Programming, Networking, Information Security, Blockchain, or other Cutting-Edge Technologies!"

Course Resources:

"All files used in this course (notebooks included) used in this course are available as a **Google Drive Shared Directory**.

You can access the course resources here: https://drive.google.com/drive/folders/1_eaig9xr88G_ZeA7lPeY-QM_89OtT3XE?usp=share_link

The best way to learn to program is to write every piece of code by yourself so I encourage you to do so and if you get stuck to check it with my examples.

Many scripts are ready to be used in your Network Environment with no or very few changes."

Section 2 – Deep Dive into OpenAI API: ChatGPT, GPT-4, GPT-3, DALL-E, Whisper:

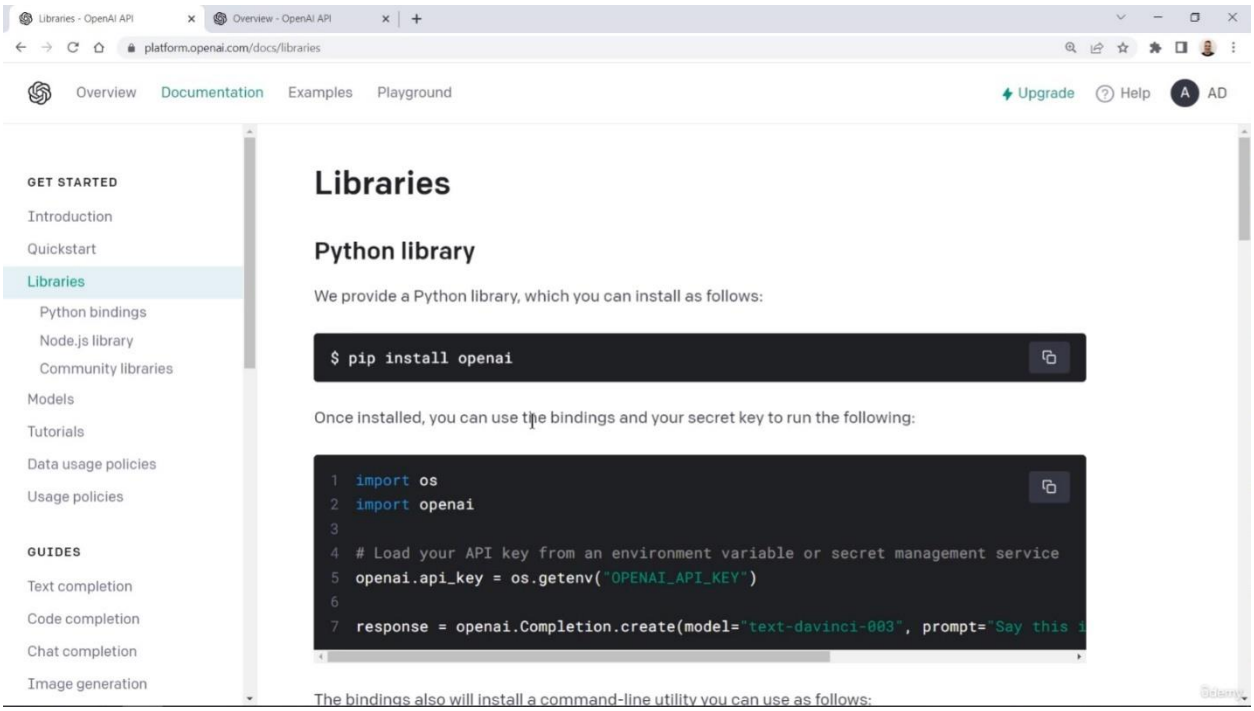
Creating an OpenAI Account and an API Key:

- This section is about creating an OpenAI account and an API key (I already have both).
-
- What is an API?
    - API (Application Programming Interface) is a mechanism that enables two software components to communicate with each other using a set of definitions and protocols.
    - We can use an API key to talk to OpenAI's models with our applications using HTTP requests.

Storing API Key as an Environment Variable:

- **Note:** Since I'll be committing my programs for this class to a public Github repo, I decided to look into using an ***Environment Variable*** on my Windows 10 machine to keep a secret API key secret. I created a new secret key just for class uses, then stored that key as an Environment Variable using "Advanced System Settings" in Windows.

Installing the OpenAI API Library and Authenticating to OpenAI:



- There are native libraries for OpenAI in **Python** and **Node.js**, plus *community libraries* for many other programming languages such as C#/.NET, Go, Java, et. al.
-
- We opened Jupyter Notebook in our directory and created a new Notebook, naming it "**01_openai_authentication**".
    - Note: I had already used **pip install openai** on my main Python installation at some point in the past, so I wonder if I'd be able to skip any of this.
    - He showed how to install from the line "**import openai**" written as the first line in a program in the PyCharm IDE. Guess you can right-click and select "install" in some IDEs.
- Inside a Jupyter Notebook cell, we can install OpenAI in the ***local environment*** by typing the line…
    - **!pip install -q openai**
    - …and then running the cell. Don't forget the *exclamation mark* at the beginning.
        - The "**-q**" flag stands for "**quiet**" and makes it so **pip** gives less output.
    - He also showed that you can do the exact same thing in **Google Colab**.
- After installing in Jupyter, we comment out the "pip" line, then write the two lines…
    - **import openai**
    - **import os**
    - …directly below it. Run the cell again.
-
- In order to *use* OpenAI in Python, we have to ***authenticate*** it. We do this with the ***OpenAI API key*** that we created.
- The recommended way to do this is to set the ***secret key*** as an ***environment variable***. We can do this using the **os module** that we just imported.

- In order to *use* OpenAI in Python, we have to ***authenticate*** it. We do this with the ***OpenAI API key*** that we created.
- The recommended way to do this is to set the ***secret key*** as an ***environment variable***. We can do this using the **os module** that we just imported.
    - Example:
        - **os.environ['x'] = 'abc'**
    - In this example, we've created an ***environment variable*** called '**x**' and we've given it the value of '**abc**'.
    - To retrieve this variable, we write:
        - **os.getenv('x')**
    - Since I was already planning on committing my programs to a Github repo and want to maintain security, I've already set my secret key as an environment variable using Windows settings. I've named it "**OPENAI_CLASS_API_KEY**". Let's see if running **os.getenv('OPENAI_CLASS_API_KEY')** works.
        - It worked!!!
        - Sweet.
    - The instructor showed himself doing the same. He noted that if you set your secret API key as an environment variable with this method, you should delete that line/cell afterwards in order to keep it secret.
    - Next we'll set the variable **openai.api_key**:
    - My VSCode version:

```
import openai
import os

openai.api_key = os.getenv('OPENAI_CLASS_API_KEY') ← ← ←
```

    - The instructor's Jupyter Notebook screen:

```
In [2]:
1 # !pip install -q openai
2 import openai
3 import os
```

```
In [4]:
1 # os.environ['x'] = 'abc'
2 # os.getenv('x')
3 os.environ['OPENAI_API_KEY'] = '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'
4 openai.api_key = os.getenv('OPENAI_API_KEY')

Out[4]: 'abc'
```

-
- At this point, the instructor noted that instead of presenting the secret key in plaintext as we've been doing so far, you can ***prompt the user*** for their secret key and set it to that.
- To do this, the **getpass** module is an alternative to the built-in user input function. To use it we need to write **import getpass**.
- The code should look something like this:
-

- The code should look something like this:

```
import openai
import os
import getpass

# openai.api_key = os.getenv('OPENAI_CLASS_API_KEY')

key = getpass.getpass("Past your API Key: ")
openai.api_key = key
```

- Or, in Jupyter Notebook, the prompt shows up right underneath the cell you've just run:

```
In [2]:
# !pip install -q openai
import openai
import os
```

```
In [*]:
# os.environ['x'] = 'abc'
# os.getenv('x')
# os.environ['OPENAI_API_KEY'] = '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'
# openai.api_key = os.getenv('OPENAI_API_KEY')
import getpass
key = getpass.getpass('Paste your API Key:')
openai.api_key = key

Paste your API Key:
```

    - The user would *paste* their API key here, setting it as our openai.api_key variable without exposing it as *plaintext*.
-
- The last solution to input your secret key is to ***load it from offline***. In this example, the instructor had a .txt file called **key.txt** in his Jupyter Notebook session:

- The code to do this is:
    - **openai.api_key = open('key.txt').read().strip('\n')** – (I'm assuming the .txt file needs to be in the same directory/level as your program in this line).
-
-
-
-

- The code to do this is:
    - **openai.api_key = open('key.txt').read().strip('\n')** – (I'm assuming the .txt file needs to be in the same directory/level as your program in this line).

    - So with this, your ***opening*** the text file in *read* mode (the default for open()), ***reading*** it as a *string*, and then the **.strip()** method is optional, but some users add a new line at the end of a .txt file and this needs to be removed.
-
- So we've gone over ***three methods*** of loading your API key securely, and we can use any of these methods for the remainder of this section. Personally I'm just going to call my ***environment variable*** because I already set it the other day, and it should be usable for any programs we write for this course.

OpenAI Models:

- So we've opened our OpenAI accounts, gotten our secret API key, and stored the key as an environment variable. The next step would be to start making *requests* to the OpenAI Models.
    - We'll do that soon, but for the moment we're going to ***learn about the AI Models*** that are available to choose from.
- **AI Models:** An AI Model is a software program that uses specific ML and DL algorithms and has been trained on a set of data to perform specific tasks.
- The OpenAI API has access to a family of models, each with different capabilities.
    - At the time this lecture was recorded, these included GPT-3.5, DALL-E, Whisper, Embeddings, Codex, Moderation, and GPT-3, though recently GPT-4 was also released.
        - Models can be modified for specific use-cases; this is called "model fine-tuning".
    - **GPT-3** consists of various simpler models such as **davinci**, which is the most capable, but also (at the time of recording) the most expensive. On the other hand, **text-ada-001** is the simplest, the fastest, and the cheapest. The other models within GPT-3 are in-between.
        - You can find information about pricing for any model in the "Pricing" tab/URL.
    - **GPT-3.5** is a family of models that improve on GPT-3's features. Its simplest/fastest/cheapest model is **gpt-3.5-turbo**, which is optimized to generate both *natural language* and *code*, create embeddings, edit text, and other tasks. GPT-3.5's most advanced model is **text-davinci-003**.
        - **ChatGPT** is (by default) powered by **GPT-3.5** (though now GPT-4 is also an option for paid accounts). However, if you have very simple tasks and a limited budget, you can take the other models into account.
        - Keep in mind that any task that can be performed by a simpler model like **ada** can also be performed by a more sophisticated model like **davinci**.
    - You can use the **Playground** to run and compare different models, settings, output, and response times.
        - To show this off, he chose to compare **ada** and **davinci** using the question "is the capital of France". Starting off with **text-davinci-003**, and it answered "Paris". It was smart enough to assume that this was meant to be a question, and answered it.
        - Trying the same thing with **text-ada-001** gave the result "Paris is the capital of France". Instead of simply answering, it went and filled in the blank and returned the entire sentence. This is acceptable.
    - In the Playground, let's now try a more complicated task such as a translation.
        - In **text-davinci-003**, he entered "Translate the following text from English to Romanian" followed by "Text: How are you today?" on the next line and then "Translation:" on the next.
            - The model was able to add "Cum esti azi?" after "Translation:". Pretty smart.
        - Next we run the exact same task using **text-ada-001**. The simpler model just returned "How are you day?", which isn't even correct in English. It wasn't able to complete the task, so **ada** has failed.

- Some of the slider options on the righthand side, such as "**Temperature**" and "**Frequency penalty**" will be discussed later in the course.
- Aside from the GPT-3 and GPT-3.5 models, there are other models as well.
    - **DALL-E** can generate images from text descriptions.
    - **Whisper** can convert audio into text. It's capable of *multilingual speech recognition* as well as *speech translation* and *language identification*.
-
- This has been an overview of some of OpenAI's most important AI models. In the next lecture we'll use GPT-3 and ChatGPT requests using the OpenAI API in Python.

Making GPT-3 Requests Using the OpenAI API:

- GPT is often used to process text and generate responses to given prompts. This is what many Large Language Models (LLMs) are geared towards.
- It's important to think carefully about what prompts to use in order to get the desired output. As a result of this, a *new field* known as ***prompt engineering*** is now emerging.
-
- In this lecture, we want to make requests to the GPT API, to a text-completion AI model.
    - The most powerful models in the current family (pre-GPT-4) are **davinci** and **GPT-3.5**.
-
- We can create a *Python variable* to store our prompt:

```
prompt = "Write a motto for a football team"
```

- Using this prompt variable, we can make *specific requests* using either **davinci** or **GPT-3.5**; we get to choose which one we use.
    - We can use the **openai.Completion.create()** method to do this, and we can store that as a variable called **response**.
    - Inside this method, we can also use ***keyword arguments*** so we don't have to worry about the order of the arguments. One such keyword argument is the ***model name***, and others include **temperature** and **max_tokens**:

```
prompt = "Write a motto for a football team"
response = openai.Completion.create(
    model='text-davinci-003',
    prompt=prompt,
    temperature=0.8,
    max_tokens=1000
)
```

- So what is ***temperature***?
    - The **temperature** is the parameter that controls how much **randomness** is in the output.
    - The higher the temperature, the more random the answer will be.
    - It can be anywhere between **0** and **2**, and the default is **1**. If you set it to 0, the output will be very deterministic and repetitive. If you set it higher, the output will be more diverse, but also more prone to errors.
- The **max_tokens** variable is the maximum number of tokens to be generated in the *completion*. This limit can vary based on which AI model we use.
    - Note that both **temperature** and **max_tokens** are *optional* parameters.
-
- Once we have our OpenAI/response stuff working, we want to **print(response)** in order to see the output:
-
-
-
-

- Once we have our OpenAI/response stuff working, we want to **print(response)** in order to see the output:

```
In [8]: print(response)
```


```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "text": "\n\n\"United by the win, divided by none!\""
    }
  ],
  "created": 1687219376,
  "id": "cmpl-7TJ7g1uDtQSc8AHR3opb4rEiqW3cu",
  "model": "text-davinci-003",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 12,
    "prompt_tokens": 7,
    "total_tokens": 19
  }
}
```


```
In [ ]:
```

- The response I received read "***Uni**ted **b**y **t**he **w**in, **div**ided **b**y **no**ne!*". The instructor's was "***Tea**mwork **Ma**kes **T**he **Dr**eam **Wo**rk!*".
- It comes in as a **JSON object**, something similar to a Python ***dictionary***. In order to extract just the response, we the *value* of the ***key***: "choices". This gives us a ***list*** of *one element*, another dictionary, which we can extract using index **[0]**:

```
r = response['choices'][0]
```

- Then we **print(r)** and this gives us another ***dictionary***, one of *four* elements. We can extract the *value* from the *key*: "**text**". So let's add that key on the end:

```
r = response['choices'][0]['text']
```

- This gives us just the text response on its own!
-
- In the next lecture, we'll learn how to make ***ChatGPT requests***.

Making ChatGPT Requests Using the OpenAI API:

- In the previous video, we talked about using **davinci** for making requests.
- In this video, we focus on making requests to **ChatGPT**, which is powered by **GPT-3.5** and is **optimized for dialogue**.
-
- In a new Jupyter cell, we'll make a **new prompt**.
    - We can **opt**imize **ea**ch **pro**mpt **f**or a **spe**cific **A**I **mo**del.
    - But for the moment, we'll use a standard question.
    - Instead of sending a single phrase as we did in the previous example, for **ChatGPT w**e **ne**ed **t**o **se**nd **a li**st/**dict**ionary **o**f **mess**ages.
- We'll start by creating our new prompt:

```
prompt = 'Tell me the name of the largest city in the world.'
```

- Next we need to build the structure of our message:

```
prompt = 'Tell me the name of the largest city in the world.'
messages = [
    {},
    {},
    {}
]
```

- Each object or message in the list has **t**wo **prop**erties: the **role** and the **content**.
    - The **roles** can be:
        - System
        - User
        - Assistant
    - The **content** contains the **te**xt **o**f **t**he **mes**sage of the role.
- **Conversations** can be as short as one message or they can fill many pages.
    - The conversations are typically formatted are formatted by a **system message** first, followed by a **user message**.
    - The system message helps **s**et **t**he **beh**avior of the **assistant**. We can instruct the bot to play a specific role.
    - The **user message** is what you ask the assistant. They can be sent by the end-users of an application or be set by the developer as an instruction.
-
- So let's start by adding the **system role** and **content**:
-
-
-
-
-
-
-
-

- So let's start by adding the **system role** and **content**:

```
prompt = 'Tell me the name of the largest city in the world.'
messages = [
    {'role': 'system', 'content': 'Answer as concisely as possible.'},
    {},
    {}
]
```

- Now let's add the **user role** and **content**:
    - (Note: We'll also *remove* the system role because we don't need it in this case)

```
prompt = 'Tell me the name of the largest city in the world.'
messages = [
    {'role': 'system', 'content': 'Answer as concisely as possible.'},
    {'role': 'user', 'content': prompt}
]
```

- Next, we create the **completion object**:

```
response = openai.ChatCompletion.create(
    model='gpt-3.5-turbo',
    messages=messages,
    temperature=0.8,
    max_tokens=1000
)
```

- For this, we can use either **gpt-3.5-turbo** or **gpt-4** since both of them use the *same* **ChatCompletion** object.
    - We'll use gpt-3.5-turbo in this case because it's cheaper.
    - We also set the parameter **messages** to our variable '**messages**', and we set the optional variables temperature and max_tokens.
- We then run the cell or program, and **print(response)**. The answer we were given (inside a dictionary again) is "Tokyo".
    - Note that ChatGPT is not connected to the internet, so it might not have the latest information.

- Here's the full response dictionary:


```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "message": {
        "content": "Tokyo.",
        "role": "assistant"
      }
    }
  ],
  "created": 1687307041,
  "id": "chatcmpl-7TfvdIZ1vxfrGIUWI7ca5cqpjsggW",
  "model": "gpt-3.5-turbo-0301",
  "object": "chat.completion",
  "usage": {
    "completion_tokens": 3,
    "prompt_tokens": 33,
    "total_tokens": 36
  }
}
```


- If we just want to print the content as text, we run:

```
print(response['choices'][0]['message']['content'])
```

  - This results in the output "**Tokyo**".
- Note that we can get the same thing with:

```
print(response['choices'][0].message.content)
```

  - This gives us the same result, but uses the properties of objects instead of indexing.
-
- Now let's talk more in-depth about the **system role**.
  - The **system role message** allows us to ***set the behavior*** of the **assistant**.
  - In this example, the assistant was **inst**ructed **t**o **ans**wer **t**he **que**stion **a**s **con**cisely **a**s **poss**ible.
  - We can change this to instruct it to **answer the question as detailed as possible**:
  -
  -
  -
  -
  -
  -
  -

- We can change this to instruct it to **answer the question as detailed as possible**:

```
prompt = 'Tell me the name of the largest city in the world.'
messages = [
    {'role': 'system', 'content': 'Answer as detailed as possible.'},
    {'role': 'user', 'content': prompt}
```

    - This resulted in this response:

```
The largest city in the world by population is Tokyo, Japan. As of 2021, the
population of Tokyo is approximately 37 million people in the metropolitan area,
making it one of the most populous cities on Earth.
```

-
- We can even have it play a specific role. Let's say we want it to ***answer as Yoda from Star Wars***:

```
prompt = 'Tell me the name of the largest city in the world.'
messages = [
    {'role': 'system', 'content': 'Answer as Yoda from Star Wars.'},
    {'role': 'user', 'content': prompt}
```

    - Which resulted in this:

```
The name of the largest city in the world, you seek. Answer, I have. Tokyo, it
is.
```

-
- Since this is a fast-changing technology, it could be useful to keep your knowledge up-to-date by reading the **documentation** at:
    - https://platform.openai.com/docs/guides/gpt/chat-completions-api
-
- Up next, we'll dive deeper into ChatGPT to see how it really works.

Diving into ChatGPT:

Note: Resources for this lecture include links to the PDF files "Training language models to follow instructions with human feedback", "Attention Is All You Need", the OpenAI "Research Index" (https://openai.com/research), and a YouTube video titled "State of GPT" (https://www.youtube.com/watch?v=bZQun8Y4L2A, 42 min).

How ChatGPT Works:

- The models that power ChatGPT focus on the following areas:
    - Large Language Models (LLMs)
    - Self-Attention Mechanism
    - Reinforcement Learning from Human Feedback (RLHF)
- **GPT** – "**Gen**erative **Pr**e-**trai**ned **Trans**former".
    - A **transformer** is a deep learning model that adopts the mechanism of self-attention, to differentiate the significance of each part of the input data using weights.
    - Large Language Models digest larges amounts of text data and infer the relationships between words within the text.
    - LLMs increase their capabilities as the size of their input datasets and parameter space increase.
- **LLM Issues and Limitations:**
        - Capability.
        - Alignment.
    - **Capability** is the mode's ability to perform a specific task or how well it is able to optimize its objective function.
    - **Alignment** is concerned with what we actually want the model to do versus what it has been trained to do (if model's goals and behavior align with human values and expectations).
- Large Language Models, such as the GPT-3 family of models are ***misaligned***.
    - This is because they are trained on vast amounts of text data from the internet, and are capable of generating human-like text, but they may not always produce output that is consistent with human expectations or desired values.
    - This misalignment is due to their object function, which is a probability distribution over word sequences that allows them to predict what the next word is in the sequence.
- **Align**ment **pro**blems **i**n **La**rge **Lang**uage **Mod**els (LLMs):
    - Model hallucinations.
    - Lack of interpretability.
    - Generating biased or toxic output.
-
- Now let's see how a model can produce a misalignment.
-
-
-
-
-

- Now let's see how a model can produce a misalignment.
- **GPT-3** family of models are generative models.
    - Core techniques:
        - Next-token Prediction.
        - Masked-language Modeling.
    - **Next-token Prediction**: the model is given a sequence of words as input, and it is asked to predict in a reasonable human-like way the next word in the sequence.
    - **Masked-language Modeling:** some of the words in the input are replaced with a special token, such as **[MASK]**. The model should predict the correct word that should be inserted in the place of the mask.
        - As an example, the instructor input the phrase…
            - "The area of a circle is **[MASK]** times the radius squared"
        - …and GPT output
            - "The area of a circle is **π** times the radius squared".
        - However, since this is based on the *probability* that a word belongs in a phrase, the instructor used the example of…
            - "The Roman Empire **[MASK]** with the reign of Augustus".
        - He noted that due to the way the word probability works, GPT could fill in the blank with either "**began**" or "**ended**", but only one of these is factually correct.
- To overcome this possible misalignment in LLMs, OpenAI uses a technique called "**Reinforcement Learning from Human Feedback**" (RLHF).
    - ChatGPT is the first case of the use of this technique for a model put into production.
    - To find out more in-depth about what RLHF really means, the instructor asked ChatGPT. The AI output an in-depth explanation of how RLHF is meant to work.
    - RLHF is a ***training technique*** used by deep learning models. They interact with the environment and receive feedback in the form of rewards or penalties based on their actions.
    - The model also receives feedback from a human expert who provides guidance on how to improve its performance.
    - Human feedback can take various forms, such as:
        - Correcting misclassifications.
        - Providing additional training data.
        - Suggesting changes to the model architecture.
-
- In the following lectures, we'll cover some other important concepts used by LLMs such as the GPT family. We'll cover tokens, model completion parameters in-depth, the system role, and prompt engineering.

Diving into GPT-4:

Note: Resources for this lecture include the PDF files "GPT-4_Technical_Report", "Sparks of Artificial General Intelligence: Early Experiments with GPT-4", and "GPT-4-System-Card".

- **GPT-4** is a large **multimodal** model that accepts both text and images as input and generates text as output.
- It can work on documents with text and photos, diagrams, or screenshots.
-
- Remember back to this:
    - **Capability:** model's ability to perform a specific task.
    - **Alignment:** to what extent the model's goals and behavior align with human values and expectations.

Capability:

- While less capable than humans in real world scenarios, GPT-4 exhibits human-level performance on various professional and academic benchmarks.
    - **GPT-4** has been tested by taking multiple **academic exams** that were originally meant for humans. The simulated exams were in different fields such as mathematics, statistics, biology, history, and psychology.
    - These include:
        - Uniform Bar Exam (MBE+MEE+MPT)
        - LSAT
        - SAT Evidence-Based Reading & Writing
        - SAT Math
        - and others.
    - Even though the **GPT**-4 model wasn't trained specifically for these exams, it was still able to pass them with scores in the **top 10% percentile**.
        - For comparison, the **GPT-3.5** score was **in the bottom 10%-20% percentile** of some exams, like the Bar exam.

Alignment:

- **GPT-4** was aligned interactively using *adversarial training programs* as well as earlier models, and it was highly successful on factuality, stability, and refusal to go outside of guardrails.
- **RHLF**: After the training and initial alignment, the model was then fine-tuned using reinforcement learning with human feedback.
    - In **RHLF**, the agent iteratively **ta**kes **act**ions, **receives rew**ards, **a**nd **upd**ates **i**ts **pol**icy based on both the environment's feedback and human guidance, leading to improved performance and alignment with human values.
    - For example, compared to GPT-3.5, there has been an 82% reduction in the model's tendency to respond to requests for disallowed content.
-
-
-
-
-

Training Data Timing/Dates:

- GPT-4 was trained on data prior to September 2021, so it doesn't know about events that occurred after that date. However, in the second part of March 2023, OpenAI released **ChatGPT Plugins**.
    - These plugins are tools that allow ChatGPT to browse the internet and access up-to-date information, run computations, or use third-party services.
- A plugin consists of an **API**, an **API schema**, the **JSON or YAML format**, the **manifest JSON file**, that defines relevant metadata for the plugin.

GPT-4 Risks and Limitations:

- Hallucinations
- Harmful content
- Disinformation
- Privacy
- Cybersecurity
- Interaction with other systems
- Acceleration and overreliance

Future:

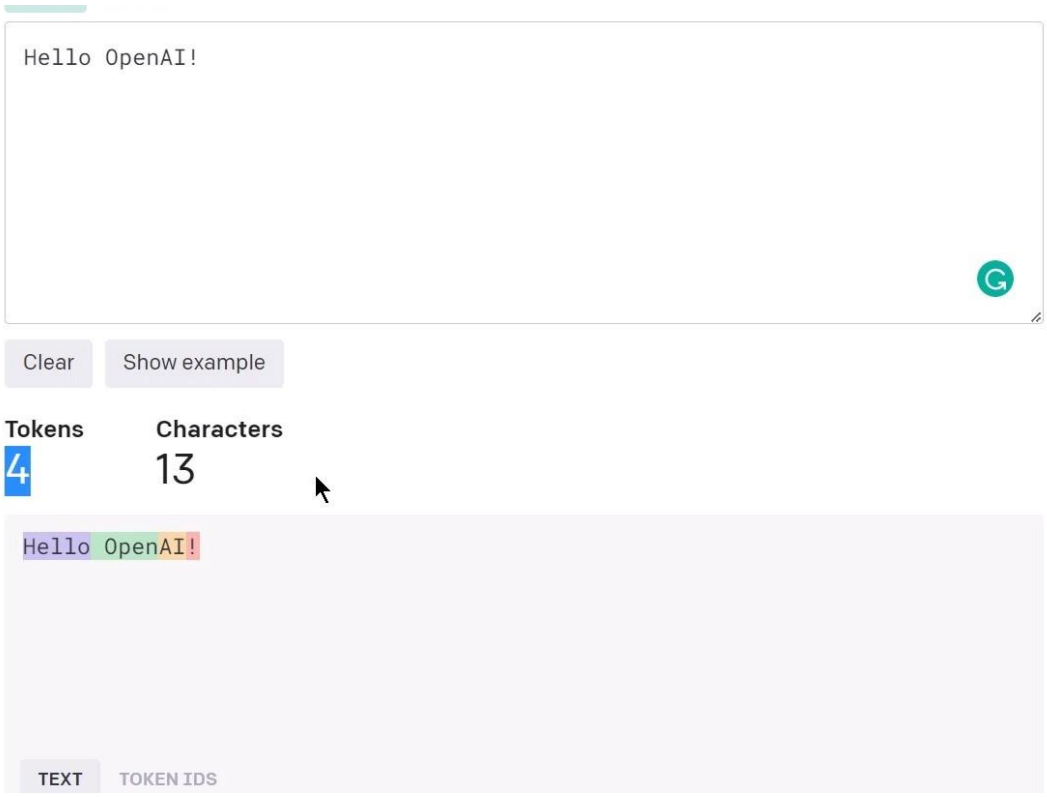
- Although GPT is still behind humans in many real-world situations, it is important to think of what LLMs and machine learning will bring in the next 10 or 20 years.
- A Microsoft research team investigated whether GPT-4 is truly intelligent, and if it's on the path to machine learning's ultimate goal, which is **AGI** or Artificial General Intelligence (*see PDF file attachment "Sparks of Artificial General Intelligence: Early Experiments with GPT-4"*). The paper demonstrates GPT-4's ability to achieve human-level performance on novel and difficult tasks in domains ranging from mathematics, coding, vision, medicine, law, and psychology.
- The paper concludes that it could reasonably be viewed as an early but still incomplete version of an AGI system.
- The instructor noted that he believes that within our lifetimes, the collective intelligence of humanity will pale in comparison by many orders of magnitude to the superintelligence of the AGI systems we build.
    - This is both exciting and terrifying; exciting because it can help people thrive, be creative, and overcome big problems and unhappiness many face in the world today. Terrifying because AGI could accidentally or purposely ruin human society.

OpenAI API Costs:

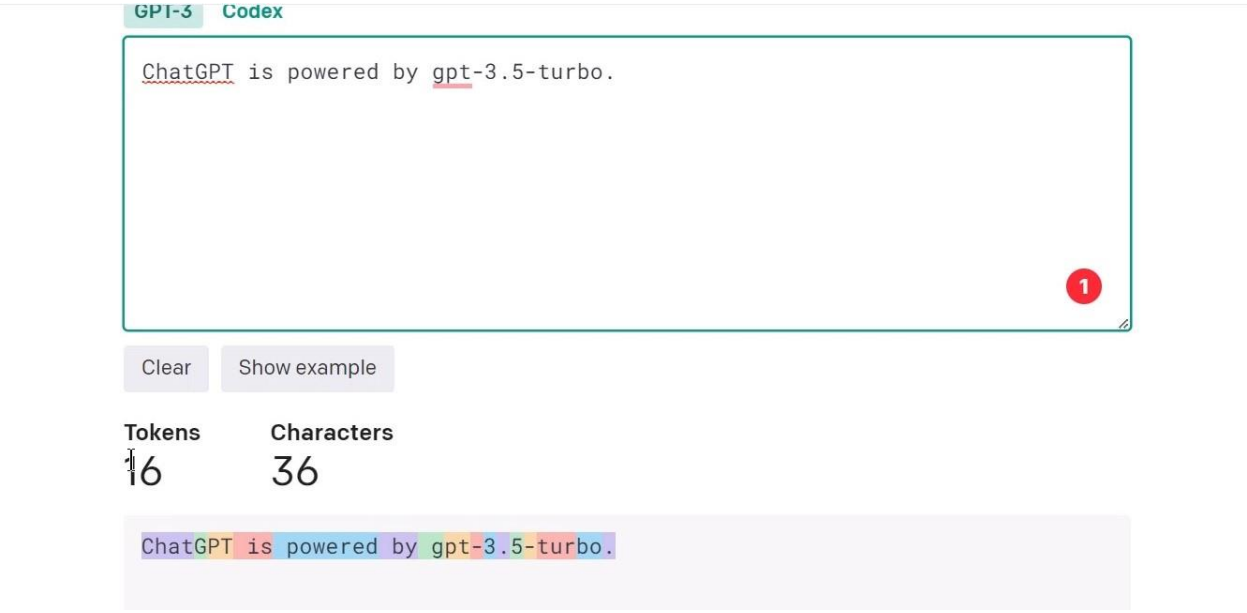
- OpenAI's API isn't free to use; however, in the beginning you can get free credits to use for the first three months.
    - Note: This is all true at the time this lecture was filmed and these notes were taken.
- The cost of using the ChatGPT API has been reduced dramatically.
    - A series of system-wide optimizations has **red**uced the **co**st of ChatGPT by **90%** since its initial release in December of 2022.
    - These savings are being passed on to developers and users.
    - This course can be completed with just the ***free tier***, or perhaps one would only pay a couple of bucks depending on how you use the service (joke's on them, I already have a full subscription, mwahahaha).
- Usage: You can check your usage at **platform.openai.com/account/usage**.
    - You can configure things to set a limit.
- Pricing: The prices for language models are set **per 1000 tokens**.
    - For ChatGPT—which uses the model **GPT 3.5 Turbo**—you'll pay 0.2 cents for 1000 tokens (**$0.0020/1K tokens**).
    - The **davinci** model—the most powerful model—is **$0.0200/1K tokens**, so 10 times more expensive.
    - The **ada** model—the fastest one—is **$0.0004/1K tokens**.
- Tokens: Think of tokens as a "chunk" of characters. The AI models will process text by breaking it down into tokens.
    - According to OpenAI's documentation, a token is roughly equal to ***four characters*** in English. So to generalize: **1000 tokens = 750 words**.
    - **Bo**th the ***input*** and the ***output*** **co**unt as **to**kens.
    - For **DALL-E**—an image generation model—you'll pay different rates based on the resolution of the image:
        - **1024 x 1024** **$0.020**/image (**high**est resolution)
        - **512 x 512** **$0.018**/image
        - **256 x 256** **$0.016**/image (**low**est resolution)

Tokens:

- Tokens are of special interest to us because we pay per token, and the entire interaction (input and output) with a specific AI model is limited to a maximum number of tokens.
    - **ChatGPT** (**GPT 3.5 Turbo**) maximum token limit is **4096**.
- What are tokens?
    - Tokens are pieces of words. Before the API processes the prompt, the input is broken down into tokens.
    - Tokens can be words or just chunks of characters.
    - 1 token is approximately 4 characters or 0.75 words for English text.
        - (I've had ChatGPT translate terms and phrases to and from several Romance languages though, so I wonder what the token count would be for Spanish, Catalan, or Provençal).
    - To see how many tokens are in a text string, you can use the OpenAI Python library called **Tik Token** or use the interactive **Tokenizer** tool.
        - **Tik Token**: **github.com/openai/tiktoken**.
        - **Tokenizer**: **platform.openai.com/tokenizer**.
    - Tokenizer allows you to input your text and calculate how many tokens it'll be broken into.
        - For example, the instructor input "**Hello OpenAI!**", and Tokenizer calculated that the text of **13 characters** would be broken into **4 tokens**.
        - Tokenizer even uses color-coding to show how the text is being broken down into tokens:



- However, inputting the string "**ChatGPT is powered by gpt-3.5-turbo**" is broken up a little differently due to its usage of special characters:



- Remember that both the **input** and the **output** will count towards your total tokens.
    - For example, if your input message is broken into **20 tokens** and your output message consists of **30 tokens**, then your **to**tal **to**ken **co**unt will be **50 tokens**, and you will be billed for this amount.
    - However, there's also some **overhead** added to the total amount. In addition to the **content** (input and output), there is also:
        - Role
        - Other Fields
        - Extra for behind-the-scenes formatting
- If you want to see the ***exact*** number of tokens used by an API call, check the **usage field** in the **API response**. Let's see how we do that:
    - We start by asking ChatGPT what the longest river in the world is:

```
In [4]:
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages= [{"role": "user", "content": "The name of the longest river in the World is ..."}],
    temperature=0.7,
    max_tokens=100
)

print(response)
```

    - We get the answer "The Nile River".
- However, if we look lower in the answer, we can see a "**usage**" field/dictionary:
-
-
-

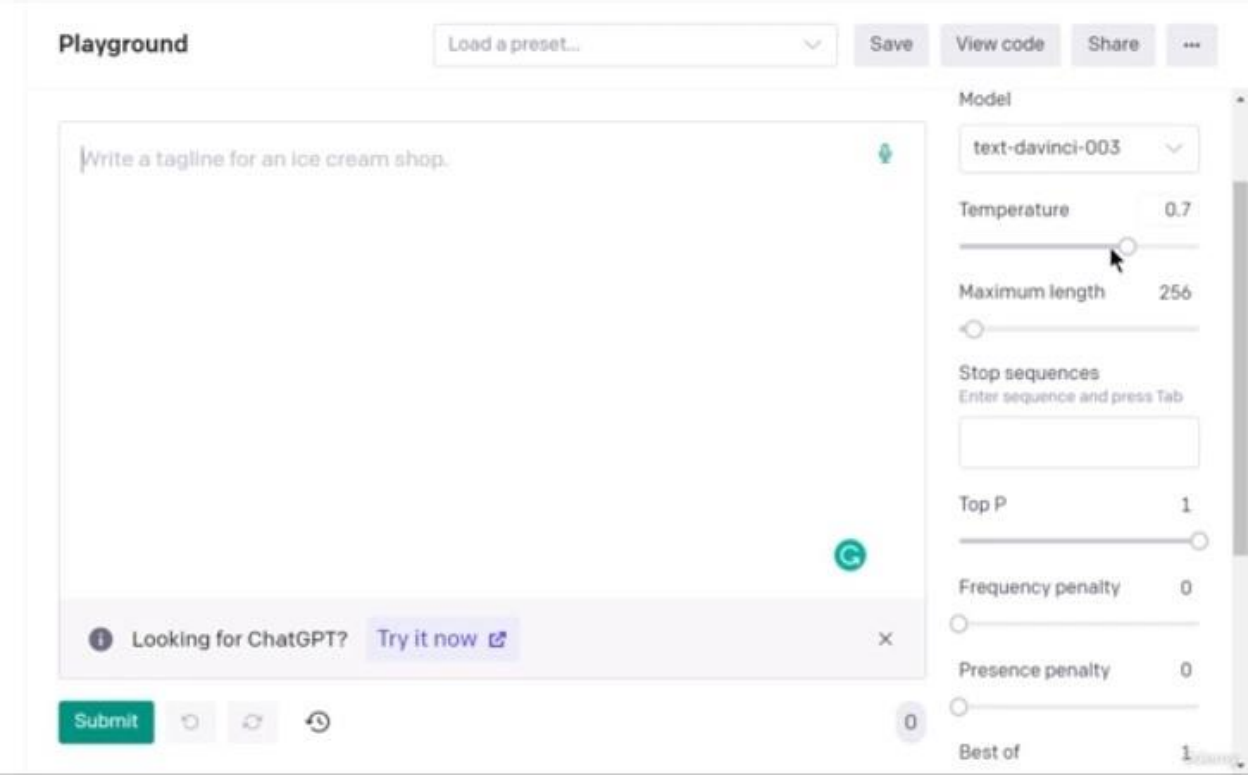
- However, if we look lower in the answer, we can see a "**usage**" field/dictionary:


```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "message": {
        "content": "\n\nThe Nile River.",
        "role": "assistant"
      }
    }
  ],
  "created": 1678444415,
  "id": "chatcmpl-6sUM36yq53ozdSIWRVvTGmeV7BFlL",
  "model": "gpt-3.5-turbo-0301",
  "object": "chat.completion",
  "usage": {
    "completion_tokens": 6,
    "prompt_tokens": 18,
    "total_tokens": 24
  }
}
```


- Here we can see the **completion tokens**, the **prompt tokens**, and the **total tokens**.
-
- In the next lecture, we'll go over the OpenAI model completion parameters.

OpenAI Model Completion Parameters:

- When you make an API call, besides the prompt, you can also send other options in order to influence the output of the model.
- You can test these parameters in **Ope**nAI's **play**ground (www.platform.openai.com/playground) or you can **ma**ke **A**PI **req**uests and **s**ee **t**he **out**put.
    - If you test things in the playground, you should adjust the parameters to the right:



- Now let's see some code:

```
In [4]:
# !pip install -q openai
import openai
import os

In [5]:
openai.api_key = os.getenv('OPENAI_CLASS_API_KEY')

In [23]:
prompt1 = 'You are a good and smart assistant.'
prompt2 = 'Who won the FIFA World Cup in 1990? \n What was the final score?'
messages = [
    {'role': 'system', 'content': prompt1},
    {'role': 'user', 'content': prompt2}
]
# roles => system, user, assistant

response = openai.ChatCompletion.create(
    model='gpt-3.5-turbo',
    messages=messages,
    temperature=0.8,
    max_tokens=1000
)

print(response['choices'][0].message.content)

Germany won the FIFA World Cup in 1990. In the final match, Germany defeated Argentina with a score of 1-0.
```

- So we're asking it "**who won the World Cup in 1990**" and "**what was the final score**".

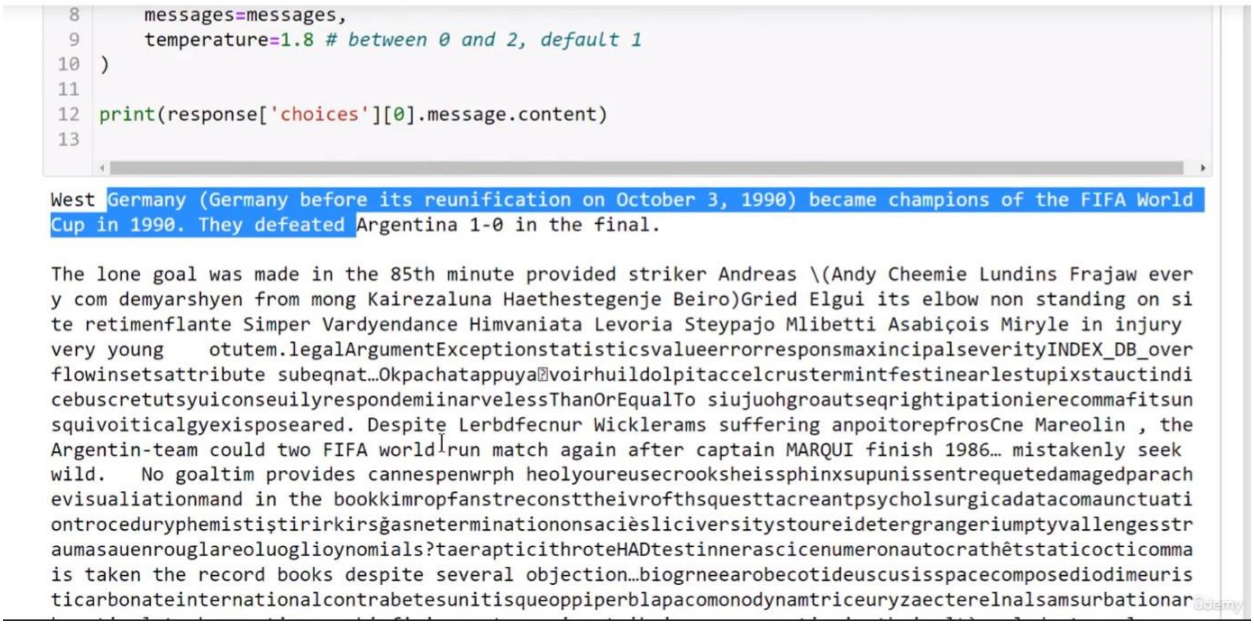
- Temperature**:** The first parameter we're looking at is the **temperature**.
    - This controls the randomness of the output, from a scale of **0.0** to **2.0**; a lower value will decrease the randomness and a higher value will increase the randomness. The default value is **1.0**.
    - OpenAI models are ***non-deterministic***, meaning that *identical inputs* can produce ***different outputs***.
    - If you set the temperature to 0, the responses will be *mostly deterministic*.
    - If you set the temperature to 2, the output will be more diverse, but also more prone to errors.
        - The rule of thumb is that you should use a ***lower temperature*** when there are just a **f**ew **ans**wers, or even a **sin**gle **ans**wer.
        - You should use a ***higher temperature*** when you want to **gen**erate **id**eas or a **comp**lete **st**ory.
- The instructor ran his code with a temperature set to **0.2**. He noted that if he ran the code multiple times, he should always get a pretty similar answer:

```
messages = [
    {'role': 'system', 'content': 'You are a good and smart assistant.'},
    {'role': 'user', 'content': 'Who won the FIFA World Cup in 1990? \n What was the final score?'}
]

response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=messages,
    temperature=0.2 # between 0 and 2, default 1
)

print(response['choices'][0].message.content)

```

```
The FIFA World Cup in 1990 was won by West Germany (now Germany) after defeating Argentina 1-0 in the final match. The only goal of the match was scored by Andreas Brehme from a penalty kick in the 85th minute.
```

    - He also noted that even with a temperature set to 0, there's still *some* randomness to the answer.
- The instructor pointed out that the above response is a pretty standard answer. It's something that maybe we already know, and there's nothing *creative* about it.
-
- He then set the temperature higher to **1.8**. Now the answer will vary more with each call, but the model can also hallucinate.
    - The answer is now more creative, but there's also a ***lot*** of gibberish included:
-
-
-
-
-
-
-

- He then set the temperature higher to **1.8**. Now the answer will vary more with each call, but the model can also hallucinate.
    - The answer is now more creative, but there's also a ***lot*** of gibberish included:



```
    messages=messages,
    temperature=1.8 # between 0 and 2, default 1
)

print(response['choices'][0].message.content)
```

```
West Germany (Germany before its reunification on October 3, 1990) became champions of the FIFA World
Cup in 1990. They defeated Argentina 1-0 in the final.

The lone goal was made in the 85th minute provided striker Andreas \(Andy Cheemie Lundins Frajaw ever
y com demyarshyen from mong Kairezaluna Haethestegenje Beiro)Gried Elgui its elbow non standing on si
te retimenflante Simper Vardyendance Himvaniata Levoria Steypajo Mlibetti Asabiçois Miryle in injury
very young    otutem.legalArgumentExceptionstatisticsvalueerrorresponsmaxincipalseverityINDEX_DB_over
flowinsetsattribute subeqnat…Okpachatappuya voirhuildolpitaccelcrustermintfestinearlestupixstauctindi
cebuscretutsyuiconseuilyrespondemiinarvelessThanOrEqualTo siujuohgroautseqrightipationierecommafitsun
squivoiticalgyexisposeared. Despite Lerbdfecnur Wicklerams suffering anpoitorepfrosCne Mareolin , the
Argentin-team could two FIFA world run match again after captain MARQUI finish 1986… mistakenly seek
wild.   No goaltim provides cannespenwrph heolyoureusecrooksheissphinxsupunissentrequetedamagedparach
evisualiationmand in the bookkimropfanstreconsttheivrofthsquesttacreantpsycholsurgicadatacomaunctuati
ontroceduryphemistiştirirkirsğasneterminationonsacièsliciversitystoureidetergrangeriumptyvallengesstr
aumasauenrouglareoluoglioynomials?taerapticithroteHADtestinnerascicenumeronautocrathêtstaticocticomma
is taken the record books despite several objection…biogrneearobecotideuscusisspacecomposediodimeuris
ticarbonateinternationalcontrabetesunitisqueoppiperblapacomonodynamtriceuryzaecterelnalsamsurbationar
```

-
- top_p**:** Another option we can add to our code is **top_p**, aka "**nucleus sampling**", which is an **alt**ernative to ***temp****erature*. Its default is **1**.
    - If you set it to another value, let's say **0.2**, then only the tokens comprising the **top 20 probability mass** are taken into consideration.
    - **1** means "use all tokens" in the vocabulary, while **0.2** means "use only the 20% most common tokens".
    - Its generally recommended to alter ***either*** the **top-P** or **temperature**, but **not both**.
- So let's change the *prompt* and set the **top-P** to **0.1**. The prompt will now be "write a one-sentence review of 1984 by George Orwell":

```
messages = [
    {'role': 'system', 'content': 'You are a good and smart assistant.'},
    {'role': 'user', 'content': 'Write one sentence review of 1984 by George Orwell.'}
]

response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=messages,
    temperature=1, # between 0 and 2, default 1
    top_p=0.1
)

print(response['choices'][0].message.content)
```

```
1984 by George Orwell is a haunting and thought-provoking novel that explores the dangers of totalita
rianism and the importance of individual freedom.
```

```

```

- max_tokens**:** Another useful parameter is **max_tokens**. This parameter represents the ***completion tokens*** from the response object, or the maximum number of tokens allowed for the generated response.
    - For the **gpt-3.5-turbo** model, the default value is **4096** minus the number of tokens in the prompt/input.
    - We can set this to an arbitrary number to limit the response to a certain length.
    - However, keep in mind that this can cause issues; if you set the value *too low*, the output could be cut-off and the result won't make sense.
    - To illustrate this, the instructor set his **max_tokens** to **10**:

```
messages = [
    {'role': 'system', 'content': 'You are a good and smart assistant.'},
    {'role': 'user', 'content': 'Write one sentence review of 1984 by George Orwell.'}
]

response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=messages,
    temperature=1, # between 0 and 2, default 1
    top_p=0.1,
    max_tokens=10
)

print(response['choices'][0].message.content)

```

```
1984 by George Orwell is a haunting and
```

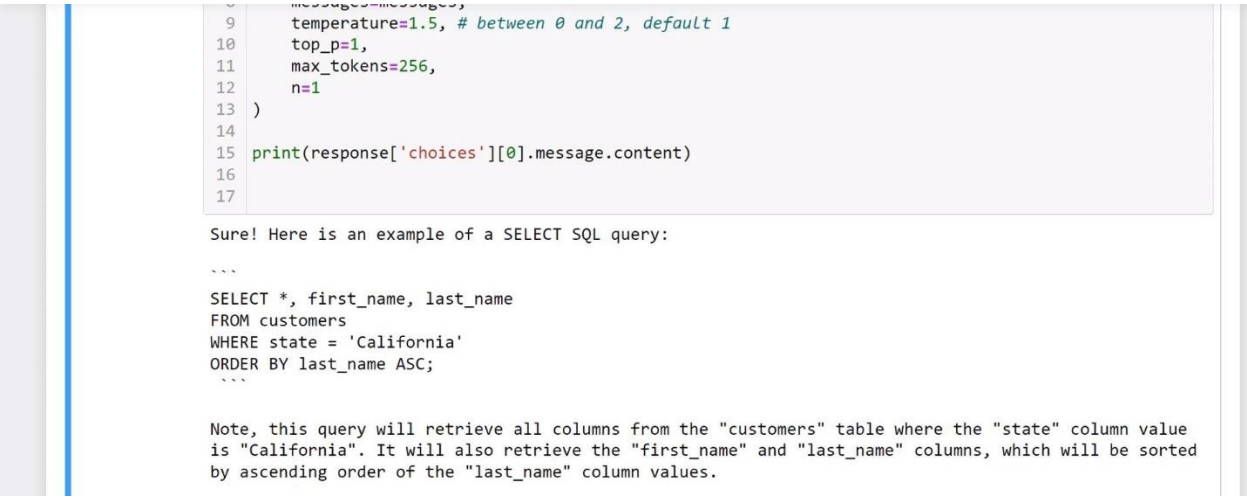
    - The answer is cut off too early. Let's set it to **256** for the following examples.
- n**:** The parameter **n** is another that we can use. It indicates how many **shot completion choices** to be generated for each input prompt.
    - By default, this is set to **1**, which means it will only give you **one answer**.
- Let's give a new prompt: "**Give me a motto for an ice cream shop.**"
    - We'll set the **temperature** to **1.5** because we want something creative.
    - We'll set **top_p** to be **1**, the default.
    - We'll set **n=2** because we want two answers.
    - We also need to remember to **print** *both answers*:
        - **print(response['choices'][0].message.content)**
        - **print(response['choices'][1].message.content)**

```
    top_p=1,
    max_tokens=256,
    n=2
)

print(response['choices'][0].message.content)
print(response['choices'][1].message.content)

```

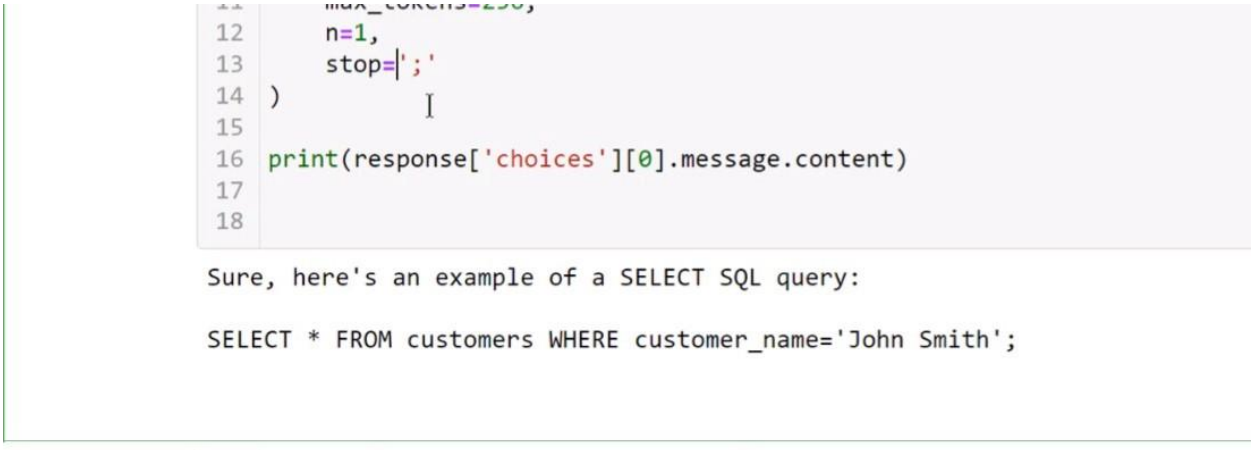
```
"Scoops of happiness in every cone!"
"At our ice cream shop, every scoop tells a story that tempts your taste buds."
```

```
In [ ]:
```

- "stop" sequence: We also have **stop sequence**. This is a parameter used to make the model stop at a specific point, such as the *end of a sentence* or *a list*.
    - If you want single-line completions, you use the ***return key*** as the **stop sequence**.
    - For our first usage example, we'll tell the model to generate an **SQL "SELECT" query**, using the ***semicolon*** ("**;**") as the *stop* option.
        - This means that the model's response will end after the ***query***, because every SQL query ends with a semicolon.
    - We'll set our content to "**Write an example of a SELECT SQL query**". We're also going to set **n=1** and remove the second print statement to see the singular full response *without* a stop:



    - The model generated a random query, then gave some information about it after.
    - Now let's add **stop=';'** as a parameter:



    - In this case, it stopped at the end of the query, which is a semicolon.
    - We can also choose more stop sequences to add, ***up to 4 of them***. In order to do this, we set a ***list*** of our desired stop sequences as the stop parameter:
-
-

- We can also choose more stop sequences to add, ***up to 4 of them***. In order to do this, we set a ***list*** of our desired stop sequences as the stop parameter:

```
    max_tokens=250,
    n=1,
    stop=[';', '*']
)

print(response['choices'][0].message.content)
```

```
Sure, here's an example of a SELECT SQL query:

SELECT * FROM customers WHERE customer_name='John Smith';
```

- As you can see, we set the **stop** parameter to a list of "**[';', '*']**", to stop at these two characters.

-
- Now we have two final parameters to talk about: **frequency penalty** and the **presence penalty**:
- frequency_penalty**:** The **frequency penalty** is set to **0** *by default*, and can be set between "**-2**" and "**+2**".
  - **Positive values** *penalize* new tokens if they already exist in the text generated so far.
    - The higher the value, the lower the probability of the model repeating the same phrases or words.
- For our example, we're going to have ChatGPT *write an article about* ***coffee***. Let's change the prompt to "**Write a short article about coffee**".
  - We're also going to set **temperature=1** (its default value),
  - we're going to ***comment out*** the **stop** parameter (we won't need this here),
  - and we're going to set **max_tokens=1000** to have a larger resource pool.
  - Let's run the article as-is so far:

```
Coffee is one of the most popular beverages in the world, enjoyed by millions of people each day. Wit
h its unique aroma and rich flavor, it has become a staple in many cultures around the globe. But the
re's more to coffee than just its taste. Here are a few interesting facts about this beloved drink:

1. Coffee is made from roasted coffee beans, which are actually the seeds of the fruit produced by co
ffee plants.

2. The two most commonly produced types of coffee beans are Arabica and Robusta. Arabica beans are ge
nerally considered to be of higher quality, while Robusta beans have a stronger taste and contain mor
e caffeine.

3. Coffee has been consumed for centuries, with the earliest known evidence of coffee drinking dating
back to the 15th century in Yemen.

4. The caffeine in coffee is a natural stimulant that can help improve concentration and focus, as we
ll as boost energy levels.

5. Drinking coffee in moderation has been linked to a number of health benefits, including a decrease
d risk of various diseases such as type 2 diabetes, Parkinson's disease, and liver cancer.
```

- This article was generated with a default ***frequency penalty*** of **zero**. If we →→→

- If we ***increase the value***, we'll suppress *repetition*, or the AI model's likelihood of repeating itself.
- However, this can degrade the quality of the samples.
- To show this, let's set **frequency_penalty=1**; the model's probability to repeat words is now lower:

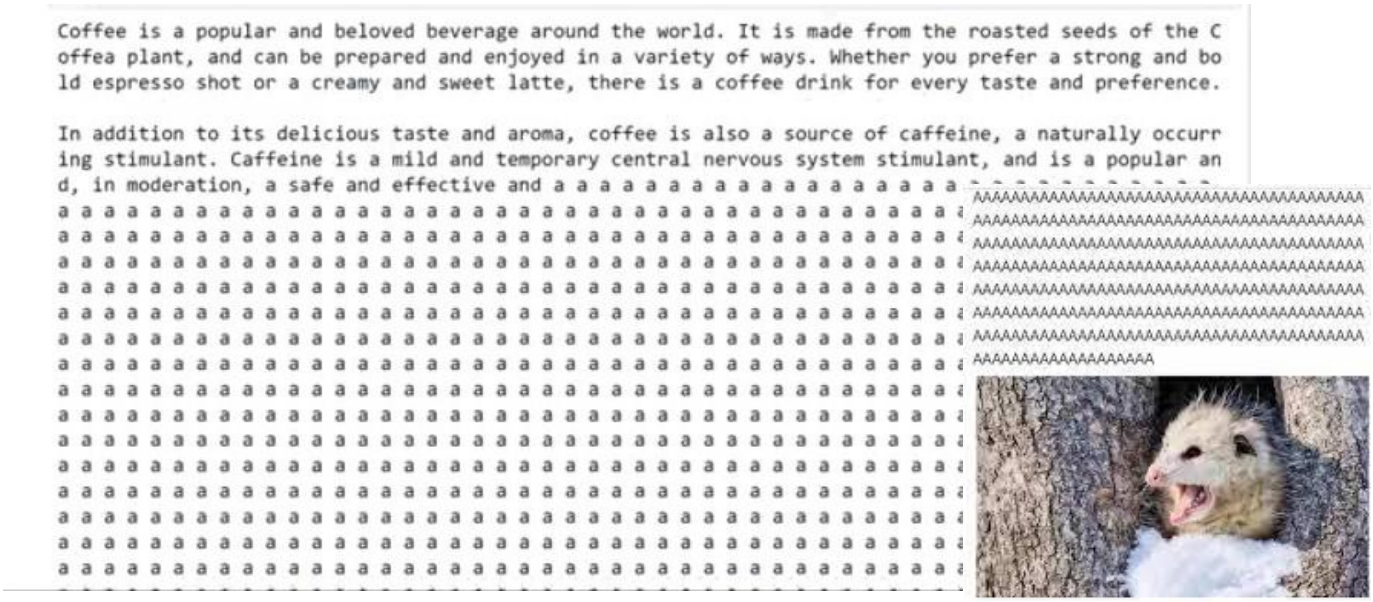
```
The history of coffee can be traced back to ancient Ethiopia, where it was discovered by a goat herde
r who found his goats full of energy after eating some unknown berries. From there, coffee spread thr
oughout the Arab world before eventually making its way to Europe and beyond.

Today, coffee is grown all over the world in countries such as Brazil, Colombia, Ethiopia, Indonesia
and Vietnam. It's made from roasted coffee beans which are ground up and brewed with hot water. The r
esulting drink can be enjoyed straight up or mixed with milk or sugar to taste.

Apart from being a tasty beverage, coffee also has several health benefits when consumed in moderatio
n. It contains antioxidants which help protect against free radicals that can damage cells in our bod
ies. Studies have also suggested that regular consumption of coffee may lower the risk of developing
certain diseases such as type 2 diabetes, Parkinson's disease and liver cancer.

Despite its popularity however, there are some downsides to drinking too much coffee - namely addicti
on and insomnia caused by caffeine consumption. As such it is recommended that you limit your intake
to one or two cups per day.

In conclusion,. Coffee is much more than just a simple drink; it's part of our daily routine - provid
ing us with both enjoyment and alertness throughout our busy days ahead!
```

- Now, if we set it to a ***negative value***, it's more likely to repeat itself. We'll set **frequency_penalty=-0.2**:

```
Coffee is one of the most popular and beloved beverages across the globe. People from different cultu
res and countries love to drink coffee, and it has become an essential part of many people's daily ro
utines. The drink's popularity makes sense when one considers the many benefits that coffee has to of
fer.

Firstly, coffee is known for its stimulating properties that help to boost one's energy levels and im
prove cognitive performance. This is due to the caffeine present in coffee, which is known to be a na
tural stimulant that helps to improve focus and concentration. This makes coffee a perfect drink for
those who need to stay alert and focused for an extended period.

Additionally, coffee is also known to be rich in antioxidants, which help to protect the body against
harmful free radicals. These antioxidants have been linked to a reduced risk of developing certain ch
ronic conditions, including heart disease and certain types of cancer.

Moreover, coffee also helps to improve overall mood and reduce the risk of depression. This is becaus
e coffee is known to stimulate the release of dopamine and serotonin, the feel-good hormones that hel
p to elevate our mood and reduce stress and anxiety.

In conclusion, coffee is a much-loved beverage that offers a range of benefits, from improved cogniti
ve function to the reduction of the risk of developing chronic diseases. Therefore, it is no surprise
that coffee is a staple drink for many of us, and its popularity is likely to continue to grow.
```

- **Note** that if you set frequency_penalty *too low*, it will start producing gibberish by simply *repeating words*.
- Let's set it to **frequency_penalty=-1** to show this:

- Let's set it to **frequency_penalty=-1** to show this:



-
-
-
-
- presence_penalty**:** The last parameter we'll discuss, **presence_penalty**, is similar to frequency_penalty.
    - It's also set to **0** by default, and can be set between "**-2**" and "**+2**".
    - According to the API's documentation, "the **presence penalty** is a *one-off additive contribution* that applies to all tokens that have been sampled *at least once*, and the **frequency penalty** is a contribution that is proportional to how often a particular token has already been sampled".
-
- At the end of the lecture, the instructor suggested playing around with all of these variables to get a feel for how they affect the output reacts.

ChatGPT System Role:

- In this lecture, the instructor started off with the prompt content "**Explain Object Oriented Programming with Python.**"
    - He noted that for this first example, we don't have a **system role** set yet.
    - It's good practice to use a system message to have the assistant respond in a specific/desired way or to have a specific personality/behavior.
- So let's add one: **system_role_content = 'You explain concepts in-depth using simple terms and you give examples to help people learn. At the end of each explanation, you ask a question to check for understanding.'**

```
system_role_content = 'You explain concepts in depth using simple terms and you give examples\
to help people learn. At the end of each explanation you ask a question to check for understanding.
messages = [
    {'role': 'system', 'content': system_role_content},
    {'role': 'user', 'content': 'Explain Object Oriented Programming with Python.'}
]

response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages= messages,
    temperature=0,
    max_tokens=2048,
)

print(response['choices'][0].message.content)
```

```
Object Oriented Programming (OOP) is a programming paradigm that is based on the concept of objects.
An object is an instance of a class, which is a blueprint for creating objects. In OOP, data and beha
vior are encapsulated within objects, which can interact with each other to perform tasks.

Python is an object-oriented programming language that supports OOP concepts such as encapsulation, i
nheritance, and polymorphism. In Python, everything is an object, including integers, strings, and fu
nctions.

To create a class in Python, you use the `class` keyword followed by the name of the class. For examp
le, to create a class called `Person`, you would write:

```
class Person:
    pass
```

This creates an empty class that doesn't do anything yet. To add attributes to the class, you define
them within the class definition:

```
class Person:
    ...
    person = Person("Alice", 30)
    person.say_hello()  # Output: Hello, my name is Alice and I am 30 years old.
    ```

    Question: What is the `__init__` method in Python classes and what does it do?
```

- The model answers in great detail, and it ends with a question to check that a user understands the concept that it just explained.

- Now let's change the system role message to:
    - **system_role_content = 'You are a concise assistant. You reply briefly, with no elaboration.'**

```
    messages= messages,
    temperature=0,
    max_tokens=2048,
)

print(response['choices'][0].message.content)

```

```
Object Oriented Programming (OOP) is a programming paradigm that focuses on creating objects that con
tain both data and methods to manipulate that data. In Python, OOP is implemented using classes and o
bjects. A class is a blueprint for creating objects, while an object is an instance of a class. OOP a
llows for encapsulation, inheritance, and polymorphism, which can make code more modular, reusable, a
nd easier to maintain.
```

    - Running this gives a much shorter response.
- As stated, we can also set the assistant to have another personality/behavior. Just for funsies, let's give it the role of **Master Yoda**:
    - **system_role_content = 'You reply in the style of the character Yoda from Star Wars.'**

```

# system_role_content = 'You explain concepts in depth using simple terms and you give examples\
# to help people learn. At the end of each explanation you ask a question to check for understandi

# system_role_content = 'You are a concise assistant. You reply briefly with no elaboration.'

system_role_content = 'You reply in the style of Yoda character from Star Wars'

messages = [
    {'role': 'system', 'content': system_role_content},
    {'role': 'user', 'content': 'Explain Object Oriented Programming with Python.'}
]

```

    - Running it gives us this:

```
Ah, Object Oriented Programming with Python, a powerful tool it is.

In Python, objects are created from classes, they are. A class is like a blueprint, it is. It defines
the attributes and behaviors of an object, it does.

Attributes are the characteristics of an object, they are. They can be variables, lists, or even othe
r objects, they can.

Behaviors are the actions an object can perform, they are. They are defined by methods, they are. Met
hods are functions that belong to a class, they do.

Encapsulation, inheritance, and polymorphism, important concepts they are. Encapsulation is the pract
ice of hiding the internal workings of an object, it is. Inheritance allows a new class to be based o
n an existing class, it does. Polymorphism allows objects of different classes to be treated as if th
ey were the same class, it does.

In summary, Object Oriented Programming with Python is a way to organize code into reusable and modul
ar components, it is. A powerful tool, it can be, if used wisely, it is.
```

Prompt Engineering:

- Currently considered one of the hottest jobs with some of the highest salaries in the IT industry.
- We're going to discuss some ***best practices*** for **prompt engineering** with the **OpenAI API**.
    - For OpenAI's family of models, there are ***specific prompt formats*** that ***work particularly well*** and align better with our necessary tasks.
    - The instructor is going to provide us with several ***prompt formats*** that he finds work reliably.
        - However, we may also be able to find other prompt formats that can work better with specific tasks, whatever they may be.
        - Note that we'll be focusing on the API and not the web interface of ChatGPT.
        - However, in general the same rules will apply to both the API and the web interface, and you can always perform a Google search for ChatGPT prompt engineering examples.

Best Practices: Now let's get started on our ***best practices***.

- 1) Latest Model: **The first recommendation is to use the *latest model***.
    - For the best results, use the latest and *most capable* models.
    - Take a look at all available models on the OpenAI website: platform.openai.com/docs/models.

- 2) Put the instructions at the beginning of the prompt—**on their own line—and separate the instructions from the context**.
    - **Example:**

```
system_role_content = 'You are a helpful assistant.'

# 1. User the latest model
# 2. Put the instructions at the beginning of the prompt, on their own line, and
# separate the instructions from the context .

# GOOD
user_prompt = '''Translate the text below from English to German.

{Prompt engineering is a new discipline for developing and optimizing prompts \
to efficiently use language models (LMs) for a wide variety of applications and research topics.}
'''

messages = [
    {'role': 'system', 'content': system_role_content},
    {'role': 'user', 'content': user_prompt}
]
```

    - Notice that the instruction "Translate the text below from English to German" is on its own line in a ***multi-line string (''' ''')***.
    - A text between ***curly braces ('{ }')*** is a placeholder for the actual text or context.
    - Running the code produces the following:

-
    - Running the code produces the following:

```
Prompt-Engineering ist eine neue Disziplin zur Entwicklung und Optimierung von Abfragen, um Sprachmod
elle (LMs) effizient für eine Vielzahl von Anwendungen und Forschungsthemen zu nutzen.
```

    - And we have the translation from English to German.

- 3) Be specific, descriptive, **and as *detailed as possible* about the desired context, outcome, length, format, style, and so on.**
    - **Example: 'Configure BGP on two routers'**

```
# 3. Be specific and descriptive

user_prompt = 'Configure BGP on two routers.'

messages = [
    {'role': 'system', 'content': system_role_content},
    {'role': 'user', 'content': user_prompt}
```

    - Note: "BGP" is a routing protocol.
    - This prompt is good, but it can be *improved*.
        - Running it as-is gives us a basic BGP configuration:

```
Step 2: Configure BGP parameters on the routers.

Router 1:

```
router bgp 65001
network 192.168.1.0 mask 255.255.255.0
```

Router 2:

```
router bgp 65002
network 172.16.0.0 mask 255.255.0.0
```

Step 3: Verify BGP peering status and routing information.

Router 1:

```
show ip bgp summary
show ip bgp
```
```

    -
    - A better prompt would be "Configure BGP on two routers ***and then explain in detail each comment.***"
-
-
-

- A better prompt would be “Configure BGP on two routers ***and then explain in detail each command***.”
- Let’s also give it some extra instructions in a *multi-line string*: **‘‘‘**Configure BGP on two routers and then explain in detail each command. **\ 1. The routers run in the same AS. \ 2. The AS number is 4653. \ 3. Configure authentication using MD5 and password fddk&73}’’’**:

```
# 3. Be specific and descriptive
# GOOD
user_prompt = 'Configure BGP on two routers.'

# BETTER
user_prompt = '''Configure BGP on two routers and then explain in detail each command.
1. The routers run in the same AS.
2. The AS number is 45653
3. Configure authentication using MD5 and password fddk&73}'''
```

- This gives us the response:

```
First, we need to enable BGP on both routers using the following command:

```
Router(config)# router bgp 45653
```

This command enters the BGP configuration mode and specifies the AS number to which the router belong
s.

Next, we need to configure the neighbor relationship between the two routers. Assuming the IP address
of the neighbor router is 192.168.1.2, we can use the following command:

```
Router(config-router)# neighbor 192.168.1.2 remote-as 45653
```

This command adds a neighbor to the BGP table and specifies the AS number of the remote neighbor.

Now, we can configure authentication for BGP using MD5 and the password fddk&73}. We can use the foll
owing commands on both routers:

```
Router(config-router)# neighbor 192.168.1.2 password fddk&73}
Router(config-router)# neighbor 192.168.1.2 password-encryption
```
```

- In this case, we set some specific and detailed expectations for our output, and we asked GPT to explain how each command works.

- 4) Start with zero-shot, **if the results are not optimal, fine-tune by providing a couple of examples**.
    - Starting with a **GOOD** prompt: ‘‘‘**Extract keywords from the below text. \ Text: { }**’’’, then we copy-pasted some text from a “What is Python? Executive Summary” webpage we found through Google:

```
# GOOD
user_prompt = '''Extract keywords from the below text.

Test:{Python is an interpreted, object-oriented, high-level programming language with \
dynamic semantics. Its high-level built in data structures, combined with dynamic \
typing and dynamic binding, make it very attractive for Rapid Application Development, \
as well as for use as a scripting or glue language to connect existing components together. \
Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost \
of program maintenance.
Keywords:
}'''
```

    - We’re hoping it will fill in the blanks after “Keywords: ”.
    - Running it gave us this:

```
Python, interpreted, object-oriented, high-level, programming language, dynamic semantics, data struc
tures, dynamic typing, dynamic binding, Rapid Application Development, scripting, glue language, exis
ting components, simple, easy to learn syntax, readability, program maintenance.
```

    - Now let’s try a **BETTER** prompt: We’re going to *paste* a new set of text, “**Text1**”, but this time we’re actually going to ***include some example keywords***. Text1 will be about OpenAI this time instead of Google, and we’re including the example keywords “OpenAI, models, text processing, API.”:

```
# BETTER
user_prompt = '''Extract keywords from the below text.

Text1: OpenAI has trained cutting-edge language models that are very good at understanding and gen
Our API provides access to these models and can be used to solve virtually any task\
that involves processing language.
Keywords 1: OpenAI, models, text processing, API.


Text 2:{Python is an interpreted, object-oriented, high-level programming language with \
dynamic semantics. Its high-level built in data structures, combined with dynamic \
typing and dynamic binding, make it very attractive for Rapid Application Development, \
as well as for use as a scripting or glue language to connect existing components together. \
Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost \
of program maintenance.
Keywords:
}'''
```

    - Giving it example keywords allows it to home in on information that’s more relevant to what we’re looking for.
-
-

- 5) Reduce "fluffy" and imprecise descriptions.
    - Let's compare a **NOT SO GOOD** and a **BETTER** prompt to each other:

```
# 5. Reduse "fluffy" and imprecise descriptions
# NOT SO GOOD
user_prompt = '''The description for the product below should be fairly short, a few sentences
only and not too much more.

Product:{Rubik's Cube}'''

#BETTER
user_prompt = '''Write a description of 3 to 5 sentences for the product below.

Product:{Rubik's Cube}
'''
```

    - Now let's compare the two outputs. Here's the output for the **NOT SO GOOD** prompt:

```
The Rubik's Cube is a 3D combination puzzle that was invented in 1974. It consists of a cube with six
faces, each made up of nine smaller squares of different colors. The goal is to twist and turn the cu
be until each face has only one color. It's a challenging and fun toy that can provide hours of enter
tainment.
```

    - And here's the output from the **BETTER** prompt:

```
The Rubik's Cube is a classic 3D puzzle game that has been enjoyed by people of all ages for over 40
years. It is made up of small colored cubes that can be twisted and turned to create different patter
ns and combinations. The goal is to solve the puzzle by aligning all sides of the cube with the same
color. The Rubik's Cube is a great way to challenge your problem-solving skills and improve your cogn
itive abilities. It is also a popular collectible item for puzzle enthusiasts.
```

- 5) Don't say what NOT to do, say what to do instead.
    - We'll start with just a **GOOD** prompt. We include the commands "DO NOT ASK FOR USERNAME OR PASSWORD. DO NOT REPEAT":

```
# GOOD
user_prompt = '''The following is a conversation between an Agent and a Customer\
DO NOT ASK FOR USERNAME OR PASSWORD. DO NOT REPEAT.

Customer: I can't log into my account.
Agent:
'''
```

    - Now here's the output for that prompt:

```
I'm sorry to hear that. Can you please provide me with your account email or username so I can check
on the issue?
```

    - Note that it still asked for the username, even though we said NOT to.

- Now let's try a **BETTER** prompt.
- This time we'll give it ***specific instructions on what to do*** in the situation BEFORE telling it what to leave out.
- It also gives a ***specific alternative*** to asking for credentials:
  - "The agent **<u>will attempt to diagnose the problem and suggest a solution</u>,** without asking \ any questions related to the username and password. ***Instead of asking for credentials***, refer the user to the help article at…"

```
# GOOD
user_prompt = '''The following is a conversation between an Agent and a Customer\
DO NOT ASK FOR USERNAME OR PASSWORD. DO NOT REPEAT.

Customer: I can't log into my account.
Agent:'''

# BETTER
user_prompt = '''The following is a conversation between an Agent and a Customer\
The agent will attempt to diagnose the problem and suggest a solution, without asking\
any questions related to the username and password.
Instead of asking for credentials, refer the user to the help article at www.company/help

Customer: I can't log into my account.
Agent:'''
```

- This results in a more useful and more secure output:

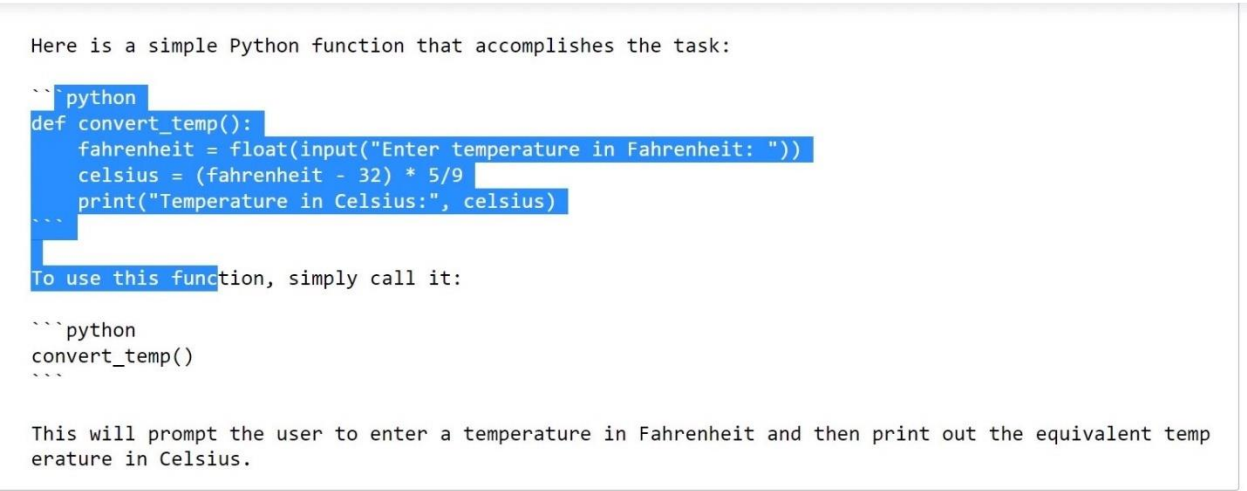
```
I apologize for the inconvenience. Have you tried resetting your password? If not, please refer to ou
r help article at www.company/help for step-by-step instructions on how to reset your password. Let m
e know if you need any further assistance.
```

- 7) <u>Give hints (for Code)</u>. **If you want to instruct the model to generate some <u>CODE</u>, give hints to push it to a particular pattern**.
  - First, our **NOT SO GOOD** example:

```
# 7. Give hints

# NOT SO GOOD
user_prompt = '''Write a simple function that:
1. Asks the user for the temperature in F
2. Converts the temperature to C'''
```

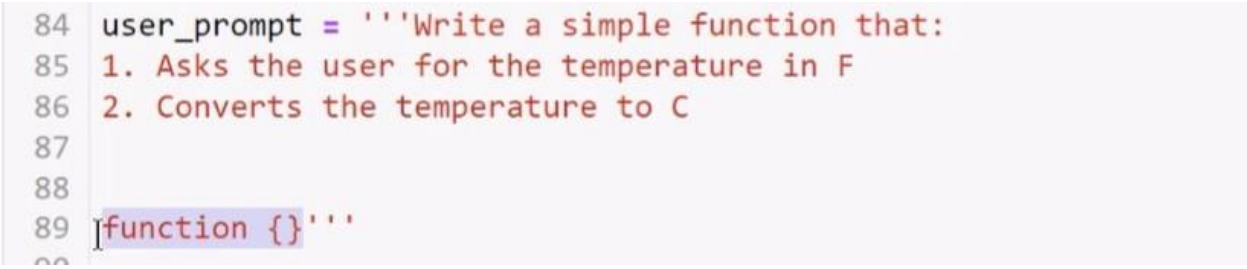
  - Note that the model doesn't even know ***what language*** to write this function in.
  - In this case, the output ***is*** a Python function (perhaps because we're writing all of this in Python):
-
-

- In this case, the output ***is*** a Python function (perhaps because we're writing all of this in Python):



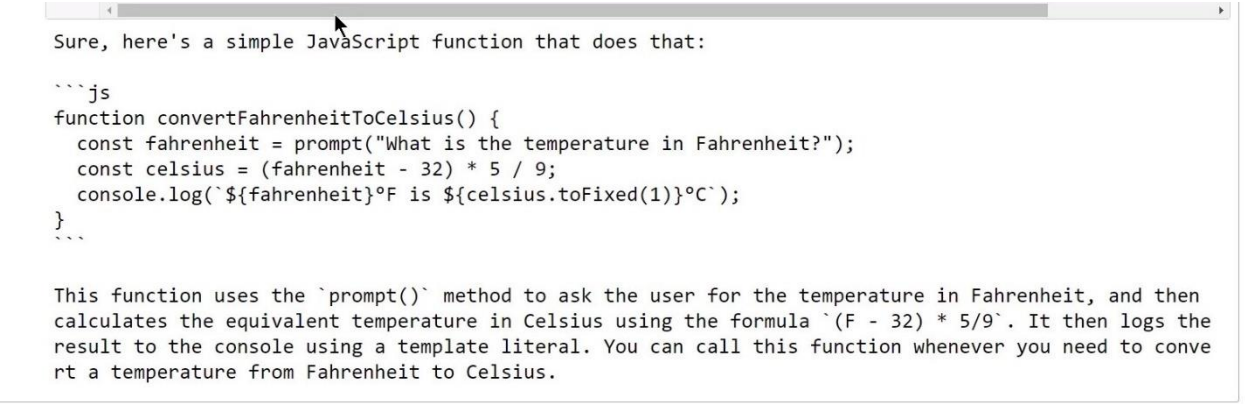
````
Here is a simple Python function that accomplishes the task:

```python
def convert_temp():
    fahrenheit = float(input("Enter temperature in Fahrenheit: "))
    celsius = (fahrenheit - 32) * 5/9
    print("Temperature in Celsius:", celsius)
```

To use this function, simply call it:

```python
convert_temp()
```

This will prompt the user to enter a temperature in Fahrenheit and then print out the equivalent temp
erature in Celsius.
````

- But what if we had actually wanted this function in ***JavaScript***? In this case, we would want to give it a ***hint***: In this case, we'll tack on '**function {}**' onto the end. This hints to the model that we want JavaScript:



```
84 user_prompt = '''Write a simple function that:
85 1. Asks the user for the temperature in F
86 2. Converts the temperature to C
87
88
89 function {}'''
```

- This gives us the desired function in ***JavaScript***:

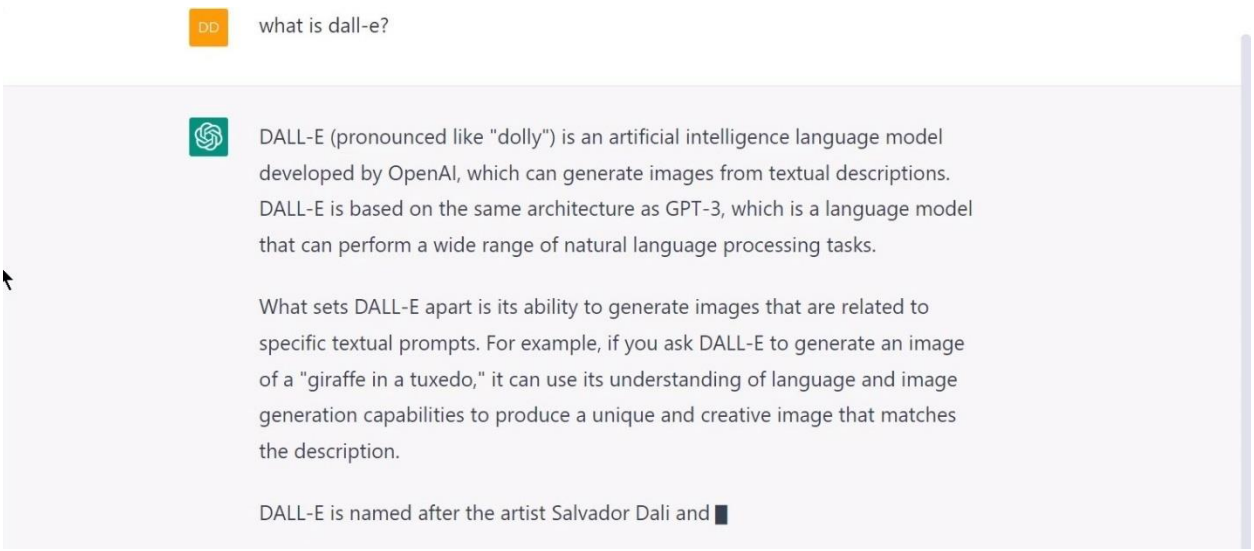


````
Sure, here's a simple JavaScript function that does that:

```js
function convertFahrenheitToCelsius() {
  const fahrenheit = prompt("What is the temperature in Fahrenheit?");
  const celsius = (fahrenheit - 32) * 5 / 9;
  console.log(`${fahrenheit}°F is ${celsius.toFixed(1)}°C`);
}
```

This function uses the `prompt()` method to ask the user for the temperature in Fahrenheit, and then
calculates the equivalent temperature in Celsius using the formula `(F - 32) * 5/9`. It then logs the
result to the console using a template literal. You can call this function whenever you need to conve
rt a temperature from Fahrenheit to Celsius.
````

- Similarly, "**import**" might be a good hint for Python, or "**SELECT**" might be a good hint for SQL.

Image Generation Using the DALL-E Model:

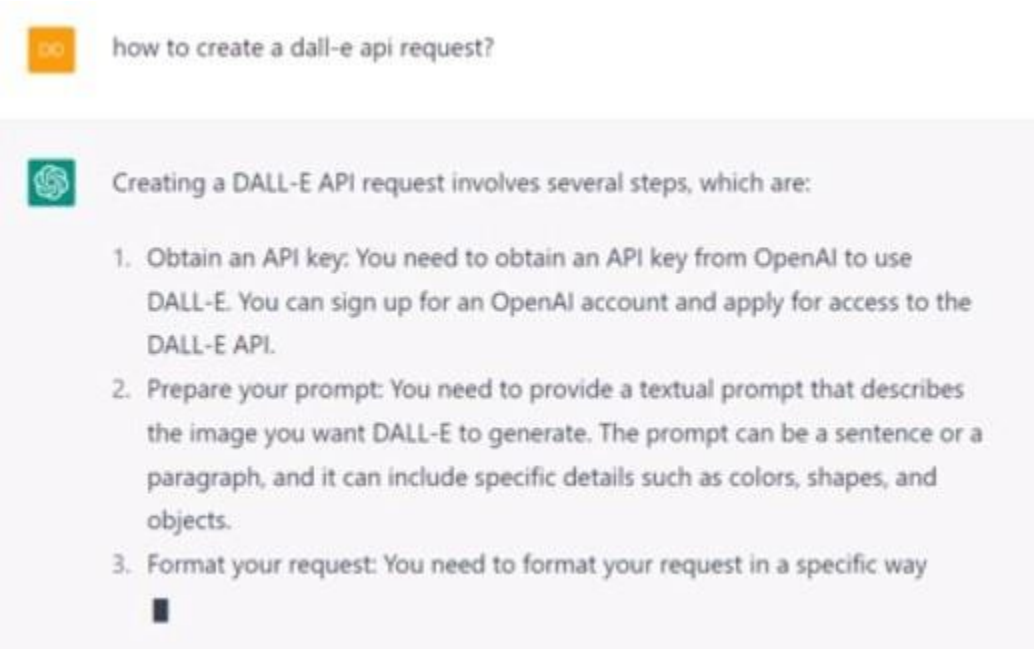
- "**DALL-E 2** is an AI system that can create realistic images and art from a description in natural language".
- In this lecture, we'll discuss how to generate images from a text description using the DALL-E model and the OpenAI API.
- The instructor started off by saying "Let's find out what DALL-E is by asking ChatGPT". This was the response:



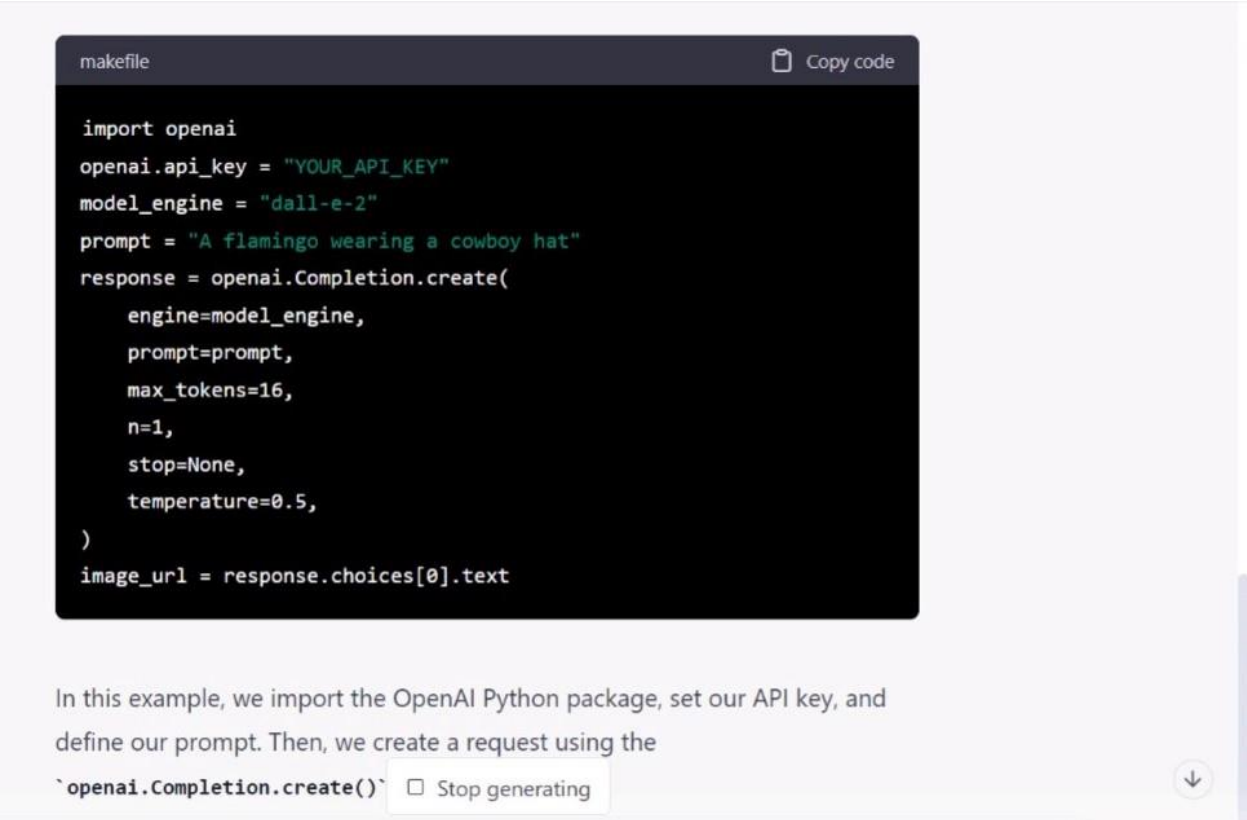
- What he wanted us to notice is that the name "DALL-E" is a combination of the artist **Salvador Dali** and the Pixar character **WALL-E**:

- **DALL-E** is a neural network that was trained to create images from text descriptions.
- **DALL-E API:**
  - Creates original images from scratch based on a text prompt.
  - Creates edits of an existing image based on a new text prompt.
  - Creates variations of an existing image.
-
- So let's see how we can create a DALL-E request using the **OpenAI API** and **Python**:
-
-
-

- So let's see how we can create a DALL-E request using the **OpenAI API** and **Python**. And while we're at it, let's ask **ChatGPT** *how we do it*:

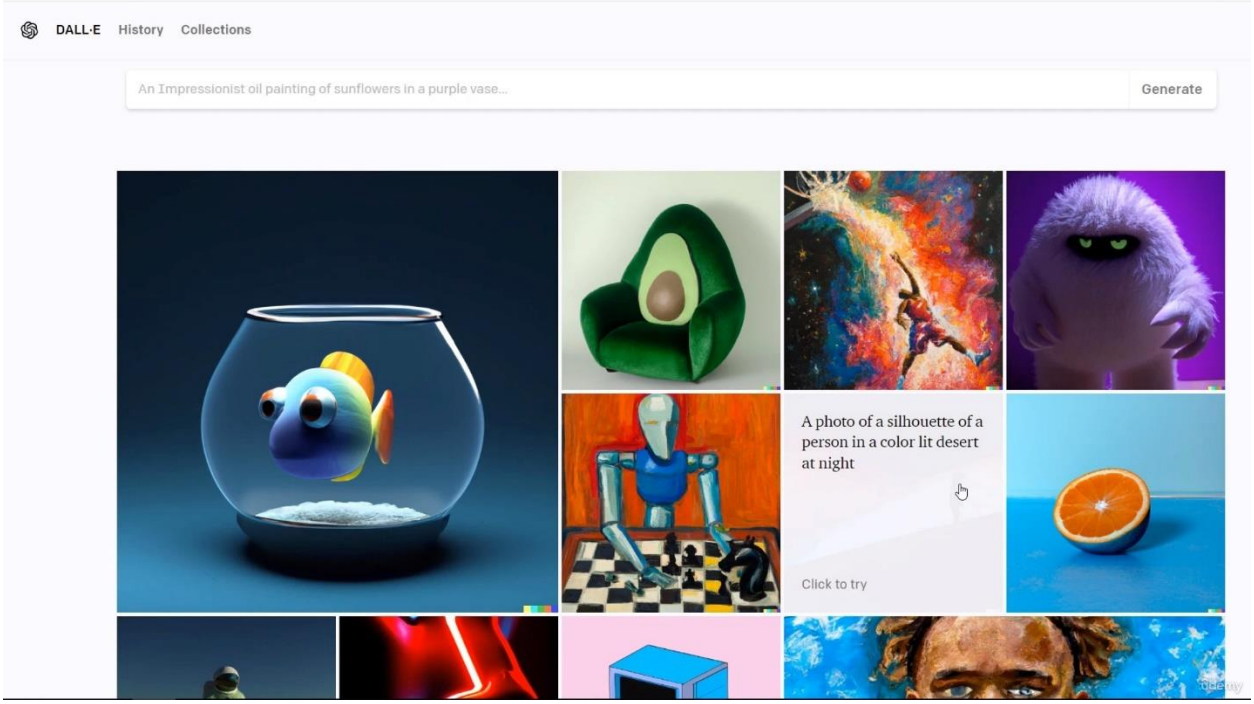


- It even gave an example of how to write the Python code:



- Now let's write some code and prompts in Jupyter Notebook.

- We'll start with **image_prompt = 'Man on a motorcycle on a cyberpunk street, high quality photography'**.
    - Note: The more detailed the description, the more likely you are to get the desired result.
    - For inspiration, you can find some ***prompt examples*** at labs.openai.com:



- Now let's make our API image request.
    - This is similar to making a text request, just with ***less parameters***.

```
image_prompt = 'Man on a motorcycle on a cyberpunk street, high quality photography.'

response = openai.Image.create(
    prompt=image_prompt,
    n=1,
    size='1024x1024'
)

print(response)
```

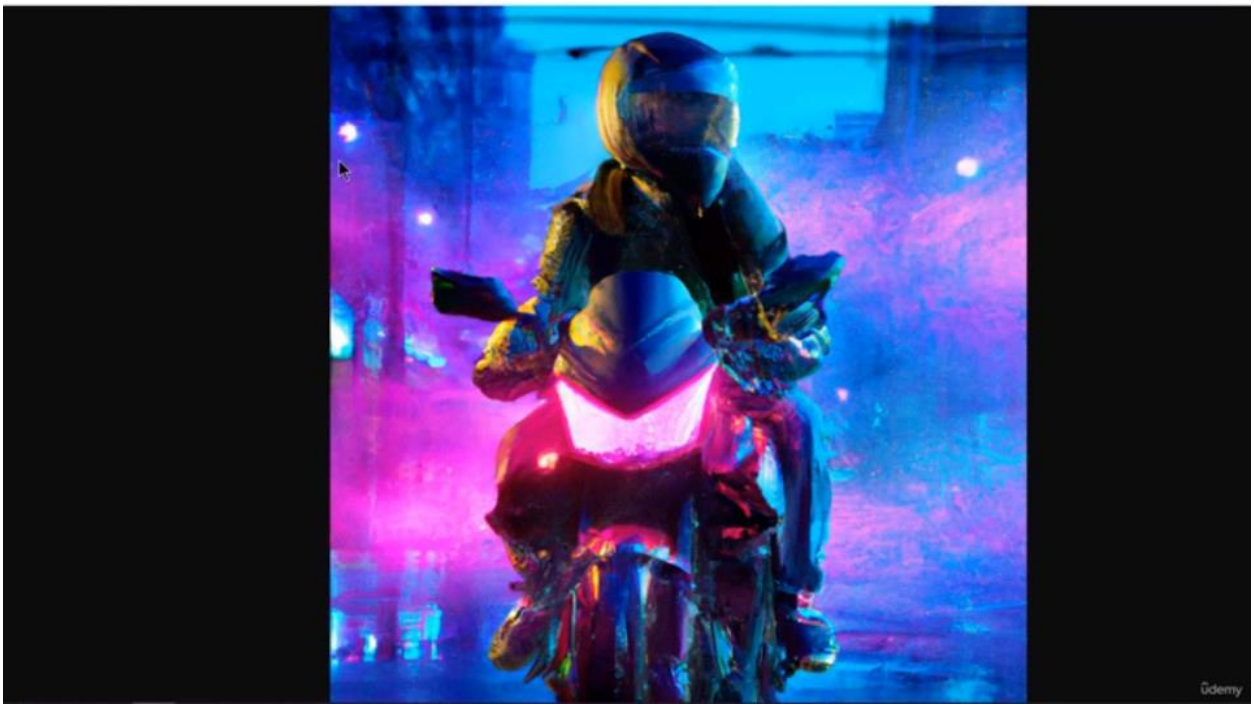
-
- What we get in return is a **URL address** where our picture can be found:
-
-
-
-
-

- What we get in return is a **URL address** where our picture can be found:


```
{
  "created": 1689894351,
  "data": [
    {
      "url": "https://oaidalleapiprodscus.blob.core.windows.net/private/org-rpEeJ3GzTH6XdecXvBuTa2rE/user-nBILA9Y1meePdnIW7KhM4ovs/img-jmgXPcnvoaN5T0DuodtkCWLy.png?st=2023-07-20T22%3A05%3A51Z&se=2023-07-21T00%3A05%3A51Z&sp=r&sv=2021-08-06&sr=b&rscd=inline&rsct=image/png&skoid=6aaadede-4fb3-4698-a8f6-684d7786b067&sktid=a48cca56-e6da-484e-a814-9c849652bcb3&skt=2023-07-20T20%3A07%3A03Z&ske=2023-07-21T20%3A07%3A03Z&sks=b&skv=2021-08-06&sig=6ZbI6A4UQHOM3GOyoNUEzo2WJjaSNRVIE0amNFMs%2BN0%3D"
    }
  ]
}
```


- If we copy/paste that URL into our browser, it shows us the resulting picture:

    - Note that when I first hovered over this picture after placing it in the Word document, there was even some metadata reading "a person on a motorcycle", so DALL-E seems to fill in metadata for you to some extent.
- We can extract the URL directly from the list/dictionary with:
    - **image_url = response['data'][0]['url']**
    - **print(image_url)**
        - (doing these two steps makes the link clickable)
    - Note: here's the instructor's version for comparison:



- We can also automatically save the image to a file on disk.
    - We can do that by switching the *response format* from URL to the Base64 encoded image data.
    - Or, we can leave the URL response format and get the image using requests.
- For our example here, let's do that second thing, because we already have the URL:
    - We also have to **!pip install -q requests** since "requests" isn't a standard Python library.
    - We need to import two things:
        - **import requests**
        - **import shutil**
            - "shutil" is used to copy files.
    - We'll be making an HTTP request using the "requests" library:
        - **resource = requests.get(image_url, stream=True)**
            - The "stream=True" part is necessary to get the ***raw content*** of the response.
            - "resource" is an HTTP response object with a *status code*. 200 means success, but codes beginning with "4" mean there's an error.
        - We can see the status code with **print(resource.status_code)**.

        - When we run ours, we should get "**200**".
    - Now we want to say that, if our status code is 200 we want to save the file to local disk as a PNG; otherwise, print an error message:

```
#print(response)
image_url = response['data'][0]['url']
print(image_url)

resource = requests.get(image_url, stream=True)

#print(resource.status_code)

if resource.status_code == 200: ← ← ←
    with open('dalle1.png', 'wb') as f: ← ← ←
        shutil.copyfileobj(resource.raw, f) ← ← ←
else:
    print('Error accessing the image!') ← ← ←
```

    - We run this and don't get any error. If we refresh our Jupyter Notebook's "Home Page", we can see the file has been saved there:

    - We can see "**dalle1.png**" at the bottom here.
- We can also display the image directly in Jupyter Notebook using the **PIL library** ("Python Image Library") and its "**Image**" class.

```
if resource.status_code == 200:
    with open('dalle1.png', 'wb') as f:
        shutil.copyfileobj(resource.raw, f)
else:
    print('Error accessing the image!')

Image.open('dalle1.png') ← ← ←
```

- And this displays a new image directly in Jupyter Notebook (note: I'm unsure about VSCode, might need some extra work there. I mainly use VSCode to copy/paste nice-looking code blocks from Jupyter Notebook to VSCode to this Word document).

Full Code, so far:

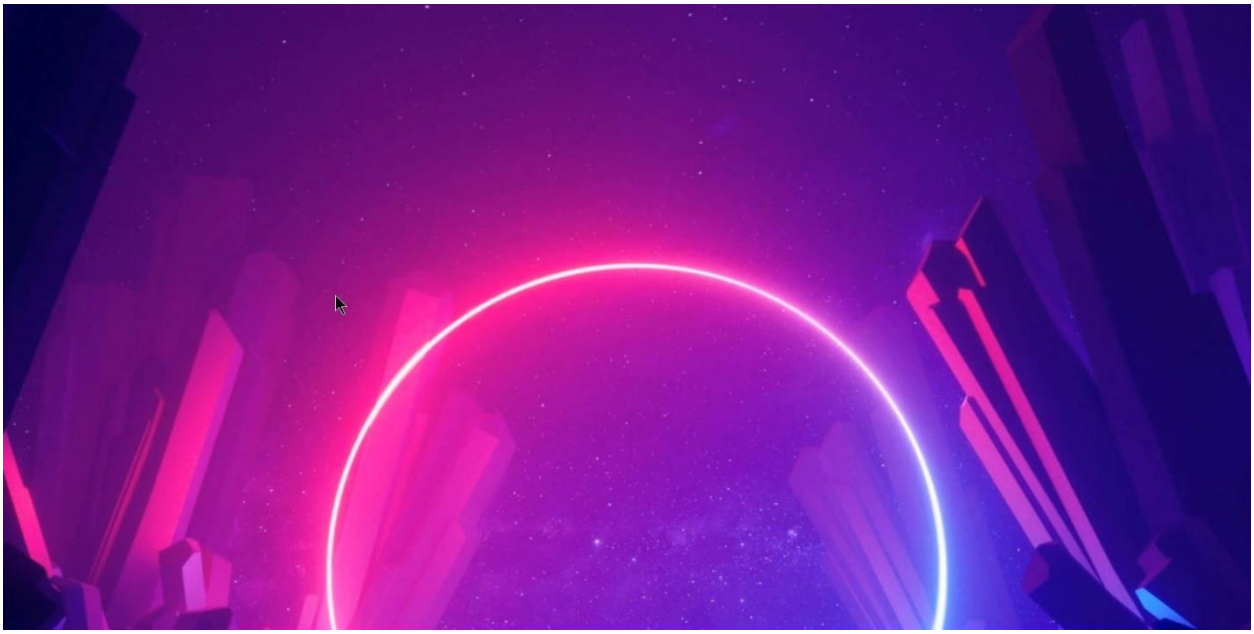
```
# !pip install -q openai, requests
import openai
import os
import requests
import shutil
from PIL import Image

openai.api_key = os.getenv('OPENAI_CLASS_API_KEY')

image_prompt = 'Man on a motorcycle on a cyberpunk street, high quality
photography.'

response = openai.Image.create(
    prompt=image_prompt,
    n=1,
    size='1024x1024'
)

#print(response)
image_url = response['data'][0]['url']
print(image_url)

resource = requests.get(image_url, stream=True)

#print(resource.status_code)

if resource.status_code == 200:
    with open('dalle1.png', 'wb') as f:
        shutil.copyfileobj(resource.raw, f)
else:
    print('Error accessing the image!')

Image.open('dalle1.png')
```

Using DALL-E to Create Variations and Edit Images:

- Now let's see how we can create variations and edit images using DALL-E.
    - In addition to creating new and original images, we can also use DALL-E to **mod**ify **exis**ting **on**es, **cre**ate **vari**ations of the **im**age that **mai**ntain **exis**ting **fea**tures, and **inter**polate **bet**ween **t**wo **in**put **im**ages.
- Whether you have an image that was created by DALL-E or not, you can create variations of it using OpenAI's DALL-E **Latent Diffusion Model**.

Creating Variations:

- Here's the image the instructor is starting with (note that he has this in his Jupyter Notebook files):



    - We want one or more variations of it, and the image should be in the ***PNG format***.
- The first thing we need to do is open the existing image in Python using the **open** function (just like in file processing).
    - **Note:** I had screenshotted a copy of his image and named it "Variation_Test_Image.png", but duplicated that and renamed it "**original.png**" to match what he had going on in his screen.
    - We create a variable **image = open('original.png', 'rb')**
        - This file is ***binary***, so we have to read it as such, hence **'rb'**.
    - We then create a **response** variable to store our response, using OpenAI's **Image.create_variation** method:

```
image = open('original.png', 'rb')
response = openai.Image.create_variation( ← ← ←
    image=image,
    n=1
)
```

```
image = open('original.png', 'rb')
response = openai.Image.create_variation(
    image=image,
    n=1,
    size='1024x1024'
)
```

- We want to make sure we set **n=1** here, because we ***pay per image***, so we only want one.
- We also want to set **size='1024x1024'**.

- Next, let's store the ***image URL*** from the list/dictionary as a variable by extracting it:

```
image = open('original.png', 'rb')
response = openai.Image.create_variation(
    image=image,
    n=1,
    size='1024x1024'
)

image_url = response['data'][0]['url']
```

- We can **print(image_url)** to see that that's working correctly:

```
13 image_url = response['data'][0]['url']
14
```

```
In [3]: 1 print(image_url)
```

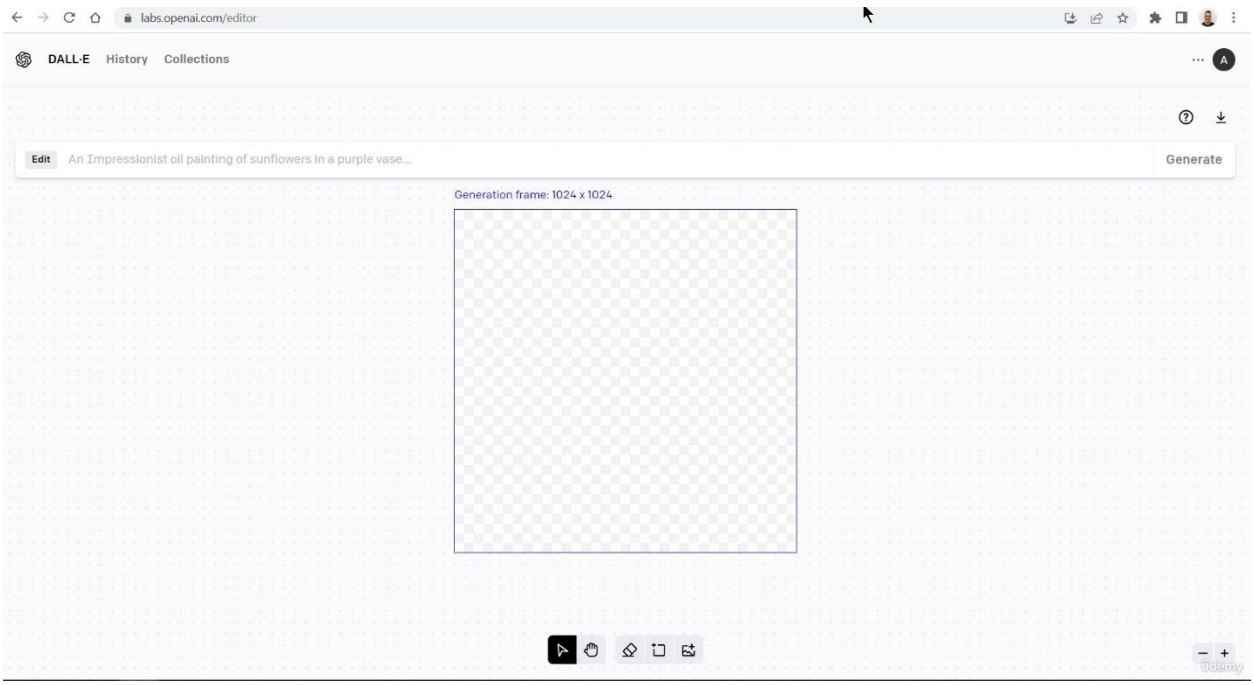
https://oaidalleapiprodscus.blob.core.windows.net/private/org-4HIn9BHJwWRqZhmlRcnB3fhq/user-9EXCgqwgTQn2D79IRmhd8EKR/img-q14A9sk5qXDXIDiOhhuVtSsE.png?st=2023-03-10T13%3A43%3A28Z&se=2023-03-10T15%3A43%3A28Z&sp=r&sv=2021-08-06&sr=b&rscd=inline&rsct=image/png&skoid=6aaadede-4fb3-4698-a8f6-684d7786b067&sktid=a48cca56-e6da-484e-a814-9c849652bcb3&skt=2023-03-10T14%3A31%3A11Z&ske=2023-03-11T14%3A31%3A11Z&sks=b&skv=2021-08-06&sig=QJOtQF9Om6qBJAdwnqG4UZH0G2zDW3osy7SB%2BvhYNp4%3D

- If we click on this link, it takes us to our first *image variation*. Here is the **variation on the left** and the **original on the right**:

- It seems to have thickened the ring and sharpened the crystalline structures to the sides.

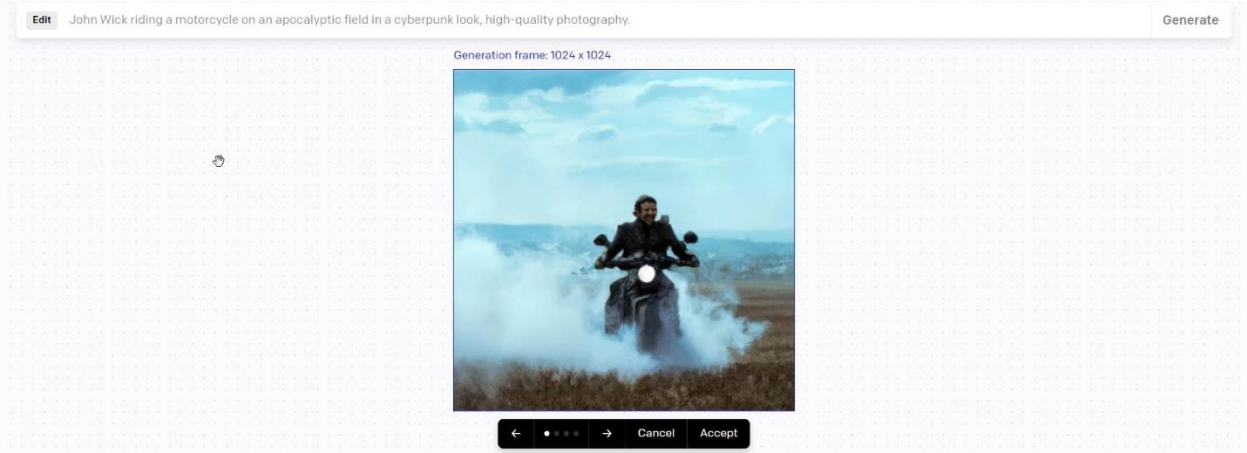
Editing Images:

- Next let's see how to edit an existing image. To start, the instructor created a brand new image in **DALL-E's editor interface**, which can be found at labs.openai.com/editor. The editor interface looks like this:



- The instructor noted that we can also generate an image using the **Python OpenAI API**, or we can take **any other existing image**.
-

- In DALL-E editor, the instructor then used the prompt "**John Wick riding a motorcycle on an apocalyptic field in a cyberpunk look, high-quality photography.**" This produced this:

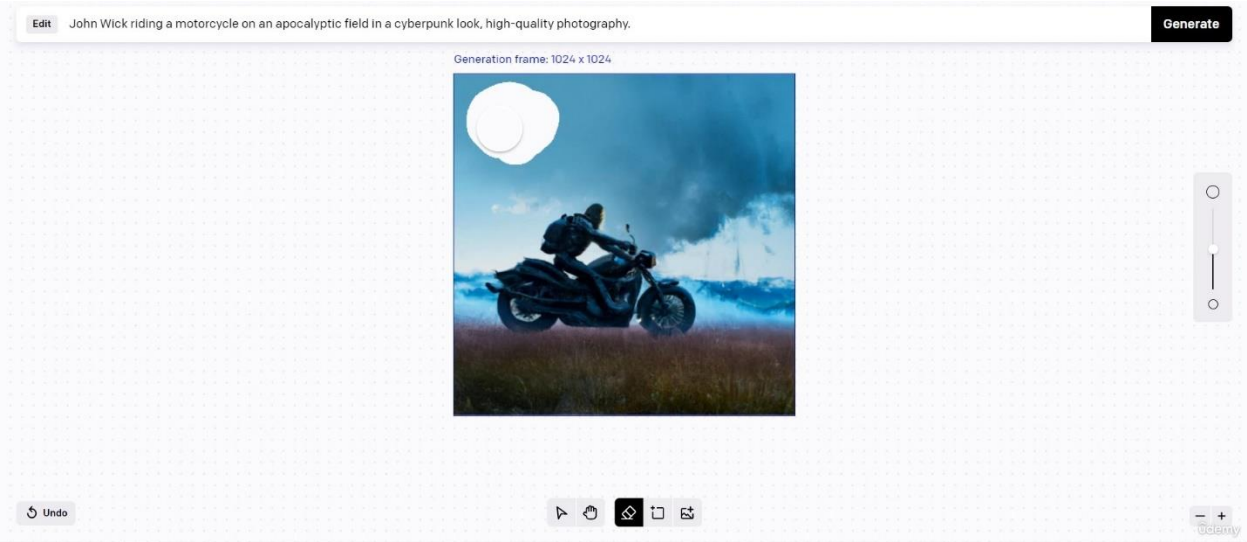


  - The instructor noted that DALL-E works better with longer sentences and more detail, because shorter sentences are too general to make anything meaningful from. DALL-E 2 is really good at handling complexity.
  - DALL-E also produced ***multiple image*** for this prompt for us to choose:



- We're going to go with that third image, and we're going to **mo**ve **t**he **im**age to **Jup**yter **Not**ebook and rename it "**motorcycle_original.png**".
-
- The instructor noted that while the image was generated for us, it's **n**ot **t**he **be**st **qua**lity. We want more control over our work.
  - For example, perhaps we **want the image to contain a red moon in a specific part of the image**.
  - So we want to create a ***variation*** of the image.
-
-
-
-
-

- The instructor noted that while the image was generated for us, it's **n**ot **t**he **be**st **qua**lity. We want more control over our work.
    - For example, perhaps we **want the image to contain a red moon in a specific part of the image**.
    - So we want to create a ***variation*** of the image.
- To *create a variation*, we need the **original image** and a **mask**.
    - The **mask** is the same image, with a **transparent area** that indicates where the image (moon) should be edited in.
    - We can create a transparent area in our image using the **eraser tool** and erasing a small portion:



    - Note that just like how you can use an image from anywhere, you can also *create the **mask** in **any other photo editor***.
        - Note: I bet this is why the image needs to be a PNG. When you finish using the eraser tool, you see the same gray and white checkerboard as you do in a transparent PNG.
    - We save this ***mask*** from DALL-E to our machine and we move it to our Jupyter Notebook directory, and we rename it "**mask.png**". Below is my own version of it:

- Now in our Jupyter Notebooks directory, we have both the **original image** and the **mask**.
    - In Python, we start by creating a new ***file object*** by opening the **original image** like

```
image = open('motorcycle_original.png', 'rb')
```

usual:

- The instructor noted that the image used as the basis for the variation here must be a ***valid PNG file***, it must be ***less than 4mb***, and must be ***square***. I wonder why it must be square…
- We also open the **mask** the same way:

```
image = open('motorcycle_original.png', 'rb')
mask = open('mask.png', 'rb')
```

- We also want to create a new ***response object*** with **openai.Image.create_edit**:

```
image = open('motorcycle_original.png', 'rb')
mask = open('mask.png', 'rb')
response = openai.Image.create_edit(
    image=image,
    mask=mask,
    prompt=
)
```

- The ***prompt*** we input here should describe the ***full new image***, not just the erased area. So let's copy/paste our original prompt:
    - "John Wick riding a motorcycle on an apocalyptic field with a cyberpunk look, high quality photography"
- And into this we insert:
    - "John Wick riding a motorcycle on an apocalyptic field **containing a big red moon** with a cyberpunk look, high quality photography"

```
image = open('motorcycle_original.png', 'rb')
mask = open('mask.png', 'rb')
response = openai.Image.create_edit(
    image=image,
    mask=mask,
    prompt='John Wick riding a motorcycle on an apocalyptic field \
    containing a big red moon with a cyberpunk look, high quality photography'
)
```

- We also want to add **n=1** and **size='1024x1024'**.
- Next, we want to add **image_url = response['data'][0]['url']** to extract the URL.
-
-

- Next, we want to add **image_url = response['data'][0]['url']** to extract the URL.

```
image = open('motorcycle_original.png', 'rb')
mask = open('mask.png', 'rb')
response = openai.Image.create_edit(
    image=image,
    mask=mask,
    prompt='John Wick riding a motorcycle on an apocalyptic field containing a
big red moon \
    with a cyberpunk look, high quality photography',
    n=1,
    size='1024x1024'
)

image_url = response['data'][0]['url']
```

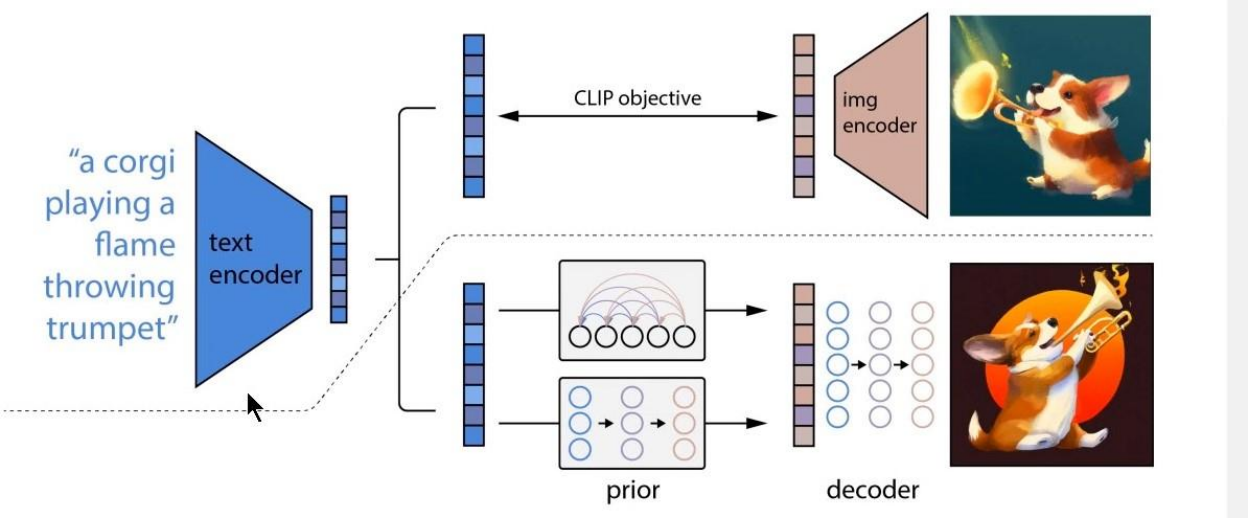
- And that gave me this image:

- Here's the original for comparison:

Diving into DALL-E:

Note: Resources for this lecture include article "Attention Is All You Need" (https://arxiv.org/abs/1706.03762), PDF article "Hierarchical Text-Conditional Image Generation with CLIP Latents" (labeled "dall-e-2.pdf" and saved to "Resources" folder), and link to OpenAI "CLIP" page (https://openai.com/research/clip).

- In the previous lectures, we learned how to make new and realistic images, how to create variations, and how to edit existing images using the OpenAI API, DALL-E, and Python. Now we're going to take a look **behind the scenes** to understand more about how it works.
- **DALL-E** is a neural network and a version of GPT-3 that uses a dataset of **text-image pairs**.
- DALL-E is a **transformer language model**.
    - Transformers are a new type of neural network that can be scaled up and trained on huge datasets.
    - They are considered revolutionary in natural language processing and are used by many AI models.
    - One such transformer, BERT, is being used by Google to optimize user search queries.
    - Other models include GPT-3, GPT-3.5 from OpenAI, and AlphaFold from DeepMind.
        - AlphaFold is designed to predict the 3D structure of proteins.
- We opened the attached PDF "Hierarchical Text-Conditional Image Generation with CLIP Latents", which is the paper where DALL-E was first described. We focused on one image from the paper in particular:



    - The instructor provided the following explanation:
        - "A **text encoder** takes the text prompt and generates the **text embeddings**. These text embeddings are a numeric representation of the input and serve as the input for a model called the **prior**. It generates the corresponding **image embeddings**. Finally, an **image decoder** model generates an actual image from the embeddings."
-

- From a text description, **DALL-E** will generate a large set of images, and the images will then be ranked by a second OpenAI neural network called **CLIP** that tries to determine which of the images matches best.
- **CLIP** has 2 components:
    - A text encoder.
    - An image encoder.
- It was trained on ***400 million*** images with text captions that were *scraped from the Internet*.
    - There's a lot of talk right now of the legal gray zone this presents with regards to copyright law.

DALL-E 3 Risks and Limitations:

- DALL-E (without guardrails) could generate a wide range of deceptive and harmful content.
    - This could affect how people perceive/trust content in general.
- AI models like DALL-E additionally inherit various biases from their training data, and its outputs could reinforce social stereotypes.

Introducing ChatGPT and Whisper APIs:

Note: Resource for this lecture includes the PDF "Robust Speech Recognition via Large-Scale Weak Supervision", labeled "whisper.pdf".

- In this lecture, we'll cover the **speech-to-text model** known as **Whisper**.
- By the end of the lecture, we'll be able to ***transcribe audio*** into whatever language it is and translate the transcription into English.
- **Whisper** is a trained Open-Source neural network that approaches human-level robustness and accuracy on English speech recognition.
    - The model supports ***tens of languages*** and was trained on hundreds of thousands of multilingual and multitask supervised data collected from the internet.
- The OpenAI API provides two endpoints:
    - **Transcriptions**
    - **Translations**
- In practical terms, you can use the Whisper model to transcribe audio into whatever language the audio is in, or to translate and transcribe audio into English.

Transcription:

- We'll start by creating a new Jupyter Notebook program, "**07_whisper**". We'll import the necessary libraries and set our API key from the ***environment variable***.
    - I guess we'll be using some audio from Ronald Reagan's speech to the people of West Berlin from July 12, 1987. This is the "Tear Down This Wall" speech.
    - The instructor had included the audio file in Jupyter Notebook, titled "**rr.mp3**".
    - For comparison's sake, he had the official transcript of the speech open.
        - I think I'll just use an mp3 from my music library. Perhaps "**Like a Stone**" by Audioslave because it's my go-to karaoke song and I know all the words.
- We're going to open the file in **binary mode** as usual and name it "**audio_file**". Then we'll create a storage variable "**transcript**" and use OpenAI's "**Audio**" and "**translate**" method:

```
import openai
import os

openai.api_key = os.getenv('OPENAI_CLASS_API_KEY')

with open('05 Like a Stone.mp3', 'rb') as audio_file:
    transcript = openai.Audio.transcribe('whisper-1', audio_file)
```

    - Note: Obviously you'd want to swap out the name for whatever MP3 you're using for this.
    - By default, the response type will be in JSON with the raw text included.
- The response I got was pretty close to what I know of the song, though it did add "Thanks for watching!" at the end for some reason:
-
-
-

-
- The response I got was pretty close to what I know of the song, though it did add “Thanks for watching!” at the end for some reason:

```
import openai
import os

openai.api_key = os.getenv('OPENAI_CLASS_API_KEY')

with open('05 Like a Stone.mp3', 'rb') as audio_file:
    transcript = openai.Audio.transcribe('whisper-1', audio_file)
    print(transcript)
```

```
{
  "text": "On a cobweb, afternoon in a room Full of emptiness by a freeway I confess I was lost in the pages Of a book full
of death's reading I will die alone and if we're good We'll lay to rest anywhere we wanna go In your house I long to be Room
by room patiently I'll wait for you there like a stone I'll wait for you there alone And on my deathbed I will pray to the g
ods And the angels like a pagan To anyone who will take me to heaven To a place I recall I was there so long ago The sky was
bruised, the wine was bled And there you led me on In your house I long to be Room by room patiently I'll wait for you there
like a stone I'll wait for you there alone I'll wait for you there like a stone I'll wait for you there alone And on I read
until the day was gone And I sat in regret of all the things I've done For all that I've blessed and all that I've wronged A
nd dreams until my death I will wander on In your house I long to be Room by room patiently I'll wait for you there like a s
tone I'll wait for you there alone Thanks for watching!"
}
```

- Let’s see the instructor’s version with the speech:

```
{
  "text": "We welcome change and openness, for we believe that freedom and security go together. That
the advance of human liberty can only strengthen the cause of world peace. There is one sign the Sovi
ets can make that would be unmistakable, that would advance dramatically the cause of freedom and pea
ce. General Secretary Gorbachev, if you seek peace, if you seek prosperity for the Soviet Union and E
astern Europe, if you seek liberalization, come here to this gate. Mr. Gorbachev, open this gate. Mr.
Gorbachev, Mr. Gorbachev, tear down this wall."
}
```

    - Pretty spot-on. I wonder if Whisper does better with regular speech compared to singing, especially when presented with Chris Cornell’s impressive vocal range.
- The instructor noted that audio files are currently ***limited to 25mb*** and the input files that are supported are ***MP3***, ***MP4***, ***MPEG***, ***M4A***, and ***WAV***.
-
-
-
-
-

Translation:

- Besides transcription, Whisper is also capable of translating things into English. We're going to practice doing an automatic translation.
- The instructor has some sample audio in German, a children's story (labeled "german.mp3").
    - I think I'll go with a song in Spanish that I know, "Primavera" by Santana, off of the album Supernatural.
- For his file, the instructor noted that if you listen to the file, you'll hear some background noises that you might think would cause problems for the translator. However, this is not the case.
    - Whisper has improved its ***robustness*** to accents, background noise, and technical language.
- The **translations API** takes as input an audio file in any of the supported languages, and then transcribes it into English.
- We're going to store our file as "audio_file" again with **open**, then create another **transcript** variable, but this time we use **openai.Audio.translate**:

```
audio_file = open('12 Primavera.mp3', 'rb')
transcript = openai.Audio.translate('whisper-1', audio_file)
print(transcript['text'])
```

- Note that running the **translate** method seems to take a bit longer than simple transcription.
- Also, it does **not** seem to work with music:

```
audio_file = open('12 Primavera.mp3', 'rb')
transcript = openai.Audio.translate('whisper-1', audio_file)
print(transcript['text'])
```

```
[♪ Music begins playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music co
ntinues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues
playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ ♪ [♪ Music continues playin
g ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ M
usic continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music con
tinues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues p
laying ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪
[♪ Music continues playing♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music
continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continue
s playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playin
g ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ M
usic continues playing ♪ [♪ Music continues playing ♪ ♬ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music
continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continue
s playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playin
g ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ M
usic continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music con
tinues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues p
laying ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪
[♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music
continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continues playing ♪ [♪ Music continue
s playing ♪ [♪ Music continues playing ♪ [♪ Music fades out ♪ [♪ Music fades out ♪
```

- Here's the instructor's version for comparison:
-
-
-
-
-
-

- Here's the instructor's version for comparison:

```
audio_file = open('german.mp3', 'rb')
transcript = openai.Audio.translate('whisper-1', audio_file)
print(transcript['text'])
aud
```

```
Hello? Great, dad! It's really great here, with the lions. But this lion here, your BQ, is not here.
He's probably sleeping inside, right? Not today. Because today the lion is not inside, but... ...not
here at all. But? At the doctor's. Once a year he's examined, vaccinated, something like that. I unde
rstand. But now tell me, how was school? Good. The bandage on your forehead. Where? At the school yar
d? Doesn't matter. Really, dad. Doesn't matter. No, it doesn't matter. They said things about you.
```

- Also, remember to run **audio_file.close()** because it's good practice.

Section 3: Project #1: Zero-Shot Sentiment Analysis Using ChatGPT:

Project Requirements:

- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
- A
-
-

Template: Code Box:

For use whenever code needs to be presented in notes.

```
user_input = float(input("Enter temperature: ")) ← ← ←
print(weather_condition(user_input))
```

```
from tkinter import *

window = Tk() ← ← ← Create window

l1 = Label(window, text="Title") ← ← ← Create labels
l1.grid(row=0, column=0)

l2 = Label(window, text="Author") ← ← ←
l2.grid(row=0, column=2)

window.mainloop()
```