Learning Objectives

Learners will be able to...

- **Generate a response using `openai.Completion.create`**
- **Interact with the `temperature` and `top_p` to produce different outputs**
- **Manipulate and change different boundaries for `n` and `best_of`**
- **Define token and tokenizer**
- **Tokenize words and sentences**
- **Describe frequency and presence penalties**

info

Make Sure You Know

You are familiar with Python.

Completion.create Method

To generate a response to a prompt, we use the `openai.Completion.create` method. Note that we have already imported the `openai` package.

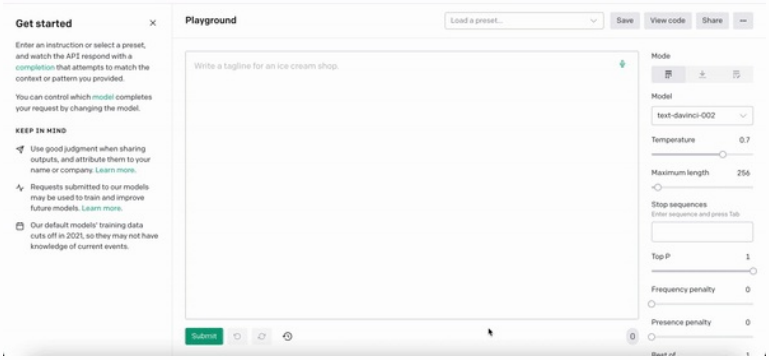
```
prompts ="Write a tagline for an ice cream shop"
response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts)

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

Copy the code above into the file on the left and click the button below to run the code file.

Notice how the method takes two keyword arguments. The **** is mandatory and the prompt is specified using the ****

There are many additional optional keyword arguments that can be specified in our API call. You can get a sense of these settings using the right-hand pane of the OpenAI playground. Open the playground (the link opens in a new tab). Clicking on **View code** on the upper right will translate the sliders and text boxes into code.



Animation showing OpenAI UI where cursor moves to top-right corner and clicks on the View code button resulting in a pop up titled View code with an API call with arguments model, prompt, temperature, max_tokens, top_p, frequency_penalty, and presence_penalty

Here is an example of an API call using more keyword arguments:

```
import os
import openai

openai.api_key = os.getenv("OPENAI_API_KEY")

response = openai.Completion.create(
  model="text-davinci-002",
  prompt="",
  temperature=0.7,
  max_tokens=256,
  top_p=1,
  frequency_penalty=0,
  presence_penalty=0
)
```

We will cover what each of these keyword arguments does throughout this assignment.

Temperature

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

Try changing the value of temperature from the **default of** 1 to 0 in the file on the left and compare how your output changes.

```
prompts ="Write a tagline for an ice cream shop"
response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts,
                                    temperature=0) ## added

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

Try running the code a few times by clicking the button below multiple times.

You probably got the same output each time. When `temperature` is set to 0 it is referred to as **argmax sampling**, meaning the option with the highest probability is always selected. The option with the highest probability will perceive as the most *"correct"* answer. Higher temperatures will generate a more diverse response.

challenge

Try these variations:

- Try running the code multiple times when you set `temperature` to `0.5`:

```
response = openai.Completion.create(model="text-davinci-
        002",
                                    prompt=prompts,
                                    temperature=0.5)
```

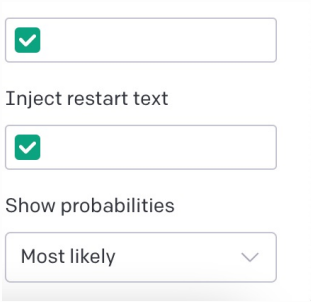
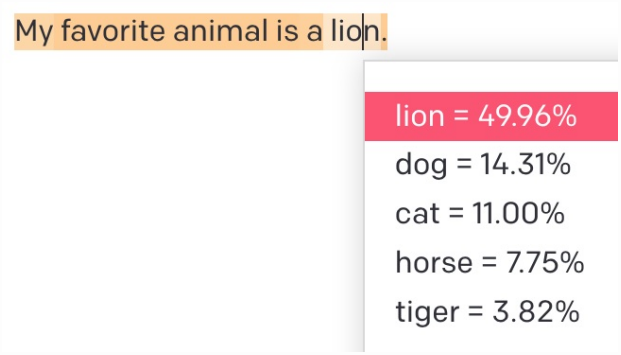
- Try running the code multiple times when you set `temperature` to `1.7`:

```
response = openai.Completion.create(model="text-davinci-
        002",
                                    prompt=prompts,
                                    temperature=1.7)
```

You might have picked up at this point that *temperature controls how much randomness is in the output.* **Lower temperature means less randomness** which is suitable for contexts which require more stable output. **Higher temperature means more randomness** which is suitable for more creative applications.

You can read more about `temperature` in the OpenAI docs.

▼ **Seeing Probabilities**

> The OpenAI playground has a **Show Probabilities** setting you can turn on that shows the probability of generated words:
>
> 
>
>
> This creates a heat map where the darker the shade of the generated text means the more confident the model is about it. You can click on a word to see the different options that were considered and their related probabilities.
>
> 
>
>
> You can run the same prompt multiple times with different temperatures to see how it affects the displayed probabilities.

To see this difference, let's generate 3 responses at a low temperature (0.2) and 3 responses at a high temperature (0.8):

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

print('Temperature 0.2:')
print('----------------')
for i in range(3):
  response = openai.Completion.create(model="text-davinci-002",
                                      prompt=prompts,
                                      temperature=0.2)

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

print('\n\nTemperature 0.8:')
print('----------------')

for i in range(3):
  response = openai.Completion.create(model="text-davinci-002",
                                      prompt=prompts,
                                      temperature=0.8)

  print(response["choices"][0]["text"].strip())
```

You can try running the above a few times but note that it is asking for 6 generations for each run so it will consume tokens quickly.

Here is an example of the output:

```
Temperature 0.2:
----------------
The best ice cream in town!
The best ice cream in town!
The best ice cream in town!


Temperature 0.8:
----------------
Make your summer sweeter with our delicious ice cream!
There's always room for ice cream!
Come in for a brain freeze!
```

We can see that when we use a lower temperature, the same text response is generated repeatedly. With the higher temperature, we get a wider variety of output.

N and Best Of

The N Keyword Argument

We can use the keyword argument `n` to specify the number of parameters in order to generate multiple completions. It can use up your tokens fairly quickly be warned. By default, it is set to `n=1`. Let's remove our for loop and add an extra argument to our response and set that equal to 5.

```
prompts ="Write a tagline for an ice cream shop"
response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts,
                                    n=5)
print(response)
```

We can further clean up our text by changing our print statement to the following:

```
for i in (response["choices"]):
  print(i["text"].strip())
```

Run the script again and Python will print five different responses to the prompt.

The Best Of Keyword Argument

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

```
response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts,
                                    n=5,
                                    best_of=4)
```

You will generate an error essentially saying you requested the server to return more choices than it will generate. `n` needs to be less than or equal to `best_of`. We don't need to use `n` in order to use `best_of`. It will generate 4 completions but only print the *"best"* one.

```
response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts,
                                    best_of=4)
```

Tokens

Copy and paste the following lines of codes into our editor on the left:

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

response = openai.Completion.create(model="text-davinci-002",
        prompt=prompts,
                                    temperature=0,
                                    max_tokens=6)
print(response)
```

You will notice, we switch out our print statement. It will make seeing the token variable easier. When we run the code we get the following as part of response being printed.

```
"completion_tokens": 6,
"prompt_tokens": 9,
"total_tokens": 15
```

In order to just have the tokens being printed, we can switch our print statement to the following.

```
print(response['usage'])
```

Before we tackle what each of the answers involving tokens means, let's start with what the tokens are.

Definition: Token

The GPT family of models process text using ***tokens***. Tokens are numerical representation of words or characters.
- The process of tokenization involves dividing plain text, such as a phrase, sentence, paragraph, or entire documents into smaller chunks of data so that it is easier to analyze.
- These smaller chunks of data are referred to as *tokens*, which may include words, phrases, or sentences.

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

One can tokenize in two ways:

- **Tokenizing by word** - The benefit of using tokenization by word is that we can pinpoint words that are frequently used. If we were analyzing a group of restaurant ads in NY, and found that the word “vegan” was used often, then we might assume that there are plenty of vegan options at these restaurants.
- **Tokenizing by sentence** - When tokenizing by sentence, we have the ability to analyze how the words in the sentence correlate with each other, which allows us to better understand the context of the words. If we were analyzing a group of restaurant ads in NY, and found that the sentence “No vegan options.” was used, then we can determine that there are not plenty of vegan options at these restaurants.

Lastly, OpenAI imposes a 2048 token limit in order to maintain latency. OpenAI works in a “pay as you go” model. Tokens are one of the units used to determine pricing for an API call.

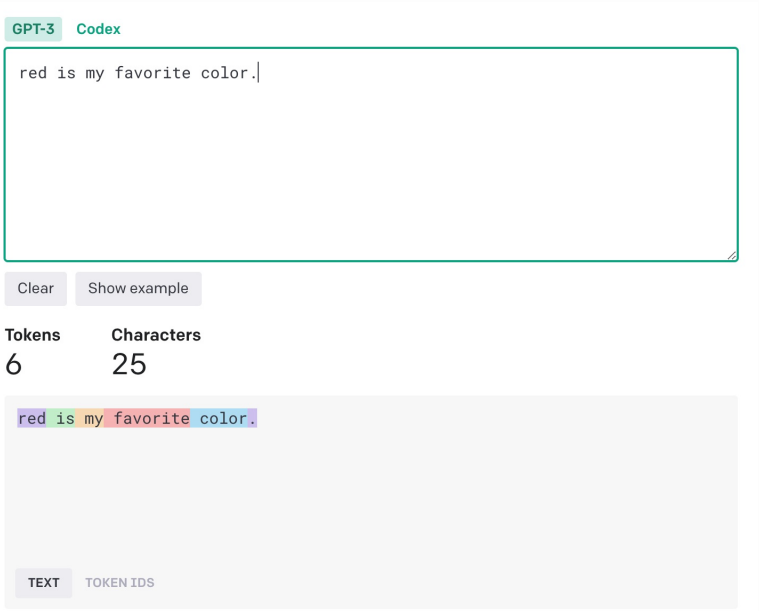
Tokenizer Playground

Word and Token

The API treats words according to their context in the text data. Remember The same word can have 2 different tokens count depending on the structure of the text.

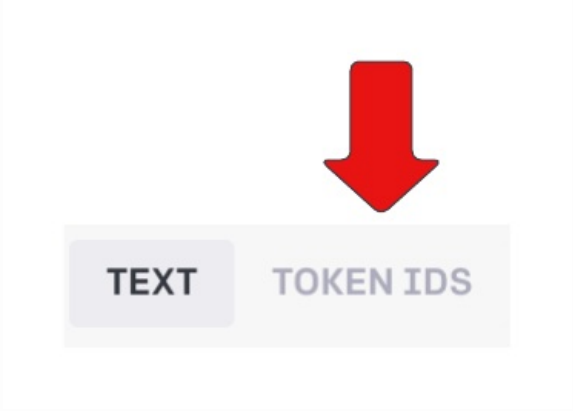
OpenAI provides a tokenizer tool with which we can experiment. Open the tool with this link. Try the following in the playground:

```
red is my favorite color.
```



picture of the OpenAI tokenizer playground box. The text “red is my favorite color” is inside the text playground box. As response one can see the number of tokens is 6 and the number of characters is 25. After that we can see the “red is my favorite color” and a period broken up and with each word being in a different color.

The playground uses different colors to show the breakdown between token assignment. When we click on the `TOKEN IDS` instead of text at the bottom, we can see the tokens associated with each word.



We see two boxes with the words Text and Token IDS. The Token IDS box is circled in red

Now change the lowercase `r` in red to capital `R`, check the token id to see if anything changed.

The same word can have 2 different tokens count depending on the structure of the text. On that same note, how we split up our words into tokens is language-dependent.

Try these variations:

Enter the following words in the tokenizer playground and compare the token IDS with the expected output below.

Enter the word: `Red`
Expected output: `[7738]`

Enter the words: `The Red`
Expected output: `[464, 2297]`

Enter the words: `The Red keep`
Expected output: `[464, 2297, 1394]`

Enter the words: `Red keep`
Expected output: `[7738, 1394]`

When comparing all the different tokens associated with `red`, you will find out that:

- The token generated for a word varies depending on its placement within the sentence or if capitalized or not
- Tokens can include trailing space characters

The more we know about tokens it can help with a better prompt design. For example, prompts ending with a space character may result in lower-quality output. This is because the API already incorporates trailing spaces in its dictionary of tokens.

More on tokens and how to count them can be found on the OpenAI website

Max Tokens

GPT-3 takes the prompt, converts the input into a list of tokens, processes the prompt, and converts the predicted tokens back to the words we see in the response.

Token Limits

Keep track of the following when using the API:

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

Enter the following code into the IDE:

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

response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts,
                                    max_tokens=6)
print(response)
```

Click the button below to see the full response.

We can see there is a **usage** being printed and there we can see the total tokens being used. It tells us the total number of tokens, and how they are split between prompt and completion. Now try modifying the `max_tokens` keyword argument from 6 to 9.

```
response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts,
                                    temperature=0,
                                    max_tokens=9)
```

We can see the completion token going from 6 to 9.

challenge

Try these variations:

- Try running the code when you set `max_tokens=10`
- Try running the code when you set `max_tokens=16`
- Try running the code when you set `max_tokens=30`

As you can see after a certain number of tokens, the completion token won't change in value. That is because we have reached the maximum number of tokens that could be assigned. Note, the **max_tokens** keyword argument has a default value of 16 and simply sets a boundary for the number of tokens to be generated in the completion.

Now try the following, change our prompt to the following:

```
prompts ="Write a tagline for an ice cream shop in Paris"
```

We can see the prompt token switch from 9 to 11. It is evident, that the `prompt_tokens` simply displays the number of tokens present in the given prompt.

Top P

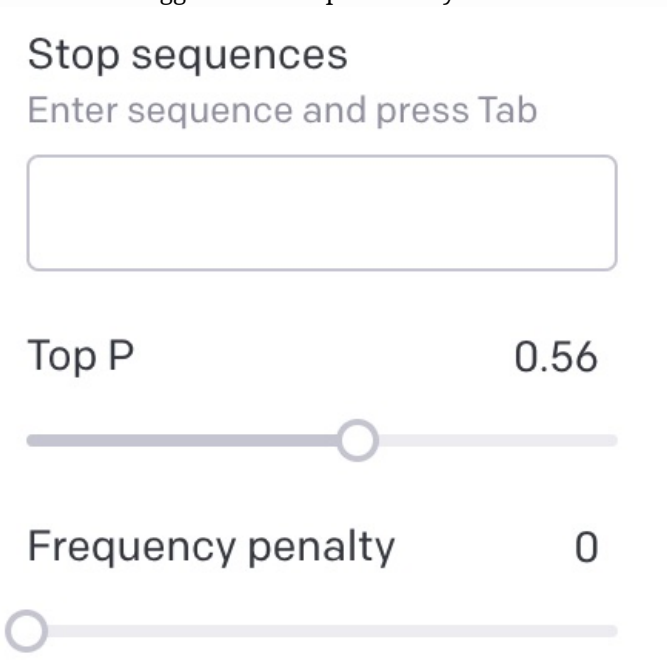
top_p, an alternative to sampling with temperature, is also referred to as nucleus sampling. Generally, it is not recommended to alter both the temperature and the `top_p`. `top_p` controls how many random results should be considered for completion as per the temperature. If we set so 0.1 means only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

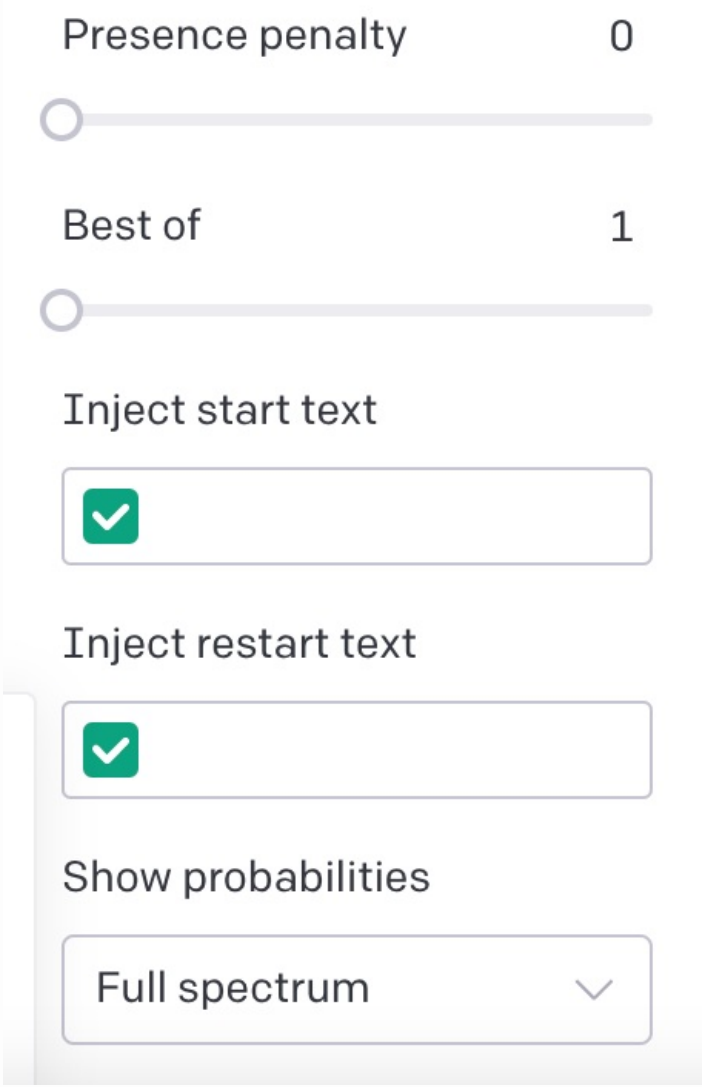
Remember how when using the temperature of 1 we had different answers show up. Let's try setting the `top_p=0.1`,meaning only top 10 of probable answers.

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

response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts,
                                    n=5,
                                    top_p=0.1)
for i in (response["choices"]):
  print(i["text"].strip())
```

We can use our probabilities tool to see the probability associated with each token. The Probability tool can be accessed in the OpenAI playground. Make sure we toggle the show probability tab.





we see the sentence “The best ice cream in town!”. The cursor hovers over the ice word and we see a drop down with words with different probabilities. ice=99.72% ,way=0.16%,place=0.04 percent,damn=0.03%,Ice=0.01%

Here we can see that the response generated 5 times was the same. That is because it is taking the top 10 most probable responses which is the same. What happens when we switch `top_p=0.8`.

```
response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts,
                                    n=5,
                                    top_p=0.8)
for i in (response["choices"]):
  print(i["text"].strip())
```

We start to see more variation as it has a bigger pool of responses to choose from.

Frequency and Presence Penalty

The last keyword arguments to introduce are `frequency_penalty` and `presence_penalty`. These two arguments are used to control the amount of repetition you see in the results. Both of these arguments have a value from -2 to 2.

Frequency Penalty

Frequency penalty is used to decrease the likelihood of the same line being repeated word for word. The lower the value for `frequency_penalty`, the more likely you will see the same line repeated. Think of `frequency_penalty` as a way to not have too many same-word repetitions. Positive values penalize new tokens based on their existing frequency in the text so far, decreasing the model's likelihood to repeat the same line verbatim. Set the value to 2 and run the program. Try it with others values such as 1 then 0

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

response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts,
                                    n=5,
                                    frequency_penalty=2,
                                    presence_penalty=0)
for i in (response["choices"]):
  print("----")
  print(i["text"].strip())
```

You should see the same completion appear more than once. Run it one or two more times to see the different ways in which completions can be repeated.

challenge

Try this variation:

- Change `frequency_penalty` to `-2`

```
frequency_penalty=-2,
```

Presence Penalty

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

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

response = openai.Completion.create(model="text-davinci-002",
                                    prompt=prompts,
                                    n=5,
                                    frequency_penalty=0,
                                    presence_penalty=2)
for i in (response["choices"]):
  print(i["text"].strip())
```

The completions should introduce new ways in which the model talks about ice cream.

challenge

Try this variation:

- Change `presence_penalty` to `-2`

```
                                    presence_penalty=-2)
```

Effects of Frequency and Presence

Running code snippets can provide different results, which makes talking about these two ideas a bit difficult. Instead, we are going to look at a prompt below to see how the completion changes when using frequency and presence.

```
"Describe the difference between cooking and baking"
```

Neutral Frequency and Presence

The completion repeats certain words (“cooking”) and phrases (“process of preparing food”). It is also of moderate length.

> Cooking is the process of preparing food by heating it, while baking is the process of preparing food by cooking it in an oven.

Maximum Value for Frequency

When frequency is increased, you no longer see the repetition of an entire phrase, though the word “cooking” occurs twice.

> Cooking is the process of preparing food by heating it, while baking is a method of cooking that uses dry heat, typically in an oven.

Maximum Value for Presence

Notice how the completion introduces new topics when it lists out the different methods of cooking food — baking, frying, and grilling.

> Cooking is the process of preparing food by heating it. This can be done in a number of ways, including baking, frying, and grilling. Baking is a type of cooking that involves using dry heat to cook food, typically in an oven.

Maximum Value for Frequency and Presence

In the last example, you see the repetition of a few words but no phrases are used more than once. You also see the introduction of ideas like the consumption of food or the explicit statement that baking is a form of cooking.

> Cooking generally refers to the process of using heat to prepare food for consumption. Baking, on the other hand, is a specific type of cooking that involves using dry heat - usually in an oven - to cook food.

You can further use this tool to generate more comparisons.

Coding Exercise

Requirement :
* Write a code so that we have the most random possible answer. Meaning setting the value that determines randomness to max
* Generate 6 responses and pick the best among them
* The completion token limit should be set 25
* Do not include any argument that are not necessary to the requirements