# Learning Objectives

---

### Learners will be able to...

- **Generate a response using `openai.ChatCompletion.create`**
- **Learn the structure and format of a conversation in the ChatGPT API, including the use of different roles such as “system”, “user”, and “assistant”.**
- **Practice creating and modifying ChatGPT API calls using various conversation scenarios.**
- **Analyze response created using the ChatGPT API**
- **Generate Multiple Responses**

info

## Make Sure You Know

You are familiar with Python.

## Limitations

This is a gentle introduction. So there is a little bit of Python programming. The information might not be the most up to date as OpenAI releases new features.

# ChatGPT-API

**APIs**, or Application Programming Interfaces, are incredibly useful tools that allow different software systems to communicate and interact with each other. In other words, APIs enable integration between different systems, allowing them to share information and work together more seamlessly. Overall, APIs are a powerful and essential tool for modern software development, enabling faster innovation and greater interoperability between systems. This makes it easier and faster to build new applications, as developers can leverage existing functionality without having to reinvent the wheel.

Developers can now integrate **ChatGPT** and **Whisper** models into their apps and products through our API. In this course, we will practice interacting with the different APIs.

---

With the OpenAI Chat API, developers can create custom applications that leverage the powerful GPT-3.5-turbo and GPT-4 models. These applications can perform a wide range of tasks including:
* Draft an email or other piece of writing
* Write Python code
* Answer questions about a set of documents
* Create conversational agents
* Give your software a natural language interface
* Tutor in a range of subjects
* Translate languages
* Simulate characters for video games and much more

---

The possibilities are virtually endless, and the API offers a flexible and scalable platform for building cutting-edge natural language processing applications. Try copy and pasting the following code:

```
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())
```

# Getting Started

Chat models take a series of messages as input, and return a model-generated message as output.

While the chat format of OpenAI's API is primarily intended to facilitate multi-turn conversations, it can be equally effective for handling single-turn tasks that don't involve any back-and-forth interaction. In fact, this capability makes it a viable alternative to traditional instruction-following models like text-davinci-003, which are optimized for generating a single response to a given prompt. Lets get started, In order to begin interacting with the API, we first need to get our libraries.

```
import os
import openai
import secret
openai.api_key=secret.api_key
```

Next step in order to generate a response we can use `openai.ChatCompletion.create` as a function. The function requires two arguments: A response, and a message. For the model throughout this course, we will use the following:`model="gpt-3.5-turbo"`.

```
response=openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
```

When you use the OpenAI Chat API, you'll need to provide some input for the program to work with. The main input is called "messages," which is a fancy word for a list of things people say to each other in a conversation. You can have as few or as many messages as you want, depending on how long you want the conversation to be.

> Each message in the list should have two pieces of information: who said it (**either the "system," the "user," or the "assistant**") and what they said (the actual words they used).

Lets try generating a response from our api.

```
response=openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
  messages=[
        {"role": "system", "content": "You are a helpful
        assistant."},
        {"role": "user", "content": "can you write me 3 senteces
        on animal behavior."}
    ]
)
```

Typically, a conversation is formatted with a system message first, followed by alternating user and assistant messages.
After pasting the following code, we should get a message saying the code was successfully executed. In order to print, our response we are going to run the following:

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

We use the `print` statement to essentially outputs the text generated by the OpenAI language model in response to the user's message specified in the messages parameter.

Switch out the print statement to the following:

```
print(response)
```

The response generated is longer and a little harder to quickly decipher. However, now we can understand the longer print statement we will use. The `print` statement extracts the text content of the response generated by the OpenAI model. It does this by indexing into the response dictionary using the keys 'choices', 0, 'message', and 'content' to access the text content of the first response choice returned by the model. The `strip()` method is then applied to remove any leading or trailing whitespace from the text content before it is printed to the console.

# Responses

## Response format

Below is a response that was generated using the prompt in our previous page.


```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "message": {
        "content": "Animal behavior can vary greatly depending
on the species and environment they inhabit.",
        "role": "assistant"
      }
    }
  ],
  "created": 1681688074,
  "id": "chatcmpl-766B8BCz36OIR7eLKRFVbIoGO8qgL",
  "model": "gpt-3.5-turbo-0301",
  "object": "chat.completion",
  "usage": {
    "completion_tokens": 14,
    "prompt_tokens": 31,
    "total_tokens": 45
  }
}
```


let’s go through each part of the generated response from the ChatGPT API:
* **choices**: This key contains an array of generated responses. In this case, there’s only one response, but depending on the API settings, there might be more.
* **finish_reason**: Every response will include a finish_reason. This key indicates why the response generation was finished.

Possible values for **finish_reason**:
**“stop”**: Complete model output returned
**“length”**: Incomplete output due to max_tokens or token limit
**“content_filter”**: Content excluded by content filters
**“null”**: Ongoing or incomplete API response

- “**index**”: This key represents the index number of the choice within the “choices” array. As there’s only one choice here, the index is 0.
- “**message**”: This key holds the actual response message from the AI assistant.

It contains two sub-keys:
1. “**content**”: This key holds the text generated by the AI. In this case, it’s a statement about animal behavior.
2. “**role**”: This key indicates the role of the entity sending the message. Here, the role is “assistant,” which represents the AI.

- “**created**”: This key represents the Unix timestamp when the response was generated.
- “**id**”: This key holds a unique identifier for the generated response. It can be useful for tracking, debugging, or referencing specific interactions.
- “**model**”: This key indicates the version of the GPT model used to generate the response. Here, it’s “gpt-3.5-turbo-0301.”
- “**object**”: This key represents the type of object returned by the API. In this case, it’s “chat.completion,” which means a completed chat message.
- “**usage**”: This key provides information about token usage in the API call.

It has three sub-keys:
1. “**completion_tokens**”: The number of tokens used in the generated response (14 in this example).
2. “**prompt_tokens**”: The number of tokens used in the input prompt (31 in this example).
3. “**total_tokens**”: The total number of tokens used in both input prompt and output response (45 in this example).

# Instructing chat models

When you use the OpenAI ChatGPT API, you'll need to provide a list of messages that represent a conversation. Each message has two pieces of information: who said it (either the "system," the "user," or the "assistant") and what they said. You can have as few or as many messages as you want in a conversation. We will cover the format for API calls, explanations, and a new example for you to try.

```
response=openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
  messages=[
        {"role": "system", "content": "You are a financial
advisor."},
        {"role": "user", "content": "What are some tips for
saving money?"},
        {"role": "assistant", "content": "Creating a budget,
reducing expenses, and saving on utilities are some ways to save
money."},
        {"role": "user", "content": "How do I create a budget?"}
    ]
)


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

Usually, a conversation starts with a message from the system, followed by alternating messages from the user and the assistant. The system message sets the behavior of the assistant, while user messages serve as prompts for the assistant's response.

The system message tells the assistant what it should do. For example, you could start with "You are a helpful assistant" as in our previous prompt. In this example, the conversation starts with a system message, setting the role of the AI as a financial advisor.

The user messages give instructions to the assistant. They can be written by end users or set by a developer. In our case, the user then asks for tips on saving money.

The assistant responds with a few general suggestions. Finally, the user asks for more information on creating a budget. This API call will generate a response from the AI, providing guidance on budget creation.

The assistant messages keep track of previous responses. They can also be written by a developer to give examples of desired behavior. Make sure to include the open and closing brackets for the list of messages.

```
 messages=[
        {"role": "system", "content": "You are a financial
advisor."},
        {"role": "user", "content": "What are some tips for
saving money?"},
        {"role": "assistant", "content": "Creating a budget,
reducing expenses, and saving on utilities are some ways to save
money."},
        {"role": "user", "content": "How do I create a budget?"}
    ]
```

It's important to include the entire conversation history if you want to refer to previous messages later on. The models used by the API have no memory of past requests, so all relevant information must be supplied via the conversation. If the conversation is too long for the model to handle, it may need to be shortened in some way. (cannot fit within the model's token limit).

# Implementing Retries

Generating multiple responses and selecting the most appropriate one can improve the quality and relevance of the AI-generated content. In this guide, we will explore the concept of retries and alternative completions in ChatGPT API interactions, learn how to generate multiple responses, and discuss strategies for selecting the best output for a given context.

We can use the n parameter to generate multiple completions for a single user query. The AI will provide 'n' different responses, allowing you to choose the most suitable one. In this case we are going to have it print the whole response in order for us to see the output of the `n` value.

```
response=openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
  messages=[
        {"role": "system", "content": "You are a helpful
        assistant."},
        {"role": "user", "content": "What's a good book to read
        on a rainy day?"}
    ],
  n=3
)

print(response)
```

We can see under choices now we have 3 different results. Manually review the generated alternatives and choose the response that best answers the user query or aligns with your desired output. The API generates three alternative completions for the user query, providing a range of book suggestions. You can then choose the most fitting response based on your criteria or user preferences.

Implementing retries and alternative completions can enhance the quality and relevance of ChatGPT API-generated content. By generating multiple responses and selecting the best completion, you can ensure the AI provides the most suitable answer for a given context or user query. If you want cleaner results and simply showing the 3 different responses. We can run the following code:

```
for i in (response["choices"]):
  print(i["message"]['content'].strip())
  print(" ////////  ")
```

Here we loop through all the `choices` and retrieve the content inside the messages. We added an extra print statement to differentiate between the responses.

# 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