Learning Objectives

Learners will be able to...

- **Effectively generate multiple images using appropriate API calls or programming techniques**
- **Apply appropriate methods for saving multiple generated images in various file formats**.
- **Create Synchronous and Asynchronous API calls**

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.

Chaining Multiple API Calls to Create a Sequence of Images with DALL-E

In the previous lessons, we have learned how to generate images using OpenAI's DALL-E API by defining specific conditions or constraints in our prompts. Today, we will take a step further and learn how to chain multiple API calls together to create a sequence of images. This technique can significantly enhance the creativity and complexity of the outputs we can generate.

Benefits of Chaining

Chaining API calls allows us to generate multiple images in a sequence based on a set of prompts. This technique has several benefits:

Storytelling: By chaining prompts, we can create a sequence of images that tell a story. This can be useful in a variety of applications, from creating storyboard visuals to generating illustrative examples for educational content.

Progressive image generation: Chaining allows us to generate images that progressively change based on our sequence of prompts. This can be used to simulate movement, transformation, or progression over time.

Exploring different variations: By chaining a series of related prompts, we can generate a set of images that show different interpretations or variations of the same concept. This can help us understand how DALL-E interprets and responds to different prompts.

```
import os
import openai
import requests
import secret

openai.api_key=secret.api_key

# Set the prompts
prompts = ["robot dog in a lab", "robot dog exploring the city",
        "robot dog watching the sunset"]

# Generate and save the images
for i, prompt in enumerate(prompts):
    response = openai.Image.create(
      prompt=prompt,
      n=1,
      size="256x256"
    )

    # Get the image URL from the response
    image_url = response['data'][0]['url']

    # Download and save the image
    img_data = requests.get(image_url).content
    with open(f"robot_dog_journey_{i+1}.jpg", 'wb') as handler:
        handler.write(img_data)
```

After clicking the `Try it` make sure to be give it a few more seconds so that the images can be generated even though we see the message that the code was executed.

In this example, we first define a list of prompts. Each prompt represents a different stage of the "robot dog's journey".

We then loop over each prompt in the list. For each prompt, we call the DALL-E API to generate an image that corresponds to that stage of the journey. We specify prompt as the prompt, `n=1` to generate one image per prompt, and `size="256x256"` to specify the size of the generated image.

From the API response, we extract the URL of the generated image. We then use the `requests` library to download the image from this URL.

Finally, we save each image with a unique filename that includes the index of the prompt in the list. This allows us to view the images in the order they were generated, effectively seeing the robot dog's journey unfold in sequence.

By chaining these API calls together, we can generate a sequence of images that tell a visually compelling story. This approach can be extended and adapted to generate sequences of images for a wide range of applications.

Exceptions in DALL-E

While using APIs, it is common to encounter various types of errors and exceptions. These could occur due to a variety of reasons like network issues, incorrect parameters, rate limiting, server errors, etc. you can ensure that your program doesn't crash unexpectedly and provides useful feedback about what went wrong. This is crucial for debugging issues and improving the reliability of your code. This is especially important when making multiple calls at the same time like when chaining.

Here are some common types of errors and exceptions you may encounter while using the DALL-E API:

HTTP Errors: These are errors returned by the server and they come with an HTTP status code. For example, a 404 error means the requested resource could not be found, and a 500 error means there was an internal server error.

API Errors: These are errors returned by the API itself due to issues like incorrect parameters, exceeding rate limits, etc. These usually come with an error message explaining what went wrong.

Network Errors: These are errors that occur due to network issues, like a timeout because the server took too long to respond.

Here is an example where we use a try-except block to catch and handle potential errors and exceptions.

```
import os
import openai
import requests
import secret

openai.api_key=secret.api_key

# Set the prompts
prompts = ["robot dog in a lab", "robot dog exploring the city",
        "robot dog watching the sunset"]

# Generate and save the images
for i, prompt in enumerate(prompts):
    try:
        response = openai.Image.create(
          prompt=prompt,
          n=1,
          size="256x256"
        )

        # Get the image URL from the response
        image_url = response['data'][0]['url']

        # Download and save the image
        img_data = requests.get(image_url).content
        with open(f"robot_dog_journey_{i+1}.jpg", 'wb') as
        handler:
            handler.write(img_data)

    except requests.exceptions.RequestException as e:
        # This will catch any general network error
        print(f"Network error: {e}")

    except openai.api_errors.APIError as e:
        # This will catch any error returned by the OpenAI API
        print(f"API error: {e}")

    except Exception as e:
        # This is a catch-all for any other exceptions
        print(f"Unexpected error: {e}")
```

The `try` block contains the code that could potentially raise an exception. If an exception is raised in the `try` block, the execution immediately moves to the `except` block that handles that specific exception.

The `requests.exceptions.RequestException` except block will handle any network errors that might occur during the API call or while downloading the image.

The `openai.api_errors.APIError` except block will handle any errors returned by the OpenAI API, such as incorrect parameters or exceeding rate limits.

Finally, the general `Exception` except block will catch any other exceptions that the specific catch blocks did not catch. This is a good practice to ensure that your program can recover from any unexpected exceptions.

Async vs Sync

In a typical **synchronous application**, each task must complete before the next one can start. This can be inefficient when dealing with I/O operations like network requests, where the program spends a lot of time waiting for responses.

Asynchronous programming allows you to perform other tasks while waiting for I/O operations to complete, resulting in more efficient use of resources.

Python's `asyncio` library allows you to write single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives.

Asynchronous

While using DALL-E's API, you might need to generate multiple images at once. If you do it synchronously, your application will have to wait for each API call to complete before making the next one. This is inefficient, especially if the API calls take a long time to process.

By making asynchronous API calls, you can send multiple requests to the API at once without having to wait for each one to complete. This can significantly speed up your application if you need to make a lot of API calls.
Synchronous example
In order to compare the time between both calls, we are going to time our previous chain request. Copy and paste the code in this page in the `sync.py` file on the left. To control the time we have to use the following code. The code will start a timer then end it when we are done testing and print the result. here is a sample code, just for your visual learning.

```
import time

start_time = time.time()

# Your existing code here

end_time = time.time()
execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")
```

By using `time.time()` function, we capture the start time before the execution of your code and the end time after the execution. Then, we calculate the difference between the two timestamps to get the execution time in seconds. Finally, we print the execution time.
To keep it consistent let's basically use the same Python code we have been using in this assignment.

```
import os
import openai
import requests
import secret
import time

openai.api_key = secret.api_key

# Set the prompts
prompts = ["robot dog in a lab", "robot dog exploring the city",
        "robot dog watching the sunset"]

start_time = time.time()  # Start measuring execution time

# Generate and save the images
for i, prompt in enumerate(prompts):
    try:
        response = openai.Image.create(
            prompt=prompt,
            n=1,
            size="256x256"
        )

        # Get the image URL from the response
        image_url = response['data'][0]['url']

        # Download and save the image
        img_data = requests.get(image_url).content
        with open(f"robot_dog_journey_{i+1}.jpg", 'wb') as
        handler:
            handler.write(img_data)

    except requests.exceptions.RequestException as e:
        # This will catch any general network error
        print(f"Network error: {e}")

    except openai.api_errors.APIError as e:
        # This will catch any error returned by the OpenAI API
        print(f"API error: {e}")

    except Exception as e:
```

```
        # This is a catch-all for any other exceptions
        print(f"Unexpected error: {e}")

end_time = time.time()  # End measuring execution time
execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")
```

Async Example

Now, let's see how you can use Python's `asyncio` and `aiohttp` libraries to make asynchronous API calls to DALL-E:

The code we are working with in the `async.py` file creates multiple tasks, each one responsible for generating an image and saving it.
These tasks will run concurrently, which can speed up the total execution time compared to running them sequentially. Please be aware that OpenAI's API might have rate limits depending on your plan, so please be cautious about making too many concurrent requests. We are going to do just two big Try It button at the end, in order not to waste a ton of our tokens.

Coding

fetch_and_save_image function: This function is designed to download and save the image from the given URL. The function takes a session (aiohttp's ClientSession), a URL, and a path to save the image. We use aiohttp's `get()` function to make a GET request to the URL, and then use `resp.read()` to read the response content (image data) asynchronously. We then open the file at the given path and write the image data to it.

```
async def fetch_and_save_image(session, url, path):
    try:
        async with session.get(url) as resp:
            img_data = await resp.read()
            with open(path, 'wb') as handler:
                handler.write(img_data)
    except Exception as e:
        print(f"Unexpected error: {e}")
```

fetch_image function: This function makes a request to OpenAI's API to generate an image for the given prompt. The function takes a prompt and an index as input. It calls `openai.Image.create()` to generate an image and gets the image URL from the response. It then creates an aiohttp `ClientSession` and calls `fetch_and_save_image()` to download and save the image.

```
async def fetch_image(prompt, i):
    try:
        response = openai.Image.create(
            prompt=prompt,
            n=1,
            size="256x256"
        )

        # Get the image URL from the response
        image_url = response['data'][0]['url']

        # Download and save the image
        async with aiohttp.ClientSession() as session:
            await fetch_and_save_image(session, image_url,
        f"robot_dog_journey_{i+1}.jpg")
    except openai.api_errors.APIError as e:
        # This will catch any error returned by the OpenAI API
        print(f"API error: {e}")

    except Exception as e:
        # This is a catch-all for any other exceptions
        print(f"Unexpected error: {e}")
```

main function: This function creates an `asyncio` task for each prompt, adds them to a list, and then runs them concurrently using `asyncio.gather()`. This means that it will start all the tasks at the same time, and then wait for all of them to finish. This is where the asynchronous magic happens - rather than processing each prompt one by one, it starts processing all of them at once, which can significantly speed up the total execution time. It also measures the execution time by recording the time before and after the tasks are run, and then subtracts the start time from the end time.

```
async def main():
    start_time = time.time()  # Start measuring execution time

    tasks = []
    for i, prompt in enumerate(prompts):
        tasks.append(fetch_image(prompt, i))

    await asyncio.gather(*tasks)

    end_time = time.time()  # End measuring execution time
    execution_time = end_time - start_time
    print(f"Execution time: {execution_time} seconds")
```

Your whole code outline in `async.py` should look like the following code. Additionally, to run the main function we have added `asyncio.run(main())` at the end of our code.

```
import os
import openai
import aiohttp
import asyncio
import secret
import time

openai.api_key = secret.api_key

# Set the prompts
prompts = ["robot dog in a lab", "robot dog exploring the city",
        "robot dog watching the sunset"]

async def fetch_and_save_image(session, url, path):
    try:
        async with session.get(url) as resp:
            img_data = await resp.read()
            with open(path, 'wb') as handler:
                handler.write(img_data)
    except Exception as e:
        print(f"Unexpected error: {e}")

async def fetch_image(prompt, i):
    try:
        response = openai.Image.create(
            prompt=prompt,
            n=1,
            size="256x256"
        )

        # Get the image URL from the response
        image_url = response['data'][0]['url']

        # Download and save the image
        async with aiohttp.ClientSession() as session:
            await fetch_and_save_image(session, image_url,
        f"robot_dog_journey_{i+1}.jpg")
    except openai.api_errors.APIError as e:
        # This will catch any error returned by the OpenAI API
        print(f"API error: {e}")

    except Exception as e:
        # This is a catch-all for any other exceptions
        print(f"Unexpected error: {e}")
```

```
async def main():
    start_time = time.time()  # Start measuring execution time

    tasks = []
    for i, prompt in enumerate(prompts):
        tasks.append(fetch_image(prompt, i))

    await asyncio.gather(*tasks)

    end_time = time.time()  # End measuring execution time
    execution_time = end_time - start_time
    print(f"Execution time: {execution_time} seconds")

# Run the main function
asyncio.run(main())
```

Use the Try It buttons below to check and compare the sync vs async time.

Please note that the `openai.Image.create` function is a blocking operation and doesn't support async. The code above assumes that this function is asynchronous, which is not correct. To truly parallelize this, you'd need an async-compatible OpenAI library or use something like Python's `concurrent.futures` to run the blocking parts in a separate thread.

Coding Exercise

Write a function `chain()` that takes a list of prompts and generates images based on the prompts. The function should use chaining to generate the images. The function should save the images to the current directory, using the following convention: `test{i+1}.jpg`, where `i` is the index of the prompt in the list. Please make sure the images are `256x256`.

For example, if the list of prompts is `["A cat", "A dog", "A bird"]`, the function should generate three images:

- `test1.jpg`: An image of a cat
- `test2.jpg`: An image of a dog
- `test3.jpg`: An image of a bird
  The function should be able to handle any number of prompts in the list