



DeepLearning.AI

# Data Serialization and Configuration- Driven Development

---

## Module introduction

# Configuration-driven development (CDD)

- Behavior, features and settings of an application are controlled by external configuration files
- More flexible software
- Non-technical teammates can modify the application

# Module overview

- Configuration-driven development overview
- Data serialization in Python and configuration file formats
- Design an app to use the DALL-E API using CDD
- Serialize output



DeepLearning.AI

# Data Serialization and Configuration- Driven Development

---

## Configuration-driven development overview

# Design paradigms

- Structured approach to software architecture and organization
- Some are baked into popular tools
- Others are supported in tools, but optional

# Brainstorming design paradigms

**Prompt** You are an expert on software design paradigms. I am working on building a simple Python-based app that will make calls to the DALL-E API and generate images for users.

The application will be deployed in many different contexts and configurations depending on the end users, and I want my less technical colleagues to be able to do some customization without editing the code itself.

What high level software design paradigms should I consider for this project?

# Configuration-driven development overview

- Behavior controlled through external configuration files
  - e.g. JSON or YAML
- Benefits
  - Non-technical teammates can change application
  - Flexible software for many environments
  - Reduce bugs
- Drawbacks
  - Potential complexity from many configuration files
  - Debugging is more complicated
  - Potential security risks



DeepLearning.AI

# Data Serialization and Configuration- Driven Development

---

## Choosing a configuration file format

# Prompting best practices

- **Be specific:** Provide detail and context about your problem
- **Assign a role:** Assign a role to tailor the output you receive
- **Request an expert opinion:** Assign an expert role and ask the LLM to evaluate the work you've already done to further refine it
- **Give feedback:** Iteratively prompt the LLM and provide feedback on the output you receive to get closer to your expected results

# JSON vs YAML

## JSON

- Support across many languages
- No comment support
- Verbose

## YAML

- High readable
- Supports comments
- Potentially error-prone
- Potential security concerns

## JSON Configuration Example

```
json  
  
{  
    "api": {  
        "key": "your_api_key_here",  
        "endpoint": "https://api.openai.com/v1/dalle/generate",  
        "timeout_seconds": 30  
    },  
    "output": {  
        "directory": "./generated_images",  
        "format": "png",  
        "resolution": {  
            "width": 1024,  
            "height": 1024  
        }  
    },  
    "operations": {  
        "retry_attempts": 3,  
        "logging_level": "info",  
        "enable_watermark": false  
    }  
}
```

## YAML Configuration Example with Comments

```
yaml  
  
api:  
    key: your_api_key_here # API key for authentication  
    endpoint: https://api.openai.com/v1/dalle/generate # DALL-E API endpoint  
    timeout_seconds: 30 # Timeout for API requests  
  
output:  
    directory: ./generated_images # Where to save generated images  
    format: png # Image format (png/jpg)  
    resolution: # Desired resolution of generated images  
        width: 1024  
        height: 1024  
  
operations:  
    retry_attempts: 3 # Number of retries for failed requests  
    logging_level: info # Logging level (debug/info/warning/error)  
    enable_watermark: false # Whether to add a watermark to generated images
```

# Writing data to a file

**Prompt:** How do I write data to a file in Python? I am an experienced developer so just give me the sample code

```
# Writing to a file
with open('example.txt', 'w') as file:
    file.write('Hello, world!')
```

**Follow up:** Can you explain the ‘with’ command?

What does the ‘w’ parameter mean?

Can you show me how to read from this file instead?



DeepLearning.AI

# Data Serialization and Configuration- Driven Development

---

## JSON and pickle

# Data serialization

- Convert data into format that allows for easy storage, movement, and reconstruction
- Different serialization formats used depending on project requirements

You are an expert Python developer who intimately understands the language and its ecosystem. You are also an excellent explainer that helps people understand whatever you tell them, whether they are a novice or expert. Please answer this question: Explain what Pickle is, and why it is useful in Python. Be very succinct.

**Pickle** is a Python module used for serializing and deserializing Python objects. Serialization (or pickling) converts a Python object into a byte stream, while deserialization (or unpickling) converts the byte stream back into a Python object.

#### Why it's useful:

1. **Persistence:** Pickle allows you to save the state of an object to a file, enabling you to store data across program runs.
2. **Data Transfer:** It facilitates transferring Python objects between different systems or over a network by converting them into a byte stream.
3. **Caching:** You can pickle large computations or data processing results to avoid re-computation, improving performance.
4. **Compatibility:** It supports a wide range of Python data types and custom objects, making it versatile for various use cases.

Pickle is particularly useful for quick, simple, and Python-specific serialization needs, though it's not recommended for security-sensitive applications as the byte stream can be manipulated to execute arbitrary code during unpickling.

You are an expert Python developer who intimately understands the language and its ecosystem. You are also an excellent explainer that helps people understand whatever you tell them, whether they are a novice or expert. Please answer this question: Explain what Pickle is, and why it is useful in Python. Be very succinct.

**Pickle** is a Python module used for serializing and deserializing Python objects. Serialization (or pickling) converts a Python object into a byte stream, while deserialization (or unpickling) converts the byte stream back into a Python object.

#### Why it's useful:

1. **Persistence:** Pickle allows you to save the state of an object to a file, enabling you to store data across program runs.
2. **Data Transfer:** It facilitates transferring Python objects between different systems or over a network by converting them into a byte stream.
3. **Caching:** You can pickle large computations or data processing results to avoid re-computation, improving performance.
4. **Compatibility:** It supports a wide range of Python data types and custom objects, making it versatile for various use cases.

Pickle is particularly useful for quick, simple, and Python-specific serialization needs, though it's not recommended for security-sensitive applications as the byte stream can be manipulated to execute arbitrary code during unpickling.

You are an expert Python developer who intimately understands the language and its ecosystem. You are also an excellent explainer that helps people understand whatever you tell them, whether they are a novice or expert. Please answer this question: Explain what Pickle is, and why it is useful in Python. Be very succinct.

**Pickle** is a Python module used for serializing and deserializing Python objects. Serialization (or pickling) converts a Python object into a byte stream, while deserialization (or unpickling) converts the byte stream back into a Python object.

#### Why it's useful:

1. **Persistence:** Pickle allows you to save the state of an object to a file, enabling you to store data across program runs.
2. **Data Transfer:** It facilitates transferring Python objects between different systems or over a network by converting them into a byte stream.
3. **Caching:** You can pickle large computations or data processing results to avoid re-computation, improving performance.
4. **Compatibility:** It supports a wide range of Python data types and custom objects, making it versatile for various use cases.

Pickle is particularly useful for quick, simple, and Python-specific serialization needs, though it's not recommended for security-sensitive applications as the byte stream can be manipulated to execute arbitrary code during unpickling.

You are an expert Python developer who intimately understands the language and its ecosystem. You are also an excellent explainer that helps people understand whatever you tell them, whether they are a novice or expert. Please answer this question: Explain what Pickle is, and why it is useful in Python. Be very succinct.

**Pickle** is a Python module used for serializing and deserializing Python objects. Serialization (or pickling) converts a Python object into a byte stream, while deserialization (or unpickling) converts the byte stream back into a Python object.

#### Why it's useful:

1. **Persistence:** Pickle allows you to save the state of an object to a file, enabling you to store data across program runs.
2. **Data Transfer:** It facilitates transferring Python objects between different systems or over a network by converting them into a byte stream.
3. **Caching:** You can pickle large computations or data processing results to avoid re-computation, improving performance.
4. **Compatibility:** It supports a wide range of Python data types and custom objects, making it versatile for various use cases.

Pickle is particularly useful for quick, simple, and Python-specific serialization needs, though it's not recommended for security-sensitive applications as the byte stream can be manipulated to execute arbitrary code during unpickling.

You are an expert Python developer who intimately understands the language and its ecosystem. You are also an excellent explainer that helps people understand whatever you tell them, whether they are a novice or expert. Please answer this question: Explain what Pickle is, and why it is useful in Python. Be very succinct.

**Pickle** is a Python module used for serializing and deserializing Python objects. Serialization (or pickling) converts a Python object into a byte stream, while deserialization (or unpickling) converts the byte stream back into a Python object.

#### Why it's useful:

1. **Persistence:** Pickle allows you to save the state of an object to a file, enabling you to store data across program runs.
2. **Data Transfer:** It facilitates transferring Python objects between different systems or over a network by converting them into a byte stream.
3. **Caching:** You can pickle large computations or data processing results to avoid re-computation, improving performance.
4. **Compatibility:** It supports a wide range of Python data types and custom objects, making it versatile for various use cases.

Pickle is particularly useful for quick, simple, and Python-specific serialization needs, though it's not recommended for security-sensitive applications as the byte stream can be manipulated to execute arbitrary code during unpickling.

You are an expert Python developer who intimately understands the language and its ecosystem. You are also an excellent explainer that helps people understand whatever you tell them, whether they are a novice or expert. Please answer this question: Explain what Pickle is, and why it is useful in Python. Be very succinct.

**Pickle** is a Python module used for serializing and deserializing Python objects. Serialization (or pickling) converts a Python object into a byte stream, while deserialization (or unpickling) converts the byte stream back into a Python object.

#### Why it's useful:

1. **Persistence:** Pickle allows you to save the state of an object to a file, enabling you to store data across program runs.
2. **Data Transfer:** It facilitates transferring Python objects between different systems or over a network by converting them into a byte stream.
3. **Caching:** You can pickle large computations or data processing results to avoid re-computation, improving performance.
4. **Compatibility:** It supports a wide range of Python data types and custom objects, making it versatile for various use cases.

Pickle is particularly useful for quick, simple, and Python-specific serialization needs, though it's not recommended for security-sensitive applications as the byte stream can be manipulated to execute arbitrary code during unpickling.



DeepLearning.AI

# Data Serialization and Configuration- Driven Development

---

## Using the DALL-E API

Search ⌘ K**API REFERENCE**

Introduction  
Authentication  
Making requests  
Streaming  
Debugging requests

**ENDPOINTS**

Audio  
Chat  
Embeddings  
Fine-tuning  
Batch  
Files  
Uploads

**Images**

**Create image**  
Create image edit  
Create image variation  
The image object

Models

Moderations

**ASSISTANTS**

Assistants  
Threads  
Messages  
Runs  
Run Steps  
Vector Stores

✉ Forum

ⓘ Help

## Create image

POST https://api.openai.com/v1/images/generations

Creates an image given a prompt.

### Request body

**prompt** string **Required**

A text description of the desired image(s). The maximum length is 1000 characters for `dall-e-2` and 4000 characters for `dall-e-3`.

**model** string **Optional** Defaults to `dall-e-2`

The model to use for image generation.

**n** integer or null **Optional** Defaults to 1

The number of images to generate. Must be between 1 and 10. For `dall-e-3`, only `n=1` is supported.

**quality** string **Optional** Defaults to standard

The quality of the image that will be generated. `hd` creates images with finer details and greater consistency across the image. This param is only supported for `dall-e-3`.

**response\_format** string or null **Optional** Defaults to `url`

The format in which the generated images are returned. Must be one of `url` or `b64_json`. URLs are only valid for 60 minutes after the image has been generated.

**size** string or null **Optional** Defaults to `1024x1024`

The size of the generated images. Must be one of `256x256`, `512x512`, or `1024x1024` for `dall-e-2`. Must be one of `1024x1024`, `1792x1024`, or `1024x1792` for `dall-e-3` models.

**style** string or null **Optional** Defaults to `vivid`

The style of the generated images. Must be one of `vivid` or `natural`. Vivid causes the model to lean towards generating hyper-real and dramatic images. Natural causes the model to produce more natural, less hyper-real looking images. This param is only supported for `dall-e-3`.

**user** string **Optional**

A unique identifier representing your end-user, which can help OpenAI to monitor and detect abuse.  
[Learn more](#).

### Returns

Returns a list of `image` objects.

#### Example request

curl ▾

```
1 curl https://api.openai.com/v1/images/generations \
2   -H "Content-type: application/json" \
3   -H "Authorization: Bearer $OPENAI_API_KEY" \
4   -d '{
5     "model": "dall-e-3",
6     "prompt": "A cute baby sea otter",
7     "n": 1,
8     "size": "1024x1024"
9   }'
```

#### Response

curl ▾

```
1 {
2   "created": 1589478378,
3   "data": [
4     {
5       "url": "https://..."
6     },
7     {
8       "url": "https://..."
9     }
10 ]
11 }
```

```
curl https://api.openai.com/v1/images/generations \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $OPENAI_API_KEY" \
-d '{
  "model": "dall-e-3",
  "prompt": "A cute baby sea otter",
  "n": 1,
  "size": "1024x1024"
}'
```

```
curl https://api.openai.com/v1/images/generations \  
-H "Content-Type: application/json" \  
-H "Authorization: Bearer $OPENAI_API_KEY" \  
-d '{  
    "model": "dall-e-3",  
    "prompt": "A cute baby sea otter",  
    "n": 1,  
    "size": "1024x1024"  
}'
```

```
curl https://api.openai.com/v1/images/generations \  
-H "Content-Type: application/json" \  
-H "Authorization: Bearer $OPENAI_API_KEY" \  
-d '{  
    "model": "dall-e-3",  
    "prompt": "A cute baby sea otter",  
    "n": 1,  
    "size": "1024x1024"  
}'
```

```
curl https://api.openai.com/v1/images/generations \  
-H "Content-Type: application/json" \  
-H "Authorization: Bearer $OPENAI_API_KEY" \  
-d '{  
  "model": "dall-e-3",  
  "prompt": "A cute baby sea otter",  
  "n": 1,  
  "size": "1024x1024"  
}'
```

```
{  
  "created": 1589478378,  
  "data": [  
    {  
      "url": "https://..."  
    },  
    {  
      "url": "https://..."  
    }  
  ]  
}
```

```
{  
  "created": 1589478378,  
  "data": [  
    {  
      "ur": "https://..."  
    },  
    {  
      "ur": "https://..."  
    }  
  ]  
}
```

# Code to call DALL-E

**Prompt :** You are a deep expert on the OpenAI ecosystem, including the client libraries, REST endpoints, and any other way of accessing functionality of GPT, DALL-E, and other models.

Please create easy to read, easy to follow code that will call DALL-E to generate an image. The code should be in Python, and all parameters should be in an external file that the Python code reads.

Please use the most up-to-date design pattern.

```
def generate_image(self):
    response = openai.Image.create(
        prompt=self.config['prompt'],
        n=self.config['n'],
        size=self.config['size']
    )
    return response['data']
```

```
def generate_image(self):
    response = openai.Image.create(
        prompt=self.config['prompt'],
        n=self.config['n'],
        size=self.config['size']
    )
    return response['data']
```

`openai.lib._old_api.APIRemovedInV1:`

You tried to access `openai.Image`, but this is no longer supported in `openai>=1.0.0` - see the README at <https://github.com/openai/openai-python> for the API.

You can run ``openai migrate`` to automatically upgrade your codebase to use the `1.0.0` interface.

Alternatively, you can pin your installation to the old version, e.g. ``pip install openai==0.28``

A detailed migration guide is available here:

<https://github.com/openai/openai-python/discussions/742>

# Code to access DALL-E via REST endpoint

**Prompt** Can you create code that accesses the OpenAI endpoint at:  
: <https://platform.openai.com/docs/api-reference/images/create>  
directly to create an image?

```
# Define your OpenAI API key
api_key = "your_openai_api_key"

# Define the endpoint URL
url = "https://api.openai.com/v1/images/generations"

# Set up the headers, including authorization with your API key
headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {api_key}"
}

# Define the payload with your desired prompt and other parameters
payload = {
    "prompt": "A futuristic cityscape at dusk with flying cars and neon lights",
    "n": 1, # Number of images to generate
    "size": "1024x1024" # Size of the generated image
}

# Send a POST request to the OpenAI API
response = requests.post(url, headers=headers, data=json.dumps(payload))
```

```
# Define your OpenAI API key
api_key = "your_openai_api_key"

# Define the endpoint URL
url = "https://api.openai.com/v1/images/generations"

# Set up the headers, including authorization with your API key
headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {api_key}"
}

# Define the payload with your desired prompt and other parameters
payload = {
    "prompt": "A futuristic cityscape at dusk with flying cars and neon lights",
    "n": 1, # Number of images to generate
    "size": "1024x1024" # Size of the generated image
}

# Send a POST request to the OpenAI API
response = requests.post(url, headers=headers, data=json.dumps(payload))
```

```
# Define your OpenAI API key
api_key = "your_openai_api_key"

# Define the endpoint URL
url = "https://api.openai.com/v1/images/generations"

# Set up the headers, including authorization with your API key
headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {api_key}"
}
```

```
# Define the payload with your desired prompt and other parameters
payload = {
    "prompt": "A futuristic cityscape at dusk with flying cars and neon lights",
    "n": 1, # Number of images to generate
    "size": "1024x1024" # Size of the generated image
}
```

```
# Send a POST request to the OpenAI API
response = requests.post(url, headers=headers, data=json.dumps(payload))
```



DeepLearning.AI

# Data Serialization and Configuration- Driven Development

---

## Implementing CDD

# Externalizing parameters

**Prompt:** I have this code that calls the REST endpoint for DALL-E to generate an image. The code has the parameter for the calls hardcoded in it. Can you externalize the parameters in a separate file?

Please also provide an exhaustive list of the parameters in the API docs here:

<https://platform.openai.com/docs/api-reference/images/create>

```
{  
    "api_key": "YOUR API KEY",  
    "url": "https://api.openai.com/v1/images/generations",  
    "headers": {  
        "Content-Type": "application/json"  
    },  
    "payload": {  
        "prompt": "A futuristic cityscape at dusk with flying cars and neon  
lights",  
        "n": 1,  
        "size": "1024x1024",  
        "response_format": "url",  
        "user": "1234"  
    }  
}
```

```
{  
    "api_key": "YOUR API KEY",  
    "url": "https://api.openai.com/v1/images/generations",  
    "headers": {  
        "Content-Type": "application/json"  
    },  
    "payload": {  
        "prompt": "A futuristic cityscape at dusk with flying cars and neon  
lights",  
        "n": 1,  
        "size": "1024x1024",  
        "response_format": "url",  
        "user": "1234"  
    }  
}
```

```
{  
    "api_key": "YOUR API KEY",  
    "url": "https://api.openai.com/v1/images/generations",  
    "headers": {  
        "Content-Type": "application/json"  
    },  
    "payload": {  
        "prompt": "A futuristic cityscape at dusk with flying cars and neon  
lights",  
        "n": 1,  
        "size": "1024x1024",  
        "response_format": "url",  
        "user": "1234"  
    }  
}
```

```
{  
    "api_key": "YOUR API KEY",  
    "url": "https://api.openai.com/v1/images/generations",  
    "headers": {  
        "Content-Type": "application/json"  
    },  
    "payload": {  
        "prompt": "A futuristic cityscape at dusk with flying cars and neon  
lights",  
        "n": 1,  
        "size": "1024x1024",  
        "response_format": "url",  
        "user": "1234"  
    }  
}
```

```
import requests
import json

# Load the configuration from the external file
with open('config.json', 'r') as config_file:
    config = json.load(config_file)

# Extract the API key and URL from the config
api_key = config['api_key']
url = config['url']

# Update headers with the authorization token
headers = config['headers']
headers['Authorization'] = f"Bearer {api_key}"

# Extract the payload from the config
payload = config['payload']
```

```
import requests
import json

# Load the configuration from the external file
with open('config.json', 'r') as config_file:
    config = json.load(config_file)

# Extract the API key and URL from the config
api_key = config['api_key']
url = config['url']

# Update headers with the authorization token
headers = config['headers']
headers['Authorization'] = f"Bearer {api_key}"

# Extract the payload from the config
payload = config['payload']
```

```
import requests
import json

# Load the configuration from the external file
with open('config.json', 'r') as config_file:
    config = json.load(config_file)
```

```
# Extract the API key and URL from the config
api_key = config['api_key']
url = config['url']
```

```
# Update headers with the authorization token
headers = config['headers']
headers['Authorization'] = f"Bearer {api_key}"
```

```
# Extract the payload from the config
payload = config['payload']
```

```
import requests
import json

# Load the configuration from the external file
with open('config.json', 'r') as config_file:
    config = json.load(config_file)

# Extract the API key and URL from the config
api_key = config['api_key']
url = config['url']

# Update headers with the authorization token
headers = config['headers']
headers['Authorization'] = f"Bearer {api_key}"

# Extract the payload from the config
payload = config['payload']
```

```
# Send a POST request to the OpenAI API
response = requests.post(url, headers=headers,
                         data=json.dumps(payload))

# Check if the request was successful
if response.status_code == 200:
    # Parse the response JSON
    data = response.json()

    # Extract the URL of the generated image
    image_url = data['data'][0]['url']

    print(f"Image URL: {image_url}")
else:
    print(f"Failed to generate image: {response.status_code}")
    print(response.text)
```

```
# Send a POST request to the OpenAI API
response = requests.post(url, headers=headers,
                         data=json.dumps(payload))
```

```
# Check if the request was successful
if response.status_code == 200:
    # Parse the response JSON
    data = response.json()

    # Extract the URL of the generated image
    image_url = data['data'][0]['url']

    print(f"Image URL: {image_url}")
else:
    print(f"Failed to generate image: {response.status_code}")
    print(response.text)
```

```
# Send a POST request to the OpenAI API
response = requests.post(url, headers=headers,
                         data=json.dumps(payload))
```

```
# Check if the request was successful
if response.status_code == 200:
    # Parse the response JSON
    data = response.json()

    # Extract the URL of the generated image
    image_url = data['data'][0]['url']

    print(f"Image URL: {image_url}")
else:
    print(f"Failed to generate image: {response.status_code}")
    print(response.text)
```

```
# Send a POST request to the OpenAI API
response = requests.post(url, headers=headers,
                         data=json.dumps(payload))

# Check if the request was successful
if response.status_code == 200:
    # Parse the response JSON
    data = response.json()

    # Extract the URL of the generated image
    image_url = data['data'][0]['url']

    print(f"Image URL: {image_url}")
else:
    print(f"Failed to generate image: {response.status_code}")
    print(response.text)
```

# Externalizing parameters

**Prompt** Can you modify this code to fix two issues:

:

First, regardless of how many images we ask for, you only parse out 1 image from the response data.

Second, can you download the images from the URL and save them locally. Add the desired filename to the JSON file, and save them as <filename>1.PNG, <filename>2.PNG etc.

```
{  
  "api_key": "YOUR API KEY",  
  "url": "https://api.openai.com/v1/images/generations",  
  "headers": {  
    "Content-Type": "application/json"  
  },  
  "payload": {  
    "prompt": "A futuristic cityscape at dusk with flying cars and neon lights",  
    "n": 1,  
    "size": "1024x1024",  
    "response_format": "url",  
    "user": "1234"  
  }  
  "download": {  
    "filename": "generated_image"  
  }  
}
```

```
if response.status_code == 200:  
    data = response.json()  
  
filename = config['download']['filename']  
  
# Iterate over the generated images and save them  
for i, image_info in enumerate(data['data']):  
    image_url = image_info['url']  
    image_response = requests.get(image_url)  
  
    if image_response.status_code == 200:  
        # Save the image with the desired filename and index  
        image_filename = f"{filename}{i+1}.png"  
        with open(image_filename, 'wb') as img_file:  
            img_file.write(image_response.content)  
        print(f"Image saved as {image_filename}")
```

```
if response.status_code == 200:  
    data = response.json()  
  
filename = config['download']['filename']  
  
# Iterate over the generated images and save them  
for i, image_info in enumerate(data['data']):  
    image_url = image_info['url']  
    image_response = requests.get(image_url)  
  
    if image_response.status_code == 200:  
        # Save the image with the desired filename and index  
        image_filename = f"{filename}{i+1}.png"  
        with open(image_filename, 'wb') as img_file:  
            img_file.write(image_response.content)  
        print(f"Image saved as {image_filename}")
```

```
if response.status_code == 200:  
    data = response.json()  
  
filename = config['download']['filename']  
  
# Iterate over the generated images and save them  
for i, image_info in enumerate(data['data']):  
    image_url = image_info['url']  
    image_response = requests.get(image_url)  
  
    if image_response.status_code == 200:  
        # Save the image with the desired filename and index  
        image_filename = f"{filename}{i+1}.png"  
        with open(image_filename, 'wb') as img_file:  
            img_file.write(image_response.content)  
        print(f"Image saved as {image_filename}")
```

```
if response.status_code == 200:  
    data = response.json()  
  
filename = config['download']['filename']  
  
# Iterate over the generated images and save them  
for i, image_info in enumerate(data['data']):  
    image_url = image_info['url']  
    image_response = requests.get(image_url)  
  
    if image_response.status_code == 200:  
        # Save the image with the desired filename and index  
        image_filename = f"{filename}{i+1}.png"  
        with open(image_filename, 'wb') as img_file:  
            img_file.write(image_response.content)  
        print(f"Image saved as {image_filename}")
```

```
(dalle) laurence@laurences-mini test2 % python3 gen_img.py  
Image saved as generated_image1.png  
Image saved as generated_image2.png  
Image saved as generated_image3.png
```



# Adding a timeout

**Prompt:** Update the configuration file and app code to include a timeout for the API calls.

# Activity: Add more parameters

- Review the DALL-E documentation and find additional parameters you wish to add like ‘quality’ or ‘style’
- Work with an LLM to add these parameters to the configuration file and then use them in the source code



# Data Serialization and Configuration- Driven Development

---

## Serializing results

# Adding parameters and serializing results

**Prompt:** Can you make 2 more modifications:

First can you add the style and quality parameters to the payload configuration, and load them to pass to the backend?

Then, after the files are downloaded, can you pickle them up along with the configuration file?

```
{  
  "api_key": "your_openai_api_key",  
  "url": "https://api.openai.com/v1/images/generations",  
  "headers": {  
    "Content-Type": "application/json"  
  },  
  "payload": {  
    "prompt": "A futuristic cityscape at dusk with flying cars and neon lights",  
    "n": 3,  
    "size": "1024x1024",  
    "response_format": "url",  
    "user": null,  
    "style": "photorealistic",  
    "quality": "high"  
  },  
  "download": {  
    "filename": "generated_image"  
  },  
  "timeout": 30  
}
```

**style** string or null Optional Defaults to vivid

The style of the generated images. Must be one of `vivid` or `natural`. Vivid causes the model to lean towards generating hyper-real and dramatic images. Natural causes the model to produce more natural, less hyper-real looking images. This param is only supported for `dall-e-3`.

```
# Dictionary to store images and config for pickling
pickle_data = {
    "config": config,
    "images": {}
}

...
# Within the loop parsing the images
pickle_data['images'][image_filename] = image_response.content

# Pickle the configuration and images
pickle_filename = f"{filename}_data.pkl"
with open(pickle_filename, 'wb') as pickle_file:
    pickle.dump(pickle_data, pickle_file)
```

```
# Dictionary to store images and config for pickling
```

```
pickle_data = {  
    "config": config,  
    "images": {}  
}
```

...

```
# Within the loop parsing the images
```

```
    pickle_data['images'][image_filename] = image_response.content
```

```
# Pickle the configuration and images
```

```
    pickle_filename = f"{filename}_data.pkl"  
    with open(pickle_filename, 'wb') as pickle_file:
```

```
        pickle.dump(pickle_data, pickle_file)
```

```
# Dictionary to store images and config for pickling
pickle_data = {
    "config": config,
    "images": {}
}
```

```
...
# Within the loop parsing the images
pickle_data['images'][image_filename] = image_response.content
```

```
# Pickle the configuration and images
pickle_filename = f"{filename}_data.pkl"
with open(pickle_filename, 'wb') as pickle_file:
    pickle.dump(pickle_data, pickle_file)
```

```
# Dictionary to store images and config for pickling
pickle_data = {
    "config": config,
    "images": {}
}

...
# Within the loop parsing the images
pickle_data['images'][image_filename] = image_response.content
```

```
# Pickle the configuration and images
pickle_filename = f"{filename}_data.pkl"
with open(pickle_filename, 'wb') as pickle_file:
    pickle.dump(pickle_data, pickle_file)
```



DeepLearning.AI

# Databases

---

## Module introduction



DeepLearning.AI

# Databases

---

Setting up a  
simple database

# Building a lightweight database setup



SQLAlchemy

# Getting advice on setting up a database

**Prompt:** Suggest a lightweight but powerful python setup that uses a file-based database so that I can practice working with databases. I want the code to be able to run in google colab.

```
!pip install sqlalchemy
```



Small. Fast. Reliable.  
Choose any three.

Home About Documentation Download License Support Purchase

Search

## What Is SQLite?

SQLite is a C-language library that implements a [small](#), [fast](#), [self-contained](#), [high-reliability](#), [full-featured](#), SQL database engine. SQLite is the [most used](#) database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day. [More Information...](#)

The SQLite [file format](#) is stable, cross-platform, and backwards compatible and the developers pledge to keep it that way [through the year 2050](#). SQLite database files are commonly used as containers to transfer rich content between systems [\[1\]](#) [\[2\]](#) [\[3\]](#) and as a long-term archival format for data [\[4\]](#). There are over 1 trillion (1e12) SQLite databases in active use [\[5\]](#).

SQLite [source code](#) is in the [public-domain](#) and is free to everyone to use for any purpose.

## Latest Release

[Version 3.46.1](#) (2024-08-13). [Download](#) [Prior Releases](#)

### Common Links

- [Features](#)
- [When to use SQLite](#)
- [Getting Started](#)
- [Try it live!](#)
- [Prior Releases](#)
- [SQL Syntax](#)
  - [Pragmas](#)
  - [SQL functions](#)
  - [Date & time functions](#)
  - [Aggregate functions](#)
  - [Window functions](#)
  - [Math functions](#)
  - [JSON functions](#)
- [C/C++ Interface Spec](#)
  - [Introduction](#)
  - [List of C-language APIs](#)
- [The TCL Interface Spec](#)
- [Quirks and Gotchas](#)
- [Frequently Asked Questions](#)
- [Commit History](#)
- [Bugs](#)
- [News](#)

```
from sqlalchemy import create_engine  
  
engine = create_engine('sqlite:///ecommerce.db', echo=True)
```

# Checking that the connection is live

**Prompt:** Here is code that I'm using to create a database in a jupyter notebook using sqlalchemy and sqlite:

```
engine = create_engine('sqlite:///ecommerce.db', echo=True)
```

Give me some simple code that I can use to check that this connection is live.



To check if the connection to your SQLite database is live using SQLAlchemy, you can execute a simple query. A common approach is to execute a raw SQL command like `SELECT 1`, which doesn't affect the database but confirms that the connection is active.

Here's a simple code snippet to check if the connection is live:

```
python Copy code
from sqlalchemy import create_engine, text

# Assuming you've already created your engine
engine = create_engine('sqlite:///ecommerce.db', echo=True)

# Check if the connection is live
with engine.connect() as connection:
    result = connection.execute(text("SELECT 1"))
    # Fetch and print the result
    print(result.scalar()) # This should print 1 if the connection is successful
```

### Explanation:

- `engine.connect()`: Establishes a connection to the database.
- `connection.execute(text("SELECT 1"))`: Executes a simple SQL query that returns 1.
- `result.scalar()`: Retrieves the scalar result from the query, which should be `1`.

If the code runs without errors and prints `1`, it confirms that the connection to the SQLite database is live.



DeepLearning.AI

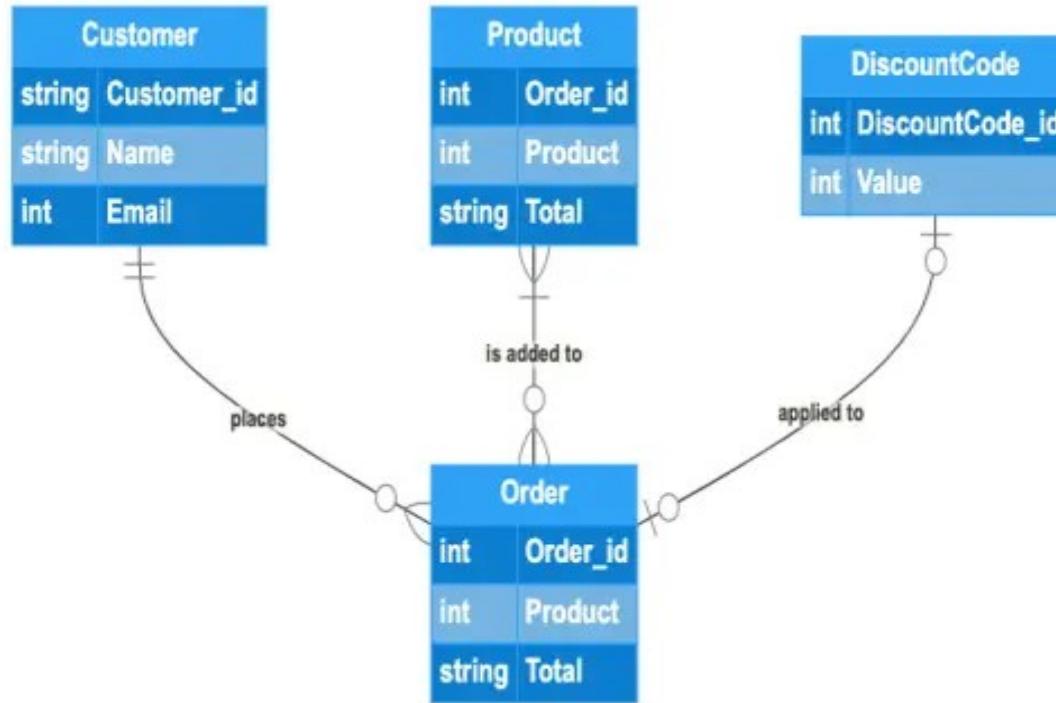
# Databases

---

Design and implement  
a database schema



# Entity-Relationship Diagrams

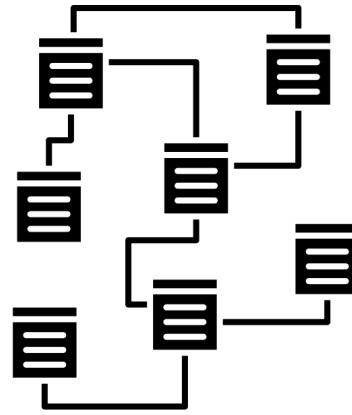
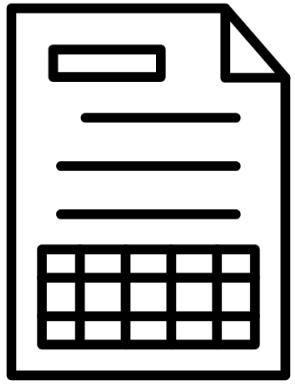


```
SET SQL_MODE="NO_AUTO_VALUE_ON_ZERO";
SET time_zone = "+00:00";
```

```
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS *,
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
```

```
--  
-- Database: `Sql627181_3`  
--
```

# LLMs can design schema from text descriptions



Business requirements

Database schema

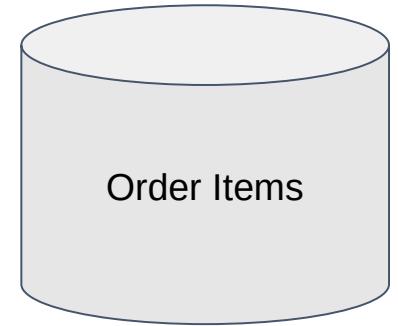
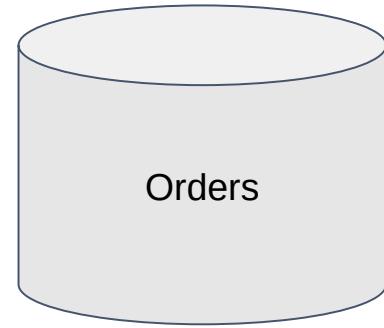
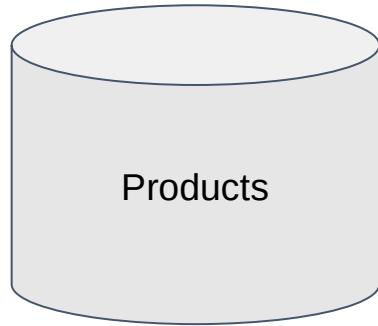
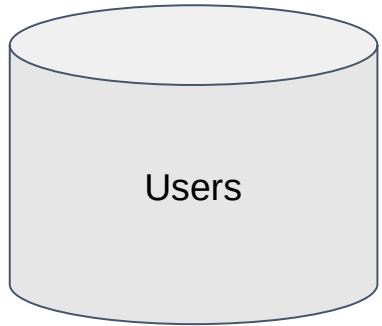
# Prompting LLMs to design database schema

**Prompt:** I need to develop a database for a social network.

The database will track individual people, and store information about them including their name, age, hometown, and hobbies.

The database will also store relationships between people, including family relationships, friendships, work colleagues, and schoolmates.

Suggest a schema that would work for this.



# Creating the e-commerce schema

**Prompt:** Design a database schema for an e-commerce application with tables for users, products, orders, and order\_items.

To design a database schema for an e-commerce application, you can create the following tables:

1. Users:

- id (Primary Key)
- name
- email (Unique)

2. Products:

- id (Primary Key)
- name
- price

3. Orders:

- id (Primary Key)
- user\_id (Foreign Key referencing Users)

4. Order\_items:

- id (Primary Key)
- order\_id (Foreign Key referencing Orders)
- product\_id (Foreign Key referencing Products)
- quantity

# Creating the e-commerce schema

**Prompt:** Design a database schema for an e-commerce application with tables for users, products, orders, and order\_items.

Try specifying specific attribute names, or allowed data types for columns, and see how the schema changes!

To design a database schema for an e-commerce application, you can create the following tables:

1. Users:

- id (Primary Key)
- name
- email (Unique)

3. Orders:

- id (Primary Key)
- user\_id (Foreign Key referencing Users)

2. Products:

- id (Primary Key)
- name
- price

4. Order\_items:

- id (Primary Key)
- order\_id (Foreign Key referencing Orders)
- product\_id (Foreign Key referencing Products)
- quantity

# Prompting LLMs to design database schema

**Prompt:** I have a sqlite database set up in Python using SQLAlchemy.  
Help me implement tables for the following schema:

1. Users:  
- id (Primary Key)  
- name  
- email (Unique)

2. Products:  
- id (Primary Key)  
- name  
- price

3. Orders:  
- id (Primary Key)  
- user\_id (Foreign Key referencing Users)

4. Order\_items:  
- id (Primary Key)  
- order\_id (Foreign Key referencing Orders)  
- product\_id (Foreign Key referencing Products)  
- quantity

```
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey

metadata = MetaData()

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('email', String, unique=True))

products = Table('products', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('price', Integer))

metadata.create_all(engine)
```

```
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey

metadata = MetaData()

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('email', String, unique=True))

products = Table('products', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('price', Integer))

metadata.create_all(engine)
```

```
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey

metadata = MetaData()

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('email', String, unique=True))

products = Table('products', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('price', Integer))

metadata.create_all(engine)
```



DeepLearning.AI

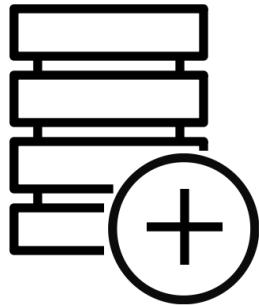
# Databases

---

## Implementing CRUD operations

# CRUD

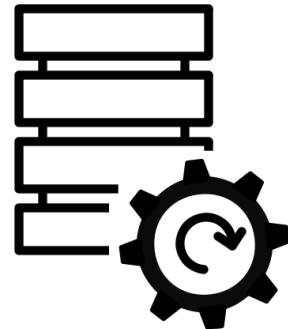
**Create**



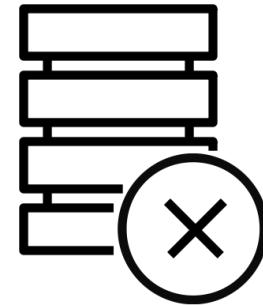
**Read**



**Update**

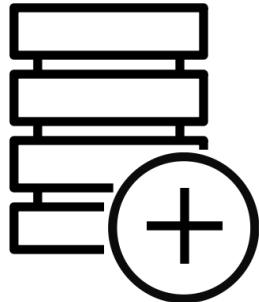


**Delete**



# CRUD

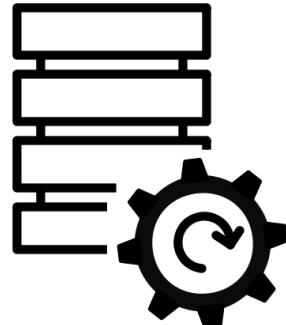
**Create**



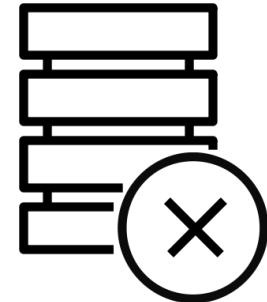
**Read**



**Update**



**Delete**



# The create operation

**Prompt:** What are the things to consider when you implement the create operation for a database?



When implementing the "Create" operation for a database, several important considerations should be taken into account:

1. **Data Validation:** Ensure that the data being inserted meets the required criteria, such as data types, formats, and constraints (e.g., not null, unique values). This helps prevent invalid data from entering the database.
2. **Security:** Implement measures to protect against SQL injection and other security vulnerabilities. Use prepared statements or ORM (Object-Relational Mapping) tools to handle input data safely.
3. **Error Handling:** Properly handle errors that may occur during the insertion process, such as constraint violations or connection issues. Provide meaningful feedback to the user or system when something goes wrong.
4. **Transaction Management:** If the "Create" operation is part of a larger transaction, ensure that the operation is atomic. If something fails, the entire transaction should be rolled back to maintain database consistency.
5. **Concurrency Control:** Consider how the database handles multiple simultaneous "Create" operations. Implement locking or use database features to manage concurrent inserts without causing conflicts.
6. **Indexing:** Ensure that appropriate indexes are in place to support efficient retrieval of newly inserted records, especially if the "Create" operation is frequently performed.
7. **Audit Logging:** If necessary, implement logging to track when and by whom records are created, which is useful for auditing and debugging purposes.
8. **Performance:** Consider the performance impact of the "Create" operation, especially in high-volume systems. Batch inserts and optimized database configurations can help manage performance issues.

# The create operation

**Prompt:** How can I create a function to add a new user to the users table of this database?

```
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey

metadata = MetaData()

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('email', String, unique=True))

products = Table('products', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('price', Integer))

...
```

```
import sqlite3

def add_record(db_path, table_name, data):
    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()
    placeholders = ', '.join('?' * len(data))

    sql = f"INSERT INTO {table_name} VALUES ({placeholders})"

    ...

```

```
from sqlalchemy.orm import sessionmaker  
  
Session = sessionmaker(bind=engine)  
session = Session()  
  
def add_user(name, email):  
    new_user = users.insert().values(name=name, email=email)  
    session.execute(new_user)  
    session.commit()  
  
# Test the function  
add_user('John Doe', 'john.doe@example.com')
```

```
from sqlalchemy.orm import sessionmaker
```

```
Session = sessionmaker(bind=engine)  
session = Session()
```

```
def add_user(name, email):  
    new_user = users.insert().values(name=name, email=email)  
    session.execute(new_user)  
    session.commit()
```

```
# Test the function
```

```
add_user('John Doe', 'john.doe@example.com')
```



```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()

def add_user(name, email):
    new_user = users.insert().values(name=name, email=email)
    session.execute(new_user)
    session.commit()

# Test the function
add_user('John Doe', 'john.doe@example.com')
```

```
2024-09-13 16:56:11,698 INFO sqlalchemy.engine.Engine BEGIN (implicit)
INFO:sqlalchemy.engine.Engine:BEGIN (implicit)
2024-09-13 16:56:11,704 INFO sqlalchemy.engine.Engine INSERT INTO users (name, email) VALUES (?, ?)
INFO:sqlalchemy.engine.Engine:INSERT INTO users (name, email) VALUES (?, ?)
2024-09-13 16:56:11,712 INFO sqlalchemy.engine.Engine [generated in 0.00798s] ('John Doe', 'john.doe@example.com')
INFO:sqlalchemy.engine.Engine:[generated in 0.00798s] ('John Doe', 'john.doe@example.com')
2024-09-13 16:56:11,717 INFO sqlalchemy.engine.Engine COMMIT
INFO:sqlalchemy.engine.Engine:COMMIT
```



```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()

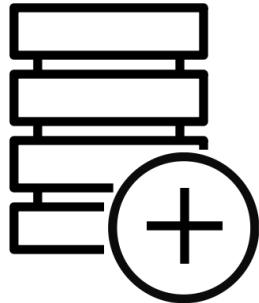
def add_user(name, email):
    new_user = users.insert().values(name=name, email=email)
    session.execute(new_user)
    session.commit()

# Test the function
add_user('John Doe', 'john.doe@example.com')
```

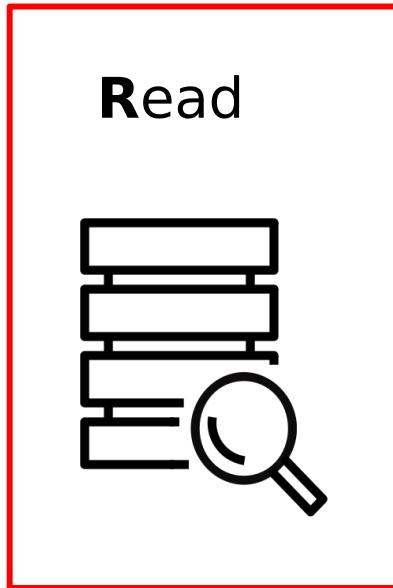
```
2024-09-13 16:56:11,698 INFO sqlalchemy.engine.Engine BEGIN (implicit)
INFO:sqlalchemy.engine.Engine:BEGIN (implicit)
2024-09-13 16:56:11,704 INFO sqlalchemy.engine.Engine INSERT INTO users (name, email) VALUES (?, ?)
INFO:sqlalchemy.engine.Engine:INSERT INTO users (name, email) VALUES (?, ?)
2024-09-13 16:56:11,712 INFO sqlalchemy.engine.Engine [generated in 0.00798s] ('John Doe', 'john.doe@example.com')
INFO:sqlalchemy.engine.Engine:[generated in 0.00798s] ('John Doe', 'john.doe@example.com')
2024-09-13 16:56:11,717 INFO sqlalchemy.engine.Engine COMMIT
INFO:sqlalchemy.engine.Engine:COMMIT
```

# CRUD

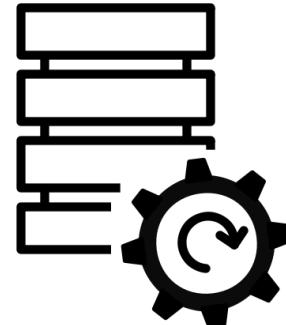
**Create**



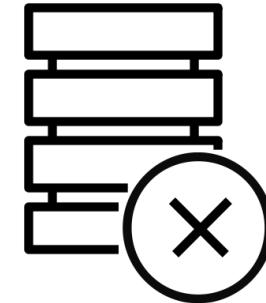
**Read**



**Update**



**Delete**



# Implementing read functionality

**Prompt:** Generate code that implements a read function for this database.  
It should return a list of all users currently in the database:

```
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey

metadata = MetaData()

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('email', String, unique=True))

products = Table('products', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('price', Integer))

...
```

```
def get_all_users():
    query = users.select()
    result = session.execute(query)
    return result.fetchall()

# Test the function
all_users = get_all_users()
print(all_users)
```



```
def get_all_users():
    query = users.select()
    result = session.execute(query)
    return result.fetchall()

# Test the function
all_users = get_all_users()
print(all_users)
```

```
→ 2024-09-13 19:21:29,947 INFO sqlalchemy.engine.Engine BEGIN (implicit)
INFO:sqlalchemy.engine.Engine:BEGIN (implicit)
2024-09-13 19:21:29,963 INFO sqlalchemy.engine.Engine SELECT users.id, users.name, users.email
FROM users
INFO:sqlalchemy.engine.Engine:SELECT users.id, users.name, users.email
FROM users
2024-09-13 19:21:29,978 INFO sqlalchemy.engine.Engine [generated in 0.01394s] ()
INFO:sqlalchemy.engine.Engine:[generated in 0.01394s] ()
[(1, 'John Doe', 'john.doe@example.com')]
```



```
def get_all_users():
    query = users.select()
    result = session.execute(query)
    return result.fetchall()

# Test the function
all_users = get_all_users()
print(all_users)
```

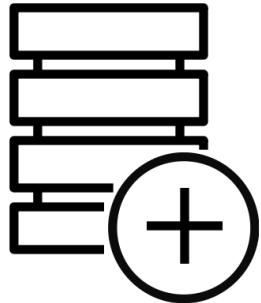
```
→ 2024-09-13 19:21:29,947 INFO sqlalchemy.engine.Engine BEGIN (implicit)
INFO:sqlalchemy.engine.Engine:BEGIN (implicit)
2024-09-13 19:21:29,963 INFO sqlalchemy.engine.Engine SELECT users.id, users.name, users.email
FROM users
INFO:sqlalchemy.engine.Engine:SELECT users.id, users.name, users.email
FROM users
2024-09-13 19:21:29,978 INFO sqlalchemy.engine.Engine [generated in 0.01394s] ()
INFO:sqlalchemy.engine.Engine:[generated in 0.01394s] ()
[(1, 'John Doe', 'john.doe@example.com')]
```

# Summary of the state of the database

- You can **create** a record using the `add_user()` function
- You can **read** all the user records using the `get_all_users()` function
- The database contains a single user record for “John Doe”, with name and email fields populated

# CRUD

**Create**



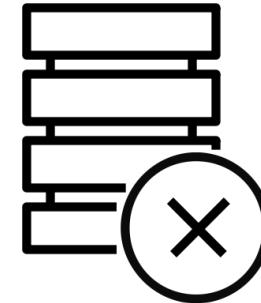
**Read**



**Update**



**Delete**



# Prompt

**Prompt:** Generate code that implements an update function for this database. I want to use it to update the email address of a user.

```
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey

metadata = MetaData()

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('email', String, unique=True))

products = Table('products', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('price', Integer))

...
```

```
def update_user_email(user_id, new_email):
    update = users.update().where(
        users.c.id == user_id).values(email=new_email)
    session.execute(update)
    session.commit()
```

## # Test the function

```
update_user_email(1, 'john.new@example.com')
```

```
def update_user_email(user_id, new_email):  
    update = users.update().where(  
        users.c.id == user_id).values(email=new_email)  
    session.execute(update)  
    session.commit()
```

## # Test the function

```
update_user_email(1, 'john.new@example.com')
```

```
def update_user_email(user_id, new_email):  
    update = users.update().where(  
        users.c.id == user_id).values(email=new_email)  
    session.execute(update)  
    session.commit()
```

## # Test the function

```
update_user_email(1, 'john.new@example.com')
```



```
▶ def update_user_email(user_id, new_email):
    update = users.update().where(
        users.c.id == user_id).values(email=new_email)
    session.execute(update)
    session.commit()

# Test the function
update_user_email(1, 'john.new@example.com')
```

```
→ 2024-09-13 19:23:58,175 INFO sqlalchemy.engine.Engine UPDATE users SET email=? WHERE users.id = ?
INFO:sqlalchemy.engine.Engine:UPDATE users SET email=? WHERE users.id = ?
2024-09-13 19:23:58,182 INFO sqlalchemy.engine.Engine [generated in 0.00916s] ('john.new@example.com', 1)
INFO:sqlalchemy.engine.Engine:[generated in 0.00916s] ('john.new@example.com', 1)
2024-09-13 19:23:58,187 INFO sqlalchemy.engine.Engine COMMIT
INFO:sqlalchemy.engine.Engine:COMMIT
```



```
all_users = get_all_users()  
print(all_users)
```

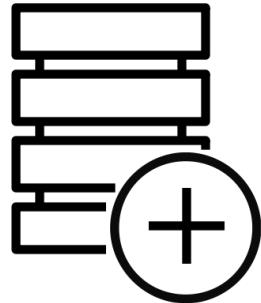


```
2024-09-13 19:28:13,000 INFO sqlalchemy.engine.Engine BEGIN (implicit)  
INFO:sqlalchemy.engine.Engine:BEGIN (implicit)  
2024-09-13 19:28:13,004 INFO sqlalchemy.engine.Engine SELECT users.id, users.name, users.email  
FROM users  
INFO:sqlalchemy.engine.Engine:SELECT users.id, users.name, users.email  
FROM users  
2024-09-13 19:28:13,011 INFO sqlalchemy.engine.Engine [cached since 403s ago] ()  
INFO:sqlalchemy.engine.Engine:[cached since 403s ago] ()  
[(1, 'John Doe', 'john.new@example.com')]
```

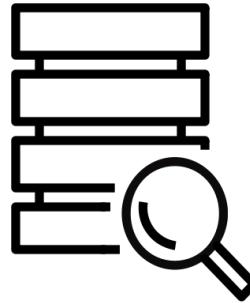


# CRUD

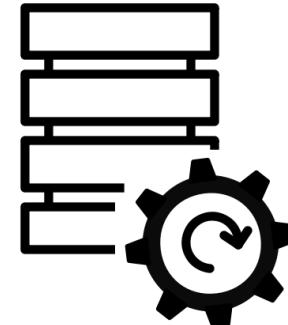
**Create**



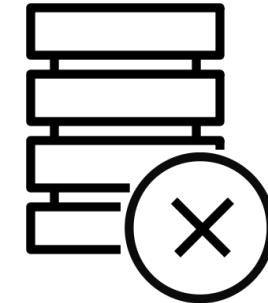
**Read**



**Update**



**Delete**



```
def delete_user(user_id):  
    delete = users.delete().where(users.c.id == user_id)  
    session.execute(delete)  
    session.commit()
```

# Test the function

```
delete_user(1)
```



```
def delete_user(user_id):
    delete = users.delete().where(users.c.id == user_id)
    session.execute(delete)
    session.commit()

# Test the function
delete_user[1]
```

```
2024-09-13 19:31:09,566 INFO sqlalchemy.engine.Engine DELETE FROM users WHERE users.id = ?
INFO:sqlalchemy.engine.Engine:DELETE FROM users WHERE users.id = ?
2024-09-13 19:31:09,573 INFO sqlalchemy.engine.Engine [generated in 0.00784s] (1,)
INFO:sqlalchemy.engine.Engine:[generated in 0.00784s] (1,)
2024-09-13 19:31:09,578 INFO sqlalchemy.engine.Engine COMMIT
INFO:sqlalchemy.engine.Engine:COMMIT
```



```
all_users = get_all_users()  
print(all_users)
```



```
2024-09-13 19:31:56,881 INFO sqlalchemy.engine.Engine BEGIN (implicit)  
INFO:sqlalchemy.engine.Engine:BEGIN (implicit)  
2024-09-13 19:31:56,889 INFO sqlalchemy.engine.Engine SELECT users.id, users.name, users.email  
FROM users  
INFO:sqlalchemy.engine.Engine:SELECT users.id, users.name, users.email  
FROM users  
2024-09-13 19:31:56,893 INFO sqlalchemy.engine.Engine [cached since 626.9s ago] ()  
INFO:sqlalchemy.engine.Engine:[cached since 626.9s ago] ()  
[]
```



```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()

def add_user(name, email):
    new_user = users.insert().values(name=name, email=email)
    session.execute(new_user)
    session.commit()

# Test the function
add_user('John Doe', 'john.doe@example.com')
```

```
2024-09-13 16:56:11,698 INFO sqlalchemy.engine.Engine BEGIN (implicit)
INFO:sqlalchemy.engine.Engine:BEGIN (implicit)
2024-09-13 16:56:11,704 INFO sqlalchemy.engine.Engine INSERT INTO users (name, email) VALUES (?, ?)
INFO:sqlalchemy.engine.Engine:INSERT INTO users (name, email) VALUES (?, ?)
2024-09-13 16:56:11,712 INFO sqlalchemy.engine.Engine [generated in 0.00798s] ('John Doe', 'john.doe@example.com')
INFO:sqlalchemy.engine.Engine:[generated in 0.00798s] ('John Doe', 'john.doe@example.com')
2024-09-13 16:56:11,717 INFO sqlalchemy.engine.Engine COMMIT
INFO:sqlalchemy.engine.Engine:COMMIT
```

# Checking for vulnerabilities

**Prompt:** I'm using this code in a SQL database. Is there a chance that it's vulnerable to SQL injection attack as SQLAlchemy appears to use SQL under-the-hood?

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()

def add_user(name, email):
    new_user = users.insert().values(name=name, email=email)
    session.execute(new_user)
    session.commit()
```



I'm using this code in a SQL database. Is there a chance that it's vulnerable to SQL injection attack as SQLAlchemy appears to use SQL Under-the-hood?

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()

def add_user(name, email):
    new_user = users.insert().values(name=name, email=email)
    session.execute(new_user)
    session.commit()

# Test the function
add_user('John Doe', 'john.doe@example.com')
```



Yes, the code you provided is potentially vulnerable to SQL injection attacks. Even though SQLAlchemy is an ORM (Object-Relational Mapper) that typically handles SQL injection risks well, your current approach directly constructs SQL expressions using the `values()` method, which can be vulnerable if user inputs are not properly sanitized or bound.

To mitigate this risk, it's best to use SQLAlchemy's parameterized queries or ORM methods that automatically handle input sanitization. Here's a safer approach using SQLAlchemy's ORM:



Yes, the code you provided is potentially vulnerable to SQL injection attacks. Even though SQLAlchemy is an ORM (Object-Relational Mapper) that typically handles SQL injection risks well, your current approach directly constructs SQL expressions using the `values()` method, which can be vulnerable if user inputs are not properly sanitized or bound.

To mitigate this risk, it's best to use SQLAlchemy's parameterized queries or ORM methods that automatically handle input sanitization. Here's a safer approach using SQLAlchemy's ORM:

```
from sqlalchemy.orm import sessionmaker
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String
```

```
Base = declarative_base()
```

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    email = Column(String)
```

```
engine = create_engine('sqlite:///ecommerce.db', echo=True)
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
session = Session()
```

```
def add_user(name, email):
    new_user = User(name=name, email=email)
    session.add(new_user)
    session.commit()
```

```
# Test the function
```

```
add_user('John Doe', 'john.doe@example.com')
```



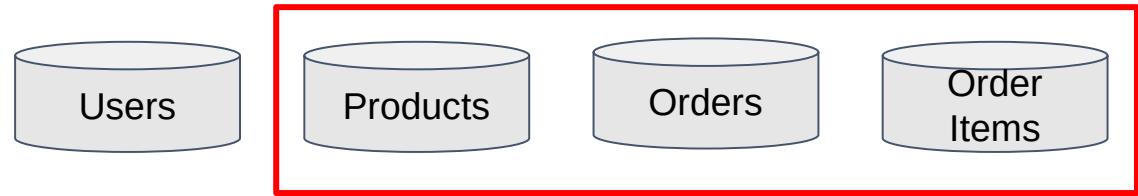
```
▶ all_users = get_all_users()  
print(all_users)
```

```
→ 2024-09-13 19:50:00,736 INFO sqlalchemy.engine.Engine BEGIN (implicit)  
INFO:sqlalchemy.engine.Engine:BEGIN (implicit)  
2024-09-13 19:50:00,741 INFO sqlalchemy.engine.Engine SELECT users.id, users.name, users.email  
FROM users  
INFO:sqlalchemy.engine.Engine:SELECT users.id, users.name, users.email  
FROM users  
2024-09-13 19:50:00,744 INFO sqlalchemy.engine.Engine [cached since 1711s ago] ()  
INFO:sqlalchemy.engine.Engine:[cached since 1711s ago] ()  
[(1, 'John Doe', 'john.doe@example.com')]
```



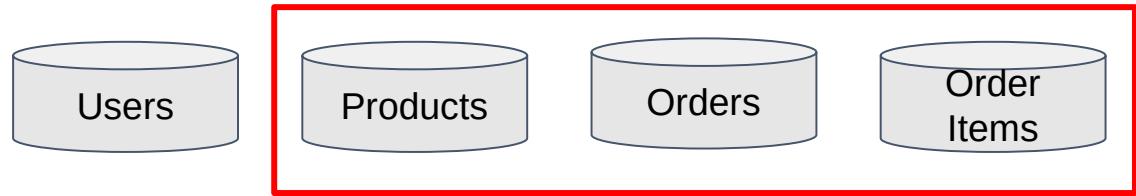
# Hands-on exercise

- Build our CRUD operations for the remaining tables in the database
  - Products
  - Orders
  - Order Items



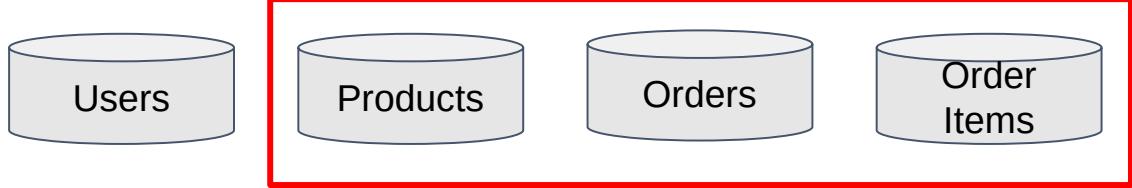
# Hands-on exercise

- Build our CRUD operations for the remaining tables in the database
  - Products
  - Orders
  - Order Items
- Next add test data



# Hands-on exercise

- Build our CRUD operations for the remaining tables in the database
  - Products
  - Orders
  - Order Items
- Next add test data
- Then test against data to check everything works





DeepLearning.AI

# Databases

---

## Module exercise

```
def add_sample_data():
    # Adding users
    user_data = [
        ("Alice", "alice@example.com"),
        ("Bob", "bob@example.com"),
        ("Charlie", "charlie@example.com"),
        ("David", "david@example.com"),
        ("Eve", "eve@example.com"),
        ("Frank", "frank@example.com"),
        ("Grace", "grace@example.com"),
        ("Heidi", "heidi@example.com"),
        ("Ivan", "ivan@example.com"),
        ("Judy", "judy@example.com")
    ]
    for name, email in user_data:
        add_user(name, email)

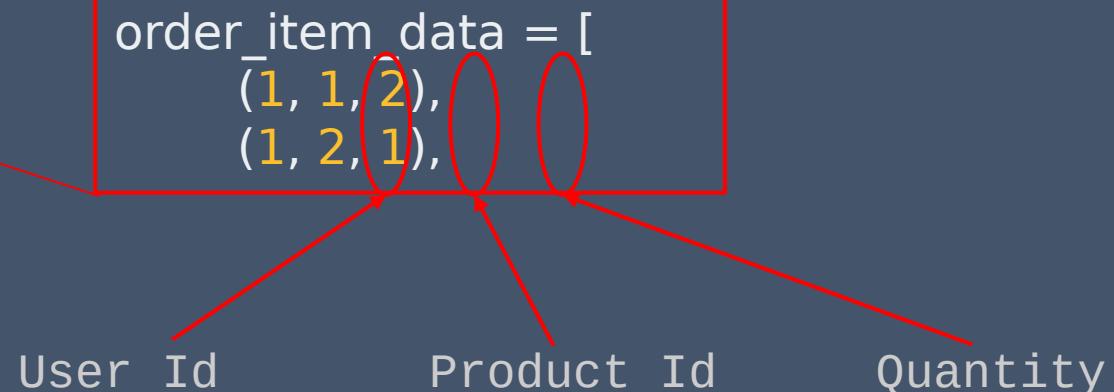
    # Adding products
    product_data = [
        ("Laptop", 1000),
        ("Smartphone", 500),
        ("Tablet", 300),
        ("Monitor", 150),
        ("Keyboard", 50),
        ("Mouse", 25),
        ("Printer", 200),
        ("Router", 75),
        ("Webcam", 60),
        ("Headphones", 80)
    ]
    for name, price in product_data:
        add_product(name, price)
```

```
# Adding orders
for i in range(1, 11):
    add_order(i)

order_item_data = [
    (1, 1, 2),
    (1, 2, 1),
    (2, 3, 1),
    (2, 4, 2),
    (3, 5, 3),
    (3, 6, 1),
    (4, 7, 2),
    (4, 8, 1),
    (5, 9, 1),
    (5, 10, 2),
    (6, 1, 1),
    (6, 2, 2),
    (7, 3, 1),
    (7, 4, 1),
    (8, 5, 2),
    (8, 6, 1),
    (9, 7, 3),
    (9, 8, 1),
    (10, 9, 1),
    (10, 10, 1)
]
for order_id, product_id, quantity in order_item_data:
    add_order_item(order_id, product_id, quantity)
```

```
# Adding orders
for i in range(1, 11):
    add_order(i)

order_item_data = [
    (1, 1, 2),
    (1, 2, 1),
    (2, 3, 1),
    (2, 4, 2),
    (3, 5, 3),
    (3, 6, 1),
    (4, 7, 2),
    (4, 8, 1),
    (5, 9, 1),
    (5, 10, 2),
    (6, 1, 1),
    (6, 2, 2),
    (7, 3, 1),
    (7, 4, 1),
    (8, 5, 2),
    (8, 6, 1),
    (9, 7, 3),
    (9, 8, 1),
    (10, 9, 1),
    (10, 10, 1)
]
for order_id, product_id, quantity in order_item_data:
    add_order_item(order_id, product_id, quantity)
```



```
# Adding orders
for i in range(1, 11):
    add_order(i)

order_item_data = [
    (1, 1, 2),
    (1, 2, 1),
    (2, 3, 1),
    (2, 4, 2),
    (3, 5, 3),
    (3, 6, 1),
    (4, 7, 2), (4, 7, 2)
    (4, 8, 1),
    (5, 9, 1),
    (5, 10, 2),
    (6, 1, 1),
    (6, 2, 2),
    (7, 3, 1),
    (7, 4, 1),
    (8, 5, 2),
    (8, 6, 1),
    (9, 7, 3), (9, 7, 3)
    (9, 8, 1),
    (10, 9, 1),
    (10, 10, 1)
]
for order_id, product_id, quantity in order_item_data:
    add_order_item(order_id, product_id, quantity)
```

2 Printers Ordered

3 Printers Ordered



DeepLearning.AI

# Databases

---

## Advanced querying

# Asking an LLM to write SQL queries

**Prompt:** Write a SQLAlchemy query to find all orders placed by a specific user in the e-commerce database.

# Asking an LLM to write SQL queries (new chat)

**Prompt:** Write a SQLAlchemy query to find all orders placed by a specific user in the following e-commerce database.

Paste your code into the prompt!

```
from sqlalchemy import create_engine, Column, Integer, String, Float, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, sessionmaker

# Define the SQLite database
engine = create_engine('sqlite:///ecommerce.db')
Base = declarative_base()

# Define the Users table
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)

    email = Column(String, unique=True, nullable=False)
```

```
from sqlalchemy.orm import joinedload

# Function to retrieve all orders by a particular user along with the items in each order
def get_orders_by_user(user_id):
    user = session.query(User)
        .options(joinedload(User.orders)
                  .joinedload(Order.order_items)
                  .joinedload(OrderItem.product))
        .filter(User.id == user_id).first()
    if user:
        return user.orders
    else:
        return None
```

```
from sqlalchemy.orm import joinedload

# Function to retrieve all orders by a particular user along with the items in each order
def get_orders_by_user(user_id):
    user = session.query(User)
        .options(joinedload(User.orders)
                  .joinedload(Order.order_items)
                  .joinedload(OrderItem.product))
        .filter(User.id == user_id).first()
    if user:
        return user.orders
    else:
        return None
```

# Schema of order item table

## 1. Users:

- id (Primary Key)
- name
- email (Unique)

## 2. Products:

- id (Primary Key)
- name
- price

## 3. Orders:

- id (Primary Key)
- user\_id (Foreign Key referencing Users)

## 4. Order\_items:

- id (Primary Key)
- order\_id (Foreign Key referencing Orders)
- product\_id (Foreign Key referencing Products)
- quantity

# Prompting for more complex SQL queries

**Prompt:** Write a SQLAlchemy query to find the total quantity of each product sold in this database.

```
from sqlalchemy import func

def get_total_quantity_sold():
    query = session.query(order_items.c.product_id,
        func.sum(order_items.c.quantity).label('total_quantity'))
        .group_by(order_items.c.product_id)
    result = session.execute(query)
    return result.fetchall()

# Test the function
total_quantity_sold = get_total_quantity_sold()
print(total_quantity_sold)
```

```
from sqlalchemy import func

def get_total_quantity_sold():
    query = session.query(order_items.c.product_id,
        func.sum(order_items.c.quantity).label('total_quantity'))
        .group_by(order_items.c.product_id)
    result = session.execute(query)
    return result.fetchall()

# Test the function
total_quantity_sold = get_total_quantity_sold()
print(total_quantity_sold)
```

```
from sqlalchemy import func

def get_total_quantity_sold():
    query = session.query(order_items.c.product_id,
        func.sum(order_items.c.quantity).label('total_quantity'))
        .group_by(order_items.c.product_id)
    result = session.execute(query)
    return result.fetchall()
```

```
# Test the function
total_quantity_sold = get_total_quantity_sold()
print(total_quantity_sold)
```

# Prompting for more complex SQL queries

**Prompt:** Write a SQLAlchemy query that prints out the date and status of each order made by all users.

## SQLAlchemy Query

python

 Copy code

```
def get_all_orders(session):
    """
    Retrieves and prints the date and status of each order made by all users.

    :param session: SQLAlchemy session object.
    """

    orders = session.query(Order.order_date, Order.status).all()

    for order_date, status in orders:
        print(f"Order Date: {order_date}, Status: {status}")

# Example usage
if __name__ == "__main__":
    session = Session() # Replace with your actual session instance
    get_all_orders(session)
```

## Explanation:

- `session.query(Order.order\_date, Order.status)` : Initiates a query that selects the `order\_date` and `status` columns from the `Order` table.
- `all()` : Executes the query and returns all rows as a list of tuples.
- **Looping through results**: The loop iterates over the result set, printing each order's date and status.

## Expected Output:

For each order, you will see an output like:

yaml

 Copy code

```
Order Date: 2024-08-15 12:00:00, Status: pending
Order Date: 2024-08-16 09:30:00, Status: shipped
Order Date: 2024-08-17 15:45:00, Status: delivered
```

# Prompting for more complex SQL queries

**Prompt:** This query isn't doing exactly what I wanted. I'd like to print out the order history for each user id, then the date and status of each of their orders.

```
def print_order_history(session):
    """
    Retrieves and prints the order history for each user, displaying the user ID,
    and the date and status of each order they made.

    :param session: SQLAlchemy session object.
    """

    # Query to get all users with their orders
    users_with_orders = session.query(User.user_id, Order.order_date, Order.status)
        .join(Order, User.user_id == Order.user_id) \
        .order_by(User.user_id, Order.order_date) \
        .all()

    # Group the orders by user and print them out
    current_user_id = None
    for user_id, order_date, status in users_with_orders:
        if user_id != current_user_id:
            print(f"\nUser ID: {user_id}")
            current_user_id = user_id
        print(f"  Order Date: {order_date}, Status: {status}")
```

```
def print_order_history(session):
    """
    Retrieves and prints the order history for each user, displaying the user ID,
    and the date and status of each order they made.

    :param session: SQLAlchemy session object.
    """

    # Query to get all users with their orders
    users_with_orders = session.query(User.user_id, Order.order_date, Order.status) \
        .join(Order, User.user_id == Order.user_id) \
        .order_by(User.user_id, Order.order_date) \
        .all()

    # Group the orders by user and print them out
    current_user_id = None
    for user_id, order_date, status in users_with_orders:
        if user_id != current_user_id:
            print(f"\nUser ID: {user_id}")
            current_user_id = user_id
        print(f"  Order Date: {order_date}, Status: {status}")
```

```
def print_order_history(session):
    """
    Retrieves and prints the order history for each user, displaying the user ID,
    and the date and status of each order they made.

    :param session: SQLAlchemy session object.
    """

    # Query to get all users with their orders
    users_with_orders = session.query(User.user_id, Order.order_date, Order.status) \
        .join(Order, User.user_id == Order.user_id) \
        .order_by(User.user_id, Order.order_date) \
        .all()

    # Group the orders by user and print them out
    current_user_id = None
    for user_id, order_date, status in users_with_orders:
        if user_id != current_user_id:
            print(f"\nUser ID: {user_id}")
            current_user_id = user_id
        print(f"  Order Date: {order_date}, Status: {status}")
```

# TODO: Add output image like in previous lessons

# Using your SQL expertise to guide the LLM

**Prompt:** Create a composite index on the Order table that I can use to execute a range query with filtering.

python

 Copy code

```
from sqlalchemy import Index, Column, Integer, ForeignKey, TIMESTAMP, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Order(Base):
    __tablename__ = 'orders'

    order_id = Column(Integer, primary_key=True, autoincrement=True)
    user_id = Column(Integer, ForeignKey('users.user_id'), nullable=False)
    order_date = Column(TIMESTAMP, nullable=False)
    status = Column(String, nullable=False, default='pending')

    # Define the composite index on user_id and order_date
    __table_args__ = (
        Index('idx_user_order_date', 'user_id', 'order_date'),
    )

# Example usage
if __name__ == "__main__":
```





DeepLearning.AI

# Databases

---

## Database optimization

# Using an LLM to optimize your database

**Prompt:** What are some best practices to improve the performance of this database?

**Note:** this assumes you are continuing your ongoing database chat with the LLM, so that the code and full context are available to the model.

# Optimizing the database

- Schema Design
- Indexing
- Query Optimization
- Caching
- Connection Management
- Monitoring and Maintenance

# Digging deeper with follow up prompts

**Prompt:** What are the best practices for indexing in a SQL database?

Best practices for indexing in a SQL database include:

1. Index columns that are frequently used in WHERE clauses.
2. Index columns that are used in JOIN operations.
3. Avoid indexing columns that have a high number of unique values, such as GUIDs.
4. Consider the size of the index; large indexes can slow down INSERT, UPDATE, and DELETE operations.
5. Regularly monitor and maintain your indexes, removing those that are not used.

To create an index in SQLAlchemy, you can use the following syntax:

```
from sqlalchemy import Index
```

```
Index('ix_products_name', products.c.name)
```

Best practices for indexing in a SQL database include:

1. Index columns that are frequently used in WHERE clauses.
2. Index columns that are used in JOIN operations.
3. Avoid indexing columns that have a high number of unique values, such as GUIDs.
4. Consider the size of the index; large indexes can slow down INSERT, UPDATE, and DELETE operations.
5. Regularly monitor and maintain your indexes, removing those that are not used.

To create an index in SQLAlchemy, you can use the following syntax:

```
from sqlalchemy import Index
```

```
Index('ix_products_name', products.c.name)
```

Best practices for indexing in a SQL database include:

1. Index columns that are frequently used in WHERE clauses.
2. Index columns that are used in JOIN operations.
3. Avoid indexing columns that have a high number of unique values, such as GUIDs.
4. Consider the size of the index; large indexes can slow down INSERT, UPDATE, and DELETE operations.
5. Regularly monitor and maintain your indexes, removing those that are not used.

To create an index in SQLAlchemy, you can use the following syntax:

```
from sqlalchemy import Index
```

```
Index('ix_products_name', products.c.name)
```

Best practices for indexing in a SQL database include:

1. Index columns that are frequently used in WHERE clauses.
2. Index columns that are used in JOIN operations.
3. Avoid indexing columns that have a high number of unique values, such as GUIDs.
4. Consider the size of the index; large indexes can slow down INSERT, UPDATE, and DELETE operations.
5. Regularly monitor and maintain your indexes, removing those that are not used.

To create an index in SQLAlchemy, you can use the following syntax:

```
from sqlalchemy import Index
```

```
Index('ix_products_name', products.c.name)
```

# Optimizing the database

- Schema Design
- Indexing
- Query Optimization
- Caching
- Connection Management
- Monitoring and Maintenance

# Improving performance with caching

**Prompt:** How can I implement basic query caching in SQLAlchemy?

To implement basic query caching in SQLAlchemy, you can use the `dogpile.cache` library. First, install the library:

```
```bash
```

```
!pip install dogpile.cache
```

```
from dogpile.cache import make_region

region = make_region().configure(
    'dogpile.cache.memory',
    expiration_time=3600
)

@region.cache_on_arguments()
def get_all_products():
    query = products.select()
    result = session.execute(query)
    return result.fetchall()

# Test the function
all_products = get_all_products()
print(all_products)
```

```
from dogpile.cache import make_region
```

```
region = make_region().configure(  
    'dogpile.cache.memory',  
    expiration_time=3600  
)
```

```
@region.cache_on_arguments()  
def get_all_products():  
    query = products.select()  
    result = session.execute(query)  
    return result.fetchall()
```

```
# Test the function  
all_products = get_all_products()  
print(all_products)
```

```
from dogpile.cache import make_region

region = make_region().configure(
    'dogpile.cache.memory',
    expiration_time=3600
)
```

```
@region.cache_on_arguments()
def get_all_products():
    query = products.select()
    result = session.execute(query)
    return result.fetchall()
```

```
# Test the function
all_products = get_all_products()

print(all_products)
```

```
from dogpile.cache import make_region

region = make_region().configure(
    'dogpile.cache.memory',
    expiration_time=3600
)

@region.cache_on_arguments()
def get_all_products():
    query = products.select()
    result = session.execute(query)
    return result.fetchall()
```

```
# Test the function
all_products = get_all_products()

print(all_products)
```

```
[7] # Measure execution time without caching
# Clear the cache first
region.invalidate()

start_time_no_cache = time.time()
all_products_no_cache = get_all_products()
end_time_no_cache = time.time()
print(f"Execution time without cache: {end_time_no_cache - start_time_no_cache} seconds")

# Measure execution time with caching
start_time_with_cache = time.time()
all_products_with_cache = get_all_products()
end_time_with_cache = time.time()
print(f"Execution time with cache: {end_time_with_cache - start_time_with_cache} seconds")

# Print the products
print(all_products_no_cache)
```

→ 2024-09-13 22:03:40,730 INFO sqlalchemy.engine.Engine BEGIN (implicit)
INFO:sqlalchemy.engine.Engine:BEGIN (implicit)
2024-09-13 22:03:40,736 INFO sqlalchemy.engine.Engine SELECT products.id, products.name, products.price
FROM products
INFO:sqlalchemy.engine.Engine:SELECT products.id, products.name, products.price
FROM products
2024-09-13 22:03:40,742 INFO sqlalchemy.engine.Engine [generated in 0.00561s] ()
INFO:sqlalchemy.engine.Engine:[generated in 0.00561s] ()
Execution time without cache: 0.016511917114257812 seconds
Execution time with cache: 0.00021004676818847656 seconds

# Optimizing the database

- Schema Design
- Indexing
- Query Optimization
- Caching
- Connection Management
- Monitoring and Maintenance

# Choosing the best data types

**Prompt:** What are the best practices for choosing data types in a SQL database?

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    email = Column(String, unique=True)

    orders = relationship("Order", back_populates="user")
```

# Optimizing the database

- Schema Design
- Indexing
- Query Optimization
- Caching
- Connection Management
- Monitoring and Maintenance



DeepLearning.AI

# Databases

---

## Debugging

# Using an LLM to debug database errors

**Prompt:** How do I handle database connection errors in SQLAlchemy?



How do I handle database connection errors in SQLAlchemy?



In SQLAlchemy, to handle database connection errors, you can catch exceptions like `OperationalError` from `sqlalchemy.exc`. Here's an example:

python

Copy code

```
from sqlalchemy.exc import OperationalError

try:
    engine.connect()
except OperationalError as e:
    print(f"Error connecting to database: {e}")
```

This approach allows you to catch errors related to operational failures, such as a database being unavailable or unreachable. You can handle different exceptions depending on the nature of the error. For connection-specific errors, `OperationalError` is commonly used. You can also implement retries or logging to handle errors more gracefully.



Message ChatGPT



ChatGPT can make mistakes. Check important info.

```
from sqlalchemy.exc import OperationalError  
  
try:  
    engine.connect()  
except OperationalError as e:  
    print(f"Error connecting to database: {e}")
```

```
from sqlalchemy.exc import OperationalError  
  
try:  
    engine.connect()  
except OperationalError as e:  
    print(f"Error connecting to database: {e}")
```

# Handling edge cases

**Prompt:** How do I handle duplicate entry errors in SQLAlchemy?



In SQLAlchemy, you can handle duplicate entry errors using `IntegrityError`, which is raised when constraints, such as unique constraints, are violated. Here's how you can catch and handle such errors:

python

Copy code

```
from sqlalchemy.exc import IntegrityError

def add_user(name, email):
    try:
        new_user = User(name=name, email=email)
        session.add(new_user)
        session.commit()
    except IntegrityError as e:
        session.rollback()
        print(f"Error: {e}")
```

By catching `IntegrityError`, you can handle cases where a user tries to insert a duplicate entry (for example, if the email is already in use). After catching the exception, rolling back the session ensures that the database remains in a consistent state.

&lt; 2/2 &gt; ⏪ ⏴ ⏵ ⏵ ⏵ ⏵



Message ChatGPT



ChatGPT can make mistakes. Check important info.



```
from sqlalchemy.exc import IntegrityError

def add_user(name, email):
    try:
        new_user = User(name=name, email=email)
        session.add(new_user)
        session.commit()
    except IntegrityError as e:
        session.rollback()
        print(f"Error: {e}")
```

```
from sqlalchemy.exc import IntegrityError

def add_user(name, email):
    try:
        new_user = User(name=name, email=email)
        session.add(new_user)
        session.commit()
    except IntegrityError as e:
        session.rollback()
        print(f"Error: {e}")
```

# Tracking down bugs with logging

**Prompt:** How do I enable SQL query logging in SQLAlchemy?



To enable SQL query logging in SQLAlchemy, you can use Python's `logging` module to configure logging for the SQLAlchemy engine. Here's an example setup:

python Copy code

```
import logging

logging.basicConfig()
logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)

# Test logging with a simple query
def get_all_users():
    return session.query(User).all()

# Execute the function to see the logged SQL query
all_users = get_all_users()
print(all_users)
```

In this setup, `logging.basicConfig()` sets up the basic configuration for logging, and `logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)` enables logging of SQL queries. When you execute a query, the SQL generated by SQLAlchemy will be printed to the console at the `INFO` level.



Message ChatGPT



ChatGPT can make mistakes. Check important info.



```
import logging

logging.basicConfig()
logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)

# Test logging with a simple query
def get_all_users():
    return session.query(User).all()

# Execute the function to see the logged SQL query
all_users = get_all_users()
print(all_users)
```

# Logging to a file

**Prompt:** Modify the logging setup to save the log output to a file on disk.



To modify the logging setup to save the SQLAlchemy log output to a file, you can specify a file handler in the logging configuration. Here's how you can do it:

python Copy code

```
import logging

# Set up logging to save to a file
logging.basicConfig(filename='sqlalchemy_log.txt', level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')
logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)
```

```
# Test logging with a simple query
def get_all_users():
    return session.query(User).all()

# Execute the function to log the SQL query
all_users = get_all_users()
print(all_users)
```

This setup will save the log output, including SQL queries, to a file named `sqlalchemy_log.txt` in the current working directory. The `format` argument customizes the format of the log messages, including timestamps for better tracking.

Message ChatGPT



ChatGPT can make mistakes. Check important info.



# Handling transaction errors

**Prompt:** How do I handle transaction errors in SQLAlchemy?

How do I handle transaction errors in SQLAlchemy?



To handle transaction errors in SQLAlchemy, you can use a `try-except` block to catch any exceptions that may occur during a transaction. Here's an example of how to handle transaction errors:

python

Copy code

```
def update_user_email(user_id, new_email):
    try:
        session.query(User).filter(User.id == user_id).update({'email': new_email})
        session.commit()
    except Exception as e:
        session.rollback()
        print(f"Error: {e}")
```

In this example, if an error occurs during the `update` or `commit` operation, the exception is caught, and `session.rollback()` is called to revert the transaction. This ensures that the database stays consistent, and any partial changes are not persisted.



Message ChatGPT



ChatGPT can make mistakes. Check important info.



```
def update_user_email(user_id, new_email):  
    try:  
        session.query(User).filter(User.id == user_id).update({'email': new_email})  
        session.commit()  
  
    except Exception as e:  
        session.rollback()  
  
        print(f"Error: {e}")
```

```
def update_user_email(user_id, new_email):  
    try:  
        session.query(User).filter(User.id == user_id).update({'email': new_email})  
        session.commit()  
  
    except Exception as e:  
        session.rollback()  
        print(f"Error: {e}")
```

# Debugging queries

**Prompt:** How do I use the EXPLAIN feature in SQLAlchemy to debug queries?

To use the `EXPLAIN` feature in SQLAlchemy to debug queries, you can execute raw SQL commands and retrieve the query plan from the database. Here's an example of how to do that:

python

Copy code

```
def explain_query(query):
    result = session.execute(f"EXPLAIN {query}")
    return result.fetchall()

# Example usage
query = "SELECT * FROM users WHERE id = 1"
explain_result = explain_query(query)
print(explain_result)
```

In this example, `EXPLAIN` is used to provide insights into how the database will execute the query. This can help you understand the performance of the query and identify potential bottlenecks. The `fetchall()` method retrieves and prints the query plan. You can use this approach to analyze and optimize SQL queries in your application.



Message ChatGPT



ChatGPT can make mistakes. Check important info.

?

```
def explain_query(query):  
    result = session.execute(f"EXPLAIN {query}")  
    return result.fetchall()
```

## # Example usage

```
query = "SELECT * FROM users WHERE id = 1"  
explain_result = explain_query(query)  
print(explain_result)
```

```
def explain_query(query):  
    result = session.execute(text(f"EXPLAIN {query}"))  
    return result.fetchall()
```

```
# Example usage  
query = "SELECT * FROM users WHERE id = 1"  
explain_result = explain_query(query)  
print(explain_result)
```

```
def explain_query(query):  
    result = session.execute(text(f"EXPLAIN {query}"))  
    return result.fetchall()
```

## # Example usage

```
query = "SELECT * FROM users WHERE id = 1"  
explain_result = explain_query(query)  
print(explain_result)
```

```
def explain_query(query):  
    result = session.execute(text(f"EXPLAIN {query}"))  
    return result.fetchall()
```

## # Example usage

```
query = "SELECT * FROM users WHERE id = 1"  
explain_result = explain_query(query)  
print(explain_result)
```

```
def explain_query(query):  
    result = session.execute(text(f"EXPLAIN {query}"))  
    return result.fetchall()
```

# Example usage

```
query = "SELECT * FROM users WHERE id = 1"
```

```
explain_result = explain_query(query)  
print(explain_result)
```

```
[(0, 'Init', 0, 9, 0, None, 0, None),  
(1, 'OpenRead', 0, 2, 0, '3', 0, None),  
(2, 'Integer', 1, 1, 0, None, 0, None),  
(3, 'SeekRowid', 0, 8, 1, None, 0, None),  
(4, 'Rowid', 0, 2, 0, None, 0, None),  
(5, 'Column', 0, 1, 3, None, 0, None),  
(6, 'Column', 0, 2, 4, None, 0, None),  
(7, 'ResultRow', 2, 3, 0, None, 0, None),  
(8, 'Halt', 0, 0, 0, None, 0, None),  
(9, 'Transaction', 0, 0, 4, '0', 1, None),  
(10, 'Goto', 0, 1, 0, None, 0, None)]
```

# Debugging queries

**Prompt:** An EXPLAIN query gave me this output -- can you please tell me what it means?

```
[(0, 'Init', 0, 9, 0, None, 0, None),  
 (1, 'OpenRead', 0, 2, 0, '3', 0, None),  
 (2, 'Integer', 1, 1, 0, None, 0, None),  
 (3, 'SeekRowid', 0, 8, 1, None, 0, None),  
 (4, 'Rowid', 0, 2, 0, None, 0, None),  
 (5, 'Column', 0, 1, 3, None, 0, None),  
 (6, 'Column', 0, 2, 4, None, 0, None),  
 (7, 'ResultRow', 2, 3, 0, None, 0, None),  
 (8, 'Halt', 0, 0, 0, None, 0, None),  
 (9, 'Transaction', 0, 0, 4, '0', 1, None),  
 (10, 'Goto', 0, 1, 0, None, 0, None)]
```



The output you provided is from an SQLite `EXPLAIN` command, which shows the virtual machine (VM) instructions that SQLite will execute for your query. Here's a breakdown of each instruction:

1. **Init (0, 9)**: Initializes the VM. Execution will start at instruction 9.
2. **OpenRead (0, 2, 3)**: Opens a B-Tree (table or index) for reading. Cursor 0 is used to read from a table that has root page 3, with a 2nd column key lookup.
3. **Integer (1, 1)**: Loads the integer value 1 into register 1.
4. **SeekRowid (0, 8, 1)**: Seeks the cursor 0 to the row with the Row ID stored in register 1. If the row is not found, jumps to instruction 8.
5. **Rowid (0, 2)**: Loads the Row ID of the current row in cursor 0 into register 2.
6. **Column (0, 1, 3)**: Loads the value of the 1st column from the current row of cursor 0 into register 3.
7. **Column (0, 2, 4)**: Loads the value of the 2nd column from the current row of cursor 0 into register 4.



DeepLearning.AI

# Databases

---

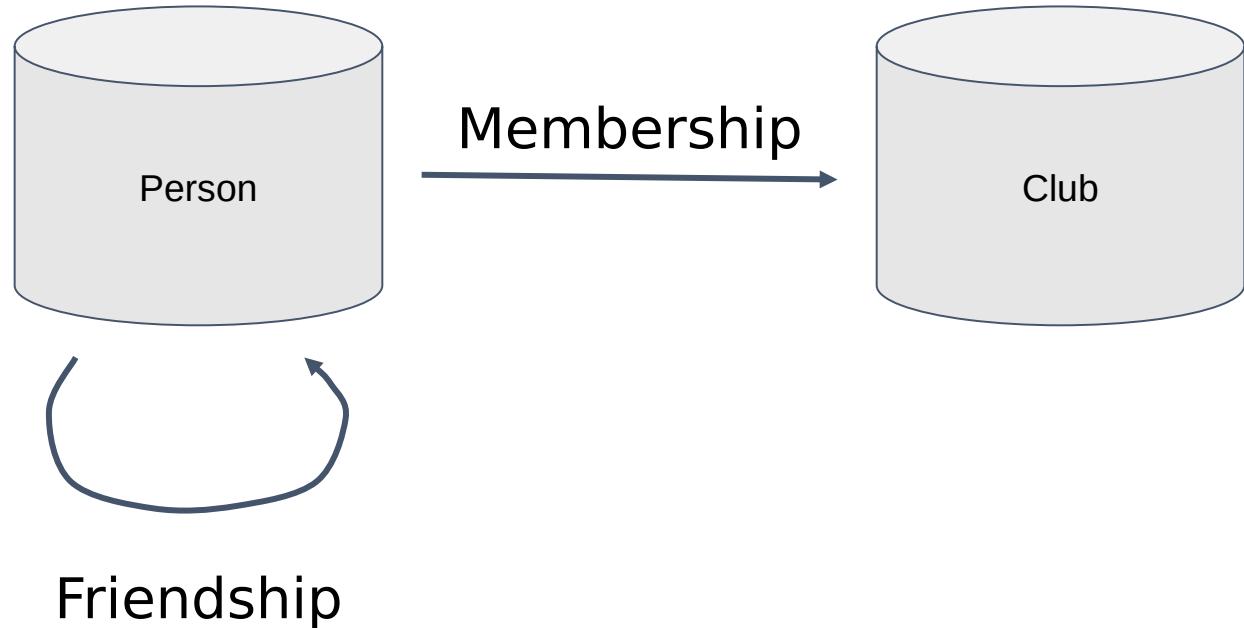
Graded lab:  
Social Network Database



+ Code + Text



```
| from sqlalchemy import create_engine, Column, Integer, String, ForeignKey, Table
| from sqlalchemy.ext.declarative import declarative_base
| from sqlalchemy.orm import relationship, sessionmaker
| import random
|
| engine = create_engine('sqlite:///social_network.db', echo=True)
| Base = declarative_base()
|
| # Define the friendship association table
| friendships = Table('friendships', Base.metadata,
|     Column('person_id', Integer, ForeignKey('people.id'), primary_key=True),
|     Column('friend_id', Integer, ForeignKey('people.id'), primary_key=True)
| )
|
| class Person(Base):
|     __tablename__ = 'people'
|     id = Column(Integer, primary_key=True)
|     name = Column(String)
|     age = Column(Integer)
|     gender = Column(String)
|     location = Column(String)
|
|     friends = relationship("Person",
|                           secondary=friendships,
|                           primaryjoin=(friendships.c.person_id == id),
|                           secondaryjoin=(friendships.c.friend_id == id),
|                           backref="friend_of")
|     clubs = relationship("Club", secondary="club_members")
```



# Graded lab task

1. Create code that, given a particular club, finds all members of that club
2. Create code that, given the name of a person, finds all the friends of that person
3. Create code that, given the name of a person, finds all the people that consider that person a friend

# Graded lab task

There's also a *major* design and logic flaw in the database that came as a result of using generated code.

- Find and fix that flaw
- HINT: Look for things that are repeated that should not be!



# Software Design Patterns

---

## Module introduction

# What are design patterns?

- Reusable solutions to common problems in software design.
- Outline standard ways to structure and organize code for specific scenarios.

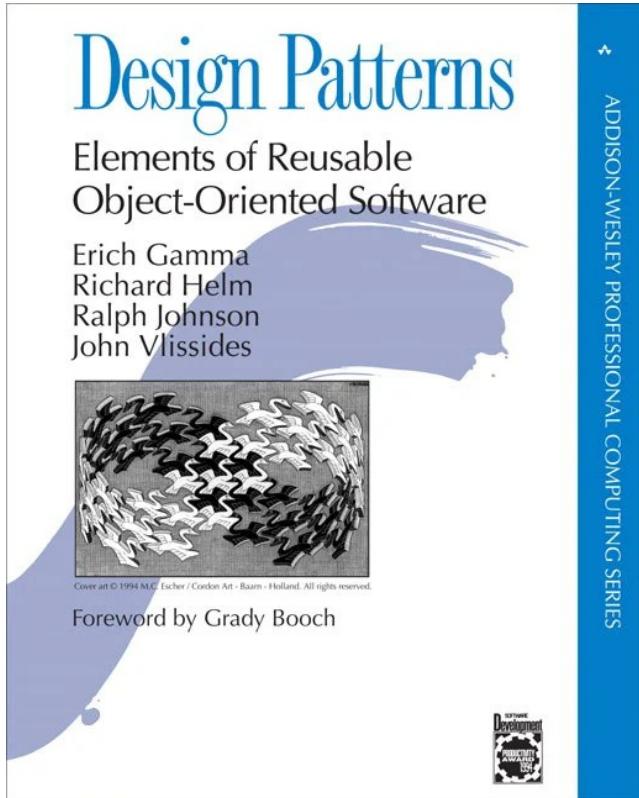


Image credit: Addison-Wesley

# Design Patterns: Elements of Reusable Object-Oriented Software

Erich Gamma, Richard Helm, Ralph Johnson,  
John Vlissides (1994)

The “Gang of Four”!

# Gang of Four design pattern groups

Creational

Structural

Behavioral



DeepLearning.AI

# Software Design Patterns

---

## Gang of Four patterns overview

# Exploring design patterns with an LLM

**Prompt:** Give me a high-level overview of the Gang of Four design patterns.

# Gang of Four design pattern groups

## Creational

Outline object creation mechanisms.

## Structural

Deal with the composition of classes or objects.

## Behavioral

Characterize the ways in which classes or objects interact and distribute responsibility.

# Gang of Four design pattern groups

## Creational

Outline object creation mechanisms.

## Structural

Deal with the composition of classes or objects.

## Behavioral

Characterize the ways in which classes or objects interact and distribute responsibility.

# Creational design patterns

- Singleton
- Builder
- Prototypes
- Factory Methods
- Abstract Factories

# Gang of Four design pattern groups

## Creational

Outline object creation mechanisms.

## Structural

Deal with the composition of classes or objects.

## Behavioral

Characterize the ways in which classes or objects interact and distribute responsibility.

# Structural design patterns

- Adapter
- Bridges
- Composites
- Decorators
- Facades
- Flyweights
- Proxies

# Gang of Four design pattern groups

## Creational

Outline object creation mechanisms.

## Structural

Deal with the composition of classes or objects.

## Behavioral

Characterize the ways in which classes or objects interact and distribute responsibility.

# Behavioral design patterns

- Iterator
- Strategy
- Template Method

# Prompting best practices

- **Be specific:** Provide detail and context about your problem
- **Assign a role:** Assign a role to tailor the output you receive
- **Request an expert opinion:** Assign an expert role and ask the LLM to evaluate the work you've already done to further refine it
- **Give feedback:** Iteratively prompt the LLM and provide feedback on the output you receive to get closer to your expected results



DeepLearning.AI

# Software Design Patterns

---

## Singletons

```
// Singleton class definition
class Singleton:

    // Static variable to hold the single instance of the class
    private static instance = null

    // Private constructor to prevent instantiation from outside
    private Singleton():
        // Initialization code (if any)
        print("Singleton instance created.")

    // Public static method to provide global access to the instance
    public static getInstance():
        // Check if the instance is null (not created yet)
        if instance == null:
            // Create the instance if it doesn't exist
            instance = new Singleton()

        // Return the single instance
        return instance
```

# Singleton Pseudocode

```
class Singleton:  
    _instance = None  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self
```

```
class Singleton:  
    _instance = None  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self
```

```
class Singleton:  
    _instance = None  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self
```

```
class Singleton:  
    _instance = None  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self
```

```
class Singleton:  
    _instance = None  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self
```

```
class Singleton:  
    _instance = None  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self
```

```
class Singleton:  
    _instance = None  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self
```

```
class Singleton:  
    _instance = None  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self  
            self.configuration = {}  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def set_config(self, key, value):  
        self.configuration[key] = value  
  
    def get_config(self, key):  
        return self.configuration.get(key, None)
```

```
class Singleton:  
    _instance = None  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self  
            self.configuration = {}  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def set_config(self, key, value):  
        self.configuration[key] = value  
  
    def get_config(self, key):  
        return self.configuration.get(key, None)
```

```
class Singleton:  
    _instance = None  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self  
            self.configuration = {}  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def set_config(self, key, value):  
        self.configuration[key] = value  
  
    def get_config(self, key):  
        return self.configuration.get(key, None)
```

```
class Singleton:  
    _instance = None  
  
    def __init__(self):  
        if Singleton._instance is not None:  
            raise Exception("This class is a singleton!")  
        else:  
            Singleton._instance = self  
            self.configuration = {}  
  
    @staticmethod  
    def getInstance():  
        if Singleton._instance is None:  
            Singleton._instance = Singleton()  
        return Singleton._instance  
  
    def set_config(self, key, value):  
        self.configuration[key] = value  
  
    def get_config(self, key):  
        return self.configuration.get(key, None)
```

```
def __init__(self):
    if Singleton._instance is not None:
        raise Exception("This class is a singleton!")
    else:
        Singleton._instance = self
        self.configuration = {}
```

```
@staticmethod
def getInstance():
    if Singleton._instance is None:
        Singleton._instance = Singleton()
    return Singleton._instance
```

```
def set_config(self, key, value):
    self.configuration[key] = value
```

```
def get_config(self, key):
    return self.configuration.get(key, None)
```

```
def display_config(self):
    for key, value in self.configuration.items():
        print(f'{key}: {value}')
```

```
# Setting configuration in one part of the application
singleton_instance = Singleton.getInstance()
singleton_instance.set_config('database', 'PostgreSQL')

# Accessing configuration in another part
another_instance = Singleton.getInstance()
another_instance.display_config()

# Both instances are the same
assert singleton_instance is another_instance

print("Instances are the same:", singleton_instance is another_instance)
```

```
# Setting configuration in one part of the application
singleton_instance = Singleton.getInstance()
singleton_instance.set_config('database', 'PostgreSQL')

# Accessing configuration in another part
another_instance = Singleton.getInstance()
another_instance.display_config()

# Both instances are the same
assert singleton_instance is another_instance

print("Instances are the same:", singleton_instance is another_instance)
```

```
# Setting configuration in one part of the application
singleton_instance = Singleton.getInstance()
singleton_instance.set_config('database', 'PostgreSQL')
```

```
# Accessing configuration in another part
another_instance = Singleton.getInstance()
another_instance.display_config()
```

```
# Both instances are the same
assert singleton_instance is another_instance

print("Instances are the same:", singleton_instance is another_instance)
```

```
# Setting configuration in one part of the application
singleton_instance = Singleton.getInstance()
singleton_instance.set_config('database', 'PostgreSQL')

# Accessing configuration in another part
another_instance = Singleton.getInstance()
another_instance.display_config()

# Both instances are the same
assert singleton_instance is another_instance

print("Instances are the same:", singleton_instance is another_instance)
```

```
# Setting configuration in one part of the application
singleton_instance = Singleton.getInstance()
singleton_instance.set_config('database', 'PostgreSQL')

# Accessing configuration in another part
another_instance = Singleton.getInstance()
another_instance.display_config()

# Both instances are the same
assert singleton_instance is another_instance

print("Instances are the same:", singleton_instance is another_instance)
```

```
# Setting configuration in one part of the application
singleton_instance = Singleton.getInstance()
singleton_instance.set_config('database', 'PostgreSQL')
```

```
# Accessing configuration in another part
another_instance = Singleton.getInstance()
another_instance.display_config()
```

```
# Both instances are the same
assert singleton_instance is another_instance

print("Instances are the same:", singleton_instance is another_instance)
```

```
database: PostgreSQL  
Instances are the same: True
```



DeepLearning.AI

# Software Design Patterns

---

Patterns advice from an  
LLM

# Building a financial services app

**Prompt:** You are an expert Python developer who builds readable code. Together we will work on an application that has a database to store information, code to retrieve data from the database, and analytics that will run on the retrieved data.

First, let's create the database, which has a table for companies. The table will have three columns, the first is an id, the second the ticker for the company, the third is the name of the company. Create this, and synthesize data for 10 companies, adding that to the database.

# Building a financial services app

**Prompt:**

You are an expert Python developer who builds readable code. Together we will work on an application that has a database to store information, code to retrieve data from the database, and analytics that will run on the retrieved data.

First, let's create the database, which has a table for companies. The table will have three columns, the first is an id, the second the ticker for the company, the third is the name of the company. Create this, and synthesize data for 10 companies, adding that to the database.

# Building a financial services app

**Prompt:** You are an expert Python developer who builds readable code. Together we will work on an application that has a database to store information, code to retrieve data from the database, and analytics that will run on the retrieved data.

First, let's create the database, which has a table for companies. The table will have three columns, the first is an id, the second the ticker for the company, the third is the name of the company. Create this, and synthesize data for 10 companies, adding that to the database.

# Generating synthetic data

**Follow up:** Now add another table called 'TimeSeries' that has four columns, an id for the row, the id for one of the companies, a value, and a date. Populate this with about 100 values per company, and have the dates be successive.

# Generating synthetic data

**Follow up:** Now add another table called 'TimeSeries' that has four columns, an id for the row, the id for one of the companies, a value, and a date.  
Populate this with about 100 values per company, and have the dates be successive.

# Loading and processing company data

**Follow up:** Now create code that when given a company ticker or an ID that it will extract the data for that company, and the time series data and load it into an company object. This object should have fields for high\_bollinger which is the same data type as the time series, low\_bollinger which is the same, moving\_average which is the same, and a grade field which is a string.

# Requesting design pattern advice

**Prompt:** You are an expert in software design patterns, particularly those from the Gang of Four, designed to make coding and maintenance more efficient. Please analyze the following code and suggest some changes that I could make based on good software engineering practice with these design patterns.

# Requesting design pattern advice

**Prompt:**

You are an expert in software design patterns, particularly those from the Gang of Four, designed to make coding and maintenance more efficient. Please analyze the following code and suggest some changes that I could make based on good software engineering practice with these design patterns.



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.

# Requesting design pattern advice iteratively

**Follow up:** Instead of making all the changes at once, please do them one at a time, going in order from Singleton to Factory to template Method to Strategy, and explain in detail why you made the changes and what impact they may have.



DeepLearning.AI

# Software Design Patterns

---

## Singleton for database connection management



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.

```
# Singleton Pattern for Database Connection
class DatabaseConnection:
    _instance = None

    def __new__(cls, db_path):
        if cls._instance is None:
            cls._instance = super(DatabaseConnection, cls).__new__(cls)
            cls._instance.connection = sqlite3.connect(db_path)
        return cls._instance

    def get_connection(self):
        return self.connection
```

```
# Singleton Pattern for Database Connection
class DatabaseConnection:
    _instance = None

    def __new__(cls, db_path):
        if cls._instance is None:
            cls._instance = super(DatabaseConnection, cls).__new__(cls)
            cls._instance.connection = sqlite3.connect(db_path)
        return cls._instance

    def get_connection(self):
        return self.connection
```

```
# Singleton Pattern for Database Connection
class DatabaseConnection:
    _instance = None

    def __new__(cls, db_path):
        if cls._instance is None:
            cls._instance = super(DatabaseConnection, cls).__new__(cls)
            cls._instance.connection = sqlite3.connect(db_path)
        return cls._instance

    def get_connection(self):
        return self.connection
```

```
# Singleton Pattern for Database Connection
class DatabaseConnection:
    _instance = None

    def __new__(cls, db_path):
        if cls._instance is None:
            cls._instance = super(DatabaseConnection, cls).__new__(cls)
            cls._instance.connection = sqlite3.connect(db_path)
        return cls._instance

    def get_connection(self):
        return self.connection
```

```
# Singleton Pattern for Database Connection
class DatabaseConnection:
    _instance = None

    def __new__(cls, db_path):
        if cls._instance is None:
            cls._instance = super(DatabaseConnection, cls).__new__(cls)
            cls._instance.connection = sqlite3.connect(db_path)
        return cls._instance

    @staticmethod
    def get_connection():
        if DatabaseConnection._instance is None:
            raise Exception("DatabaseConnection has not been initialized.
                            Call DatabaseConnection(db_path) first.")

        return DatabaseConnection._instance.connection
```

# Requesting Design Pattern Advice Iteratively

**Prompt:** What would the problem be if `get_connection()` wasn't a static method?



If `get_connection()` were not a static method, it would imply that an instance of `DatabaseConnection` is required to call it. This could lead to several issues:

1. **Multiple Instances:** Without a static method, you might inadvertently create multiple instances of the `DatabaseConnection` class to access the connection, which defeats the purpose of the Singleton pattern. Each instance could potentially try to manage its own connection, leading to multiple open connections to the database, consuming more resources, and possibly causing conflicts or inconsistencies.
2. **Initialization Requirement:** Each time you need to get a connection, you would need to ensure that you have an instance of the `DatabaseConnection` class. This adds unnecessary complexity and can lead to code that is harder to manage and maintain.
3. **Encapsulation Violation:** The Singleton pattern is intended to control the instantiation of a class and ensure there is only one instance. If `get_connection()` is not static, the control over instance creation is weakened, as the access pattern does not enforce the Singleton behavior strictly.

```
db1 = DatabaseConnection('company_database.db')
conn1 = db1.get_connection()

db2 = DatabaseConnection('company_database.db')
conn2 = db2.get_connection()
```

```
db1 = DatabaseConnection('company_database.db')  
conn1 = db1.get_connection()
```

```
db2 = DatabaseConnection('company_database.db')  
conn2 = db2.get_connection()
```

```
db1 = DatabaseConnection('company_database.db')  
conn1 = db1.get_connection()
```

```
db2 = DatabaseConnection('company_database.db')  
conn2 = db2.get_connection()
```

```
DatabaseConnection('company_database.db')  
conn = DatabaseConnection.get_connection()
```

```
DatabaseConnection('company_database.db')
```

```
conn = DatabaseConnection.get_connection()
```

```
DatabaseConnection('company_database.db')  
conn = DatabaseConnection.get_connection()  
conn2 = DatabaseConnection.get_connection()  
conn3 = DatabaseConnection.get_connection()  
conn4 = DatabaseConnection.get_connection()
```

```
// Singleton class definition
class Singleton:

    // Static variable to hold the single instance of the class
    private static instance = null

    // Private constructor to prevent instantiation from outside
    private Singleton():
        // Initialization code (if any)
        print("Singleton instance created.")

    // Public static method to provide global access to the instance
    public static getInstance():
        // Check if the instance is null (not created yet)
        if instance == null:
            // Create the instance if it doesn't exist
            instance = new Singleton()

        // Return the single instance
        return instance
```

# Singleton Pseudocode

```
# Singleton Pattern for Database Connection
class DatabaseConnection:
    _instance = None

    def __new__(cls, db_path):
        if cls._instance is None:
            cls._instance = super(DatabaseConnection, cls).__new__(cls)
            cls._instance.connection = sqlite3.connect(db_path)
        return cls._instance

    def get_connection(self):
        return self.connection
```



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.

# Requesting design pattern advice iteratively

**Prompt:** Enhance the following code to use the gang-of-four patterns. Strictly follow the common conventions for any patterns you choose.

# Requesting design pattern advice iteratively

**Follow up:** Enhance the following code to use the Singleton gang-of-four pattern. Strictly follow the common conventions for the pattern.

Start by explaining the conventions for the Singleton pattern and then describe how the code modifications you made strictly follow the conventions.



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.



DeepLearning.AI

# Software Design Patterns

---

## Factory Patterns



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.

# Requesting information on the Factory Pattern

**Prompt:** Let's go back to the factory pattern, can you explain it in much more detail? It looks like this code may not benefit from it, but can you speculate about extensions to the code that might?

Note: This prompt continued a previous chat

# Implementing the Factory Pattern

**Prompt:** Enhance the following code to use the Factory gang-of-four pattern. Strictly follow the common conventions for the pattern. Start by explaining the conventions for the Factory pattern and why it makes sense to use it here. Then describe how the code modifications you made strictly follow the conventions of the pattern.

YOUR CODE HERE...



Absolutely, let's dive deeper into the **Factory Pattern** and how it could be beneficial in different scenarios.

## Factory Pattern: Detailed Explanation

The Factory Pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. The main goal is to decouple the creation of objects from their usage.

In the context of our example, the `CompanyFactory` is responsible for creating `Company` objects. Here's a deeper look at the pattern and potential benefits:

### Benefits of Factory Pattern

1. **Decoupling Object Creation:** The pattern decouples the creation of objects from their usage, making the code more modular and easier to maintain.
2. **Simplifies Code:** Object creation logic is centralized, reducing redundancy and making it easier to manage.
3. **Promotes Code Reuse:** Common creation logic can be reused, and new types of objects can be introduced with minimal changes to the existing codebase.
4. **Encapsulation:** Object creation logic is encapsulated in one place, adhering to the single responsibility principle.



Absolutely, let's dive deeper into the **Factory Pattern** and how it could be beneficial in different scenarios.

## Factory Pattern: Detailed Explanation

The Factory Pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. The main goal is to decouple the creation of objects from their usage.

In the context of our example, the `CompanyFactory` is responsible for creating `Company` objects.

Here's a deeper look at the pattern and potential benefits:

### Benefits of Factory Pattern

1. **Decoupling Object Creation:** The pattern decouples the creation of objects from their usage, making the code more modular and easier to maintain.
2. **Simplifies Code:** Object creation logic is centralized, reducing redundancy and making it easier to manage.
3. **Promotes Code Reuse:** Common creation logic can be reused, and new types of objects can be introduced with minimal changes to the existing codebase.
4. **Encapsulation:** Object creation logic is encapsulated in one place, adhering to the single responsibility principle.



Absolutely, let's dive deeper into the **Factory Pattern** and how it could be beneficial in different scenarios.

## Factory Pattern: Detailed Explanation

The Factory Pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. The main goal is to decouple the creation of objects from their usage.

In the context of our example, the `CompanyFactory` is responsible for creating `Company` objects. Here's a deeper look at the pattern and potential benefits:

### Benefits of Factory Pattern

1. **Decoupling Object Creation:** The pattern decouples the creation of objects from their usage, making the code more modular and easier to maintain.
2. **Simplifies Code:** Object creation logic is centralized, reducing redundancy and making it easier to manage.
3. **Promotes Code Reuse:** Common creation logic can be reused, and new types of objects can be introduced with minimal changes to the existing codebase.
4. **Encapsulation:** Object creation logic is encapsulated in one place, adhering to the single responsibility principle.

```
class Company:  
    def __init__(self, company_id, ticker, name):  
        self.company_id = company_id  
        self.ticker = ticker  
        self.name = name  
        self.time_series = None  
        self.high_bollinger = None  
        self.low_bollinger = None  
        self.moving_average = None  
        self.grade = None
```

```
# Synthesize data for 10 companies
```

```
companies = [
```

```
(1, 'AAPL', 'Apple Inc.'),  
(2, 'GOOGL', 'Alphabet Inc.'),  
(3, 'MSFT', 'Microsoft Corporation'),  
(4, 'AMZN', 'Amazon.com Inc.'),  
(5, 'TSLA', 'Tesla Inc.'),  
(6, 'FB', 'Meta Platforms Inc.'),  
(7, 'NVDA', 'NVIDIA Corporation'),  
(8, 'NFLX', 'Netflix Inc.'),  
(9, 'ADBE', 'Adobe Inc.'),  
(10, 'ORCL', 'Oracle Corporation')
```

```
]
```

```
# Synthesize data for 10 companies
```

```
companies = [
```

```
(1, 'AAPL', 'Apple Inc.'),  
(2, 'GOOGL', 'Alphabet Inc.'),  
(3, 'MSFT', 'Microsoft Corporation'),  
(4, 'AMZN', 'Amazon.com Inc.'),  
(5, 'TSLA', 'Tesla Inc.'),  
(6, 'FB', 'Meta Platforms Inc.'),  
(7, 'NVDA', 'NVIDIA Corporation'),  
(8, 'NFLX', 'Netflix Inc.'),  
(9, 'ADBE', 'Adobe Inc.'),  
(10, 'ORCL', 'Oracle Corporation')
```

```
]
```

```
class Company:  
    def __init__(self, company_id, ticker, name):  
        self.company_id = company_id  
        self.ticker = ticker  
        self.name = name  
        self.time_series = None  
        self.high_bollinger = None  
        self.low_bollinger = None  
        self.moving_average = None  
        self.grade = None  
  
    def load_time_series(self, conn):  
        # Code to load time Series  
  
    def calculate_bollinger_bands(self):  
        # Code to calculate bollinger_bands  
  
    def assign_grade(self):  
        # Code to assign a grade  
  
    def display(self):  
        # Code to display the company details
```

```
class Company:  
    def __init__(self, company_id, ticker, name):  
        self.company_id = company_id  
        self.ticker = ticker  
        self.name = name  
        self.time_series = None  
        self.high_bollinger = None  
        self.low_bollinger = None  
        self.moving_average = None  
        self.grade = None  
  
    def load_time_series(self, conn):  
        # Code to load time Series  
  
    def calculate_bollinger_bands(self):  
        # Code to calculate bollinger_bands  
  
    def assign_grade(self):  
        # Code to assign a grade  
  
    def display(self):  
        # Code to display the company details
```

```
def get_company_by_ticker(ticker, conn):  
    cursor = conn.cursor()  
    query = 'SELECT id, ticker, name FROM companies WHERE ticker = ?'  
    cursor.execute(query, (ticker,))  
    row = cursor.fetchone()  
    if row:  
        return Company(row[0], row[1], row[2])
```



Absolutely, let's dive deeper into the **Factory Pattern** and how it could be beneficial in different scenarios.

## Factory Pattern: Detailed Explanation

The Factory Pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. The main goal is to decouple the creation of objects from their usage.

In the context of our example, the `CompanyFactory` is responsible for creating `Company` objects.

Here's a deeper look at the pattern and potential benefits:

### Benefits of Factory Pattern

1. **Decoupling Object Creation:** The pattern decouples the creation of objects from their usage, making the code more modular and easier to maintain.
2. **Simplifies Code:** Object creation logic is centralized, reducing redundancy and making it easier to manage.
3. **Promotes Code Reuse:** Common creation logic can be reused, and new types of objects can be introduced with minimal changes to the existing codebase.
4. **Encapsulation:** Object creation logic is encapsulated in one place, adhering to the single responsibility principle.

# Factory Pattern elements

- A **factory class** handles overall production of objects
- **Concrete classes** provide specific instructions to make variations of those objects



# Factory Pattern elements

- A **factory class** handles overall production of objects
- **Concrete classes** provide specific instructions to make variations of those objects

# Multiple company types using the Factory Pattern

**Prompt:** Go back to the code that adds data to the database and synthesize some foreign companies (which will have a unique ID, but the ticker is always 'ZZZZ' ) and the data for them, both in the companies table and in the time series table.

Then fully explain the Factory pattern by having multiple company types -- a Domestic Company that is denoted by its ticker, and a Foreign Company that is denoted by its ID.

Create code that shows how

# Multiple company types using the Factory Pattern

**Prompt:**

Go back to the code that adds data to the database and synthesize some foreign companies (which will have a unique ID, but the ticker is always 'ZZZZ' ) and the data for them, both in the companies table and in the time series table.

Then fully explain the Factory pattern by having multiple company types -- a Domestic Company that is denoted by its ticker, and a Foreign Company that is denoted by its ID.

Create code that shows how

```
# Insert synthesized foreign companies
foreign_companies = [
    (1001, 'ZZZZ', 'Foreign Company A'),
    (1002, 'ZZZZ', 'Foreign Company B')
]
cursor.executemany('INSERT INTO companies (id, ticker, name) VALUES (?, ?, ?)',
    foreign_companies)
```

# Multiple company types using the Factory Pattern

**Prompt:** Go back to the code that adds data to the database and synthesize some foreign companies (which will have a unique ID, but the ticker is always 'ZZZZ' ) and the data for them, both in the companies table and in the time series table.

Then fully explain the Factory pattern by having multiple company types -- a Domestic Company that is denoted by its ticker, and a Foreign Company that is denoted by its ID.

Create code that shows how

```
class CompanyFactory:  
    @staticmethod  
    def get_company(identifier, conn):  
        cursor = conn.cursor()  
  
        if isinstance(identifier, str):  
            query = 'SELECT id, ticker, name FROM companies WHERE ticker = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                return DomesticCompany(row[0], row[1], row[2])  
        else:  
            query = 'SELECT id, ticker, name FROM companies WHERE id = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                # If ticker is equal to ZZZZ, it's a foreign company  
                if row[1] == 'ZZZZ':  
                    return ForeignCompany(row[0], row[1], row[2])  
                else:  
                    return DomesticCompany(row[0], row[1], row[2])  
        return None
```

```
class CompanyFactory:  
    @staticmethod  
    def get_company(identifier, conn):  
        cursor = conn.cursor()  
  
        if isinstance(identifier, str):  
            query = 'SELECT id, ticker, name FROM companies WHERE ticker = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                return DomesticCompany(row[0], row[1], row[2])  
            else:  
                query = 'SELECT id, ticker, name FROM companies WHERE id = ?'  
                cursor.execute(query, (identifier,))  
                row = cursor.fetchone()  
                if row:  
                    # If ticker is equal to ZZZZ, it's a foreign company  
                    if row[1] == 'ZZZZ':  
                        return ForeignCompany(row[0], row[1], row[2])  
                    else:  
                        return DomesticCompany(row[0], row[1], row[2])  
        return None
```

```
class DomesticCompany(Company):
    def __init__(self, company_id, ticker, name):
        super().__init__(company_id, ticker, name)
        self.company_type = 'Domestic'
```

```
class ForeignCompany(Company):
    def __init__(self, company_id, ticker, name):
        super().__init__(company_id, ticker, name)
        self.company_type = 'Foreign'
```

```
class CompanyFactory:  
    @staticmethod  
    def get_company(identifier, conn):  
        cursor = conn.cursor()  
  
        if isinstance(identifier, str):  
            query = 'SELECT id, ticker, name FROM companies WHERE ticker = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                return DomesticCompany(row[0], row[1], row[2])  
            else:  
                query = 'SELECT id, ticker, name FROM companies WHERE id = ?'  
                cursor.execute(query, (identifier,))  
                row = cursor.fetchone()  
                if row:  
                    # If ticker is equal to ZZZZ, it's a foreign company  
                    if row[1] == 'ZZZZ':  
                        return ForeignCompany(row[0], row[1], row[2])  
                    else:  
                        return DomesticCompany(row[0], row[1], row[2])  
        return None
```

```
class CompanyFactory:  
    @staticmethod  
    def get_company(identifier, conn):  
        cursor = conn.cursor()  
  
        if isinstance(identifier, str):  
            query = 'SELECT id, ticker, name FROM companies WHERE ticker = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                return DomesticCompany(row[0], row[1], row[2])  
            else:  
                query = 'SELECT id, ticker, name FROM companies WHERE id = ?'  
                cursor.execute(query, (identifier,))  
                row = cursor.fetchone()  
                if row:  
                    # If ticker is equal to ZZZZ, it's a foreign company  
                    if row[1] == 'ZZZZ':  
                        return ForeignCompany(row[0], row[1], row[2])  
                    else:  
                        return DomesticCompany(row[0], row[1], row[2])  
        return None
```

```
class CompanyFactory:  
    @staticmethod  
    def get_company(identifier, conn):  
        cursor = conn.cursor()  
  
        if isinstance(identifier, str):  
            query = 'SELECT id, ticker, name FROM companies WHERE ticker = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                return DomesticCompany(row[0], row[1], row[2])  
        else:  
            query = 'SELECT id, ticker, name FROM companies WHERE id = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                # If ticker is equal to ZZZZ, it's a foreign company  
                if row[1] == 'ZZZZ':  
                    return ForeignCompany(row[0], row[1], row[2])  
                else:  
                    return DomesticCompany(row[0], row[1], row[2])  
        return None
```

```
class CompanyFactory:  
    @staticmethod  
    def get_company(identifier, conn):  
        cursor = conn.cursor()  
  
        if isinstance(identifier, str):  
            query = 'SELECT id, ticker, name FROM companies WHERE ticker = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                return DomesticCompany(row[0], row[1], row[2])  
        else:  
            query = 'SELECT id, ticker, name FROM companies WHERE id = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                # If ticker is equal to ZZZZ, it's a foreign company  
                if row[1] == 'ZZZZ':  
                    return ForeignCompany(row[0], row[1], row[2])  
                else:  
                    return DomesticCompany(row[0], row[1], row[2])  
        return None
```

```
class CompanyFactory:  
    @staticmethod  
    def get_company(identifier, conn):  
        cursor = conn.cursor()  
  
        if isinstance(identifier, str):  
            query = 'SELECT id, ticker, name FROM companies WHERE ticker = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                return DomesticCompany(row[0], row[1], row[2])  
        else:  
            query = 'SELECT id, ticker, name FROM companies WHERE id = ?'  
            cursor.execute(query, (identifier,))  
            row = cursor.fetchone()  
            if row:  
                # If ticker is equal to ZZZZ, it's a foreign company  
                if row[1] == 'ZZZZ':  
                    return ForeignCompany(row[0], row[1], row[2])  
                else:  
                    return DomesticCompany(row[0], row[1], row[2])  
        return None
```

```
domestic_company = CompanyFactory.get_company('AAPL', conn)
```

```
foreign_company = CompanyFactory.get_company(1001, conn)
```

```
class Company:  
    def __init__(self, company_id, ticker, name):  
        self.company_id = company_id  
        self.ticker = ticker  
        self.name = name  
        self.time_series = None  
        self.high_bollinger = None  
        self.low_bollinger = None  
        self.moving_average = None  
        self.grade = None  
  
    def load_time_series(self, conn):  
        # Code to load time Series  
  
    def calculate_bollinger_bands(self):  
        # Code to calculate bollinger_bands  
  
    def assign_grade(self):  
        # Code to assign a grade  
  
    def display(self):  
        # Code to display the company details
```

```
class Company:  
    def __init__(self, company_id, ticker, name):  
        self.company_id = company_id  
        self.ticker = ticker  
        self.name = name  
        self.time_series = None  
        self.high_bollinger = None  
        self.low_bollinger = None  
        self.moving_average = None  
        self.grade = None  
  
    def load_time_series(self, conn):  
        # Code to load time Series  
  
    def calculate_bollinger_bands(self):  
        # Code to calculate bollinger_bands  
  
    def assign_grade(self):  
        # Code to assign a grade  
  
    def display(self):  
        # Code to display the company details
```

# Self-paced exercise

- Implement a third kind of company - a **crypto currency**
  - Traded 24 hours, so has no daily “closing” price
- Can you use the current TimeSeries database, or do you need a new one?
- Can you modify the code to calculate Bollinger bands, moving average etc. for these companies?



DeepLearning.AI

# Software Design Patterns

---

## Template Method Pattern



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.



To enhance this code using software design patterns from the Gang of Four, we can introduce a few key patterns to make the design more modular, maintainable, and extendable:

1. **Singleton Pattern**: Ensure only one instance of the database connection exists.
2. **Factory Pattern**: Create a factory for creating `Company` objects.
3. **Template Method Pattern**: Use a template method for operations that follow a specific sequence.
4. **Strategy Pattern**: For the different grading strategies.

# Requesting Information on the Template Pattern

**Prompt:** Great - the third pattern you mentioned was the Template Method pattern. Can you demonstrate where, how, and why you would use it in this code? What advantages would it bring?



Sure! The **Template Method** pattern is a behavioral design pattern that defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of an algorithm without changing its structure.

## Applying the Template Method Pattern

In the context of our `Company` class, we can use the Template Method pattern to define the process of loading and processing time series data, which might have some common steps and some steps that can vary for different types of companies.

### Steps to Implement the Template Method Pattern

1. **Define the Template Method:** In the `Company` base class, we'll define a method that outlines the steps for processing time series data.
2. **Implement Common Steps:** Implement the common steps in the base class.
3. **Defer Variable Steps to Subclasses:** Allow subclasses to implement specific steps if needed.



Sure! The **Template Method** pattern is a behavioral design pattern that defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of an algorithm without changing its structure.

## Applying the Template Method Pattern

In the context of our `Company` class, we can use the Template Method pattern to define the process of loading and processing time series data, which might have some common steps and some steps that can vary for different types of companies.

### Steps to Implement the Template Method Pattern

1. **Define the Template Method:** In the `Company` base class, we'll define a method that outlines the steps for processing time series data.
2. **Implement Common Steps:** Implement the common steps in the base class.
3. **Defer Variable Steps to Subclasses:** Allow subclasses to implement specific steps if needed.



Sure! The **Template Method** pattern is a behavioral design pattern that defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of an algorithm without changing its structure.

## Applying the Template Method Pattern

In the context of our `Company` class, we can use the Template Method pattern to define the process of loading and processing time series data, which might have some common steps and some steps that can vary for different types of companies.

### Steps to Implement the Template Method Pattern

1. **Define the Template Method:** In the `Company` base class, we'll define a method that outlines the steps for processing time series data.
2. **Implement Common Steps:** Implement the common steps in the base class.
3. **Defer Variable Steps to Subclasses:** Allow subclasses to implement specific steps if needed.



Sure! The **Template Method** pattern is a behavioral design pattern that defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of an algorithm without changing its structure.

## Applying the Template Method Pattern

In the context of our `Company` class, we can use the Template Method pattern to define the process of loading and processing time series data, which might have some common steps and some steps that can vary for different types of companies.

### Steps to Implement the Template Method Pattern

1. **Define the Template Method:** In the `Company` base class, we'll define a method that outlines the steps for processing time series data.
2. **Implement Common Steps:** Implement the common steps in the base class.
3. **Defer Variable Steps to Subclasses:** Allow subclasses to implement specific steps if needed.

## # Template Method

```
def process_time_series(self, conn):  
    self.load_time_series(conn)  
    self.preprocess_data()  
    self.calculate_bollinger_bands()  
    self.postprocess_data()
```

## # Template Method

```
def process_time_series(self, conn):
    self.load_time_series(conn)
    self.preprocess_data()
    self.calculate_hollinger_bands()
    self.postprocess_data()
```

```
domestic_company.load_time_series(conn)
domestic_company.calculate_bollinger_bands()
grading_strategy = BollingerBandGradingStrategy()
domestic_company.assign_grade(grading_strategy)
domestic_company.display()
```

```
domestic_company.load_time_series(conn)
domestic_company.calculate_bollinger_bands()
grading_strategy = BollingerBandGradingStrategy()
domestic_company.assign_grade(grading_strategy)
domestic_company.display()
```

```
domestic_company.load_time_series(conn)
domestic_company.calculate_bollinger_bands()
grading_strategy = BollingerBandGradingStrategy()
domestic_company.assign_grade(grading_strategy)
domestic_company.display()
```



```
domestic_company.load_time_series(conn)
domestic_company.calculate_bollinger_bands()
grading_strategy = BollingerBandGradingStrategy()
domestic_company.assign_grade(grading_strategy)
domestic_company.display()
```

## # Template Method

```
def process_time_series(self, conn):  
    self.load_time_series(conn)  
    self.preprocess_data()  
    self.calculate_bollinger_bands()  
    self.postprocess_data()
```

## # Template Method

```
def process_time_series(self, conn):  
    self.load_time_series(conn)  
    self._  
    self.preprocess_data()  
    self._  
    self.calculate_hollinger_bands()  
    self._  
    self.postprocess_data()
```

```
# Hook method for preprocessing data, can be overridden by  
subclasses
```

```
def preprocess_data(self):  
    pass
```

```
# Hook method for postprocessing data, can be overridden by  
subclasses
```

```
def postprocess_data(self):  
    pass
```

```
// Abstract methods to be implemented by subclasses  
protected abstract void preprocessData();  
protected abstract void postprocessData();
```

```
class ForeignCompany(Company):  
    def __init__(self, company_id, ticker, name):  
        super().__init__(company_id, ticker, name)  
        self.company_type = 'Foreign'
```

```
def postprocess_data(self):  
    # Example postprocessing step specific to ForeignCompany  
    print(f'Postprocessing data for ForeignCompany: {self.name}')
```

```
class ForeignCompany(Company):  
    def __init__(self, company_id, ticker, name):  
        super().__init__(company_id, ticker, name)  
        self.company_type = 'Foreign'  
  
    def postprocess_data(self):  
        # Example postprocessing step specific to ForeignCompany  
        print(f'Postprocessing data for ForeignCompany: {self.name}')
```

```
class ForeignCompany(Company):  
    def __init__(self, company_id, ticker, name):  
        super().__init__(company_id, ticker, name)  
        self.company_type = 'Foreign'  
  
    def postprocess_data(self):  
        # Example postprocessing step specific to ForeignCompany  
        print(f'Postprocessing data for ForeignCompany: {self.name}')
```

# Exploring more advanced functions

**Prompt:** The example of how to use the postprocess\_data method is very simple right now. What are some interesting examples of ways I could actually make use of the flexibility provided by that postprocess\_data method?

## 1. Currency Conversion

Foreign companies might report their financial data in different currencies. You could convert the time series data to a common currency for consistency.

```
python

class ForeignCompany(Company):
    def __init__(self, company_id, ticker, name, currency):
        super().__init__(company_id, ticker, name)
        self.company_type = 'Foreign'
        self.currency = currency

    def postprocess_data(self):
        # Example: Convert time series data to a common currency
        conversion_rate = self.get_conversion_rate(self.currency, 'USD')
        self.time_series['value'] = self.time_series['value'] * conversion_rate
        print(f'Converted time series data to USD for ForeignCompany: {self.name}')

    def get_conversion_rate(self, from_currency, to_currency):
        # Placeholder for actual conversion rate retrieval logic
        conversion_rates = {'EUR': 1.2, 'JPY': 0.009, 'GBP': 1.3} # Example rates
        return conversion_rates.get(from_currency, 1)
```

## 1. Currency Conversion

Foreign companies might report their financial data in different currencies. You could convert the time series data to a common currency for consistency.

```
python

class ForeignCompany(Company):
    def __init__(self, company_id, ticker, name, currency):
        super().__init__(company_id, ticker, name)
        self.company_type = 'Foreign'
        self.currency = currency

    def postprocess_data(self):
        # Example: Convert time series data to a common currency
        conversion_rate = self.get_conversion_rate(self.currency, 'USD')
        self.time_series['value'] = self.time_series['value'] * conversion_rate
        print(f'Converted time series data to USD for ForeignCompany: {self.name}')

    def get_conversion_rate(self, from_currency, to_currency):
        # Placeholder for actual conversion rate retrieval logic
        conversion_rates = {'EUR': 1.2, 'JPY': 0.009, 'GBP': 1.3} # Example rates
        return conversion_rates.get(from_currency, 1)
```

## 2. Time Zone Adjustment

Foreign companies might operate in different time zones. You could adjust the timestamps of the time series data to a common time zone.

```
python

class ForeignCompany(Company):
    def __init__(self, company_id, ticker, name, time_zone):
        super().__init__(company_id, ticker, name)
        self.company_type = 'Foreign'
        self.time_zone = time_zone

    def postprocess_data(self):
        # Example: Adjust time series data to a common time zone
        self.time_series['date'] = self.time_series['date'].dt.tz_localize(self.ti
            print(f'Adjusted time series data to UTC for ForeignCompany: {self.name}')
```



DeepLearning.AI

# Software Design Patterns

---

## Strategy Pattern

# Learning more about the Strategy Pattern

**Prompt:** Ok, so next let's explore the strategy pattern. What is it, how does it work, why would you use it, and demonstrate and clearly explain the code for it.

Also, please give me some scenarios where additions to this code might be made easier with this pattern?

```
from abc import ABC, abstractmethod
```

```
class GradingStrategy(ABC):
```

```
    @abstractmethod
```

```
    def assign_grade(self, company):
```

```
        pass
```

```
from abc import ABC, abstractmethod
```

```
class GradingStrategy(ABC):
```

```
    @abstractmethod
```

```
    def assign_grade(self, company):
```

```
        pass
```

```
from abc import ABC, abstractmethod
```

```
class GradingStrategy(ABC):  
    @abstractmethod  
    def assign_grade(self, company):  
        pass
```

```
from abc import ABC, abstractmethod

class GradingStrategy(ABC):
    @abstractmethod
    def assign_grade(self, company):
        pass
```

```
from abc import ABC, abstractmethod
```

```
class GradingStrategy(ABC):
```

```
    @abstractmethod
```

```
    def assign_grade(self, company):
```

```
        pass
```

# Strategy Pattern in Python

## Components

1. **Strategy Interface:** An interface common to all supported algorithms.
2. **Concrete Strategies:** Classes that implement the Strategy interface with specific algorithms.
3. **Context:** The `Company` class that uses a Strategy to assign a grade.

## Implementation

1. **Strategy Interface:** Define an interface for the grading strategies.

```
python
from abc import ABC, abstractmethod

class GradingStrategy(ABC):
    @abstractmethod
    def assign_grade(self, company):
        pass
```

2. **Concrete Strategies:** Implement different grading algorithms.

```
class BollingerBandGradingStrategy(GradingStrategy):  
    def assign_grade(self, company):  
        latest_value = company.time_series['value'].iloc[-1]  
        if latest_value > company.high_bollinger.iloc[-1]:  
            company.grade = 'A'  
        elif latest_value < company.low_bollinger.iloc[-1]:  
            company.grade = 'C'  
        else:  
            company.grade = 'B'
```

```
class BollingerBandGradingStrategy(GradingStrategy):
```

```
    def assign_grade(self, company):
```

```
        latest_value = company.time_series['value'].iloc[-1]
```

```
        if latest_value > company.high_bollinger.iloc[-1]:
```

```
            company.grade = 'A'
```

```
        elif latest_value < company.low_bollinger.iloc[-1]:
```

```
            company.grade = 'C'
```

```
        else:
```

```
            company.grade = 'B'
```

```
class BollingerBandGradingStrategy(GradingStrategy):
```

```
    def assign_grade(self, company):  
        latest_value = company.time_series['value'].iloc[-1]  
        if latest_value > company.high_bollinger.iloc[-1]:  
            company.grade = 'A'  
        elif latest_value < company.low_bollinger.iloc[-1]:  
            company.grade = 'C'  
        else:  
            company.grade = 'B'
```

```
class ThresholdGradingStrategy(GradingStrategy):
```

```
    def assign_grade(self, company):  
        latest_value = company.time_series['value'].iloc[-1]  
        if latest_value > 200:  
            company.grade = 'A'  
        elif latest_value < 100:  
            company.grade = 'C'  
        else:  
            company.grade = 'B'
```

```
class ThresholdGradingStrategy(GradingStrategy):  
    def assign_grade(self, company):  
        latest_value = company.time_series['value'].iloc[-1]  
        if latest_value > 200:  
            company.grade = 'A'  
        elif latest_value < 100:  
            company.grade = 'C'  
        else:  
            company.grade = 'B'
```

```
# Get domestic company by ticker
try:
    domestic_company = CompanyFactory.get_company('AAPL', conn)
    if domestic_company:
        domestic_company.process_time_series(conn)
        domestic_company.assign_grade(BollingerBandGradingStrategy())
        domestic_company.display()
    else:
        print("Domestic company not found")
except Exception as e:
    print(f"Error processing domestic company: {e}")
```

```
# Get domestic company by ticker
try:
    domestic_company = CompanyFactory.get_company('AAPL', conn)
    if domestic_company:
        domestic_company.process_time_series(conn)
        domestic_company.assign_grade(BollingerBandGradingStrategy())
        domestic_company.display()
    else:
        print("Domestic company not found")
except Exception as e:
    print(f"Error processing domestic company: {e}")
```

```
# Get foreign company by ID
```

```
try:
```

```
    foreign_company = CompanyFactory.get_company(1001, conn)
```

```
    if foreign_company:
```

```
        foreign_company.process_time_series(conn)
```

```
        foreign_company.assign_grade(ThresholdGradingStrategy())
```

```
        foreign_company.display()
```

```
    else:
```

```
        print("Foreign company not found")
```

```
except Exception as e:
```

```
    print(f"Error processing foreign company: {e}")
```

```
# Get foreign company by ID
try:
    foreign_company = CompanyFactory.get_company(1001, conn)
    if foreign_company:
        foreign_company.process_time_series(conn)
        foreign_company.assign_grade(ThresholdGradingStrategy())
        foreign_company.display()
    else:
        print("Foreign company not found")
except Exception as e:
    print(f"Error processing foreign company: {e}")
```



# Software Design Patterns

---

Course conclusion