INSTITUTO FEDERAL
Fluminense
Campus Itaperuna

MINISTÉRIO DA
EDUCAÇÃO

SQLAlchemy em Python

Apostila Didática

1ª Ed., Setembro de 2023

Sumário

I Parte Um

1 Introdução à biblioteca SQLAlchemy em Python 9

1.1 O que é ORM 9

1.2 SQLAlchemy 10

1.3 Criando um ambiente virtual em Python 11

1.3.1 Criando um ambiente virtual no Visual Studio Code 11

1.3.2 Criando um ambiente virtual via prompt de comando 12

1.4 Instalando a biblioteca SQLAlchemy 13

1.4.1 Principais funções 14

1.4.2 Conexão ao banco de dados 15

1.4.3 Extensões de produtividade 16

2 Caso de uso - Sistema para Teatro 19

2.1 Minimundo Teatro 19

2.1.1 Modelo de entidade e relacionamento 19

2.2 Criando um projeto com Visual Studio Code 20

2.2.1 Mapeando as tabelas do banco de dados com SQLAlchemy 21

2.2.2 Conectando ao banco de dados teatro 25

2.3 Conhecendo o padrão de projeto Repository 25

2.3.1 O que é o padrão de projeto Repository 26

2.3.2 Criando um exemplo prático 29

2.3.3 Testando as classes 31
2.3.4 SCRIPT SQL - MySQL 32

II Parte Dois

3 QTDesigner e Pyside6 37

3.1 Instalando o PySide6 39

Parte Um

1 Introdução à biblioteca SQLAlchemy em Python 9

1.1 O que é ORM

1.2 SQLAlchemy

1.3 Criando um ambiente virtual em Python

1.4 Instalando a biblioteca SQLAlchemy

2 Caso de uso - Sistema para Teatro . . . 19

2.1 Minimundo Teatro

2.2 Criando um projeto com Visual Studio Code

2.3 Conhecendo o padrão de projeto Repository

1. Introdução à biblioteca SQLAlchemy em Python

1.1 O que é ORM

ORM (Object-Relational Mapping) é uma técnica de programação que permite mapear objetos em um sistema orientado a objetos para tabelas em um banco de dados relacional. Em outras palavras, é uma forma de representar objetos e suas relações em um banco de dados relacional usando classes e objetos em uma linguagem de programação.

O ORM facilita o trabalho com bancos de dados relacionais, permitindo que os desenvolvedores realizem operações de criação, leitura, atualização e exclusão (CRUD) em um banco de dados por meio de objetos e métodos familiares em uma linguagem de programação, como Python.

Com o ORM, você pode definir modelos de dados como classes Python que representam tabelas no banco de dados. Cada atributo da classe representa uma coluna da tabela e os objetos dessa classe representam registros da tabela. O ORM cuida de mapear automaticamente os objetos e seus atributos para as tabelas e colunas correspondentes no banco de dados, eliminando a necessidade de escrever consultas SQL manualmente.

Além disso, o ORM oferece recursos para trabalhar com relacionamentos entre tabelas, como associações, chaves estrangeiras e junções. Isso facilita o estabelecimento e a navegação de relações entre objetos, tornando mais simples e intuitivo o acesso a dados relacionados.

Os ORM mais populares para Python incluem a SQLAlchemy, Django ORM (parte do framework Django) e o Peewee. Cada um desses ORM possui suas próprias características e sintaxe, mas todos têm o objetivo comum de simplificar o trabalho com bancos de dados relacionais, abstraindo as complexidades do SQL e fornecendo uma interface de programação orientada a objetos.

Em resumo, o ORM é uma técnica que permite o mapeamento entre objetos e tabelas de banco de dados relacionais, facilitando a interação com o banco de dados usando uma linguagem de programação orientada a objetos. Ele oferece uma abstração de nível mais alto e ajuda a reduzir a quantidade de código

repetitivo e complexo associado ao acesso direto a bancos de dados.

1.2 SQLAlchemy

A biblioteca SQLAlchemy é uma poderosa ferramenta de mapeamento objeto-relacional (ORM) para Python. Ela fornece uma interface abstrata para interagir com bancos de dados relacionais de forma mais simples e intuitiva, permitindo que você trabalhe com objetos Python em vez de escrever consultas SQL manualmente. A SQLAlchemy oferece suporte a uma variedade de bancos de dados, como SQLite, MySQL, PostgreSQL e Oracle.

Com a SQLAlchemy, você pode:

- Estabelecer conexões com bancos de dados: A biblioteca fornece uma API para criar e gerenciar conexões com o banco de dados.
- Mapear classes Python para tabelas de banco de dados: Você pode criar classes Python que representam tabelas no banco de dados. Essas classes são chamadas de modelos ou classes de entidade. Os atributos dessas classes representam as colunas da tabela.
- Executar consultas e operações de banco de dados: A SQLAlchemy oferece uma linguagem de consulta expressiva chamada SQL Alchemy Expression Language (SAEL), que permite escrever consultas de banco de dados de forma legível e intuitiva usando métodos e operadores Python.
- Realizar operações de inserção, atualização e exclusão: Você pode criar, modificar e excluir registros de banco de dados usando a SQLAlchemy, utilizando métodos e propriedades dos modelos.
- Trabalhar com relacionamentos entre tabelas: A SQLAlchemy facilita o estabelecimento e a navegação de relacionamentos entre tabelas, permitindo que você defina associações entre modelos e consulte dados relacionados.
- Gerenciar transações: A biblioteca suporta o conceito de transações, permitindo que você agrupe operações de banco de dados em unidades lógicas e as execute como uma única transação.

A SQLAlchemy é altamente flexível e modular. Ela oferece diferentes componentes, como Core, ORM e SQL, que podem ser usados de forma independente ou combinados para atender às suas necessidades específicas. Além disso, a biblioteca fornece suporte para consultas complexas, agregações, junções, subconsultas e muito mais.

Em resumo, a SQLAlchemy é uma biblioteca poderosa e popular para trabalhar com bancos de dados relacionais em Python, oferecendo uma abstração conveniente e intuitiva sobre a escrita de consultas SQL. Seu uso pode simplificar o desenvolvimento de aplicativos que interagem com bancos de dados e melhorar a produtividade dos desenvolvedores.

1.3 Criando um ambiente virtual em Python

Um ambiente virtual é uma ferramenta que permite isolar as dependências de um projeto Python. Em outras palavras, ele cria um ambiente separado onde você pode instalar bibliotecas e pacotes específicos para o seu projeto sem interferir no ambiente Python global do sistema.

Ao criar um ambiente virtual, você pode definir um conjunto específico de dependências para o seu projeto, incluindo versões específicas de bibliotecas. Isso é útil quando diferentes projetos requerem versões diferentes das mesmas bibliotecas ou quando você precisa garantir a reprodutibilidade do seu ambiente de desenvolvimento.

Ao trabalhar com ambientes virtuais, é uma prática recomendada manter um arquivo de requisitos (requirements.txt) para registrar todas as bibliotecas e suas versões. Isso facilita a replicação do ambiente em outro local e ajuda na colaboração com outros desenvolvedores.

Os ambientes virtuais são amplamente utilizados na comunidade Python devido à sua flexibilidade e capacidade de gerenciamento de dependências. Eles são uma ótima maneira de garantir que cada projeto tenha suas próprias dependências isoladas e possa ser executado de forma consistente em diferentes ambientes.

1.3.1 Criando um ambiente virtual no Visual Studio Code

Para configurar um ambiente virtual em Python no Visual Studio Code utilize a tecla de atalho "Control + Shift + P", seguindo os passos abaixo:

1. Abra o Visual Studio Code.
2. Abra o diretório do seu projeto ou crie um novo diretório para o projeto.
3. Pressione **"Control + Shift + P"** (ou **"Command + Shift + P"** no macOS) para abrir a paleta de comandos.
4. Na paleta de comandos, digite **"Python: Create Enviroment"** e pressione Enter.
5. Uma lista suspensa será exibida com as opções de interpretadores Python disponíveis. Selecione o interpretador desejado para criar um ambiente virtual.
6. O Visual Studio Code criará automaticamente o ambiente virtual para o interpretador Python selecionado no diretório do seu projeto.
7. Uma vez que o ambiente virtual tenha sido criado, ele será ativado automaticamente no Visual Studio Code.

A partir desse ponto, você pode instalar pacotes e bibliotecas específicos para o seu projeto dentro do ambiente virtual usando o terminal integrado do Visual Studio Code.

Lembre-se de ativar o ambiente virtual sempre que abrir o Visual Studio Code para trabalhar no projeto. Isso garantirá que o ambiente virtual esteja ativado e as bibliotecas instaladas estejam disponíveis para o seu código.

1.3.2 Criando um ambiente virtual via prompt de comando

Para criar um ambiente virtual em Python, você pode usar a biblioteca venv, que é uma biblioteca padrão a partir da versão 3.3 do Python. Siga as instruções abaixo para criar um ambiente virtual:

1. Abra o terminal ou prompt de comando.
2. Navegue até o diretório em que você deseja criar o ambiente virtual.
3. Execute o seguinte comando para criar o ambiente virtual:

   No Windows:

   ```
   python -m venv nome_do_ambiente
   ```

   No macOS e Linux:

   ```
   python3 -m venv nome_do_ambiente
   ```

   Substitua "nome_do_ambiente"pelo nome que desejar dar ao seu ambiente virtual. Pode ser qualquer nome válido.
4. Aguarde até que o comando seja concluído. Ele criará uma nova pasta com o nome do ambiente virtual no diretório atual.
5. Para ativar o ambiente virtual, execute o comando apropriado para o seu sistema operacional:

   No Windows:

   ```
   nome_do_ambiente\Scripts\activate
   ```

   No macOS e Linux:

   ```
   source nome_do_ambiente/bin/activate
   ```

   Após a ativação do ambiente virtual, você verá que o prompt de comando será modificado para mostrar o nome do ambiente entre parênteses, indicando que o ambiente virtual está ativo.
6. Agora você pode instalar as bibliotecas e pacotes específicos do seu projeto dentro do ambiente

virtual, usando o **pip**.

7. Para desativar o ambiente virtual, execute o comando **deactivate**.

Ao criar um ambiente virtual, você terá um espaço isolado onde as dependências e pacotes instalados não interferirão em outros ambientes virtuais ou no ambiente Python global do sistema. Isso permite que você mantenha suas dependências de projeto separadas e evite conflitos entre diferentes projetos.

Lembre-se de ativar o ambiente virtual sempre que for trabalhar em seu projeto, para garantir que as bibliotecas e pacotes instalados sejam usados corretamente.

1.4 Instalando a biblioteca SQLAlchemy

Para instalar a biblioteca SQLAlchemy em seu ambiente Python, você pode usar o gerenciador de pacotes pip. Siga as instruções abaixo:

1. Abra o terminal ou prompt de comando.
2. Execute o seguinte comando para instalar a SQLAlchemy:

```
1  pip install sqlalchemy
```

Aguarde até que o processo de instalação seja concluído. O pip baixará e instalará a biblioteca SQLAlchemy e suas dependências, se necessário.

Após a conclusão da instalação, você poderá importar a biblioteca SQLAlchemy em seus scripts Python sem problemas.

Certifique-se de que você tenha uma conexão com a internet durante o processo de instalação para que o pip possa baixar os pacotes necessários.

Lembrando que é recomendado criar um ambiente virtual para seus projetos Python, pois isso ajuda a isolar as dependências e evita conflitos entre diferentes projetos. Você pode usar ferramentas como o virtualenv ou conda para criar e gerenciar ambientes virtuais.

Com a biblioteca SQLAlchemy devidamente instalada, você estará pronto para começar a usá-la em seus projetos Python para interagir com bancos de dados relacionais de forma mais fácil e eficiente.

1.4.1 Principais funções

Criar uma instância do mecanismo de conexão ao banco de dados

```
1  from sqlalchemy import create_engine
2  engine = create_engine('mysql://user:password@localhost/mydatabase')
```

A função **create_engine()** cria uma instância do mecanismo de conexão ao banco de dados. Você precisa fornecer a URL de conexão correta para o seu banco de dados específico.

Criar tabelas no banco de dados

```
1  from sqlalchemy.ext.declarative import declarative_base
2  from sqlalchemy import Column, Integer, String
3
4  Base = declarative_base()
5
6  class Pessoa(Base):
7      __tablename__ = 'pessoa'
8      id = Column(Integer, primary_key=True)
9      nome = Column(String(50))
10
```

A seguir uma explicação detalhada da classe **Pessoa**:

- **class Pessoa(Base):**

  Definimos uma classe chamada Pessoa, que é uma subclasse da classe base Base. A classe base Base deve ser uma classe que você definiu anteriormente usando a função declarative_base(). Essa classe base é responsável por fornecer funcionalidades de mapeamento objeto-relacional para a classe Pessoa. Pessoa será mapeada para uma tabela no banco de dados.

- **__tablename__ = 'pessoa':**

  Usamos o atributo especial __tablename__ para definir o nome da tabela correspondente no banco de dados. Nesse caso, o nome da tabela será 'pessoa'.

- **id = Column(Integer, primary_key=True):**

  Criamos um atributo chamado id que representa a coluna 'id' na tabela 'pessoa'. Usamos a função Column para definir a coluna. Passamos Integer como primeiro argumento, indicando que a coluna

armazenará valores inteiros. O argumento primary_key=True indica que essa coluna será a chave primária da tabela.

- **nome = Column(String(50)):**

  Criamos um atributo chamado nome que representa a coluna 'nome' na tabela 'pessoa'. Usamos a função Column novamente para definir a coluna. Passamos String(50) como primeiro argumento, indicando que a coluna armazenará strings com no máximo 50 caracteres. Dessa forma, a classe Pessoa está mapeada para a tabela 'pessoa' no banco de dados. A tabela terá duas colunas: 'id' e 'nome'. O atributo id será a chave primária da tabela e o atributo nome armazenará os nomes das pessoas.

Essa estrutura permite que você interaja com o banco de dados usando objetos da classe Pessoa. Por exemplo, você pode criar instâncias dessa classe, atribuir valores aos atributos id e nome, e usar a biblioteca SQLAlchemy para realizar operações de banco de dados, como inserir, atualizar ou consultar registros na tabela 'pessoa'.

Lembre-se de que este é apenas um exemplo isolado da definição da classe Pessoa. Em um projeto completo, você pode ter várias classes de modelo representando diferentes tabelas no banco de dados e estabelecendo relacionamentos entre elas, dependendo dos requisitos do seu sistema.

1.4.2 Conexão ao banco de dados

Para se conectar a um banco de dados usando a biblioteca SQLAlchemy em Python, você precisa fornecer as informações de conexão, como o tipo de banco de dados, o host, a porta, o nome do banco de dados, o usuário e a senha (quando aplicável). Aqui está um exemplo de como realizar a conexão:

```
1  from sqlalchemy import create_engine
2
3  # Defina a string de conexão
4  # O exemplo abaixo é para um banco de dados SQLite
5  # Substitua as informações de acordo com o seu banco de dados
6  connection_string = 'sqlite:///caminho/para/banco_de_dados.db'
7
8  # Crie o objeto de conexão
9  engine = create_engine(connection_string)
10
```

```
11  # Realize a conexão
12  connection = engine.connect()
13
14  # Agora você está conectado ao banco de dados!
15  # Você pode executar consultas e operações no banco de dados usando o objeto "connection"
```

No exemplo acima, usamos o SQLAlchemy para criar um objeto de conexão chamado engine usando a função **create_engine()**. Passamos a string de conexão apropriada para o banco de dados que você está usando. No exemplo, usamos o SQLite como banco de dados de exemplo, mas você pode substituir a string de conexão de acordo com o seu banco de dados específico.

Em seguida, chamamos o método **connect()** do objeto engine para estabelecer a conexão com o banco de dados. Isso retorna um objeto de conexão que pode ser usado para executar consultas e operações no banco de dados.

Após estabelecer a conexão, você pode usar o objeto **connection** para executar consultas e operações no banco de dados, conforme necessário. Por exemplo, você pode usar o método **execute()** para executar uma consulta SQL ou usar métodos auxiliares, como **execute()** e **fetchall()**, para recuperar os resultados das consultas.

Lembre-se de ajustar a string de conexão de acordo com o seu banco de dados específico. Consulte a documentação da **SQLAlchemy** para obter mais informações sobre as diferentes sintaxes de string de conexão e as opções específicas para cada tipo de banco de dados. A seguir, a tabela 1.1 apresenta alguns exemplos de strings de conexão com banco de dados.

Banco de Dados	**String de Conexão**
SQLite	`sqlite:///caminho/para/banco_de_dados.db`
MySQL	`mysql+pymysql://usuario:senha@host/nome_do_banco`
PostgreSQL	`postgresql+psycopg2://usuario:senha@host/nome_do_banco`
Oracle	`oracle+cx_oracle://usuario:senha@host:porta/nome_do_banco`

Tabela 1.1: Strings de conexão

1.4.3 Extensões de produtividade

Para melhorar a produtividade dos desenvolvedores que utilizam Python em conjunto com o VS Code, existem várias extensões disponíveis que adicionam recursos e funcionalidades específicas para o framework. Essas extensões ajudam a simplificar tarefas comuns, oferecem ferramentas para depuração e testes, além de melhorar a experiência geral de desenvolvimento.

Para instalar extensões, pressione a tecla de atalho **Ctrl + Shift + X**. A seguir, apresentamos algumas das principais extensões de produtividade para desenvolver aplicativos Python com o Visual Studio Code:

- **Python:** A extensão Python para o Visual Studio Code é uma das extensões mais populares e essenciais para desenvolvedores Python. Ela oferece um conjunto abrangente de recursos para tornar o desenvolvimento Python mais eficiente e agradável.
- **Erro Lens:** é uma ferramenta de produtividade geral para desenvolvedores que trabalham com várias linguagens de programação no Visual Studio Code. Ela oferece recursos avançados para realçar, anotar e fornecer informações adicionais sobre erros e avisos no código-fonte. Essa extensão é útil para ajudar a identificar e entender problemas em seu código, independentemente da linguagem de programação que você está usando.
- **Qt for Python:** O **Qt for Python** é uma extensão que fornece suporte ao desenvolvimento de aplicativos desktop usando o framework Qt em conjunto com Python. O Qt é uma biblioteca amplamente utilizada para criar interfaces gráficas de usuário (GUI) multiplataforma.
- **Project Manager:** A extensão Project Manager é útil para organizar e gerenciar seus projetos dentro do Visual Studio Code. Ela permite que você defina pastas de trabalho específicas para projetos e salve os projetos para fácil acesso posterior.

2. Caso de uso - Sistema para Teatro

2.1 Minimundo Teatro

No sistema de teatro, deseja-se controlar informações sobre atores, peças teatrais, apresentações das peças, teatros e o controle de ingressos vendidos.

Atores: Os atores são profissionais de teatro que participam de peças teatrais. Cada ator possui um nome, data de nascimento, gênero, nacionalidade e outros atributos relevantes. Um ator pode participar de várias peças.

Peças Teatrais: As peças são produções teatrais que envolvem a atuação de diversos atores. Cada peça possui um título, uma descrição, uma lista de atores que participam dela e outros atributos relevantes. Uma peça pode ter vários atores e um ator pode participar de várias peças.

Apresentações: As apresentações são eventos específicos em que uma peça é encenada em um teatro. Cada apresentação possui uma data e hora específicas, uma peça relacionada, o teatro onde ocorrerá e outros atributos relevantes. Uma peça pode ter várias apresentações.

Teatros: Os teatros são locais onde as apresentações das peças ocorrem. Cada teatro possui um nome, um endereço, uma capacidade de assentos e outros atributos relevantes. Várias apresentações podem ocorrer em um teatro.

Controle de Ingressos: O sistema deve permitir o controle de ingressos vendidos para cada apresentação. Cada ingresso possui um preço, tipo de cadeira uma data e está associado a uma apresentação específica.

2.1.1 Modelo de entidade e relacionamento

A seguir, a figura 2.1 ilustra o diagrama de entidade e relacionamento e na seção 2.3.4 apresenta código SQL para o banco de dados MySQL.

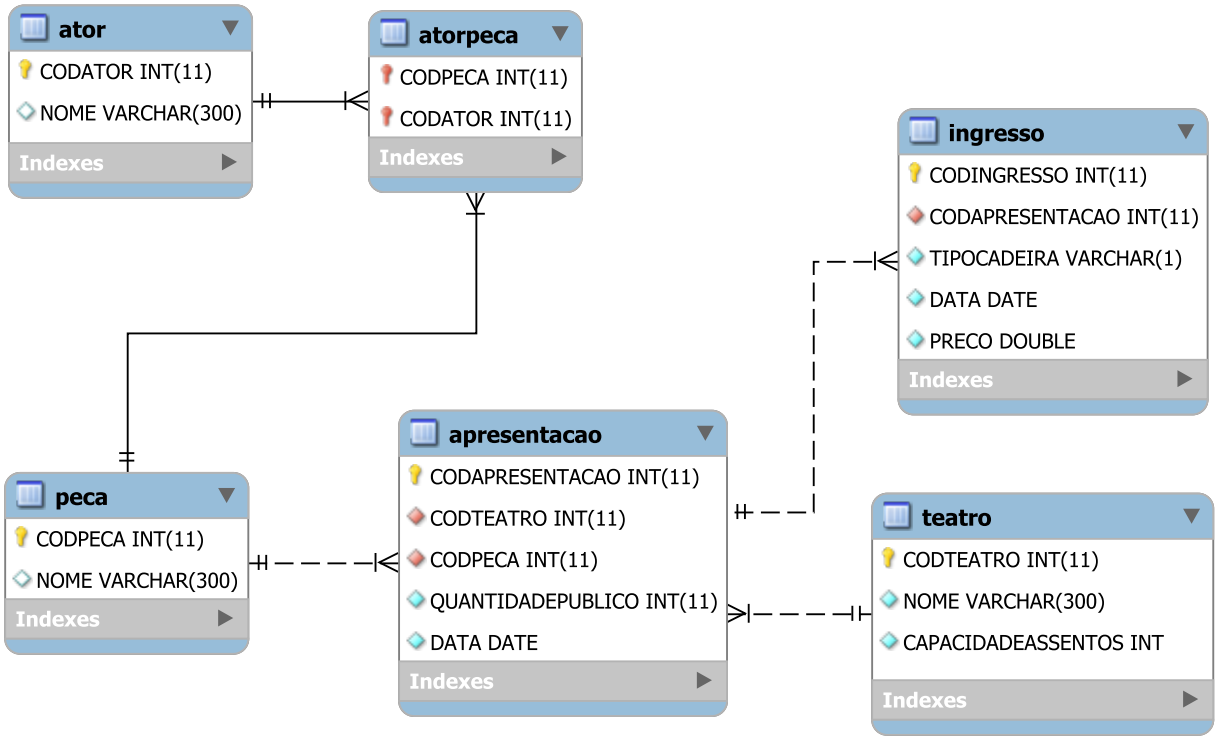

Figura 2.1: Modelo de Entidade e Relacionamento - Teatro

2.2 Criando um projeto com Visual Studio Code

Primeiro crie uma pasta chamada **teatro** e abra com o Visual Studio Code. Após abrir a pasta, crie um ambiente virtual em python conforme a seção 1.3.1.

A seguir, abra o terminal no Visual Studio Code e instale a biblioteca SQLAlchemy usando o seguinte comando:

```
pip install sqlalchemy
```

Com a biblioteca instalada, crie a seguinte estrutura de diretórios (2.2) e arquivos na pasta do projeto chamada **teatro**.

```
teatro
├── dominio
│   └── ...
└── main.py
```

Figura 2.2: Estrutura de diretório e arquivos do projeto **teatro**.

- **dominio:** Pasta onde será adicionado todas as regras de negócio da aplicação, por exemplo, inserções, exclusões, alterações e consultas.

2.2.1 Mapeando as tabelas do banco de dados com SQLAlchemy

Nesta seção, iremos aprender como podemos mapear as tabelas do banco de dados em modelos de classes para que possamos utilizar a biblioteca SQLAchemy.

Importações dos pacotes

O primeiro passo é importar as classes e funções necessárias da biblioteca SQLAlchemy. Para isso, crie um arquivo em python chamado **entidades.py**, Figura 2.3, implemente o seguinte código de acordo com a Figura 2.4.

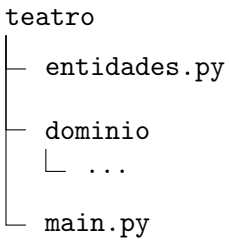

Figura 2.3: Estrutura de diretório e arquivos do projeto **teatro**.

```
1  from sqlalchemy import Column, Integer, String, Date, Double, ForeignKey
2  from sqlalchemy.ext.declarative import declarative_base
3  from sqlalchemy.orm import relationship
4
5  Base = declarative_base()
```

Figura 2.4: Importe as classes e funções necessárias da biblioteca SQLAlchemy.

Observe que, na Figura 2.4 importamos várias classes e funções da biblioteca **SQLAlchemy**. A seguir, descrevemos o objetivo das classes e funções.

- **Column**: é usado para definir as colunas das tabelas.
- **Integer**, **String**, **Date** e **Double** são tipos de dados específicos para as colunas.
- **ForeignKey** é usado para definir as chaves estrangeiras.
- **declarative_base** é uma função usada para criar a classe base para as classes de modelo.
- **relationship** é usado para definir relacionamentos entre as tabelas.

Observe a Figura 2.4 linha 5, criamos uma classe **Base** usando a função **declarative_base**. Todas as classes de modelo serão subclasses dessa classe base. Essa classe servirá como base para todas as outras classes de modelo que criaremos posteriormente.

Como próximo passo, devemos implementar as classes que representam as entidades do banco de dados no arquivo **entidades.py**. Cada classe é uma subclasse da classe base do modelo que criamos

anteriormente. Para cada classe de modelo, usamos a anotação **__tablename__** para definir o nome da tabela correspondente no banco de dados. Dentro de cada classe de modelo, definimos os atributos da tabela usando a função **Column** e especificando o tipo de dado, restrições de chave primária, chave estrangeira e outras opções. Também podemos definir relacionamentos entre as tabelas usando a função **relationship** e especificando a classe de modelo relacionada. A classe **Teatro** é a primeira que iremos implementar conforme a Figura 2.5 no arquivo **entidades.py**.

```
class Teatro(Base):
    __tablename__ = 'teatro'
    CODTEATRO = Column(Integer, primary_key=True)
    NOME = Column(String(300), nullable=False)
    CAPACIDADEASSENTOS = Column(Integer)
```

Figura 2.5: Classe Teatro

Ao implementar a classe **Teatro** (Figura 2.5) os atributos da classe **DEVEM** possuir o mesmo nome das colunas das tabelas do banco de dados.

Outras considerações importantes: A classe Teatro é definida como uma subclasse da classe base **Base**. O atributo **__tablename__** define o nome da tabela correspondente no banco de dados. Em seguida, definimos as colunas da tabela **teatro**. Nesse caso, temos a coluna **CODTEATRO** do tipo **Integer** como chave primária, a coluna **NOME** do tipo **String** e a coluna **CAPACIDADEASSENTOS** do tipo **Integer**.

Implemente também a classe **Ator** conforme a Figura 2.6.

```
class Ator(Base):
    __tablename__ = 'ator'
    CODATOR = Column(Integer, primary_key=True)
    NOME = Column(String(300))
```

Figura 2.6: Classe Ator

Agora vamos implementar as classes **Peca** (Figura 2.7) e **AtorPeca** (Figura 2.8) que representam as tabelas do banco de dados **'peca'** e **'atorpeca'** respectivamente. Diferentemente das classes **Teatro**, **Ator** e **Peca** que mapeia tabelas do banco de dados que são entidades **"fortes"**. A tabela **'atorpeca'** é uma entidade **"fraca"** que possuiu a chave estrangeira proveniente das tabelas **'ator'** e **'peca'**.

Agora vamos analisar um novo atributo chamado **atores** na classe **Peca**. Criamos um atributo chamado

atores que representa o relacionamento entre as tabelas **'peca'** e **'ator'**. Usamos a função **relationship** para definir o relacionamento. Passamos **'Ator'** como primeiro argumento, indicando a classe de modelo que representa a tabela **ator**. O argumento **secondary='atorpeca'** especifica que a tabela de associação entre **peca** e **ator** é a tabela **atorpeca**. Essa tabela de associação é usada para estabelecer o relacionamento muitos-para-muitos entre **'peca'** e **'ator'**, onde uma peça pode ter vários atores e um ator pode estar em várias peças. O relacionamento entre **'peca'** e **'ator'** é estabelecido por meio do atributo **atores**, que representa a tabela de associação **'atorpeca'**. Esse relacionamento permite acessar os atores associados a uma peça por meio desse atributo.

```
class Peca(Base):
    __tablename__ = 'peca'
    CODPECA = Column(Integer, primary_key=True)
    NOME = Column(String(300))
    atores = relationship('Ator', secondary='atorpeca')
```

Figura 2.7: Classe Peca

```
class AtorPeca(Base):
    __tablename__ = 'atorpeca'
    CODPECA = Column(Integer, ForeignKey('peca.CODPECA'), primary_key=True)
    CODATOR = Column(Integer, ForeignKey('ator.CODATOR'), primary_key=True)
```

Figura 2.8: Classe AtorPeca

Com relação a classe **AtorPeca** (Figura 2.8) podemos observar os seguintes detalhes para os atributos **CODPECA** e **CODATOR**:

- **CODPECA = Column(Integer, ForeignKey('peca.CODPECA'), primary_key=True):** Criamos um atributo chamado **CODPECA** que representa a coluna **'CODPECA'** na tabela **'atorpeca'**. Usamos a função **Column** para definir a coluna. Passamos **Integer** como primeiro argumento, indicando que a coluna armazenará valores inteiros. O argumento **ForeignKey('peca.CODPECA')** especifica que essa coluna é uma chave estrangeira que faz referência à coluna **'CODPECA'** na tabela **'peca'**. O argumento **primary_key=True** indica que essa coluna também será parte da chave primária composta da tabela.
- **CODATOR = Column(Integer, ForeignKey('ator.CODATOR'), primary_key=True):** Criamos um atributo chamado **CODATOR** que representa a coluna **'CODATOR'** na tabela **'atorpeca'**. Usamos a função **Column** novamente para definir a coluna. Passamos **Integer** como primeiro argumento, indicando que a coluna armazenará valores inteiros. O argumento **Foreign-**

Key('ator.CODATOR') especifica que essa coluna é uma chave estrangeira que faz referência à coluna **'CODATOR'** na tabela **'ator'**. O argumento **primary_key=True** indica que essa coluna também será parte da chave primária composta da tabela.

Com esse código, a classe **AtorPeca** está mapeada para a tabela **'atorpeca'** no banco de dados. A tabela terá duas colunas: **'CODPECA'** e **'CODATOR'**. Ambas as colunas fazem parte da chave primária composta da tabela, o que significa que juntas elas identificam exclusivamente cada registro na tabela.

Essa tabela **'atorpeca'** é usada como tabela de associação entre as tabelas **'ator'** e **'peca'**. Ela estabelece o relacionamento muitos-para-muitos entre atores e peças, indicando quais atores estão associados a quais peças. As chaves estrangeiras **'CODPECA'** e **'CODATOR'** fazem referência às chaves primárias das tabelas **'peca'** e **'ator'**, respectivamente.

Por fim, podemos implementar as duas últimas classes, **Apresentacao** (Figura 2.9) e **Ingresso** (Figura 2.10). Classes que estão mapeadas para as tabelas **'apresentacao'** e **'ingresso'**.

```
class Apresentacao(Base):
    __tablename__ = 'apresentacao'
    CODAPRESENTACAO = Column(Integer, primary_key=True)
    CODTEATRO = Column(Integer, ForeignKey('teatro.CODTEATRO'))
    CODPECA = Column(Integer, ForeignKey('peca.CODPECA'))
    QUANTIDADEPUBLICO = Column(Integer, default=0)
    DATA = Column(Date)

    teatro = relationship('Teatro')
    peca = relationship('Peca')
    ingressos = relationship('Ingresso')
```

Figura 2.9: Classe Apresentacao

```
class Ingresso(Base):
    __tablename__ = 'ingresso'
    CODINGRESSO = Column(Integer, primary_key=True)
    CODAPRESENTACAO = Column(Integer, ForeignKey('apresentacao.CODAPRESENTACAO'))
    TIPOCADEIRA = Column(String(1), default='S')
    DATA = Column(Date)
    PRECO = Column(Double, default=0)

    apresentacao = relationship('Apresentacao')
```

Figura 2.10: Classe Ingresso

2.2.2 Conectando ao banco de dados teatro

Nesta seção, iremos aprender como podemos criar uma conexão com o banco de dados MySQL. Como próximo passo, é necessário criar um arquivo chamado **conexao.py** na pasta **teatro** conforme a Figura 2.11.

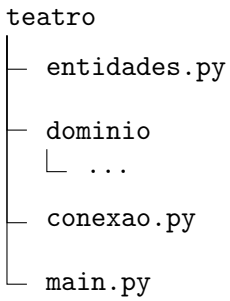

Figura 2.11: Estrutura de diretório e arquivos do projeto **teatro**.

Após criar o arquivo **conexao.py**, implemente o seguir código conforme a Figura 2.12.

A seguir, uma descrição sobre o código apresentado na Figura 2.12:

- Você deve substituir 'seu_usuario', 'sua_senha', 'localhost', '3306' e 'nome_do_banco_de_dados' com as informações corretas para o seu ambiente MySQL.
- A URL de conexão é criada usando a sintaxe 'mysql+pymysql://'. O pacote **pymysql** é um driver Python para MySQL que é usado pelo SQLAlchemy para se conectar ao banco de dados.
- Em seguida, criamos um mecanismo de conexão chamando a função **create_engine** e passando a URL de conexão como argumento.
- Em seguida, tentamos estabelecer uma conexão com o banco de dados usando o método **connect** do mecanismo de conexão. Se a conexão for bem-sucedida, uma mensagem *"Conexão estabelecida com sucesso!"* será impressa no console. No caso de uma falha na conexão, uma mensagem de erro será impressa.

Lembre-se de instalar o pacote pymysql antes de executar o código. Você pode instalar o pacote usando o comando **pip install pymysql**.

2.3 Conhecendo o padrão de projeto Repository

Até o presente momento, implementamos a estrutura base do projeto, mapeamento das classes e o mecanismo de conexão com o banco de dados. A partir desse ponto iremos implementar as operações responsáveis por manipular os dados, tais como, consultas, inclusões, exclusão e alterações. Entretanto, devemos entender o que é o padrão de projeto **Repository**.

```python
from sqlalchemy import create_engine

# Configuração da conexão com o banco de dados
user = 'seu_usuario'
password = 'sua_senha'
host = 'localhost'
port = '3306'
database = 'nome_do_banco_de_dados'

# Criação da URL de conexão
url = f'mysql+pymysql://{user}:{password}@{host}:{port}/{database}'

# Criação do mecanismo de conexão
engine = create_engine(url)
session = None

# Teste de conexão
try:
    connection = engine.connect()
    print("Conexão estabelecida com sucesso!")
    connection.close()

    Base.metadata.create_all(engine)

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

except Exception as e:
    print("Falha na conexão:", e)
```

Figura 2.12: Arquivo conexao.py

2.3.1 O que é o padrão de projeto Repository

O padrão de projeto Repository é um padrão arquitetural que tem como objetivo separar a lógica de negócio da lógica de persistência de dados. Ele fornece uma camada de abstração entre a aplicação e a fonte de dados, permitindo que as operações de leitura e gravação sejam realizadas de forma transparente para a lógica de negócio.

O Repository atua como uma interface entre a aplicação e a camada de persistência de dados. Ele encapsula o acesso aos dados, fornecendo métodos para realizar operações CRUD (Create, Read, Update, Delete) em uma determinada entidade. O Repository esconde os detalhes de implementação da persistência, permitindo que a lógica de negócio se concentre apenas na manipulação dos objetos do domínio.

Principais benefícios do padrão Repository:

Separação de responsabilidades: O Repository separa as operações de acesso aos dados da lógica de negócio, promovendo uma melhor organização e modularidade do código.

Abstração do acesso aos dados: O Repository fornece uma interface consistente e bem definida para acessar os dados, ocultando os detalhes de implementação da camada de persistência.

Flexibilidade na troca de fonte de dados: O Repository permite que a aplicação troque facilmente de fonte de dados, seja ela um banco de dados relacional, um banco de dados NoSQL, um serviço web, etc., sem afetar a lógica de negócio.

Reutilização de código: Ao encapsular a lógica de acesso aos dados em um Repository, é possível reutilizar esse código em diferentes partes da aplicação, evitando duplicação e facilitando a manutenção.

Testabilidade: O Repository facilita a realização de testes, pois é possível criar implementações de Repositories em memória ou mocká-las para simular o acesso aos dados durante os testes.

Em resumo, o padrão Repository ajuda a separar as preocupações entre a lógica de negócio e a persistência de dados, promovendo uma melhor organização do código, reutilização, flexibilidade e testabilidade. Ele simplifica o acesso e a manipulação dos dados, permitindo que a aplicação se concentre na lógica de negócio e na manipulação dos objetos do domínio.

A ideia central do padrão Repository é criar uma interface que define as operações de acesso aos dados (como buscar, salvar, atualizar e excluir), e em seguida, fornecer implementações concretas dessa interface para lidar com as operações de persistência em diferentes fontes de dados, como bancos de dados relacionais, bancos de dados NoSQL, serviços web, arquivos, etc. Essas implementações concretas são responsáveis por lidar com os detalhes específicos de cada fonte de dados, enquanto a lógica de negócio permanece isolada e não dependente desses detalhes.

A arquitetura do padrão Repository geralmente envolve as seguintes camadas:

- **Camada de Domínio (ou Modelo):**

  Nessa camada, são definidas as entidades de domínio do sistema, ou seja, as classes que representam os conceitos principais do negócio. Essas entidades possuem atributos e comportamentos relacionados ao negócio e não estão diretamente relacionadas à persistência de dados.

- **Camada de Repositório:**

  Essa camada contém as interfaces de repositório, que definem os contratos para as operações de acesso aos dados, como buscar, salvar, atualizar e excluir. As interfaces de repositório são independentes da tecnologia de persistência utilizada. Além das interfaces, também existem as implementações concretas dos repositórios, que são responsáveis por realizar as operações de persistência em uma fonte de dados específica, como um banco de dados.

- **Camada de Serviço (ou Aplicação):**

  Essa camada contém a lógica de negócio do sistema. Ela utiliza os repositórios para realizar operações de acesso aos dados de forma transparente, sem se preocupar com os detalhes de como os dados são persistidos. Os serviços (ou classes de aplicação) na camada de serviço orquestram as operações de negócio, utilizando os repositórios para obter ou persistir os dados necessários.

- **Camada de Infraestrutura (ou Persistência):**

  Essa camada contém as implementações concretas dos repositórios, que são responsáveis por realizar as operações de persistência em uma fonte de dados específica. Também pode incluir classes auxiliares, como uma fábrica de conexões com o banco de dados, classes de mapeamento objeto-relacional, entre outras.

2.3.2 Criando um exemplo prático

Vamos considerar as seguintes entidades: **Peca**, **Teatro**, **Apresentacao**, **Ator** e **Ingresso**. Vamos fornecer um exemplo básico das classes de repositório para cada uma dessas entidades.

Primeiro passo - Classe Base do Repositório

Vamos começar criando uma classe base para os repositórios. Essa classe irá conter as operações básicas de acesso ao banco de dados, como inserir, atualizar, excluir e consultar registros. Crie um arquivo chamado **repositorio.py**.

```python
from sqlalchemy.orm import Session
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

class BaseRepository:
    def __init__(self, session: Session):
        self.session = session

    def add(self, entity):
        self.session.add(entity)

    def update(self, entity):
        self.session.merge(entity)

    def delete(self, entity):
        self.session.delete(entity)

    def get_all(self):
        return self.session.query(self.model).all()

    def get_by_id(self, id):
        return self.session.query(self.model).get(id)
```

Figura 2.13: Classe BaseRepository.py

Segundo passo - Classe Repositórios para cada entidade

Como segundo passo, vamos criar os repositórios específicos para cada entidade no arquivo **repositorio.py**.

```python
class PecaRepository(BaseRepository):

    def __init__(self, session: Session):

        super().__init__(session)
```

```
        self.model = Peca

class TeatroRepository(BaseRepository):
    def __init__(self, session: Session):
        super().__init__(session)
        self.model = Teatro

class TeatroRepository(BaseRepository):
    def __init__(self, session: Session):
        super().__init__(session)
        self.model = Teatro

class TeatroRepository(BaseRepository):
    def __init__(self, session: Session):
        super().__init__(session)
        self.model = Teatro

class IngressoRepository(BaseRepository):
    def __init__(self, session: Session):
        super().__init__(session)
        self.model = Ingresso
```

Essas são as classes básicas de repositório para gerenciar os dados das entidades do sistema de teatro usando o padrão de projeto Repository com a biblioteca SQLAlchemy. Você pode adicionar métodos adicionais aos repositórios, conforme necessário, para realizar consultas mais complexas ou operações específicas.

Lembre-se de criar uma instância do mecanismo de conexão ao banco de dados, criar a sessão e passá-la para os repositórios para que eles possam interagir com o banco de dados corretamente.

2.3.3 Testando as classes

```
from dominio import *
from conexao import *

peca_repository = PecaRepository(session=session)
resultados = peca_repository.get_all()

for peca in resultados:
    print(peca.NOME)
```

Figura 2.14: Testando as classes Repositórios.

2.3.4 SCRIPT SQL - MySQL

```
-- -----------------------------------------------------
-- Schema teatro
-- -----------------------------------------------------
CREATE SCHEMA IF NOT EXISTS `teatro` DEFAULT CHARACTER SET utf8mb4 ;
USE `teatro` ;
-- -----------------------------------------------------
-- Table `teatro`.`peca`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `teatro`.`peca` (
  `CODPECA` INT(11) NOT NULL AUTO_INCREMENT,
  `NOME` VARCHAR(300) NULL DEFAULT NULL,
  PRIMARY KEY (`CODPECA`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4;
-- -----------------------------------------------------
-- Table `teatro`.`teatro`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `teatro`.`teatro` (
  `CODTEATRO` INT(11) NOT NULL AUTO_INCREMENT,
  `NOME` VARCHAR(300) NOT NULL DEFAULT '',
  `CAPACIDADEASSENTOS` INT NOT NULL,
  PRIMARY KEY (`CODTEATRO`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4;
-- -----------------------------------------------------
-- Table `teatro`.`apresentacao`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `teatro`.`apresentacao` (
  `CODAPRESENTACAO` INT(11) NOT NULL AUTO_INCREMENT,
  `CODTEATRO` INT(11) NOT NULL,
```

```
  `CODPECA` INT(11) NOT NULL,
  `QUANTIDADEPUBLICO` INT(11) NOT NULL DEFAULT 0,
  `DATA` DATE NOT NULL DEFAULT '0000-00-00',
  PRIMARY KEY (`CODAPRESENTACAO`),
  INDEX `FK_TEATRO_AP` (`CODTEATRO` ) ,
  INDEX `FK_PECA_AP` (`CODPECA` ) ,
  CONSTRAINT `FK_PECA_AP`
    FOREIGN KEY (`CODPECA`)
    REFERENCES `teatro`.`peca` (`CODPECA`),
  CONSTRAINT `FK_TEATRO_AP`
    FOREIGN KEY (`CODTEATRO`)
    REFERENCES `teatro`.`teatro` (`CODTEATRO`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4;
-- -----------------------------------------------------
-- Table `teatro`.`ator`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `teatro`.`ator` (
  `CODATOR` INT(11) NOT NULL AUTO_INCREMENT,
  `NOME` VARCHAR(300) NULL DEFAULT NULL,
  PRIMARY KEY (`CODATOR`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4;
-- -----------------------------------------------------
-- Table `teatro`.`atorpeca`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `teatro`.`atorpeca` (
  `CODPECA` INT(11) NOT NULL,
  `CODATOR` INT(11) NOT NULL,
  PRIMARY KEY (`CODPECA`, `CODATOR`),
  INDEX `FK_ATOR_ATORPECA` (`CODATOR` ) ,
  CONSTRAINT `FK_ATOR_ATORPECA`
```

```
    FOREIGN KEY (`CODATOR`)
    REFERENCES `teatro`.`ator` (`CODATOR`),
  CONSTRAINT `FK_PECA_ATORPECA`
    FOREIGN KEY (`CODPECA`)
    REFERENCES `teatro`.`peca` (`CODPECA`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4;

-- -----------------------------------------------------
-- Table `teatro`.`ingresso`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `teatro`.`ingresso` (
  `CODINGRESSO` INT(11) NOT NULL AUTO_INCREMENT,
  `CODAPRESENTACAO` INT(11) NOT NULL,
  `TIPOCADEIRA` VARCHAR(1) NOT NULL DEFAULT 'S',
  `DATA` DATE NOT NULL DEFAULT '0000-00-00',
  `PRECO` DOUBLE NOT NULL DEFAULT 0,
  PRIMARY KEY (`CODINGRESSO`),
  INDEX `FK_AP_INGRESSO` (`CODAPRESENTACAO` ) ,
  CONSTRAINT `FK_AP_INGRESSO`
    FOREIGN KEY (`CODAPRESENTACAO`)
    REFERENCES `teatro`.`apresentacao` (`CODAPRESENTACAO`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4;
```

II Parte Dois

3 QTDesigner e Pyside6 37

3.1 Instalando o PySide6

3. QTDesigner e Pyside6

Qt Designer

O Qt Designer é uma ferramenta de design visual fornecida como parte da estrutura Qt, que é uma das plataformas mais populares para o desenvolvimento de aplicativos multiplataforma. O Qt Designer permite criar interfaces gráficas de usuário (GUIs) de maneira eficiente e intuitiva, sem a necessidade de escrever código manualmente. Com ele, você pode criar interfaces complexas e interativas arrastando e soltando elementos da interface, definindo propriedades visuais e organizando a disposição dos widgets.

Principais recursos do Qt Designer:

- Interface Visual: O Qt Designer oferece uma interface visual onde você pode criar e personalizar widgets, layouts e janelas.
- Arrastar e Soltar: Os widgets podem ser arrastados e soltos no espaço de design, facilitando a criação da estrutura da GUI.
- Propriedades Personalizáveis: É possível ajustar as propriedades visuais e comportamentais dos widgets através de uma janela de propriedades.
- Suporte a Internacionalização: O Qt Designer permite a criação de interfaces que podem ser facilmente traduzidas para diferentes idiomas.
- Integração com Código: O design da interface pode ser salvo como um arquivo .ui, que pode ser integrado ao código Python usando uma biblioteca como o PySide6.

PySide6

PySide6 é uma biblioteca Python que fornece acesso ao conjunto de ferramentas Qt. Ela permite criar aplicativos de desktop ricos em recursos que podem ser executados em várias plataformas, como Windows, macOS e Linux. O PySide6 é uma das várias alternativas para acessar as funcionalidades do Qt a partir do Python.

Características do PySide6:

- Conectividade com o Qt: O PySide6 permite que você crie aplicativos com uma interface gráfica usando as classes e recursos do Qt.
- Sinais e Slots: Um dos conceitos mais poderosos do Qt é o sistema de sinais e slots, que facilita a comunicação entre diferentes partes do seu aplicativo.
- Licença de Código Aberto: PySide6 é distribuído sob a licença LGPL, permitindo seu uso em projetos comerciais e de código aberto.
- Documentação Rica: A documentação abrangente do Qt se aplica ao PySide6, tornando mais fácil aprender e usar a biblioteca.
- Compatibilidade com Qt Designer: O PySide6 pode carregar e interagir com arquivos .ui criados no Qt Designer, facilitando a criação de interfaces no Designer e sua utilização no código Python.

Em resumo, o Qt Designer é uma ferramenta visual para criar interfaces gráficas de usuário, enquanto o PySide6 é uma biblioteca Python que permite a criação de aplicativos interativos e multiplataforma, aproveitando as capacidades do Qt. Juntos, eles fornecem uma maneira eficaz de criar interfaces de usuário visualmente atraentes e funcionais para seus aplicativos Python.

3.1 Instalando o PySide6

```python
import sys
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton, QVBoxLayout, QWidge
from PySide6.QtUiTools import QUiLoader
from PySide6.QtCore import QFile, QIODevice, QObject, Slot

class MyMainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # Carregar o arquivo UI usando o QUiLoader
        loader = QUiLoader()
      ui_file_name = "mainwindow.ui"
        ui_file = QFile(ui_file_name)
        if not ui_file.open(QIODevice.ReadOnly):
            print(f"Cannot open {ui_file_name}: {ui_file.errorString()}")
            sys.exit(-1)
        ui_file.open(QFile.ReadOnly)
        self.ui = loader.load(ui_file, self)
        ui_file.close()

        # Conectar sinais aos slots
        self.ui.pushButton.clicked.connect(self.on_button_clicked)
        self.ui.lineEdit.textChanged.connect(self.on_text_changed)

        self.setCentralWidget(self.ui)

    @Slot()
    def on_button_clicked(self):
        self.ui.label.setText("Botão clicado!")

    @Slot(str)
    def on_text_changed(self, text):
        self.ui.label.setText(f"Texto alterado para: {text}")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MyMainWindow()
    window.show()
    sys.exit(app.exec_())
```

Figura 3.1: Hello World QTDesigner e PySide6