
alpha
<ed/tech>
BANCOS DE DADOS
</>

Banco de dados Relacional

PSYCOPG

Olá pessoal! Na aula de hoje iremos finalizar nossos estudos de banco de dados relacional com a seguinte pergunta: Como conectar meu projeto python ao banco de dados?

O **psycopg** é um dos módulos mais populares para se conectar a um banco de dados PostgreSQL usando a linguagem de programação Python. Ele permite que os desenvolvedores executem consultas SQL, manipulem dados e interajam com um banco de dados PostgreSQL de forma eficiente. O psycopg fornece uma API simples e poderosa para comunicação com o banco de dados, facilitando a execução de operações de leitura e escrita.

Instalação do psycopg

1. **Verificação dos pré-requisitos**
   Antes de instalar o psycopg, é importante garantir que o ambiente Python já esteja instalado e configurado corretamente. Além disso, é necessário ter o PostgreSQL instalado no sistema.

2. **Instalação do psycopg via pip**
   O método mais comum para instalar o psycopg é através do gerenciador de pacotes Python, o pip. Para isso, basta abrir o terminal ou prompt de comando e executar o seguinte comando:

```
pip install psycopg2
```

   Este comando irá baixar e instalar o psycopg no ambiente Python.

3. **Teste de conexão**
   Após a instalação do psycopg, é recomendável testar a conexão com um banco de dados PostgreSQL para garantir que a instalação foi bem-sucedida. Um exemplo de código para testar a conexão pode ser:

```
import psycopg2

# Conectando ao banco de dados
conn = psycopg2.connect(
    dbname="nome_do_banco",
    user="usuario",
    password="senha",
    host="localhost"
)

# Criando um cursor
cur = conn.cursor()
# . . .
```

Banco de dados Relacional

PSYCOPG

Instalação do psycopg

```
# Executando uma consulta de teste
cur.execute("SELECT version()")

# Obtendo o resultado da consulta
print(cur.fetchone())

# Fechando o cursor e a conexão
cur.close()
conn.close()
```

Ao executar o código, se a conexão for bem-sucedida, o resultado da consulta deve ser exibido no console.

Considerações finais:

É importante lembrar de sempre verificar a compatibilidade da versão do psycopg com a versão do Python e do PostgreSQL em uso. Além disso, a documentação oficial do psycopg pode fornecer informações mais detalhadas sobre a instalação e uso do adaptador.

Boas práticas

1. Utilização segura de consultas SQL:
   - Uso de parâmetros nas consultas para prevenir injeção de SQL.
   - Evitar a concatenação de strings para formar consultas.
   - Utilização de placeholders para dados dinâmicos.

2. Gerenciamento de conexões e cursores:
   - Abertura e fechamento adequados de conexões e cursores.
   - Uso de `with` para garantir o fechamento automático de recursos.
   - Pooling de conexões para melhorar o desempenho.

3. Manipulação de erros e exceções:
   - Tratamento adequado de erros ao executar consultas.
   - Utilização de blocos `try-except` para capturar e lidar com exceções.
   - Log de erros para rastrear problemas no código.

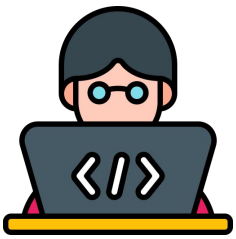
4. Segurança de dados:
   - Criptografia de dados sensíveis antes de armazená-los no banco.
   - Validação de dados de entrada para prevenir ataques.

5. Otimização de consultas:
   - Uso de índices adequados nas consultas para melhorar a performance.
   - Limitação de resultados em consultas para reduzir o tráfego de dados.

Ao seguir essas boas práticas, os desenvolvedores podem garantir que suas aplicações Python que utilizam o psycopg sejam eficientes, seguras e confiáveis no acesso a bancos de dados PostgreSQL.

Banco de dados Relacional

PSYCOPG



Conexão ao Banco de Dados

Estabelecer um ponto de conexão entre Python e PostgreSQL inclui medidas como a definição dos parâmetros de conexão, a abertura de conexão, dentre outros. Abaixo temos um exemplo de como isso pode ser feito:

```
import psycopg2

try:
  connection = psycopg2.connect(
    database = "nome_do_banco",
    user = "nome de usuario",
    password = "senha",
    host = "host",
    port = "porta"
  )

  print("Conexão estabelecida com sucesso!")

except Exception as e:
  print(f"Ocorreu um erro: {e}")
```

Em uma cadeia de conexão SQL típica, "database" é o nome do seu banco de dados, "user" é o nome do usuário pelo qual você está conectando, "password" é a senha para esse usuário, "host" é onde o servidor está localizado (pode ser "localhost") e "port" é a porta na qual o servidor está escutando (geralmente "5432" para PostgreSQL).

Conexão Segura

Para assegurar uma comunicação eficiente e segura, é altamente recomendável usar as diretrizes de melhor prática do psycopg, tais como o uso de declarações preparadas e o manejo adequado de erros e exceções.

Além disso, para a segurança, é sempre melhor não incluir a senha diretamente no script, mas sim referenciá-la de uma variável de ambiente ou de um arquivo de configuração fora do seu código.

Banco de dados Relacional

PSYCOPG

Consultas com psycopg

Segurança de Consultas

Trabalhar com consultas SQL pode abrir vulnerabilidades de segurança, como os ataques de injeção de SQL, onde um agente mal-intencionado pode inserir comandos SQL nocivos. Consultas parametrizadas são uma forma de proteger seu código contra essas vulnerabilidades.
Ao usar consultas parametrizadas, os parâmetros são separados da consulta SQL em si e inseridos posteriormente, o que impede a injeção de SQL.

Consultas Parametrizadas

Usar consultas parametrizadas não é apenas uma prática de segurança, mas também uma prática recomendada para limpeza de código e eficiência. Em vez de inserir os valores diretamente na string da consulta SQL, os parâmetros são usados, geralmente na forma de uma tupla ou dicionário. A string da consulta, em seguida, tem espaços reservados (tipicamente %s) onde os valores devem ser inseridos. Veja este exemplo:

```
cursor.execute('SELECT * FROM tabela WHERE coluna = %s', (valor,))
```

Trabalhando com Resultados de Consultas

Depois de executar uma consulta, você irá manipular os resultados. Esses resultados são retornados como uma lista de tuplas, onde cada tupla corresponde a uma linha da tabela. Você pode percorrer essas linhas e acessar os valores individuais.

As funções fetchone() e fetchall() são comumente usadas para recuperar os resultados. fetchone() recupera a próxima linha de um conjunto de resultados da consulta, enquanto fetchall() recupera todas as linhas.

Aqui está um exemplo de como recuperar e manipular os resultados de uma consulta:

```
cursor.execute('SELECT * FROM tabela')
resultados = cursor.fetchall()
for linha in resultados:
    print(linha)
```

Fechando a Conexão

Lembre-se sempre de fechar a conexão quando terminar de executar as consultas. Isso pode ser feito usando o método close() na conexão. Caso contrário, você poderá ter uma conexão de banco de dados aberta desnecessariamente, o que pode causar problemas de desempenho.

```
conexão.close()
```

Banco de dados Relacional

PSYCOPG

Manipulação de dados com psycopg

Agora vamos aprender como inserir novos dados, atualizar registros existentes e excluir informações de tabelas PostgreSQL usando psycopg. Discutiremos boas práticas ao lidar com modificações de dados.

Uma vez conectados com êxito ao banco de dados, podemos executar consultas SQL. Para isso, é necessário criar um objeto cursor que permitirá a execução das consultas. Vejamos como uma consulta SQL básica pode ser executada:

```
cur = con.cursor()
cur.execute("SELECT * FROM nome_da_tabela")
rows = cur.fetchall()
for row in rows:
    print(row)
```

Para proteger suas consultas SQL de ataques de injeção SQL, é recomendável usar consultas parametrizadas. Uma consulta parametrizada garante que os parâmetros de entrada são tratados como literal string ao invés de parte do comando SQL, dificultando a execução de comandos SQL mal-intencionados.

Consultas Parametrizadas: Aqui está como uma consulta parametrizada pode parecer:

```
cur = con.cursor()
postgreSQL_select_Query = "SELECT * FROM nome_da_tabela WHERE
id = %s"
record_to_select = (3, )
cur.execute(postgreSQL_select_Query, record_to_select)
```

Os parâmetros são passados como uma tupla, mesmo que seja apenas um parâmetro.

Lidando com Resultados de Consultas: fetchone(), fetchall() e fetchmany()

Três métodos são disponibilizados pelo psycopg para buscar resultados de consultas: `*fetchone()*`, `*fetchall()*` e `*fetchmany()*`.

- `**fetchone()**` pega o próximo registro na fila de resultados do banco de dados.
- `**fetchall()**` pega todos os registros na fila de resultados.
- `**fetchmany([número_de_linhas])**` pega determinado número de linhas.

Banco de dados Relacional

PSYCOPG

Tratamento de Erros e Exceções

Para iniciar, deve-se entender o que são erros e exceções. Erros são problemas que ocorrem durante a execução de um programa, enquanto exceções são circunstâncias não comuns que podem modificar o fluxo normal do programa. Ambos podem acarretar em problemas, desde a quebra do programa até seus resultados serem comprometidos.

Psycopg e Tratamento de Exceções

Psycopg é eficiente e capaz de lidar com muitos dos desafios que surgem ao trabalhar com um banco de dados. No entanto, como qualquer outra ferramenta, ele não está imune a erros e exceções.

O psycopg fornece seu próprio conjunto de exceções que podem ser usadas para lidar com erros específicos do banco de dados. Essas exceções herdam da classe base `psycopg2.Error`, que por sua vez herda da classe base `Exception` do Python.

Por exemplo, podemos usar o tratamento de exceções do psycopg para lidar com erros de conexão com o banco de dados. Veja o exemplo abaixo:

```
import psycopg2

    try:
        con = psycopg2.connect(database="mydb", user="myuser",
password="mypassword", host="127.0.0.1", port="5432")
        print("Database opened successfully")

    except psycopg2.DatabaseError as e:
        print("Error %s" % e)
```

Neste exemplo, tentamos abrir uma conexão com o banco de dados. Se a conexão falhar, o psycopg levanta uma exceção do tipo `DatabaseError`, que então capturamos e manipulamos.

Além do `DatabaseError`, o psycopg fornece várias outras exceções, como `UniqueViolation`, `ForeignKeyViolation`, `NotNullViolation`, entre outros, cada uma tratando de um tipo de erro específico que pode ocorrer ao interagir com o banco de dados.

Banco de dados Relacional

PSYCOPG

Transações

Transações são um conjunto de operações executadas em um banco de dados consideradas uma unidade de trabalho.

As transações são de suma importância para garantir a segurança dos dados. Ajudam na manutenção de um sistema acurado ao assegurar que, se alguma parte da transação falha, todo o sistema de transações não será comprometido. Além disso, assegura que todas as transações sejam concluídas com sucesso, mantendo a integridade do banco de dados.

Para se trabalhar com transações no psycopg, precisa-se iniciar a transação através do comando "BEGIN", executar os comandos necessários e fazer commit da transação através do comando "COMMIT".

Em caso de erro, pode-se usar o comando "ROLLBACK" para desfazer todas as modificações feitas na transação atual.

```
conn = psycopg2.connect(...)

try:
    # Inicia a transação
    conn.autocommit = False

    # Executar comandos
    ...

    # Faz o commit
    conn.commit()
except:
    # Se ocorrer um erro, realiza um rollback
    conn.rollback()
```

A consistência dos dados é garantida ao se usar transações, pois uma vez que a transação tenha sido executada, os efeitos da transação são permanentemente no banco de dados. No entanto, é essencial garantir que todas as transações sejam concluídas com sucesso para manter a integridade do sistema. No psycopg, isso pode ser assegurado ao fazer commit da transação depois que todas as operações necessárias tenham sido concluídas com sucesso. Caso a transação seja abortada devido a um erro, todas as modificações são revertidas, garantindo assim que o banco de dados permaneça consistente.

Banco de dados Relacional

PSYCOPG

Fechamento de Conexão

Por que é tão importante garantir que uma conexão com um banco de dados seja fechada corretamente? Essencialmente, cada conexão aberta consome recursos do sistema. Portanto, se mantivermos conexões desnecessariamente abertas, podemos acabar desperdiçando recursos preciosos que poderiam ter sido utilizados para melhorar a performance da nossa aplicação. Além disso, conexões abertas que não são mais necessárias podem levar a vazamentos de recursos, onde os recursos do sistema são reservados mas não liberados corretamente.

Agora que entendemos a importância de fechar as conexões com o banco de dados, vamos discutir como fazer isso corretamente usando o psycopg.

Um método confiável é usar o método `close()` do objeto de conexão. Isso fechará a conexão com o banco de dados:

```
conn = psycopg2.connect("dbname=test user=postgres")
# executar consultas ao banco de dados aqui...
conn.close()
```

No entanto, é importante notar que este método não será executado se ocorrer um erro em algum lugar no seu código entre a abertura e o fechamento da conexão. Para lidar com isso, podemos usar a instrução `with` do Python, que garantirá que a conexão seja fechada mesmo se um erro ocorrer:

```
with psycopg2.connect("dbname=test user=postgres") as conn:
    # executar consultas ao banco de dados aqui...
```

Com a cláusula `with`, o Python automaticamente chama `conn.close()` quando o bloco de código dentro do `with` é terminado, mesmo se há um erro.

A falta de encerramento adequado da conexão pode levar a diversos problemas, tais como performance degrada, vazamento de recursos e até mesmo falhas no sistema.

Certificando-se de que você está gerenciando e fechando corretamente suas conexões, você consegue manter a saúde e a performance da sua aplicação.

Obrigado e até a próxima aula!