



Curso Programação Backend com Python

Banco de dados

© Cleuton Sampaio 2021

Sumário

Bancos de dados para backend	3
PostgreSQL remoto com ElephantSQL	4
Conexão com o PostgreSQL	8
Inserção, atualização e eliminação de registros	13

Bancos de dados para backend

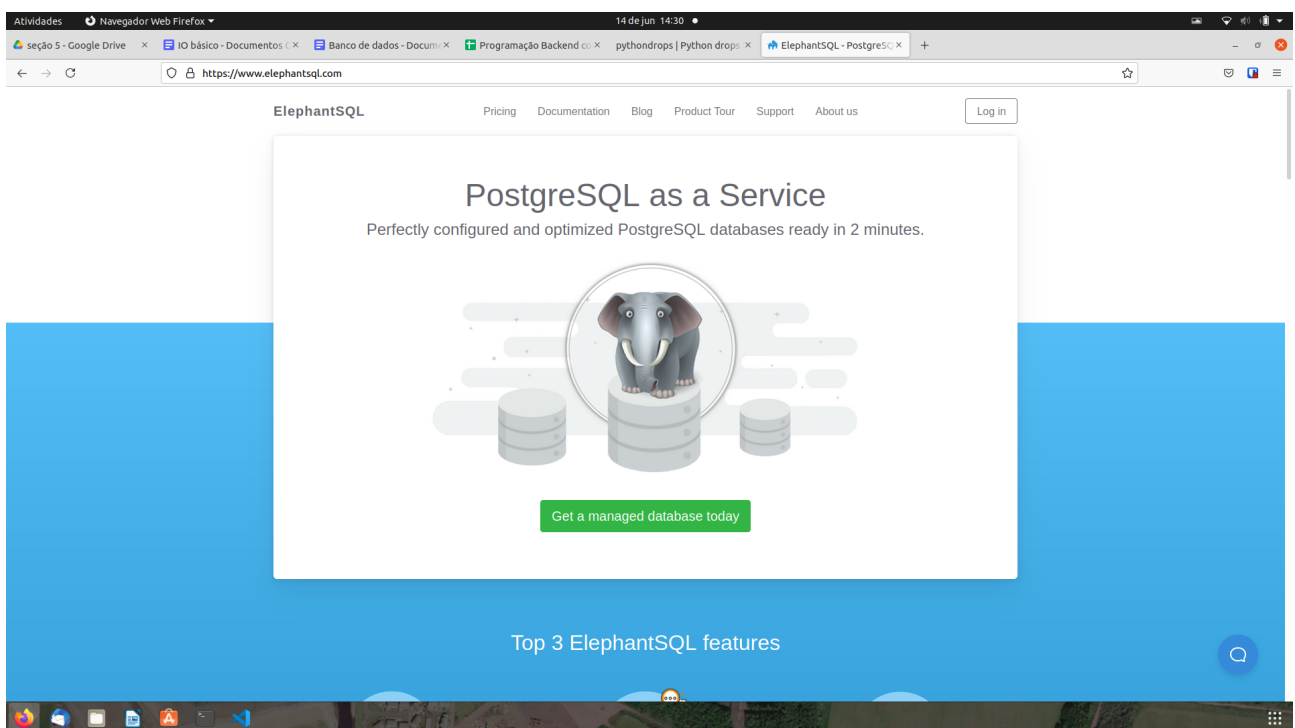
Sem dúvida alguma, a programação backend pressupõe o conhecimento de banco de dados para a maioria das aplicações servidoras.

Os bancos de dados podem ser Relacionais, como o **PostgreSQL** ou o **Oracle**, ou No-SQL, como o **MongoDB**.

Para estudar um SGBD (Sistema Gerenciador de Banco de Dados) proprietário, como o Oracle, teríamos que obter uma licença, o que pode dificultar as coisas.

E também quero evitar ao máximo possível que você tenha que instalar coisas em sua máquina sem necessidade. Então, para estudar SGBD relacionais vamos usar o PostgreSQL, em um serviço de nuvem: **ElephantSQL**:

<https://www.elephantsql.com/>



É um serviço em nuvem que permite criar e acessar banco de dados PostgreSQL remotamente e gratuitamente. É claro que há limitações, mas, para aprendizado, não representam problemas.

Se você preferir, pode baixar e instalar o PostgreSQL em sua máquina:

<https://www.postgresql.org/download/>

Outra opção popular é utilizar uma imagem **Docker** (caso tenha o Docker instalado e saiba utilizá-lo):

```
docker run --name some-postgres -e
POSTGRES_PASSWORD=mysecretpassword -d postgres
```

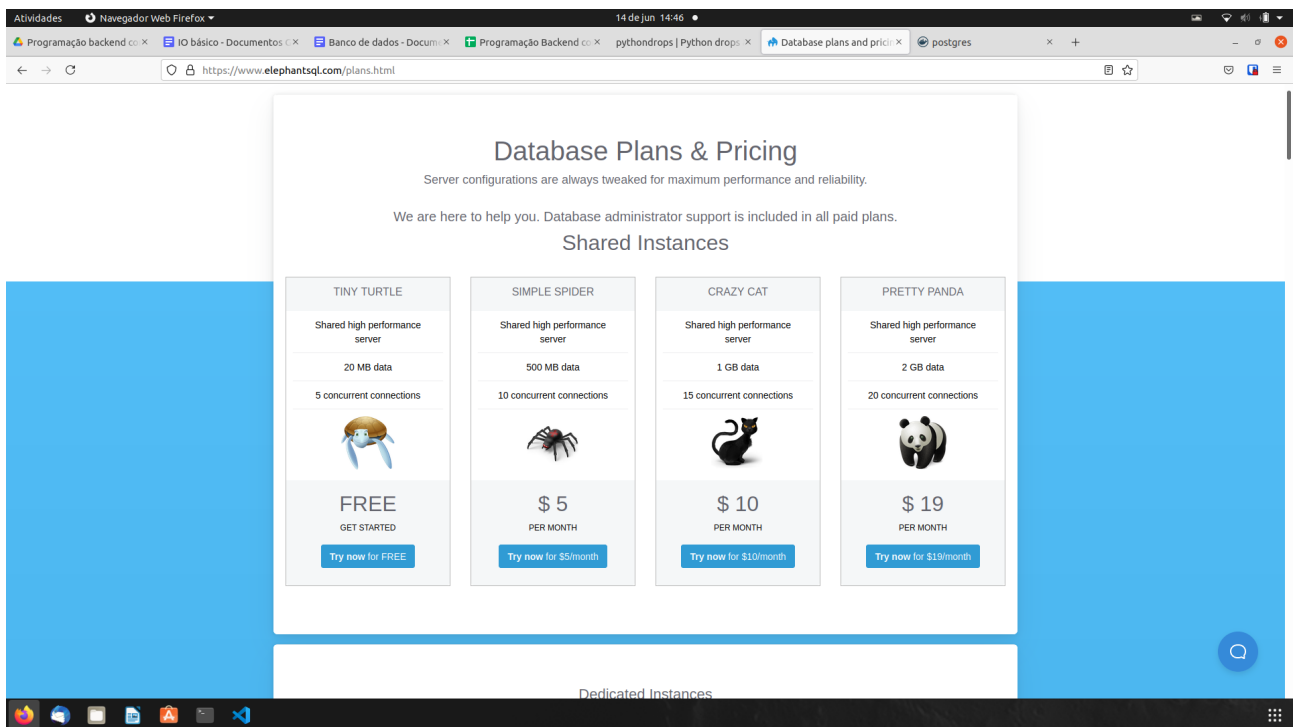
Mas, para este curso, vamos utilizar um banco de dados em nuvem e não haverá necessidade de instalar coisa alguma.

Você conhece SQL e Bancos de dados, certo?

Não é objetivo do curso ensinar bancos de dados relacionais e linguagem SQL. Há muitos cursos no Mercado que podem atender à sua necessidade.

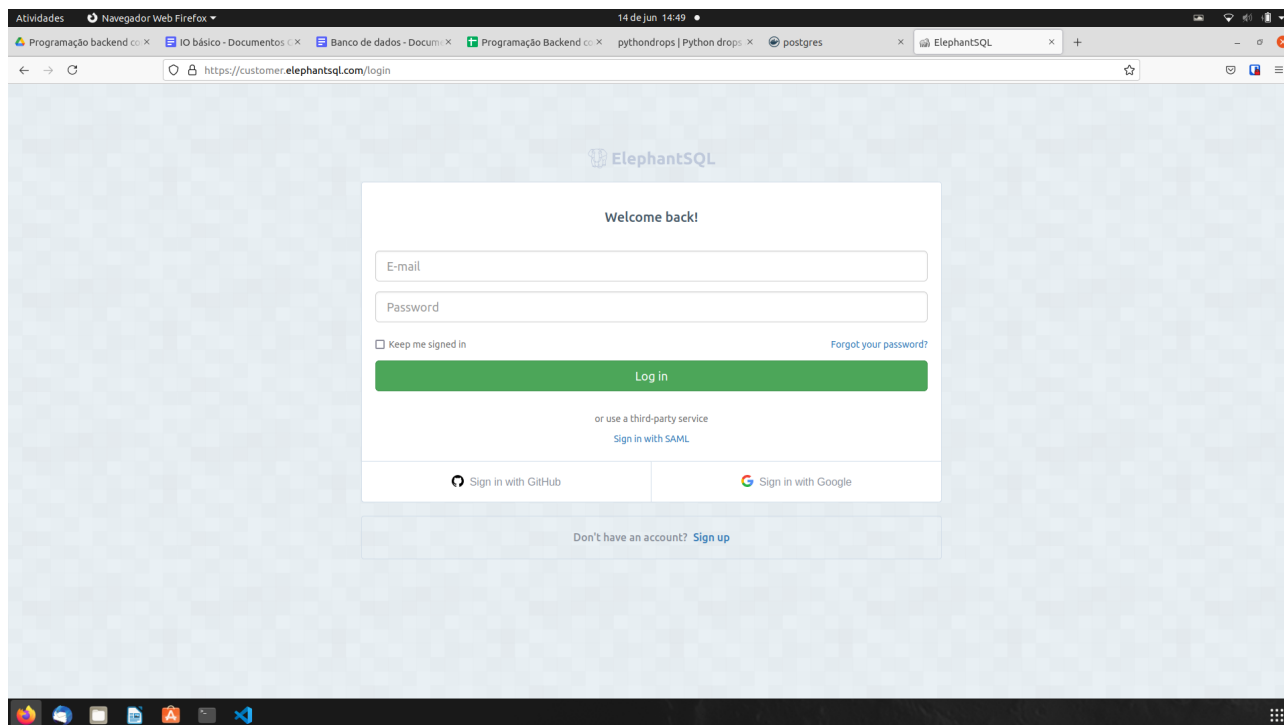
PostgreSQL remoto com ElephantSQL

Ao acessar o ElephantSQL pela primeira vez, você deve criar uma conta. Ele possui vários planos de uso, que vão desde o gratuito (Tiny turtle) até o avançado, incluindo suporte comercial:



Escolha o **TINY TURTLE** que é gratuito!

Você pode clicar no botão LOGIN e depois no SIGN UP:

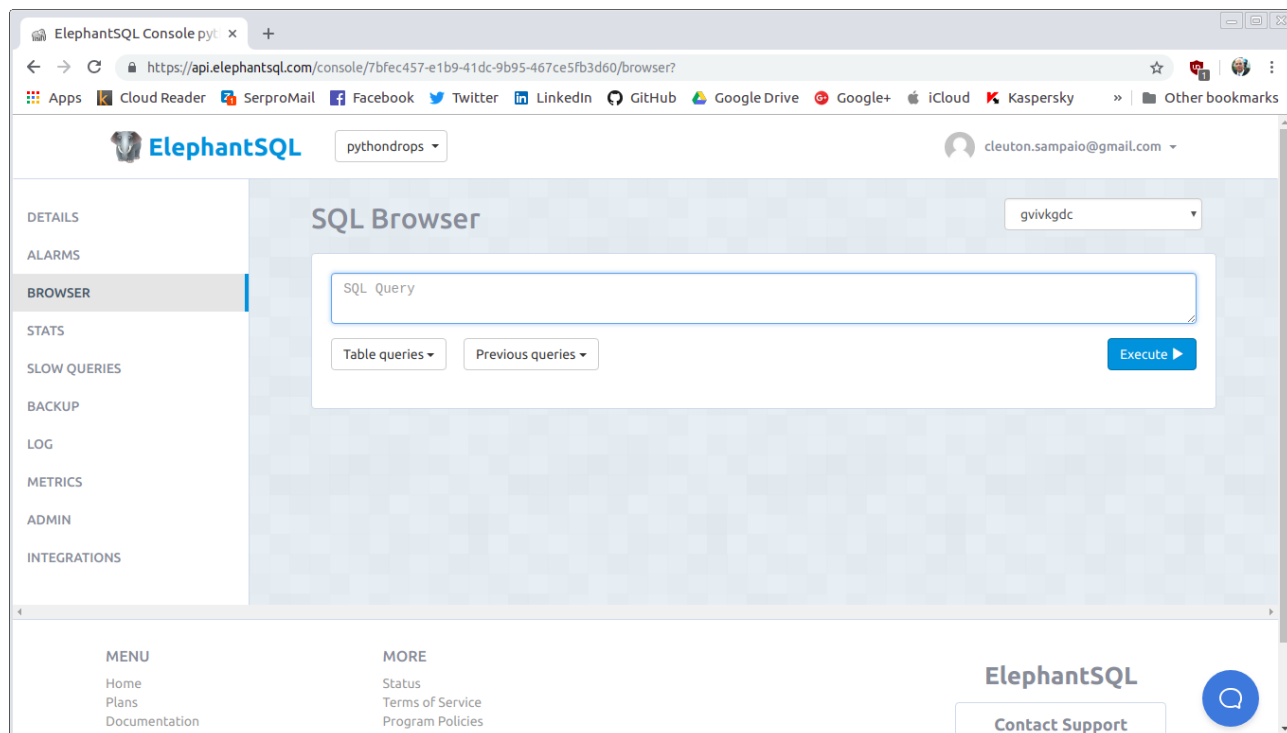


E você poderá fazer LOG IN com sua conta Google ou Github (se tiver uma).

Após criar sua conta e fazer LOG IN, crie um banco de dados. Os passos são simples:

1. Crie uma instância (use a conta gratuita do plano "Tiny turtle");
2. Selecione sua instância clicando sobre o nome que informou;
3. Use a opção BROWSER e Digite sua consulta SQL e execute.

Você deve criar pelo menos uma tabela.



Para criar uma tabela simples, execute um CREATE TABLE:

```
CREATE TABLE account(  
  user_id serial PRIMARY KEY,  
  username VARCHAR (50) UNIQUE NOT NULL,  
  password VARCHAR (50) NOT NULL,  
  email VARCHAR (355) UNIQUE NOT NULL,  
  created_on TIMESTAMP NOT NULL,  
  last_login TIMESTAMP  
);
```

A chave primária é "user_id".

Se você não conhece SQL e nem PostgreSQL, então é melhor dar uma olhada neste tutorial:

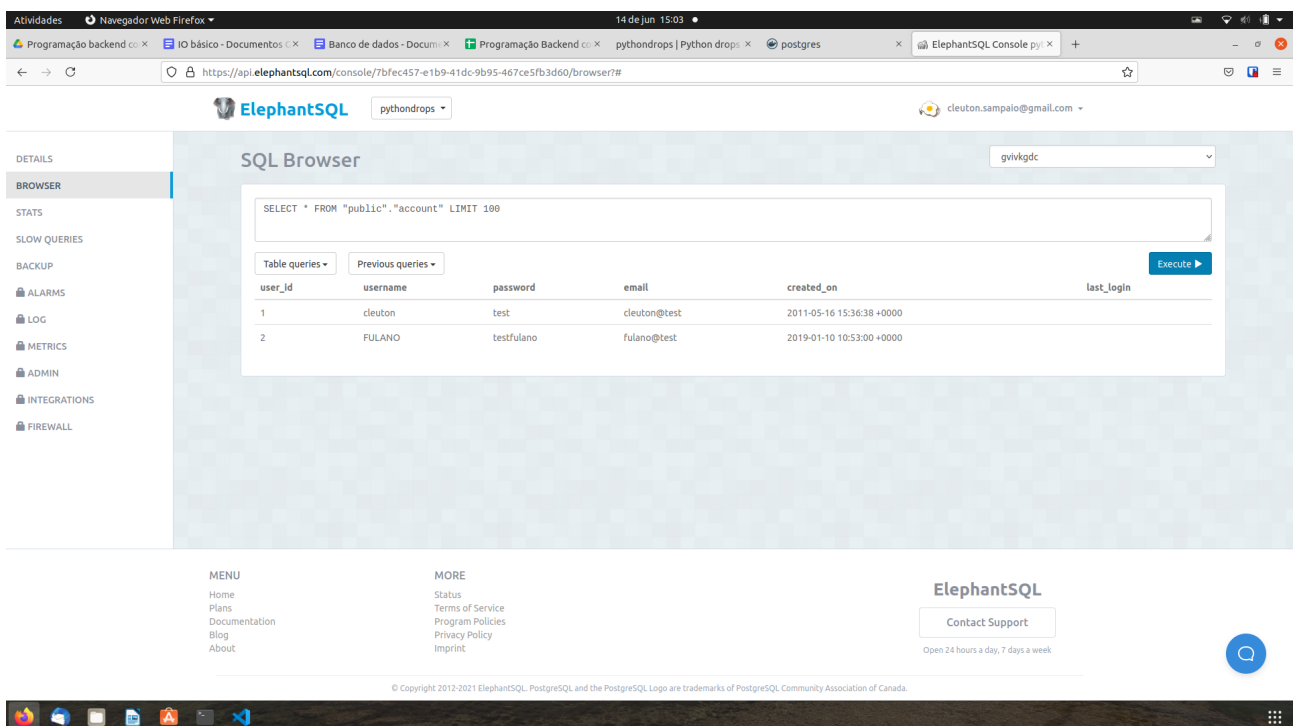
<http://www.postgresqltutorial.com/postgresql-create-table/>

Depois, insira alguns dados, por exemplo:

```
insert into account
(user_id,username,password,email,created_on,last_login)
values(2, 'FULANO', 'testfulano', 'fulano@test',
      '2019-01-10 10:53:00', NULL)
```

Ele converte o string para timestamp automaticamente.

Podemos rodar uma consulta SQL para ver os dados da tabela:



The screenshot shows the ElephantSQL web interface in a Firefox browser. The URL is <https://api.elephantsql.com/console/7bfec457-e1b9-41dc-9b95-467ce5fb3d60/browser?#>. The interface includes a sidebar with navigation options like DETAILS, BROWSER, STATS, SLOW QUERIES, BACKUP, ALARMS, LOG, METRICS, ADMIN, INTEGRATIONS, and FIREWALL. The main area is titled 'SQL Browser' and shows a query: `SELECT * FROM "public"."account" LIMIT 100`. Below the query, there is a table with the following data:

user_id	username	password	email	created_on	last_login
1	cleuton	test	cleuton@test	2011-05-16 15:36:38 +0000	
2	FULANO	testfulano	fulano@test	2019-01-10 10:53:00 +0000	

The interface also includes a footer with a menu, more links, and the ElephantSQL logo.

Para trabalhar com PostgreSQL em python é necessário instalar uma dependência, que é o **psycopg2**:

```
pip install psycopg2-binary → Não instale sem um ambiente virtual!!!!
```

Como eu disse na seção 3, recomendo que você sempre crie um ambiente virtual com o VENV ao criar um novo projeto. Isso evita que você “bagunce” o python básico da sua máquina e permite instalar dependências diferentes para cada projeto.

Antes de instalar o psycopg2, vamos lembrar os passos e criar um ambiente virtual:

1. Crie uma pasta para o seu projeto (para este que vou mostrar agora);
2. Nessa pasta, crie um ambiente virtual python com o venv:

```
python3 -m venv <pasta onde quer criar o ambiente python>
```
3. Crie um arquivo com as dependências do seu projeto, chamado "requirements.txt" e escreva "psycopg2-binary" nele;
4. Rode o comando: `pip3 install -r requirements.txt`

Conexão com o PostgreSQL

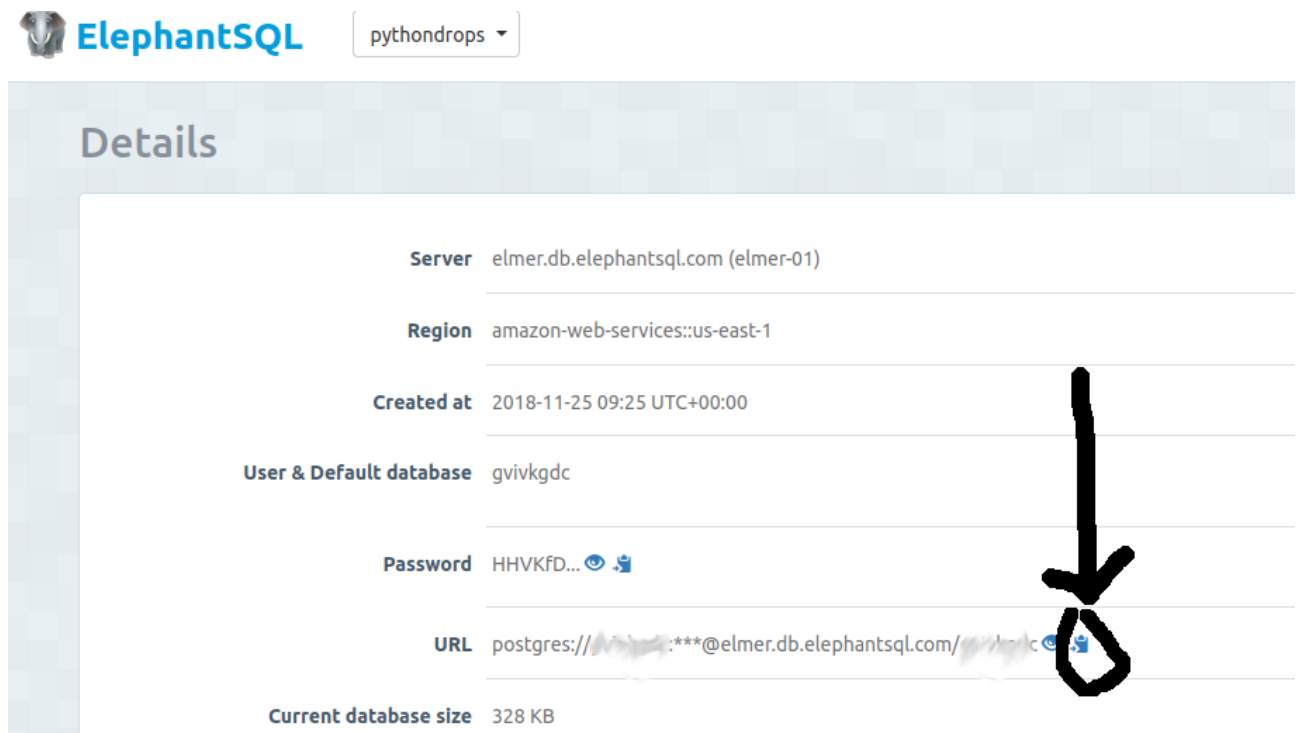
Todos os exemplos estão no arquivo "exemplos_bd.zip".

Abra um editor e crie um arquivo python. Vamos começar importando as bibliotecas necessárias:

```
import psycopg2
import urllib.parse as up
```






A primeira dependência é a biblioteca para acessar o PostgreSQL, e a segunda é um parser (analisador) de URL que vamos utilizar para conectar com o banco no ElephantSQL. Esta é uma necessidade do ElephantSQL e não do PostgreSQL. Ele nos fornece uma URL de acesso e precisamos saber os dados separadamente.

Você pode copiar a sua URL de acesso clicando em DETAILS e no botão com a prancheta ao lado da URL, como nessa figura:



ElephantSQL pythondrops ▾

Details

Server	elmer.db.elephantsql.com (elmer-01)
Region	amazon-web-services::us-east-1
Created at	2018-11-25 09:25 UTC+00:00
User & Default database	gvivkgdc
Password	HHVKfD...  
URL	postgres://  :***@elmer.db.elephantsql.com/  
Current database size	328 KB

Agora, vamos montar a string de conexão com o servidor e o banco de dados PostgreSQL. Com a URL que você copiou:

```
url = up.urlparse("postgres://...") # cole a sua URL Aqui
```

Nota: A URL tem seu usuário e a senha! Cuidado!

Agora é só conectar com o banco:

```
db = psycopg2.connect(database=url.path[1:],  
                        user=url.username,  
                        password=url.password,  
                        host=url.hostname,  
                        port=url.port  
)
```

Navegação por cursor

A coisa mais básica que vamos fazer é navegar por uma tabela (ou consulta SQL) utilizando um cursor. Vamos acessar a tabela “account” que criamos anteriormente:

```
cursor = db.cursor()
cursor.execute('select * from account')
registros = cursor.fetchall()
for registro in registros:
    print (registro)
db.close()
```

O cursor é o mecanismo para navegarmos pelos resultados de uma consulta SQL. O método “execute” permite executar consultas ou executar atualizações. Depois de uma consulta, o método “execute” preenche os resultados que podem ser acessados através do método “fetchall()”, por exemplo. Depois, podemos navegar pelos registros.

Veja o código completo aqui:

```
import psycopg2
import urllib.parse as up

url = up.urlparse("postgres://...")

db = psycopg2.connect(database=url.path[1:],
    user=url.username,
    password=url.password,
    host=url.hostname,
    port=url.port
)

cursor = db.cursor()
cursor.execute('select * from account')
```

```
registros = cursor.fetchall()
for registro in registros:
    print (registro)
db.close()
```

```
db.close()
```

O resultado seria algo assim:

```
python3 acessa_account.py
(1, 'cleuton', 'test', 'cleuton@test', datetime.datetime(2011, 5, 16, 15, 36, 38), None)
(2, 'FULANO', 'testfulano', 'fulano@test', datetime.datetime(2019, 1, 10, 10, 53), None)
(3, 'Beltrano', 'bbb', 'beltrano@test', datetime.datetime(2021, 6, 14, 16, 51, 8), None)
```

O método “fetchall” do cursor retorna todos os registros lidos como uma lista de tuplas, e podemos acessar cada elemento individualmente como neste outro exemplo (sql2.py):

```
cursor.execute('select * from account')
registros = cursor.fetchall()
for registro in registros:
    print('Usuário: {} email: {} data criação: {}'.format(registro[0],registro[3],registro[4]))
```

Aqui vale um registro interessante. Ao utilizar o método “fetchall” TODOS os registros foram transferidos como uma lista de tuplas! Isso pode ser válido quando temos poucos registros, mas e quando temos muitos?

E como tratamos os erros?

O exemplo abaixo é um pouco melhor (sql5.py):

```
import psycopg2
import urllib.parse as up
url = up.urlparse("postgres://...")

try:
    db = psycopg2.connect(database=url.path[1:],
        user=url.username,
        password=url.password,
        host=url.hostname,
        port=url.port
    )
    cursor = db.cursor()
    cursor.execute('select * from account')
    for registro in cursor:
        print('Usuário: {} email: {} data criação:
        {}'.format(registro[0],registro[3],registro[4]))
except (Exception, psycopg2.DatabaseError) as error:
    print(error)
finally:
    db.close()
```

O resultado deste código é:

```
python3 sql5.py
Usuário: 1 email: cleuton@test data criação: 2011-05-16 15:36:38
Usuário: 2 email: fulano@test data criação: 2019-01-10 10:53:00
Usuário: 3 email: beltrano@test data criação: 2021-06-14 17:04:51
```

Neste exemplo, estou usando o próprio objeto **cursor** como iterável, pegando cada registro do banco conforme necessito. É como se eu tivesse utilizado o método “fetchone” que traz um registro por vez.

Inserção, atualização e eliminação de registros

Podemos criar comandos de atualização do banco e executá-los. Vejamos este exemplo (sql3.py):

```
import psycopg2
import urllib.parse as up
import datetime

data = datetime.datetime.now()

url =
up.urlparse("postgres://gvivkgdc:HHVKfD57JlGKwAxUt9ni-WfXDr9jkOWh@elmer.db.elephan
tsql.com/gvivkgdc")

db = psycopg2.connect(database=url.path[1:],
                        user=url.username,
                        password=url.password,
                        host=url.hostname,
                        port=url.port
)

cursor = db.cursor()
cursor.execute(
    """INSERT INTO account
    (user_id,username,password,email,created_on,last_login)
VALUES (%s, %s, %s, %s, %s, %s)""",
    (3, 'Beltrano', 'bbb', 'beltrano@test',
    data.strftime('%Y-%m-%d %H:%M:%S'), None))

db.commit()
db.close()
```

Ele executa um comando SQL INSERT na tabela account. Note que utilizei o string de múltiplas linhas (" " ") para especificar o comando SQL. Este é um recurso interessante pois evita problemas na hora de postar um código-fonte (quebras estranhas de linha).

O método “execute” para atualizações pode receber 2 argumentos: O primeiro é o string e o segundo são os valores, na forma de tupla.

E se eu quiser remover? Se você conhece SQL, sabe que é o comando DELETE:

```
cursor = db.cursor()
cursor.execute(
    "delete from account where user_id=3"
)
```

Finalmente, temos a atualização, ou UPDATE. Vamos mudar o nome do usuário que acabamos de inserir (sql4.py):

```
cursor = db.cursor()
sql = """update account set username='Beltrano de Tal'
        where user_id=%s"""
cursor.execute(sql, (3,))
```

Tanto no INSERT como no UPDATE, nós podemos passar o comando em string, utilizando marcadores **%s** onde queremos os valores dos campos, e depois passar uma tupla com os valores em si. Preste atenção na tupla que eu passei: **(3,)**! Por que? Como ela só tem um valor, é preciso acrescentar uma vírgula, caso contrário o método pensará que é apenas um inteiro e não uma tupla.