INSTITUTO FEDERAL
Paraíba
Campus João Pessoa

Aula
8

Banco de Dados II

Atualizada em 11/11/2019

Conectividade com Banco de Dados

Professor:
Dr. Alex Sandro da Cunha Rêgo

alex@ifpb.edu.br

Introdução

- O pacote **psycopg2** de Python
  - Um adaptador escrito na linguagem python, para conexão com o banco de dados **PostgreSQL**
  - Projetado para ser simples, pequeno e rápido na comunicação com o BD
- Instalando a **API-Python psycopg2**
  - Prompt de commando do Windows com PIP

```
c:\> pip install psycopg2
```

  - Documentação official

http://initd.org/psycopg/docs/install.html

  - Há outras opções de adaptadores disponíveis

Introdução

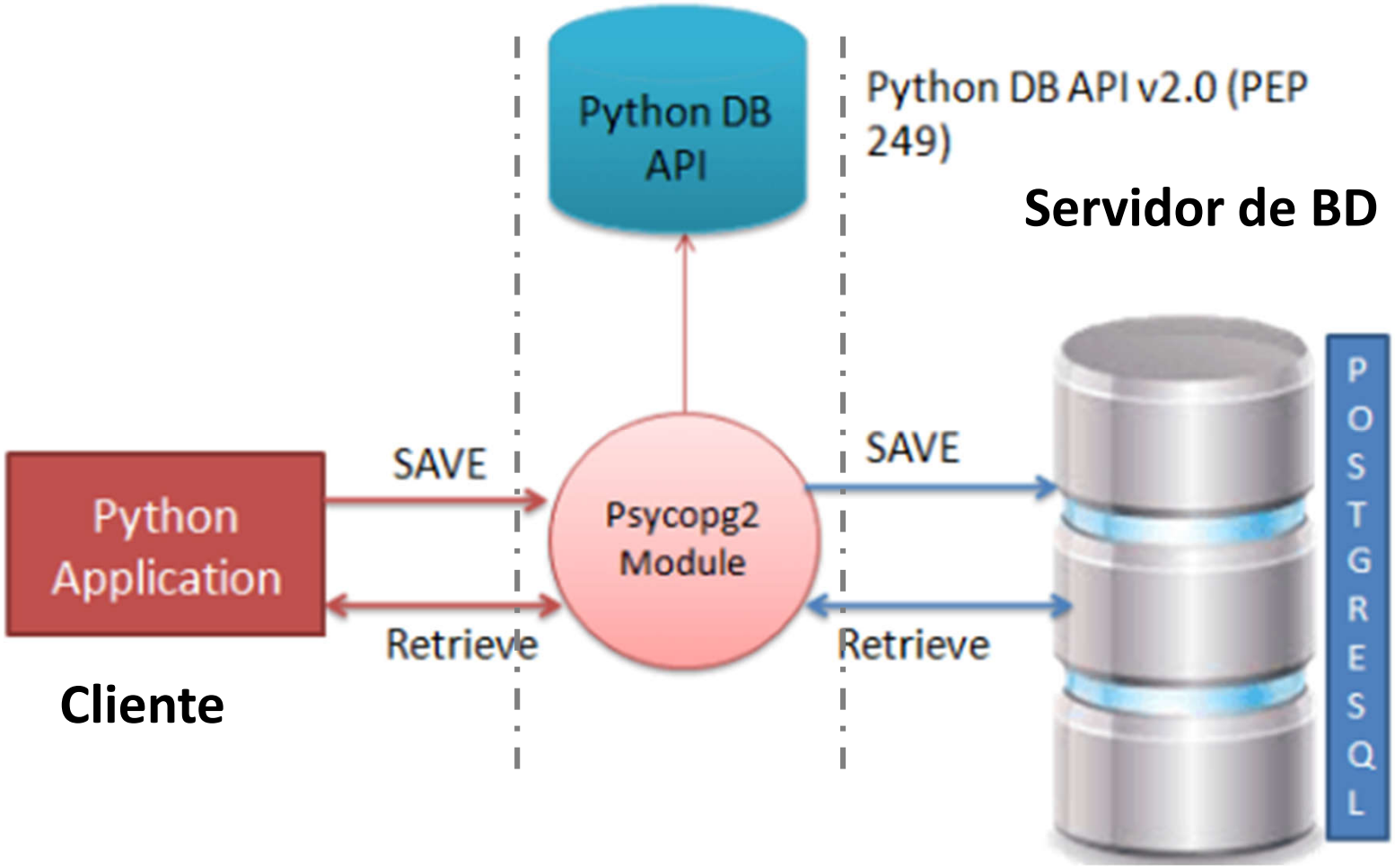
- Benefícios da **API-Python**
  - A própria linguagem Python é considerada simples e eficiente quando comparada a outras linguagens
  - Independência de plataforma
    - ✓ Não é mais necessário escrever um programa de conexão com o banco para cada plataforma
  - Portabilidade
    - ✓ O mesmo código pode ser utilizado para se conectar a diferentes BD's instalados na mesma máquina
  - Permite realizar instruções SQL, DML (update, insert e delete) e DDL (create table)

Introdução

- Benefícios da API-Python
  - Permite a chamada de stored procedures
  - Ativação de transações via linhas de código
  - Permite a manipulação de campos BLOB
  - Suporta a implementação de aplicações *multi-thread* (thread podem compartilhar conexões)

Arquitetura

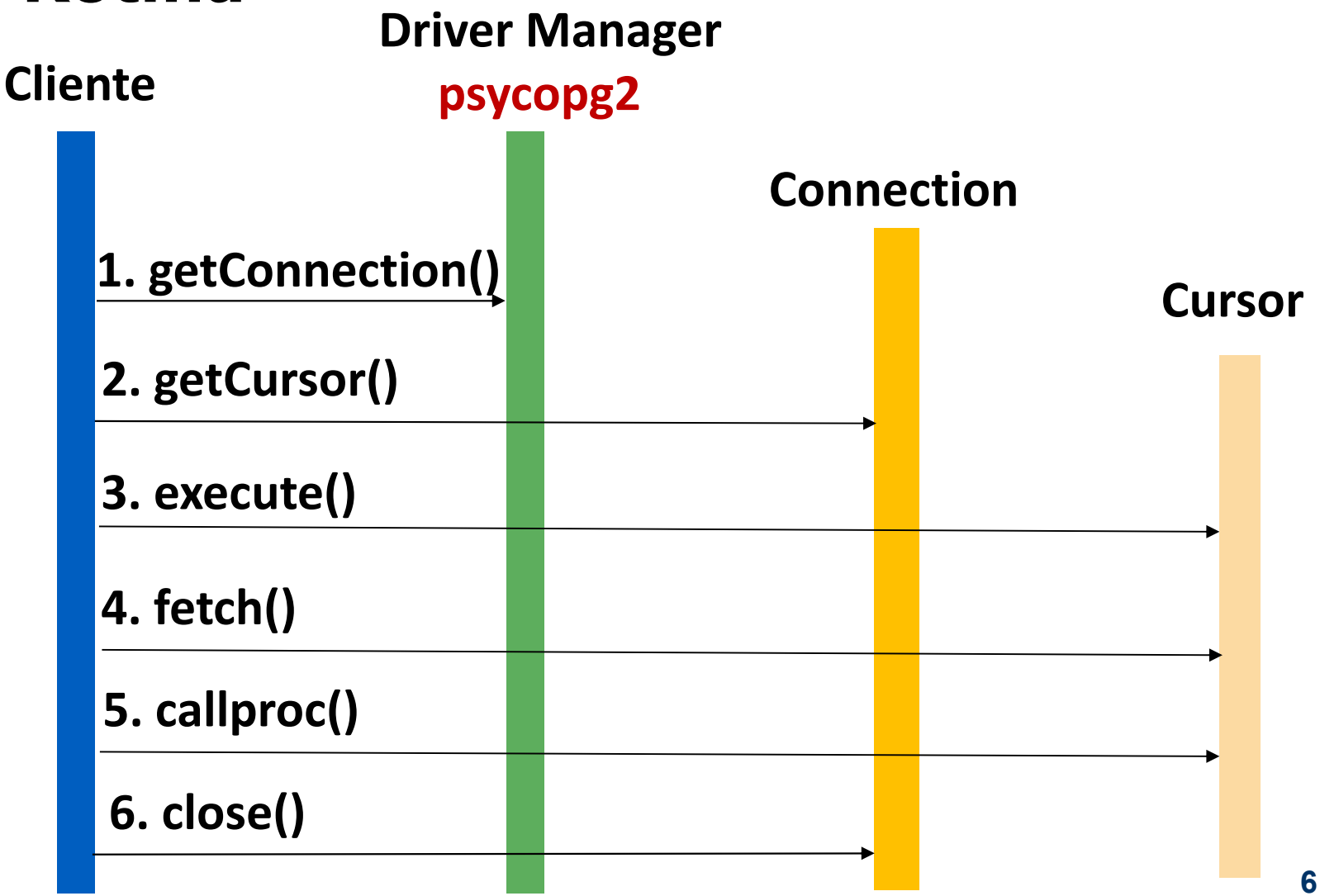
- **Modelo em 2/n camadas**



Modelo Cliente/Servidor

Adaptador de Banco de Dados

- **Rotina**



API-Python

1 Importando o pacote psycopg2

```
import psycopyg2
```

2 Conectando com o BD

Porta padrão: 5432

```
psycopyg2.connect( host, database,
                   user, password [, port])
```

- **host**:endereço do servidor de banco de dados. Pode ser localhost ou um endereço IP
- **database**: nome do banco de dados que deseja se conectar
- **user**: nome do usuário para autenticação
- **password**: senha para autenticação

API-Python

2 Conectando com o BD (cont.)

- Utilize um bloco **try-except**

connectdbpython1.py

```
import psycopg2
try:
    conn = psycopg2.connect(host='127.0.0.1',
    port=5432, database='dvdrental',
    user='postgres', password='ifpb')
    print('Sucesso na conexão')
except:
    print('Erro na conexão com o banco de
dados')
```

API-Python

3 Obtendo o cursor

- Criar um objeto **cursor** a partir da conexão retornada

```
connection.cursor()
```

- O cursor é importante para poder **executar queries SQL** ao Postgres usando Python

```
import psycopg2
try:
    ...
    print('Sucesso na conexão')
    cursor = conn.cursor()
except:
    ...
```

API-Python

4 Executando instruções SQL

```
cursor.execute( sql [,optionalParams] )
```

- Prepara e executa uma operação no banco de dados (*query* ou comando)
- **sqlstatement**: Um SELECT a ser enviado ao BD

```
import psycopg2
try:
    ...
    cursor = conn.cursor()
    cursor.execute('SELECT version() ')
except:
    ...
```

API-Python

4 Instruções SQL Parametrizadas

- Parametrização com **%s**
  - ✓ Na instrução SQL, corresponde ao espaço reservado ao parâmetro. O segundo argumento é a sequência de valores
- Valores: ordenados em uma **tupla**

```
cursor.execute('''
INSERT INTO tabela (id,nome)
VALUES (%s,%s);
''', (1001, 'Ana') )
```

Não fazer '%s'

  - ✓ Mesmo se o valor for um inteiro ou do tipo data, a parametrização será com **%s.** Não use **%d** ou **%f**

API-Python

4 Instruções SQL Parametrizadas

- Argumentos nomeados
  - ✓ Na instrução SQL, o posicionamento dos argumentos são definidos por **%(name)s**. Os valores são especificados em um mapeamento
- Valores: fornecidos por um **dicionário**

```
cursor.execute('''
SELECT * FROM tabela
WHERE id= %(inteiro)s AND data=%(data)s AND
nome=%(nome)s;
''', {'inteiro':100, 'nome':'Alex',
      'data':datetime.date(2019,04,22)} )
```

API-Python

5 Recuperando linhas de um SELECT

- Retornando uma única linha do result set, ou nenhuma quando não houver dados disponíveis

```
variavel = cursor.fetchone()
```

```
cursor = conn.cursor()
cursor.execute('SELECT version()')
versao = cursor.fetchone()
```

- Retornando todas as linhas remanescentes do *result set* como um **list**

```
myList = cursor.fetchall()
```

```
cursor.execute('SELECT id,nome FROM tabela')
funcionarios = cursor.fetchall()
```

API-Python

5 Recuperando linhas de um SELECT

- Processando linha a linha do *result set*

```
cursor.execute('SELECT id,nome FROM tabela')
while(True):
    row = cursor.fetchone()
    if( row == None):
      break
    print(row)
cursor.execute('SELECT id,nome FROM tabela')
rows = cursor.fetchall()
for row in rows:
    print('id',row[0],'name',row[1])
```

API-Python

6 Fechando a conexão

- Fecha a conexão com o banco de dados

```
connection.close()
```

  - ✓ **Importante**: `close()` não realiza uma chamada automática ao `commit()`

```
try:
    conn = psycopg2.connect(host='127.0.0.1',
    port=5432, database='dvdrental',
    user='postgres', password='ifpb')
except:
    print('Erro na conexão com o banco de dados')
finally:
  cursor.close()
  conn.close()
```

Feche também o cursor!

API-Python

- **Chamadas a stored procedures**

```
cursor.callproc( nome_sproc [,parameters])
```

  - Executa a stored procedures nomeada por **nome_sproc.**
  - A sequência de argumentos deve contemplar cada parâmetro que a stored procedure espera
  - Internamente, o método `callproc()` traduz a chamada do método e seus valores de entrada na seguinte instrução:

```
SELECT * FROM nome_sproc( valor1, valor2 )
```

API-Python

- **Chamadas a stored procedures**
  - Função a ser invocada, que retorna uma tabela

```
CREATE FUNCTION alunosPorCurso(idCurso integer)
  RETURNS TABLE(matricula INTEGER, nome VARCHAR)
AS $$
BEGIN
 RETURN QUERY
   SELECT a.matricula, a.nome FROM alunos a
   WHERE a.idCurso = idCurso;
END; $$
LANGUAGE plpgsql;
...
```

API-Python

- **Chamadas a stored procedures**
  - Recuperar todos os registros de alunos matriculados no curso de código 5

```
...
cursor = connection.cursor()
cursor.callproc('alunosPorCurso', (5,))
row = cursor.fetchone()
while row is not None:
    print(row)
    row = cursor.fetchone()
cursor.close()
..
```

API-Python

- **Transações** no psycopg
  - A classe **`connection`** é responsável por gerenciar as transações
  - **Funcionamento**
    - ✓ Quando um objeto **`cursor`** enviar a primeira instrução SQL ao BD, o psycopg cria automaticamente a transação
    - ✓ Todas as operações subsequentes pertencem à mesma transação
    - ✓ Se qualquer instrução falhar, o psycopg aborta a transação
  - Métodos da classe **`connection`** para finalização de uma transação: **`commit()`** e **`rollback()`**

API-Python

- **Transações** no psycopg
  - A classe **`connection`** possui o atributo **`autocommit`** (boolean), que determina se as modificações realizadas no banco são imediatamente efetivadas
    - ✓ Rollback não é possível
    - ✓ O default é **False**
  - Sintaxe

```
connection.autocommit = False # ou True
```

API-Python

- **Transações** no psycopg
  - Efetivando todas as modificaçõs da transação permanentemente (persistência no DB)

```
connection.commit()
```

  - Cancelando as modificações feitas pela transação

```
connection.rollback()
```

    - ✓ O fechamento de um objeto **connection** ou destruição do mesmo usando **del** resulta em um **rollback** implícito
  - Execute **commit** ou **rollback** antes de deixar o **connection** sem uso por um longo tempo
    - ✓ Bloqueio de registros

API-Python

- O atributo *read-only* **rowcount**
  - Atributo que retorna o número total de linhas de banco de dados afetados por um commando SELECT, INSERT, UPDATE ou DELETE

```
cursor.rowcount
```

API-Python

- **Finalizando uma transação**

connectdbpython2.py

```
cursor = connection.cursor()
cursor.execute('UPDATE funcionario \
        SET salario = salario * 1.2 \
        WHERE id BETWEEN 5 AND 25')
connection.commit()
print('Total de linhas
afetadas:',cursor.rowcount)
```

- Considere o mesmo código para um DELETE, SELECT ou INSERT

Aprimorando conhecimento

- **Pesquise você mesmo**
  - cursor.**executemany**()
  - cursor.**fetchmany()**

- A classe **DatabaseConnection()**
  - Classe disponibilizada para conexão com o banco em programas orientado a objeto
  - Configuração parametrizável via arquivo texto
  - Importa o drive de acesso ao PostgreSQL, porém, pode ser facilmente ajustado para se comunicar com outro fabricante de banco de dados

  DatabaseConnection.py

Referências Bibliográficas

- PostgreSQL Python: Connect To PostgreSQL Database Server: http://www.postgresqltutorial.com/postgresql-python/connect/
- Python PostgreSQL Tutorial Using Psycopg2: https://pynative.com/python-postgresql-tutorial/
- PostgreSQL – Python Interface: https://www.tutorialspoint.com/postgresql/postgresql_python.htm
- Using PostgreSQL in Python: https://www.datacamp.com/community/tutorials/tutorial-postgresql-python
- Python PostgreSQL Connection Pooling Using Psycopg2. https://pynative.com/psycopg2-python-postgresql-connection-pooling/

Referências Bibliográficas

- PEP 249 -- Python Database API Specification v2.0. Disponível em: https://www.python.org/dev/peps/pep-0249/