UNIVERSIDADE DE SÃO PAULO
Instituto de Matemática e Estatística
Ciências da Computação

Tutorial (DRAFT)

MAC0350 - Introdução ao Desenvolvimento de Sistemas de Software

Versão 2017

Python e Postgresql

Décio Lauro Soares
deciolauro@gmail.com
Eduardo Dias Filho
edudf90@gmail.com
Bruno Padilha
brnopdlha@gmail.com

Professor responsável:
Professor Doutor João Eduardo Ferreira
jef@ime.usp.br
Departamento de Ciências de Computação
IME - USP
11 de setembro de 2017

Introdução

Este tutorial tem a finalidade de apresentar um passo a passo de como integrar a linguagem de programação *Python* com o Sistema de Gerenciamento de Banco de Dados (SGBD) *PostgreSQL* em uma máquina que utilize o sistema operacional *Linux*.

Ainda que essa integração independa da distribuição de *Linux* utilizada, em alguns momentos, especialmente durante as configurações iniciais, será necessário à instalação e execução de comandos usando à interface de linha de comando (*Shell*).

Dada à impossibilidade de abranger todas as distribuições existentes (e as que venham à ser criadas após a realização deste), optou-se pela tentativa de ser o mais genérico possível e, quando estritamente necessário à indicação de comandos específicos, utilizou-se a sintaxe presente no conjunto de distribuições mais utilizadas, baseadas no sistema Debian que, segundo à distrowatch [9], ocupam três das cinco distribuições de maior interesse quando da realização deste trabalho. (Mint, Debian e Ubuntu)

Deste modo, cabe ao leitor adaptar alguns desses comandos específicos para sua distribuição de preferência, caso esta não venha a ser baseada em Debian.

Sobre o tutorial, seu objetivo é propiciar uma introdução guiada para facilitar o desenvolvimento e implementação de trabalhos na disciplina MAC0350 – Introdução ao Desenvolvimento de Sistemas de Software e está composto de 4 capítulos.

O primeiro capítulo (**Convenção de sintaxe** 1) traz as definições de sintaxe utilizadas ao longo deste texto em todos os exemplos.

Já o segundo capítulo (**Configuração Inicial** 2) apresenta os requisitos básicos, como instalá-los e configurá-los para a realização dos capítulos posteriores.

No terceiro capítulo (**Manipulação e manutenção do SGBD** 3), apresentamos a sintaxe de comandos para manipulação do SGBD tanto via ferramenta de acesso direto (*psql*) como através de uma API em Python (*psycopg2*).

O quarto capítulo (**Integração com o Django** 4) mostra mais exemplos de utilização da API apresentada no capítulo anterior, em conjunto com a introdução do framework Django.

Finalmente, a **Bibliografia** mostra uma relação de todo o material citado (ou relevantemente consultado) durante a realização deste trabalho enquanto que o capítulo **Outros materiais** trará alguns documentos previamente mencionados ou que ajudem a esclarecer melhor pontos específicos.

Sumário

Listings

Convenção de sintaxe

Ao longo desse texto, utilizamos à seguinte convenção de sintaxe:

Comandos no Shell e no SGBD

Todos os comandos executados no Shell (csh, tcsh, sh, bash, ksh, zsh, ...), bem como os comandos executados dentro do ambiente do SGBD (*psql*) serão apresentados em um Listing onde:

- **Comentários** de auxilio à compreensão de chamadas (ajuda), independentemente se são comentários no Shell ou no SGBD, serão precedidos por um @ e serão preferencialmente escritos em maiúsculas para destaque (Não confundir com comentários de código que terão sintaxe relativa à linguagem, por exemplo, em Python #, em SQL –, ...),
- **Comandos** executados em nível de usuário são precedidos por um $,
- **Comandos** executados em nível de super usuário são precedidos por um #,
- **Comandos** executados no SGBD são precedidos do nome do banco e dos símbolos = ou =>
- **Retornos** da execução de comandos não terão símbolo de precedência e serão, sempre que possível, apresentados de forma idêntica ao que será exibido na tela, eventualmente com uma versão resumida contendo apenas os valores relevantes.
- Sintaxe genérica de comando, parâmetro ou opção **obrigatória** será apresentada entre *angular brackets* (Por exemplo: `python <NOME_DO_PROGRAMA>.py`).
- Sintaxe genérica de comando, parâmetro ou opção **opcional** será apresentada entre *square brackets* (Por exemplo: `gcc test.c [-o test]`)

Sempre que houver possibilidade de confusão de sintaxe, utilizaremos comentários de ajuda para melhor compreensão.

Entretanto, para os casos básicos onde a interpretação não seja comprometida, esperamos um certo grau de maturidade do leitor para que este não interprete "cegamente" as definições aqui apresentadas.

Por exemplo, se o Listing está apresentando **apenas** um exemplo de código em Python, uma linha iniciada por # é claramente um comentário e não um comando de super usuário.

Além disso, para o caso de retornos "muito grandes", em alguns casos optaremos pela apresentação apenas de uma parte significativa do retorno, utilizando uma quebra do bloco do Listing e a reticencias vertical (⋮) para representar essa simplificação.

O Listing 1.1 apresenta uma pequena compilação dessas convenções de sintaxe.

Listing 1.1: Definição e exemplo de sintaxe de comandos

```
@Meu comentário                          ← (Isso é um comentário de ajuda)

@OUTRO COMENTÁRIO EM MAIUSCÚLAS          ← (Comentário de ajuda em destaque)

@PARA CHECAR SUA VERSÃO DO PYTHON        ← (Comentário de ajuda descritivo)
$ python --version                       ← (Comando em nível de usuário)
Python 2.7.12                            ← (Retorno de comando completo)

@ACESSAR UM BANCO LOCAL                  ← (Comentário de ajuda descritivo)
$ psql <NOME\_DO\_BANCO>                  ← (ex.  "Parametro" obrigatório)

@OUTRA FORMA DE ACESSO                   ← (Comentário de ajuda descritivo)
$ psql [-U <USUÁRIO> -d ]<BANCO>         ← (ex.  "Parametro" opcional)

# apt-get install sudo                   ← (Comando em nível de super usuário)
Reading package lists... Done            ← (Retorno de comando)
Building dependency tree                 ← (Retorno de comando)
Reading state information... Done        ← (Retorno de comando)
               ⋮                         ← (Abreviação de retorno)

meubanco=>\dt                            ← (Comando no SQGD)
            List of relations            ← (Inicio de retorno de dt)
 Schema |      Name       | Type  | Owner
--------+-------------------+-------+-------
 public | le01courses       | table | decio
               ⋮                         ← (Abreviação interna de retorno)
(207 rows)                               ← (Fim do retorno de dt)
```

Shell scripts e programas

Eventualmente, apresentaremos scripts em shell, bem como programas/scripts em Python. Esses scripts/programas também serão apresentados em um Listing, com a diferença de que o cabeçalho apresentará o nome completo do script/programa.

Além disso, assume-se que os leitores tenham algum conhecimento em shell scripting e em Python, ao menos para serem capazes de ler/compreender o que o script/programa se propõe a executar.

Apesar disso, tentaremos ao máximo comentar os casos mais complexos, embora esperemos alguma compreensão básica.

O Listing 1.2 mostra um exemplo da apresentação de um script/programa ingênuo em Python para realização de login.

Listing 1.2: simpleencrypt.py

```
#!/usr/bin/python2.7
###############################################################################
#        Simple password encryption for database program                      #
# (C) Copyright 2016 Decio Lauro Soares (deciolauro@gmail.com)                #
# This program is free software: you can redistribute it and/or               #
# modify it under the terms of the GNU General Public License as              #
# published by the Free Software Foundation, either version 3 of the          #
# License, or (at your option) any later version.                             #
#                                                                             #
# This program is distributed in the hope that it will be useful,             #
# but WITHOUT ANY WARRANTY; without even the implied warranty of              #
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the               #
# GNU General Public License for more details.                                #
#                                                                             #
# You should have received a copy of the GNU General Public License           #
# along with this program. If not, see <http://www.gnu.org/licenses/>#
###############################################################################
from __future__ import print_function
from Crypto.Cipher import XOR
import sys
import base64
import getpass

try:
    input = raw_input
except NameError:
    pass

# GLOBAL SETTINGS VARIABLES
safePass = True
mainCipher = "MyCipher1234"


def encrypt(plaintext, key=mainCipher):
    cipher = XOR.new(key)
    return base64.b64encode(cipher.encrypt(plaintext))


def decrypt(ciphertext, key=mainCipher):
    cipher = XOR.new(key)
    return cipher.decrypt(base64.b64decode(ciphertext))


def perguntaSN(d="N"):
    try:
        ask = input()
    except:
        ask = d
    if not ask:
        return d
    return ask.upper()


try:
    test = encrypt("test")
except NameError:
    print("Sera impossivel importar modulo para seguranca de senha!")
    print("Sua senha sera armazenada diretamente em modo texto")
    print("Para evitar esse risco instale o modulo Crypto para python 2.7")
    print("Pode haver possivel comprometimento da seguranca do banco")
    print("Deseja continuar sem instalar?(s/N)")
    ask = perguntaSN()
    if ask.upper() == "S":
        safePass = False
    else:
        sys.exit(1)
```

Configuração Inicial

Este capítulo trata da instalação e configuração inicial do sistema.

Em resumo, para acompanhar os próximos capítulos, é preciso que o seu sistema tenha:

- GNU make version 3.80 ou superior,
- Um compilador ISO/ANSI C, C89-compliant pelo menos (Em geral, se sua distribuição é relativamente recente, a versão do seu gcc é mais que suficiente),
- Caso haja o interesse de instalar a partir da fonte, para descompactar a distribuição é necessário a ferramenta *tar*, além de *gzip* ou *bzip2*,
- The GNU Readline library para utilizar o psql (provavelmente sua distribuição já possuí essa biblioteca instalada por padrão, caso contrário, você precisa dos pacotes readline and readline-devel),
- A biblioteca zlib compression (é possível instalar sem ela, mas você irá perder os comandos *pg_dump* e *pg_restore* que serão extremamente úteis),
- OpenSSL, OpenLDAP, e/ou PAM para suporte ao Kerberos (autenticação e encriptação),
- Python e algumas bibliotecas de suporte (vide abaixo),
- Espaço suficiente em disco: $\approx$ 120MB para instalação do Postgresql, $\approx$ 35MB + espaço dos dados para o banco de dados (um banco vazio ocupa $\approx$ 35MB e os dados ocupam aproximadamente cinco vezes a quantidade de espaço que um arquivo texto com os mesmos dados ocupariam), $\approx$ 150MB para eventuais análises de regressão, além de espaço para instalação do Python, e dos requisitos listados anteriormente.

Para o Python, tanto a versão 2 (Python 2.3 ou superior) quanto a versão 3 (Python 3.1 ou superior) são passiveis de utilização.

Entretanto, dada a incompatibilidade de regressão em alguns aspectos, **utilizaremos a sintaxe da versão de Python 2** para apresentação de todos os exemplos.

Apesar disso, o postgresql implementa interfaces para Python 2 (plpython2u) e Python 3 (plpython3u), ficando a cargo do leitor a adaptação dos códigos caso venha a optar por Python 3.

Além da instalação da biblioteca dessa linguagem, algumas API's também são necessárias, de modo que seu sistema deve ter espaço suficiente para instalá-las. (em especial, o sistema **deve** conter o pacote *psycopg2*)

As próximas seções trazem um breve explicação sobre como verificar e instalar esses requisitos, bem como realizar a configuração inicial do SGBD.

Verificação de requisitos e instalação

Para praticamente todos os requisitos listados na seção anterior, sua verificação manual é feita através do comando apresentado no Listing 2.1.

É importante notar que os valores de retorno são apresentados apenas a titulo de exemplo. Os valores obtidos em suas máquinas podem diferir dos aqui apresentados, bastando apenas que eles se adequem aos requisitos listados anteriormente.

Além disso, caso você use uma distribuição Linux dentre as mais conhecidas (família Red Hat, família Debian, Ubuntu, SuSE, OpenSuSE), é possível seguir o passo a passo de instalação do PostgreSQL presente em: https://www.postgresql.org/download/ sem se preocupar com verificação manual das dependências.

O mesmo vale para a instalação do Python, presente em https://www.python.org/downloads/, do gerenciador de pacotes *pip* (https://pypi.python.org/pypi/pip) e do pacote *psycopg2* (http://initd.org/psycopg/docs/).

Listing 2.1: Exemplo de verificação manual de dependências

```
$ <NOME_DO_REQUISITO> --version

@EXEMPLOS:
$ make --version
GNU Make 4.1
    ⋮

$ python --version
Python 2.7.12

$ gcc --version
gcc 5.4.0 20160609
    ⋮
```

No caso particular do Ubuntu, a instalação do Postgresql pode ser feita por:

Listing 2.2: Instalação do Postgresql no Ubuntu

```
@Edite/crie o arquivo /etc/apt/sources.list.d/pgdg.list, adicionando:
# deb http://apt.postgresql.org/pub/repos/apt/ YOUR_UBUNTU_VERSION_HERE-pgdg main

@Atualize a chave e faça um update dos repositórios
$ wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -
OK
$ sudo apt-get update

$ sudo apt-get install postgresql-9.6
```

Para instalação do pacote *psycopg2*, necessário para criar a interface do SGBD com a linguagem Python, a melhor forma de instalá-lo é através do gerenciador de pacotes do próprio Python, o *pip*.

Listing 2.3: Instalação do Psycopg2

```
$ pip install -U pip
$ pip install psycopg2
```

Setup inicial do PostgreSQL

Supondo que todos os pré-requisitos listados foram devidamente instalados, durante a instalação, o postgreSQL criou um grupo/usuário (*postgres*) com permissões administrativas sobre todos os bancos presentes no sistema. (similar ao *root* do Linux)

Dado que esse usuário tem permissões elevadas sobre qualquer banco do sistema, vamos primeiramente reforçar a segurança.

Listing 2.4: Configurações iniciais de segurança

```
@PRIMEIRAMENTE MUDE PARA O USUÁRIO postgres
$ sudo su - postgres
postgres@machine$

@CONECTE NO BANCO COM ESSE USUÁRIO
postgres@machine$ psql
psql (9.6.5)
Type "help" for help.

postgres=#

@ALTERE A SENHA DO ADMINISTRADOR (NÃO ESQUEÇA O ; NO FINAL)
@SENHA ENTRE ASPAS SIMPLES
@CUIDADO, A SENHA IRÁ APARECER EM PLAIN TEXT NA TELA
postgres=# ALTER USER postgres WITH ENCRYPTED PASSWORD '<sua_senha>';
ALTER ROLE

@DESCONECTE DO BANCO
postgres=# \q
postgres@machine$

@VOLTE AO SEU USUÁRIO E EDITE O ARQUIVO pg_hba.conf
postgres@machine$ exit
$ sudo vim /etc/postgresql/9.6/main/pg_hba.conf

@NA LINHA REFERENTE AO postgres (PADRÃO LINHA 85)
local all   postgres  peer
@TROQUE peer POR md5
local all   postgres  md5

@REINICIE O BANCO DE DADOS
$ sudo /etc/init.d/postgresql restart
* Restarting PostgreSQL 9.6 database server

@TESTE SE AS ALTERAÇÕES FORAM IMPLEMENTADAS
$ psql -U postgres
Password for user postgres:

@SE TUDO CORRER BEM, APÓS O PASSWORD VOCÊ CONECTARÁ NO BANCO
@PARA DESCONECTAR NOVAMENTE, USE:
postgres=# \q
```

Com a segurança do usuário administrativo reforçada, agora vamos fazer uso de um *SQL DATA CONTROL LANGUANGE* (DCL) para criar um ROLE (Tipo de usuário + permissões) para o seu próprio usuário afim de evitar o uso desnecessário e perigoso do usuário postgres. O comando DCL à ser utilizado será o *CREATE ROLE* cuja sintaxe SQL em postgreSQL pode ser encontrada em https://www.postgresql.org/docs/current/static/sql-createrole.html

Por questões de simplicidade, usaremos no SGBD o mesmo <USERNAME> utilizado no Linux. Entretanto, nada impede que o você escolha um <USERNAME> diferente, desde que realize os ajustes necessários, principalmente na edição do arquivo de `pg_hba.conf`.

Assim, para o setup inicial, faremos:

Listing 2.5: Criação de ROLE

```
@CASO VOCÊ NÃO SAIBA SEU <USERNAME>, DIGITE:
$ whoami
decio
@A MENOS QUE VOCÊ TENHA ESSE NOME LINDO (OU SEJA UM FÃ) :-)
@O OUTPUT DO SEU <USERNAME> SERÁ DIFERENTE

@AGORA CONECTE NOVAMENTE NO BANCO COM O USUARIO postgres
$ psql -U postgres
psql (9.6.5)
Type "help" for help.

postgres=#


@CRIE UM ROLE PARA O SEU USUÁRIO
postgres=# CREATE ROLE <USERNAME> WITH ENCRYPTED PASSWORD '<sua_senha>';

@VERIFIQUE O ROLE
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 decio     | Cannot login                                               | {}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

Como podemos ver, o SGBD possui agora dois *Role*'s, o super usuário *postgres* e o usuário que acabamos de criar.

Entretanto, o *role* decio ainda não tem qualquer permissão sobre o SGBD.

Precisamos então modificar as permissões atuais, para pelo menos, permitir o login ao banco e a criação de novos bancos de dados.

Uma explicação detalhada das permissões possíveis será apresentada nos slides de aula e está além do escopo deste tutorial.

Apesar disso, a documentação completa com explicação sobre todos os campos pode ser encontrada em https://www.postgresql.org/docs/current/static/sql-createrole.html.

Listing 2.6: Modificação de permissões

```
@DANDO PERMISSÕES (LOGIN, CREATEDB) AO NOSSO USUARIO
postgres=# ALTER ROLE <USERNAME> WITH LOGIN CREATEDB VALID UNTIL '2018-01-01';
ALTER ROLE
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 decio     | Create DB                                                  +| {}
           | Password valid until 2018-01-01 00:00:00-02                |
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}

@QUASE ACABANDO, DESCONECTE DO BANCO
postgres=# \q

@EDITE NOVAMENTE O ARQUIVO pg_hba.conf
$ sudo vim /etc/postgresql/9.6/main/pg_hba.conf

@NA LINHA REFERENTE A CONEXÃO local (PADRÃO LINHA 90)
@ABAIXO DE: # "local" is for Unix domain socket connections only
@INSIRA:
local   all             <USERNAME>                              md5

@SEU ARQUIVO DEVE FICAR ASSIM:
# Database administrative login by Unix domain socket
local   all             postgres                                md5

# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             decio                                   md5
local   all             all                                     peer
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
# IPv6 local connections:
host    all             all             ::1/128                 md5

@SALVE O ARQUIVO E REINICIE O BANCO DE DADOS
$ sudo /etc/init.d/postgresql restart
* Restarting PostgreSQL 9.6 database server

@O ÚLTIMO PASSO AGORA É CRIAR UM BANCO PARA O SEU USUÁRIO (USAMOS ESSE NOS EXEMPLOS)
$ createdb                                          ← cria um banco com mesmo nome do usuário

@OPCIONALMENTE, VOCÊ PODE TAMBÉM ATRIBUIR UM NOME AO BANCO, POR EXEMPLO
$ createdb meubanco                                 ← cria o banco meubanco cujo owner é o seu usuário

@PRONTO, SEU SISTEMA ESTÁ CONFIGURADO
@PARA TESTAR FAÇA:
$ psql -U <USERNAME>
psql (9.6.5)
Type "help" for help.

<USERNAME>=>

@OU SE VOCÊ NOMEOU O BANCO
$ psql <NOME_DO_BANCO>
```

Manipulação e manutenção do SGBD

Existem diversos modos de manipulação e manutenção dos dados presentes em um SGBD, sejam eles gráficos ou textuais.

Para o PostgreSQL em particular, abordaremos dois deles:

- Acesso utilizando sua ferramenta padrão (*psql*)
- Acesso utilizando uma API em Python (*psycopg2*)

Acesso via Psql

O *psql* é um cliente no modo terminal do PostgreSQL que foi utilizado durante a configuração inicial.

Ele permite digitar comandos interativamente (ou via arquivo), submetê-los para o PostgreSQL e verificar seus resultados.

Além disso, o *psql* disponibiliza vários meta-comandos e diversas funcionalidades semelhantes às do interpretador de comandos (*Shell*) para facilitar a criação de scripts e automatizar um grande número de tarefas.

Sua utilização/inicialização segue a seguinte sintaxe:

Listing 3.1: Sintaxe de chamada do psql

```
psql [opção...] [nome_do_banco_de_dados [nome_do_usuário]]

@Exemplo: Lista todos os comandos possíveis:
$ psql -?

@Executa o login no banco meubanco do usuário decio no servidor 192.168.1.15
§ psql -h 192.168.1.15 -U decio -d meubanco
```

Após à inicialização do terminal, o *psql* também suporta alguns comandos básicos de auxilio à manutenção e gerência do banco, como:

Listing 3.2: Comandos básicos de interface em psql

```
@DESCONECTAR DO BANCO: \q
meubanco=> \q

@LISTA OS ROLES E ATRIBUTOS: \du
meubanco=> \du

@LISTA OS SCHEMAS E SEUS DONOS: \dn
meubanco=> \dn

@LISTA OS SCHEMAS, SEUS DONOS, PRIVILÉGIOS E DESCRIÇÃO: \dn+
meubanco=> \dn+

@LISTA TODOS OS BANCOS DE DADOS QUE O USUÁRIO TEM ACESSO: \l
meubanco=> \l                                   ← Nota: L minúsculo

@LISTA TODAS AS TABELAS, VIEW, SEQUENCIAS PRESENTES NO SCHEMA ATUAL: \dt
meubanco=> \dt

@LISTA PROPRIEDADES ESPECIFICAS: \d+
meubanco=> \d+ [NOME_DA_TABELA]
```

Além desse conjunto de comandos básicos de acesso, alguns dos quais não passam de simplificações de acesso aos catálogos do sistema, o terminal também aceita uma infinidade de comandos SQL, sejam eles do tipo *Data Definition Language* (DDL), *Data Manipulation Language* (DML), *Data Control Language* (DCL) ou *Transaction Control Language* (TCL).

No capítulo anterior, vimos alguns exemplos de comandos DCL (CREATE ROLE, ALTER ROLE).

Por hora, à classificação do tipo de comando não será discutida, uma vez que ela será extensivamente trabalhada durante à disciplina.

Entretanto, faremos agora uma introdução guiada de uma sequencia de comandos DDL e DML sobre um cenário fictício de criação e manipulação de uma base de dados simples com comandos que serão muito utilizados.

Para isso, é importante que você tenha completado a configuração inicial ou que possua o postgreSQL instalado e tenha permissões suficientes para executar os próximos comandos.

Além disso, por questões de sintaxe de linguagem e/ou questões de boas práticas, adotaremos o seguinte padrão nos exemplos à seguir:

- Todos os **comandos** (não confunda com linhas) SQL devem ser terminados com um ponto e vírgula,
- Comandos de múltiplas linhas foram devidamente identados,
- Palavras reservadas foram escritas em maiúsculas,
- Toda relação (tabela) possuí um prefixo indicativo do projeto e de sua ordem de criação (Ex. em le01emissora, o “le01” indica que está é a primeira (01) relação criada grupo “learning”),
- Todo atributo (coluna), traz um prefixo indicativo do nome da relação associada (Ex. “emi_nome” identifica o atributo nome na relação emissora).

Com exceção do primeiro item, que é obrigatório, à adoção dos outros padrões é opcional e, desde que sua codificação seja consistente, você é livre para adaptá-los para o padrão que preferir.

Listing 3.3: Exemplos do DDL CREATE TABLE

```
@CONECTE NO BANCO COM O SEU USUARIO
$ psql -U decio
psql (9.6.5)
Type "help" for help.

decio=>

@CRIAÇÃO DE UMA TABELA COM PRIMARY KEY E CONSTRAINT
decio=> CREATE TABLE le01emissora (
decio(>     emi_nome                character(20),
decio(>     emi_nro_reporteres      integer DEFAULT 0,
decio(>     CONSTRAINT pk_emissora  PRIMARY KEY (emi_nome),
decio(>     CONSTRAINT check_nr     CHECK (emi_nro_reporteres >= 0)          ← Não tem virgula
decio(> );
CREATE TABLE

@CHECANDO À CRIAÇÃO DA TABELA
decio=> \dt
          List of relations
 Schema |     Name      | Type  | Owner
--------+---------------+-------+-------
 public | le01emissora  | table | decio
(1 row)

@PROPRIEDADES ESPECIFICAS DA RELAÇÃO EMISSORA
decio=> \d+ le01emissora
                                Table "public.le01emissora"
       Column        |      Type       | Modifiers | Storage  | Stats target | Description
---------------------+-----------------+-----------+----------+--------------+-------------
 emi_nome            | character(20)   | not null  | extended |              |
 emi_nro_reporteres  | integer         | default 0 | plain    |              |
Indexes:
    "pk_emissora" PRIMARY KEY, btree (emi_nome)
Check constraints:
    "check_nr" CHECK (emi_nro_reporteres >= 0)

@E AS PROPRIEDADES ESPECIFICAS DO INDICE pk_emissora
decio=> \d+ pk_emissora
           Index "public.le01emissora"
  Column  |      Type      | Definition | Storage
----------+----------------+------------+----------
 emi_nome | character(20) | emi_nome   | extended
primary key, btree, for table "public.le01emissora"
```

No exemplo apresentado no Listing 3.3, temos à criação de uma entidade chamada de emissora, que possuí dois atributos:

- Nome (emi_nome) que traz seu nome e é chave primaria,
- Número de reporteres (emi_nro_reporteres) que define o número de reporteres e, caso este não seja indicado, tem por padrão 0 reporteres.
- Além disso, o número de reporteres também possuí uma restrição de domínio, indicando que o número de reporteres deve ser maior ou igual à zero.

Suponha agora à evolução não prevista do sistema, onde à entidade emissora também devesse guardar o país de origem. Essa alteração pode ser realizada pelo comando DDL ALTER TABLE (https://www.postgresql.org/docs/9.6/static/sql-altertable.html).

Para fins didáticos, suponha ainda que o responsável pela implementação dessa evolução desconhecesse o padrão utilizado sobre o prefixo das iniciais da relação sobre o nome do atributo. Uma possível sequencia de comandos de ajustes seria (sinta-se à vontade para checar cada alteração via um dos comandos básicos, como `\d+`):

Listing 3.4: Exemplos de DDL parte 2

```
@INSERÇÃO DO ATRIBUTO pais À RELAÇÃO emissora
decio=> ALTER TABLE le01emissora ADD COLUMN pais character(20);
ALTER TABLE

@AJUSTE DO NOME DO ATRIBUTO AO PADRÃO UTILIZADO
decio=> ALTER TABLE le01emissora RENAME pais TO emi_pais;
ALTER TABLE

@AJUSTE NA RESTRIÇÃO DO ATRIBUTO PARA NÃO PERMITIR VALORES NULOS
decio=> ALTER TABLE le01emissora ALTER COLUMN emi_pais SET NOT NULL;
ALTER TABLE

@REMOÇÃO DO ATRIBUTO emi_pais
decio=> ALTER TABLE le01emissora DROP COLUMN emi_pais;
ALTER TABLE
```

Partindo agora para comandos DML (INSERT, SELECT, ...), vamos manipular alguns dados na relação emissora.

Listing 3.5: Exemplos de DML parte 1

```
@INSERÇÃO DE TUPLAS EM emissora
decio=> INSERT INTO le01emissora (emi_nome, emi_nro_reporteres) VALUES
decio-> ('SBT', 100000),
decio-> ('GLOBO', 200000),
decio-> ('REDE TV', 50000),
decio-> ('CULTURA', 10000);
INSERT 0 4

@CHECAGEM DA CONSTRAINT DO NÚMERO DE REPORTERES
decio=> INSERT INTO le01emissora VALUES ('BAND', -5000);
ERROR:  new row for relation "le01emissora" violates check constraint "check_nr"
DETAIL:  Failing row contains (BAND                , -5000).

@SELECT DAS TUPLAS DE le01emissora
decio=> SELECT * FROM le01emissora;
       emi_nome        | emi_nro_reporteres
-----------------------+--------------------
 SBT                   |             100000
 GLOBO                 |             200000
 REDE TV               |              50000
 CULTURA               |              10000
(4 rows)

@SELECT DAS EMISSORAS COM MAIS QUE 80000 FUNCIONARIOS EM ORDEM DECRESCENTE
decio=> SELECT * FROM le01emissora
decio-> WHERE emi_nro_reporteres > 80000
decio-> ORDER BY emi_nro_reporteres DESC;
       emi_nome        | emi_nro_reporteres
-----------------------+--------------------
 GLOBO                 |             200000
 SBT                   |             100000
(2 rows)

@REMOÇÃO DA EMISSORA CULTURA
decio=> DELETE FROM le01emissora WHERE emi_nome = 'CULTURA';
DELETE 1

@ALTERAÇÃO DO NUMERO DE REPORTERES DA GLOBO COM UPSERT
decio=> INSERT INTO le01emissora (emi_nome, emi_nro_reporteres) VALUES
decio->      ('GLOBO', 250000)
decio->      ON CONFLICT (emi_nome) DO UPDATE
decio->      SET emi_nro_reporteres=EXCLUDED.emi_nro_reporteres;
INSERT 0 1
decio=> select * from le01emissora;
       emi_nome        | emi_nro_reporteres
-----------------------+--------------------
 SBT                   |             100000
 REDE TV               |              50000
 GLOBO                 |             250000
(3 rows)
```

Finalmente, vamos voltar ao DDL, agora para "limpar" nosso banco com DROP TABLE.

Listing 3.6: Exemplos de DDL parte 3

```
@REMOÇÃO DA RELAÇÃO emissora
decio=> DROP TABLE le01emissora;
DROP TABLE

@CHECAGEM
decio=> \dt
No relation found
```

O intuito desses exemplos foi o de apenas apresentar uma breve introdução ao terminal do *psql*.

Entretanto, alguns conceitos foram abordados apenas superficialmente, enquanto que outros sequer foram mencionados.

Com essa introdução inicial e à documentação apesentada em [8], recomenda-se agora que você tente verificar/implementar:

- Como implementar relacionamentos entre duas relações? (1x1? 1xN? NxM?)
- Como criar chave secundária, terciária, ...
- Como criar chave primária composta de mais de um atributo?
- O que acontece quando fazemos um DROP TABLE de uma tabela referenciada por outra?
- O que acontece quando alteramos um atributo de uma tabela que já contém tuplas?
- O que acontece quando tentamos atribuir à constraint PRIMARY KEY a uma tabela que já contém tuplas? E se houver repetições?
- O que acontece quando tentamos atribuir uma constraint CHECK a uma tabela que já contém tuplas?
- O que acontece quando tentamos remover à constraint de PRIMARY KEY de uma tabela referenciada?
- ⋮

Acesso via Psycopg2

Essa seção introduz o Psycopg2 que é um adaptador de banco de dados PostgreSQL para a linguagem de programação Python. A implementação atual do Psycopg2 dá suporte a:

- Python 2 versões de 2.6 a 2.7
- Python 3 versões de 3.2 a 3.6
- PostgreSQL server versões de 7.4 a 9.6
- PostgreSQL client library versões a partir de 9.1

Supondo que você seguiu o tutorial de instalação apresentado no capítulo anterior, ou que seu sistema já está configurado, as próximas sessões deste texto apresentarão à forma de uso do *Psycopg2* em um programa/script escrito em Python.

Exemplo básico

Para utilizar o Psycopg2 em um código Python, devemos antes importar o pacote no código através de `import psycopg2`.

Para criar uma sessão no banco de dados e receber um objeto de conexão no banco de dados podemos usar a função `connect`:

Listing 3.7: Exemplos de script para conexão

```
def logBase():
    dbName, userName, hostName, passCrypt, dbPort = readConfig()
    dbPassword = decrypt(passCrypt)
    try:
        connectOptions = "dbname='" + dbName +\
                         "' user='" + userName +\
                         "' host='" + hostName +\
                         "' password='" + dbPassword +\
                         "' port='" + dbPort + "'"
        conn = psycopg2.connect(connectOptions)
    except:
        print("Nao foi possivel conectar a base de dados\n")
        sys.exit(1)
    cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
    return conn, cur
```

A partir do objeto de conexão, precisamos usar um cursor para realizar operações no banco de dados:

Listing 3.8: Exemplos de cursor usando o Psycopg2

```
try:
    conn, cur = logBase()                          ← Função apresentada no Listing anterior
    cur.execute("""                                ← 3 aspas duplas para comandos de múltiplas linhas
CREATE TABLE le01emissora (
    emi_nome                character(20),
    emi_nro_reporteres      integer DEFAULT 0,
    CONSTRAINT pk_emissora  PRIMARY KEY (emi_nome),
    CONSTRAINT check_nr     CHECK (emi_nro_reporteres >= 0)        ← Não tem virgula
);
  """)                                             ← Fecha a string do cursor
except psycopg2.ProgrammingError, e:
    print("Erro no criação da tabela")
    print(e)
    waitKey()
    print("RollBack realizado!")
    conn.rollback()
```

O classe cursor possuí ainda algumas formas de recuperação de dados, principalmente dados provenientes de um DML SELECT. O Listing 3.9 mostra algumas utilizações.

Além disso, recomendamos à leitura da documentação completa da classe que pode ser encontrada em http://initd.org/psycopg/docs/cursor.html.

Finalmente, o Listing 3.10 apresenta à gestão de persistencia sobre o banco e o fechamento das comunicações no Python script.

Listing 3.9: Exemplos de recuperação de dados da classe cursor

```
try:
    conn, cur = logBase()
    cur.execute("SELECT * FROM le01emissora;")      ← Não esqueça o ponto e vírgula
except psycopg2.ProgrammingError, e:
    print("Erro no select")
    print(e)
    waitKey()
    print("RollBack realizado!")
    conn.rollback()

# Primeira forma, iterando sobre o próprio cursor
for tupla in cur:
    print tupla

# Usando o fetchone()
primeira_tupla = cur.fetchone()

# Só o nome da emissora
primeira_emissora = cur.fetchone()[0]

# Usando o fetchmany([numero de elementos])
lista_com_3_emissoras = cur.fetchmany(3)

# Ou colocando todas em uma lista com fetchall()
lista_completa = cur.fetchall()
```

Listing 3.10: Persistência e fechamento de conexão Psycopg2

```
try:
    conn, cur = logBase()
    cur.execute("""
INSERT INTO le01emissora (emi_nome, emi_nro_reporteres) VALUES
    ('BAND', 90000),
    ('RECORD', 76000);
    """)                                            ← fecha a string do cursor
except psycopg2.ProgrammingError, e:
    print("Erro no insert")
    print(e)
    waitKey()
    print("RollBack realizado!")
    conn.rollback()

# Neste ponto, se não houve raise de exception, temos duas possibilidades:

# Tornar persistentes os inserts acima com
conn.commit()
# Ou reverter o banco para o estado inicial com
conn.rollback()

# Finalmente, para encerrar, fechamos o cursor e a conexão
cur.close()
conn.close()
```

É fácil notar que a interface do *psycopg2* permite a utilização identica à apresentada via terminal do *psql* na seção anterior, já que aceita diretamente a sintaxe SQL.

Integração com o Django

Django é um framework de desenvolvimento para web na linguagem Python.

Instalação

Antes de instalar o Django, é necessário instalar Python, PostgreSQL e Psycopg2. Instruções para instalar tais requisitos são encontradas nos capítulos anteriores.

O jeito mais prático de instalar o django é através do pip. Os seguintes comandos devem ser executados para instalar o Django e testar a instalação:

Listing 4.1: Instalação Django

```
$ pip install django

@Para confirmar se a instalação foi realizada com sucesso
$ python -m django --version
```

Criação de projeto e de app

Um app é uma aplicação web com uma funcionalidade que pode ser comum a vários projetos. Um projeto consiste de um conjunto de configurações e de apps.

Para criar um projeto com o Django, basta executar o django-admin com a opção startproject, o que gerará uma estrutura de pastas como a mostrada a seguir:

Listing 4.2: Criação de projeto

```
$ django-admin startproject <nome_projeto>

@No caso, o nome do meu projeto foi "meuprojeto". Estrutura de pastas gerada:
@meuprojeto/
@    manage.py
@    meuprojeto/
@        __init__.py
@        settings.py
@        urls.py
@        wsgi.py
```

Por enquanto, é importante saber dessa estrutura que:

- A pasta “meuprojeto” mais externa é a pasta raiz do projeto e apenas serve como container do projeto.
- O arquivo “manage.py” é um programa que oferece algumas interações importantes com o projeto.
- A pasta “meuprojeto” interna é o pacote python do projeto.
- O arquivo “settings.py” é aonde o projeto é configurado.
- O arquivo “urls.py” é aonde são declaradas as URLs deste projeto.

Para verificar que o projeto está funcionando, é possível executar o projeto em um servidor de desenvolvimento com o seguinte comando:

Listing 4.3: Runserver

```
$ cd meuprojeto
$ python manage.py runserver [ip:porta]

@Os parâmetros ip e porta tem default localhost e 8000 respectivamente.
@O retorno da operação deve ser algo como:
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.

September 06, 2017 - 15:50:53
Django version 1.11, using settings 'meuprojeto.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Para criar um app no projeto, podemos usar o seguinte comando:

Listing 4.4: Criação de app

```
$ python manage.py startapp <nome_app>

@No caso, o nome do meu app foi "meuapp". Estrutura de pastas gerada:
@meuapp/
@    __init__.py
@    admin.py
@    apps.py
@    migrations/
@        __init__.py
@    models.py
@    tests.py
@    views.py
```

Os arquivos gerados nesta estrutura são discutidos nas próximas sessões que trazem exemplos de desenvolvimento de apps em Django.

Exemplo básico

Este exemplo mostra a criação de um app básico, sem acesso ao banco de dados, sem preocupação com apresentação. O app no exemplo contém apenas uma requisição e uma resposta.

Para iniciar o desenvolvimento, abrir o arquivo “views.py” que está na pasta do app criado. Neste arquivo deve ser adicionado o seguinte trecho:

Listing 4.5: meuapp/views.py

```
from django.http import HttpResponse


def index(request):
    return HttpResponse("Testando meuapp.")
```

A view criada gera uma resposta HTTP com o texto “Testando meuapp” quando for requisitada. Devemos agora indicar ao projeto como deve ser requisitada a view.

Por motivos de organização, deve ser criado o arquivo “meuapp/urls.py”. Tal arquivo será responsável por mapear padrões de URL, especificados em expressões regulares, em views dentro do app.

Listing 4.6: meuapp/urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

E então, no arquivo “meuprojeto/urls.py”, deve ser incluído o arquivo “meuapp/urls.py”, mapeando suas URLs dentro do projeto.

Listing 4.7: meuprojeto/urls.py

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^meuapp/', include('meuapp.urls')),
    url(r'^admin/', admin.site.urls),
]
```

Deste modo, é dito ao projeto que o caminho “meuapp/” deve retornar ao usuário uma resposta HTTP com o texto “Testando meuapp.”. Para realizar o teste, basta executar o servidor de desenvolvimento como mostrado na sessão anterior e em um navegador, digitar na barra de endereços “localhost:8000/meuapp/” e verificar se o texto “Testando meuapp.” é exibido.

Configurando Django com banco de dados

No arquivo “meuprojeto/settings.py” existe uma série de configurações, uma delas é um dicionário vazio de nome DATABASE, que deve ser configurado com os dados definidos no PostgreSQL da seguinte forma:

Listing 4.8: Configuração do projeto

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': '<nome_banco>',
        'USER': '<usuario_banco>',
        'PASSWORD': '<senha_usuario_banco>',
        'HOST': '<ip_servidor_banco>',
        'PORT': '<porta_servidor_banco>',
    }
}
```

Observe que há também uma configuração INSTALLED_APPS, que contém os nomes dos apps contidos no projeto. Por padrão, existem alguns apps que vêm com o Django com funcionalidades úteis como sessão, autenticação, etc. Alguns destes apps utilizam tabelas no banco de dados e para criar tais tabelas automaticamente é necessário executar o comando migrate:

Listing 4.9: Comando migrate

```
$ python manage.py migrate
```

O comando migrate olha a configuração INSTALED_APPS e verifica se algum dos apps listados utiliza alguma tabela que não está no banco configurado em DATABASE e cria o que estiver faltando no banco.

Para indicar quais tabelas um app usa no banco é necessário definir os modelos do app, no arquivo "meuapp/models.py". Suponhamos que meuapp ofereça uma funcionalidade de enquete e que para isto, é necessário usar uma tabela de perguntas e uma de alternativas, então o arquivo "meuapp/models.py" fica deste jeito:

Listing 4.10: meuprojeto/models.py

```
from django.db import models


class Pergunta(models.Model):
    texto_pergunta = models.CharField(max_length=200)
    data_publicacao = models.DateTimeField('date published')


class Alternativa(models.Model):
    pergunta = models.ForeignKey(Pergunta, on_delete=models.CASCADE)
    texto_alternativa = models.CharField(max_length=200)
    votos = models.IntegerField(default=0)
```

As classes que herdam de models.Model são representações de tabelas e seus atributos são representações de colunas da tabela referente à classe do atributo. É importante observar que os atributos são objetos de classes que refletem o tipo da coluna que representam e que os construtores de tais classes podem receber parâmetros. Maiores detalhes sobre a definição de modelos no Django podem ser encontrados nesta seção da documentação do Django [4].

Finalmente, para aplicar no banco as alterações especificadas no modelo do app deve-se antes adicionar na configuração de INSTALLED_APPS a classe "meuapp.apps.MeuappConfig", e então os seguintes comandos:

Listing 4.11: Migração

```
$ python manage.py makemigrations <nome_do_app>
Migrations for 'meuapp':
  meuapp/migrations/0001_initial.py:
    - Create model Alternativa
    - Create model Pergunta
    - Add field pergunta to alternativa

@O comando makemigrations não aplica as mudanças no banco de dados,
   apenas cria as migrações a serem aplicadas pelo comando migrate.

$ python manage.py migrate
```

Uso dos modelos

As classes de modelos oferecem uma abstração das funcionalidades de criação, recuperação, alteração e remoção de dados que podem ser usadas tanto em views do app quanto no próprio shell do Python. Maiores detalhes sobre as operações de interação com o banco de dados oferecidas pelos modelos podem ser encontrados nesta seção da documentação do Django [3].

Pode-se, por exemplo, alterar a view index para retornar uma resposta HTTP com o texto das 5 últimas perguntas publicadas:

Listing 4.12: meuapp/views.py

```
from django.http import HttpResponse

from .models import Pergunta


def index(request):
    lista_perguntas = Pergunta.objects.order_by('-data_publicacao')[:5]
    output = ', '.join([p.texto_pergunta for p in lista_perguntas])
    return HttpResponse(output)
```

Além de usar os modelos para interagir com o banco de dados nas views, também é possível usar os modelos no shell do python através da opção shell de manage.py, como mostrado no exemplo abaixo:

Listing 4.13: Modelos no shell

```
$ python manage.py shell

>>> from meuapp.models import Pergunta

@Conjunto de perguntas está inicialmente vazio
>>> Pergunta.objects.all()
<QuerySet []>

@Criação de pergunta
>>> from django.utils import timezone
>>> p = Pergunta(texto_pergunta="What's new?",
   data_publicacao=timezone.now())
>>> p.save()

@Acesso a atributos da instância criada
>>> p.texto_pergunta
"What's new?"

@Alteração no texto da pergunta
>>> p.texto_pergunta = "What's up?"
>>> p.save()
```

SQL puro no Django

As funcionalidades oferecidas pelas classes de modelo do Django são práticas e auxiliam o desenvolvimento rápido, porém são limitadas e não permitem consultas muito sofisticadas. Por conta disso, Django oferece um jeito de escrever consultas em SQL puro que retornam instâncias

de um modelo e um jeito de executar comandos SQL puro. Vale observar que os parâmetros usados no SQL puro devem ser devidamente escapados para proteção a ataques de SQL-injection.

Para realizar consultas em SQL que retornam modelos pode-se usar o método raw, como mostrado neste exemplo simples:

Listing 4.14: Modelos no shell

```
>>> for p in Pergunta.objects.raw('SELECT * FROM meuapp_pergunta'):
>>>     print(p)
```

Para realizar consultas SQL cujo retorno não é mapeado para um modelo ou executar comandos SQL pode-se usar o método cursor do objeto django.db.connection, que representa a conexão com o banco de dados. Segue um exemplo de consulta e de execução de comando usando o método cursor:

Listing 4.15: cursorexemplo.py

```
from django.db import connection

def my_custom_sql(self):
    with connection.cursor() as cursor:
        cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
        cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
        row = cursor.fetchone()

    return row
```

Maiores detalhes sobre o método raw e o método cursor encontram-se nesta seção da documentação do Django [5].

Apresentação

O Django também oferece uma camada de apresentação para o usuário final na forma de templates. Um template combina elementos HTML com uma sintaxe própria para gerar conteúdo HTML dinamicamente.

Para utilizar templates, deve-se configurar o arquivo settings.py do projeto alterando o item TEMPLATES. Com o projeto configurado, os arquivos de templates devem ser criados dentro do projeto para serem importados e renderizados pelas views utilizando os parâmetros desejados. Detalhes sobre a configuração, sintaxe dos templates e exemplos de uso podem ser encontrados nesta seção da documentação do Django. [6]

Referências Bibliográficas

[1] Ramez Elmasri and Shamkant B. Navathe. *Database Systems, 7th Ed.* Pearson, 2015.

[2] Django Software Foundation. Django documentation. https://docs.djangoproject.com/en/1.11/. Accessed: 09/09/17.

[3] Django Software Foundation. Django documentation : Making queries. https://docs.djangoproject.com/en/1.11/topics/db/queries/. Accessed: 10/09/17.

[4] Django Software Foundation. Django documentation : Models. https://docs.djangoproject.com/en/1.11/topics/db/models/. Accessed: 10/09/17.

[5] Django Software Foundation. Django documentation : Performing raw sql queries. https://docs.djangoproject.com/en/1.11/topics/db/sql/. Accessed: 10/09/17.

[6] Django Software Foundation. Django documentation : Templates. https://docs.djangoproject.com/en/1.11/topics/templates/. Accessed: 10/09/17.

[7] Federico Di Gregorio and Daniele Varrazzo. Psycopg – postgresql database adapter for python (documentation). http://initd.org/psycopg/docs/. Accessed: 05/09/17.

[8] The PostgreSQL Global Development Group. Postgresql 9.5.3 documentation. https://www.postgresql.org/docs/9.5/static/index.html. Accessed: 18/06/16.

[9] Unsigned Integer Limited. Distrowatch.com: Put the fun back into computing. use linux, bsd. https://distrowatch.com/. Accessed: 05/09/17.