

INTERFACE GRÁFICA E BANCO DE DADOS

Visão geral

Conceitos

Até o dado momento, vimos como criar uma aplicação com componentes de interface gráfica e como interagir com um banco de dados.

Neste módulo, criaremos uma aplicação que integra tanto elementos de interface gráfica quanto operações com banco de dados.

A nossa aplicação implementa as operações CRUD, que são: inserção, seleção, atualização e exclusão de dados.

O usuário fará a entrada de dados mediante componentes de caixas de texto (widget entry) e confirmará a ação que deseja quando pressionar o botão correspondente.

Além disso, os dados que estão armazenados no banco são exibidos em um componente do tipo grade (widget treeview). O usuário tem a possibilidade de selecionar um registro na grade, o qual será exibido nas caixas de texto, onde poderá ser modificado ou excluído.

Observe a interface gráfica da nossa aplicação:

Bem Vindo a Aplicação de Banco de Dados

Código do Produto:

Nome do Produto

Preço

	Código	Nome	Preço
	10	produto_1	990.73
	11	produto_2	957.78
	12	produto_3	365.53
	13	produto_4	658.71
	14	produto_5	140.69
	15	produto_6	417.61
	16	produto_7	842.33
	17	produto_8	195.5
	18	produto_9	450.32
	19	produto_10	209.48

Interface da aplicação com Tkinter.

Atenção!

Perceba que alguns dados já estão armazenados no banco e são exibidos na grade. Veja que o usuário selecionou o “produto_7” na grade e seus dados estão exibidos nas caixas de texto.

Criação de tabelas e geração de dados

Criação de tabelas no Postgresql

A primeira ação a ser feita é a criação da tabela.

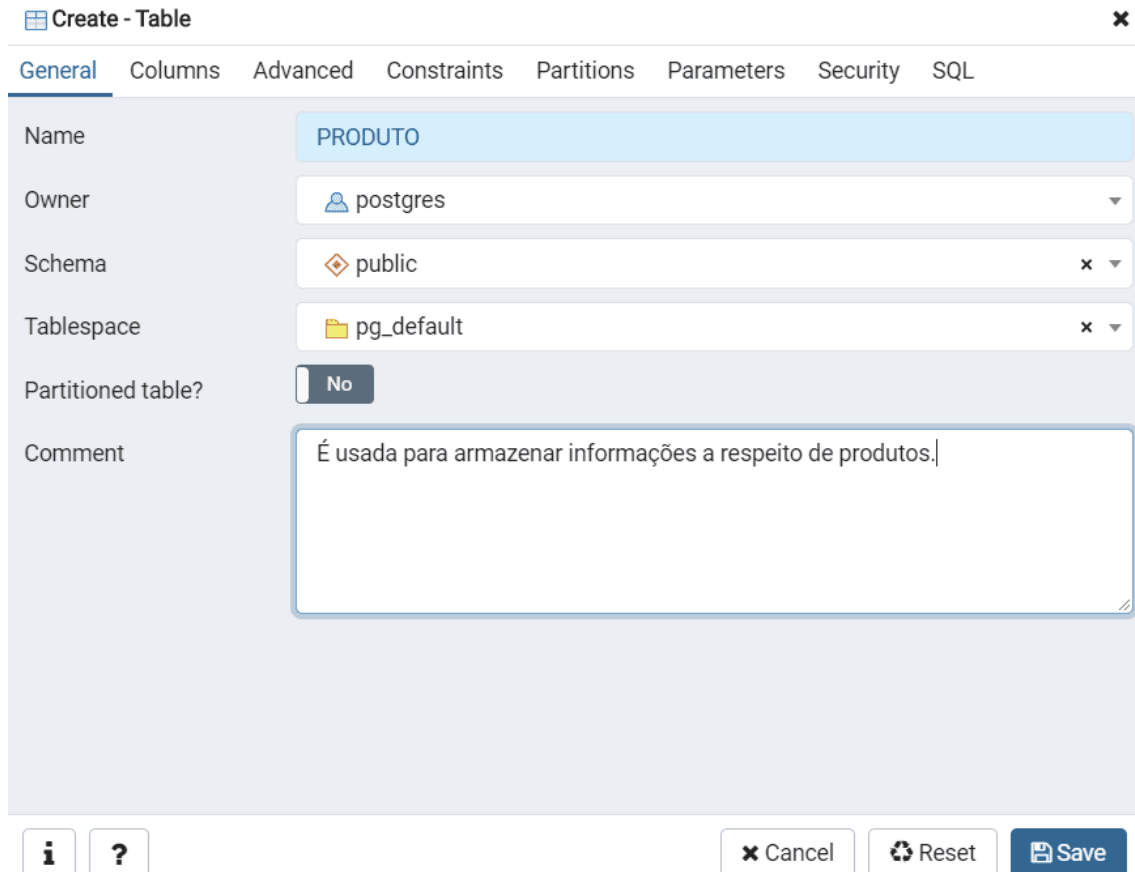
Comentário

No caso do nosso sistema, vamos criar a tabela Produto, que tem três campos: “CODIGO”, “NOME” e “PREÇO”.

Estamos usando o PostgreSQL para gerenciar nossos dados. O PostgreSQL é um sistema gerenciador de banco de dados de licença gratuita e é considerado bastante robusto para aplicações de um modo geral.

Com a ferramenta pgAdmin, o desenvolvedor pode criar a tabela produto.

Veja como criar uma tabela no pgAdmin:



The screenshot shows the 'Create - Table' dialog box in pgAdmin, with the 'General' tab selected. The dialog has a title bar with a close button (X) and a tab bar with 'General', 'Columns', 'Advanced', 'Constraints', 'Partitions', 'Parameters', 'Security', and 'SQL'. The 'General' tab contains the following fields:

- Name:** A text field containing 'PRODUTO'.
- Owner:** A dropdown menu showing 'postgres' with a user icon.
- Schema:** A dropdown menu showing 'public' with a diamond icon and a close button (X).
- Tablespace:** A dropdown menu showing 'pg_default' with a folder icon and a close button (X).
- Partitioned table?:** A toggle switch set to 'No'.
- Comment:** A text area containing 'É usada para armazenar informações a respeito de produtos.'

At the bottom of the dialog, there are three buttons: 'Cancel' (with an X icon), 'Reset' (with a circular arrow icon), and 'Save' (with a floppy disk icon). To the left of these buttons are two small icons: an information icon (i) and a help icon (?).

Criação de tabela no pgAdmin.

O campo "owner" é o usuário proprietário do banco de dados que terá acesso à tabela "PRODUTO". No caso, usamos o usuário padrão do PostgreSQL.







Em seguida, precisamos criar os campos tabelas. Para isso, selecionamos a opção "Columns" da imagem anterior. Será exibida a tela da imagem seguinte, na qual podemos adicionar as colunas por meio da opção "+" no canto superior direito da tela.



Create - Table ✕

General Columns Advanced Constraints Partitions Parameters Security SQL

Inherited from table(s)

Columns +

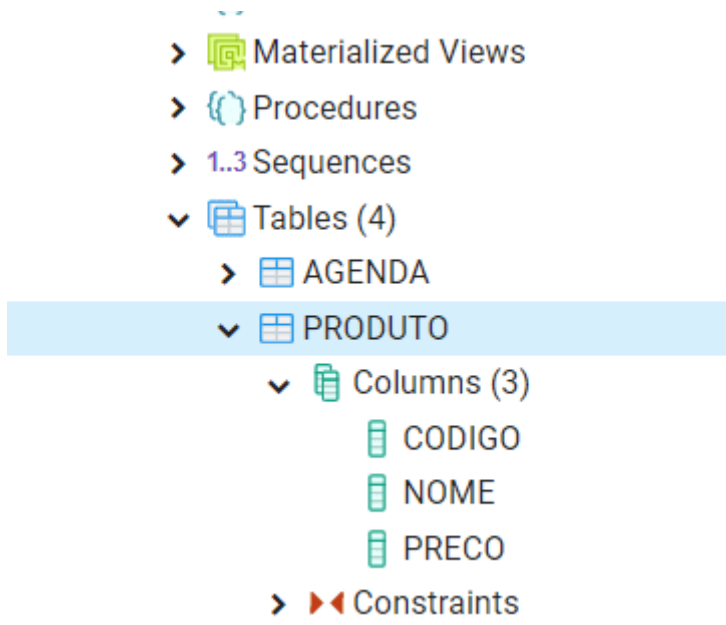
	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
 	CODIGO	integer			<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes
 	NOME	text			<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
 	PRECO	real			<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No

  ✕ Cancel

Criação dos campos da tabela no pgAdmin.

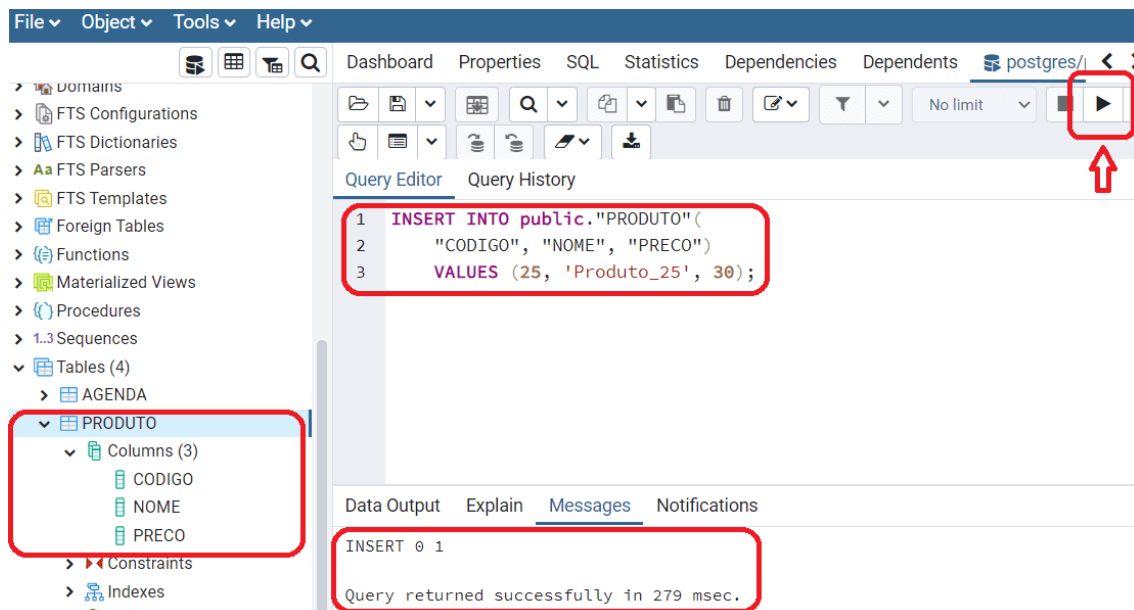
Perceba que o campo “CODIGO” é a chave primária da tabela e, além disso, todos os campos não podem ser vazios.

Agora, salvamos a tabela e, finalmente, ela é criada no banco de dados. Na imagem a seguir, podemos ver a tabela no pgAdmin.



Visualização da tabela no pgAdmin.

O pgAdmin é bastante útil para fazermos operações sobre as tabelas. Na imagem seguinte, mostramos como inserir dados na tabela “PRODUTO”.



Inserção de dados no pgAdmin.

Para executar o comando de inserção, basta pressionar a seta no canto direito superior da tela.

Outra forma de criar uma tabela é mediante o próprio python, com o uso da biblioteca "psycopg2".

Na próxima imagem, mostramos o código para criação da tabela "PRODUTO". O código para criação da tabela pode ser baixado clicando aqui.

```
8 import psycopg2
9 conn = psycopg2.connect(database = "postgres", user = "postgres",
10                        password = "senha123",
11                        host = "127.0.0.1", port = "5432")
12 print("Conexão com o Banco de Dados feita com Sucesso!")
13 cur = conn.cursor()
14 cur.execute('CREATE TABLE PRODUTO(CODIGO INT PRIMARY KEY NOT NULL,
15                                NOME TEXT NOT NULL,
16                                PRECO REAL NOT NULL);')
17 print("Tabela criada com sucesso!")
18 conn.commit()
19 conn.close()
```

Criação de tabela no python.

Agora, as principais linhas serão analisadas.

- Linha 8 - Importamos a biblioteca psycopg2.
- Linha 9 - Fazemos a conexão ao banco de dados "postgres" com o usuário "postgres", senha "senha123", host local (127.0.0.1) e porta "5432".
- Linha 13 - Abrimos o cursor. Lembre-se de que é com o cursor que fazemos as operações no banco de dados.
- Linha 14 a 16 - Executamos o comando sql "Create Table" para criar a tabela "PRODUTO" com os campos "CODIGO", "NOME" e "PRECO".
- Linha 18 - Executamos o comando "commit" para confirmar o comando sql.
- Linha 19 - Fechamos a conexão com o banco de dados.

Geração de dados aleatórios

Uma boa forma de iniciar o projeto é inserindo dados aleatórios na tabela. Para isso, vamos usar o pacote “faker”, que é bastante útil para gerar dados aleatórios.

Na imagem a seguir, mostramos o código para gerar os dados aleatórios.

```
14 #-----
15 from faker import Faker
16 import psycopg2
17 #-----
18 conn = psycopg2.connect(database = "postgres", user = "postgres",
19                        password = "senha123", host = "127.0.0.1", port = "5432")
20 print ("Conexão aberta com sucesso!")
21 cursor = conn.cursor()
22 fake = Faker('pt_BR')
23 #-----
24 n=10
25 for i in range(n):
26     codigo = i+10
27     nome = 'produto_'+str(i+1)
28     preco = fake.pyfloat(left_digits=3, right_digits=2, positive=True,
29                        min_value=5, max_value=1000)
30     print(preco)
31     print(nome)
32
33
34 comandoSQL = """ INSERT INTO public."PRODUTO" ("CODIGO", "NOME", "PRECO")
35 VALUES (%s,%s,%s) """
36 registro = (codigo, nome, preco)
37 cursor.execute(comandoSQL, registro)
38 #-----
39 conn.commit()
40 print ("Inserção realizada com sucesso!");
41 conn.close()
42 #-----
```

Geração de dados aleatórios.

- Linha 15 - Importamos a biblioteca “faker”.
- Linha 22 - Instanciamos um objeto para gerar dados aleatórios no “português do Brasil”.
- Linha 25 a 31 - Geramos os dados aleatórios que serão inseridos na tabela.
- Linha 34 e 35 - Montamos o comando sql. Mas nesse momento, trata-se apenas de um texto (string) que deve seguir uma sintaxe cuidadosa: três aspas duplas no início e no final do texto. Além disso, perceba os “%s” no trecho do “VALUES” que serão usados para entrar com os dados das variáveis.
- Linha 36 - Criamos um registro com as variáveis que serão armazenadas na tabela.
- Linha 37 - Aplicamos o comando “execute”, que executa o comando sql com a entrada de dados que definimos na variável registro.

Comentário

Esse exemplo é simples, mas ilustra testes que podem ser feitos logo no começo do projeto que são úteis para validar a entrada de dados, além de ser útil para os desenvolvedores aprenderem mais sobre o próprio sistema.

Interação entre o sistema e o banco de dados

Interação com o banco de dados

Nesta seção, apresentaremos a parte do sistema responsável pelas operações CRUD.

O sistema foi desenvolvido usando programação orientada a objetos. O arquivo do programa foi salvo com o nome `crud.py`.

A classe com os métodos para realizar as operações para interagir com o banco de dados é a `AppBD`, conforme podemos ver na próxima imagem.

```
7  #-----
8  #Essa classe possui métodos CRUD
9  #-----
10 import psycopg2
11
12 class AppBD:
13     def __init__(self):
14         print('Método Construtor')
15
16     def abrirConexao(self):
17         try:
18             self.connection = psycopg2.connect(user="postgres",
19                                                 password="senha123",
20                                                 host="127.0.0.1",
21                                                 port="5432",
22                                                 database="postgres")
23         except (Exception, psycopg2.Error) as error :
24             if(self.connection):
25                 print("Falha ao se conectar ao Banco de Dados", error)
```

Classe para operações CRUD.

Antes de continuar, lembre-se de que a indentação (espaçamento) faz parte da sintaxe do python.

- Linha 12 - Declaramos a classe `AppBD`.
- Linha 13 - Implementamos o construtor da classe, que é o método que é chamado logo que um objeto da classe `AppBD` for instanciado.
- Linha 16 a 25 - Implementamos o método para abrir a conexão. Perceba as cláusulas `try` e `except`. Isso é fundamental para criar um programa confiável, ou seja, com tolerância a falhas, pois o programa tenta seguir o fluxo normal de execução, ou seja, abrir a conexão com o banco de dados.

Comentário

Caso ocorra algum problema, ao invés de o programa interromper a execução e exibir uma mensagem de erro, revelando vulnerabilidades do sistema que podem ser exploradas por um atacante, ele vai exibir uma mensagem amigável para o usuário, no caso, será `Falha ao se conectar ao Banco de Dados`.

O próximo método é o que faz consulta no banco de dados:

```

26 #-----
27 #Selecionar todos os Produtos
28 #-----
29 def selecionarDados(self):
30     try:
31         self.abrirConexao()
32         cursor = self.connection.cursor()
33
34         print("Selecionando todos os produtos")
35         sql_select_query = """select * from public."PRODUTO" """
36
37
38         cursor.execute(sql_select_query)
39         registros = cursor.fetchall()
40         print(registros)
41
42
43     except (Exception, psycopg2.Error) as error:
44         print("Error in select operation", error)
45
46     finally:
47         # closing database connection.
48         if (self.connection):
49             cursor.close()
50             self.connection.close()
51         print("A conexão com o PostgreSQL foi fechada.")
52     return registros

```

Seleção de dados.

Vamos destacar os principais pontos do método “selecionarDados”.

- Linha 35 - Montamos a instrução de consulta do sql.
- Linha 38 - Executamos a instrução sql.
- Linha 39 - Recuperamos as linhas que retornaram da consulta sql.
- Linha 52 - Retornamos os registros para quem faz a chamada para o método “selecionarDados”.

Agora, vamos explicar o método para fazer a inserção de dados. O código do método está na imagem a seguir.

```

53 #-----
54 #Inserir Produto
55 #-----
56 def inserirDados(self, codigo, nome, preco):
57     try:
58         self.abrirConexao()
59         cursor = self.connection.cursor()
60         postgres_insert_query = """ INSERT INTO public."PRODUTO"
61         ("CODIGO", "NOME", "PRECO") VALUES (%s,%s,%s)"""
62         record_to_insert = (codigo, nome, preco)
63         cursor.execute(postgres_insert_query, record_to_insert)
64         self.connection.commit()
65         count = cursor.rowcount
66         print(count, "Registro inserido com sucesso na tabela PRODUTO")
67     except (Exception, psycopg2.Error) as error :
68         if(self.connection):
69             print("Falha ao inserir registro na tabela PRODUTO", error)
70     finally:
71         #closing database connection.
72         if(self.connection):
73             cursor.close()
74             self.connection.close()
75         print("A conexão com o PostgreSQL foi fechada.")
76

```

Inserção de dados.

- Linha 56 - Implementamos a função “inserirDados” e passamos como parâmetros os “codigo”, “nome” e “preco” que serão inseridos na tabela. Além disso, passamos o parâmetro “self”, que é usado para fazer referência aos atributos e métodos da própria classe.
- Linha 60 e 61 - Montamos a instrução sql para fazer a inserção dos dados.
- Linha 62 - Montamos o registro que será inserido na tabela.
- Linha 63 - Executamos a instrução sql para fazer a inserção do registro da variável “record_to_insert”.

Agora, vamos analisar o método responsável pela atualização de dados. O código é apresentado na imagem seguinte.

```

77  #-----
78  #Atualizar Produto
79  #-----
80  def atualizarDados(self, codigo, nome, preco):
81  try:
82      self.abrirConexao()
83      cursor = self.connection.cursor()
84
85      print("Registro Antes da Atualização ")
86      sql_select_query = """select * from public."PRODUTO"
87      where "CODIGO" = %s"""
88      cursor.execute(sql_select_query, (codigo,))
89      record = cursor.fetchone()
90      print(record)
91      # Atualizar registro
92      sql_update_query = """Update public."PRODUTO" set "NOME" = %s,
93      "PRECO" = %s where "CODIGO" = %s"""
94      cursor.execute(sql_update_query, (nome, preco, codigo))
95      self.connection.commit()
96      count = cursor.rowcount
97      print(count, "Registro atualizado com sucesso! ")
98      print("Registro Depois da Atualização ")
99      sql_select_query = """select * from public."PRODUTO"
100     where "CODIGO" = %s"""
101     cursor.execute(sql_select_query, (codigo,))
102     record = cursor.fetchone()
103     print(record)
104 except (Exception, psycopg2.Error) as error:
105     print("Erro na Atualização", error)
106 finally:
107     # closing database connection.
108     if (self.connection):
109         cursor.close()
110         self.connection.close()
111         print("A conexão com o PostgreSQL foi fechada.")
112

```

Atualização dos dados.

- Linha 80 - Implementamos a função “atualizarDados” e passamos como parâmetros os “codigo”, “nome” e “preco” que serão modificados na tabela.
- Linha 92 e 93 - Montamos a instrução sql para fazer a modificação dos dados da tabela.
- Linha 94 - Executamos a instrução sql para fazer a modificação do registro de acordo com a tupla “(nome, preco, codigo)”.

Agora, vamos analisar o método que trata da exclusão de dados. O código é apresentado na imagem a seguir.

```

113 #-----
114 #Excluir Produto
115 #-----
116 def excluirDados(self, codigo):
117     try:
118         self.abrirConexao()
119         cursor = self.connection.cursor()
120         # Atualizar registro
121         sql_delete_query = """Delete from public."PRODUTO"
122         where "CODIGO" = %s"""
123         cursor.execute(sql_delete_query, (codigo, ))
124
125         self.connection.commit()
126         count = cursor.rowcount
127         print(count, "Registro excluído com sucesso! ")
128     except (Exception, psycopg2.Error) as error:
129         print("Erro na Exclusão", error)
130     finally:
131         # closing database connection.
132         if (self.connection):
133             cursor.close()
134             self.connection.close()
135             print("A conexão com o PostgreSQL foi fechada.")
136

```

Exclusão de dados.

- Linha 116 - Implementamos a função “excluirDados” e passamos o “codigo” do produto que será excluído da tabela.
- Linha 121 e 122 - Montamos a instrução sql para fazer a exclusão dos dados da tabela.
- Linha 123 - Executamos a instrução sql para fazer a exclusão do registro de acordo com o parâmetro “codigo” passado para o método.

GUI: Interação com o usuário

Interface gráfica

Nesta seção, vamos apresentar a parte do sistema responsável pela interação com o usuário por meio de uma interface gráfica.

Do mesmo modo que a classe “AppBD”, esse programa também foi desenvolvido em programação orientada a objetos.

O arquivo do programa foi salvo com o nome “aplicacaoCRUD.py”. A classe com os atributos e métodos para trabalhar com a interface gráfica e interagir com a classe responsável pelas operações com o banco de dados é a “PrincipalBD”, conforme podemos ver na imagem a seguir.

```

9 import tkinter as tk
10 from tkinter import ttk
11 import crud as crud
12
13 class PrincipalBD:
14     def __init__(self, win):
15         self.objBD = crud.AppBD()
16         #componentes
17         self.lbCodigo=tk.Label(win, text='Código do Produto:')
18         self.lblNome=tk.Label(win, text='Nome do Produto')
19         self.lblPreco=tk.Label(win, text='Preço')
20
21         self.txtCodigo=tk.Entry(bd=3)
22         self.txtNome=tk.Entry()
23         self.txtPreco=tk.Entry()
24         self.btnCadastrar=tk.Button(win, text='Cadastrar', command=self.fCadastrarProduto)
25         self.btnAtualizar=tk.Button(win, text='Atualizar', command=self.fAtualizarProduto)
26         self.btnExcluir=tk.Button(win, text='Excluir', command=self.fExcluirProduto)
27         self.btnLimpar=tk.Button(win, text='Limpar', command=self.fLimparTela)

```

Classe principal.

- Linha 9 - Importamos a biblioteca Tkinter para interagir com os componentes gráficos.
- Linha 10 - Importamos o módulo ttk para podermos trabalhar com o componente “TreeView”, que foi usado como uma grade para exibir os dados armazenados na tabela “PRODUTO”.
- Linha 14 - Implementamos o construtor (__init__) da classe PrincipalBD, que será chamado logo que um objeto do tipo PrincipalBD for instanciado.
- Linha 17 a 23 - Instanciamos os componentes rótulos (“label”) e caixas de texto (“entry”).
- Linha 24 a 27 - Instanciamos os componentes botões (“button”), que vão acionar as operações CRUD.

Atenção!

Observe que os métodos “fCadastrarProduto”, “fAtualizarProduto”, “fExcluirProduto” e “fLimparProduto” estão vinculados aos botões, ou seja, quando o usuário pressionar um botão, o respectivo método será chamado.

O construtor ainda possui mais duas partes. Uma delas é responsável por instanciar e configurar o componente “TreeView”, conforme podemos ver na próxima imagem.

```

28 #----- Componente TreeView -----
29 self.dadosColunas = ("Código", "Nome", "Preço")
30
31 self.treeProdutos = ttk.Treeview(win,
32                                 columns=self.dadosColunas,
33                                 selectmode='browse')
34
35 self.verscrlbar = ttk.Scrollbar(win,
36                                orient="vertical",
37                                command=self.treeProdutos.yview)
38 self.verscrlbar.pack(side='right', fill='x')
39
40 self.treeProdutos.configure(yscrollcommand=self.verscrlbar.set)
41
42 self.treeProdutos.heading("Código", text="Código")
43 self.treeProdutos.heading("Nome", text="Nome")
44 self.treeProdutos.heading("Preço", text="Preço")
45
46 self.treeProdutos.column("Código",minwidth=0,width=100)
47 self.treeProdutos.column("Nome",minwidth=0,width=100)
48 self.treeProdutos.column("Preço",minwidth=0,width=100)
49
50 self.treeProdutos.pack(padx=10, pady=10)
51
52 self.treeProdutos.bind("<<TreeviewSelect>>",
53                        self.apresentarRegistrosSelecioneados)

```

Configuração do componente Treeview.

Observe nas linhas 52 e 53 que o método “apresentarRegistrosSelecionados” é vinculado à instância do componente “TreeView”. Esse método será explicado mais à frente.

E a outra posiciona os componentes na tela, conforme podemos ver na imagem seguinte.

```
54 #-----
55 #posicionamento dos componentes na janela
56 #-----
57 self.lbCodigo.place(x=100, y=50)
58 self.txtCodigo.place(x=250, y=50)
59
60 self.lblNome.place(x=100, y=100)
61 self.txtNome.place(x=250, y=100)
62
63 self.lblPreco.place(x=100, y=150)
64 self.txtPreco.place(x=250, y=150)
65
66 self.btnCadastrar.place(x=100, y=200)
67 self.btnAtualizar.place(x=200, y=200)
68 self.btnExcluir.place(x=300, y=200)
69 self.btnLimpar.place(x=400, y=200)
70
71 self.treeProdutos.place(x=100, y=300)
72 self.verscrlbar.place(x=805, y=300, height=225)
73 self.carregarDadosIniciais()
```

Configuração de componentes.

Observe que, na linha 73, fazemos chamada para o método “carregarDadosIniciais”. Mais à frente, vamos explicá-lo com mais detalhes.

Agora, vamos analisar o método “apresentarRegistrosSelecionados”, conforme podemos ver na imagem seguinte.

```
74 #-----
75 def apresentarRegistrosSelecionados(self, event):
76     self.fLimparTela()
77     for selection in self.treeProdutos.selection():
78         item = self.treeProdutos.item(selection)
79         codigo,nome,preco = item["values"][0:3]
80         self.txtCodigo.insert(0, codigo)
81         self.txtNome.insert(0, nome)
82         self.txtPreco.insert(0, preco)
```

Exibir dados selecionados no Treeview.

Este método exibe os dados selecionados na grade (componente “TreeView”) nas caixas de texto, de modo que o usuário possa fazer alterações, ou exclusões sobre eles.

- Linha 76 - Fazemos a chamada para a função “fLimparTela”, que limpa o conteúdo das caixas de texto.
- Linha 77 - Obtemos os registros que foram selecionados na grade de registros.
- Linha 79 - Os dados do item selecionados são, agora, associados às variáveis “codigo”, “nome” e “preco”.
- Linha 80 a 82 - Os valores das variáveis são associados às caixas de texto.

Agora, vamos analisar o método “carregarDadosIniciais”, que é apresentado na imagem a seguir.

```

83 #-----
84 def carregarDadosIniciais(self):
85     try:
86         self.id = 0
87         self.iid = 0
88         registros=self.objBD.selecionarDados()
89         print("***** dados dsponíveis no BD *****")
90         for item in registros:
91             codigo=item[0]
92             nome=item[1]
93             preco=item[2]
94             print("Código = ", codigo)
95             print("Nome = ", nome)
96             print("Preço = ", preco, "\n")
97
98             self.treeProdutos.insert('', 'end',
99                                     iid=self.iid,
100                                     values=(codigo,
101                                             nome,
102                                             preco))
103             self.iid = self.iid + 1
104             self.id = self.id + 1
105         print('Dados da Base')
106     except:
107         print('Ainda não existem dados para carregar')

```

Carregar dados da tabela no Treeview.

Este método carrega os dados que já estão armazenados na tabela para serem exibidos na grade de dados (componente “TreeView”).

- Linha 86 e 87 - Os atributos “id” e “iid” são iniciados com valor 0. Eles são necessários para gerenciar o componente “TreeView”.
- Linha 88 - É feita a chamada para o método “selecionarDados” que está na classe “AppBD”. Ele recupera todos os registros armazenados na tabela.
- Linha 91 e 93 - Obtemos os valores dos registros e associamos às respectivas variáveis.
- Linha 98 a 102 - Os dados são adicionados ao componente “TreeView”.

Agora, vamos apresentar o método “fLerCampos”, conforme podemos ver na imagem abaixo.

```

108 #-----
109 #LerDados da Tela
110 #-----
111 def fLerCampos(self):
112     try:
113         print("***** dados dsponíveis *****")
114         codigo = int(self.txtCodigo.get())
115         print('codigo', codigo)
116         nome=self.txtNome.get()
117         print('nome', nome)
118         preco=float(self.txtPreco.get())
119         print('preco', preco)
120         print('Leitura dos Dados com Sucesso!')
121     except:
122         print('Não foi possível ler os dados.')
123     return codigo, nome, preco

```

Entrada de dados.

Este método lê os dados que estão nas caixas de texto e os retorna para quem faz a chamada.

Por exemplo, na linha 114, a variável “codigo” recebe o valor da caixa de texto “txtCodigo” depois que ele é convertido para um valor do tipo “inteiro”.

Na linha 123, as variáveis “codigo”, “nome” e “preco” retornam para quem faz a chamada do método.

Agora, vamos apresentar o método “fCadastrarProduto”, conforme podemos ver na imagem a seguir.

```
124 -----
125 #Cadastrar Produto
126 -----
127 def fCadastrarProduto(self):
128     try:
129         print("***** dados dsponiveis *****")
130         codigo, nome, preco= self.fLerCampos()
131         self.objBD.inserirDados(codigo, nome, preco)
132         self.treeProdutos.insert(' ', 'end',
133                                 iid=self.iid,
134                                 values=(codigo,
135                                       nome,
136                                       preco))
137         self.iid = self.iid + 1
138         self.id = self.id + 1
139         self.fLimparTela()
140         print('Produto Cadastrado com Sucesso!')
141     except:
142         print('Não foi possível fazer o cadastro.')
```

Inserção de dados no banco.

Este método tem como objetivo fazer a inserção dos dados na tabela “PRODUTOS”.

- Linha 130 - Os dados digitados nas caixas de texto são recuperados nas variáveis “codigo”, “nome” e “preco”.
- Linha 131 - Fazemos a chamada ao método “inserirDados”, que fará a inserção dos dados na tabela “PRODUTO”.
- Linha 132 a 136 - Os dados são inseridos no componente grade (“TreeView”).

Agora, vamos analisar o método “fAtualizarProduto”, conforme podemos ver na próxima imagem.

```
143 -----
144 #Atualizar Produto
145 -----
146 def fAtualizarProduto(self):
147     try:
148         print("***** dados dsponiveis *****")
149         codigo, nome, preco= self.fLerCampos()
150         self.objBD.atualizarDados(codigo, nome, preco)
151         #recarregar dados na tela
152         self.treeProdutos.delete(*self.treeProdutos.get_children())
153         self.carregarDadosIniciais()
154         self.fLimparTela()
155         print('Produto Atualizado com Sucesso!')
156     except:
157         print('Não foi possível fazer a atualização.')
```

Atualização de dados no banco.

O objetivo deste método é atualizar os dados que o usuário selecionou na grade de dados (o componente “TreeView”).

- Linha 149 - Os dados selecionados da grade de dados são recuperados nas variáveis “codigo”, “nome” e “preco”.

- Linha 150 - Chamamos a função “atualizarDados”, que fará as modificações dos dados na tabela “PRODUTO”.
- Linha 152 - Os dados selecionados são removidos da grade de dados.
- Linha 153 - Fazemos a chamada ao método “carregarDadosIniciais” para recarregar a grade de dados com os dados da tabela.

Agora, vamos analisar o método “fExcluirProduto”, conforme podemos ver na imagem seguinte.

```

158 #-----
159 #Excluir Produto
160 #-----
161 def fExcluirProduto(self):
162     try:
163         print("***** dados disponíveis *****")
164         codigo, nome, preco= self.fLerCampos()
165         self.objBD.excluirDados(codigo)
166         #recarregar dados na tela
167         self.treeProdutos.delete(*self.treeProdutos.get_children())
168         self.carregarDadosIniciais()
169         self.fLimparTela()
170         print('Produto Excluído com Sucesso!')
171     except:
172         print('Não foi possível fazer a exclusão do produto.')
```

Exclusão de dados no banco.

O objetivo deste método é excluir os dados que o usuário selecionou na grade de dados (o componente “TreeView”).

- Linha 164 - Os dados selecionados da grade de dados são recuperados nas variáveis “codigo”, “nome” e “preco”.
- Linha 165 - Chamamos a função “excluirDados”, que excluirá os dados da tabela “PRODUTO”.
- Linha 167 - Os dados selecionados são removidos da grade de dados.
- Linha 168 - Fazemos a chamada ao método “carregarDadosIniciais” para recarregar a grade de dados com os dados da tabela.

Agora, vamos analisar o método “fLimparTela”, conforme podemos ver na imagem a seguir.

```

173 #-----
174 #Limpar Tela
175 #-----
176 def fLimparTela(self):
177     try:
178         print("***** dados disponíveis *****")
179         self.txtCodigo.delete(0, tk.END)
180         self.txtNome.delete(0, tk.END)
181         self.txtPreco.delete(0, tk.END)
182         print('Campos Limpos!')
183     except:
184         print('Não foi possível limpar os campos.')
```

Limpar os componentes da tela.

Este método limpa o conteúdo das caixas de texto, conforme podemos ver nas linhas 179 a 181.

Por fim, apresentamos o programa principal que vai iniciar a execução do sistema, conforme podemos ver na próxima imagem.

```
185 #-----
186 #Programa Principal
187 #-----
188 janela=tk.Tk()
189 principal=PrincipalBD(janela)
190 janela.title('Bem Vindo a Aplicação de Banco de Dados')
191 janela.geometry("820x600+10+10")
192 janela.mainloop()
```

Programa principal.

- Linha 188 - Instanciamos o objeto raiz da aplicação gráfica que chamamos de “janela”.
- Linha 189 - Instanciamos o objeto principal que vai gerenciar a execução da aplicação.
- Linha 190 - Escrevemos uma mensagem para o título da janela.
- Linha 192 - Configuramos as dimensões da janela.
- Linha 192 - Chamamos o método “mainloop”, que coloca a aplicação para executar até que o usuário interrompa a execução.