Notas do Curso por

Sanderson C. Ribeiro

Python & Banco de Dados

Integração e Aplicações Práticas

LInEA

INCT do e-Universo

Sumário

1 Python e Banco de Dados

Neste curso, vamos abordar a integração que podemos realizar entre a linguagem de programação Python e bancos de dados. Além disso, exploraremos aplicações utilizando dados provenientes dos bancos de dados hospedados na infraestrutura do LIneA (Laboratório Interinstitucional de e-Astronomia).

O curso será realizado na seguinte ordem:

Parte 1: Revisão

1. **Revisão de Python**

   a. Revisão dos conceitos de Python necessários para manipulação de dados em banco (*loops*, listas, Dicionários);

   b. Funções úteis para manipulação de *strings* e dados estruturados.

2. **Revisão de Banco de Dados**

   a. Exemplos e vantagens de usar um SGBD para manipulação de grandes volumes de dados;

   b. Chaves primárias e chaves estrangeiras;

   c. Exemplo de um modelo simples de banco de dados;

   d. Revisão dos comandos básicos: `CREATE TABLE`, `ALTER TABLE`, `INSERT`, `UPDATE`, `DELETE`, `SELECT`.

2 Revisão de Python

2.1 *Loops*

Os *loops* são estruturas de repetição que permitem que um bloco de comandos seja executado diversas vezes. Em Python temos dois tipos de repetição:

- **Repetição condicional:** executa um bloco de código **enquanto** uma condição lógica for verdadeira (**while**);
- **Repetição contável:** executa um bloco de código **por** um número predeterminado de vezes (**for**). Desta forma conseguimos especificar quantas vezes um determinado bloco de código será repetido.

Vamos analisar alguns exemplos de blocos de códigos nos quais adotamos essas estruturas de repetição. Primeiramente vamos começar entendendo de forma resumida o a estrutura de condição que chamamos de **while**.

Exemplo 2.1 (Bomba)**.** Considere o seguinte algoritmo que registra o funcionamento de uma bomba hipotética:

```
numero = int(input('Digite um número (inteiro):'))
while numero>0:
  numero = numero-1
  print(numero)
print('Boom!!!')
```

Nesse caso o usuário irá digitar um número inteiro e enquanto `numero>0`, o bloco de código continuará repetindo.

Na prática, acontece que esse número será decrescido de uma unidade inteira e será impresso na tela o seu resultado. Por exemplo, digamos que o número fornecido pelo usuário seja 24. Na tela será impresso o número 23 que agora se torna a variável `numero`. O bloco vai executar novamente e vai decrescer em uma unidade o número 23, imprimindo na tela o número 22 que se tornará então a variável `numero`, e assim por diante até que o número seja 1, pois quando for decrescido de uma unidade ele será igual a 0, fazendo com que a condição para que o *loop* continue se torne falsa já que a variável `numero` será igual a 0 e não maior que 0. Assim o *loop* cessa e é impresso na tela a palavra Boom!!!

Exemplo 2.2 (*Loop* infinito)**.** Uma coisa que precisamos prestar atenção quando trabalhamos com a estrutura de repetição `while` envolve uma condição que em algum momento é possível de ocorrer, caso contrário caímos no que chamamos de *loop* infinito, na área de programação, ou seja uma repetição que nunca se torna falsa, sempre é verdadeira. Um exemplo de algoritmo em que ocorre isso é representado a seguir:

```
i=1
while True:
  i=i+1
  print(i)
```

Nesse caso temos um bloco de código que nos diz que **enquanto** for **verdadeiro** o bloco seja executado, i.e., nunca teremos uma condição falsa já que foi definido que a condição é `True` sempre. Assim temos que a execução deste bloco será infinita ou até que se ocorra o seu interrompimento. Ressaltamos a atenção para evitar esses casos. **Evite forçar *loop* infinito.**

O **for** pode ser usado como estrutura de repetição contável. Geralmente usamos ele ou com uma lista ou com um contador, uma função chamada **range**, na qual podemos especificar um intervalo com início, fim e incremento. Essa função repete de início até fim-1, com passo incremento. Vamos compreender melhor como ela funciona no exemplo a seguir.

Exemplo 2.3 (Faixa de valores)**.** Com a função **range** conseguimos estabelecer o início e o fim de um intervalo, e o quanto que os números dentro desse intervalo vão crescer, ou seja o passo entre cada elemento do intervalo. Considere o bloco de código a seguir:

```
for i in range (0,5,1):
  print(i)
```

Neste caso temos como **início** do intervalo o 0, que é **opcional** nesse caso, já que quando omitimos o início, este sempre ocorre em 0 de forma padronizada. Temos também o **fim** do intervalo representado pelo algarismo 5 que é **obrigatório**. E o terceiro argumento entre parênteses é o **incremento** que nesse caso também é **opcional**, uma vez que se for omitido o incremento sempre ocorrerá, por padrão, de 1 em 1. Então, outra maneira de escrevermos o mesmo código é da seguinte forma:

```
for i in range (5):
  print(i)
```

Sendo que a cada repetição do **for** temos um valor substituindo o seu valor anterior na variável `i`. E ao executarmos esse código serão impressos: 0, 1, 2, 3, 4. Lembrando que apesar do final do intervalo ser definido em 5, a contagem em incrementos de 1 não chega até esse valor, pois a contagem sempre começa do zero em programação quando se trata de um intervalo aberto, como é o caso. Por isso se trata de uma repetição contável, pois sabemos quantas vezes o bloco de código vai executar.

Exemplo 2.4 (**for**)**.** Faça um programa que imprime a soma de todos os números pares entre dois números quaisquer, incluindo-os. Um programa para esse tipo de funcionalidade pode ser dado por:

```
numero1 = int(input('Digite o primeiro valor: '))
numero2 = int(input('Digite o segundo valor: '))
soma = 0
for i in range(numero1, numero2+1):
   if i%2 ==0
      soma = soma+i
print('A soma é', soma)
```

Vamos para a explicação do programa:

- O programa solicita dois números inteiros (`numero1` e `numero2`) como entrada do usuário.
- A variável `soma` é inicializada como 0. Ela armazenará a soma dos números pares encontrados no intervalo.
- O `for` percorre todos os números no intervalo [`numero1`, `numero2`] (inclusive o segundo número, devido ao `numero2 + 1`, pois queremos que o `range` chegue até o número que o usuário digitou como final do intervalo).
- O programa testa se o número atual na variável `i` é par usando `i % 2 == 0` em que `%` é um operador em Python que nos fornece o resto da divisão. Dizemos então: se o resto da divisão de `i` por 2 for zero, então isso significa que o valor atual armazenado em `i` é par, e ele adiciona esse número à variável `soma` pois queremos somar os pares como foi requisitado no enunciado.
- Após o término do laço, ele sai do bloco de código **for** e a soma de todos os números pares no intervalo é exibida com o comando `print`.

Entretanto, ele não ajusta os limites se `numero1 > numero2`. Se você quiser lidar com essa situação, pode adicionar uma verificação para inverter os números, como por exemplo:

```
if numero1 > numero2:
numero1, numero2 = numero2, numero1
```

Esse bloco de código antes do bloco que inicia o `for` garante que o maior número sempre será o número que delimita o final do intervalo.

Basicamente era isso o que queríamos abordar na revisão sobre estruturas de repetição que são muito úteis quando trabalhamos com banco de dados, com recuperação, exploração e análise de dados relacionados a banco de dados.

2.2 Listas

Em Python temos as variáveis normais que acumulam valores de um determinado tipo, e o que chamamos de listas ou vetores que da mesma forma que uma variável, possuem

um nome, mas podem armazenar vários valores que podem ou não ser do mesmo tipo. Um exemplo seria uma lista com médias de notas de uma determinada turma de colégio.

```
notas =[10,5,6.7,2,7.5]
```

Neste caso o nome da lista é `notas` e temos valores colocados dentro dela que se referem as médias de notas dos alunos. Note que temos valores do tipo inteiro e do tipo *float* (números com casas decimais, não-inteiros). Essa é uma das estruturas de dados que permite isso: manter vários tipos de dados e diferentes valores dentro dela. São mais flexíveis e por isso são mais usadas. Sendo que existem outras estruturas que também permitem esse tipo de armazenamento como as tuplas (que são semelhantes as listas, mas imutáveis), Dicionários e conjuntos. Além disso para podermos declarar uma lista de dados, **precisamos delimitar o seu conjunto de dados com colchetes no início e fim da lista, e separamos cada elemento com o uso de vírgulas**. Assim a estrutura interna do Python reconhece que se trata de uma lista e não uma variável comum. Outro exemplo de lista com valores e tipos de dados pode ser:

```
notas =['A',1,2,'Casa',2.3]
```

Temos agora 3 tipos de dados diferentes: *string* (dados de texto entre aspas simples), inteiro e *float*.

2.2.1 Operações com listas

Quando trabalhamos com bancos de dados temos algumas operações com listas que são mais utilizadas. Como o **acesso a elementos** de uma lista e sublistas (partes de uma lista), o **operador `in`** que nos permite verificar se um determinado elemento está contido numa lista ou sublista, e como **percorrer** uma lista e/ou sublista e **contar** os seus elementos.

Acessando elementos de uma lista

Para acessar (ler ou escrever) uma posição da lista, basta informar a posição ou índice relacionado ao elemento em que queremos trabalhar e que está entre colchetes. Sendo que a primeira posição será sempre zero, por padrão, quando se trata de listas. Considere o exemplo a seguir.

Exemplo 2.5 (Acessando um elemento da lista)**.** Considere um bloco de código que calcula a média entre 3 notas de um mesmo aluno:

```
notas = [8.0,5.5,1.5]
media = (notas[0] + notas[1] + notas[2])/3
```

Temos que cada elemento da lista tem um índice vinculado a ele. Se quisermos acessar o primeiro elemento da lista, precisamos acessar o índice 0, o segundo elemento é acessado pelo índice 1 e o terceiro pelo índice 2. Nesse caso usamos a seguinte sintaxe: o **nome da lista** que queremos acessar **seguida do índice entre colchetes do elemento em**

questão que queremos acessar, ou seja, para acessar o primeiro elemento, por exemplo, usamos `notas[0]` (índice 0). Assim estamos dizendo ao Python que queremos acessar o valor que está guardado no primeiro índice da minha lista. No exemplo, `notas[0]` se trata do valor `8.0`, e os demais `notas[1] = 5.5` (segundo elemento) e `notas[2] =1.5` (terceiro elemento). Assim o bloco de código acessa as 3 notas, somando-as e dividindo tudo por 3, i.e., faz a média das 3 notas.

Existe uma forma de acessarmos todos os elementos de uma lista de uma só vez. Nesse caso dizemos que vamos iterar (percorrer cada elemento de forma sequencial) por todos os valores de uma lista. Para o exemplo anterior, se quisermos apenas acessar as notas de um dos alunos, basta utilizarmos o seguinte bloco de código:

```
notas=[8.0,5.5,1.5]
for i in range(len(notas)):
  print(notas[i])
```

Neste bloco de código, o laço `for` será executado tantas vezes quanto a quantidade de elementos da lista `notas`. E, a cada repetição, a variável `i` receberá o índice correspondente ao elemento atual da lista, até que se chegue no último elemento desta. Neste caso o `i` chegará até o terceiro elemento, ou seja até o índice 2, pois ressaltamos que a função `range` sempre começa acessando o elemento com o índice 0. Existe uma outra forma de iteração na qual ao invés de usarmos o índice `i` e o tamanho da lista, que é informado pelo comando `len`, podemos iterar diretamente sobre os elementos da lista.

Operador `in`

Em poucas palavras o operador `in` em Python nos informa se algo está contido em outra coisa. Nesse sentido ele é muito útil se queremos saber se um determinado dado está contido em uma lista, sublista ou conjunto de elementos. Se o dado estiver na lista o operador retorna `True`, caso contrário retorna `False`. Considere como exemplo a seguinte lista:

```
lista=[1,2,3,4]
```

Nesse caso, se quisermos saber se alguns números inteiros estão nessa lista basta usarmos o operador `in` da seguinte forma:

```
print(2 in lista) # imprime True
print(0 in lista) # imprime False
print(5 in lista) # imprime False
print(4 in lista) # imprime True
```

Percorrendo uma lista

Os elementos de uma lista podem ser acessados usando uma estrutura de repetição como já vimos. Existem algumas opções para percorrermos os elementos de uma lista que

são mostrados nos blocos de código a seguir, onde consideramos como exemplo uma lista com 3 nomes.

```
lista=['Maria','Alice','Eduarda']
i = 0
while i<len(lista):
print(lista[i])
i+=1
```

Para este primeiro caso estamos iterando com a lista usando o comando `while`. Para tal precisamos de uma variável `i` que usaremos como índice. A repetição prossegue enquanto `i` for menor do que o tamanho da lista. Quando for igual ou maior ela cessa. Sendo que a cada repetição será impresso o elemento da lista referente a posição `i` que será incrementada em uma unidade (por isso a linha `i+=1`).

```
lista=['Maria','Alice','Eduarda']
for i in range(len(lista)):
print(lista[i])
```

Esse segundo caso já vimos anteriormente e segue o mesmo raciocínio.

```
lista=['Maria','Alice','Eduarda']
for elemento in lista:
print(elemento)
```

Este terceiro caso é o que normalmente usamos. Simplesmente fazemos uso do `for` diretamente sobre a lista. E a cada repetição deste laço o Python retorna o elemento que está na lista. Sendo que aqui a variável `elemento` vai assumir o valor de cada item da lista em cada iteração, e o comando `print` exibe o valor atual da variável `elemento` no console sempre que ocorrer uma repetição do bloco de código. Nesse sentido o interpretador reconhece automaticamente o início e o final de uma lista ao usar um *loop* `for`. Isso ocorre porque a linguagem foi projetada para trabalhar com iteráveis, como listas, de forma intuitiva e eficiente.

Nos três casos será impresso:

```
Maria
Alice
Eduarda
```

A diferença de performance é mínima entre um bloco de código e outro. É mais uma questão de complexidade no que diz respeito a analisar o código e entender o que está acontecendo.

Contando elementos

Uma outra operação útil quando trabalhamos com listas envolve a contagem de seus elementos. Considere por exemplo uma lista com vários números inteiros no bloco de

código abaixo. Vamos contar quantas vezes o número 10 aparece.

```
lista=[1,10,2,10,3,10,4,5,6]
cont=0
for j in lista:
  if j==10:
    cont+=1
print(cont)
```

Se olharmos para a lista conseguimos ver que o número 10 aparece 3 vezes. Mas e se quisermos que o interpretador do Python no computador nos diga quantas vezes o 10 aparece?

Nesse caso podemos primeiramente inicializar uma variável (`cont=0`)[1] na qual vamos guardar a quantidade de vezes que um determinado elemento da lista aparece. E usamos o último método que abordamos anteriormente para percorrer a lista. Sendo que para cada repetição do comando `for` a variável `j` vai acumular um elemento específico e sequencial da lista. E **se** o elemento for igual a 10, significa que achamos um elemento 10 na lista e então guardamos essa informação, a quantidade, na variável `cont` incrementando-a em uma unidade com a linha `cont+=1`. A repetição prossegue sendo que para cada 10 que for encontrado na lista teremos um incremento em uma unidade para `cont`. Quando terminar de passar por todos os elementos da lista será impresso o contador, que no caso do nosso exemplo, terá como impressão na tela o valor 3, referente a quantidade de vezes que o número 10 aparece na lista.

Como o Python tem várias funções prontas que otimizam o trabalho do usuário, podemos usar uma outra alternativa que envolve a função `count`, que serve basicamente para o que queremos: contar quantas vezes aparece um determinado elemento em uma lista. Se estivermos, novamente, buscando o número 10, basta simplesmente executarmos o seguinte bloco de código:

```
lista=[1,10,2,10,3,10,4,5,6]
cont=lista.count(10)
print(cont)
```

Poderíamos imprimir direto o `lista.count(10)` sem a necessidade da definição de um nome de variável, resumindo ainda mais as linhas de código. O lado negativo dessa abordagem é que não sabemos o que **count** de fato para contar os elementos, mas normalmente são funções bem otimizadas que na prática funcionam de acordo com as suas funções.

Em suma essas são as operações principais que podemos fazer com listas quando estamos trabalhando com bancos de dados.

[1] Começa com valor zero, pois ainda não começamos a contar.

2.3 Dicionários

Assim como listas, os Dicionários permitem manter vários tipos de dados e diferentes valores armazenados em sua estrutura, mas apresentam um princípio diferente: não trabalhamos com índices da forma que estamos acostumados quando trabalhamos com listas. Em Dicionários temos as chaves, em que cada chave pode ter um valor associado. Dessa forma dizemos que **Dicionários** se tratam de uma **estrutura de dados que armazena pares chave-valor**.

Os Dicionários em Python funcionam semelhantemente a um Dicionário de gramática, onde você procura uma "chave" (palavra) para encontrar seu "valor" (definição). Isso significa que para acessarmos um item[2] guardado dentro de um Dicionário, precisamos passar como índice a chave atribuída para determinado valor. Na Figura 2.1 temos como exemplo uma estrutura de dados de Dicionário. Note que para cada valor temos uma chave associada. Se tivermos caracteres, tanto na chave quanto no valor, devemos usar aspas simples, que não são necessárias para delimitar números. Além disso, separamos os itens por vírgula e delimitamos o conteúdo de um Dicionário usando o símbolo de chaves. Um trocadilho, temos uma ou mais chaves (*string* ou número que representam os valores) entre chaves.

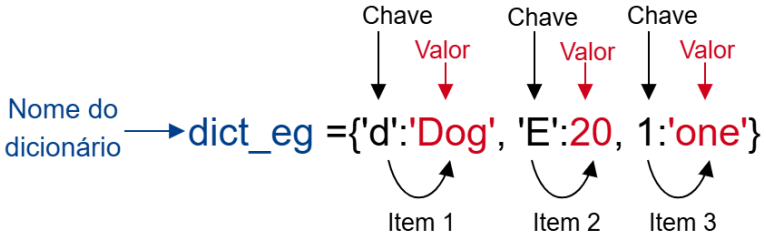

Figura 2.1: Exemplo de uma estrutura de Dicionário.

Neste sentido, se declararmos esse Dicionário e quisermos acessar o índice `d` nos será retornada a palavra `dog`. Sendo que os **Dicionários são mutáveis**, o que significa que seu conteúdo pode ser alterado após a sua criação. Em outras palavras, podemos criar o Dicionário `dict_eg` e depois ir adicionando novos itens à sua estrutura.

Em programação temos uma outra estrutura muito utilizada chamada JSON (*JavaScript Object Notation*), que apesar de estar intimamente relacionado ao Dicionário, tem algumas diferenças importantes. O JSON é um formato de texto universal usado para comunicação entre sistemas e armazenamento de dados estruturados, enquanto que o Dicionário é uma estrutura de dados nativa do Python, usada para manipulação interna de dados. Além disso apresentam sintaxes diferentes, já que o JSON foi criado baseado na linguagem de programação JavaScript, entretanto é comum converter[3] JSON para

[2]Em Dicionários nos referimos ao elemento guardado nele como item, por padrão.

[3]Muitas vezes esse processo de conversão não funciona. Depende muito da estrutura de dados do JSON e do Dicionário, pois podemos ter incompatibilidade de dados entre um e outro. Em geral, a conversão entre JSON e Dicionário funciona bem para dados simples. Problemas podem ocorrer devido

Dicionários Python e vice-versa para facilitar o trabalho em programas Python.

Os principais pontos com relação a Dicionários são:

- **Pares de chave-valor:** cada item em um Dicionário é um par que consiste em uma chave e seu valor correspondente;
- **Não ordenado:** ao contrário das listas, os Dicionários não armazenam itens em nenhuma ordem específica. Em versões abaixo de 3.7 do Python, tínhamos que tomar cuidado quando realizávamos uma iteração com um Dicionário, pois a ordem dela não era garantida. Acima desta versão, os Dicionários mantêm a ordem de inserção, mas ainda não usam indexação (índices para cada item) e por isso é importante evitar modificar o Dicionário durante a iteração ou fazer suposições sobre ordem sem analisá-la explicitamente;
- **As chaves devem ser exclusivas:** Cada chave em um Dicionário deve ser exclusiva. Se você atribuir um novo valor a uma chave existente, ela substituirá o valor anterior;
- **As chaves são imutáveis:** apesar do Dicionário ser mutável, as chaves podem ser de qualquer tipo imutável, como inteiros, cadeias de caracteres ou tuplas.

2.3.1 Operações em Dicionários

Vamos agora realizar as principais operações que podemos fazer com Dicionários para uso de banco de dados.

Acessando valores

Para acessarmos um valor guardado em um Dicionário temos que realizar o acesso pela sua chave correspondente. Considere por exemplo o seguinte Dicionário:

```
estudante={
'nome':'Maria'
'idade':25,
'nota':9}
```

Temos 3 itens nesse dicionário. Note que para cada item temos um par chave-valor. Nesse caso, se quisermos acessar um item deste dicionário colocamos no lugar do índice, como fazíamos com listas, a chave do item que queremos acessar. Suponha que queremos acessar o nome do estudante. Usamos a seguinte linha de comando:

```
print(estudante['nome'])
```

Neste caso teremos com retorno o nome da estudante que está guardado no Dicionário que seria `Maria`.

a tipos de dados não suportados, chaves inválidas, estruturas complexas ou codificação, e para contornar isso é necessário ajustar os dados ou usar funções personalizadas para lidar com as diferenças.

Adição e Atualização

Considere o caso em que queremos adicionar um item novo com uma chave chamada 'formacao'[4]. Para a adição de uma nova chave-valor basta usarmos a seguinte linha de comando:

```
estudante['formacao']='física'
```

Sendo que nesse exemplo supomos que o estudante tem formação em Física. E, para atualização do valor em um par chave-valor basta usarmos a mesma linha, pois o novo valor vai sobrescrever o antigo, assumindo o seu lugar, ou seja, suponha que trocamos o valor da chave `'idade'` de 25 para 23. Para tal usamos a seguinte linha de comando:

```
estudante['idade']=23
```

Se quisermos visualizar os valores de um dicionário, basta mandarmos imprimir o Dicionário com o comando:

```
print(estudante)
```

Iteração em Dicionários

É possível percorrer as chaves, os valores ou ambos em um Dicionário. Para percorrer a chave podemos usar um *loop* `for` como o bloco de código a seguir.

```
for chave in estudante:
  print(chave,estudante[chave])
```

Quando fazemos um `for` direto, usando o nome do dicionário, é impresso as chaves que tem neste. No nosso caso mandamos imprimir também o item com o seu valor correspondente que seria:

```
nome Maria
idade 23
nota 9
formacao física
```

Métodos em Dicionários

A estrutura de Dicionários apresentam alguns métodos e/ou funções que valem a pena destacar.

- `keys()`: retorna uma exibição em forma de lista de todas as chaves;

```
keys = estudante.keys()
dict_keys(['nome','idade','nota','formacao'])
```

[4]Lembrando que para nomes de variáveis e/ou constantes devemos sempre omitir acentos e outros símbolos como o cedilha em *strings*.

- `values()`: retorna uma exibição em forma de lista de todos os valores;

```
values = estudante.values()
dict_values(['Maria',23,9,'física'])
```

- `items()`: retorna uma exibição semelhante a uma lista de todos os pares de valores-chave como tuplas;

```
items = estudante.items()
dict_items([('nome','Maria'),('idade',23),
            ('nota',9),('formacao','física')])
```

- `get()`: retorna o valor da chave fornecida, se presente no Dicionário. Caso contrário, ele retornará uma mensagem (`None por default`).
  Exemplo 1:

```
get1 = estudante.get('formacao')
física
```

  Exemplo 2:

```
get1 = estudante.get('serie','Inválido')
Inválido
```

  Sendo que adicionamos um segundo parâmetro que será a mensagem imprimida na tela caso a chave não esteja presente no Dicionário, como foi o caso. Caso contrário será impresso `None por default`.

Dicionários aninhados

Dicionários também podem conter outros Dicionários, permitindo estruturas de dados mais complexas. Chamamos isso de aninhamento de Dicionários. Considere por exemplo um Dicionário chamado `estudantes` no qual temos uma lista de outros Dicionários (`estudante1` e `estudante2`) guardados nele, como mostrado no bloco de código a seguir.

```
estudantes ={
  'estudante1':{
    'nome': 'Maria', 'idade':23
  },
  'estudante2':{
    'nome': 'João', 'idade':25
  }
}
print(estudantes['estudante1']['nome'])
Maria
```

Caso seja de nosso interesse saber o nome do `estudante1`, basta mandarmos imprimir como mostrado no exemplo, com o nome do dicionário seguido da chave referente ao `estudante1` que é um dicionário, seguido da chave `nome` na qual temos como valor o nome do estudante 1 que nesse caso é Maria.

2.4 Funções úteis para trabalhar com banco de dados

A seguir vamos listar algumas funções úteis para se trabalhar com dados de banco de dados e suas respectivas funcionalidade.

2.4.1 Funções para *strings*

- `len()`: Para saber o tamanho da *string*;
- `str.lower()` e `str.upper()`: converter os caracteres de uma *string* para minúsculas e maiúsculas, respectivamente;
- `str.split()`: divide uma *string* em pares;
- `str.join()`: junta uma lista de *strings* numa só lista;
- `str.replace()`: substitui partes de uma *string*. Serve por exemplo, para substituir todos os espaços em uma *string* por pontos, caso seja de seu interesse;
- `str.format()`: para formatação de *string*, o que nos permite inserir valores novos em uma *string* formatada.

2.4.2 Funções para Dados Estruturados

- `len()`: para saber o tamanho da lista ou dicionário;
- `sorted()`: ordena os elementos de uma lista;
- `sum()`: soma os valors de uma lista numérica;
- `list.append()` e `list.remove()`: para adicionar ou remover elementos de uma lista;
- `dict.get()`: busca uma chave de forma segura em um dicionário.

3 Revisão de Banco de Dados

Um banco de dados é um sistema organizado para armazenar, gerenciar e recuperar informações de forma eficiente. Ele é projetado para lidar com grandes volumes de dados e permitir que esses dados sejam acessados, consultados, modificados e manipulados de maneira estruturada e segura. Em resumo, é uma coleção organizada de dados armazenados eletronicamente.

As componentes de um Banco de Dados são os **dados**, o **SGBD (Sistema Gerenciador de Banco de Dados)**[1] e os **usuários** que vão usar o Banco de Dados através do SGBD.

Existem vários tipos de Banco de Dados, que listamos a seguir. Sendo que, em um primeiro momento, estaremos mais interessados nos dados **Relacionais**. Nesse tipo de banco de dados os dados são armazenados em tabelas (linhas e colunas) e têm relações entre si. Sendo que usam o que chamamos de linguagem SQL (*Structured Query Language*; em português-BR linguagem de consulta estruturada) para interagir com os dados. Alguns exemplos são o MySQL e PostgreSQL.

Existem várias **vantagens** em se trabalhar com banco de dados. Podemos citar:

- **Organização:** refere-se à maneira como os dados são estruturados, armazenados e relacionados para facilitar o acesso, consulta e manutenção;
- **Acessibilidade:** refere-se à capacidade de os usuários ou sistemas acessarem o banco de dados de maneira eficiente e sem interrupções, desde que estejam devidamente autorizados;
- **Segurança:** um banco de dados garante que os dados estejam protegidos contra acesso não autorizado, perda ou corrupção;
- **Integridade:** assegura que os dados armazenados no banco sejam precisos, consistentes e confiáveis;
- **Escalabilidade:** refere-se à capacidade de o banco de dados crescer (ou reduzir) de forma eficiente para lidar com volumes maiores de dados ou usuários. Assim mais usuários conseguem acessar o banco de dados sem perca de desempenho.

3.1 Vantagens e exemplos de um SGBD

Alguns tipos de pesquisa requerem o tratamento de quantidades massivas de dados, principalmente na área científica. Uma vez que ao analisar fenômenos e/ ou fazer coleta de dados, acabamos gerando um volume muito grande de dados. E uma forma eficiente para se trabalhar com esse volume grande de dados reside no seu armazenamento em um banco de dados. Assim, bancos de dados são essenciais para a comunidade científica no

[1] É justamente o SGBD que vai nos permitir trabalhar com os Dados de um Banco de Dados.

geral, pois eles permitem que as análises ocorram de forma mais otimizada e usar funções sob esses dados, por exemplo. Alguns exemplos de SGBD's são:

- **Genômica:** no armazenamento e análise de sequências genéticas. Área na qual podemos citar, como exemplo, o banco de dados GenBank;
- **Física:** no gerenciamento de dados de experimentos de alta energia (CERN)[2]
- **Astronomia:** na disponibilização de dados astronômicos capturados através de telescópios.

3.2 Conceitos importantes de Banco de Dados

Quando trabalhamos com Banco de Dados têm alguns conceitos que são importantes da gente saber.

3.2.1 Chaves primárias

Os bancos de dados do tipo relacionais tem tabelas como já mencionamos. Sendo que uma tabela terá dados sobre alguma coisa em comum dispostos em uma coluna. Nesse sentido atribuímos a esses dados o que chamamos **de chave primária** (PK - do inglês *Primary Key*) que são atributos únicos, utilizados para identificar um registro em uma tabela.

Exemplo 3.1 (CodigoCliente)**.** Geralmente quando você vai em um mercado e se cadastra para promoções de clientes, os seus dados são incluídos em um banco de dados. Neste sentido, considere como exemplo um banco de dados com dados de clientes de um supermercado como o mostrado na Tabela.

Cliente		
CodigoCliente	**Nome**	**Telefone**
1	Maria	23 12154584
2	João	42 654874527

Tabela 3.1: Exemplo de um banco de dados de clientes de um supermercado.

Neste caso precisamos que um dos dados seja uma chave primária, um dado que identifique uma linha da tabela, i.e. o cliente no banco de dados. A chave primária deste exemplo identifica o cliente por números inteiros em ordem crescente na coluna CodigoCliente tabelada. E é bem mais fácil identificá-los assim do que pelo nome, por exemplo, pois o mercado pode ter outras clientes chamadas Maria. Note ainda que a função de uma chave primária é equivalente a de uma chave em um Dicionário Python.

[2] Antigo acrônimo para *Conseil Européen pour la Recherche Nucléaire*.

3.2.2 Chaves estrangeiras

As **chaves estrangeiras** (FK - do inglês *Foreign Key*) são utilizadas para relacionar um registro de uma tabela com um registro de outra tabela. Neste sentido considerando o Exemplo 3.1 podemos querer saber qual cliente fez uma compra específica. Tendo um banco de dados de compras podemos sempre relacionar um cliente específico com uma ou mais compras específicas. Na Tabela 3.2 temos um banco de dados de compras, onde vemos o mesmo cliente, Maria com CodigoCliente 1, realizando compras em datas diferentes e com outros gastos.

Compra			
CodigoCompra	**Data**	**Valor**	**CodCliente**
1	17/06/2024	15,90	1
2	23/06/2024	23,50	1

Tabela 3.2: Exemplo de um banco de dados de compras de um supermercado que se relaciona com o registro da Tabela 3.1.

Neste contexto as FK são usadas para saber quais foram as compras de um determinado cliente. Para cada compra inserida identificamos o cliente que comprou com a sua chave primária que para o banco de dados **Compra** será uma FK, pelo fato de ter vindo de outro banco de dados (**Cliente** da Tabela 3.1).

3.2.3 Modelo simples de banco de dados - SQL e principais comandos

Um modelo simples de banco de dados SQL organiza os dados de maneira estruturada, utilizando tabelas relacionadas. Ele é escalável e pode crescer conforme as necessidades do sistema, sendo amplamente usado em aplicações como lojas, sistemas de gerenciamento de usuários e mais. Daí a importância de se entender o que são chaves primárias e estrangeiras, as diferenças entre si e suas principais funções. Vamos agora entender alguns dos principais comandos em banco de dados.

Comando `CREATE TABLE`

O comando `CREATE TABLE` é uma instrução SQL **usada para criar uma nova tabela** em um banco de dados. Ele define a estrutura da tabela, incluindo seu nome, colunas, tipos de dados e, opcionalmente, restrições como chaves primárias ou estrangeiras. Ele é altamente personalizável e permite implementar relacionamentos, restrições e organizar dados de forma eficiente. A sintaxe básica para esse comando se dá na seguinte forma:

```
CREATE TABLE nome_da_tabela (
   nome_da_coluna1 tipo_de_dado1 restrições1,
   nome_da_coluna2 tipo_de_dado2 restrições2,
   ...);
```

Temos então o argumento `nome_da_tabela` que se refere ao nome da tabela que deve ser único dentro do banco de dados, `nome_da_colunaX` que deve ser único dentro da tabela, `tipo_de_dadoX` que define o tipo de dado que cada coluna pode armazenar — Exemplos são: `INT` (números inteiros), `VARCHAR(100)` (cadeias de caracteres), `DATE` (datas) —, e as `restriçõesX` que são opcionais e definem regras para os dados. Algumas restrições muito usadas são:

- `PRIMARY KEY`: Define a chave primária da tabela;
- `FOREIGN KEY`: Define um relacionamento com outra tabela;
- `NOT NULL`: Exige que a coluna tenha sempre um valor;
- `UNIQUE`: Garante que todos os valores na coluna sejam únicos.

Note também que entre cada coluna temos uma vírgula e para finalizar a criação de uma tabela usamos ponto e vírgula. Pois assim podemos criar uma outra tabela em sequência que pode ou não estar relacionada com a primeira tabela.

Um exemplo prático de criação de tabela é:

```
CREATE TABLE Clientes (
   ID INT AUTO_INCREMENT PRIMARY KEY,
   Nome VARCHAR(100) NOT NULL,
   Email VARCHAR(100) UNIQUE,
   DataNascimento DATE);
```

A tabela `Clientes` tem 4 colunas: `ID` que é um inteiro e a chave primária que deve ser única e incrementada automaticamente sempre que houver a inserção de um novo dado, `Nome` (dado do tipo *string* de até 100 caracteres, sendo obrigatório devido a restrição `NOT NULL`), `Email` (uma *string* de até 100 caracteres e deve ser único devido a restrição `UNIQUE`) e `DataNascimento` que armazena dados no formato de datas.

Comando `ALTER TABLE`

O comando `ALTER TABLE` é uma instrução SQL **usada para modificar a estrutura de uma tabela já existente** em um banco de dados. Ele permite adicionar, remover ou alterar colunas, bem como definir ou remover restrições na tabela. Ele é essencial para ajustar o esquema do banco de dados à medida que os requisitos do sistema evoluem. A sintaxe básica para esse comando é

```
ALTER TABLE nome_da_tabela <comando>;
```

Suponha, por exemplo, que queremos adicionar uma coluna de Telefones na Tabela Clientes que criamos com o comando `CREATE TABLE`. Nesse caso usamos o seguinte bloco de código:

```
ALTER TABLE Clientes ADD Telefone VARCHAR(15);
```

Sendo que para remover uma coluna em particular usamos:

```
ALTER TABLE Clientes DROP COLUMN email;
```

que remove a coluna relacionada ao e-mail dos clientes. Ambos os comandos são colocados uma linha após o ponto e vírgula da Tabela já criada. E se quisermos adicionar uma FK a uma Tabela de Compras relacionando a mesma com a tabela Clientes fazemos como mostra a Figura 3.1.

```
CREATE TABLE cliente (
     nome           VARCHAR(100),
     telefone       VARCHAR(30),
     codigocliente INT AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE compra (
     data          VARCHAR(10),
     valor         FLOAT(10,2),
     codcliente    INT NOT NULL,
     codigocompra INT AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE produto (
     nome           VARCHAR(80),
     preco         FLOAT(10,2),
     codigoproduto INT AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE compraproduto (
     idcompra   INT,
     idproduto INT,
     unidades   INT);
ALTER TABLE compra
  ADD FOREIGN KEY(codcliente) REFERENCES cliente
(codigocliente);
```

Figura 3.1: Exemplo de uso dos comandos `CREATE TABLE` e `ALTER TABLE` usado para relacionar a tabela **compra** com **cliente** através de uma FK chamada `codcliente`.

Comando `INSERT`

O comando `INSERT` é usado em SQL para **adicionar novos registros (linhas) a uma tabela existente** em um banco de dados. Ele permite inserir dados nas colunas especificadas, seguindo o formato e as restrições definidas na estrutura da tabela. Ele oferece flexibilidade para adicionar registros individuais, múltiplos ou até gerar dados a partir de outras consultas. A sintaxe básica é:

```
INSERT INTO nome_da_tabela (atributo1, atributo2,...) VALUES (valor1,
valor2, ...);
```

Um exemplo simples seria inserir o nome e telefone na tabela Clientes de um novo cliente chamado Joaquim. Usamos nesse caso o seguinte comando:

```
INSERT INTO Clientes (nome, Telefone) VALUES (Joaquim, 43 564214574);
```

Neste caso é criado automaticamente uma chave primária para esse dado em particular. Isso acontece pois ele foi definido lá no bloco de código `CREATE TABLE` com restrição de ser auto incrementado. Ressaltando que devemos respeitar a ordem e o tipo de dados das colunas quando os inserimos na tabela. E se você não especificar as colunas, i.e. os atributos como foram colocados, os valores devem seguir a ordem definida na tabela.

Comando `UPDATE`

O comando `UPDATE` em SQL é **usado para modificar os dados existentes em uma tabela**. Ele permite alterar os valores de uma ou mais colunas em registros específicos, definidos por uma cláusula `WHERE`. É extremamente importante usar a cláusula `WHERE` para restringir o que deve ser atualizado, pois se a mesma for omitida, o comando atualizará todos os registros da tabela. O que seria um desastre!! O comando `UPDATE` deve ser usado com cuidado, especialmente em tabelas com muitos dados ou relacionamentos complexos, para evitar inconsistências e erros. A sua sintaxe básica é:

```
UPDATE nome_da_tabela SET coluna1=novovalor1, coluna2=novovalor2
WHERE condição;
```

Um exemplo seria a atualização do Telefone de um cliente chamado John que foi digitado de forma errada quando inserido pela primeira vez no banco de dados. Nesse caso temos o seguinte comando:

```
UPDATE Clientes SET telefone='42 548422775' WHERE nome='John';
```

Assim atualizamos o número de telefone na coluna chamada `nome` com dado John da tabela Clientes.

Comando `DELETE`

O comando `DELETE` em SQL é **usado para remover registros (linhas) de uma tabela** no banco de dados. Ele é flexível e pode ser usado para excluir registros específicos ou, caso nenhuma condição seja fornecida, todos os registros da tabela. Ele também é usado em conjunto com a cláusula `WHERE` para restringir o que deve ser apagado. **Se ele não for usado, todos os registros da tabela serão apagados!** Ele deve ser usado com atenção, especialmente em tabelas críticas ou com relacionamentos complexos, para evitar exclusões acidentais e perda de dados importantes. A sua sintaxe básica é:

```
DELETE FROM nome_da_tabela WHERE condição;
```

Um exemplo seria:

```
DELETE FROM Clientes WHERE ID=2;
```

Assim será deletado todos os dados relacionados ao cliente com chave primária igual a 2. Além disso temos o comando `TRUNCATE` que tem como mesma funcionalidade deletar os dados, e apesar de ser mais rápido do que o comando `DELETE`, ele não suporta condições e não pode ser revertido, uma vez que podemos sempre desfazer o comando `DELETE` com o que chamamos de `ROLLBACK`[3]. Sem falar que o `TRUNCATE` pode falhar se existirem relacionamentos entre tabelas.

Comando `SELECT`

O comando `SELECT` é um dos mais importantes e utilizados em SQL. Ele é **usado para consultar dados de tabelas** no banco de dados. Com ele, é possível selecionar colunas específicas, aplicar filtros, ordenar resultados, agrupar informações e realizar cálculos diretamente nas consultas. Ele é altamente flexível, podendo ser combinado com diversas cláusulas e funções para extrair informações detalhadas e formatadas diretamente do banco de dados. Sua sintaxe básica é dada por:

```
SELECT atributo1, atributo2 FROM nome_da_tabela;
```

Se usarmos

```
SELECT nome, telefone FROM Clientes;
```

teremos como retorno o nome e telefone de todos os clientes da tabela Clientes. Se não especificarmos a coluna que queremos com o `SELECT` precisamos colocar um asterístico $\star$ e, assim, teremos como retorno todas as colunas com os dados de todos os clientes da tabela Clientes.

[3]Quando vamos realizar algum procedimento no banco de dados relacional, estaremos trabalhando com o que chamamos de **transação**. Neste sentido, enquanto você não disser para o SGBD que terminou de fazer a sua transação, seja por um *commit* ou execução de código, ele fornece a possibilidade de voltar atrás com o `ROLLBACK`. Entretanto, se você já terminou de realizar a transação dificilmente conseguirá recuperar dados deletados, se não tiver um *backup* dos mesmos.

4 Parte Prática

Vamos primeiramente acessar a infraestrutura do LIneA. Acesse o site linea.org.br. Role a página que foi aberta até achar a aba de Plataformas Científicas, e clique sobre Jupyter Hub que será o ambiente no qual iremos realizar a parte prática, como vemos na Figura 4.1.

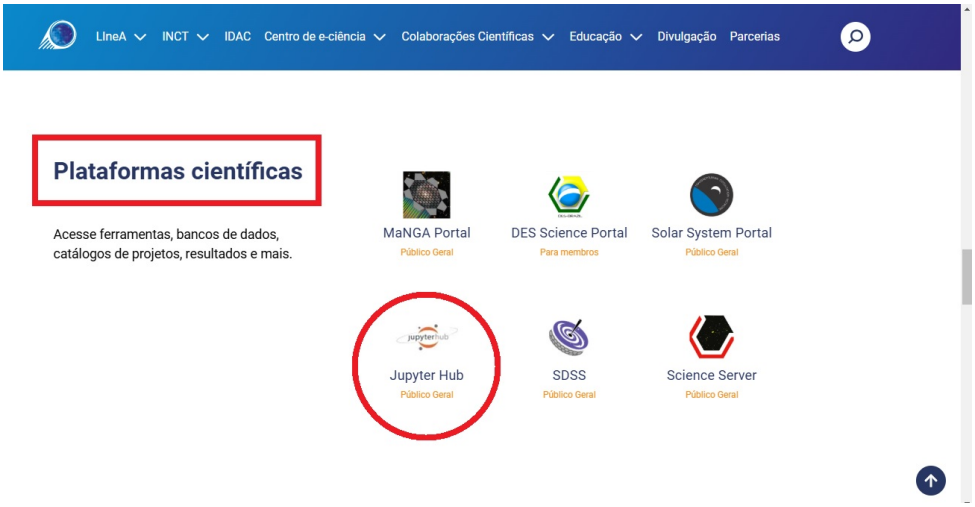

Figura 4.1: Seção de Plataformas Científicas da página inicial do LIneA.

Uma nova aba em seu navegador vai carregar. Faça o Login em sua conta de acordo com o seu cadastro prévio (CAFE, CILOGON ou Google) como mostrado na Figura 4.2. Caso não tenha registro para uso de plataformas, ferramentas e banco de dados do LIneA, você deve se registrar. Para tal, basta clicar sobre `Register here`.

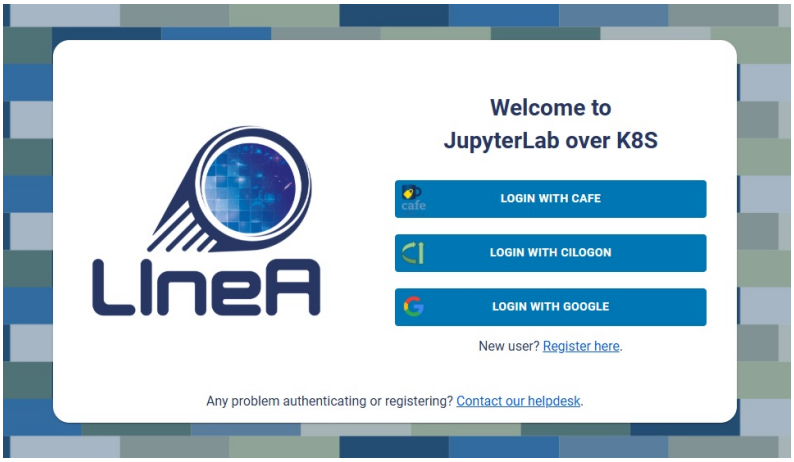

Figura 4.2: Aba de Login e registro de novos usuários para uso do jupyter hub do LIneA.

Uma janela de informações sobre as ferramentas disponíveis, dicas, dentre outras coisas vai abrir, role a página até que apareça um botão de `START`. Clique sobre ele. O servidor

vai iniciar. A primeira coisa que você verá provavelmente será o ambiente do Jupyter notebook com o terminal aberto do sistema no navegador. Acesse então o repositório do GitHub para a parte prática:
https://github.com/helenocampos/curso_python_bd

Em seguida clique sobre o botão verde `Code` e copie o link HTTPS clicando sobre o ícone copiar (duas folhas) como mostra a Figura 4.3. Você verá a mensagem `Copied!` e uma marcação em verde momentânea que lembra o sinal de "visto".

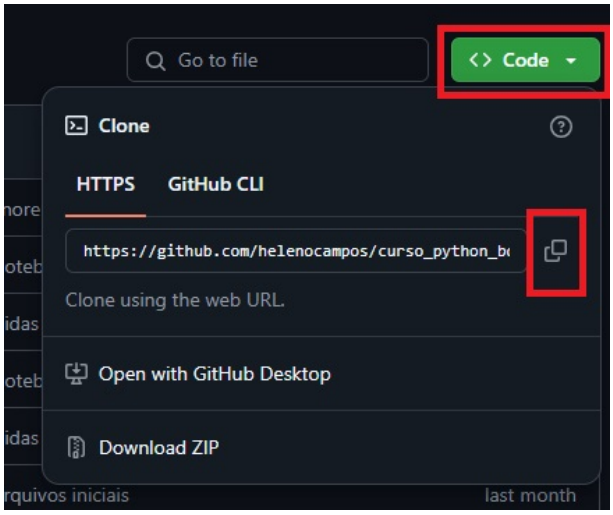

Figura 4.3: Copiando um repositório no GitHub.

Retorne ao terminal do Jupyter notebook e digite o seguinte comando no *prompt* — indicado na cor azul no terminal emulado a seguir — (substitua `url_HTTPS` pelo link que você acabou de copiar):

```
user.name@jupyter-user:~$ git clone url_HTTPS
```

Então você verá na janela ao lado do terminal os arquivos que estavam no repositório que você acessou do GitHub copiados para o seu ambiente de trabalho no Jupyter notebook. Caso não esteja vendo os arquivos, clique com o botão esquerdo do *mouse* sobre o ícone *File Browser* ou use as teclas `Ctrl+Shift+F` do seu teclado. Para os próximos passos veja a parte prática (a partir dos 55m15s) da Aula no Youtube em [Curso] Python e Banco de Dados: Integração e Aplicações Práticas. Bons estudos!