Este é o CS50

Introdução à Ciência da Computação (CS50)

OpenCourseWare

Doar (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Aula 7

Bem-vindo!

- Nas semanas anteriores, apresentamos a vocês o Python, uma linguagem de programação de alto nível que utiliza os mesmos blocos de construção que aprendemos em C. No entanto, apresentamos essa nova linguagem não com o objetivo de aprender "apenas

mais uma linguagem". Em vez disso, fazemos isso porque algumas ferramentas são melhores para certas tarefas e não tão boas para outras!

- Esta semana, continuaremos a trabalhar com mais sintaxe relacionada ao Python.
- Além disso, integraremos esse conhecimento aos dados.
- Por fim, discutiremos *SQL* ou *Linguagem de Consulta Estruturada* , uma forma específica de domínio pela qual podemos interagir com dados e modificá-los.
- Em geral, um dos objetivos deste curso é aprender a programar de forma geral – e não apenas como programar nas linguagens descritas neste curso.

Banco de dados de arquivo simples

- Como você provavelmente já viu antes, os dados podem frequentemente ser descritos em padrões de colunas e linhas.
- Planilhas como as criadas no Microsoft Excel e no Google Sheets podem ser exportadas para um arquivo *de valores separados por vírgula (CSV)* `csv` .
- Se você observar um `csv` arquivo, notará que ele é plano, pois todos os nossos dados são armazenados em uma única tabela representada por um arquivo de texto. Chamamos esse formato de dados de *banco de dados de arquivo plano* .
- Todos os dados são armazenados linha por linha. Cada coluna é separada por uma vírgula ou outro valor.
- O Python oferece suporte nativo a `csv` arquivos.
- Primeiro, baixe o arquivo favorites.csv (https://cdn.cs50.net/2023/fall/lectures/7/src7/favorites/favorites.csv) e carregue-o no seu explorador de arquivos dentro da pasta cs50.dev (https://cs50.dev) . Segundo, ao examinar esses dados, observe que a primeira linha é especial, pois define cada coluna. Em seguida, cada registro é armazenado linha por linha.
- Na janela do terminal, digite `code favorites.py` e escreva o código da seguinte forma:

```
# Prints all favorites in CSV using csv.reader

import csv

# Open CSV file
with open("favorites.csv", "r") as file:

    # Create reader
    reader = csv.reader(file)

    # Skip header row
    next(reader)

    # Iterate over CSV file, printing each favorite
    for row in reader:
        print(row[1])
```

Observe que a `csv` biblioteca foi importada. Além disso, criamos um objeto `reader` que armazenará o resultado de `println` `csv.reader(file)`. A `csv.reader` função lê cada linha do arquivo e, em nosso código, armazenamos os resultados em `println` `reader`. `print(row[1])` Portanto, `println` imprimirá o idioma do `favorites.csv` arquivo.

- Você pode melhorar seu código da seguinte forma:

```
# Stores favorite in a variable

import csv

# Open CSV file
with open("favorites.csv", "r") as file:

    # Create reader
    reader = csv.reader(file)

    # Skip header row
    next(reader)

    # Iterate over CSV file, printing each favorite
    for row in reader:
        favorite = row[1]
        print(favorite)
```

Observe que o `favorite` valor é armazenado e depois impresso. Observe também que usamos a `next` função para pular para a próxima linha do nosso leitor.

- Uma das desvantagens da abordagem acima é que estamos confiando que essa `row[1]` será sempre a opção preferida. No entanto, o que aconteceria se as colunas tivessem sido reorganizadas?

- Podemos corrigir esse possível problema. O Python também permite indexar por meio das chaves de uma lista. Modifique seu código da seguinte forma:

```
# Prints all favorites in CSV using csv.DictReader

import csv

# Open CSV file
with open("favorites.csv", "r") as file:

    # Create DictReader
    reader = csv.DictReader(file)

    # Iterate over CSV file, printing each favorite
    for row in reader:
        favorite = row["language"]
        print(favorite)
```

Observe que este exemplo utiliza diretamente a `language` chave na instrução print. `favorite` indexa o `reader` dicionário de `row["language"]`.

- Isso poderia ser ainda mais simplificado para:

```
# Prints all favorites in CSV using csv.DictReader

import csv

# Open CSV file
with open("favorites.csv", "r") as file:

    # Create DictReader
    reader = csv.DictReader(file)

    # Iterate over CSV file, printing each favorite
    for row in reader:
        print(row["language"])
```

- Para contar o número de idiomas favoritos expressos no `csv` arquivo, podemos fazer o seguinte:

```
# Counts favorites using variables

import csv

# Open CSV file
with open("favorites.csv", "r") as file:

    # Create DictReader
    reader = csv.DictReader(file)

    # Counts
    scratch, c, python = 0, 0, 0

    # Iterate over CSV file, counting favorites
    for row in reader:
        favorite = row["language"]
        if favorite == "Scratch":
            scratch += 1
        elif favorite == "C":
            c += 1
        elif favorite == "Python":
            python += 1

# Print counts
print(f"Scratch: {scratch}")
print(f"C: {c}")
print(f"Python: {python}")
```

  Observe que cada idioma é contabilizado usando `if` declarações. Além disso, observe os `==` sinais de igual duplo nessas `if` declarações.

- Python nos permite usar um dicionário para contar as ocorrências `counts` de cada idioma. Considere a seguinte melhoria em nosso código:

```
# Counts favorites using dictionary

import csv

# Open CSV file
with open("favorites.csv", "r") as file:
```

```
    # Create DictReader
    reader = csv.DictReader(file)

    # Counts
    counts = {}

    # Iterate over CSV file, counting favorites
    for row in reader:
        favorite = row["language"]
        if favorite in counts:
            counts[favorite] += 1
        else:
            counts[favorite] = 1

# Print counts
for favorite in counts:
    print(f"{favorite}: {counts[favorite]}")
```

Observe que o valor com `counts` a chave `favorite` é incrementado quando já existe. Se não existir, definimos `counts[favorite]` e atribuímos o valor 1. Além disso, a string formatada foi aprimorada para apresentar o `counts[favorite]`.

- Python também permite ordenação `counts`. Melhore seu código da seguinte forma:

```
# Sorts favorites by key

import csv

# Open CSV file
with open("favorites.csv", "r") as file:

    # Create DictReader
    reader = csv.DictReader(file)

    # Counts
    counts = {}

    # Iterate over CSV file, counting favorites
    for row in reader:
        favorite = row["language"]
        if favorite in counts:
            counts[favorite] += 1
        else:
            counts[favorite] = 1

# Print counts
for favorite in sorted(counts):
    print(f"{favorite}: {counts[favorite]}")
```

Observe o caractere `sorted(counts)` na parte inferior do código.

- Se você consultar os parâmetros da `sorted` função na documentação do Python, verá que ela possui muitos parâmetros predefinidos. Você pode aproveitar alguns desses parâmetros predefinidos da seguinte forma:

```
# Sorts favorites by value using .get
```

```
import csv

# Open CSV file
with open("favorites.csv", "r") as file:

    # Create DictReader
    reader = csv.DictReader(file)

    # Counts
    counts = {}

    # Iterate over CSV file, counting favorites
    for row in reader:
        favorite = row["language"]
        if favorite in counts:
            counts[favorite] += 1
        else:
            counts[favorite] = 1

# Print counts
for favorite in sorted(counts, key=counts.get, reverse=True):
    print(f"{favorite}: {counts[favorite]}")
```

Observe os argumentos passados para `sort` `sorted`. O `key` argumento `sort` permite que você informe ao Python o método que deseja usar para ordenar os itens. Neste caso, `sort` `counts.get` é usado para ordenar pelos valores. ` sort` `reverse=True` indica `sorted` que a ordenação deve ser feita do maior para o menor.

- Python possui diversas bibliotecas que podemos utilizar em nosso código. Uma dessas bibliotecas é a `std::string` `collections`, da qual podemos importar a função ` counts` `Counter`. `Counter` Ela permite acessar a contagem de cada idioma sem a complexidade de todas as `if` instruções `if` vistas no código anterior. Você pode implementá-la da seguinte forma:

```
# Sorts favorites by value using .get

import csv

from collections import Counter

# Open CSV file
with open("favorites.csv", "r") as file:

    # Create DictReader
    reader = csv.DictReader(file)

    # Counts
    counts = Counter()

    # Iterate over CSV file, counting favorites
    for row in reader:
        favorite = row["language"]
        counts[favorite] += 1

# Print counts
```

```
for favorite, count in counts.most_common():
    print(f"{favorite}: {count}")
```

Observe como isso `counts = Counter()` permite o uso desta `Counter` classe importada de `collections` .

- Você pode aprender mais sobre o método `sorted` (https://docs.python.org/3/howto/sorting.html) na documentação do Python (https://docs.python.org/3/howto/sorting.html) .

Bancos de dados relacionais

- O Google, o X e o Meta utilizam bancos de dados relacionais para armazenar suas informações em grande escala.
- Os bancos de dados relacionais armazenam dados em linhas e colunas em estruturas chamadas *tabelas* .
- O SQL permite quatro tipos de comandos:

```
Create
Read
Update
Delete
```

- Essas quatro operações são carinhosamente chamadas de *CRUD* .
- Podemos criar um banco de dados com a sintaxe SQL `CREATE TABLE table (column type, ...);` . Mas onde você executa esse comando?
- `sqlite3` É um tipo de banco de dados SQL que possui os recursos principais necessários para este curso.
- Podemos criar um banco de dados SQL no terminal digitando `sqlite3 favorites.db` . Ao ser solicitado, confirmaremos a criação `favorites.db` pressionando `y` .
- Você notará um prompt diferente, pois agora estamos usando um programa chamado `sqlite` .
- Podemos entrar `sqlite` no `csv` modo digitando `.mode csv` . Em seguida, podemos importar nossos dados do `csv` arquivo digitando `.import favorites.csv favorites` . Parece que nada aconteceu!
- Podemos digitar `.schema` para visualizar a estrutura do banco de dados.
- Você pode ler itens de uma tabela usando a sintaxe `SELECT columns FROM table` .
- Por exemplo, você pode digitar `SELECT * FROM favorites;` que imprimirá todas as linhas em `favorites` .
- Você pode obter um subconjunto dos dados usando o comando `SELECT language FROM favorites;` .
- O SQL suporta diversos comandos para acessar dados, incluindo:

```
AVG
COUNT
DISTINCT
LOWER
MAX
MIN
UPPER
```

- Por exemplo, você pode digitar `SELECT COUNT(*) FROM favorites;`. Além disso, você pode digitar `SELECT DISTINCT language FROM favorites;` para obter uma lista dos idiomas individuais presentes no banco de dados. Você pode até digitar `SELECT COUNT(DISTINCT language) FROM favorites;` para obter a contagem desses idiomas.
- O SQL oferece comandos adicionais que podemos utilizar em nossas consultas:

```
WHERE       -- adding a Boolean expression to filter our data
LIKE        -- filtering responses more loosely
ORDER BY    -- ordering responses
LIMIT       -- limiting the number of responses
GROUP BY    -- grouping responses together
```

  Observe que costumamos `--` escrever um comentário em SQL.

SELECIONAR

- Por exemplo, podemos executar `SELECT COUNT(*) FROM favorites WHERE language = 'C';`. Uma contagem é apresentada.
- Além disso, poderíamos digitar `SELECT COUNT(*) FROM favorites WHERE language = 'C' AND problem = 'Hello, World';`. Observe como o `AND` é utilizado para refinar nossos resultados.
- Da mesma forma, poderíamos executar `SELECT language, COUNT(*) FROM favorites GROUP BY language;`. Isso ofereceria uma tabela temporária que mostraria o idioma e a contagem.
- Poderíamos melhorar isso digitando `SELECT language, COUNT(*) FROM favorites GROUP BY language ORDER BY COUNT(*);`. Isso ordenará a tabela resultante pelo `count`.
- Da mesma forma, poderíamos executar `SELECT COUNT(*) FROM favorites WHERE language = 'C' AND (problem = 'Hello, World' OR problem = 'Hello, It''s Me');`. Observe que existem duas `''` marcas para permitir o uso de aspas simples de uma forma que não confunda o SQL.
- Além disso, poderíamos executar `SELECT COUNT(*) FROM favorites WHERE language = 'C' AND problem LIKE 'Hello, %';` para encontrar quaisquer problemas que comecem com `Hello,` (incluindo um espaço).
- Também podemos agrupar os valores de cada idioma executando o comando `SELECT language, COUNT(*) FROM favorites GROUP BY language;`.

- Podemos ordenar a saída da seguinte forma: `SELECT language, COUNT(*) FROM favorites GROUP BY language ORDER BY COUNT(*) DESC;`.
- Podemos até criar aliases, como variáveis em nossas consultas: `SELECT language, COUNT(*) AS n FROM favorites GROUP BY language ORDER BY n DESC;`.
- Finalmente, podemos limitar nossa saída a um ou mais valores: `SELECT language, COUNT(*) AS n FROM favorites GROUP BY language ORDER BY n DESC LIMIT 1;`.

INSERIR

- Também podemos `INSERT` inserir dados em um banco de dados SQL utilizando o formulário `INSERT INTO table (column...) VALUES(value, ...);`.
- Podemos executar `INSERT INTO favorites (language, problem) VALUES ('SQL', 'Fiftyville');`.
- Você pode verificar a adição deste favorito executando o comando `SELECT * FROM favorites;`.

EXCLUIR

- `DELETE` Permite excluir partes dos seus dados. Por exemplo, você pode usar o comando ` `DELETE FROM favorites WHERE Timestamp IS NULL;`. Isso exclui qualquer registro onde o valor `Timestamp` seja nulo `NULL`.

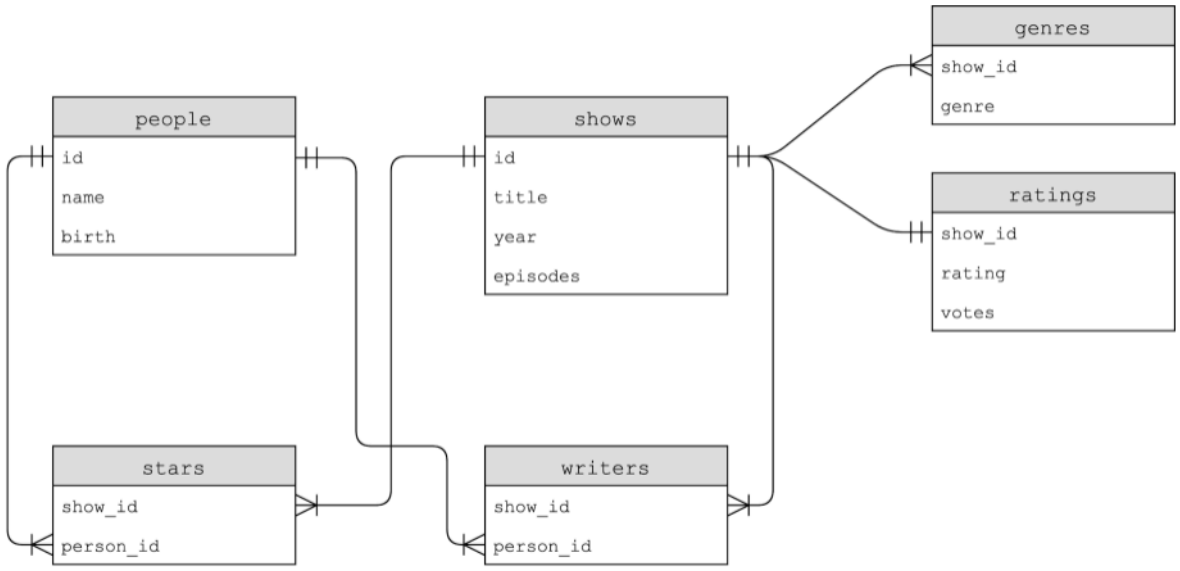
ATUALIZAR

- Também podemos utilizar o `UPDATE` comando para atualizar seus dados.
- Por exemplo, você pode executar `UPDATE favorites SET language = 'SQL', problem = 'Fiftyville';`. Isso resultará na sobrescrita de todas as instruções anteriores onde C e Scratch eram as linguagens de programação preferidas.
- Note que essas consultas têm um poder imenso. Portanto, em um cenário real, você deve considerar quem tem permissão para executar determinados comandos e se você possui backups disponíveis!

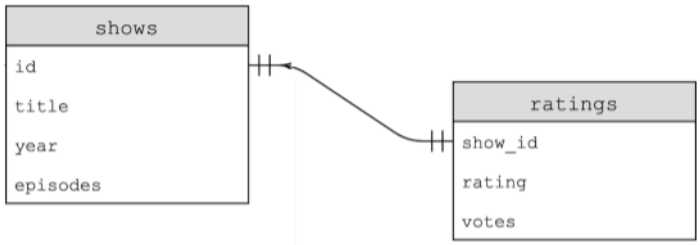
IMDb

- Podemos imaginar um banco de dados que gostaríamos de criar para catalogar vários programas de TV. Poderíamos criar uma planilha com colunas como `title` " `star` estrelas", `star` " `star` estrelas", `star` "estrelas" e mais estrelas. Um problema com essa abordagem é que ela desperdiça muito espaço. Alguns programas podem ter apenas uma estrela. Outros podem ter dezenas.

- Poderíamos dividir nosso banco de dados em várias planilhas. Poderíamos ter uma `shows` planilha "Pessoas", uma `stars` planilha "Programas" e uma `people` planilha "Programas". Na `people` planilha "Pessoas", cada pessoa teria um ID único `id`. Na `shows` planilha "Programas", cada programa também teria um ID único `id`. Em uma terceira planilha chamada " `stars` Programas", poderíamos relacionar como cada programa tem pessoas associadas a ele, usando um ID `show_id` e um ID `person_id`. Embora isso represente uma melhoria, não é um banco de dados ideal.
- O IMDb oferece um banco de dados de pessoas, programas, escritores, estrelas, gêneros e avaliações. Cada uma dessas tabelas está relacionada às outras da seguinte forma:



- Após o download `shows.db` (https://cdn.cs50.net/2024/fall/lectures/7/src7/imdb/shows.db), você pode executar o comando `sqlite3 shows.db` na janela do terminal.
- Vamos nos concentrar na relação entre duas tabelas dentro do banco de dados, chamadas `table1` `shows` e `table2` `ratings`. A relação entre essas duas tabelas pode ser ilustrada da seguinte forma:



- Para ilustrar a relação entre essas tabelas, podemos executar o seguinte comando: `SELECT * FROM ratings LIMIT 10;`. Examinando a saída, podemos executar `SELECT * FROM shows LIMIT 10;`.
- Ao analisarmos os dados de `show` `shows` e `rating` `rating`, podemos observar que existe uma relação direta entre eles: cada programa possui uma classificação específica.
- Para entender o banco de dados, ao executá-lo, `.schema` você encontrará não apenas cada uma das tabelas, mas também os campos individuais dentro de cada uma delas.
- Mais especificamente, você pode executar o comando `.schema shows` para entender os campos internos `shows`. Você também pode executar o comando `.schema ratings` para visualizar os campos internos `ratings`.
- Como você pode ver, `show_id` existe em todas as tabelas. Na `shows` tabela, é simplesmente chamado de `id`. Este campo comum entre todos os campos é chamado de *chave* . As chaves primárias são usadas para identificar um registro único em uma tabela. *As chaves estrangeiras* são usadas para construir relacionamentos entre tabelas, apontando para a chave primária em outra tabela. Você pode ver no esquema de `ratings` que `show_id` é uma chave estrangeira que referencia `id` em `shows`.
- Ao armazenar dados em um banco de dados relacional, como descrito acima, os dados podem ser armazenados de forma mais eficiente.
- No *SQLite* , temos cinco tipos de dados, incluindo:

```
BLOB       -- binary large objects that are groups of ones and zeros
INTEGER    -- an integer
NUMERIC    -- for numbers that are formatted specially like dates
REAL       -- like a float
TEXT       -- for strings and the like
```

- Além disso, as colunas podem ser configuradas para adicionar restrições especiais:

```
NOT NULL
UNIQUE
```

- Podemos explorar ainda mais esses dados para entender essas relações. Execute `SELECT * FROM ratings;`. Há muitas avaliações!
- Podemos restringir ainda mais esses dados executando a seguinte consulta `SELECT show_id FROM ratings WHERE rating >= 6.0 LIMIT 10;`. A partir dessa consulta, você pode ver que 10 programas são apresentados. No entanto, não sabemos qual programa cada um `show_id` representa.
- Você pode descobrir quais são esses programas executando o seguinte: `SELECT * FROM shows WHERE id = 626124;`
- Podemos tornar nossa consulta ainda mais eficiente executando o seguinte comando:

```
SELECT title
FROM shows
WHERE id IN (
    SELECT show_id
    FROM ratings
    WHERE rating >= 6.0
    LIMIT 10
)
```

  Observe que esta consulta aninha duas consultas. Uma consulta interna é usada por uma consulta externa.

`JOIN`s

- Estamos extraindo dados de `shows` e `ratings`. Observe como ambos `shows` e `ratings` têm um `id` em comum.
- Como podemos combinar tabelas temporariamente? As tabelas podem ser unidas usando o `JOIN` comando.
- Execute o seguinte comando:

```
SELECT * FROM shows
  JOIN ratings on shows.id = ratings.show_id
  WHERE rating >= 6.0
  LIMIT 10;
```

  Note que isso resulta em uma tabela mais ampla do que as que vimos anteriormente.

- Enquanto as consultas anteriores ilustraram a relação *um-para-um* entre essas chaves, vamos examinar algumas relações *um-para-muitos* . Concentrando-se na `genres` tabela, execute o seguinte comando:

```
SELECT * FROM genres
LIMIT 10;
```

  Observe como isso nos dá uma ideia dos dados brutos. Você pode notar que um programa tem três valores. Essa é uma relação de um para muitos.

- Podemos aprender mais sobre a `genres` tabela digitando `.schema genres`.

- Execute o seguinte comando para saber mais sobre as diversas comédias presentes no banco de dados:

```
SELECT title FROM shows
WHERE id IN (
  SELECT show_id FROM genres
  WHERE genre = 'Comedy'
  LIMIT 10
);
```

  Observe como isso gera uma lista de comédias, incluindo *Catweazle* .

- Para saber mais sobre o Catweazle, junte-se a várias tabelas através de uma junção:

```
SELECT * FROM shows
JOIN genres
ON shows.id = genres.show_id
WHERE id = 63881;
```

  Note que isso resulta em uma tabela temporária. Não há problema em ter uma tabela duplicada.

- Em contraste com os relacionamentos um-para-um e um-para-muitos, podem existir relacionamentos *muitos-para-muitos* .

- Podemos aprender mais sobre a série *The Office* e os atores que a compõem executando o seguinte comando:

```
SELECT name FROM people WHERE id IN
    (SELECT person_id FROM stars WHERE show_id =
        (SELECT id FROM shows WHERE title = 'The Office' AND year = 2005));
```

  Observe que isso resulta em uma tabela que inclui os nomes de várias estrelas por meio de consultas aninhadas.

- Encontramos todas as séries em que Steve Carell atuou:

```
SELECT title FROM shows WHERE id IN
    (SELECT show_id FROM stars WHERE person_id =
        (SELECT id FROM people WHERE name = 'Steve Carell'));
```

  Isso resulta em uma lista de títulos de séries em que Steve Carell atuou.

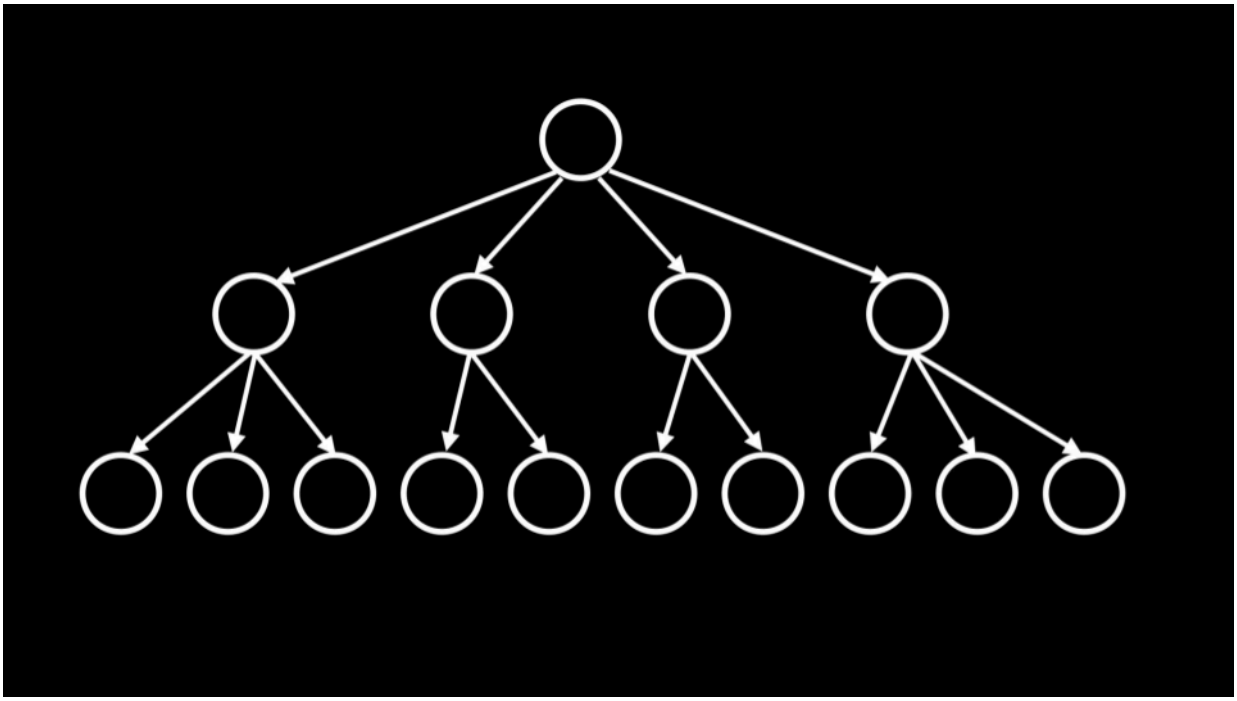
- Isso também poderia ser expresso desta forma:

```
SELECT title FROM shows, stars, people
WHERE shows.id = stars.show_id
AND people.id = stars.person_id
AND name = 'Steve Carell';
```

- O `%` operador curinga pode ser usado para encontrar todas as pessoas cujos nomes começam com `Steve C` um, podendo-se empregar a sintaxe `SELECT * FROM people WHERE name LIKE 'Steve C%';` .

Índices

- Embora os bancos de dados relacionais tenham a capacidade de serem mais rápidos e robustos do que a utilização de um `CSV` arquivo, os dados podem ser otimizados dentro de uma tabela usando *índices* .
- Os índices podem ser utilizados para acelerar nossas consultas.
- Podemos monitorar a velocidade de nossas consultas executando- `.timer on` as em `sqlite3` .
- Para entender como os índices podem acelerar nossas consultas, execute o seguinte comando: `SELECT * FROM shows WHERE title = 'The Office';` Observe o tempo exibido após a execução da consulta.
- Em seguida, podemos criar um índice com a sintaxe `CREATE INDEX title_index ON shows (title);` . Isso instrui o sistema `sqlite3` a criar um índice e executar algumas otimizações internas especiais relacionadas a essa coluna `title` .
- Isso criará uma estrutura de dados chamada *Árvore B* , uma estrutura de dados semelhante a uma árvore binária. No entanto, ao contrário de uma árvore binária, pode haver mais de dois nós filhos.



- Além disso, podemos criar índices da seguinte forma:

```
CREATE INDEX name_index ON people (name);
CREATE INDEX person_index ON stars (person_id);
```

- Ao executar a consulta, você perceberá que ela é executada muito mais rapidamente!

```
SELECT title FROM shows WHERE id IN
    (SELECT show_id FROM stars WHERE person_id =
        (SELECT id FROM people WHERE name = 'Steve Carell'));
```

- Infelizmente, indexar todas as colunas resultaria na utilização de mais espaço de armazenamento. Portanto, há uma contrapartida para uma maior velocidade.

Utilizando SQL em Python

- Para auxiliar no trabalho com SQL neste curso, a biblioteca CS50 pode ser utilizada da seguinte forma em seu código:

```
from cs50 import SQL
```

- Assim como em usos anteriores da biblioteca CS50, esta biblioteca auxiliará nas etapas complexas de utilização de SQL em seu código Python.
- Você pode ler mais sobre a funcionalidade SQL da biblioteca CS50 na documentação (https://cs50.readthedocs.io/libraries/cs50/python/#cs50.SQL) .
- Utilizando nosso novo conhecimento de SQL, agora podemos aproveitar o Python em conjunto.
- Modifique seu código da `favorites.py` seguinte forma:

```
# Searches database popularity of a problem

from cs50 import SQL

# Open database
db = SQL("sqlite:///favorites.db")

# Prompt user for favorite
favorite = input("Favorite: ")

# Search for title
rows = db.execute("SELECT COUNT(*) AS n FROM favorites WHERE language = ?", fa

# Get first (and only) row
row = rows[0]

# Print popularity
print(row["n"])
```

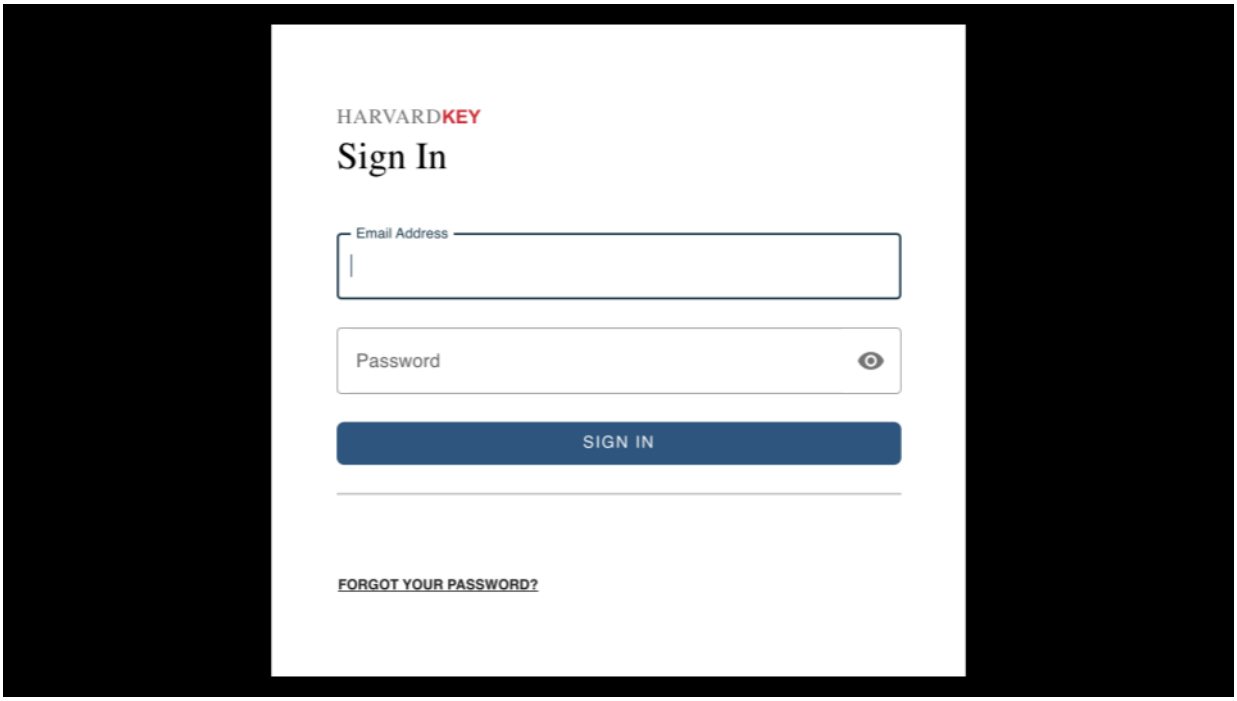
Observe que `db = SQL("sqlite:///favorites.db")` o Python recebe o caminho do arquivo de banco de dados. Em seguida, a linha que começa com `rows` `execute` executa comandos SQL utilizando ` `db.execute`. De fato, esse comando passa a sintaxe entre aspas para a `db.execute` função. Podemos executar qualquer comando SQL usando essa sintaxe. Além disso, observe que `rows` `rows` é retornado como uma lista de dicionários. Nesse caso, há apenas um resultado, uma linha, retornada à lista de linhas como um dicionário.

Condições da corrida

- A utilização de SQL pode, por vezes, resultar em alguns problemas.
- É possível imaginar uma situação em que vários usuários estejam acessando o mesmo banco de dados e executando comandos ao mesmo tempo.
- Isso pode resultar em falhas onde o código é interrompido pelas ações de outras pessoas. Isso pode resultar em perda de dados.
- Recursos integrados do SQL, como `BEGIN TRANSACTION`, `COMMIT`, e , `ROLLBACK` ajudam a evitar alguns desses problemas de condição de corrida.

Ataques de injeção de SQL

- Agora, ainda considerando o código acima, você pode estar se perguntando o que os `?` pontos de interrogação fazem. Um dos problemas que podem surgir em aplicações reais de SQL é o que chamamos de *ataque de injeção* . Um ataque de injeção ocorre quando um agente malicioso insere código SQL malicioso.
- Por exemplo, considere a seguinte tela de login:



- Sem as devidas proteções em nosso próprio código, um agente malicioso poderia executar código malicioso. Considere o seguinte:

```
rows = db.execute("SELECT COUNT(*) FROM users WHERE username = ? AND password
```

  Note que, como o parâmetro `?` já está definido, a validação pode ser executada `favorite` antes que ele seja aceito cegamente pela consulta.
- Você nunca deve utilizar strings formatadas em consultas como as acima, nem confiar cegamente na entrada do usuário.
- Utilizando a biblioteca CS50, a biblioteca irá *higienizar* e remover quaisquer caracteres potencialmente maliciosos.

Resumindo

Nesta lição, você aprendeu mais sobre a sintaxe do Python. Além disso, aprendeu como integrar esse conhecimento com dados em formato de arquivos planos e bancos de dados relacionais. Por fim, aprendeu sobre *SQL* . Especificamente, discutimos...

- Bancos de dados de arquivo simples
- Bancos de dados relacionais
- Comandos SQL como `SELECT`, `CREATE`, `INSERT`, `DELETE`, e `UPDATE`.
- Chaves primárias e estrangeiras
- `JOIN`s
- Índices
- Utilizando SQL em Python
- Condições de corrida
- ataques de injeção de SQL

Até a próxima!