












Isso é CS50

Introdução do CS50 à Ciência da Computação

OpenCourseWare

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

David J. Malan (<https://cs.harvard.edu/malan/>)
malan@harvard.edu

 (<https://www.clubhouse.com/@davidjmalan>)  (<https://www.facebook.com/dmalan>) 
(<https://github.com/dmalan>)  (<https://www.instagram.com/davidjmalan/>) 
(<https://www.linkedin.com/in/malan/>)  (<https://orcid.org/0000-0001-5338-2522>) 
(<https://www.quora.com/profile/David-J-Malan>)  (<https://www.reddit.com/user/davidjmalan>) 
(<https://www.tiktok.com/@davidjmalan>)  (<https://davidjmalan.t.me/>) 
(<https://twitter.com/davidjmalan>)

Lecture 7

- [Welcome!](#)
- [Flat-File Database](#)
- [Relational Databases](#)
- [IMDb](#)
- [JOIN s](#)
- [Indexes](#)
- [Using SQL in Python](#)
- [Race Conditions](#)
- [SQL Injection Attacks](#)
- [Summing Up](#)

Welcome!

- In previous weeks, we introduced you to Python, a high-level programming language that utilized the same building blocks we learned in C.
- Nesta semana, continuaremos com mais sintaxe relacionada ao Python.
- Além disso, estaremos integrando esse conhecimento com dados.
- Por fim, discutiremos *SQL* ou *Linguagem de consulta estruturada*.
- No geral, um dos objetivos deste curso é aprender a programar em geral – não apenas como programar nas linguagens descritas neste curso.

Banco de Dados de Arquivo Simples

- Como você provavelmente já viu antes, os dados geralmente podem ser descritos em padrões de colunas e tabelas.
- Planilhas como as criadas no Microsoft Excel e no Google Sheets podem ser geradas em um `csv` arquivo *de valores separados por vírgulas*.
- Se você olhar para um `csv` arquivo, notará que o arquivo é simples, pois todos os nossos dados são armazenados em uma única tabela representada por um arquivo de texto. Chamamos essa forma de dados de *banco de dados de arquivo simples*.
- Python vem com suporte nativo para `csv` arquivos.
- 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 é importada. Além disso, criamos um `reader` que conterá o resultado de `csv.reader(file)`. A `csv.reader` função lê cada linha do arquivo e em nosso código armazenamos os resultados em `reader`. `print(row[1])`, portanto, imprimirá o idioma do `favorites.csv` arquivo.

- Você pode melhorar seu código da seguinte maneira:

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

- Python também permite que você indexe pelas chaves de uma lista. Modifique seu código da seguinte maneira:

```
# 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"])
```

Observe que este exemplo utiliza diretamente a `language` chave na instrução `print`.

- 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 contado usando `if` instruções.

- Python nos permite usar um dicionário para contar o `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 `counts` com a chave `favorite` é incrementado quando já existe. Se não existir, definimos `counts[favorite]` e definimos como 1. Além disso, a string formatada foi aprimorada para apresentar o `counts[favorite]`.

- Python também permite a classificação `counts`. Melhore seu código da seguinte maneira:

```

# 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 `sorted(counts)` na parte inferior do código.

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

```

# Sorts favorites by value

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

def get_value(language):
    return counts[language]

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

```

Observe que uma função chamada `get_value` é criada e que a própria função é passada como um argumento para a `sorted` função. O `key` argumento permite que você diga ao Python o método que deseja usar para classificar os itens.

- Python tem uma capacidade única que não vimos até agora: permite a utilização de funções *anônimas* ou `lambda`. Essas funções podem ser utilizadas quando você não quer se preocupar em criar uma função totalmente diferente. Observe a seguinte modificação:

```

# Sorts favorites by value using lambda function

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=lambda language: counts[language], reverse=True):
    print(f"{favorite}: {counts[favorite]}")

```

Observe que a `get_value` função foi removida. Em vez disso, `lambda language: counts[language]` faz em uma linha o que nossa função anterior de duas linhas fazia.

- Podemos alterar a coluna que estamos examinando, concentrando-nos em nosso problema favorito:

```

# Favorite problem instead of favorite language

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["problem"]

```

```

    if favorite in counts:
        counts[favorite] += 1
    else:
        counts[favorite] = 1

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

```

Observe que `problem` substituído `language`.

- E se quiséssemos permitir que os usuários forneçam entrada diretamente no terminal? Podemos modificar nosso código, aproveitando nosso conhecimento anterior sobre a entrada do usuário:

```

# Favorite problem instead of favorite language

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["problem"]
        if favorite in counts:
            counts[favorite] += 1
        else:
            counts[favorite] = 1

# Print count
favorite = input("Favorite: ")
if favorite in counts:
    print(f"{favorite}: {counts[favorite]}")

```

Observe como nosso código é compacto em comparação com nossa experiência em C.

Bancos de Dados Relacionais

- Google, Twitter e Meta usam bancos de dados relacionais para armazenar suas informações em escala.
- 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 SQL no terminal digitando `sqlite3 favorites.db`. Ao ser solicitado, concordaremos que queremos criar `favorites.db` pressionando `y`.
- Você notará um prompt diferente, pois agora estamos dentro de um programa chamado `sqlite3`.
- Podemos colocar `sqlite3` no `csv` modo digitando `.mode csv`. Então, podemos importar nossos dados de nosso `csv` arquivo digitando `.import favorites.csv favorites`. Parece que nada aconteceu!

- Podemos digitar `.schema` para ver 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;` o que irá iterar cada linha em `favorites`.
- Você pode obter um subconjunto dos dados usando o comando `SELECT language FROM favorites;`.
- SQL suporta muitos comandos para acessar dados, incluindo:

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

- Por exemplo, você pode digitar `SELECT COUNT(language) FROM favorites;`. Além disso, você pode digitar `SELECT DISTINCT(language) FROM favorites;` para obter uma lista dos idiomas individuais no banco de dados. Você pode até digitar `SELECT COUNT(DISTINCT(language)) FROM favorites;` para obter uma contagem deles.

```
# Searches database popularity of a problem

import csv

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(*) FROM favorites WHERE problem LIKE ?", "%" + favorite +

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

# Print popularity
print(row["COUNT(*)"])
```

- 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
```

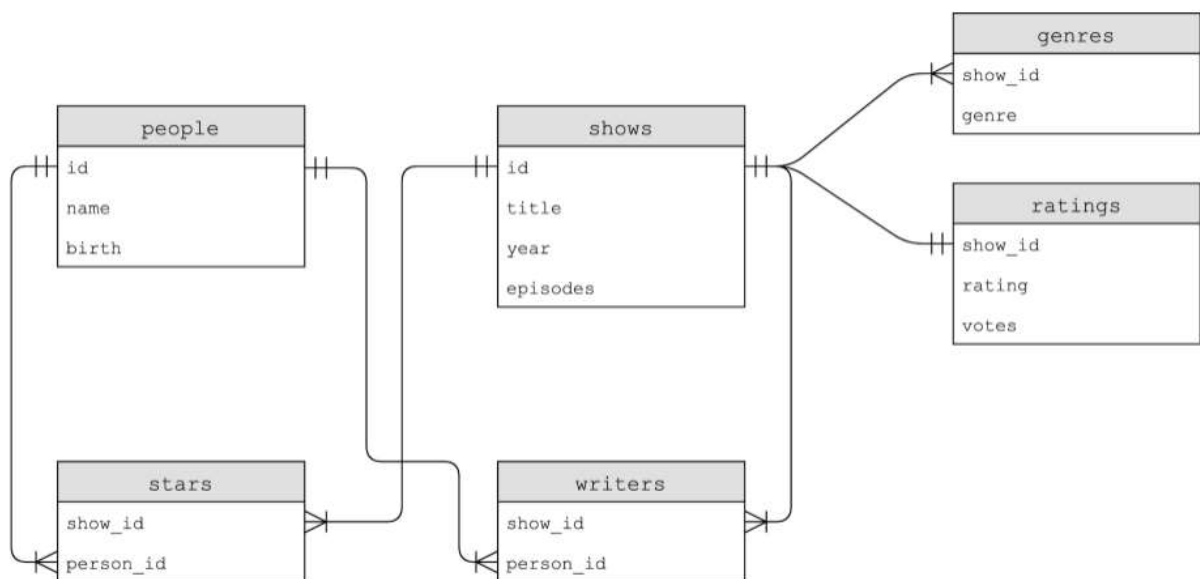
Observe que usamos `--` para escrever um comentário em SQL.

- 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 = 'Mario';`. Observe como o `AND` é utilizado para restringir 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`.
- Podemos também `INSERT` 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')`;
- Também podemos utilizar o `UPDATE` comando para atualizar seus dados.
- Por exemplo, você pode executar `UPDATE favorites SET language = 'C++' WHERE language = 'C'`; Isso resultará na substituição de todas as instruções anteriores em que C era a linguagem de programação favorita.
- Observe que essas consultas têm um poder imenso. Da mesma forma, no cenário do mundo real, você deve considerar quem tem permissões para executar determinados comandos.
- `DELETE` permite que você exclua partes de seus dados. Por exemplo, você poderia `DELETE FROM favorites WHERE problem = 'Tideman'`;

IMDb

- IMDb oferece um banco de dados de pessoas, shows, escritores, gêneros e avaliações. Cada uma dessas tabelas está relacionada entre si da seguinte maneira:



- Após o download `shows.db` (<https://github.com/cs50/lectures/blob/2022/fall/7/src7/imdb/shows.db>), você pode executar `sqlite3 shows.db` na janela do seu terminal.
- Ao executar, `.schema` você encontrará não apenas cada uma das tabelas, mas os campos individuais dentro de cada um desses campos.
- Como você pode ver pela imagem acima, `shows` tem um `id` campo. A `genres` tabela possui um `show_id` campo que possui dados comuns entre ela e a `shows` tabela.
- Como podem ver também na imagem acima, `show_id` existe em todas as tabelas. Na `shows` tabela, ele é chamado simplesmente de `id`. Esse campo comum entre todos os campos é chamado de *chave*. As chaves primárias são usadas para identificar um registro exclusivo em uma tabela. *Chaves estrangeiras* são usadas para construir relacionamentos entre tabelas apontando para a chave primária em outra tabela.

- Ao armazenar dados em um banco de dados relacional, como 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 definidas para adicionar restrições especiais:

```
NOT NULL
UNIQUE
```

- Para ilustrar melhor o relacionamento entre essas tabelas, poderíamos executar o seguinte comando: `SELECT * FROM people LIMIT 10;`. Examinando a saída, poderíamos executar `SELECT * FROM shows LIMIT 10;`. Além disso, poderíamos executar `SELECT * FROM stars LIMIT 10;`. `show_id` é uma chave estrangeira nesta consulta final porque `show_id` corresponde ao `id` campo único em `shows`. `person_id` corresponde ao `id` campo único na `people` coluna.
- Podemos brincar ainda mais com esses dados para entender essas relações. Execute `SELECT * FROM genres;`. Existem muitos gêneros!
- Podemos limitar ainda mais esses dados executando `SELECT * FROM genres WHERE genre = 'Comedy' LIMIT 10;`. A partir desta consulta, você pode ver que existem 10 shows apresentados.
- Você pode descobrir quais programas são esses executando `SELECT * FROM shows WHERE id = 626124;`
- Podemos aprofundar nossa consulta para ser mais eficiente executando

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

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

- Podemos refinar ainda mais executando

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

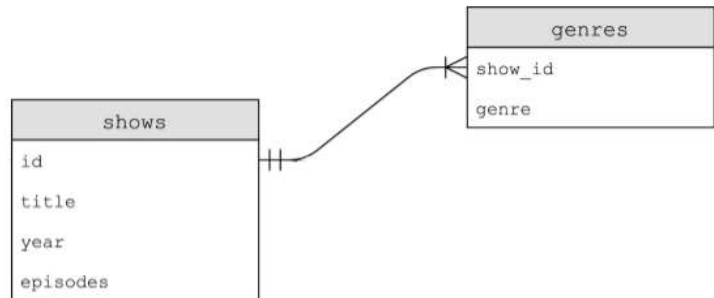
- E se você quisesse encontrar todos os programas estrelados por Steve Carell? Você poderia executar `SELECT * FROM people WHERE name = 'Steve Carell';` Você encontraria seu indivíduo `id`. Você pode usar isso `id` para localizar muitos `shows` em que ele aparece. No entanto, seria tedioso tentar isso um por um. Como poderíamos seguir nossas consultas para tornar isso mais simplificado? Considere o seguinte:

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

Observe que essa consulta longa resultará em um resultado final útil para descobrir a resposta à nossa pergunta.

JOINS

- Considere as duas tabelas a seguir:



- Como poderíamos 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 title = 'The Office';
```

- Agora você pode ver todos os programas que foram chamados *de The Office*.
- Da mesma forma, você pode aplicar `JOIN` à nossa consulta de Steve Carell acima executando o seguinte:

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

Observe como cada `JOIN` comando nos diz quais colunas estão alinhadas entre si.

- Isso poderia ser implementado de forma semelhante da seguinte forma:

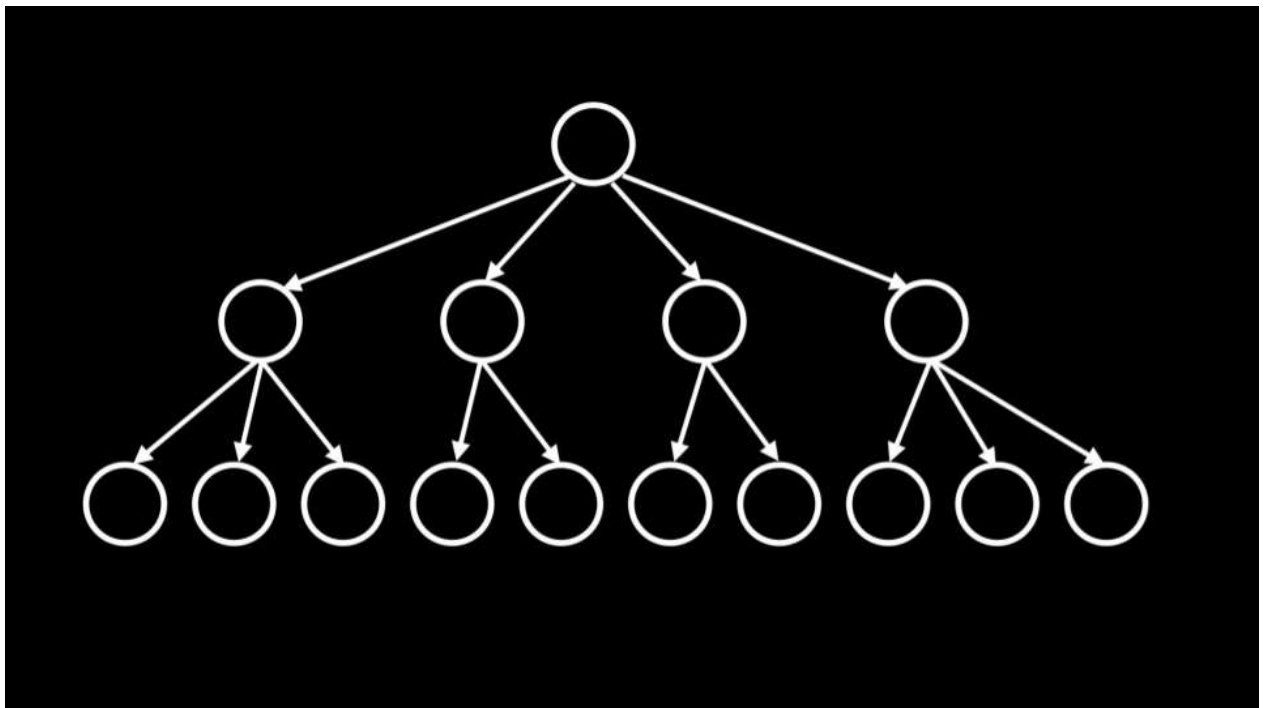
```
SELECT title FROM people, stars, shows
WHERE people.id = stars.person_id
AND stars.show_id = shows.id
AND name = 'Steve Carell';
```

Observe que isso atinge os mesmos resultados.

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

Índices

- Embora os bancos de dados relacionais tenham a capacidade de ser mais rápidos e robustos do que a utilização de um `csv` arquivo, os dados podem ser otimizados em uma tabela usando *índices*.
- Os índices podem ser utilizados para acelerar nossas consultas.
- Podemos acompanhar a velocidade de nossas consultas executando `.timer on` em `sqlite3`.
- Para entender como os índices podem acelerar nossas consultas, execute o seguinte: `SELECT * FROM shows WHERE title = 'The Office';` Observe o tempo exibido após a execução da consulta.
- Então, podemos criar um índice com a sintaxe `CREATE INDEX title_index on shows (title);`. Isso diz `sqlite3` para criar um índice e executar algumas otimizações ocultas especiais relacionadas a esta coluna `title`.
- Isso criará uma estrutura de dados chamada *B Tree*, uma estrutura de dados que se parece com uma árvore binária. No entanto, ao contrário de uma árvore binária, pode haver mais de duas notas filhas.



- Executando a query `SELECT * FROM shows WHERE title = 'The Office';`, você notará que a query roda muito mais rápido!
- Unfortunately, indexing all columns would result in utilizing more storage space. Therefore, there is a tradeoff for enhanced speed.

Using SQL in Python

- To assist in working with SQL in this course, the CS50 Library can be utilized as follows in your code:

```
from cs50 import SQL
```

- Similar to previous uses of the CS50 Library, this library will assist with the complicated steps of utilizing SQL within your Python code.

- You can read more about the CS50 Library's SQL functionality in the [documentation](https://cs50.readthedocs.io/libraries/cs50/python/#cs50.SQL) (<https://cs50.readthedocs.io/libraries/cs50/python/#cs50.SQL>).
- Recall where we last left off in `favorites.py`. Your code should appear as follows:

```
# Favorite problem instead of favorite language

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["problem"]
        if favorite in counts:
            counts[favorite] += 1
        else:
            counts[favorite] = 1

# Print count
favorite = input("Favorite: ")
if favorite in counts:
    print(f"{favorite}: {counts[favorite]}")
```

- Modify your code as follows:

```
# Searches database popularity of a problem

import csv

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(*) FROM favorites WHERE problem LIKE ?", "%" + favorite +

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

# Print popularity
print(row["COUNT(*)"])
```

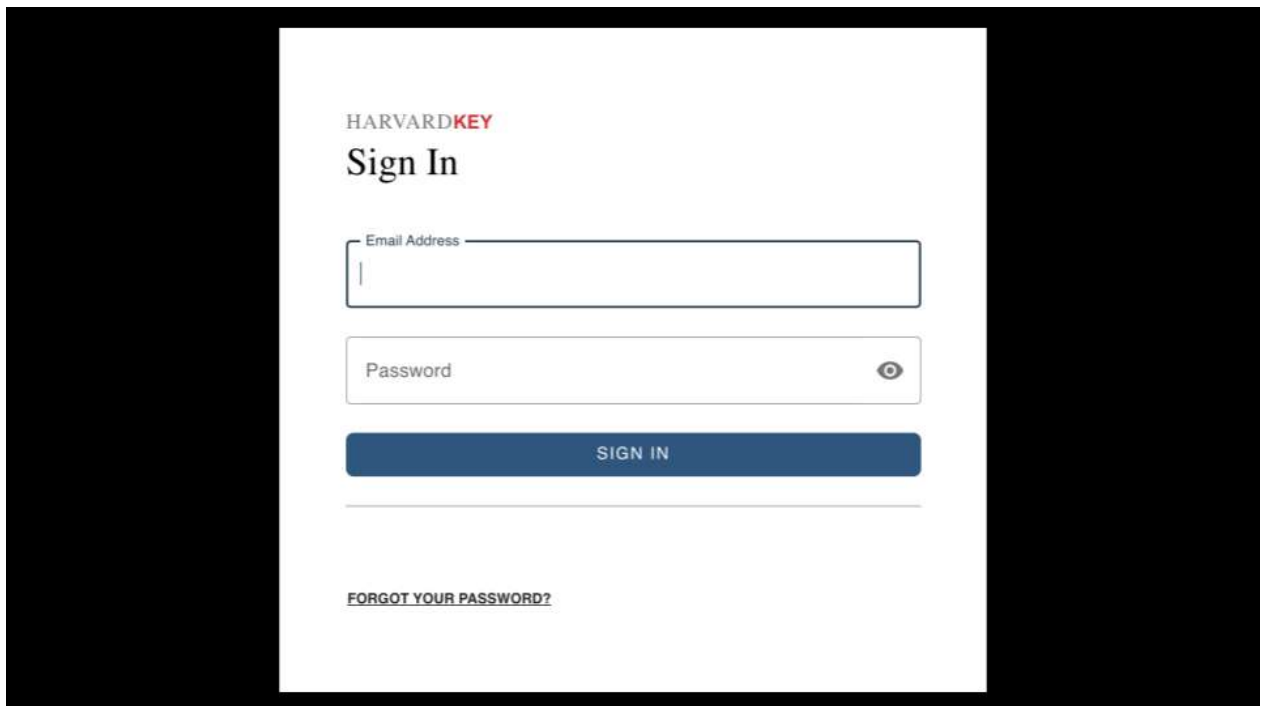
Notice that `db = SQL("sqlite:///favorites.db")` provide Python the location of the database file. Then, the line that begins with `rows` executes SQL commands utilizing `db.execute`. Indeed, this command passes the syntax within the quotation marks to the `db.execute` function. We can issue any SQL command using this syntax. Further, notice that `rows` is returned as a list of dictionaries. In this case, there is only one result, one row, returned to the `rows` list as a dictionary.

Race Conditions

- Utilization of SQL can sometimes result in some problems.
- You can imagine a case where multiple users could be accessing the same database and executing commands at the same time.
- This could result in glitches where code is interrupted by other people's actions. This could result in a loss of data.
- Built-in SQL features such as `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` help avoid some of these race condition problems.

SQL Injection Attacks

- Now, still considering the code above, you might be wondering what the `?` question marks do above. One of the problems that can arise in real-world applications of SQL is what is called an *injection attack*. An injection attack is where a malicious actor could input malicious SQL code.
- For example, consider a login screen as follows:



- Without the proper protections in our own code, a bad actor could run malicious code. Consider the following:

```
rows = db.execute("SELECT COUNT(*) FROM favorites WHERE problem LIKE ?", "%" + favorite +
```

Notice that because the `?` is in place, validation can be run on `favorite` before it is blindly accepted by the query.

- Você nunca deseja utilizar strings formatadas em consultas como acima ou confiar cegamente na entrada do usuário.
- Utilizando a Biblioteca CS50, a biblioteca *limpará* e removerá quaisquer caracteres potencialmente maliciosos.

Resumindo

Nesta lição, você aprendeu mais sintaxe relacionada ao Python. Além disso, você aprendeu como integrar esse conhecimento com dados na forma de arquivos simples e bancos de dados relacionais. Finalmente, você aprendeu sobre *SQL*. Especificamente, discutimos...

- Bancos de dados de arquivo simples
- Bancos de dados relacionais
- SQL
- JOINs
- Índices
- Usando SQL em Python
- Condições da corrida
- Ataques de injeção SQL

Vejo você na próxima vez!