

# This is CS50


CS50's Introduction to  
Computer Science


OpenCourseWare

Donate 

David J. Malan  
malan@harvard.edu



 CS50x Puzzle Day 2023

CS50 Educator Workshop  
Gallery of Final Projects 

Week 0 Scratch

Week 1 C

Week 2 Arrays

Week 3 Algorithms

Week 4 Memory

Week 5 Data Structures

Week 6 Python

Week 7 SQL

Week 8 HTML, CSS, and JavaScript

## 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.
- This week, we will be continuing more syntax related to Python.
- Further, we will be integrating this knowledge with data.
- Finally, we will be discussing *SQL* or *Structured Query Language*.
- Overall, one of the goals of this course is to learn to program generally – not simply how to program in the languages described in this course.

### Flat-File Database

- As you have likely seen before, data can often be described in patterns of columns and tables.
- Spreadsheets like those created in Microsoft Excel and Google Sheets can be outputted to a `csv` or *comma-separated values* file.
- If you look at a `csv` file, you'll notice that the file is flat in that all of our data is stored in a single table represented by a text file. We call this form of data a *flat-file database*.
- Python comes with native support for `csv` files.
- In your terminal window, type `code favorites.py` and write code as follows:

```
# 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])
```

Notice that the `csv` library is imported. Further, we created a `reader` that will hold the result of `csv.reader(file)`. The `csv.reader` function reads each row from the file, and in our code we store the results in `reader`. `print(row[1])`, therefore, will print the language from the `favorites.csv` file.

- You can improve your code as follows:

```
# 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)
```

Notice that `favorite` is stored and then printed. Also notice that we use the `next` function to skip to the next line of our reader.

- Python also allows you to index by the keys of a list. Modify your code as follows:

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

```
# 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"])
```

Notice that this example directly utilizes the `language` key in the print statement.

- To count the number of favorite languages expressed in the `csv` file, we can do the following:

```
# 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}")
```

Notice that each language is counted using `if` statements.

- Python allows us to use a dictionary to count the `counts` of each language. Consider the following improvement upon our code:

```
# 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]}")
```

Notice that the value in `counts` with the key `favorite` is incremented when it exists already. If it does not exist, we define `counts[favorite]` and set it to 1. Further, the formatted string has been improved to present the `counts[favorite]`.

- Python also allows sorting `counts`. Improve your code as follows:

```
# 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):
```

Notice the `sorted(counts)` at the bottom of the code.

- If you look at the parameters for the `sorted` function in the Python documentation, you will find it has many built-in parameters. You can leverage some of these built-in parameters as follows:

```
# 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]}")
```

Notice that a function called `get_value` is created, and that the function itself is passed in as an argument to the `sorted` function. The `key` argument allows you to tell Python the method you wish to use to sort items.

- Python has a unique ability that we have not seen to date: It allows for the utilization of *anonymous* or `lambda` functions. These functions can be utilized when you want to not bother creating an entirely different function. Notice the following modification:

```
# 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]}")
```

Notice that the `get_value` function has been removed. Instead, `lambda language: counts[language]` does in one line what our previous two-line function did.

- We can change the column we are examining, focusing on our favorite problem instead:

```
# 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]}")
```

Notice that `problem` replaced `language`.

- What if we wanted to allow users to provide input directly in the terminal? We can modify our code, leveraging our previous knowledge

```
# 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]}")
```

Notice how compact our code is compared to our experience in C.

## Relational Databases

- Google, Twitter, and Meta all use relational databases to store their information at scale.
- Relational databases store data in rows and columns in structures called *tables*.
- SQL allows for four types of commands:

```
Create
Read
Update
Delete
```

- These four operations are affectionately called *CRUD*.
- We can create a SQL database at the terminal by typing `sqlite3 favorites.db`. Upon being prompted, we will agree that we want to create `favorites.db` by pressing `y`.
- You will notice a different prompt as we are now inside a program called `sqlite3`.
- We can put `sqlite3` into `csv` mode by typing `.mode csv`. Then, we can import our data from our `csv` file by typing `.import favorites.csv favorites`. It seems that nothing has happened!
- We can type `.schema` to see the structure of the database.
- You can read items from a table using the syntax `SELECT columns FROM table`.
- For example, you can type `SELECT * FROM favorites;` which will iterate every row in `favorites`.
- You can get a subset of the data using the command `SELECT language FROM favorites;`.
- SQL supports many commands to access data, including:

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

- For example, you can type `SELECT COUNT(language) FROM favorites;`. Further, you can type `SELECT DISTINCT(language) FROM favorites;` to get a list of the individual languages within the database. You could even type `SELECT COUNT(DISTINCT(language)) FROM favorites;` to get a count of those.

```
# 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(*)"])
```

- SQL offers additional commands we can utilize in our queries:

```

LIMIT      -- limiting the number of responses
GROUP BY   -- grouping responses together

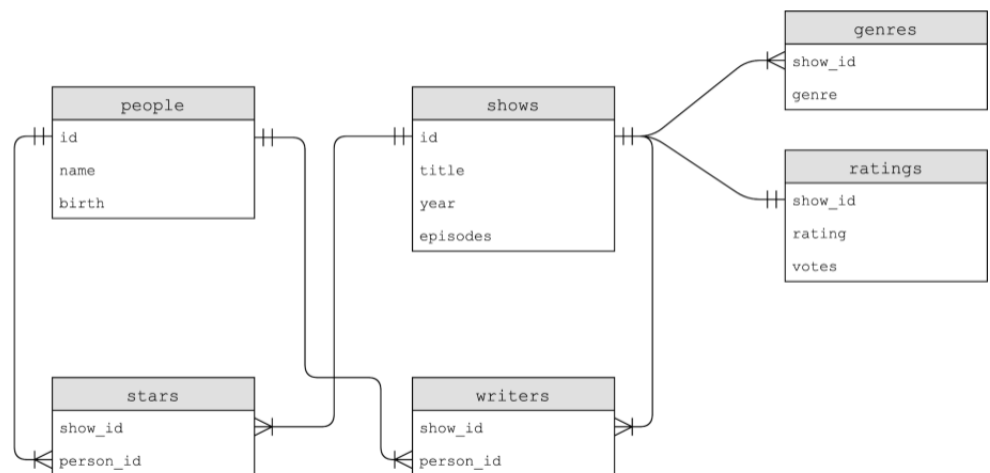
```

Notice that we use `--` to write a comment in SQL.

- For example, we can execute `SELECT COUNT(*) FROM favorites WHERE language = 'C';`. A count is presented.
- Further, we could type `SELECT COUNT(*) FROM favorites WHERE language = 'C' AND problem = 'Mario';`. Notice how the `AND` is utilized to narrow our results.
- Similarly, we could execute `SELECT language, COUNT(*) FROM favorites GROUP BY language;`. This would offer a temporary table that would show the language and count.
- We could improve this by typing `SELECT language, COUNT(*) FROM favorites GROUP BY language ORDER BY COUNT(*);`. This will order the resulting table by the `count`.
- We can also `INSERT` into a SQL database utilizing the form `INSERT INTO table (column...) VALUES(value, ...);`.
- We can execute `INSERT INTO favorites (language, problem) VALUES ('SQL', 'Fiftyville');`.
- We can also utilize the `UPDATE` command to update your data.
- For example, you can execute `UPDATE favorites SET language = 'C++' WHERE language = 'C';`. This will result in overwriting all previous statements where C was the favorite programming language.
- Notice that these queries have immense power. Accordingly, in the real-world setting, you should consider who has permissions to execute certain commands.
- `DELETE` allows you to delete parts of your data. For example, you could `DELETE FROM favorites WHERE problem = 'Tideman';`.

## IMDb

- IMDb offers a database of people, shows, writers, stars, genres, and ratings. Each of these tables is related to one another as follows:



- After downloading `shows.db`, you can execute `sqlite3 shows.db` in your terminal window.
- Upon executing `.schema` you will find not only each of the tables but the individual fields inside each of these fields.
- As you can see by the image above, `shows` has an `id` field. The `genres` table has a `show_id` field which has data that is common between it and the `shows` table.
- As you can see also in the image above, `show_id` exists in all of the tables. In the `shows` table, it is simply called `id`. This common field between all the fields is called a *key*. Primary keys are used to identify a unique record in a table. *Foreign keys* are used to build relationships between tables by pointing to the primary key in another table.
- By storing data in a relational database, as above, data can be more efficiently stored.
- In `sqlite`, we have five datatypes, including:

```

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

```

- Additionally, columns can be set to add special constraints:

```

NOT NULL
UNIQUE

```

- To illustrate the relationship between these tables further, we could execute the following command: `SELECT * FROM people LIMIT 10;`. Examining the output, we could execute `SELECT * FROM shows LIMIT 10;`. Further, we could execute `SELECT * FROM stars LIMIT 10;`. `show_id` is a foreign key in this final query because `show_id` corresponds to the unique `id` field in `shows`. `person_id` corresponds to the unique `id` field in the `people` column.
- We can further play with this data to understand these relationships. Execute `SELECT * FROM genres;`. There are a lot of genres!
- We can further limit this data down by executing `SELECT * FROM genres WHERE genre = 'Comedy' LIMIT 10;`. From this query, you

- You can discover what shows these are by executing `SELECT * FROM shows WHERE id = 626124;`
- We can further our query to be more efficient by executing

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

Notice that this query nests together two queries. An inner query is used by an outer query.

- We can refine further by executing

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

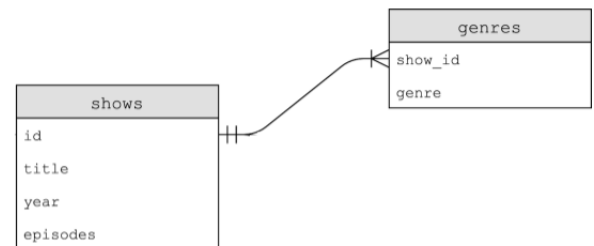
- What if you wanted to find all shows in which Steve Carell stars? You could execute `SELECT * FROM people WHERE name = 'Steve Carell';` You would find his individual `id`. You could utilize this `id` to locate many `shows` in which he appears. However, this would be tedious to attempt this one by one. How could we next our queries to make this more streamlined? Consider the following:

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

Notice that this lengthy query will result in a final result that is useful in discovering the answer to our question.

## JOINS

- Consider the following two tables:



- How could we combine tables temporarily? Tables could be joined together using the `JOIN` command.
- Execute the following command:

```
SELECT * FROM shows
JOIN ratings ON shows.id = ratings.show_id
WHERE title = 'The Office';
```

- Now you can see all the shows that have been called *The Office*.
- You could similarly apply `JOIN` to our Steve Carell query above by executing the following:

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

Notice how each `JOIN` command tells us which columns are aligned to each other columns.

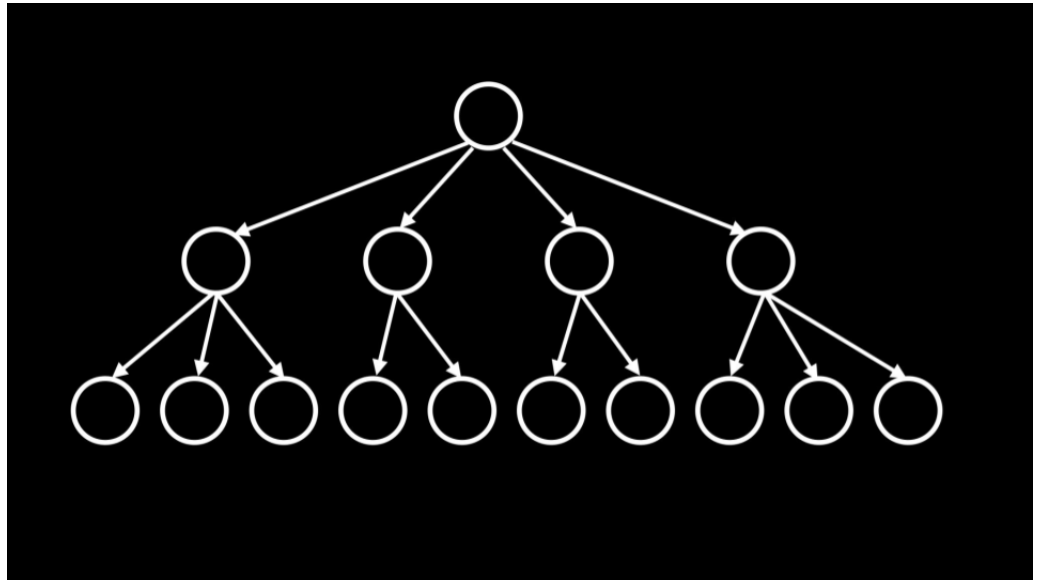
- This could be similarly implemented as follows:

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

- The wildcard `%` operator can be used to find all people whose names start with `Steve`. One could employ the syntax `SELECT * FROM people WHERE name LIKE 'Steve %';`.

## Indexes

- While relational databases have the ability to be more fast and more robust than utilizing a `CSV` file, data can be optimized within a table using *indexes*.
- Indexes can be utilized to speed up our queries.
- We can track the speed of our queries by executing `.timer on` in `sqlite3`.
- To understand how indexes can speed up our queries, run the following: `SELECT * FROM shows WHERE title = 'The Office';`. Notice the time that displays after the query executes.
- Then, we can create an index with the syntax `CREATE INDEX title_index on shows (title);`. This tells `sqlite3` to create an index and perform some special under-the-hood optimization relating to this column `title`.
- This will create a data structure called a *B Tree*, a data structure that looks similar to a binary tree. However, unlike a binary tree, there can be more than two child nodes.



- Running the query `SELECT * FROM shows WHERE title = 'The Office';`, you will notice that the query runs much more quickly!
- 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](#).
- 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:

```
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(*)"])
```

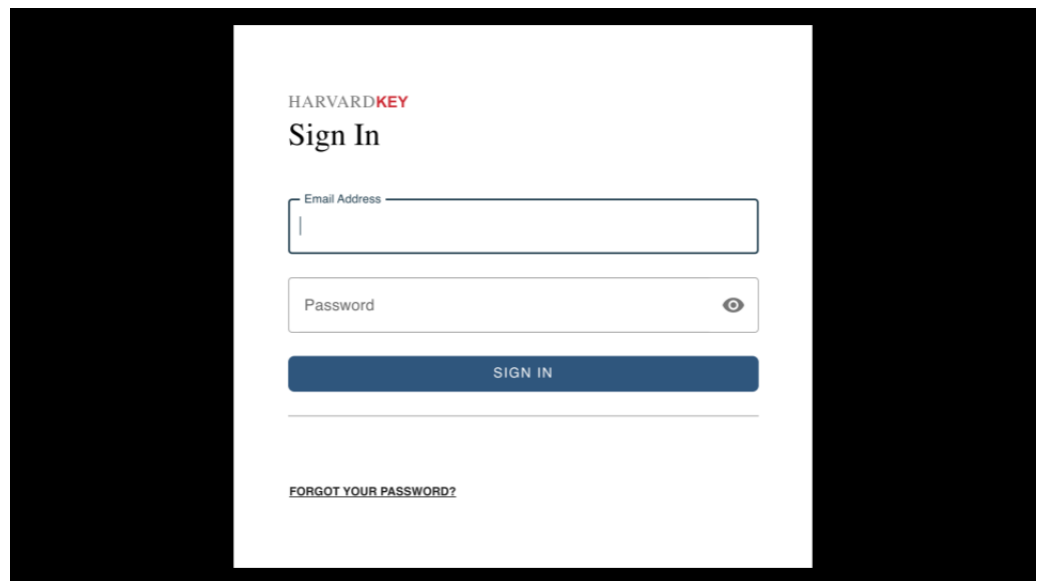
Notice that `db = SQL("sqlite:///favorites.db")` provide Python the location of the database file. Then, the line that begins with `rows` executes SQL commands utilizing `db.execute`. Indeed, this command passes the syntax within the quotation marks to the `db.execute` function. We can issue any SQL 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.

- You never want to utilize formatted strings in queries as above or blindly trust the user's input.
- Utilizing the CS50 Library, the library will *sanitize* and remove any potentially malicious characters.

## Summing Up

In this lesson, you learned more syntax related to Python. Further, you learned how to integrate this knowledge with data in the form of flat-file and relational databases. Finally, you learned about SQL. Specifically, we discussed...

- Flat-file databases
- Relational databases
- SQL





- Using SQL in Python
- Race conditions
- SQL injection attacks

See you next time!