🐍 Python Notes: Days 51-60 (Angela Yu's 100 Days of Code)

Here are the detailed notes for Days 51 through 60, covering advanced browser automation with Selenium and the fundamentals of web development with the Flask framework.

Day 51: Internet Speed Twitter Complaint Bot 🐦

Objective: Build a bot that checks your internet speed and, if it's below a promised threshold, automatically logs into Twitter and posts a complaint.

Key Concepts:

- **Selenium WebDriver:** An automation tool that allows your Python script to control a web browser. It simulates a real user clicking, typing, and navigating through web pages.
- **Locating Elements:** Finding specific HTML elements on a page to interact with them. Selenium provides various methods:
  - `find_element(By.ID, "value")`
  - `find_element(By.NAME, "value")`
  - `find_element(By.XPATH, "xpath_expression")`
  - `find_element(By.CSS_SELECTOR, "css_selector")`
  - `find_element(By.CLASS_NAME, "class_name")`
- **Explicit Waits:** Pausing your script until a certain condition is met (e.g., an element becomes clickable). This is crucial for dealing with modern websites that load content dynamically (asynchronously). Without waits, your script might try to click a button before it has appeared, causing an error.
  - Use `WebDriverWait` combined with `expected_conditions` .

Project Workflow:

1. **Setup:**
   - Install Selenium: `pip install selenium`
   - Download the appropriate `WebDriver` for your browser (e.g., `chromedriver` for Chrome).
   - Store sensitive information (Twitter username, password, promised speeds) in variables or environment variables.
2. **Part 1: Get Internet Speed**
   - The bot navigates to `https://www.speedtest.net/` .
   - It waits for the "GO" button to be clickable and then clicks it.

    - It waits for the results to appear (e.g., wait until the download/upload speed elements are visible). This can take a minute.
    - It scrapes the download and upload speed text from the page.
3. **Part 2: Tweet the Complaint**
    - The bot checks if the scraped speeds are below the promised speeds.
    - If they are, it opens a new tab and navigates to Twitter.
    - **Login:** It finds the username/email field, types the email, clicks next, finds the password field, types the password, and clicks the login button. *Note: Twitter's login flow can change; you may need to handle intermediate steps like entering a username.*
    - **Compose Tweet:** It waits for the tweet input box to be visible, then types out a formatted complaint message including the actual download/upload speeds.
    - It clicks the "Tweet" button to post.

Example Code Snippet (Waiting and Clicking):

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time

# --- Inside a class method ---
# Wait for the "GO" button to be clickable for a maximum of 60 seconds
go_button = WebDriverWait(self.driver, 60).until(
    EC.element_to_be_clickable((By.CSS_SELECTOR, ".start-button a"))
)
go_button.click()

# Wait for download speed element to be present
time.sleep(60) # A simple but less robust wait
down_speed_element = self.driver.find_element(By.CSS_SELECTOR, ".download-speed")
self.down = float(down_speed_element.text)
```

Day 52: Instagram Follower Bot 🤖

Objective: Create a bot that logs into Instagram, finds a target account, and follows all of that account's followers.

Key Concepts:

- **Handling Pop-ups:** Websites often have pop-ups (e.g., "Save Login Info?", "Turn on Notifications"). The script must identify and dismiss them to continue.
- **Scrolling Inside an Element:** To load all followers, you need to scroll down within the followers list pop-up, not the main page. This requires JavaScript execution.
- **Exception Handling:** The script should be robust. Use `try-except` blocks to handle cases where an element isn't found or an unexpected pop-up appears.

Project Workflow:

1. **Login to Instagram:**
   - Navigate to Instagram's login page.
   - Use `time.sleep()` initially to give the page time to load (later, explicit waits are better).
   - Find the username and password fields and `send_keys()` .
   - Click the login button.
   - Handle the "Save Info" and "Turn on Notifications" pop-ups by finding the "Not Now" buttons and clicking them.
2. **Find Target Account's Followers:**
   - Navigate to the target account's profile (e.g., `https://www.instagram.com/target_username/`).
   - Click on the "followers" link to open the followers list modal.
3. **Scrape and Follow:**
   - The followers list is a scrollable pop-up.
   - To scroll it, you need to get the modal element and then execute a JavaScript command.
   - **JavaScript Execution:**
     `driver.execute_script("arguments[0].scrollTop = arguments[0].scrollHeight", modal_element)`
   - The bot enters a loop:
     - Find all the "Follow" buttons inside the modal.
     - Iterate through the buttons and click each one.
     - Use a `try-except` block because some buttons might change to "Following" or be blocked, causing errors.
     - After a cycle of following, scroll the modal down to load more followers.
     - Repeat until the desired number of users have been followed or the bottom of the list is reached.

Example Code Snippet (Scrolling a Modal):

```python
# Assuming 'modal' is the WebElement for the follower list pop-up
# Find the modal first, e.g., by its XPATH
modal = self.driver.find_element(By.XPATH, '/html/body/div[6]/div[1]/div/div[2]/div/div/div/div/d

for i in range(10): # Scroll 10 times
    # In the browser, open dev tools and run the JS command to check it
    self.driver.execute_script("arguments[0].scrollTop = arguments[0].scrollHeight", modal)
    time.sleep(2) # Wait for new followers to load
```

Day 53: Data Entry Job Automation (Zillow Web Scraping) 🏡

Objective: Scrape rental property data (address, price, link) from Zillow and automatically fill out a Google Form with this information.

Key Concepts:

- **Combining Scraping Libraries:** Use **BeautifulSoup** for parsing static HTML content and **Selenium** for automating browser actions (data entry).
- **HTTP Headers:** Websites like Zillow may block simple `requests` calls. To appear like a real browser, you must provide HTTP headers, especially `User-Agent` and `Accept-Language` .
- **Web Scraping (Part 1 - BeautifulSoup):**
    i. Use the `requests` library to get the HTML of the Zillow search results page. Provide the necessary headers.
    ii. Create a BeautifulSoup `soup` object from the response text.
    iii. Inspect the page's HTML to find the selectors for the listing card, price, address, and link for each property.
    iv. Use `soup.select()` or `soup.find_all()` to extract all listings.
    v. Loop through the results, cleaning the data (e.g., removing text like "/mo" from the price) and storing it in a list of dictionaries.
- **Data Entry (Part 2 - Selenium):**
    i. Create a Google Form with fields for Address, Price, and Link.
    ii. Use Selenium to open the live Google Form link.
    iii. Loop through the list of scraped property data.
    iv. For each property:
        - Find the input fields in the form using their XPATH or other selectors.
        - Use `send_keys()` to fill in the address, price, and link.
        - Click the "Submit" button.
        - To submit another response, you'll need to click the "Submit another response" link.

Example Code Snippet (Scraping with BeautifulSoup):

```python
import requests
from bs4 import BeautifulSoup

ZILLOW_URL = "YOUR_ZILLOW_SEARCH_URL"
HEADERS = {
    "User-Agent": "Your User Agent String",
    "Accept-Language": "en-US,en;q=0.9"
}

response = requests.get(ZILLOW_URL, headers=HEADERS)
soup = BeautifulSoup(response.text, "html.parser")

# CSS selector might change, always inspect the page first!
all_link_elements = soup.select(".StyledPropertyCardDataWrapper a")
all_address_elements = soup.select(".StyledPropertyCardDataWrapper address")
all_price_elements = soup.select(".PropertyCardWrapper__StyledPriceLine")

# Process and store the data...
```

Day 54: Introduction to Web Development with Flask 🌐

Objective: Understand the basics of the Flask framework by creating a simple web server.

Key Concepts:

- **Web Framework:** A collection of libraries and modules that helps developers build web applications without having to handle low-level details like protocols, sockets, or thread management. Flask is a *micro-framework* because it's lightweight and provides only the essentials.
- **Flask Application:** The core of a Flask site is a `Flask` object instance.

  ```python
  from flask import Flask
  app = Flask(__name__)
  ```

- **Routing:** The process of mapping URLs to Python functions. This is done with the `@app.route()` decorator. The function decorated by the route is called a **view function**.

```python
@app.route('/') # This is the route for the homepage
def hello_world():
    return 'Hello, World!'
```

- **Running the Server:**
  - The development server is run from the command line or within an `if __name__ == "__main__":` block.
  - Setting `debug=True` enables the debugger and automatically reloads the server when you make code changes.

Full Basic "Hello, World!" App:

```python
from flask import Flask

# 1. Create a Flask app instance
app = Flask(__name__)

# 2. Define a route for the homepage URL ("/")
@app.route("/")
def hello_world():
    # 3. The function returns the content to be displayed in the browser
    return "<p>Hello, World!</p>"

# This block allows you to run the app directly from the Python script
if __name__ == "__main__":
    # debug=True enables auto-reloading and an interactive debugger in the browser
    app.run(debug=True)
```

Running from Terminal:

1. Set the environment variable: `export FLASK_APP=main.py` (on Mac/Linux) or `set FLASK_APP=main.py` (on Windows).
2. Run the server: `flask run`

Day 55: Higher Lower URL Game 🎲

Objective: Build a simple "guess the number" game in the browser using Flask, where the user guesses by changing the URL.

Key Concepts:

- **Python Decorators:** A function that takes another function as input, adds some functionality to it, and returns the modified function. `@app.route()` is a decorator. You can create your own to wrap multiple view functions with shared logic (e.g., making text bold).
- **Variable Rules / Dynamic URLs:** You can capture parts of a URL and pass them as arguments to your view function.

  ```
  @app.route("/username/<name>")
  def greet(name):
      return f"Hello there {name}!"
  ```

  You can also specify a converter, like `<int:number>`, to automatically convert the URL part to an integer.

Project Workflow:

1. **Generate a Random Number:** When the server starts, pick a random number between 0 and 9. Store it in a global variable.
2. **Create the Homepage (`/`):**
   - This page should display a heading like "Guess a number between 0 and 9" and show a GIF.
3. **Create the Guessing Route (`/<int:guess>`):**
   - This route captures the user's guess from the URL.
   - The view function compares the `guess` to the randomly generated number.
   - It returns different HTML based on the comparison:
     - If `guess < random_number`, return "Too low!" with a "too low" GIF.
     - If `guess > random_number`, return "Too high!" with a "too high" GIF.
     - If `guess == random_number`, return "You found me!" with a "correct" GIF.

Example Code Snippet:

```python
from flask import Flask
import random

app = Flask(__name__)

# Generate the number once when the app starts
random_number = random.randint(0, 9)
print(f"Pssst, the number is {random_number}")

@app.route('/')
def home():
    return '<h1>Guess a number between 0 and 9</h1>' \
           '<img src="https://media.giphy.com/media/3o7aCSPqXE5C6T8tBC/giphy.gif">'

@app.route('/<int:guess>')
def check_guess(guess):
    if guess < random_number:
        return '<h1 style="color: red;">Too low, try again!</h1>' \
               '<img src="https://media.giphy.com/media/jD4DwBtqPXRXa/giphy.gif">'
    elif guess > random_number:
        return '<h1 style="color: purple;">Too high, try again!</h1>' \
               '<img src="https://media.giphy.com/media/3o6ZtaO9BZHcOjmErm/giphy.gif">'
    else:
        return '<h1 style="color: green;">You found me!</h1>' \
               '<img src="https://media.giphy.com/media/4T7e4DmcrP9du/giphy.gif">'

if __name__ == "__main__":
    app.run(debug=True)
```

Day 56: Rendering HTML/Static Files and Using Website Templates 🎨

Objective: Learn how to serve professional, multi-file websites with Flask by separating HTML, CSS, and JavaScript from your Python code.

Key Concepts:

- **Project Structure:** Flask expects a specific folder structure to find your files:

```
/project_folder
    /static
        /css
            style.css
        /js
            script.js
        /images
            image.png
    /templates
        index.html
        about.html
    main.py
```

- **`render_template()`:** This Flask function looks for a file in the `templates` folder, renders it, and sends it to the browser. You no longer return HTML strings from your view functions.

  ```
  from flask import render_template

  @app.route('/')
  def home():
      return render_template('index.html')
  ```

- **`url_for()`:** This function generates a URL for a static file or another view function. **Crucially**, you should use `url_for()` to link to your CSS and JS files in your HTML. This prevents broken links if you change your URL structure.

Example Usage in `index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Website</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
</head>
<body>
    <h1>Hello from a Template!</h1>
</body>
</html>
```

The project for this day is to take a pre-built static HTML/CSS personal site and convert it into a running Flask application using this structure.

Day 57: Templating with Jinja in Flask Applications 🧩

Objective: Make websites dynamic by passing data from your Python backend to your HTML templates using the Jinja templating engine.

Key Concepts:

- **Jinja:** A templating language for Python. Flask uses it automatically when you call `render_template()` . Jinja allows you to embed code-like expressions directly into your HTML.
- **Expressions (`{{ ... }}`):** Used to print the value of a variable passed from the backend.

```
# In main.py
@app.route('/')
def home():
    current_year = 2025
    return render_template('index.html', year=current_year, name="Angela")
```

```
<footer>
    <p>Copyright {{ year }}. Built by {{ name }}.</p>
</footer>
```

- **Statements (`{% ... %}`):** Used for control flow, like `for` loops and `if` conditions.

```
{% for post in all_posts: %}
    <h2>{{ post.title }}</h2>
    <h3>{{ post.subtitle }}</h3>
{% endfor %}
```

- **Template Inheritance:** A powerful feature to avoid repeating code (like headers and footers).
  i. **Create a `base.html` :** This file contains the common HTML structure (`<html>` , `<head>` , `<body>` , header, footer).
  ii. **Define `blocks` :** In `base.html` , define placeholders using `{% block block_name %}{% endblock %}` .
  iii. **Extend the base:** In other templates (e.g., `index.html`), use `{% extends "base.html" %}` at the top.
  iv. **Fill the blocks:** Override the blocks with specific content for that page.

Project: Build a dynamic blog that fetches post data from an API (or a local Python list) and displays it on the homepage. Create a separate page (`post.html`) to display the full content of a single blog post when a user clicks on it.

Day 58: Bootstrap & Flask - Website Portfolio Project 💼

Objective: Integrate the Bootstrap CSS framework into a Flask project to quickly build a responsive, professionally styled website.

Key Concepts:

- **Bootstrap:** A popular front-end toolkit that provides pre-styled components (like navbars, buttons, cards, and grids) and a powerful responsive grid system. It saves you from writing a lot of CSS from scratch.
- **Integration Methods:**
    i. **CDN (Content Delivery Network):** The easiest way. Copy-paste the Bootstrap CSS `<link>` and JS `<script>` tags from the official Bootstrap website into your `base.html` template's `<head>` and `<body>` sections.
    ii. **Local Files:** Download the Bootstrap files and place them in your `static` folder. Then link to them using `url_for()` .
- **Bootstrap Grid System:** A mobile-first system for creating layouts. It's based on a 12-column grid. You use classes like `.container` , `.row` , and `.col-md-4` to arrange your content. `md` stands for medium-sized screens; it changes for different screen sizes (e.g., `sm` , `lg` , `xl`).

Project: Rebuild the personal/portfolio website from Day 56 using Bootstrap components.

- Replace the custom CSS navbar with Bootstrap's responsive `navbar` component.
- Use the Bootstrap grid to create a multi-column layout for projects or skills.
- Style buttons and forms with Bootstrap classes (`.btn` , `.btn-primary` , `.form-control`).

Day 59: Blog Capstone Project Part 2 - Adding Styling 💅

Objective: Apply the Bootstrap knowledge from Day 58 to the blog project from Day 57 to create a fully styled, multi-page blog website.

Project Workflow:

1. **Integrate Bootstrap:** Add the Bootstrap CDN links to your blog's `base.html` .
2. **Create a Consistent Layout:**
    - **Header:** Create a `header.html` partial template. Inside, use Bootstrap's `navbar` component to create a navigation bar with links to "Home" and "About" pages. Use `{% include "header.html" %}` in `base.html` .

    - **Footer:** Create a `footer.html` partial. Add social media links and a copyright notice. Include it in `base.html`.
3. **Style the Homepage (`index.html`):**
    - Wrap the content in a Bootstrap `.container`.
    - Display a clean "Jumbotron" or large header image.
    - List the blog post previews. Each preview should link to the full post page. Use Bootstrap classes to style the post titles, subtitles, and "Read More" links.
4. **Style the Post Page (`post.html`):**
    - Inherit the header and footer from `base.html`.
    - Display the full post title, subtitle, and body content within a `.container`.
5. **Create About/Contact Pages:** Create simple static pages (`about.html` , `contact.html`) that extend `base.html` to ensure they share the same styling and navigation. Create routes for them in `main.py`.

Day 60: Blog Capstone Project Part 3 - POST Requests & Making Forms 📬

Objective: Add a working contact form to the blog, allowing users to send messages that are processed by the Flask backend.

Key Concepts:

- **HTTP Methods:**
    - **`GET` :** Used to request data from a server (e.g., loading a webpage). This is the default method.
    - **`POST` :** Used to send data *to* a server to create or update a resource (e.g., submitting a form, logging in).
- **Handling `POST` in Flask:**
    - You must explicitly allow the `POST` method in your route decorator: `@app.route('/contact', methods=["GET", "POST"])`.
    - Check which method was used with `if request.method == "POST":`.
- **HTML `<form>` Element:**
    - `action` : The URL where the form data should be sent.
    - `method` : The HTTP method to use (e.g., `post`).
    - `<input>` elements must have a `name` attribute. This `name` becomes the key for accessing the data in Flask.
- **`request` Object:** Imported from Flask (`from flask import request`), this object contains all the information about the current request.

  - `request.form` : A dictionary-like object that holds the data from a submitted form. You can access values using keys that match the `name` attributes of your inputs (e.g., `request.form['username']`).

Project Workflow:

1. **Create the HTML Form:** In `contact.html` , build a form with inputs for name, email, and message. Set `action="/contact"` and `method="post"` .
2. **Update the Flask Route:**
   - Modify the `/contact` route to accept both `GET` and `POST` requests.
   - If the method is `GET` , just render the `contact.html` template as before.
   - If the method is `POST` :
     - Access the submitted data using `request.form.get("name")` , `request.form.get("email")` , etc.
     - (For this project) Print the received data to the console or implement logic to send an email with the data.
     - Change the heading on the page to "Successfully sent your message." to give the user feedback.

Example Code Snippet:

```python
from flask import Flask, render_template, request

app = Flask(__name__)

# ... other routes ...

@app.route("/contact", methods=["GET", "POST"])
def contact():
    if request.method == "POST":
        # Get data from the form
        data = request.form
        name = data["name"]
        email = data["email"]
        message = data["message"]
        print(f"New message from {name} ({email}): {message}")
        return render_template("contact.html", msg_sent=True)
    # This is for the GET request
    return render_template("contact.html", msg_sent=False)
```

```
{% if msg_sent %}
    <h1>Successfully sent your message.</h1>
{% else %}
    <h1>Contact Me</h1>
    <form action="{{ url_for('contact') }}" method="post">
        <input name="name" type="text" placeholder="Name" required>
        <input name="email" type="email" placeholder="Email" required>
        <textarea name="message" placeholder="Your Message" required></textarea>
        <button type="submit">Send</button>
    </form>
{% endif %}
```