Web Automation with Python

1. What Is Web Automation?

Web automation means using a program to control actions that a person would normally perform inside a web browser.

For example, a person might:

1. Open a website
2. Log in
3. Search for something
4. Click a button
5. Fill out a form
6. Download a file
7. Read information from the page
8. Take a screenshot

With web automation, Python can perform these steps automatically.

The basic idea is:

```
Human Action
     ↓
Browser
     ↓
Python Automation Script
     ↓
Navigate → Find → Interact → Read → Save
```

Web automation is useful when a task is:

- repetitive
- predictable
- time-consuming
- performed through a browser
- difficult to perform manually at scale

Modern browser automation tools such as Playwright can control Chromium, Firefox, and WebKit from Python. Selenium is another widely used browser automation framework.

---

2. Web Automation vs Web Scraping

These concepts are related but not identical.

Web scraping

The main goal is:

> Get information from a website.

Example:

```
Website
   ↓
Extract product names
   ↓
Save to CSV
```

Web automation

The goal is:

> Perform actions in a website.

Example:

```
Open website
   ↓
Login
   ↓
Search
   ↓
Click button
   ↓
Download report
```

A project can use both.

For example:

```
Automated browser
      ↓
Open dashboard
      ↓
Apply filters
      ↓
Read results
      ↓
Save results to CSV
```

That is both automation and data extraction.

---

3. Why Browser Automation Is Different From API Requests

You have already learned how Python can communicate directly with APIs.

An API-based program might do:

```
Python
   ↓
HTTP request
   ↓
API
   ↓
JSON response
```

Browser automation works differently:

```
Python
   ↓
Browser
   ↓
Website
   ↓
HTML + CSS + JavaScript
   ↓
Rendered interface
```

This distinction matters.

If a website provides a suitable API, using the API is often simpler and more reliable than controlling a browser.

Browser automation becomes useful when you need to interact with the actual web interface.

Examples:

- clicking buttons
- filling forms
- selecting dropdowns
- interacting with JavaScript applications
- downloading files through a UI
- testing a complete user workflow

---

4. Common Web Automation Tools

Python has several options.

Playwright

Playwright is a modern browser automation library supporting Chromium, Firefox, and WebKit. It provides both synchronous and asynchronous Python APIs.

Install it:

```
pip install playwright
playwright install
```

The second command installs the browser binaries used by Playwright.

Selenium

Selenium is another established browser automation framework.

Install:

```
pip install selenium
```

Modern Selenium versions include Selenium Manager, which can automatically handle much of the browser-driver setup.

Example:

```
from selenium import webdriver

driver = webdriver.Chrome()

driver.get("https://example.com")

print(driver.title)

driver.quit()
```

Which One Should You Learn?

For a modern Python web automation project, Playwright is a strong option because it provides:

- Chromium support
- Firefox support
- WebKit support
- sync and async APIs
- browser contexts
- locators
- auto-waiting

- screenshots
- network controls
- testing support

Selenium remains important, particularly when working with existing Selenium-based projects or teams.

For this guide, we will use **Playwright**.

---

5. Installing Playwright

First create a virtual environment.

```
python -m venv .venv
```

Activate it.

Windows:

```
.venv\Scripts\activate
```

macOS/Linux:

```
source .venv/bin/activate
```

Install Playwright:

```
pip install playwright
```

Then install the browsers:

```
playwright install
```

---

6. Your First Automated Browser

Create:

```
automation.py
```

Add:

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch()

    page = browser.new_page()
```

```
    page.goto("https://example.com")

    print(page.title())

    browser.close()
```

Run:

```
python automation.py
```

The program:

1. Starts Playwright
2. Launches Chromium
3. Creates a page
4. Opens the website
5. Reads its title
6. Closes the browser

Playwright's official Python documentation uses the same basic `launch → new_page → goto → close` workflow.

---

7. Headless vs Headed Mode

By default, Playwright runs browsers in headless mode.

That means:

```
Python
 ↓
Browser runs
 ↓
No visible browser window
```

You can make the browser visible:

```
browser = p.chromium.launch(headless=False)
```

This is useful while developing and debugging.

Example:

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
```

```
    page = browser.new_page()

    page.goto("https://example.com")

    print(page.title())

    browser.close()
```

Once the automation works, headless mode is often convenient for automated jobs and CI environments.

---

8. Understanding Browser, Context and Page

A useful mental model is:

```
Playwright
    ↓
Browser
    ↓
Browser Context
    ↓
Page
```

Browser

Represents the browser process.

```
browser = p.chromium.launch()
```

Browser Context

Represents an isolated browser session.

```
context = browser.new_context()
```

Page

Represents a browser tab.

```
page = context.new_page()
```

Example:

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch()
```

```
    context = browser.new_context()

    page = context.new_page()

    page.goto("https://example.com")

    print(page.title())

    browser.close()
```

A browser context is particularly useful when you need isolated sessions or multiple independent users. Playwright pages represent individual tabs or popup windows.

---

9. Navigating Websites

The basic navigation method is:

```
page.goto("https://example.com")
```

You can inspect the current URL:

```
print(page.url)
```

You can inspect the title:

```
print(page.title())
```

Example:

```
page.goto("https://example.com")

print("URL:", page.url)
print("Title:", page.title())
```

Playwright's navigation API handles navigation caused both by direct URL navigation and by page interactions.

---

10. Finding Elements

Automation becomes useful when the program can find elements on a page.

Suppose a page contains:

```
<input id="search">
<button id="submit">Search</button>
```

You can locate the input:

```
search_box = page.locator("#search")
```

Then fill it:

```
search_box.fill("Python")
```

And click the button:

```
page.locator("#submit").click()
```

The workflow becomes:

```
Locate
   ↓
Interact
   ↓
Observe result
```

---

11. Using Locators

A locator identifies an element on a page.

Example:

```
page.locator("#email")
```

Another:

```
page.locator(".submit-button")
```

You can also use text:

```
page.get_by_text("Sign in")
```

And semantic roles:

```
page.get_by_role("button", name="Submit")
```

Role-based locators are often easier to understand because they describe what the user sees rather than depending entirely on implementation details.

Example:

```
page.get_by_role("button", name="Search").click()
```

---

12. Filling Forms

Suppose a page contains:

```
<input id="username">
<input id="password">
<button>Login</button>
```

Automation:

```
page.locator("#username").fill("Hafsa")

page.locator("#password").fill("example-password")

page.get_by_role("button", name="Login").click()
```

The browser can perform the same interaction a user would perform manually.

Never hard-code real passwords into source code.

Use environment variables for secrets.

```
import os

username = os.environ["APP_USERNAME"]
password = os.environ["APP_PASSWORD"]
```

Then:

```
page.locator("#username").fill(username)
page.locator("#password").fill(password)
```

---

13. Clicking Elements

Basic:

```
page.locator("#submit").click()
```

Using a role:

```
page.get_by_role("button", name="Submit").click()
```

Using a link:

```
page.get_by_role("link", name="Resources").click()
```

Using text:

```
page.get_by_text("Continue").click()
```

The goal is not simply to make the script work once.

The goal is to select elements in a way that remains stable when the website changes.

---

14. Reading Text From a Page

Suppose:

```
<h1>Python Resources</h1>
```

You can read it:

```
heading = page.locator("h1").inner_text()

print(heading)
```

For multiple elements:

```
items = page.locator(".resource").all_inner_texts()

for item in items:
    print(item)
```

This is where browser automation can also become a data collection tool.

---

15. Checking Whether an Element Exists

You may need to make decisions based on what appears on the page.

Example:

```
if page.get_by_text("Success").is_visible():
    print("Operation completed")
```

You can also inspect attributes:

```
link = page.locator("a").first

print(link.get_attribute("href"))
```

Automation should generally observe the page before deciding what to do next.

---

16. Waiting for the Web Page

Modern websites are dynamic.

A page might load:

```
HTML
 ↓
JavaScript
 ↓
API request
 ↓
Data arrives
 ↓
UI updates
```

Therefore, immediately assuming that everything exists can make automation unreliable. Playwright provides auto-waiting for many actions and assertions. Its documentation specifically recommends relying on Playwright's waiting behavior rather than unnecessarily inserting fixed delays.

Instead of:

```
import time

time.sleep(5)
```

prefer waiting for the actual condition.

For example:

```
page.get_by_role("button", name="Submit").click()

page.get_by_text("Success").wait_for()
```

The principle is:

> Wait for a condition, not an arbitrary amount of time.

---

17. Screenshots

Screenshots are useful for debugging and monitoring.

```
page.screenshot(path="homepage.png")
```

Full-page screenshot:

```
page.screenshot(
    path="homepage.png",
    full_page=True
)
```

A screenshot can help answer:

```
Did the page load?
Did the button appear?
Did the layout change?
Did the automation reach the expected screen?
```

---

18. Automating a Search Workflow

Consider a simple workflow:

```
Open website
    ↓
Find search box
    ↓
Enter query
    ↓
Click search
    ↓
Read results
```

Example structure:

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch()

    page = browser.new_page()

    page.goto("https://example.com")

    page.locator("#search").fill("Python")

    page.get_by_role("button", name="Search").click()

    results = page.locator(".result").all_inner_texts()

    for result in results:
        print(result)

    browser.close()
```

The selectors will depend on the actual website.

---

19. Handling Multiple Pages

A workflow may open a new tab.

For example:

```
Page A
   ↓
Click link
   ↓
Page B opens
```

Playwright represents browser tabs as `Page` objects.

You can work with multiple pages through a browser context.

```
context = browser.new_context()

page1 = context.new_page()
page2 = context.new_page()
```

Then:

```
page1.goto("https://example.com")
page2.goto("https://playwright.dev")
```

---

20. Handling Downloads

Browser automation can also automate downloads.

A download workflow typically looks like:

```
Click download
      ↓
Browser creates download
      ↓
Python receives download
      ↓
Save file
```

Conceptually:

```
with page.expect_download() as download_info:
    page.get_by_text("Download").click()

download = download_info.value
```

```
download.save_as("report.pdf")
```

This is useful for:

- reports
- CSV files
- PDFs
- generated exports
- invoices
- backups

---

21. Browser Automation With File Processing

Web automation becomes more powerful when combined with other Python skills.

Example:

```
Website
   ↓
Automated login
   ↓
Download CSV
   ↓
Python reads CSV
   ↓
Process data
   ↓
Save report
```

This combines:

- browser automation
- file handling
- CSV processing
- data processing
- logging
- error handling

Python is particularly useful here because automation does not have to stop at the browser.

---

22. A Real Automation Pipeline

Imagine a company dashboard where an employee manually downloads a daily report.

Manual process:

```
Open browser
 ↓
Login
 ↓
Open dashboard
 ↓
Select date
 ↓
Click export
 ↓
Download CSV
 ↓
Rename file
 ↓
Move file into folder
```

Automation:

```
Python Script
      ↓
Launch Browser
      ↓
Open Dashboard
      ↓
Authenticate
      ↓
Select Date
      ↓
Export Report
      ↓
Save CSV
      ↓
Organize File
      ↓
Log Result
```

This is a complete automation pipeline.

---

23. Building Reliable Automation

A beginner often writes:

```
open website
click button
sleep
click another button
sleep
```

That may work temporarily.

A production-quality automation should instead think in terms of states.

```
Expected State
      ↓
Perform Action
      ↓
Verify Result
      ↓
Continue
```

For example:

```
Login page visible
      ↓
Enter credentials
      ↓
Click Login
      ↓
Dashboard visible?
      ↓
Yes → Continue
No  → Handle failure
```

This makes the automation easier to debug.

---

24. Error Handling

Browser automation can fail because:

- website is unavailable
- selector changed
- login failed
- network is slow
- page structure changed
- expected element does not appear

- download fails
- browser crashes

Use exception handling.

```
from playwright.sync_api import sync_playwright

try:
    with sync_playwright() as p:
        browser = p.chromium.launch()

        page = browser.new_page()

        page.goto("https://example.com")

        print(page.title())

        browser.close()

except Exception as error:
    print("Automation failed:", error)
```

For larger applications, catch specific exceptions where practical rather than using one giant `except Exception` around everything.

---

25. Logging Automation

Automation should produce useful information about what happened.

```
import logging

logging.basicConfig(level=logging.INFO)

logging.info("Starting browser")
logging.info("Opening dashboard")
logging.info("Downloading report")
```

A useful log might look like:

```
INFO Starting automation
INFO Opening dashboard
INFO Login successful
INFO Selecting date
INFO Download started
```

```
INFO Report saved
INFO Automation completed
```

Logs become extremely valuable when the script runs without supervision.

---

26. Screenshots on Failure

One useful debugging strategy is:

```
Automation fails
      ↓
Capture screenshot
      ↓
Save error information
      ↓
Investigate
```

For example:

```
try:
    page.get_by_role("button", name="Export").click()

except Exception:
    page.screenshot(path="automation_error.png")
    raise
```

This gives you evidence of what the browser looked like when the failure occurred.

---

27. Avoiding Fragile Selectors

A fragile selector might depend on generated CSS:

```
page.locator(
    "div.container > div:nth-child(2) > button:nth-child(4)"
)
```

If the developer changes the layout, the automation can break.

Prefer meaningful selectors.

For example:

```
page.get_by_role("button", name="Export").click()
```

Or a stable identifier:

```
page.locator("#export-report").click()
```

Good automation depends on stable interaction points.

---

28. Browser Automation Is Not Magic

A website can change.

Suppose your automation expects:

```
"Download Report"
```

The website changes it to:

```
"Export Report"
```

Your script may fail.

Therefore:

```
Automation
    ≠
Permanent compatibility
```

You need to maintain automation just like normal software.

---

29. Authentication

Some websites require authentication.

Common approaches include:

- username/password
- session cookies
- authentication tokens
- OAuth
- organization-specific login systems

Do not store credentials directly in source code.

Bad:

```
password = "MyRealPassword123"
```

Better:

```
import os

password = os.environ["APP_PASSWORD"]
```

For automation involving sensitive accounts, use the website's supported authentication mechanism and respect its security controls.

---

30. Browser Contexts and Sessions

A browser context can represent an isolated session.

This is useful for scenarios such as:

```
Context A → User A
Context B → User B
```

Each context can maintain separate browser state.

This makes contexts useful for testing and multi-session automation.

---

31. Async Web Automation

Playwright supports both synchronous and asynchronous Python APIs.

Synchronous:

```
from playwright.sync_api import sync_playwright
```

Asynchronous:

```
from playwright.async_api import async_playwright
```

Example:

```
import asyncio

from playwright.async_api import async_playwright


async def main():
    async with async_playwright() as p:
        browser = await p.chromium.launch()

        page = await browser.new_page()

        await page.goto("https://example.com")

        print(await page.title())

        await browser.close()
```

```
asyncio.run(main())
```

Async automation can be useful when a larger application already uses `asyncio`.

Do not choose async simply because it looks more advanced.

Use it when the application's architecture benefits from asynchronous execution.

---

32. Browser Automation vs `requests`

You have already learned HTTP requests.

A useful decision tree is:

```
Do I need to interact with the rendered browser UI?
              ↓
        Yes          No
         ↓            ↓
    Playwright      HTTP client
                    requests/
                    urllib
```

For example:

API available

```
response = requests.get(
    "https://example.com/api/resources"
)
```

Prefer direct HTTP/API communication when appropriate.

Browser interaction required

```
page.goto("https://example.com")
page.get_by_role("button", name="Export").click()
```

Use browser automation.

Python's standard `urllib` package provides URL-handling and request functionality, while higher-level HTTP clients can make HTTP work more convenient.

---

33. Web Automation vs Testing

Browser automation and browser testing use many of the same techniques.

Automation:

```
Open website
 ↓
Perform task
 ↓
Download report
```

Testing:

```
Open website
 ↓
Perform user action
 ↓
Verify expected result
```

Testing adds assertions.

For example:

```
from playwright.sync_api import expect

expect(page).to_have_title("Python Resources")
```

Playwright is designed for end-to-end testing as well as general browser automation.

---

34. A Simple Automated Test

Example:

```
from playwright.sync_api import sync_playwright, expect

with sync_playwright() as p:
    browser = p.chromium.launch()

    page = browser.new_page()

    page.goto("https://example.com")

    expect(page).to_have_title("Example Domain")

    browser.close()
```

The important difference is:

```
Automation:
Perform action
```

```
Testing:
Perform action + verify expected result
```

---

35. Web Automation for haas.dev

Imagine haas.dev has a resource page.

A hypothetical automation workflow could be:

```
Open resources page
       ↓
Collect resource cards
       ↓
Read title
       ↓
Read category
       ↓
Read PDF link
       ↓
Check links
       ↓
Save results
```

Example structure:

```
from playwright.sync_api import sync_playwright

URL = "https://dev-roast-app.vercel.app/resources"

with sync_playwright() as p:
    browser = p.chromium.launch()

    page = browser.new_page()

    page.goto(URL)

    resources = page.locator(".resource-card")

    for resource in resources.all():
        title = resource.locator("h3").inner_text()
        print(title)

    browser.close()
```

The exact selectors depend on the website's actual HTML.

The important concept is the automation flow:

```
Page
 ↓
Resource collection
 ↓
Individual resource
 ↓
Extract fields
 ↓
Validate/store
```

---

36. A Better haas.dev Automation Project

Build a **Resource Link Checker**.

Goal:

> Automatically visit every resource link and report broken links.

Architecture:

```
resources page
      ↓
extract links
      ↓
visit each link
      ↓
check result
      ↓
classify
      ↓
save report
```

Possible output:

```
Resource                        Status
------------------------------------------------
Python Basics                   OK
OOP in Python                   OK
FastAPI Guide                   OK
Testing Guide                   BROKEN
Git Guide                       OK
```

This is a practical automation project because it combines browser navigation, extraction, validation, error handling, and reporting.

---

37. Automation Safety

Automation can perform actions extremely quickly.

That makes safety important.

Before automating an action, ask:

```
What does this action change?
Can it delete something?
Can it submit something?
Can it send something?
Can it purchase something?
Can it affect another user?
Can it trigger an irreversible operation?
```

Prefer safe workflows.

For example:

```
Read-only automation
       ↓
Test environment
       ↓
Dry run
       ↓
Human verification
       ↓
Production automation
```

Never use automation to bypass access controls, CAPTCHAs, rate limits, authentication protections, or other security mechanisms.

Respect the website's terms, access rules, and applicable policies.

---

38. Rate Limiting and Responsible Automation

A script can send actions much faster than a human.

For example:

```
Human:
1 request every few seconds
```

```
Script:
100 requests immediately
```

This can overload a service.

Responsible automation should:

- use reasonable request rates
- avoid unnecessary actions
- cache information when appropriate
- use APIs when available
- respect applicable service policies
- avoid aggressive concurrency
- stop when the service signals excessive traffic

Automation should reduce repetitive work, not create unnecessary load.

---

39. Common Web Automation Mistakes

Mistake 1: Using fixed sleeps everywhere

```
time.sleep(10)
```

Problem:

The page might finish in one second or need longer than ten seconds.

Better:

```
Wait for the actual condition.
```

Mistake 2: Hard-coding passwords

Bad:

```
password = "secret"
```

Better:

```
password = os.environ["APP_PASSWORD"]
```

Mistake 3: Using fragile selectors

Bad:

```
div:nth-child(7)
```

Better:

```
page.get_by_role("button", name="Submit")
```

Mistake 4: No verification

Bad:

```
page.get_by_role("button", name="Login").click()

# Assume login worked
```

Better:

```
Click Login
 ↓
Check dashboard
 ↓
Continue
```

Mistake 5: No error evidence

When automation fails, you should know:

- which step failed
- which page was open
- what URL was active
- what error occurred
- optionally what the page looked like

Mistake 6: Automating the browser when an API exists

If an official API can perform the required operation directly, browser automation may add unnecessary complexity.

---

40. A Production-Oriented Project Structure

A small project might start as:

```
web_automation/
│
├── automation.py
├── config.py
├── logger.py
├── requirements.txt
└── README.md
```

A larger project could become:

```
web_automation/
│
├── app/
│   ├── browser.py
│   ├── pages/
│   │   ├── login_page.py
│   │   └── dashboard_page.py
│   ├── services/
│   │   └── report_service.py
│   └── config.py
│
├── tests/
│   └── test_dashboard.py
│
├── screenshots/
│
├── reports/
│
├── requirements.txt
└── README.md
```

The exact structure depends on project size.

Do not create ten modules for a twenty-line script.

Start simple and introduce structure when complexity appears.

---

41. Page Object Model

For larger browser automation projects, you may separate page-specific behavior into classes.

Example:

```
class LoginPage:

    def __init__(self, page):
        self.page = page

    def login(self, username, password):
        self.page.locator("#username").fill(username)
        self.page.locator("#password").fill(password)
        self.page.get_by_role(
            "button",
```

```
            name="Login"
        ).click()
```

Then:

```
login_page = LoginPage(page)

login_page.login(username, password)
```

The advantage is that selectors and page-specific behavior are kept together.

If the login form changes, you have a clear place to update.

---

42. Think in Workflows

Instead of thinking:

> I need to click this button.

Think:

> What business task am I automating?

For example:

```
Business Task:
Download daily sales report
```

Break it down:

```
1. Open dashboard
2. Authenticate
3. Navigate to reports
4. Select today's date
5. Export CSV
6. Save file
7. Validate file
8. Log completion
```

This is the same thinking framework used in good software design:

```
Understand
   ↓
Break
   ↓
Design
   ↓
```

```
Build
   ↓
Verify
```

---

43. Mini Project: Automated Website Reporter

Build a Python program that:

1. Opens a website
2. Reads its title
3. Records the current URL
4. Takes a screenshot
5. Saves the page text
6. Logs the operation

Example:

```
from pathlib import Path
from playwright.sync_api import sync_playwright

OUTPUT = Path("output")
OUTPUT.mkdir(exist_ok=True)

with sync_playwright() as p:
    browser = p.chromium.launch()

    page = browser.new_page()

    page.goto("https://example.com")

    print("Title:", page.title())
    print("URL:", page.url)

    page.screenshot(
        path=str(OUTPUT / "page.png"),
        full_page=True
    )

    text = page.locator("body").inner_text()

    (OUTPUT / "page.txt").write_text(
        text,
        encoding="utf-8"
```

```
    )

    browser.close()
```

You have now combined:

- browser automation
- navigation
- DOM interaction
- screenshots
- file handling

---

44. 5 Minute Challenge

Create a script that opens a website and:

```
1. Opens the homepage
2. Prints the title
3. Prints the URL
4. Takes a screenshot
5. Saves the visible text
```

Then add:

```
6. Detect whether a specific heading exists
```

Finally:

```
7. Print SUCCESS or FAILED
```

---

45. Exercises

Exercise 1

Open a website and print:

```
Title
URL
```

Exercise 2

Find a heading and print its text.

Exercise 3

Find a button and click it.

Exercise 4

Fill a search input.

Exercise 5

Extract all links from a page.

Exercise 6

Take a full-page screenshot.

Exercise 7

Create a script that reports whether a specific element exists.

Exercise 8

Create a browser automation script that downloads a publicly available file.

Exercise 9

Create a resource checker for a website.

Exercise 10

Convert your script into a reusable class.

---

46. Mini Quiz

1. What is web automation?

A. Designing websites

B. Automatically controlling browser-based tasks

C. Creating databases

D. Compiling Python

Answer: B

2. What does `page.goto()` do?

A. Creates a browser

B. Opens/navigates to a URL

C. Saves a file

D. Closes a browser

Answer: B

3. Why are fixed sleeps often unreliable?

A. They always crash Python

B. They wait for a fixed amount of time rather than the required condition

C. They delete browser data

D. They disable JavaScript

Answer: B

4. What should you use instead of hard-coding passwords?

A. Comments

B. Environment variables or another secure secret mechanism

C. Global variables

D. `print()`

Answer: B

5. When might an API be preferable to browser automation?

A. When you need to click a visual button

B. When a suitable API directly provides the required operation

C. When a page requires rendering

D. When you need screenshots

Answer: B

---

47. Interview Questions

Beginner

1. What is web automation?
2. What is the difference between web automation and web scraping?
3. What is Playwright?
4. What is Selenium?
5. What does `page.goto()` do?
6. What is a locator?
7. What is headless browser automation?
8. Why do automated browsers need waiting?

Intermediate

9. What is the difference between a browser, browser context, and page?
10. Why are stable selectors important?
11. Why should fixed sleeps be avoided?
12. How would you handle a failed automation step?
13. How would you store credentials securely?
14. When would you choose an API instead of browser automation?
15. What is the Page Object Model?
16. How would you debug a failing browser automation script?

17. How can screenshots help with automation debugging?
18. What makes a browser automation script reliable?

Advanced

19. How would you design automation for multiple independent sessions?
20. How would you handle dynamic pages?
21. How would you make browser automation maintainable?
22. How would you integrate browser automation into CI?
23. How would you control concurrency responsibly?
24. How would you design retry behavior?
25. How would you detect when a website's UI has changed?

---

48. Web Automation Cheat Sheet

Install

```
pip install playwright
playwright install
```

Launch browser

```
browser = p.chromium.launch()
```

Visible browser

```
browser = p.chromium.launch(headless=False)
```

New page

```
page = browser.new_page()
```

Navigate

```
page.goto("https://example.com")
```

Get title

```
page.title()
```

Get URL

```
page.url
```

Locate element

```
page.locator("#id")
```

Fill input

```
page.locator("#email").fill("user@example.com")
```

Click

```
page.get_by_role("button", name="Submit").click()
```

Read text

```
page.locator("h1").inner_text()
```

Screenshot

```
page.screenshot(path="page.png")
```

Close browser

```
browser.close()
```

---

49. The Web Automation Mental Model

Remember this:

```
UNDERSTAND
What repetitive browser task am I solving?
        ↓
BREAK
What actions make up the workflow?
        ↓
DESIGN
What pages, elements, states and outcomes matter?
        ↓
BUILD
Automate one reliable step at a time.
        ↓
VERIFY
Confirm every important action produced the expected result.
        ↓
```

```
MONITOR
Log failures and maintain the automation as the website changes.
```

The goal is not:

> Make Python click buttons.

The goal is:

> Turn a repeatable browser workflow into reliable software.

---

50. Key Takeaways

- Web automation allows Python to control browser-based workflows.
- Playwright and Selenium are major browser automation options.
- Playwright supports Chromium, Firefox, and WebKit.
- A Playwright workflow commonly involves a browser, context, and page.
- `page.goto()` navigates to a URL.
- Locators identify elements that automation needs to interact with.
- Forms can be filled automatically.
- Buttons and links can be clicked programmatically.
- Modern web applications require condition-based waiting.
- Screenshots are useful for debugging.
- Credentials should never be hard-coded.
- Stable selectors make automation more maintainable.
- APIs should be preferred when they provide the required operation directly.
- Browser automation and browser testing use many of the same techniques.
- Reliable automation verifies results instead of blindly executing actions.
- Logs and error handling are essential for unattended automation.
- Responsible automation should respect access rules, service policies, and reasonable request rates.
- Good web automation is software engineering, not simply a sequence of clicks.

Website:

https://dev-roast-app.vercel.app

More resources:

https://dev-roast-app.vercel.app/resources

Visit haas.dev for more resources and guides.