**🔥 Hardest Parts to Understand in Playwright**

## here’s the **5 hardest concepts** in Playwright(especially for those coming from Selenium or BS4):

| **#** | **Concept** | **Why It’s Hard** |
| --- | --- | --- |
| 1️⃣ | **Async/Await Syntax** | Everything is async by default → confusing if you’ve only used sync Python (like in Selenium) |
| 2️⃣ | **Page Lifecycle & Waits** | Knowing *when* a page is fully loaded is not obvious |
| 3️⃣ | **Selectors & Locators** | It’s stricter than BS4/Selenium — and powerful but verbose |
| 4️⃣ | **Browser Contexts** | Many devs don’t understand how contexts help isolate sessions |
| 5️⃣ | **Stealth & Anti-Bot Avoidance** | Most beginners skip proper detection-avoidance techniques |

## 🧠 Let’s Pick the Hardest for Most Beginners:

# 1️⃣ Async/Await — Explained Simply and Broadly

**🧩 What’s Async?**

In simple terms:

Instead of waiting for one task to finish before moving to the next, Python can do other work **while waiting**, using async def and await.

**🐍 Normal vs Async Example**

**Regular (Synchronous) Python:**

**import time**

**def run():**

**print("Start")**

**time.sleep(5) # This blocks the thread for 5 seconds**

**print("Done")**

**run()**

🟩 2. Asynchronous version (using await asyncio.sleep(5))

import asyncio

async def run():

print("Start")

await asyncio.sleep(5) # This suspends only this coroutine

print("Done")

asyncio.run(run())

| **Feature** | **time.sleep()** | **await asyncio.sleep()** |
| --- | --- | --- |
| Type | Synchronous (blocking) | Asynchronous (non-blocking) |
| Blocks whole thread? | ✅ Yes | ❌ No |
| Good for I/O tasks? | ❌ No | ✅ Yes |
| Needs event loop? | ❌ No | ✅ Yes (asyncio.run) |
| Syntax complexity | Simple | More advanced (async/await) |

**✅ Asynchronous (asyncio.sleep) – *Concurrent / Multitasking-Like***

* The program can **pause one task** without blocking others.
* await asyncio.sleep(5) suspends only that coroutine.
* During that wait, **other async tasks can continue running**.
* **Use when:**
  + You're handling multiple I/O-bound tasks (network, scraping, API calls).
  + You want performance by **not blocking the system** during waits.
* 🚀 **Example** with concurrency:

**import asyncio**

**async def task(n):**

**print(f"Task {n} start")**

**await asyncio.sleep(5)**

**print(f"Task {n} done")**

**async def main():**

**await asyncio.gather(task(1), task(2), task(3)) # All run "at the same time"**

**asyncio.run(main())**

**🔸 Output:**

**Task 1 start**

**Task 2 start**

**Task 3 start**

**(5 seconds later)**

**Task 1 done**

**Task 2 done**

**Task 3 done**

**✅ What is await?**

**await is used in asynchronous functions to:**

* **Pause execution of the current coroutine until another asynchronous operation finishes.**
* **But — while it's paused — other coroutines can continue running in the background.**

**✅ Summary**

Use **time.sleep()**:

* For simple scripts or CPU-bound code where concurrency isn't needed.

Use **asyncio.sleep()**:

* For building **asynchronous programs** (e.g., scrapers, bots, servers) where you need to **wait without blocking**.

## **✅ Synchronous Web Scraping (One by One – Slow)**

**import requests**

**import time**

**urls = [**

**"https://example.com/page1",**

**"https://example.com/page2",**

**"https://example.com/page3"**

**]**

**def fetch(url):**

**print(f"Fetching {url}")**

**response = requests.get(url)**

**print(f"Done: {url}")**

**return response.text**

**start = time.time()**

**for url in urls:**

**fetch(url)**

**print("Total time:", time.time() - start)**

**❌ Problem:**

* **Each request waits for the last one to finish.**
* **If each takes ~2 seconds, total = 6 seconds for 3 URLs.**
* **Total time: 4.1349101066589355**

**✅ Async Web Scraping (All at Once – Fast 🚀)**

**We'll use:**

* **aiohttp: async version of requests**
* **asyncio: to run multiple fetches concurrently**

### 🛠️ Install first (if not installed):

**pip install aiohttp**

import asyncio

import aiohttp

import time

urls = [

"https://example.com/page1",

"https://example.com/page2",

"https://example.com/page3"

]

async def fetch(session, url):

print(f"Fetching {url}")

async with session.get(url) as response:

text = await response.text()

print(f"Done: {url}")

return text

async def main():

async with aiohttp.ClientSession() as session:

tasks = [fetch(session, url) for url in urls]

results = await asyncio.gather(\*tasks)

# results now contains the HTML from all URLs

start = time.time()

asyncio.run(main())

print("Total time:", time.time() - start)

| **Thing** | **What it does** |
| --- | --- |
| **async def fetch()** | **Defines an async function to fetch data** |
| **await session.get(url)** | **Makes a non-blocking HTTP call** |
| **asyncio.gather(\*tasks)** | **Runs all fetch tasks concurrently** |
| **asyncio.run(main())** | **Starts the event loop and runs async code** |

**🚀 Result:**

* **Instead of 6 seconds (3×2s), total time ≈ 2 seconds**
* **All pages fetched concurrently, thanks to await**

### QUESTIONS:

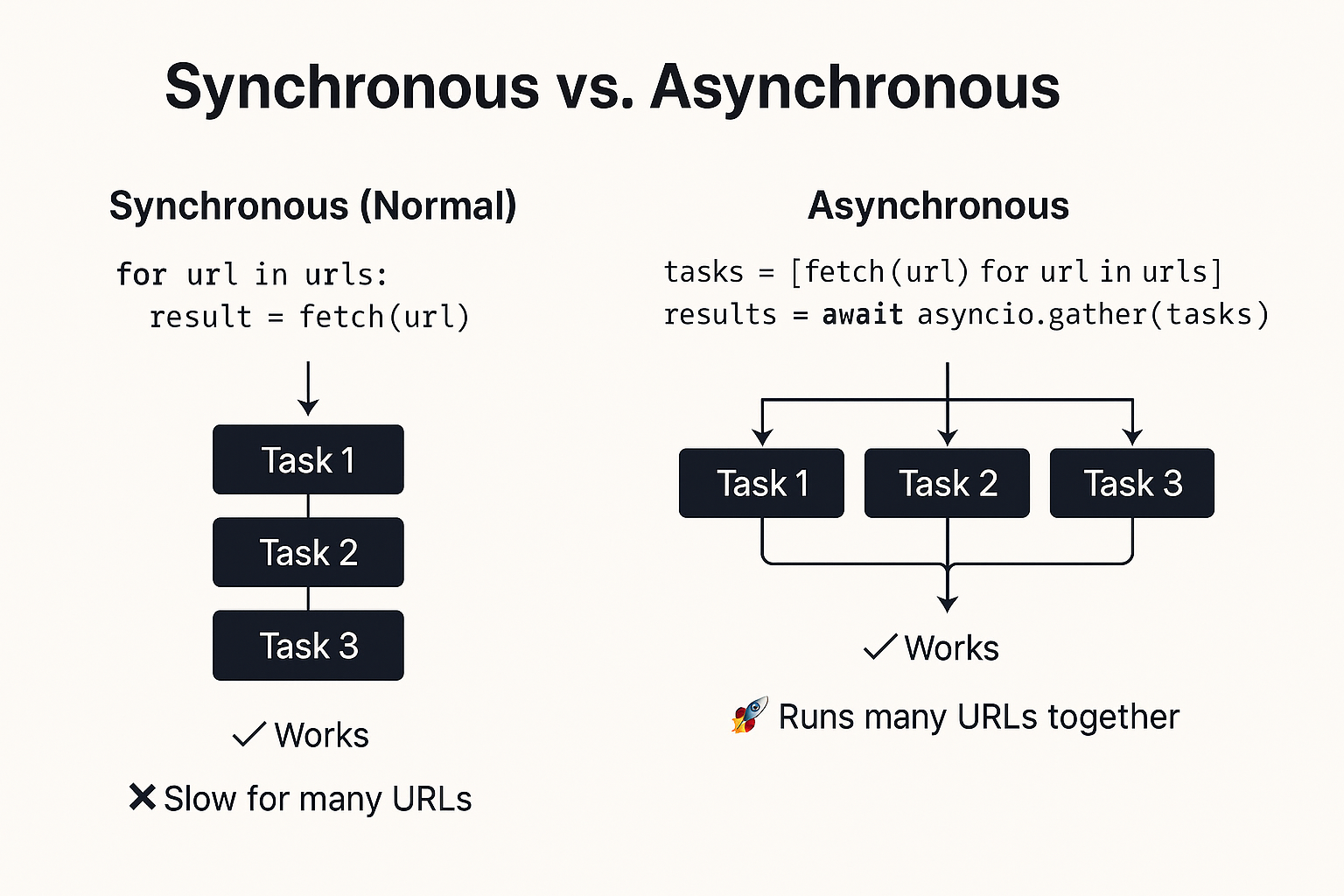
**✅ Part 1: Why async def in front of functions?**

async def fetch(url):

**🔹 Why use async?**

* Because inside the function, you want to use **await**.
* Python **does not allow await inside regular def functions**.
* Only async def functions can:
  + await other async operations
  + Be paused and resumed
  + Be run concurrently using asyncio

| **Concept** | | | **Meaning** |
| --- | --- | --- | --- |
| \*tasks | | | Unpacks the list, like passing one-by-one |
| await asyncio.gather(\*tasks) | | | Runs all tasks at the same time and waits for all to finish |
| await | | | Tells Python: "Pause here until this thing is done" |
| async | | | Makes a function that can use await |
|  |  |  |
| async def fetch() | | | To define an async function that can use await |
| tasks = [fetch(...)] | | | Creates coroutine objects |
| asyncio.gather(\*tasks) | | | Schedules them to run concurrently |
| await asyncio.gather(\*tasks) | | | Waits for them to finish and gets results |
| async with ... | | | Enters and exits context asynchronously (like aiohttp session) |



**Difference Between Sync and Async in Python**

This table compares synchronous and asynchronous behavior in Python. Synchronous code runs tasks one-by-one, while asynchronous code uses coroutine objects and can run tasks concurrently using `await asyncio.gather(\*tasks)`.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | What It Stores | How It Runs | Speed |
| Synchronous | Direct function results | One by one (waits each) | Slower for many tasks |
| Asynchronous | Coroutine objects (not running) | All together with await | Faster (concurrent run) |

**Notes:**

- In async: tasks = [fetch(url) for url in urls] → creates coroutine objects (not running yet)  
- Then await asyncio.gather(\*tasks) → runs all tasks concurrently and waits for all to finish  
- In sync: for url in urls: fetch(url) → each task runs immediately, one after the other

**Phase 1: Playwright Fundamentals**

**What is Playwright?**

Playwright is a modern web automation library that controls real browsers (Chrome, Firefox, Safari) programmatically. Unlike older tools like Selenium, Playwright is **built for modern web apps** and handles JavaScript-heavy sites much better.

**Why Async/Await?**

The most important concept to understand: **Playwright is completely asynchronous**. This means:

* When you tell it to "click a button," it doesn't wait for the click to finish
* You must use await to wait for actions to complete
* This makes it much faster because it can do multiple things at once

# How Await Works in Playwright

In Playwright, 'await' is used to ensure that asynchronous actions complete before the script continues. It is more than just waiting for an element — it also waits for conditions like visibility, stability, and full action completion.

## 🔍 Step-by-Step: await page.click("#submit")

Step 1️⃣: Find the element matching '#submit' (auto-waits if not found immediately)

Step 2️⃣: Wait until the element is:  
 ✅ Visible  
 ✅ Stable  
 ✅ Enabled

Step 3️⃣: Perform the click (mouse down → up)

Step 4️⃣: Optionally wait for page navigation or related effects

## ✅ What Does await Do?

Using 'await' means: "Do everything needed to safely complete this action and wait until it's done."  
Without 'await', the action may run instantly and move on before the element is ready or the task is done.

**If you use await inside a function, that function pauses at the await and waits for the task to finish.** It does **not** move to the next step until the awaited task is complete.

## 🧾 Summary Table

|  |  |
| --- | --- |
| Playwright Action with await | What it Waits For |
| await page.click() | Element ready + click finished + (optionally) page load |
| await page.fill() | Element ready + text typed in completely |
| await page.wait\_for\_selector() | Waits for element to appear or meet a condition |
| No await | Action starts but doesn’t wait — may break the script |

# Installation Process

**# Install the Python library**

**pip install playwright**

**# Download actual browser binaries (Chrome, Firefox, etc.)**

**playwright install**

## Your First Script - Explained Line by Line

**import asyncio**

**from playwright.async\_api import async\_playwright**

**async def main():**

**# This creates a Playwright instance that manages browsers**

**async with async\_playwright() as p:**

**# Launch Chrome browser (headless=False shows the window)**

**browser = await p.chromium.launch(headless=False)**

**# Think of this as opening a new tab**

**page = await browser.new\_page()**

**# Navigate to a website (like typing URL and pressing Enter)**

**await page.goto('https://example.com')**

**# Take a screenshot (useful for debugging)**

**await page.screenshot(path='example.png')**

**# Close the browser**

**await browser.close()**

**# This runs our async function**

**asyncio.run(main())**

**Key Concepts:**

* **async with: Automatically handles setup and cleanup**
* **await: Waits for the action to complete before moving to next line**
* **headless=False: Shows browser window (great for learning/debugging)**
* **page: Think of it as a browser tab**

**Page Navigation & Element Interaction**

**Understanding Web Page Loading**

**Modern websites don't load all at once. They load HTML first, then JavaScript runs to load more content. This is why waiting strategies are crucial.**

**Navigation Strategies Explained**

**async def navigation\_examples(page):**

**# Basic navigation - goes to URL but doesn't wait for JavaScript**

**await page.goto('https://example.com')**

**# Better approach - waits until no network requests for 500ms**

**await page.goto('https://example.com', wait\_until='networkidle')**

**# Set timeout - fail if page doesn't load in 30 seconds**

**await page.goto('https://example.com', timeout=30000)**

**Why networkidle?** Many sites load content via JavaScript after the initial page loads. networkidle waits until the network is quiet, meaning most content has loaded.

## ⚡ 2️⃣ Page Lifecycle & Smart Waiting Strategies (Indepth Guide)

## **🧠 Why This Is Critical**

## **In scraping:**

## **Getting the element too early = None or error** **Waiting too long = slow scraper**

## So Playwright gives you **smart control** to know *when a page is ready and when to act*.

**🔍 Core Concepts to Understand**

**✅ goto() Load Events**

await page.goto(url, wait\_until='networkidle')

| **Option** | **Meaning** |
| --- | --- |
| 'load' | Wait for full page load (default) |
| 'domcontentloaded' | Wait for HTML/DOM parsed, but not full assets |
| 'networkidle' | Wait until no network calls for 500ms (best for JS-heavy sites) |

**Playwright waitUntil Guide: Understanding DOM and Load Events**

**What is the DOM (Document Object Model)?**

The **DOM (Document Object Model)** is a tree-like structure that represents the entire structure and content of a web page. When a browser loads a web page:

1. It parses the HTML.
2. It builds the DOM tree with nodes for each element (like <div>, <p>, <img>, etc.).
3. JavaScript can then dynamically manipulate this DOM to update content, styles, or behavior — without reloading the whole page.
4. Each tag becomes a **node**, and JavaScript (or Playwright) can **read or change** these nodes.

**🧠 Example:**

Given this HTML:

<html>

<body>

<h1>Hello</h1>

<p>This is a paragraph.</p>

</body>

</html>

The DOM would look like this:

Document

└── html

└── body

├── h1 → "Hello"

└── p → "This is a paragraph."

This structure can be accessed and changed using JavaScript. For example:

document.querySelector('h1').textContent = "Hi there!";

The DOM is central to all modern websites because it allows real-time updates, interactivity, and manipulation.

**Playwright waitUntil Options**

| **Option** | **Waits For** | **Best For** | **Example Websites** |
| --- | --- | --- | --- |
| 'domcontentloaded' | HTML parsed, DOM ready (no images, CSS, JS yet) | Fast access to structure/content | Medium.com, blogs |
| 'load' | Full page loaded (DOM, images, styles, JS) | General scraping, full content | Amazon.in, Flipkart |
| 'networkidle' | No network activity for 500ms (good for JS-heavy sites) | SPA/Dynamic content sites | Twitter.com, Instagram |

**Visual Analogy**

* **domcontentloaded**: Structure is built, but furniture (images, styles) is still arriving.
* **load**: Entire house is ready to live in — fully furnished.
* **networkidle**: All delivery trucks have left, nothing more is loading.

**Code Examples**

**1. DOM Content Loaded**

await page.goto("https://medium.com", wait\_until="domcontentloaded")

Use for fast page parsing when you just need structured text/content.

**2. Full Load**

await page.goto("https://www.amazon.in", wait\_until="load")

Use for pages with full product data, images, ads.

**3. Network Idle**

await page.goto("https://twitter.com", wait\_until="networkidle")

Use for JS-heavy sites where content loads after the initial page load.

**Summary**

| **waitUntil** | **Use When You Need...** |
| --- | --- |
| 'domcontentloaded' | Basic HTML structure only |
| 'load' | All page assets fully loaded |
| 'networkidle' | Dynamic JS content finished loading |

Let this guide help you choose the right waitUntil setting depending on the type of site and data you are scraping or interacting with using Playwright.

## ✅ page.wait\_for\_selector()

## Wait for a **specific element** to be ready:

## await page.wait\_for\_selector('.product-title', timeout=10000)

## Waits max 10 seconds for .product-title to appear

## Throws error if not found

## ✅ Why wait\_for\_selector is helpful after login (with intentional delay):

## ⏳ **Handles delayed content**: Some websites load data slowly on purpose after login.

## 🧠 **Prevents errors**: Avoids "element not found" errors if you try too early.

## 🔒 **Bypasses bot traps**: Many sites use slow JS rendering to confuse bots.

## 📡 **Covers network/API lag**: Waits for elements that load via AJAX or API.

## 🚀 **Improves reliability**: Only moves forward when the element is really there.

## ✅ Playwright Auto-Waiting (Why It Matters)

## Playwright automatically waits for elements before interacting — no need for manual sleep() or custom retry loops.

## 🧠 Auto-Wait Applies To

## **await** page.click("button") *# Waits until the button is visible & enabled* **await** page.text\_content("h1") *# Waits until <h1> exists and is ready*

## These commands **won’t fail immediately** if the element isn’t ready.

## Playwright retries until timeout (default: 30 seconds).

## ⚠️ Still Use wait\_for\_selector() When:

## Use it before **critical logic or complex steps** to ensure the page is truly ready:

## **await** page.wait\_for\_selector(".dashboard-loaded")

## Good Situations to Use It:

## After **login redirects**

## On **JS-heavy pages** (e.g., Twitter, Instagram)

## When **important data or UI elements** load via API calls

## If the site **intentionally delays** content to deter bots

## 💡 Best Practice:

## Combine auto-wait + wait\_for\_selector() for reliable scraping and automation.

## **await** page.wait\_for\_selector(".product") **await** page.click(".product")

## This ensures the element exists **before** trying to interact.

## **🧾 Comparison Table with Simple Story and Code Examples**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature** | **wait\_for\_selector()** | **Auto-Wait (e.g., click(), text\_content())** | **Story-style Understanding** | **Code Example** |
| What it does | Just **waits** for the element to appear | **Waits, then acts** (click, read text, etc.) | You **look at the door and wait** for it to open | await page.wait\_for\_selector(".product") |
| When it runs | When **you call it manually** | Happens **inside the action** | You say: "Open the door" — and the system waits a bit before trying | await page.click(".product") |
| Action performed | ❌ No action (only waits) | ✅ Yes, performs action after waiting | click() = Waits → Then clicks | await page.text\_content(".price") |
| Control level | ✅ Full control — when and how long to wait | ❌ Less control — it's automatic | Like waiting yourself vs someone doing it for you | — |
| Best use case | Critical steps, conditions, slow loading | Simple interactions — buttons, input, read text | wait is for planning, click() is like pressing the button and hoping | Login + wait → then action |

## Smart Wait Techniques You Must Know

| **Technique** | **Example** | **When to Use** |
| --- | --- | --- |
| wait\_for\_selector() | await page.wait\_for\_selector(".price") | Scraping JS-rendered values |
| wait\_for\_load\_state() | await page.wait\_for\_load\_state("networkidle") | After a page click or redirect |
| is\_visible() | await locator.is\_visible() | Check before action |
| try/except wait | Retry on failure | Unreliable pages (like Meesho!) |

**✅ Playwright: Using is\_visible() with wait\_for\_selector()**

**🔍 Purpose:**

To safely check if an element is **visible** on the page.

**✅ Best Practice:**

Use wait\_for\_selector() first to ensure the element **exists**, then check visibility:

await page.wait\_for\_selector(".product")

if await page.is\_visible(".product"):

print("Product is visible.")

else:

print("Product is hidden.")

**💡 Summary:**

* wait\_for\_selector() → Waits until the element exists (even if hidden)
* is\_visible() → Returns True if the element is shown on the page

Together, they help avoid errors and write safe, conditional automation logic.

------------------------------------------------------------------------------------------------------------------------------

**🧪 Real Example: Flipkart Product Wait Strategy**

await page.goto(url, wait\_until="networkidle")

try:

await page.wait\_for\_selector(".\_30jeq3", timeout=7000)

price = await page.text\_content(".\_30jeq3")

except:

price = "Not Found"

| **Scenario** | **Why Use timeout?** |
| --- | --- |
| 🔒 **Login Redirects** | Avoid waiting forever on failed logins |
| ⏳ **Slow API-based UI elements** | Allow time for dynamic content to load |
| ⚠️ **Optional elements (popups, errors)** | Don't wait full 30s if element may not appear |
| 🧪 **Test speed limits** | Speed up test failure if something breaks |
| 🌐 **Bad networks** | Reduce unnecessary delay on timeout pages |

**Custom Wait with Retry Logic**

async def safe\_wait(page, selector, retries=3):

for \_ in range(retries):

try:

await page.wait\_for\_selector(selector, timeout=5000)

return await page.text\_content(selector)

except:

continue

return None

**Common Mistakes to Avoid**

| **Mistake** | **Fix** |
| --- | --- |
| ❌ Clicking before page loads | ✅ Use wait\_for\_load\_state() |
| ❌ Relying only on goto() | ✅ Combine with wait\_for\_selector() |
| ❌ Using time.sleep() | ✅ Use Playwright's async wait tools |
| ❌ Timeout errors | ✅ Use try/except around waits |
| ❌ Wrong selector | ✅ Inspect element via Chrome DevTools! |

**Visual Debug Mode**

Turn off headless to **see what’s happening**:

await p.chromium.launch(headless=False, slow\_mo=300)

* Helps you visualize if Playwright is *waiting properly*
* Add slow\_mo to slow interactions — better debugging

**✅ Playwright: wait\_for\_load\_state() Explained**

**🔍 Purpose:**

Waits until the page finishes loading **after a click or navigation**.

**✅ Basic Use:**

await page.click("a.login-link")

await page.wait\_for\_load\_state() # Waits until new page is ready

await page.fill("#username", "test")

**🧠 Why Use It?**

* Prevents errors like "element not found"
* Ensures the **next page or content is ready** before interaction

**📦 Load States:**

| **Option** | **Meaning** |
| --- | --- |
| "load" | Full page loaded (HTML, images, styles) |
| "domcontentloaded" | Only HTML DOM loaded |
| "networkidle" | No network/API activity for 500ms |

Example:

await page.wait\_for\_load\_state("networkidle")

**💡 Summary:**

Use wait\_for\_load\_state() **after actions that trigger a new page or dynamic change** — it ensures you're not interacting too early.

**✅ Summary**

| **Must-Know Waits** | **Purpose** |
| --- | --- |
| goto(..., wait\_until='networkidle') | Wait for page finish |
| wait\_for\_selector() | Wait for element to appear |
| wait\_for\_load\_state() | Wait for navigation after click |
| slow\_mo + headless=False | Debug visually |
| Custom retry logic | Handle flaky sites |

**🧪 Flipkart Scraper: Price, Rating, Reviews (One Product Page)**

**🔧 Requirements:**

pip install playwright

playwright install

**🚀 flipkart\_scraper.py**

import asyncio

from playwright.async\_api import async\_playwright

from datetime import datetime

async def scrape\_flipkart(url):

async with async\_playwright() as p:

browser = await p.chromium.launch(headless=False, slow\_mo=250)

context = await browser.new\_context()

page = await context.new\_page()

print(f"\nScraping: {url}")

await page.goto(url, wait\_until='networkidle')

data = {}

# --- Title ---

try:

await page.wait\_for\_selector(".\_2FEgSb", timeout=8000)

data["title"] = await page.text\_content(".\_2FEgSb")

except:

data["title"] = "Title not found"

# --- Price ---

try:

await page.wait\_for\_selector(".\_30jeq3", timeout=8000)

data["price"] = await page.text\_content(".\_30jeq3")

except:

data["price"] = "Price not found"

# --- Rating ---

try:

await page.wait\_for\_selector(".\_3LWZlK", timeout=8000)

data["rating"] = await page.text\_content(".\_3LWZlK")

except:

data["rating"] = "Rating not found"

# --- Reviews ---

try:

await page.wait\_for\_selector(".\_2\_R\_DZ", timeout=8000)

review\_block = await page.text\_content(".\_2\_R\_DZ")

data["reviews"] = review\_block

except:

data["reviews"] = "Reviews not found"

data["scraped\_at"] = datetime.now().isoformat()

await browser.close()

return data

# Test with a Flipkart product

if \_\_name\_\_ == "\_\_main\_\_":

product\_url = "https://www.flipkart.com/pintola-dark-chocolate-peanut-butter-creamy/p/itmc121d193cb0f4"

scraped\_data = asyncio.run(scrape\_flipkart(product\_url))

print("\n📦 Scraped Data:\n", scraped\_data)

**📊 Output Example**

{

"title": "Pintola Dark Chocolate Peanut Butter (Creamy)",

"price": "₹325",

"rating": "4.4",

"reviews": "1,038 Ratings & 106 Reviews",

"scraped\_at": "2025-07-09T22:12:55"

}

**✅ context** = await browser.new\_context()  
Creates a **new, clean browser session** — like an **incognito tab**.

* No cookies or login from other sessions
* Useful for simulating different users
* Needed before creating a page

context = await browser.new\_context()

page = await context.new\_page()

**✅ 3️⃣ Playwright Selectors & Locators – In Depth Guide**

*This is how you tell Playwright* ***what*** *to click, read, or wait for.*

**🎯 What Is a Selector?**

A **selector** is a way to identify an element on the page.  
In Playwright, you can use:

**🎯 Playwright Selector Cheat Sheet**

Quick reference for selecting elements using different strategies in Playwright.

**✅ Common Selector Types**

| **Type** | **Example** | **Description** |
| --- | --- | --- |
| **CSS** | "button.buy-now" | Targets <button class="buy-now"> |
| **XPath** | "//button[contains(text(), 'Buy Now')]" | Finds a <button> with text that includes “Buy Now” |
| **Text** | "text='Add to Cart'" | Finds element with exact text "Add to Cart" |
| **Regex Text** | /Add\s+to\s+Cart/ | Finds text using flexible matching (e.g. spacing, cases) |
| **Attribute** | "img[alt='product']" | Targets <img> elements with a specific alt attribute |
| **Role** | "role=button[name='Submit']" | Finds an ARIA button with the name “Submit” (great for accessibility) |

**🔥 Tips**

* Prefer **CSS** for most static content
* Use **Text** or **Role** for better readability and test stability
* **Regex** helps when text isn't consistent
* **XPath** is powerful but less readable — use only when needed

Use these selectors with functions like:

await page.click("text='Add to Cart'")

await page.fill("input[name='email']", "user@example.com")

await page.is\_visible("img[alt='product']")

Stay consistent with selector types for clean, maintainable scripts!

**Element Selection - The Foundation of Scraping**

Think of selectors as "addresses" for elements on a webpage. There are several ways to address the same element:

*# CSS Selectors (most common)*

await page.click('#submit-btn') *# Element with ID "submit-btn"*

await page.click('.login-button') *# Element with class "login-button"*

await page.click('button') *# First <button> element*

await page.click('input[name="email"]') *# Input with name="email"*

*# Text-based (most reliable for scraping)*

await page.click('text=Submit') *# Button containing "Submit"*

await page.click('text="Exact text"') *# Exact text match*

*# XPath (powerful but complex)*

await page.click('//button[contains(text(), "Submit")]')

**Pro Tip:** Text-based selectors are often most reliable because they're less likely to break when websites change their CSS.

**Form Interaction Explained:**

**async def form\_interaction(page):**

**# Fill text inputs**

**await page.fill('#username', 'myuser') # Types text into input**

**await page.fill('input[name="password"]', 'mypass')**

**# Press keys**

**await page.press('#search-box', 'Enter') # Simulates pressing Enter**

**# Select from dropdown**

**await page.select\_option('select#country', 'USA') # Selects option by value**

**What's happening:** These actions simulate exactly what a human would do - typing, clicking, selecting options.

**💡 Playwright Best Practice: locator() API with Examples**

The locator() API is the **recommended way** to select and interact with elements in Playwright.

**✅ Why locator() is Better**

* **Reusable reference to elements**  
  Once created, a locator points to the same element and can be used multiple times. Perfect when you need to check visibility, extract text, and click the same element in different steps.
* **Built-in auto-waiting**  
  Every locator action waits for the element to appear and be ready — you don’t need to call wait\_for\_selector() separately. This prevents race conditions.
* **Chainable for actions**  
  You can chain actions like .click(), .fill(), .text\_content() directly, making your code cleaner and more readable.
* **Cleaner and more reliable for real-world scraping/testing**  
  It handles timing issues, retries intelligently, and works well with dynamic pages — making your automation more stable.

**🖱️ Real-World Analogy: Like a Human Using a Mouse**

Using locator() is like pointing at an object with your mouse and using it again and again for different tasks — just like a human would:

| **Human Action** | **Playwright Equivalent** |
| --- | --- |
| You look at a "Buy Now" button | locator("text='Buy Now'") |
| You point your mouse to it | You create a locator targeting that button |
| You click it | await locator.click() |
| You read the text on it | await locator.text\_content() |
| You want the first from many buttons | locator(".btn").first() |

This makes your automation feel more like a human interaction — one pointer, many tasks.

**🔍 Real Example: Flipkart Product Data Scraping**

# Select and extract product data using locators

# Product title

title = await page.locator("span.B\_NuCI").text\_content()

# Product price

price = await page.locator(".\_30jeq3").text\_content()

# Product rating (stars)

rating = await page.locator(".\_3LWZlK").text\_content()

# Total number of reviews

reviews = await page.locator(".\_2\_R\_DZ span:nth-child(1)").text\_content()

# Print results

print("Title:", title)

print("Price:", price)

print("Rating:", rating)

print("Reviews:", reviews)

**🧠 Notes:**

* All locators **auto-wait** for the element to be visible.
* Use CSS selectors inside locator().
* You can also chain with .first(), .last(), .nth(i), .count().

✅ **Summary**: Use locator() for clean, stable, and modern Playwright scripts.  
Great for scraping, testing, and real-time automation.

**🧾 Selector Cheat Sheet for Beginners (No HTML/CSS Knowledge Needed)**

|  |  |  |
| --- | --- | --- |
| Selector Type | Example | Meaning |
| **Text Selector** | text="Add to Cart" | Finds element with **exact visible text** |
| **Regex Text** | /Add\s+to\s+Cart/ | Matches text with flexible spacing |
| **Role Selector** | role=button[name='Submit'] | Matches ARIA role buttons (like "Submit") |
| **Attribute** | img[alt='product'] | Finds image tag with alt="product" |
| **Class** | .buy-now | Matches elements with class buy-now |
| **ID** | #login-button | Matches element with id login-button |
| **XPath** | //button[contains(text(), 'Buy Now')] | XPath to find a button with specific text (advanced) |

🔹 Use **Text**, **Role**, or **Attribute** selectors if you're new — they're easier to understand.

**🧱 Selector Strategies**

| **Strategy** | **Use When** |
| --- | --- |
| **Text Selector** | Button with unique text |
| **CSS** | Most elements with classes/IDs |
| **XPath** | Complex logic (contains, starts-with) |
| **Role Selectors** | Accessibility (screen readers) |
| **Nth-child** | Lists or rows with same class |

**🧪 Bonus: Looping Through Multiple Products**

products = page.locator(".product-item")

count = await products.count()

for i in range(count):

title = await products.nth(i).locator(".title").text\_content()

print(f"{i+1}. {title}")

| **Line** | **What It Does** |
| --- | --- |
| products = page.locator(".product-item") | Finds **all elements** matching the .product-item class and stores them as a group (like a list). |
| count = await products.count() | Counts how many .product-item elements were found. |
| for i in range(count): | Loops through **each product** one by one using its index. |
| products.nth(i) | Selects the **i-th product** from the list (like picking one from a shelf). |
| .locator(".title") | Inside that product, it looks for an element with class .title. |
| .text\_content() | Gets the text inside that title (like the product name). |
| print(f"{i+1}. {title}") | Prints it in a numbered list (e.g. 1. Product A). |

**✅ Playwright Inspector (Use This!)**

Run with DEBUG=pw:api or use:

playwright codegen https://flipkart.com

🧠 This opens a visual browser that **records your actions as code** and helps you find perfect selectors.

⚡ Quick Guide: Playwright Inspector & Codegen

Playwright Inspector is a tool that helps you visually debug scripts and record code by clicking on a website.

▶️ Start the Visual Recorder

playwright codegen https://example.com

✅ Opens a browser and records your actions as Playwright code.

Example output:

await page.click("text='Login'")

await page.fill("input[type='text']", "user@example.com")

You can also save it directly:

playwright codegen https://example.com > script.py

📱 Emulate Devices Easily

Record for mobile or tablet:

playwright codegen --device="iPhone 13" https://example.com

✅ Helps you test and record automation on different screen sizes.

🐞 Debug Existing Scripts Visually

Add this to your script:

browser = await playwright.chromium.launch(headless=False, slow\_mo=50)

Run with:

DEBUG=pw:api python script.py

✅ Shows each action step-by-step in a browser window.

🧠 Selector Tips

* Hover elements in codegen to highlight their best selector
* Prefer text=, role=, or stable class names
* Avoid long or auto-generated XPath when possible

🎯 Why Use It?

| **Tool** | **Purpose** |
| --- | --- |
| **codegen** | **Record code by interacting visually** |
| **--device** | **Emulate mobile/tablet for testing** |
| **DEBUG=pw:api** | **Debug each Playwright step live** |
| **slow\_mo** | **Slow motion to watch actions clearly** |
| **>** | **Save your recorded script to file** |

✅ Summary

Use Playwright Inspector to:

* Record working code fast
* Find perfect selectors visually
* Debug your scripts like a pro
* Test across devices and screen sizes
* Save hours of manual inspection

## **🔥 Pro Tips**

| **Tip** | **Why** |
| --- | --- |
| ✅ Always use locator() over querySelector style | Resilience & retries |
| ✅ Prefer text= or role= for buttons | Less fragile |
| ❌ Avoid absolute XPath | Breaks easily |
| ✅ Use nth() or loop for multiple items | For lists/grids |
| ✅ Use page.locator("selector").is\_visible() before clicking | Avoid hidden bugs |

## **🧠 Summary Table**

| **Selector Type** | **Use Example** | **Good For** |
| --- | --- | --- |
| text= | "text='Buy Now'" | Buttons, links |
| css | ".product-title" | Most standard scrapes |
| xpath | "//div[contains(text(),'₹')]" | Flexible conditions |
| role= | "role=button[name='Add']" | Accessibility-rich sites |
| locator().nth(i) | nth(0) | Looping in grid or cards |

**🎯 Playwright Selector Comparison Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Selector Type | Syntax Example | Use Case | Playwright Support |
| CSS Selector | div.product-title | Fast selection by class, ID, tag, or structure | ✅ Yes |
| XPath Selector | //div[contains(text(),'Laptop')] | Complex logic or text/attribute-based matching | ✅ Yes |
| Text Selector | page.get\_by\_text("ASUS") | Selects element by visible text | ✅ Yes (Playwright-only) |
| Role Selector | page.get\_by\_role("button", name="Buy") | Semantic roles (great for accessibility testing) | ✅ Yes |
| Label Selector | page.get\_by\_label("Email") | Select form inputs by associated label | ✅ Yes |
| Placeholder Selector | page.get\_by\_placeholder("Search...") | Target inputs with placeholder text | ✅ Yes |
| Alt Text Selector | page.get\_by\_alt\_text("Logo") | Target images by alt attribute | ✅ Yes |
| Test ID Selector | page.get\_by\_test\_id("submit-btn") | Targets custom test attributes like data-testid | ✅ Yes |
| CSS with Text | div:has-text('Laptop') | Combines structure with visible text (Playwright-only) | ✅ Yes |

**⏱️ Minutes 30-45: Waiting Strategies & Error Handling**

**The Waiting Problem**

**The #1 cause of scraping failures:** Trying to interact with elements before they exist. Modern websites load content dynamically, so you must wait for elements to appear.

**The 5 Types of Waits - When to Use Each**

async def waiting\_strategies(page):

# 1. Wait for element to appear (most common)

await page.wait\_for\_selector('#dynamic-content')

# Use when: Content loads via JavaScript

# 2. Wait for element to disappear

await page.wait\_for\_selector('#loading-spinner', state='hidden')

# Use when: Waiting for loading indicators to disappear

# 3. Wait for network to be idle

await page.wait\_for\_load\_state('networkidle')

# Use when: Page loads content via multiple API calls

# 4. Wait for specific timeout

await page.wait\_for\_timeout(2000) # 2 seconds

# Use when: You know exactly how long something takes

# 5. Wait for JavaScript condition

await page.wait\_for\_function('window.dataLoaded === true')

# Use when: Waiting for specific JavaScript variables

| **Method** | **Inside Quotes (📥 Input)** | **What It Refers To** | **Comes From** |
| --- | --- | --- | --- |
| wait\_for\_selector('#id') | CSS selector (e.g. '#id', '.class', 'div') | An HTML element in the page | ✅ Webpage HTML |
| wait\_for\_selector("text='Buy'") | Text-based selector | Element with exact visible text | ✅ Webpage text |
| wait\_for\_selector('//div[@id="x"]') | XPath selector | Uses XPath to locate HTML | ✅ Webpage HTML |
| wait\_for\_function('window.ready') | JavaScript condition (string) | A JS variable/flag in the browser | ✅ Webpage JS |
| wait\_for\_load\_state('networkidle') | Page loading state: 'load', 'domcontentloaded', 'networkidle' | Browser-level state | ✅ Web browser engine |
| wait\_for\_timeout(2000) | Milliseconds (not a selector) | Just waits a fixed time | ❌ Not related to HTML |
| locator("img[alt='product']") | CSS attribute selector | <img alt="product"> in HTML | ✅ Webpage HTML |

**Smart Waiting with Locators (Recommended Approach)**

async def smart\_waiting(page):

# Playwright automatically waits for elements with locators

button = page.locator('button')

await button.click() # Auto-waits for button to be visible and clickable

# You can also specify what to wait for

await page.locator('#submit').wait\_for(state='visible')

await page.locator('#loading').wait\_for(state='hidden')

**Why locators are better:** They automatically handle waiting, making your code more reliable.

**Error Handling - Making Your Scraper Bulletproof**

async def error\_handling\_examples(page):

try:

await page.goto('https://example.com', timeout=30000)

# Handle optional elements (might not exist)

try:

await page.click('#optional-button', timeout=5000)

except:

print("Optional button not found, continuing...")

# Check if element exists before interaction

if await page.locator('#conditional-element').count() > 0:

await page.click('#conditional-element')

except Exception as e:

print(f"Error occurred: {e}")

await page.screenshot(path='error.png') # Debug screenshot

**Why error handling matters:** Websites change constantly. Your scraper must handle missing elements gracefully.

**Understanding Data Extraction**

Data extraction is about getting information from HTML elements. You need to understand:

* **Text content**: What users see
* **Attributes**: Hidden information (URLs, IDs, etc.)
* **Structure**: How data is organized on the page

**Text Extraction Explained**

async def text\_extraction(page):

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

# Extract visible text from element

title = await page.text\_content('h1') # Gets "Welcome to Our Site"

# Extract from multiple elements

links = await page.locator('a').all\_text\_contents() # Gets all link texts

# Extract with fallback (if element doesn't exist)

price = await page.text\_content('.price') or 'N/A'

# Extract HTML content (includes tags)

content = await page.inner\_html('.content') # Gets "<p>Hello</p>"

# Extract attributes (hidden data)

href = await page.get\_attribute('a', 'href') # Gets "https://example.com"

src = await page.get\_attribute('img', 'src') # Gets image URL

**Structured Data Extraction - The Professional Approach**

async def extract\_product\_list(page):

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

# Find all product containers

product\_cards = page.locator('.product-card')

products = []

# Loop through each product

for i in range(await product\_cards.count()):

card = product\_cards.nth(i) # Get the i-th product

# Extract data from this specific product

product = {

'name': await card.locator('.product-name').text\_content(),

'price': await card.locator('.price').text\_content(),

'rating': await card.locator('.rating').text\_content(),

'image': await card.locator('img').get\_attribute('src')

}

products.append(product)

return products

**Key Insight:** Instead of getting one element, you get a list of similar elements and extract data from each.

**Complete Example with Explanations**

import asyncio

from playwright.async\_api import async\_playwright

import json

from datetime import datetime

async def scrape\_product(url):

"""

Scrapes product data from a single product page

Returns structured data as a dictionary

"""

async with async\_playwright() as p:

# Launch browser (headless=False for learning)

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

page = await browser.new\_page()

try:

# Navigate and wait for page to fully load

await page.goto(url, wait\_until='networkidle')

# Wait for main content to appear

await page.wait\_for\_selector('h1')

# Extract product data

product\_data = {

'title': await page.text\_content('h1'),

'price': await page.text\_content('.price, .a-price-whole'),

'rating': await page.text\_content('.rating, .a-icon-alt'),

'availability': await page.text\_content('.availability, .a-color-success'),

'scraped\_at': datetime.now().isoformat(),

'url': url

}

# Clean up data (remove extra whitespace)

for key, value in product\_data.items():

if value:

product\_data[key] = value.strip()

return product\_data

except Exception as e:

print(f"Error scraping {url}: {e}")

return None

finally:

await browser.close() # Always close browser

**What this does step-by-step:**

1. Opens a browser
2. Goes to the product page
3. Waits for content to load
4. Extracts specific data points
5. Cleans up the data
6. Returns structured data
7. Handles errors gracefully

**🎯 Practice Exercises (Build Understanding Step by Step)**

**Exercise 1: Basic Navigation (10 minutes)**

**Goal:** Understand browser control and page loading

1. Create a script that visits 3 different websites
2. Take screenshots of each
3. Print the page titles

**Learning Point:** You'll see how different sites load at different speeds

**Exercise 2: Form Interaction (10 minutes)**

**Goal:** Understand user simulation

1. Go to a search engine
2. Fill in a search query
3. Submit the form
4. Wait for results to load

**Learning Point:** You'll understand timing and waiting

**Exercise 3: Data Extraction (15 minutes)**

**Goal:** Understand element selection

1. Visit a news website
2. Extract all article headlines
3. Save them to a list
4. Print the count

**Learning Point:** You'll see how to work with multiple similar elements

**Exercise 4: Error Handling (10 minutes)**

**Goal:** Understand failure modes

1. Try to visit a non-existent website
2. Handle the error gracefully
3. Take a screenshot when errors occur

**Learning Point:** You'll learn how things can go wrong

**Exercise 5: Complete Product Scraper (15 minutes)**

**Goal:** Put it all together

1. Choose any e-commerce product page
2. Extract title, price, and rating
3. Save to JSON file
4. Add proper error handling

**Learning Point:** You'll have a working scraper!

**🧠 Key Mental Models to Develop**

**1. Async Thinking**

* Every action takes time
* You must wait for actions to complete
* Multiple actions can happen simultaneously

**2. Element Selection Strategy**

* Always have backup selectors
* Text-based selectors are most reliable
* Test selectors in browser dev tools first

**3. Waiting Strategy**

* Modern sites load content dynamically
* Always wait for elements before interacting
* Use specific waits, not generic timeouts

**4. Error Handling Mindset**

* Websites change constantly
* Elements might not exist
* Always have fallback plans

**🚨 Common Beginner Mistakes & How to Avoid Them**

**1. Forgetting await**

python

*# WRONG - This doesn't wait for the page to load*

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

*# CORRECT - This waits for the page to load*

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

**2. Not handling dynamic content**

python

*# WRONG - Might fail if content loads slowly*

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

title = await page.text\_content('h1')

*# CORRECT - Waits for content to appear*

await page.goto('https://example.com', wait\_until='networkidle')

await page.wait\_for\_selector('h1')

title = await page.text\_content('h1')

**3. Hardcoded selectors**

python

*# FRAGILE - Breaks when CSS changes*

await page.click('#submit-btn-v2-new')

*# ROBUST - Works even if CSS changes*

await page.click('text=Submit')

**✅ Phase 1 Mastery Checklist**

You've mastered Phase 1 when you can:

* Explain why Playwright uses async/await
* Launch browsers and navigate pages confidently
* Choose appropriate selectors for different situations
* Implement proper waiting strategies
* Handle errors gracefully
* Extract structured data from web pages
* Build a complete working scraper

**Success Indicator:** You can build a product scraper for any e-commerce site in under 30 minutes.

**🎯 Ready for Phase 2?**

You're ready for **Phase 2: Anti-Bot Detection & Stealth** when you:

* Feel comfortable with async/await patterns
* Can reliably extract data from any website
* Understand why waiting strategies matter
* Have built at least 3 different scrapers

**Next Up:** We'll learn how to make your scrapers undetectable and handle anti-bot measures!

*💡 Remember: Understanding beats memorizing. Focus on why each technique works, not just how to use it.*