Playwright con Python: Guía Completa de Automatización Moderna de Pruebas

Aprende a crear pruebas rápidas, fiables y escalables para la web con Playwright paso a paso

Creado por "Roberto Arce"

Este libro está basado en experiencias reales y contiene opiniones sobre el ejercicio profesional de la calidad en proyectos de software.
Nombres de productos, empresas o situaciones reales se mencionan únicamente con fines educativos.

Primera edición: 2025
Diseño y estrategia editorial: **QA sin filtros**

Publicado por el autor a través de Amazon Kindle Direct Publishing (KDP)
www.amazon.com/kdp

Prólogo

En el mundo del desarrollo de software moderno, la velocidad y la calidad ya no son opcionales: son requisitos indispensables. Las entregas continuas, el desarrollo ágil y la experiencia del usuario han puesto a prueba a los equipos de QA como nunca antes. Es aquí donde entra en juego **Playwright**, una herramienta revolucionaria que lleva la automatización de pruebas al siguiente nivel.

Este libro nace de la necesidad de contar con una guía **práctica, estructurada y actualizada** para dominar Playwright desde cero, sin importar si vienes de Selenium, Cypress u otras herramientas. A lo largo de sus páginas, recorrerás desde los fundamentos esenciales hasta la construcción de **frameworks de testing avanzados**, explorando aspectos como pruebas en múltiples navegadores, ejecución paralela, integración continua, testing visual y mucho más.

Si estás comenzando tu camino en la automatización, te recomiendo también explorar mis otros títulos:

1. La Biblia de Python: un recurso completo para dominar el lenguaje que potencia muchas soluciones de automatización modernas.
2. La Biblia de Selenium: ideal para quienes inician en automatización tradicional basada en WebDriver.

Mi objetivo no es solo enseñarte comandos y scripts, sino ayudarte a **pensar como un QA moderno**, capaz de integrar pruebas automatizadas en cualquier stack de desarrollo, con buenas prácticas, diseño limpio y enfoque profesional.

Ya seas desarrollador, QA manual, tester en transición o ingeniero de automatización, este libro te proporcionará **el conocimiento técnico y las habilidades prácticas** para destacar en entornos exigentes.

Bienvenido a una nueva era de automatización web.
Prepárate para dominar Playwright.

ÍNDICE GENERAL

Una guía completa para dominar la automatización de pruebas modernas con Playwright, desde los fundamentos y el diseño de proyectos hasta la integración continua y estrategias de migración profesional.

CAPÍTULO 4: Selección de Elementos en Playwright con CSS y XPath

- Introducción a los selectores
- Selectores CSS en Playwright
- Selectores XPath en Playwright
- Comparativa técnica: CSS vs XPath
- Buenas prácticas para selectores robustos y mantenibles
- **Ejemplos prácticos:** seleccionando elementos en Wikipedia
- **Ejercicios prácticos:** localización avanzada de elementos

CAPÍTULO 5: Pruebas Avanzadas con Playwright y Python

- Esperas inteligentes (*auto-waiting*)
- Capturas y evidencias de ejecución
  - Capturas de pantalla (*screenshots*)
  - Trazas de ejecución (*tracing*)
  - Grabación de video de las pruebas

- Manejo de múltiples pestañas e *iframes*
  - Apertura y control de nuevas páginas
  - Trabajo con *iframes* embebidos

- **Ejercicios prácticos:** evidencia y depuración avanzada

CAPÍTULO 6: Paralelismo y Escalabilidad con Playwright y Pytest

- Introducción al paralelismo y escalabilidad
- Preparación del entorno de ejecución paralela
- Ejecución de pruebas en paralelo con **pytest-xdist**
- Configuración de *workers* y gestión de recursos
- Ejemplo práctico: pruebas paralelas en diferentes navegadores
- Escalabilidad en entornos CI/CD y Docker
- **Conclusión:** mayor velocidad y rendimiento con Playwright

CAPÍTULO 7: Integración Continua con GitHub Actions y Reportes

- Configuración de CI/CD con Playwright
- Integración con **GitHub Actions**
  - Ejecución automática de tests en cada *commit*
  - Matriz de navegadores (Chrome, Firefox, WebKit)

- Reportes HTML y artefactos en GitHub Actions
- Estructura profesional del repositorio
- Buenas prácticas para pipelines de pruebas
- **Conclusión:** automatizando la calidad con integraciones modernas

CAPÍTULO 8: Casos Prácticos con Playwright en Python

- **Automatización de login:** credenciales válidas e inválidas
- **Formularios interactivos:**
  - Campos de texto, email, *select*, *checkbox* y validaciones
  - Envío exitoso y manejo de errores

- **Prueba End-to-End completa:**
  - Login → navegación a catálogo → agregar al carrito → checkout → confirmación

- **Ejercicios prácticos:** replicando flujos reales de usuario

CAPÍTULO 9: Comparativa y Estrategias de Migración a Playwright

- Comparativa técnica con Selenium y Cypress
  - Arquitectura
  - Rendimiento
  - Experiencia de desarrollo
  - Cobertura de navegadores y lenguajes
  - Facilidad de mantenimiento
  - Ecosistema y soporte

- Estrategias de migración
  - Desde Selenium a Playwright
  - Desde Cypress a Playwright

- Recomendaciones según el contexto del proyecto
- **¡Da el siguiente paso!** — profesionaliza tus pruebas con Playwright

BONUS: Recursos Profesionales

- Instaladores y dependencias útiles
- Plugins y librerías recomendadas
- Extensiones para VSCode y PyCharm
- Frameworks complementarios (pytest-html, allure, xdist)
- Comunidades y fuentes de actualización sobre Playwright

Capítulo 1: Introducción a Playwright

¿Qué es Playwright? Propósito, historia y ecosistema

Playwright es un **framework de automatización de pruebas end-to-end** para aplicaciones web, de código abierto y mantenido por Microsoft. Su propósito principal es permitir a desarrolladores y testers **simular las acciones de un usuario** en un navegador (clics, entradas de texto, navegación, etc.) de forma automatizada, con el fin de verificar el comportamiento y la calidad de las aplicaciones web. En otras palabras, con Playwright podemos escribir scripts que abren páginas web, interactúan con elementos de la interfaz y validan resultados, todo ello sin intervención humana.

Breve historia: Playwright fue lanzado inicialmente en **2020** por un equipo de Microsoft que anteriormente había trabajado en la librería Puppeteer de Google. A diferencia de Puppeteer (que solo funcionaba con Chrome/Chromium), Playwright nació con una visión *multi-navegador*. Desde su aparición, ha ganado popularidad rápidamente gracias a su enfoque moderno y a las mejoras que ofrece frente a herramientas más antiguas. Al ser un proyecto respaldado por Microsoft y la comunidad open-source, recibe actualizaciones frecuentes y un soporte activo.

Ecosistema: Una de las fortalezas de Playwright es su amplio ecosistema y soporte multiplataforma. Algunas características clave de este ecosistema son:

- **Soporte de múltiples navegadores:** Playwright permite automatizar *Chromium/Chrome (y derivados como Edge)*, *Firefox* y *WebKit (motor de Safari)* con una sola API. Esto significa que un mismo script de prueba puede ejecutarse en todos esos navegadores para verificar la compatibilidad. (Internamente, Playwright descarga versiones específicas de estos navegadores para garantizar resultados consistentes.)
- **Soporte de múltiples lenguajes:** Aunque en este libro nos centraremos en **Python**, Playwright ofrece bibliotecas en otros lenguajes populares como **JavaScript/TypeScript**, **Java** y **C# (.NET)**. La API es muy similar entre idiomas, lo que facilita a los equipos usar Playwright sin importar su lenguaje preferido.
- **Arquitectura moderna:** Playwright se integra directamente con los navegadores mediante protocolos de bajo nivel (como el protocolo DevTools de Chrome) a través de una conexión WebSocket. Esto permite una comunicación rápida y confiable, evitando la necesidad de procesos intermedios. Gracias a esta arquitectura, las pruebas con Playwright suelen ser **más rápidas** y menos propensas a errores de sincronización.
- **Herramientas integradas:** El ecosistema incluye utilidades que mejoran la productividad. Por ejemplo, Playwright cuenta con un **generador de código**

(`playwright codegen`) que puede grabar nuestras acciones en el navegador y generar el código de prueba automáticamente. También ofrece un **inspector** y un **visor de trazas** (trace viewer) para depurar pruebas, permitiendo ver paso a paso qué ocurrió en cada ejecución. Estas herramientas hacen más fácil crear y mantener pruebas robustas.

- **Ejecutabilidad en paralelo y CI:** Playwright está diseñado pensando en la ejecución en paralelo de pruebas y la integración continua. Es sencillo ejecutar múltiples pruebas a la vez (incluso lanzar múltiples navegadores simultáneamente) para acelerar la suite de testing. Además, funciona en **Windows, Linux y macOS**, tanto en entornos de desarrollo locales como en servidores de CI/CD (incluso en modo *headless*, es decir, sin interfaz gráfica). Esto lo hace muy adaptable a distintos flujos de trabajo de QA.

En resumen, Playwright es un framework moderno que proporciona una **solución completa para pruebas automatizadas** de aplicaciones web, combinando la compatibilidad de Selenium con la simplicidad y velocidad de herramientas recientes. A continuación, exploraremos cómo se compara específicamente con Selenium y Cypress, dos de los frameworks más conocidos en esta área.

Ventajas de Playwright frente a Selenium y Cypress

Playwright surge en un ecosistema donde **Selenium** ha sido la herramienta tradicional por más de una década y **Cypress** una alternativa más reciente enfocada en desarrolladores front-end. A continuación, analizamos conceptualmente las ventajas que ofrece Playwright en comparación con Selenium y Cypress:

- **Arquitectura rápida y confiable:** Playwright utiliza una comunicación directa con el navegador mediante una conexión persistente (WebSocket), lo que reduce la latencia en la ejecución de comandos. En Selenium, los comandos se envían vía el protocolo WebDriver (HTTP), introduciendo más sobrecarga y posibles demoras. En la práctica, esto significa que **Playwright ejecuta las pruebas más rápido** que Selenium en muchos casos. En cuanto a **Cypress**, su arquitectura ejecuta las pruebas dentro del propio navegador; esto también resulta veloz, pero Cypress está limitado a un solo proceso/navegador a la vez. Playwright, por su parte, puede **controlar múltiples navegadores o pestañas en paralelo**, aprovechando mejor los recursos y escalando en entornos de CI. En definitiva, a nivel de rendimiento y arquitectura, Playwright está optimizado para la velocidad sin sacrificar estabilidad.
- **Soporte multi-navegador moderno:** Una ventaja distintiva de Playwright es su soporte nativo para los principales motores de navegador: Chromium (que cubre Chrome, Edge, Brave, etc.), Firefox y WebKit (Safari). **Selenium** también soporta muchos navegadores (incluyendo Internet Explorer o Safari en macOS) a través de drivers específicos, pero la configuración puede ser más compleja y algunos navegadores legacy no son relevantes en aplicaciones web modernas. **Cypress**, por otro lado, históricamente se ha enfocado en Chromium; si bien en versiones recientes añadió soporte experimental para Firefox, **no soporta Safari/WebKit** oficialmente. Playwright permite *ejecutar la misma prueba en Chrome, Firefox y Safari* de forma consistente, lo cual es ideal para asegurar

compatibilidad cross-browser. Además, Playwright maneja internamente la gestión de esos navegadores (descargándolos en la instalación), evitando al usuario tener que instalar drivers o dependencias adicionales por separado.

- **Soporte de múltiples lenguajes de programación:** Playwright hereda de Selenium la filosofía de ser multi-lenguaje. **Selenium** cuenta con bindings en Java, Python, C#, Ruby, JavaScript, entre otros (es muy versátil en este aspecto). **Cypress**, en cambio, está restringido al ecosistema JavaScript/TypeScript; solo se puede escribir pruebas en JS/TS (lo cual es adecuado si el equipo es puramente front-end, pero limita su adopción en equipos que usan otros lenguajes). **Playwright** admite JavaScript/TypeScript, Python, C# y Java, cubriendo así los lenguajes más comunes en desarrollo y pruebas. Esto significa que equipos de QA pueden integrar Playwright sin importar si sus skills son en Python o en otro lenguaje, y también facilita migrar proyectos de Selenium a Playwright (pues es posible reusar conocimientos de lenguaje). Aunque Selenium aún soporta más lenguajes en total, Playwright cubre los más populares manteniendo una API consistente en todos ellos.
- **Sincronización automática y estabilidad:** Un problema clásico con Selenium es la **sincronización**: las pruebas pueden fallar por intentar interactuar con elementos que aún no aparecen o no están listos, a menos que se agreguen esperas explícitas (p. ej., `WebDriverWait`) o se configuren esperas implícitas. Playwright aborda este desafío con *esperas automáticas integradas*. Esto significa que operaciones como `page.click()` o `page.fill()` en Playwright **esperan automáticamente** a que el elemento objetivo esté disponible en el DOM y sea interactuable antes de ejecutar la acción. También espera a que las navegaciones de página se completen en ciertos comandos de forma inteligente. El resultado es que las pruebas en Playwright tienden a ser **menos frágiles** (*flaky*) que en Selenium, ya que no requieren agregar muchas pausas o comprobaciones manuales de estado. **Cypress** comparte una filosofía similar de auto-sincronización: sus comandos `cy.get()` reintentan hasta encontrar el elemento, etc. La ventaja de Playwright sobre Cypress en este aspecto es que aplica estas esperas en *todos los lenguajes* y de forma más flexible, mientras que Cypress lo limita al entorno JavaScript. En resumen, Playwright ofrece **mayor estabilidad** en las pruebas con menos esfuerzo del tester.
- **Facilidad de uso y configuración:** Configurar y comenzar a usar Playwright resulta relativamente sencillo, especialmente en comparación con Selenium. Con Selenium, el usuario debe asegurarse de tener instalados los **drivers** correctos para cada navegador (ChromeDriver, geckodriver, etc.) o utilizar Selenium Grid; esto a veces complica el arranque para principiantes. Playwright simplifica este proceso: con un solo comando se instalan las dependencias y navegadores necesarios, y el código para lanzar un navegador es mínimo. Por ejemplo, iniciar un browser en Selenium requiere especificar el driver; en Playwright es solo `browser = p.chromium.launch()` (sin preocuparse de rutas de drivers). **Cypress** también es sencillo de iniciar (una sola dependencia npm y un comando `cypress open`), pero requiere conocimiento del ecosistema Node.js. Para alguien en Python, Playwright ofrece una **experiencia nativa** más cómoda. Además, la API de Playwright es **intuitiva y consistente**: métodos claros como `page.goto()`, `page.click()`, `page.fill()` hacen que escribir pruebas sea directo. Muchos testers encuentran que el código de Playwright es más legible y limpio que el equivalente en Selenium (que suele necesitar más código boilerplate). En cuanto a herramientas de apoyo, Playwright proporciona, como mencionamos,

utilidades integradas (codegen, inspector) que agilizan la creación de pruebas. Cypress destaca por su interfaz gráfica interactiva donde se ven los pasos de la prueba en tiempo real, algo que Playwright no tiene de la misma forma en Python (aunque sí en su versión de Node con "UI mode"). No obstante, Playwright contrarresta con la capacidad de generar scripts automáticamente y con un detallado registro (tracing) que se puede inspeccionar después de la ejecución. En síntesis, para un QA autodidacta, **la curva de aprendizaje de Playwright es muy amigable**, combinando lo mejor de la simpleza de Cypress con la flexibilidad de Selenium.

- **Escalabilidad y paralelismo:** A medida que los proyectos crecen, es importante que las herramientas de automatización soporten bien la ejecución de **grandes suites de pruebas**. Selenium ha sido utilizado en entornos enormes, pero para ejecutar pruebas en paralelo típicamente se configura Selenium Grid o se manejan múltiples hilos/procesos manualmente. Playwright fue concebido con el paralelismo en mente: es sencillo crear múltiples contextos o ventanas de navegador para distribuir pruebas concurrentemente. De hecho, en su versión para Node.js, Playwright incluye un test runner que ejecuta casos en paralelo por defecto. En Python, podemos igualmente lanzar múltiples instancias de navegador en threads o usar frameworks como PyTest xdist. **Cypress**, por su diseño, ejecuta una instancia de navegador por proceso de test, y si bien permite paralelizar en múltiples máquinas (especialmente con su Dashboard de pago), en local está algo más limitado. La ventaja de Playwright es que **escala eficazmente** en entornos de CI sin mucho esfuerzo adicional, permitiendo reducir el tiempo total de ejecución de la suite de pruebas. Esto es crucial en pipelines de integración continua donde se busca feedback rápido.

En conclusión, Playwright ofrece numerosas ventajas que lo convierten en una opción muy atractiva para la automatización de pruebas web. **Combina la amplia compatibilidad de Selenium (múltiples navegadores y lenguajes) con la simplicidad y solidez de Cypress (esperas automáticas, instalación sencilla)**, agregando además mejoras de rendimiento y herramientas modernas. Cada framework tiene sus méritos: Selenium sigue siendo el más maduro y con mayor comunidad, Cypress brinda una experiencia integrada muy cómoda para pruebas de frontend; pero Playwright logra un equilibrio que lo posiciona como una solución de **próxima generación para QA**.

Ejemplo comparativo de código en Playwright, Selenium y Cypress

Para ilustrar algunas diferencias en la práctica, veamos un **ejemplo sencillo** de cómo sería un script de prueba equivalente en Playwright, Selenium y Cypress. El escenario será simular una búsqueda de la frase "Hola Mundo" en un campo de texto y hacer clic en un botón de búsqueda en un sitio de ejemplo. Observe cómo se ve el código en cada caso:

Playwright (Python): *Script de prueba usando la API síncrona de Playwright*. Notemos que no es necesario configurar drivers y que las acciones son métodos del objeto `page`._

```python
from playwright.sync_api import sync_playwright
with sync_playwright() as p:
    browser = p.chromium.launch()            # Lanzar navegador Chromium
    page = browser.new_page()                # Abrir una nueva pestaña/página
    page.goto("https://example.com")         # Navegar a la URL objetivo
    page.fill('input[name="q"]', "Hola Mundo")  # Escribir "Hola Mundo" en el campo de texto (name="q")
    page.click("#search_button")             # Hacer clic en el botón con id "search_button"
    # (Playwright espera automáticamente a que el elemento exista antes de clicar)
    browser.close()                          # Cerrar el navegador
```

Selenium (Python): *Automatización equivalente usando Selenium WebDriver*. Requiere la configuración de un driver (en este ejemplo ChromeDriver debe estar instalado o en el PATH). Las interacciones se realizan mediante métodos del `driver` y se identifican elementos con selectores proporcionados por la clase `By`._

```python
from selenium import webdriverfrom selenium.webdriver.common.by import By

driver = webdriver.Chrome()                  # Iniciar navegador Chrome con su driver
driver.get("https://example.com")            # Navegar a la URL objetivo
campo = driver.find_element(By.NAME, "q")    # Encontrar el campo de texto por su nombre (name="q")
campo.send_keys("Hola Mundo")                # Escribir "Hola Mundo" en el campo de texto
boton = driver.find_element(By.ID, "search_button")  # Encontrar el botón por id
boton.click()                                # Hacer clic en el botón de búsqueda# (En Selenium, es posible
que se requieran esperas explícitas si el elemento tarda en aparecer)
driver.quit()                                # Cerrar el navegador y terminar la sesión WebDriver
```

Cypress (JavaScript): *Prueba escrita en Cypress. Este código típicamente iría dentro de un archivo de especificaciones* `.spec.js` *y se ejecutaría con el runner de Cypress. Cypress maneja la sincronización de comandos automáticamente y no requiere cerrar el navegador manualmente (lo gestiona internamente).*

```javascript
// Cypress - prueba de ejemplo (JavaScript/TypeScript)it('Busca la frase Hola Mundo', () => {
  cy.visit('https://example.com')              // Navegar a la URL objetivo
  cy.get('input[name="q"]').type('Hola Mundo') // Seleccionar el campo por nombre y tipear "Hola Mundo"
  cy.get('#search_button').click()             // Seleccionar el botón por id y hacer clic
  // Cypress automáticamente espera a que los elementos aparezcan y maneja las promesas internamente
})
```

En los snippets anteriores, podemos observar algunas diferencias claras:

- En Playwright y Selenium usamos **Python**, mientras que Cypress utiliza **JavaScript** (ya que solo funciona en ese entorno).
- Playwright inicia el navegador mediante su propia instancia (`p.chromium.launch()`), sin necesidad de descargar o especificar drivers manualmente. Selenium necesita que el **driver** del navegador esté disponible (en este ejemplo, ChromeDriver). Cypress tampoco requiere drivers externos, pero corre dentro de su propio entorno controlado.

- La sintaxis de Playwright y Cypress para seleccionar e interactuar con elementos es concisa (selectores CSS o atributos sencillos). Selenium es algo más **verboso** (hay que llamar a `find_element` con un tipo de selector `By.X` y luego realizar la acción).
- Playwright y Cypress **esperan automáticamente** a que la página y los elementos estén listos; en Selenium, a menudo tendríamos que agregar esperas o comprobaciones si la carga es dinámica.

A pesar de estas diferencias, los tres frameworks cumplen el mismo objetivo: automatizar la interacción con la página web. Playwright se destaca por combinar la simplicidad de las acciones (similar a Cypress) con la flexibilidad de ejecución (similar a Selenium, pero más moderna). En las siguientes secciones del libro profundizaremos mucho más en cómo escribir pruebas efectivas con Playwright, pero antes, asegurémonos de tener todo listo para empezar a usarlo.

Instalación y configuración inicial de Playwright (Python)

Ahora que conocemos las bases y ventajas de Playwright, pasemos a la **instalación y configuración inicial** para poder comenzar a crear nuestras propias pruebas. En este libro nos centraremos en Playwright para Python, por lo que los pasos a continuación asumen que trabajarás con este lenguaje. Se describirá el proceso tanto para **Windows** como para **Linux** (la mayoría de comandos también aplican a macOS, dado que es similar a Linux en su terminal).

Requisitos previos: Debes tener **Python 3** instalado en tu sistema. Puedes verificar esto abriendo una terminal y ejecutando `python --version` (o `python3 --version` en algunas distribuciones). Además, se asume familiaridad básica con la línea de comandos de tu sistema (Command Prompt/PowerShell en Windows, o la shell Bash en Linux).

Entorno virtual: Es altamente recomendable crear un **entorno virtual** de Python para tu proyecto de Playwright. Un entorno virtual (*virtual environment* o "venv") es un aislamiento donde podrás instalar Playwright y sus dependencias sin afectar a otras instalaciones de Python en tu máquina. Esto ayuda a evitar conflictos de versiones y mantiene organizado el proyecto.

A continuación, se describen los pasos para instalar Playwright en Python y preparar un proyecto básico:

1. **Crear un entorno virtual de Python** (opcional pero recomendado):
   En Windows, abre la consola (CMD o PowerShell) en la carpeta de tu proyecto y ejecuta:

   ```
   python -m venv venv
   ```

   En Linux/Mac, abre una terminal y ejecuta:

```
python3 -m venv venv
```

Esto creará una carpeta `venv` con una copia aislada de Python donde instalaremos Playwright. Puedes reemplazar `venv` por otro nombre de carpeta si lo deseas.

2. **Activar el entorno virtual:**

- En **Windows:** Ejecuta el script de activación:

```
venv\Scripts\activate
```

Verás que el prompt de la consola ahora comienza con (venv), indicando que el entorno virtual está activo.

- En Linux/Mac: Activa el entorno con el comando:

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

Del mismo modo, el prompt mostrará `(venv)` al inicio. A partir de ahora, cualquier paquete que instales con pip se ubicará dentro de este entorno virtual.

1. **Instalar la librería de Playwright:** Con el entorno virtual activado, instala Playwright mediante **pip** (el gestor de paquetes de Python). Ejecuta:

```
pip install playwright
```

Esto descargará e instalará la biblioteca `playwright` en tu entorno. (Nota: Si lo deseas, también puedes instalar opcionalmente `pytest` u otros frameworks de testing en este punto, pero para la configuración inicial no es estrictamente necesario).

2. **Instalar los navegadores soportados:** La biblioteca de Playwright instalada en el paso anterior incluye la API, pero los **ejecutables de los navegadores** (Chromium, Firefox, WebKit) se instalan por separado. Afortunadamente, Playwright proporciona un comando para descargar automáticamente las versiones adecuadas de estos navegadores. Ejecuta:

```
playwright install
```

Esto descargará los navegadores necesarios (puede tardar unos minutos la primera vez). Verás en la salida de la consola qué componentes se descargan. Al finalizar, Playwright quedará listo con todo lo necesario para empezar a ejecutar pruebas en Chrome/Edge, Firefox y WebKit. *(Si en tu entorno no tienes acceso a Internet, asegúrate de hacerlo en uno que sí, o descarga los paquetes manualmente según la documentación oficial.)*

3. **Estructura básica de proyecto y primer script de prueba:** Con Playwright instalado, ya podemos escribir nuestro primer test. Lo ideal es organizar los archivos del proyecto de forma clara. Por ejemplo, podrías tener una estructura inicial así:

```
MiProyectoPlaywright/
├── venv/                (entorno virtual de Python)
├── tests/               (directorio para archivos de prueba)
│   └── test_navegacion.py  (archivo de prueba de ejemplo)
└── README.md            (documentación del proyecto, opcional)
```

En la carpeta `tests` colocaremos nuestros scripts de pruebas. Puedes nombrar los archivos con prefijo `test_` para reconocerlos fácilmente (especialmente útil si usas frameworks como PyTest, que detecta automáticamente archivos y funciones que comienzan con `test`).

A continuación, vamos a crear un archivo de prueba sencillo, por ejemplo `test_navegacion.py`, con el siguiente contenido:

```python
from playwright.sync_api import sync_playwright
with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)   # Lanzar navegador (Chromium) en modo visible
    page = browser.new_page()
    page.goto("https://example.com")              # Navegar a una página de ejemplo
    print("Título de la página:", page.title())   # Imprimir el título de la página en la consola
    browser.close()
```

Analicemos este script básico: primero importamos `sync_playwright` para usar la API síncrona. Usamos un bloque `with sync_playwright() as p` que inicializa Playwright y se asegura de cerrarlo correctamente al salir. Luego lanzamos un navegador Chromium (pasando `headless=False` para que la ventana del navegador sea visible; si quisiéramos ejecución en segundo plano, podríamos dejarlo en modo headless por defecto). Abrimos una nueva página con `browser.new_page()` y navegamos a "https://example.com". Después obtenemos el título de la página con `page.title()` y lo mostramos por consola. Finalmente cerramos el navegador con `browser.close()`.

Para ejecutar este script, asegúrate de estar dentro del entorno virtual (recordando activar `venv` si abriste una nueva terminal) y simplemente ejecuta:

```
(venv) $ python tests/test_navegacion.py
```

Deberías observar que se abre una ventana de navegador Chrome/Chromium, carga la página de ejemplo y luego se cierra. En la terminal se imprimirá el título de la página (por ejemplo, "Example Domain"). ¡Felicidades! Has realizado tu primera automatización con Playwright. Si todo ha funcionado hasta aquí, la instalación y configuración inicial han sido exitosas.

(En caso de problemas: verifica que Python esté correctamente instalado y que activaste el entorno virtual. Si el comando `playwright install` *falló, revisa tu conexión a internet. Para cualquier error, la documentación oficial de Playwright y la comunidad de QA en foros pueden ser de ayuda.)*

Con estos pasos, tienes preparado un entorno de Playwright en Python tanto en Windows como en Linux. A partir de aquí, puedes comenzar a escribir casos de prueba más elaborados. En capítulos posteriores nos adentraremos en las **buenas prácticas de automatización**, el uso de **aserciones** para validar resultados, y cómo aprovechar al máximo las capacidades de Playwright (como el manejo de múltiples pestañas, simulación de dispositivos móviles, interceptación de redes, etc.). Por ahora, has sentado las bases: sabes qué es Playwright, en qué se diferencia de otros frameworks, y ya lo tienes funcionando en tu equipo listo para automatizar pruebas web de forma eficaz. ¡Manos a la obra en los siguientes capítulos!

Capítulo 2:Primeros Pasos con Playwright en Python

Lanzamiento del Navegador e Interacciones Básicas

En este capítulo abordaremos los primeros pasos prácticos con Playwright. Aprenderemos cómo **lanzar un navegador** y **abrir una página web real** utilizando Python. También cubriremos las **interacciones básicas** con la página, como hacer clic en elementos, escribir texto en campos de entrada y seleccionar opciones en listas desplegables. Finalmente, veremos cómo realizar **validaciones simples con** `assert` para comprobar que la página muestra lo esperado tras nuestras interacciones. El objetivo es sentar una base sólida para automatizar la navegación web con Playwright, con un enfoque técnico-profesional y claridad pedagógica.

Lanzar un navegador y abrir una página web

Antes de interactuar con una página, necesitamos **inicializar Playwright** y lanzar un navegador. Playwright soporta varios navegadores (Chromium, Firefox y WebKit). En Python, la forma más sencilla de iniciar es usar un contexto síncrono proporcionado por `sync_playwright`. A continuación veremos un script básico que abre un navegador Chromium y navega a la página principal de Wikipedia en español:

```python
from playwright.sync_api import sync_playwright
# Iniciar Playwright de forma síncronawith sync_playwright() as p:
    # Lanzar una instancia de navegador (Chromium en modo no-cabeza para ver la ventana)
    browser = p.chromium.launch(headless=False)
    # Abrir una nueva pestaña o página en el navegador
    page = browser.new_page()
    # Navegar a la URL deseada (Wikipedia en este caso)
    page.goto("https://es.wikipedia.org")
    # Opcionalmente, podemos obtener el título de la página y mostrarlo
    print(page.title())
    # Al salir del bloque with, el navegador se cerrará automáticamente
```

En el código anterior podemos identificar los siguientes pasos clave:

1. **Inicialización de Playwright:** La instrucción `with sync_playwright() as p:` inicia el contexto de Playwright. Dentro de este bloque, podemos acceder a los navegadores disponibles mediante `p`. Al salir del bloque `with`, Playwright cerrará automáticamente cualquier navegador abierto, garantizando una limpieza adecuada de recursos.

2. **Lanzamiento del navegador:** Usamos `p.chromium.launch()` para lanzar una instancia del navegador Chromium. Hemos pasado `headless=False` para abrir el navegador en modo visible (con interfaz gráfica). Si quisiéramos ejecutar en segundo plano (sin interfaz), habríamos dejado `headless=True` (valor por defecto). Playwright también permite lanzar `p.firefox.launch()` o `p.webkit.launch()` de forma similar, dependiendo del motor de navegador que necesitemos.
3. **Nueva página o pestaña:** Con `browser.new_page()` creamos una nueva página (similar a abrir una pestaña nueva en un navegador manualmente). El objeto retornado (`page`) representa esta pestaña, sobre la cual realizaremos las acciones de navegación e interacción.
4. **Navegación a una URL:** El método `page.goto("https://es.wikipedia.org")` carga la página web de la URL especificada. En este caso, navega a la portada de Wikipedia en español. Este método es equivalente a escribir una dirección en la barra de direcciones del navegador y presionar Enter. Playwright esperará automáticamente hasta que la página termine de cargarse antes de proseguir con el siguiente comando.
5. **Recuperar información de la página:** Opcionalmente, podemos usar métodos como `page.title()` para obtener el título de la página actual. En el ejemplo, `print(page.title())` imprimirá el título de la página de Wikipedia (por ejemplo, debería mostrar algo como *"Wikipedia, la enciclopedia libre"*). Esto nos confirma que la navegación fue exitosa.

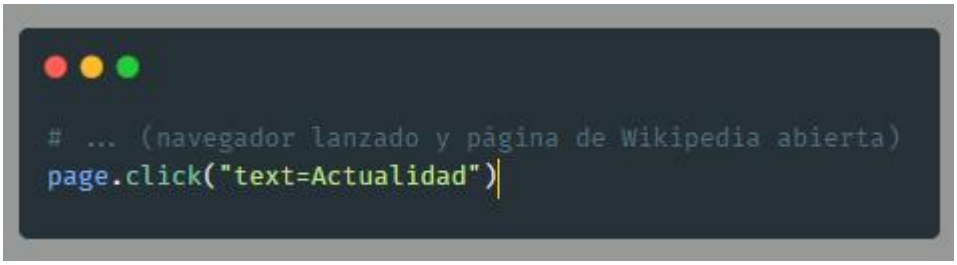
Nota: No es obligatorio cerrar el navegador manualmente cuando usamos el contexto with sync_playwright(). Al salir del bloque, Playwright cerrará el navegador automáticamente. Sin embargo, si no utilizáramos un contexto (with), sería importante llamar a browser.close() al final de nuestras operaciones para liberar los recursos.

Llegados a este punto, hemos logrado abrir un navegador y cargar una página web real. A continuación, aprenderemos a interactuar con los elementos de esa página.

Hacer clic en elementos

Una vez que tenemos una página cargada, querríamos interactuar con ella. Una interacción fundamental es **hacer clic** en enlaces, botones u otros elementos clicables. Con Playwright, podemos simular clics usando el método `page.click(selector)` proporcionando un selector que identifique el elemento objetivo.

Por ejemplo, supongamos que en la página de Wikipedia queremos hacer clic en el enlace "Actualidad" (que lleva al portal de noticias actuales). Podemos seleccionar este enlace por su texto visible y hacer clic de la siguiente manera:



En esta llamada, `"text=Actualidad"` **es un selector** proporcionado por Playwright que busca un elemento cuyo texto visible sea "Actualidad". Playwright hará lo siguiente internamente: buscar el enlace (u otro elemento) con ese texto y simular un evento de clic sobre él, igual que si un usuario real pulsara el botón izquierdo del ratón.

Al ejecutar `page.click("text=Actualidad")` en la portada de Wikipedia, el navegador navegará a la página *"Portal:Actualidad"*. Playwright esperará automáticamente a que la nueva página termine de cargar antes de continuar. Después de este clic, podríamos, por ejemplo, verificar que la URL o el título de la página corresponden al portal de Actualidad (veremos las validaciones con asserts más adelante).

Seleccionar elementos para hacer clic: Además de la estrategia `text=` para buscar por texto visible, Playwright soporta múltiples tipos de selectores. Por ejemplo:

- Selectores CSS, e.g. `page.click("button.login")` haría clic en un botón con clase CSS "login".
- Selectores por id, e.g. `page.click("#miElemento")` clicaría el elemento con id="miElemento".
- Selectores de rol (ARIA roles) y otras estrategias avanzadas.

Por ahora, usar el texto o un selector CSS simple suele ser suficiente para nuestros ejemplos básicos. Es importante asegurarse de que el selector que usemos identifique de forma única al elemento deseado para evitar confusiones.

***Consejo**: Playwright espera automáticamente a que el elemento esté disponible y sea interactuable antes de efectuar el clic. En la mayoría de casos no necesitaremos insertar demoras o esperas explícitas. Si el elemento no aparece, page.click lanzará un error después de un tiempo de espera por defecto (timeout), lo que nos indica que quizá el selector es incorrecto o el elemento tarda demasiado en cargar.*

Escribir texto en campos de entrada

Otra interacción común es **ingresar texto** en campos, como barras de búsqueda, formularios de login, etc. Con Playwright, podemos simular la escritura de texto usando el método `page.fill(selector, texto)`. Este método localiza el campo (por ejemplo un `<input>` o `<textarea>`) y rellena su contenido con el texto proporcionado, reemplazando cualquier contenido previo.

Siguiendo con el ejemplo de Wikipedia, podríamos usar la barra de búsqueda para encontrar un artículo. El campo de búsqueda en la página principal de Wikipedia tiene el identificador `searchInput`. Veamos cómo ingresar un término de búsqueda (por ejemplo, "Python") y luego **simular presionar la tecla Enter** para ejecutar la búsqueda:

En este fragmento:

- `page.fill("input#searchInput", "Python")` encuentra el campo de texto cuyo atributo `id` es "searchInput" (el cuadro de búsqueda) y escribe el texto "Python" en él. Si el campo ya tuviera algún texto, `fill` lo sobrescribirá completamente. Playwright se asegura de que el elemento esté listo (visible y habilitado) antes de escribir, por lo que no necesitamos esperas manuales. Tras esta operación, el campo de búsqueda de Wikipedia mostrará la palabra "Python".
- `page.press("input#searchInput", "Enter")` simula la pulsación de la tecla Enter mientras el foco (focus) está en el campo de búsqueda. Equivale a escribir "Python" y luego pulsar Enter manualmente. Esta acción envía el formulario de búsqueda de Wikipedia. Como resultado, el navegador navegará a la página de resultados (o al artículo directamente si el término coincide exactamente con una entrada). Playwright nuevamente esperará a que la navegación resultante cargue completamente.

***Nota**: Alternativamente, podríamos haber hecho clic en el botón de búsqueda de Wikipedia (generalmente con id="searchButton"). Ambas acciones (presionar Enter en el campo o hacer clic en el botón de búsqueda) logran el mismo resultado de iniciar la búsqueda. Playwright ofrece flexibilidad para elegir la forma de interacción que mejor se adapte al caso.*

También existe un método para simular mecanografía tecla por tecla (similar a cómo un usuario escribiría). En la versión Python de Playwright, la manera recomendada de ingresar texto es `fill`, ya que inserta la cadena completa de manera rápida. Solo en situaciones donde necesitemos probar un comportamiento específico de teclado (por ejemplo, validaciones en tiempo real mientras se escribe) podríamos recurrir a enviar eventos de teclado individuales. Para la mayoría de tests, `page.fill` es más conveniente y rápido.

Seleccionar opciones en listas desplegables

Muchas páginas incluyen **listas desplegables** (`<select>` en HTML) para que el usuario elija entre varias opciones predefinidas (por ejemplo, seleccionar un país, una

provincia, etc.). Playwright facilita la interacción con estos elementos mediante el método `page.select_option(selector, option)`. Podemos seleccionar una opción especificando el valor, la etiqueta visible o el índice de la opción que deseamos.

Imaginemos un formulario con un menú desplegable de país, cuyo elemento `<select>` tiene `id="pais"` y contiene opciones como "España", "México", "Argentina", etc. Supongamos que queremos automatizar la selección de "España". Tenemos un par de formas de hacerlo:

```python
# Seleccionar por la etiqueta visible de la opción
page.select_option("select#pais", label="España")
# (Alternativa) Seleccionar por el valor de la opción
page.select_option("select#pais", value="ES")
```

En el primer caso, Playwright buscará dentro del `<select id="pais">` una `<option>` cuyo texto visible sea exactamente "España" y la seleccionará. En el segundo caso, buscará una `<option>` con atributo `value="ES"` (imaginando que el código de país para España es "ES") y la seleccionará.

Playwright se encarga de desplegar la lista y elegir la opción indicada, igual que lo haría un usuario. Después de `select_option`, el elemento `<select>` quedará con la opción elegida marcada.

Otras formas de identificar opciones: Además de `label` y `value`, también es posible seleccionar por índice (posición) usando, por ejemplo, `page.select_option("select#pais", index=0)` para la primera opción de la lista. Sin embargo, apoyarse en índices puede ser menos fiable si el contenido de la lista cambia. En general, es recomendable seleccionar por `label` (lo que el usuario ve) o por `value` (un identificador interno definido en el HTML).

***Nota**: page.select_option acepta también una lista si necesitamos seleccionar múltiples opciones en un <select multiple>. Para un <select> normal de única selección, proveer un solo valor/label es suficiente. Si la opción a seleccionar no existe, Playwright lanzará un error, lo cual ayuda a detectar situaciones donde la página no cargó correctamente las opciones o el selector no es correcto.*

Validar elementos con asserts

Después de realizar interacciones, es crucial **verificar que la página refleja el resultado esperado**. En las pruebas automatizadas, esto se realiza mediante *aserciones* o *asserts*. En Python, podemos usar la sentencia `assert` para comprobar condiciones lógicas. Si la condición es `True`, la ejecución continúa; si es `False`, se lanza una excepción AssertionError indicando que la prueba no pasó.

Veamos cómo podríamos validar algunas condiciones tras nuestras interacciones en Wikipedia. Continuando el ejemplo de la búsqueda de "Python", podríamos querer comprobar que:

- El campo de búsqueda contiene el texto ingresado ("Python").
- La página navegó a un resultado cuyo título tiene la palabra "Python".
- El encabezado `<h1>` de la nueva página (título del artículo o resultados) contiene "Python".

Podemos implementar esas verificaciones así:

```python
# ... (después de realizar la búsqueda de "Python" en Wikipedia)
# Verificar que el campo de búsqueda contiene el texto ingresado
assert page.get_attribute("input#searchInput", "value") == "Python"

# Verificar que el título de la página contiene la palabra "Python"
assert "Python" in page.title()

# Verificar que el encabezado principal de la página (h1) contiene "Python"
assert "Python" in page.inner_text("h1")
```

Explicación de estas aserciones:

- **Valor de un campo (`get_attribute`)**: Usamos `page.get_attribute("input#searchInput", "value")` para obtener el atributo `value` del campo de búsqueda. Esto devuelve el texto actualmente presente en el campo. La aserción comprueba que dicho valor es exactamente "Python", asegurando que nuestro comando `page.fill` funcionó correctamente. Esta verificación es útil en casos donde la página puede modificar o limpiar el campo tras alguna acción, pero en nuestro escenario confirma simplemente que el texto se ingresó.
- **Título de la página (`page.title()`)**: `page.title()` retorna el título actual de la pestaña (lo que usualmente aparece en la barra del navegador). Después de buscar "Python", esperamos que el título de la nueva página tenga la palabra "Python". Por ejemplo, si la búsqueda llevó al artículo "Python (lenguaje de programación)", el título podría ser algo como *"Python (lenguaje de programación) - Wikipedia"*, que ciertamente contiene la palabra "Python". La aserción `assert "Python" in page.title()` verifica este caso de manera flexible (no exigimos igualdad exacta, solo que contenga el término).
- **Texto de un elemento (`inner_text`)**: `page.inner_text("h1")` obtiene el texto visible dentro del primer elemento `<h1>` que encuentre en la página (generalmente el título del artículo o encabezado principal). Tras la búsqueda, es razonable que el encabezado de la página de resultados o del artículo incluya "Python". La aserción comprueba que sea así. Usamos `inner_text` en lugar de `text_content` para obtener el texto "como lo ve el usuario" (sin espacios adicionales ni contenido oculto).

Estas aserciones, si todas pasan sin error, nos dan confianza de que la interacción produjo el efecto correcto en la página. En caso de que alguna falle, significa que algo no salió como se esperaba: o bien la página no navegó al contenido correcto, o el elemento buscado no muestra el texto esperado, etc. Esto ayuda a detectar problemas en nuestra automatización o en la propia aplicación web.

***Otras validaciones comunes**: Además de texto y títulos, podemos validar atributos específicos (como hicimos con el valor del campo). Por ejemplo, podríamos verificar que un enlace tiene el URL correcto: **assert page.get_attribute("a#miEnlace", "href") == "https://destino.esperado.com"**. También es útil comprobar la visibilidad o existencia de elementos, para lo cual Playwright ofrece métodos como **page.is_visible(selector)** (devuelve True/False si un elemento está visible en la página). En nuestras pruebas simples podemos emplear **assert page.is_visible(selector)** para afirmar que cierto elemento se muestra tras una interacción.*

En resumen, el uso de `assert` integrado en Python nos permite construir **pruebas verificables**. Cada interacción importante en la página debería ir acompañada de alguna comprobación que confirme que la página reaccionó adecuadamente. Esto convierte nuestros scripts en pruebas automatizadas confiables en lugar de simples secuencias de acciones.

Ejercicios prácticos

Para reforzar los conceptos aprendidos, a continuación se proponen algunos ejercicios prácticos. Intente resolverlos escribiendo scripts con Playwright en Python, utilizando las funciones presentadas en este capítulo:

1. **Abrir una página y verificar el título:** Utilizando Playwright, abre la página principal de Wikipedia en español. Una vez cargada, comprueba con un `assert` que el título de la pestaña (`page.title()`) contiene la palabra "Wikipedia".
2. **Búsqueda en Wikipedia:** Escribe un script que navegue a `https://es.wikipedia.org`, ingrese un término de búsqueda (el que tú prefieras, por ejemplo "Playwright") en el campo de búsqueda y ejecute la búsqueda (puedes presionar Enter o hacer clic en el botón de búsqueda). Después, verifica que en la página resultante aparezca el término buscado, ya sea en el título de la página o en el encabezado `<h1>` del artículo/resultados.
3. **Navegación mediante clic:** Desde la portada de Wikipedia, automatiza un clic en algún enlace de la página (por ejemplo, "Portales" o "Actualidad" u otro enlace visible). Utiliza una aserción para comprobar que la navegación ocurrió correctamente, verificando por ejemplo que `page.url` o `page.title()` correspondan a la nueva página esperada tras el clic.
4. **Interacción con una lista desplegable:** Si tienes acceso a una página de ejemplo con una lista desplegable (por ejemplo, un formulario de registro con campo de país, o una página de pruebas), escribe un script que seleccione una opción específica de ese `<select>`. Usa `page.select_option` para elegir la opción por texto visible o valor. Luego, comprueba que la selección se realizó correctamente. Por ejemplo, podrías obtener el valor seleccionado con JavaScript (usando

`page.evaluate` para leer `element.value`) o verificar algún cambio en la página que ocurra al seleccionar esa opción.

Cada ejercicio busca poner en práctica una o más de las habilidades presentadas: abrir páginas, hacer clic, escribir texto, seleccionar opciones y usar asserts para validarlo todo. ¡Manos a la obra! Al completar estos ejercicios, afianzarás tus conocimientos básicos de Playwright antes de avanzar a temas más complejos. Buena suerte y buena exploración.