«Talento Tech»

Automation Testing

Clase 10

Buenos Aires Ciudad

Clase N° 10: Manejo de Datos de Prueba

Temario

- ¿Por qué necesitamos separar los datos del código de prueba?
- Parametrización de pruebas con Pytest
- Lectura de datos desde archivos (CSV, JSON)
- Data-driven testing: separar datos de lógica
- Generación de datos aleatorios con Faker
- Ejercicio práctico TalentoLab: login y carrito parametrizados

Objetivos de la clase

En esta clase abordaremos el manejo de datos de prueba, un componente clave para construir tests robustos, flexibles y representativos de situaciones reales. Comenzaremos por entender por qué es una buena práctica separar los datos del código de automatización, y cómo esto mejora la mantenibilidad y reutilización de nuestras pruebas. A continuación, aprenderemos a ejecutar un mismo test con múltiples combinaciones de datos usando `@pytest.mark.parametrize`, lo que nos permitirá cubrir más escenarios sin duplicar código. También exploraremos cómo cargar datos desde archivos CSV y JSON, y cómo utilizar la biblioteca Faker para generar datos aleatorios pero realistas de forma automática. Todos estos conceptos los aplicaremos en ejercicios prácticos sobre SauceDemo, reforzando su uso en contextos concretos y funcionales.

¿Por qué separar datos del código?

En la clase anterior trabajamos en la **organización y robustez de nuestro código de automatización** aplicando el patrón Page Object Model (POM). Aprendimos cómo encapsular los selectores y acciones de cada página en clases específicas, logrando así un framework más mantenible y escalable. Esta estructura nos permitió separar la lógica de interacción con la interfaz (por ejemplo, el login o el manejo del carrito) del resto de los tests, haciendo que cualquier cambio en la UI pueda gestionarse desde un solo lugar.

Sin embargo, aunque logramos desacoplar los detalles técnicos de la interfaz, **nuestros datos de prueba** (como usuarios, contraseñas y productos) **siguen "hardcodeados" en el código de los tests**. Esto trae consigo varios problemas: cada vez que queremos agregar nuevos escenarios, debemos duplicar funciones o modificar código fuente, lo que aumenta el riesgo de errores, dificulta la colaboración con personas no técnicas y hace que el mantenimiento sea poco eficiente.

Aquí es donde damos el siguiente paso profesional:
Hoy vamos a aprender por qué y cómo **separar los datos del código**. Este enfoque es fundamental en cualquier framework de testing moderno porque permite:

- Automatizar decenas o cientos de escenarios sin repetir código.
- Hacer que el código de los tests sea más limpio, genérico y fácil de entender.
- Delegar la actualización de los datos a cualquier persona del equipo (no solo programadores).
- Preparar la base para estrategias avanzadas como el data-driven testing y la integración con pipelines de CI/CD.

Vamos a ver cómo este pequeño cambio de mentalidad nos acerca a las buenas prácticas de la industria, mejora la calidad de nuestras pruebas y simplifica el crecimiento de nuestros proyectos.

Hasta ahora, hemos estado escribiendo nuestros datos directamente en el código:

❌ Datos "hardcodeados" en el código

```
def test_login():
    login_page.completar_usuario("standard_user")
    login_page.completar_clave("secret_sauce")
```

¿Qué problemas tiene esto?

- Si queremos probar con 10 usuarios diferentes, necesitamos 10 funciones casi idénticas
- Si cambian las credenciales, tenemos que buscar y cambiar esas credenciales en todo el código
- Es difícil que personas no técnicas (como Product Owners) revisen o modifiquen los datos en varias partes del código.

La solución: separar datos de lógica

✅ Datos separados

```
usuarios_prueba = [
    ("standard_user", "secret_sauce"),
    ("locked_out_user", "secret_sauce"),
    ("problem_user", "secret_sauce")
]

def test_login(usuario, clave):
        login_page.completar_usuario(usuario)
        login_page.completar_clave(clave)
```

De esta manera solo necesitarás cambiar las credenciales solamente en ese punto.

Parametrización con Pytest: Un test, muchos datos

¿Qué es la parametrización?

Es una técnica que permite ejecutar el mismo test múltiples veces, cada vez con datos diferentes. En lugar de escribir 3 tests separados, escribimos 1 test y le decimos a Pytest que lo ejecute con 3 conjuntos de datos.

Ejemplo básico paso a paso

Paso 1: Sin parametrizar (❌ repetitivo)

```python
def test_login_standard_user(driver):
    login = LoginPage(driver)
    login.abrir()
    login.completar_usuario("standard_user")
    login.completar_clave("secret_sauce")
    login.enviar()
    assert "inventory.html" in driver.current_url

def test_login_locked_user(driver):
    login = LoginPage(driver)
    login.abrir()
    login.completar_usuario("locked_out_user")
    login.completar_clave("secret_sauce")
    login.enviar()
    assert login.hay_error()  # Este usuario está bloqueado

def test_login_invalid_user(driver):
    login = LoginPage(driver)
    login.abrir()
    login.completar_usuario("usuario_inexistente")
    login.completar_clave("password_malo")
    login.enviar()
    assert login.hay_error()
```

Paso 2: Con parametrización (✅ eficiente)

```
import pytest
from pages.login_page import LoginPage
# Definimos nuestros casos de prueba
CASOS_LOGIN = [
    ("standard_user", "secret_sauce", True),    # usuario válido, login exitoso
    ("locked_out_user", "secret_sauce", False), # usuario bloqueado, login falla
    ("usuario_malo", "password_malo", False),      # credenciales inválidas, login falla
]
```

```
@pytest.mark.parametrize("usuario, clave, debe_funcionar", CASOS_LOGIN)
def test_login_parametrizado(driver, usuario, clave, debe_funcionar):
    login = LoginPage(driver)
    login.abrir()
    login.completar_usuario(usuario)
    login.completar_clave(clave)
    login.enviar()

    if debe_funcionar:
        # Si debe funcionar, verificamos que llegamos al inventario
        assert "inventory.html" in driver.current_url
    else:
        # Si no debe funcionar, verificamos que hay un error
        assert login.hay_error()
```

¿Qué cambios realizamos y qué mejoras obtenemos?

Cambios:

- **Eliminamos duplicación de código**: Los 3 tests separados se convirtieron en 1 solo test parametrizado
- **Centralizamos los datos**: Creamos la lista CASOS_LOGIN con todos los escenarios en un solo lugar
- **Añadimos lógica condicional:** Usamos if debe_funcionar para manejar diferentes tipos de validaciones en el mismo test
- **Aplicamos el decorador @pytest.mark.parametrize**: Esto le dice a Pytest que ejecute la función múltiples veces con diferentes datos. *(Este decorador lo vemos ahora en el siguiente punto.*)

Mejoras:

- **Mantenimiento simplificado:** Si necesitas cambiar la lógica de login, solo modificas una función en lugar de tres
- **Fácil escalabilidad:** Agregar un nuevo escenario de prueba solo requiere añadir una tupla a CASOS_LOGIN
- **Código más limpio:** Menos repetición = menos posibilidades de errores y mejor legibilidad
- **Reportes más organizados:** Pytest muestra cada combinación de datos como un test individual, pero con nombres descriptivos
- **Datos centralizados:** Todos los casos de prueba están en un solo lugar, facilitando la revisión y actualización
- **Reutilización:** La misma lista CASOS_LOGIN puede usarse en otros tests relacionados

¿Qué hace `@pytest.mark.parametrize`?

`@pytest.mark.parametrize` es un **decorador** de pytest que permite ejecutar la misma función de test múltiples veces, cada vez con diferentes conjuntos de datos. Es como decirle a pytest: *"Toma esta función y ejecutala varias veces, pero cada vez con datos distintos"*.

Ejemplo visual:

```python
@pytest.mark.parametrize("usuario, clave", [
    ("user1", "pass1"),
    ("user2", "pass2"),
    ("user3", "pass3")
])
def test_login(usuario, clave):
    # Esta función se ejecutará 3 veces
    # Primera vez: usuario="user1", clave="pass1"
    # Segunda vez: usuario="user2", clave="pass2"
    # Tercera vez: usuario="user3", clave="pass3"
```

En lugar de escribir 3 funciones separadas, escribís 1 función y pytest la ejecuta 3 veces automáticamente.

Ahora veamos cómo funciona:

1. **Primer parámetro**: Los nombres de las variables que recibirá la función
2. **Segundo parámetro**: Una lista de tuplas con los valores para cada ejecución
3. **Resultado**: Pytest ejecuta la función una vez por cada tupla

¿Cómo se ve en el reporte?

Cuando ejecutes `pytest -v`, verás algo así:

```
test_login_parametrizado[standard_user-secret_sauce-True] PASSED
test_login_parametrizado[locked_out_user-secret_sauce-False] PASSED
test_login_parametrizado[usuario_malo-password_malo-False] PASSED
```

¡Pytest crea automáticamente un test por cada combinación de datos!

Leyendo datos desde archivos

Hasta la clase anterior, con Page Object Model (POM), logramos separar el **cómo** interactuamos con la aplicación (selectores, métodos) del código de los tests. Pero los **datos** que usábamos para probar seguían “pegados” en el mismo código de Python (hardcodeados).

¿Qué problema hay con eso?

Cada vez que querés probar con usuarios distintos, productos nuevos o credenciales diferentes, hay que editar el código, duplicar funciones o modificar muchas líneas. Esto es poco práctico y no escala para proyectos grandes.

¿Qué cambia con “leer datos desde archivos”?

Ahora, en vez de escribir los datos de prueba dentro del código, los guardamos en archivos externos (como CSV o JSON) y el test los “lee” y usa automáticamente. Así:

- Los datos se pueden actualizar fácil, sin tocar el código Python.
- Personas no técnicas (ej: Product Owners, Marketing) pueden modificar los datos de prueba.
- Podemos agregar decenas de casos nuevos solo editando un archivo, sin copiar código.

Primero vamos trabajar con CSV

CSV significa "Comma Separated Values" (Valores Separados por Comas). Es como una tabla de Excel pero en formato texto.

¿Qué vamos a hacer con CSV?

Vamos a crear un **sistema de 3 pasos** que nos permitirá leer datos desde archivos CSV y usarlos en nuestros tests parametrizados:

El objetivo: Convertir nuestros datos "hardcodeados" en datos externos que cualquiera pueda modificar.

Los 3 pasos que seguiremos:

1. **Crear el archivo CSV** → Almacenar nuestros casos de prueba en formato tabla
2. **Crear función lectora** → Programar una función que convierta CSV a formato pytest
3. **Conectar con el test** → Usar esos datos en nuestro test parametrizado

El flujo completo será:

```
None
CSV (datos) → función lectora → pytest.mark.parametrize → test ejecutado múltiples veces
```

Resultado esperado:

- Los datos estarán en un archivo separado que cualquiera puede editar
- El mismo código de test funcionará con todos los datos del CSV
- Agregar nuevos casos será tan simple como añadir una fila al archivo

Ahora veamos cómo implementarlo paso a paso:

Paso 1: Crear el archivo de datos

Crea una carpeta `datos/` en tu proyecto y dentro un archivo `login.csv`:

```
usuario,clave,debe_funcionar
standard_user,secret_sauce,True
locked_out_user,secret_sauce,False
problem_user,secret_sauce,True
usuario_malo,password_malo,False
```

Paso 2: Crear función para leer CSV

(Todo el contenido del proyecto lo tenes en el repo: Clase10)

Crea un archivo `utils/datos.py`:

```python
import csv
import pathlib

def leer_csv_login(ruta_archivo):
    """
    Lee un archivo CSV y devuelve una lista de tuplas
    para usar en parametrización de pytest
    """
    datos = []
    ruta = pathlib.Path(ruta_archivo)
```

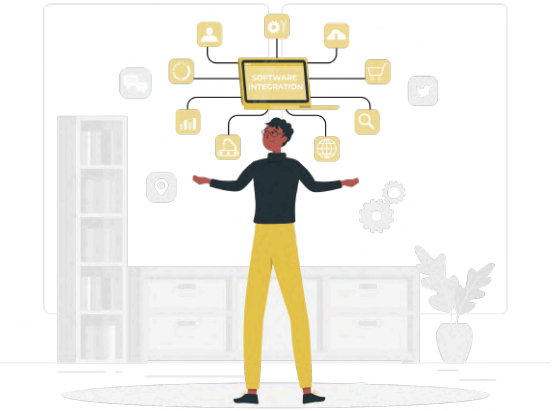
```
    with open(ruta, newline='', encoding='utf-8') as archivo:
        lector = csv.DictReader(archivo)
        for fila in lector:
            # Convertir string 'True'/'False' a booleano
            debe_funcionar = fila['debe_funcionar'].lower() == 'true'
            datos.append((fila['usuario'], fila['clave'], debe_funcionar))

    return datos

# Ejemplo de uso
if __name__ == "__main__":
    casos = leer_csv_login('datos/login.csv')
    print(casos)
    # Resultado: [('standard_user', 'secret_sauce', True), ...]
```

Paso 3: Usar en el test (login_page.py)

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

class LoginPage:
    # URL de la página de login
    URL = "https://www.saucedemo.com/"

    # Locators (selectores de elementos)
    _USER_INPUT = (By.ID, "user-name")
    _PASS_INPUT = (By.ID, "password")
    _LOGIN_BUTTON = (By.ID, "login-button")
    _ERROR_MESSAGE = (By.CSS_SELECTOR, "[data-test='error']")

    def __init__(self, driver):
        """
        Constructor que recibe la instancia del WebDriver
        """
        self.driver = driver
        self.wait = WebDriverWait(driver, 10)

    def abrir(self):
        """Navegar a la página de login"""
        self.driver.get(self.URL)
```

```
        return self

    def completar_usuario(self, usuario):
        """Escribir el nombre de usuario"""
                                                                campo          =
self.wait.until(EC.visibility_of_element_located(self._USER_INPUT))
        campo.clear()
        campo.send_keys(usuario)
        return self

    def completar_clave(self, clave):
        """Escribir la contraseña"""
        campo = self.driver.find_element(*self._PASS_INPUT)
        campo.clear()
        campo.send_keys(clave)
        return self

    def enviar(self):
        """Hacer clic en el botón de login"""
        self.driver.find_element(*self._LOGIN_BUTTON).click()
        return self

    def login_completo(self, usuario, clave):
        """Método de conveniencia para hacer login completo"""
        self.completar_usuario(usuario)
        self.completar_clave(clave)
        self.enviar()
        return self

    def hay_error(self):
        """Verificar si hay un mensaje de error visible"""
        try:

self.wait.until(EC.visibility_of_element_located(self._ERROR_MESSAGE))
            return True
        except:
            return False

    def obtener_mensaje_error(self):
        """Obtener el texto del mensaje de error"""
        if self.hay_error():
            return self.driver.find_element(*self._ERROR_MESSAGE).text
        return ""
```

¿Qué logramos con este enfoque?



Con esta implementación hemos creado un **sistema de datos desacoplado** donde:

Los datos viven separados del código: El archivo CSV puede ser editado sin tocar Python

La función `leer_csv_login()` actúa como traductor: Convierte el formato CSV al formato que Pytest entiende

El Page Object Model se mantiene intacto: LoginPage no sabe ni le importa de dónde vienen los datos

Fácil expansión: Para agregar nuevos escenarios, solo necesitas una nueva fila en el CSV

Ahora vamos a trabajar con JSON

¿Qué es JSON? JSON es un formato para almacenar datos estructurados. Es muy usado en APIs y configuraciones.

¿Qué vamos a hacer con JSON?

Ahora vamos a trabajar con **datos más complejos** usando JSON. Mientras que CSV es perfecto para datos simples (como credenciales), JSON nos permite manejar **objetos con múltiples propiedades**.

El objetivo: Probar el carrito de compras con productos que tienen nombre, precio y descripción.

Los 3 pasos que seguiremos:

1. **Crear archivo JSON** → Almacenar productos con todas sus propiedades
2. **Crear función lectora** → Extraer la información que necesitamos para los tests
3. **Conectar con el test** → Parametrizar tests de carrito con productos reales

El flujo completo será:

```
None
JSON (productos) → función lectora → pytest.mark.parametrize → test de carrito
```

¿Por qué JSON en lugar de CSV para productos?

- **CSV:** `Backpack,29.99,Una mochila` (difícil de leer, solo strings)
- **JSON:** `{"nombre": "Backpack", "precio": 29.99, "descripcion": "Una mochila"}` (estructura clara, tipos correctos)

Ventaja clave: JSON preserva los tipos de datos (números, booleanos) y permite estructuras más complejas que usaremos en tests avanzados.

Ahora veamos la implementación:

Paso 1: Crear archivo de productos

Crea `datos/productos.json`:

```
[
 {
   "nombre": "Sauce Labs Backpack",
   "precio": 29.99,
   "descripcion": "Una mochila práctica"
 },
 {
   "nombre": "Sauce Labs Bike Light",
   "precio": 9.99,
   "descripcion": "Luz para bicicleta"
 },
 {
   "nombre": "Sauce Labs Bolt T-Shirt",
   "precio": 15.99,
   "descripcion": "Camiseta con diseño único"
 }
]
```

Paso 2: Función para leer JSON

```python
import json

def leer_json_productos(ruta_archivo):
    """
    Lee un archivo JSON con información de productos
    """
    with open(ruta_archivo, 'r', encoding='utf-8') as archivo:
        productos = json.load(archivo)

    # Extraer solo los nombres para parametrización
    nombres = [producto['nombre'] for producto in productos]
    return nombres

# Ejemplo de uso
if __name__ == "__main__":
    productos = leer_json_productos('datos/productos.json')
    print(productos)
    # Resultado: ['Sauce Labs Backpack', 'Sauce Labs Bike Light', ...]
```

Paso 3: Test parametrizado con productos

Aquí tienes el ejemplo de como haríamos un test utilizando toda la información desde un json. Te dejo el link a la carpeta dentro del repo: Data from Json

Para ejecutarlo:
python3 -m pytest tests/test_carrito_json.py -s -v

¿Qué estamos haciendo con este código que incluye JSON?

- Almacenamos datos complejos: Cada producto tiene múltiples propiedades (nombre, precio, descripción) que se mantienen organizadas
- Flexibilidad de estructura: A diferencia del CSV, JSON permite agregar nuevos campos sin romper la estructura existente

- Preservación de tipos: Los números (precio) se mantienen como números, no como strings que habría que convertir
- Preparación para escalabilidad: Si mañana necesitamos agregar campos como "categoria", "stock", o "imagen", JSON lo maneja naturalmente

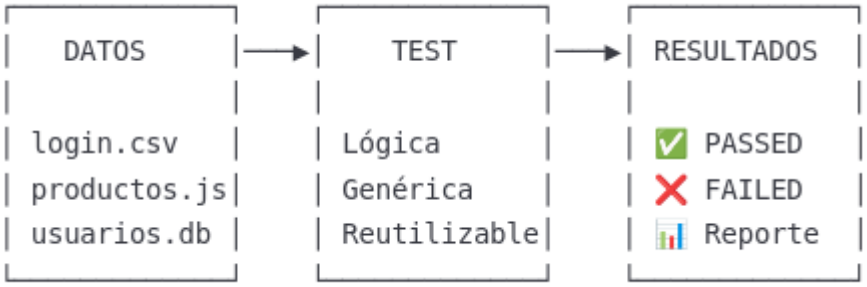
En el contexto real: Este enfoque es muy común cuando los datos de prueba provienen de APIs (que responden en JSON) o cuando necesitamos simular objetos complejos del mundo real.

Esto que hicimos anteriormente con CSV y con Json se llama Data-Driven.

Data-driven testing: El concepto

Data-driven testing significa que tus tests están "conducidos" o "dirigidos" por los datos. El test es genérico, y los datos específicos vienen de afuera.

Ejemplo visual:



Ventajas del enfoque data-driven:

1. **Mantenimiento fácil**: Cambiar datos no requiere tocar código
2. **Escalabilidad**: Agregar 100 usuarios nuevos = agregar 100 líneas al CSV
3. **Colaboración**: Product Owners pueden editar datos sin saber programar
4. **Reutilización**: Los mismos datos sirven para UI, API, y otros tipos de tests
5. **Auditoría**: Es fácil revisar qué casos se están probando

¿Cómo se implementa en la industria?

En empresas como Netflix o Spotify, los equipos de QA manejan miles de casos de prueba diarios. Por ejemplo, para probar el login con 50 tipos de usuarios en 20 países (1000 combinaciones), usan data-driven testing donde el mismo código se ejecuta automáticamente con datos almacenados en archivos externos. Cuando Spotify lanza una nueva funcionalidad, prueban con datos reales de usuarios (anonimizados) que tienen diferentes patrones de uso, géneros musicales y dispositivos. Esta técnica es tan estándar que muchas empresas tienen equipos dedicados exclusivamente a mantener estos datasets

de prueba, ya que la calidad de los datos impacta directamente en la efectividad de las pruebas automatizadas.

¿Cómo se aplicará en tu Proyecto Final?

En tu framework final, implementarás este enfoque para crear un sistema robusto y profesional. Los mismos datos que uses para probar el login en la UI también servirán para validar la autenticación por API, maximizando la cobertura sin duplicar esfuerzo. Además, cuando presentes tu proyecto, podrás demostrar cómo cambiar los datos de prueba sin tocar una línea de código, algo que impresiona mucho a los reclutadores porque refleja las prácticas reales de la industria. Esta separación de datos también facilitará la integración con CI/CD, ya que podrás tener diferentes conjuntos de datos para distintos ambientes (desarrollo, testing, producción).

Conectando Data-driven testing con Faker

Hasta ahora hemos visto **dos enfoques** para el data-driven testing:

- **Datos fijos** (CSV/JSON): Perfectos para casos específicos y reproducibles
- **Datos dinámicos** (Faker): Ideales para probar con variedad y detectar bugs inesperados

¿Cuándo usar cada enfoque?

Datos fijos (CSV/JSON) - Úsalos cuando:

- Necesitas casos específicos conocidos
- Quieres reproducir exactamente el mismo test
- Tienes escenarios de negocio definidos
- Ejemplo: *"Probar que el usuario 'admin' puede acceder al panel"*

Datos aleatorios (Faker) - Úsalos cuando:

- Quieres probar con gran variedad de datos
- Buscas descubrir bugs inesperados
- Necesitas simular usuarios reales diversos
- Ejemplo: *"Probar que cualquier email válido puede registrarse"*

🔄 **Enfoque híbrido - Lo mejor de ambos mundos:**

```python
# Datos críticos fijos + datos variables aleatorios
```

```
@pytest.mark.parametrize("usuario_tipo",    ["admin",    "user",
"guest"])

def test_registro_completo(usuario_tipo):

    # Usuario tipo viene del CSV (fijo)

    # Datos personales vienen de Faker (aleatorios)

    fake_email = fake.email()

    fake_name = fake.name()
```

Ahora veamos cómo implementar Faker.

Generación de datos aleatorios con Faker

Faker es una librería que genera datos falsos pero realistas. Es útil cuando:

- No quieres crear manualmente 1000 usuarios de prueba
- Necesitas datos únicos en cada ejecución
- Quieres probar con diferentes tipos de caracteres y formatos
- Buscas descubrir bugs con datos inesperados

Instalación:

pip install faker

Datos útiles que puede generar Faker:

```
fake = Faker()

# Información personal
fake.name()            # Nombre completo
fake.first_name()      # Nombre
fake.last_name()       # Apellido
fake.email()           # Email
fake.phone_number()    # Teléfono

# Direcciones
fake.address()         # Dirección completa
fake.city()            # Ciudad
fake.country()         # País
fake.postcode()        # Código postal

# Internet
fake.url()             # URL
fake.domain_name()     # Dominio
fake.user_name()       # Nombre de usuario
fake.password()        # Contraseña

# Números y fechas
fake.random_int(1, 100)  # Número entero aleatorio
fake.date()            # Fecha
fake.time()            # Hora

# Texto
fake.text()            # Párrafo de texto
```

```
fake.sentence()      # Oración
fake.word()          # Palabra
```

¿Por qué Faker es tan potente para testing?

Faker no solo genera datos aleatorios, sino que también puede configurarse para diferentes idiomas y regiones usando `Faker('es_ES')` para español o `Faker('en_US')` para inglés americano, lo que es crucial cuando pruebas aplicaciones internacionales. Una ventaja clave es que permite detectar bugs de encoding y caracteres especiales que datos "limpios" nunca revelarían - por ejemplo, nombres con acentos, apellidos con apostrofes, o direcciones con caracteres unicode. En testing de carga, Faker es invaluable porque puede generar millones de registros únicos sin consumir memoria excesiva, algo imposible de hacer manualmente. También incluye un sistema de "seeds" que permite reproducir exactamente los mismos datos aleatorios cuando necesitas debuggear un test que falló, usando `Faker.seed(1234)` al inicio de tu código. Por último, Faker se integra perfectamente con frameworks como pytest y puede combinarse con data-driven testing para crear hybrid approaches donde algunos datos vienen de archivos y otros se generan dinámicamente.

Ejemplo práctico: Test de registro con datos aleatorios

Aca tenés todo el código comentado para realizar una prueba funcional con Faker.
Test con Faker

🗨️**Storytelling**

Automatizando TalentoLab

El equipo de QA de TalentoLab recibió una nueva tarea importante. Marketing preparó una lista de usuarios reales que participarán en un sorteo interno, y también definieron los productos que tendrán descuento la próxima semana.

Silvia (Product Owner) explica:

"Matías, necesitamos asegurar que el sistema funcione perfectamente con todos estos usuarios y productos antes de lanzar la campaña. Los datos están en archivos separados para que Marketing pueda editarlos fácilmente."

Matías (Automation Lead) responde:

"Perfecto. Vamos a parametrizar nuestros tests y leeremos los datos desde archivos externos. Esto nos prepara también para el pipeline de CI/CD que implementaremos pronto."

Ejercicio Práctico

Refactorizar nuestros tests existentes para que:

1. Los datos vengan de archivos CSV y JSON
2. Se puedan ejecutar con múltiples combinaciones
3. Sean fáciles de mantener y actualizar

Actividad 1: Preparar la estructura de archivos

Organizá el proyecto creando carpetas como `datos/` y `utils/`. Así mantendrás separado el código de los tests, los datos de prueba y las funciones auxiliares, logrando un framework ordenado y fácil de escalar.

Creá la siguiente estructura en tu proyecto:

```
proyecto/
├── datos/
│   ├── login.csv
│   └── productos.json
├── utils/
│   └── datos.py
├── tests/
│   ├── test_login_csv.py
│   └── test_carrito_json.py
└── pages/
    ├── login_page.py
    └── inventory_page.py
```

Actividad 2: Crear archivos de datos

- **`datos/login.csv`** [datos.csv]
- **`datos/productos.json`** [datos.json]

Guardá la información de prueba en archivos externos (`login.csv`, `productos.json`). Esto te permitirá agregar, editar o revisar los datos de test sin tocar el código Python, facilitandote el mantenimiento y la colaboración.

Actividad 3: Crear funciones auxiliares

En `utils/datos.py` escribí funciones para leer los datos de los archivos CSV y JSON. Estas funciones transforman la información en un formato que Pytest puede usar para parametrizar los tests, desacoplando aún más los datos de la lógica.

Ejemplo:

`utils/datos.py`:

```
def leer_csv_login(ruta_archivo):
    """
    Lee el archivo CSV de credenciales de login
    Retorna lista de tuplas para pytest.mark.parametrize
    """
def leer_json_productos(ruta_archivo):
    """
    Lee el archivo JSON de productos
    Retorna lista de productos para parametrización
    """
```

Actividad 4: Test parametrizado de login

Refactorizá los tests de login para que tomen los usuarios y contraseñas desde el archivo CSV, usando `pytest.mark.parametrize`. Así podrás probar distintos escenarios de login de manera automática y sin duplicar código.

Ejemplo:

`tests/test_login_csv.py`:

```
@pytest.mark.parametrize("usuario, clave, debe_funcionar, descripcion", CASOS_LOGIN)
def test_login_desde_csv(driver, usuario, clave, debe_funcionar, descripcion):
    """
    Test parametrizado que verifica el login con datos del CSV
    """
@pytest.mark.smoke
def test_login_usuario_valido_smoke(driver):
    """
    Test de smoke para verificar que al menos un login funciona
    """
```

Actividad 5: Test parametrizado de carrito

Creá tests que agregan productos al carrito utilizando los datos guardados en el JSON. De esta forma, comprobarás la funcionalidad con diferentes productos y combinaciones, aprovechando al máximo la flexibilidad del enfoque data-driven.

Ejemplo:

tests/test_carrito_json.py:

```
@pytest.fixture
def usuario_logueado(driver):
    """
    Fixture que realiza login antes de cada test de carrito
    """

@pytest.mark.parametrize("producto", PRODUCTOS)
def test_agregar_producto_desde_json(usuario_logueado, producto):
    """
    Test que agrega cada producto del JSON al carrito
    """

@pytest.mark.smoke
def test_carrito_smoke(usuario_logueado):
    """
    Test de smoke que verifica funcionalidad básica del carrito
"""
```

Ejecutar los tests

- **# Ejecutar todos los tests**
  pytest tests/ -v -s
- **# Ejecutar solo tests de login**
  pytest tests/test_login_csv.py -v -s
- **# Ejecutar solo tests de carrito**
  pytest tests/test_carrito_json.py -v -s
- **# Ejecutar tests de smoke**
  pytest -m smoke -v -s

Conexión con el Proyecto Final

Todo lo que estás aprendiendo en esta clase será fundamental para tu proyecto final:

1. **Parametrización**: Te permitirá probar múltiples escenarios sin duplicar código
2. **Datos externos**: Harán tu framework más flexible y mantenible
3. **Data-driven approach**: Es una práctica profesional estándar
4. **Estructura organizada**: Las carpetas `datos/` y `utils/` serán parte de tu framework final

En las próximas clases usaremos estos mismos archivos de datos para:

- Tests de API (Clase 11-12)
- Reportes avanzados (Clase 13)
- Integración con CI/CD (Clase 15)

Próximos pasos

En la **Clase 11** comenzaremos con **Automatización de Pruebas de API** usando la librería `requests`. Utilizaremos algunos de los datos que creamos hoy para probar endpoints REST y validar respuestas JSON.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad