**Test Senaryoları Yazma ve Yapılandırma**

Selenium ile test senaryoları yazarken, temel amacımız tarayıcıları otomatik olarak kontrol etmek ve belirli bir web sayfası üzerindeki kullanıcı etkileşimlerini simüle etmektir. Bu senaryolar, web uygulamasının işlevselliğini test etmek için kullanılır.

* Basit bir test senaryosu, bir web sayfasını açmak, bir öğeyi bulmak, etkileşime geçmek ve sonucu doğrulamayı içerir.
* Test senaryolarında genellikle sonuçları doğrulamak için **assert** komutları kullanılır. Bu, beklenen sonucun elde edilip edilmediğini kontrol etmemizi sağlar.
* Selenium ile bir sayfadaki çeşitli öğelerle (butonlar, metin kutuları, açılır menüler vb.) etkileşime geçebilirsiniz.
* Web uygulamalarında bazı öğeler yavaş yüklenebilir. Selenium'da time.sleep() kullanmak yerine, daha etkin yöntemler olan implicit wait veya explicit wait kullanabilirsiniz.
  + **Implicit Wait**: Belirtilen süre boyunca tüm öğelerin yüklenmesini bekler.
  + **Explicit Wait**: Belirli bir öğe için bekler.

Bu adımlar, Selenium ile test senaryoları yazmaya başlamanız için temel oluşturacaktır. **unittest** veya **pytest** gibi yaygın olarak kullanılan frameworkler kullanarak testleri yapılandırarak daha sağlam ve düzenli bir test ortamı kurabilirsiniz.

Python unittest modülü, Python’un standart kütüphanesine dahil olan ve test yazmayı kolaylaştıran bir birim test çerçevesidir. Birim test (unit testing), yazılımın en küçük parçalarını (genellikle fonksiyonları veya sınıfları) bağımsız olarak test etmek anlamına gelir. unittest modülü, test fonksiyonlarının yazılması, çalıştırılması, geçme veya kalma durumlarının raporlanması gibi temel işlemleri sağlar.

**unittest Modülünün Özellikleri**

* **Test Sınıfları**: unittest modülünde testler, unittest.TestCase sınıfından türeyen özel sınıflarda yazılır.
* **assert Metodları**: assertEqual, assertTrue, assertIn gibi çeşitli assert metodları ile testlerin doğrulama işlemleri yapılır.
* **setUp ve tearDown**: Testlerden önce veya sonra yapılması gereken işlemler için setUp() ve tearDown() metotları kullanılır.
* **Test Koşucu (Test Runner)**: unittest testleri otomatik olarak bulup çalıştıran bir test koşucu içerir (unittest.main()).

pytest, Python dilinde yazılmış bir test çerçevesidir ve özellikle kolay kullanımı, esnekliği ve güçlü özellikleri ile popülerdir. pytest, birim testlerin yanı sıra fonksiyonel testler ve kapsamlı sistem testleri yazmak için de kullanılabilir. Test yazmayı ve çalıştırmayı kolaylaştıran bu araç, basit assert ifadeleriyle kontroller yapmanıza, testleri organize etmenize, testlerin tekrar kullanılabilirliğini sağlamanıza yardımcı olur.

**pytest’in Özellikleri**

* **Basit assert ifadeleri**: Test doğrulaması için assert ifadelerini kullanarak test yazmayı kolaylaştırır.
* **Fixture Desteği**: @pytest.fixture dekoratörü ile test öncesi veya sonrası yapılması gereken işlemler tanımlanabilir (örneğin, bir tarayıcı açıp kapatma).
* **Kolay Test Organizasyonu**: pytest, unittest gibi sınıflara ihtiyaç duymaz; sadece fonksiyon bazlı test yazmak yeterlidir.
* **Parametrik Test Desteği**: Parametreli test yazmanıza olanak tanır; bu sayede aynı testi farklı veri setleriyle çalıştırabilirsiniz.
* **Eklenti Desteği**: pytest birçok eklentiye sahiptir, örneğin pytest-django, pytest-cov (kapsam raporu), pytest-xdist (paralel test çalıştırma) gibi.

**1. unittest ve pytest ile Test Yapıları Oluşturma**

* **unittest**:
  + unittest Python’un yerleşik test modülüdür.
  + Test sınıfı içinde her bir test fonksiyonunu test\_ ile başlatmalısınız.
  + Setup ve teardown işlemlerini ayrı birer fonksiyonla yönetebilirsiniz.

# unittest modülünü içe aktarıyoruz, bu modül test yazmamıza yardımcı olur

import unittest

# Selenium modülünden webdriver'ı içe aktarıyoruz. Bu, tarayıcıyı kontrol etmemizi sağlar

from selenium import webdriver

# TestExample adlı bir test sınıfı oluşturuyoruz, unittest.TestCase sınıfından türetilmiştir

class TestExample(unittest.TestCase):

# setUp metodu, her testten önce çalışır ve gerekli başlangıç ayarlarını yapar

def setUp(self):

# Chrome tarayıcısını başlatıyoruz

self.driver = webdriver.Chrome()

# test\_open\_example adlı bir test fonksiyonu tanımlıyoruz

def test\_open\_example(self):

# Web tarayıcısı ile belirtilen URL'ye gidiyoruz

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

# Sayfa başlığını kontrol ediyoruz; eğer beklenen başlıkla eşleşmezse test başarısız olur

self.assertEqual(self.driver.title, "Expected Title")

# tearDown metodu, her testten sonra çalışır ve temizleme işlemlerini gerçekleştirir

def tearDown(self):

# Tarayıcıyı kapatıyoruz

self.driver.quit()

# Bu satır, dosya doğrudan çalıştırıldığında unittest.main() fonksiyonunu çağırır ve tüm testleri çalıştırır

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

unittest.main()

* **pytest**:
  + Daha sade bir yapıya sahiptir ve daha az kodla test yazmanızı sağlar.
  + pytest.fixture ile setup ve teardown işlevleri sağlanır.

# pytest modülünü içe aktarıyoruz; bu modül, Python'da test yazmayı ve çalıştırmayı kolaylaştırır

import pytest

# selenium modülünden webdriver'ı içe aktarıyoruz; bu, tarayıcıyı kontrol etmemizi sağlar

from selenium import webdriver

# driver adında bir pytest fixture tanımlıyoruz

@pytest.fixture

def driver():

# Chrome tarayıcısını başlatıyoruz

driver = webdriver.Chrome()

# driver nesnesini test fonksiyonuna teslim ediyoruz (yield kullanılarak)

yield driver

# test tamamlandıktan sonra tarayıcıyı kapatıyoruz

driver.quit()

# test\_open\_example adında bir test fonksiyonu tanımlıyoruz, driver parametresi fixture'dan geliyor

def test\_open\_example(driver):

# Web tarayıcısı ile belirtilen URL'ye gidiyoruz

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

# Sayfa başlığını kontrol ediyoruz; eğer beklenen başlıkla eşleşmezse test başarısız olur

assert driver.title == "Expected Title"

**2. Testlerin Bağımsız Çalışabilmesi İçin Yapılandırma**

Testlerin bağımsız çalışabilmesi, her testin kendi özel ortamında çalışmasını sağlar. Bu, testlerin birbirini etkilemesini engeller. **unittest** için setUp ve tearDown, **pytest** için @pytest.fixture kullanarak her testin başlangıç ve bitiş işlemlerini düzenleyebilirsiniz.

**3. Setup ve Teardown İşlemleri (Tarayıcı Açma/Kapama)**

Her test çalıştırıldığında tarayıcının açılması ve sonunda kapanması, testlerin bağımsız ve temiz bir ortamda çalışmasını sağlar. setUp fonksiyonu tarayıcıyı başlatırken, tearDown tarayıcıyı kapatır.

* Python’da unittest kütüphanesinde **tearDown()** metodu, her bir testin ardından otomatik olarak çalıştırılan temizlik kodlarını içerir.
* pytest ile de aynı işlemler için yield veya finalizer kullanılabilir.

# unittest ile örnek

def setUp(self):

self.driver = webdriver.Chrome()

def tearDown(self):

self.driver.quit()

# pytest ile örnek

@pytest.fixture

def driver():

driver = webdriver.Chrome()

yield driver

driver.quit()

**4. Test Sonuçlarının Doğrulanması (assert Komutları)**

Doğrulama komutları testlerin başarılı olup olmadığını belirler. En yaygın doğrulama komutları şunlardır:

* **unittest.assertEqual(a, b)**: a ile b'nin eşit olduğunu doğrular.
* **unittest.assertTrue(x)**: x değerinin True olduğunu doğrular.
* **pytest assert komutları**: assert ile kontrol edilir.

*unittest daha fazla yöntemle (assertEqual, assertTrue, assertIn gibi) çalışırken, pytest bu kontrolleri basit assert ifadeleriyle yapar ve hata durumunda otomatik olarak hata mesajları sağlar.*

# unittest

self.assertEqual(driver.title, "Expected Title")

# pytest

assert driver.title == "Expected Title"

**a. Sayfa Başlığını Kontrol Etme**

# unittest

self.assertEqual(driver.title, "Expected Title")

# pytest

assert driver.title == "Expected Title"

**b. Sayfada Belirli Bir Metnin Yer Aldığını Kontrol Etme**

# unittest

self.assertIn("Example", driver.page\_source)

# pytest

assert "Example" in driver.page\_source

**c. Sayfa URL'sinin Beklenen URL ile Eşleştiğini Kontrol Etme**

# unittest

self.assertEqual(driver.current\_url, "https://example.com")

# pytest

assert driver.current\_url == "https://example.com"

**d. Sayfadaki Belirli Bir Öğenin Görünür Olduğunu Kontrol Etme**

# unittest

element = driver.find\_element(By.ID, "example-element")

self.assertTrue(element.is\_displayed())

# pytest

element = driver.find\_element(By.ID, "example-element")

assert element.is\_displayed()

**e. Sayfadaki Belirli Bir Öğe Sayısının Beklenen Değerle Eşleştiğini Kontrol Etme**

# unittest

elements = driver.find\_elements(By.CLASS\_NAME, "example-class")

self.assertEqual(len(elements), 3)

# pytest

elements = driver.find\_elements(By.CLASS\_NAME, "example-class")

assert len(elements) == 3

**f. Sayfada Belirli Bir Öğenin Mevcut Olmadığını Kontrol Etme**

# unittest

try:

driver.find\_element(By.ID, "non-existent-element")

self.fail("Öğe mevcut, ancak mevcut olmaması bekleniyordu")

except NoSuchElementException:

pass

# pytest

with pytest.raises(NoSuchElementException):

driver.find\_element(By.ID, "non-existent-element")

**g. Sayfa Başlığının Beklenen Anahtar Kelimeyi İçermediğini Kontrol Etme**

# unittest

self.assertNotIn("Forbidden", driver.title)

# pytest

assert "Forbidden" not in driver.title

**h. Sayfadaki Belirli Bir Öğenin Metnini Kontrol Etme**

# unittest

element = driver.find\_element(By.TAG\_NAME, "h1")

self.assertEqual(element.text, "Expected Text")

# pytest

element = driver.find\_element(By.TAG\_NAME, "h1")

assert element.text == "Expected Text"

**5. Başarısızlık Durumlarını Yönetme ve Hata Ayıklama**

Başarısızlık durumları, testlerde karşılaşılan beklenmeyen sonuçlardır. Hata ayıklamak için test sonuçlarını gözden geçirip, Selenium’da try/except bloklarıyla hata kaynağını inceleyebilirsiniz. Ayrıca, pytest’in sunduğu --tb=short veya --tb=long gibi bayraklarla hata raporlamayı özelleştirebilirsiniz.

* **try/except Örneği**:

try:

assert driver.title == "Expected Title"

except AssertionError:

print("Test başarısız: Başlık beklenenden farklı.")

* **pytest ile detaylı çıktı**:

pytest --tb=short # Kısa hata mesajları

pytest --tb=long # Detaylı hata mesajları

Senaryolarınıza aşama ekledikçe her aşama için ayrı fonksiyonlar tanımlamak iyi bir uygulamadır. Bu yaklaşım, kodunuzu daha düzenli, okunabilir ve yönetilebilir hale getirir. Ayrıca, her bir fonksiyonun belirli bir işlevi olduğu için hata ayıklama ve testlerin bakımını kolaylaştırır. Avantajları:

1. **Modülerlik**: Her aşama için ayrı fonksiyonlar oluşturmak, her bir fonksiyonu bağımsız bir modül gibi düşünmenizi sağlar. Bu sayede, her bir fonksiyonu gerektiğinde bağımsız olarak güncelleyebilir veya değiştirebilirsiniz.
2. **Okunabilirlik**: Fonksiyonlar, kodunuzun amacını açıkça belirtir. Her fonksiyon belirli bir görevi yerine getirdiği için, kodu okuyan biri fonksiyon isimlerinden ne yaptığını anlayabilir.
3. **Yeniden Kullanılabilirlik**: Eğer belirli bir aşama birden fazla senaryoda tekrar ediyorsa, bu aşamayı bir fonksiyon haline getirip farklı testlerde kullanabilirsiniz. Bu, kod tekrarı ve bakım yükünü azaltır.
4. **Test Edilebilirlik**: Her bir fonksiyonu ayrı ayrı test edebilir ve belirli bir aşamada hata olduğunda sadece o fonksiyonu inceleyebilirsiniz.
5. **Bakım Kolaylığı**: Kodunuzu güncellerken, belirli bir aşamayı etkileyen değişiklikleri yalnızca o aşamanın fonksiyonunda yaparak diğer aşamalardan izole edebilirsiniz.