## Estructura y ejecución de pruebas

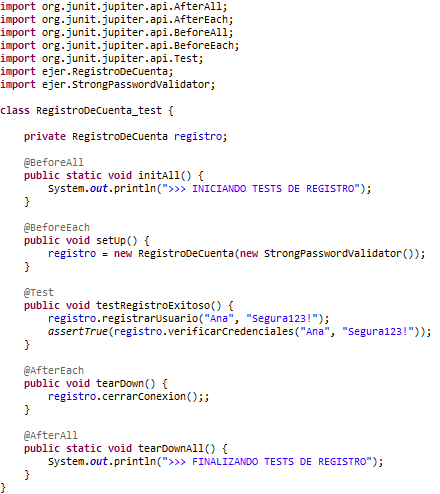
Las pruebas unitarias garantizan que el código funcione correctamente y se mantenga estable a lo largo del desarrollo. En Java (Junit) y Python (Pytest), las pruebas siguen estructuras diferentes pero tienen el mismo propósito.

En Java, la estructura de pruebas sigue el principio de clases espejo: Cada clase principal del proyecto tiene su correspondiente clase de prueba.  
Ejemplo:

Si tengo la clase RegistroDeCuenta, debe hacerse la clase de prueba RegistroDeCuenta\_test, que contendrá las pruebas para los métodos de esa clase.

Los métodos de prueba pueden tener cualquier nombre, pero deben tener encima una anotación que indique que es un método de prueba.

Ejemplo de prueba unitaria para RegistroDeCuenta:



Hay varias anotaciones algunas de las más utilizadas son:

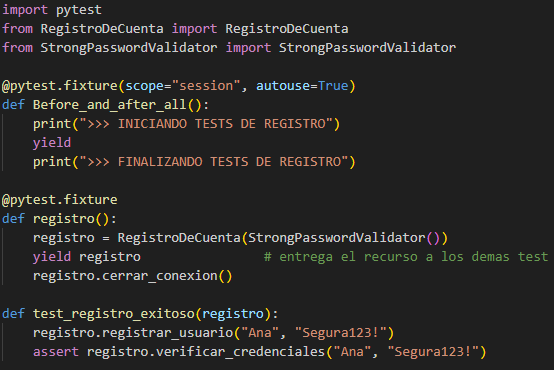
* @Test : Marca un método como test.
* @BeforeEach : Indica que este método se ejecuta antes de cada uno de los test, como una inicialización.
* @AfterEach : Indica que este método se ejecuta despues de cada uno de los test. En general, innecesario para la limpieza, ya que, el garbage collector se encarga de liberar la memoria automáticamente. Salvo que estemos trabajando por ejemplo con una conexion a bases de datos que debe cerrarse).

Por otra parte, los metodos tanto para @BeforeEach como @AfterAll, deben ser static.

* @BeforeAll : A diferencia de @BeforeEach, solo se ejecuta al principio antes de todos los tests de la clase, es decir, solo una vez.
* @AfterAll :  A diferencia de @AfterEach solo se ejecuta después de todos los tests, es decir, al final.

La ejecución de los test en Java cuando corremos los métodos de prueba en Eclipse nos brinda una vista con los resultados de cada test si paso el test o no. Lo cual nos ayuda a ver rápidamente en qué test tenemos fallas.

En Python, pytest usa convención de nombres en lugar de anotaciones:



Los archivos de prueba deben comenzar o terminar, por ejemplo, así test\_RegistroDeCuenta.py.

Los métodos deben comenzar con test\_ para que pytest las reconozca como pruebas.

El decorador @pytest.fixture permite crear una función que configura algo antes de cada test y puede limpiarlo después.

Si dentro del fixture se utiliza yield, todo lo que esté **antes del yield** se ejecuta **antes del test** (como un @BeforeEach), y lo que esté **después del yield** se ejecuta **después del test** (como un @AfterEach).

Además, si dentro del fixture se utiliza yield teniendo entre paréntesis:

 scope="session": significa que el fixture se ejecuta una sola vez por toda la sesión de pruebas, es decir, antes del primer test y después del último. Esto es el equivalente a @BeforeAll y @AfterAll en Java.

 autouse=True: hace que la fixture se aplique automáticamente, sin necesidad de nombrarla en los test.

El test usa registro como parámetro, por lo tanto, Pytest ve que existe una fixture llamada registro y la inyecta en aquellos métodos que lo reciban como parámetro.

## Tipos de aserciones

Las aserciones se ocupan para ver si una condición es válida o no, así con esto se puede ver los errores del código. Si la aserción da como válida quiere decir que este test paso correctamente, si no pasa se reporta la falla del test.

Las aserciones en JUnit tiene una clase que se llama Assertions que nos brinda los métodos estáticos para hacer las distintas comparaciones.

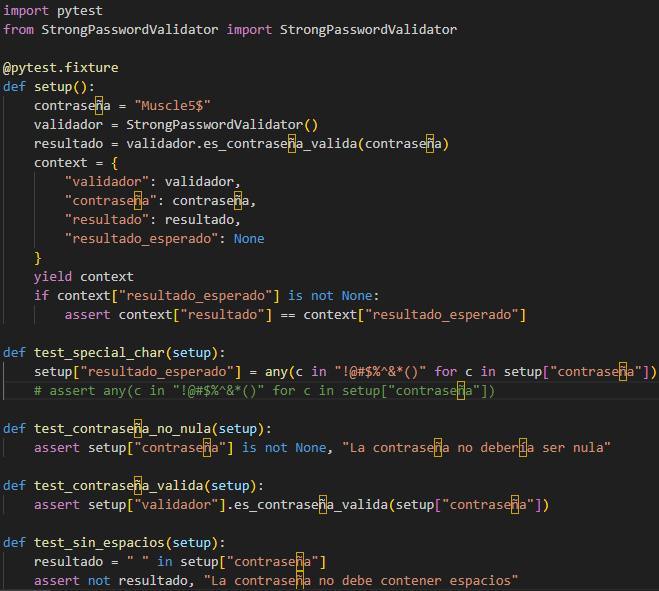


Las más comunes con:

* assertEquals(valor\_esperado, actual) : Que dos valores sean iguales.
* assertNotEquals(valor\_esperado, actual): Que dos valores sean diferentes.
* assertTrue(condición): Que una condición sea verdadera.
* assertFalse(condición): Que una condición sea falsa.
* assertNull(valor): Que el valor sea null.
* assertNotNull(valor): Que el valor no sea null.

Python (Pytest)

En pytest directamente se usa “assert” para verificar los valores esperados con los valores en los test. Pytest sobrecarga a assert para mostrar mensajes de error más legibles. Ejemplo: test\_contraseña\_no\_nula.



Los más comunes son:

* assert a == b : que a y b sean iguales.
* assert a != b : que a y b sean distintos.
* assert a is None : que a sea Nula.
* assert a is not None : que a no sea Nula.
* assert a in lista : que a este en un lista.
* assert cond : que cualquier condición boolean sea True.
* assert not cond : que cualquier condición boolean sea False.

## Manejo de excepciones en los test

El manejo de excepciones en pruebas unitarias es una técnica utilizada para verificar que un código responde adecuadamente ante condiciones inesperadas o errores predecibles. En lugar de que el sistema falle de manera abrupta, las pruebas aseguran que las excepciones se generen de manera controlada y con mensajes claros, permitiendo una depuración más efectiva y una mejor experiencia de usuario.

En JUnit ocupamos la clase Assertions para emplear los métodos que tiene sobre excepciones.

Los que tenemos son:

* assertThrows() : verifica que se lance una excepción del tipo esperado o una subclase. Estructura: assertThrows(TipoDeExcepcion.class, () -> { /\* cód \*/ });
* assertThrowsExactly() : verifica que se lance exactamente el tipo de excepcion que debería lanzarse según los parametros recibidos en el metodo. Este tipo de test falla si se lanza cualquier subclase distinta de, por ejemplo, IllegalArgumentException. La estructura es igual a la anterior.
* assertDoesNotThrow() : que no se lance ninguna excepción.



En Pytest se utiliza with pytest.raises() para verificar que la ejecución de un bloque de código genera la excepción correcta. Dentro del paréntesis va el código que debe lanzar la excepción. Es muy conciso y expresivo. También se puede ver la excepción capturada y mirar su mensaje.

Ejemplo:



## Uso de mocks o dobles de prueba

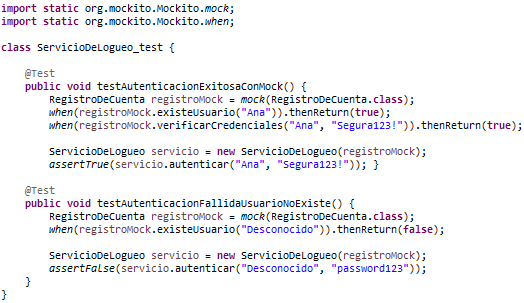
A veces en las pruebas unitarias se necesita simular el comportamiento de objetos externos ejemplo como APIs, base de datos, archivos, etc, para no depender de su estado real existen los dobles de prueba, el más común es el mock.

Tipos de dobles de prueba:

* **Dummy**: no se usa realmente, solo ocupa un lugar.
* **Stub**: devuelve datos fijos.
* **Spy**: graba llamadas.
* **Mock**: verifica llamadas y comportamientos esperados.
* **Fake**: reemplazo funcional simple (como una base de datos en memoria).

En JUnit no tiene mocks por sí mismo, así que se junta con Mockito que es una librería de mocking.

Ejemplo:



Esta es la forma manual de crear mocks con Mockito. Creamos explícitamente el mock, configuramos su comportamiento con when(...), y lo pasamos manualmente al objeto que vamos a testear (“Cuenta” en este caso).

Esta es la forma ideal para Tests simples funcionando muy bien y otorgando control total, pero requiere que hagamos toda la configuración a mano.

Recomendación: Usar esta forma manual cuando tengamos pocos mocks o deseemos un control más directo sobre la creación de objetos.

Ahora bien, cuando estemos trabajando con clases que tienen varias dependencias, y quieras evitar escribir mucho código repetido, es decir, muchos mocks usa @Mock, @InjectMocks y @BeforeEach + MockitoAnnotations.openMocks(this) donde Mockito se encarga automáticamente de inyectar los mocks en el objeto bajo prueba.

Ejemplo:

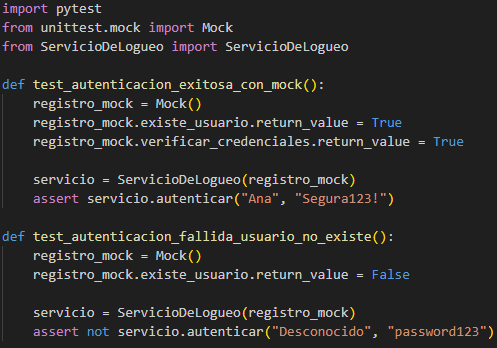


Python (Pytest)

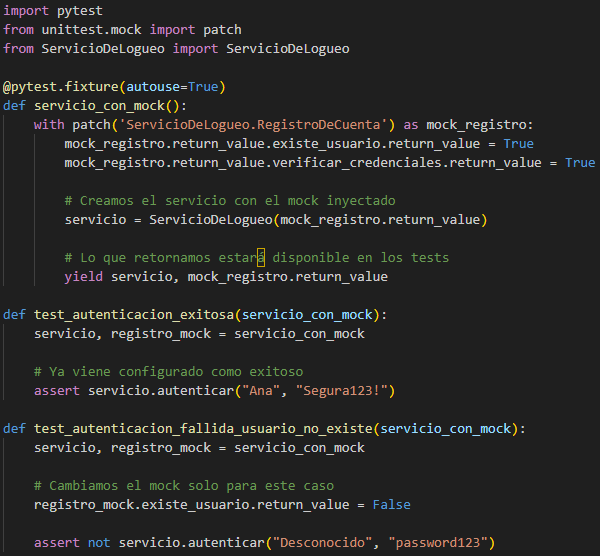
En Python, el equivalente a la creación manual de mocks en Mockito es unittest.mock, que permite crear objetos simulados, definir su comportamiento y pasarlos manualmente al código que queremos probar. Funciona de manera similar a Mockito en Java, pero con una sintaxis diferente.

Ejemplo:

1. Creamos un mock manualmente (Mock()) en lugar de usar una implementación real de RegistroDeCuenta.
2. Definimos respuestas predecibles (return\_value = True/False) para cualquier par de datos que se manden al metodo autenticar, se configuran retornos defaults, simulando una autenticación.
3. Pasamos el mock aServicioDeLogueo sin depender de datos reales, ya que, retorna true/false dependiendo de lo que se le haya definido antes de mandarle el mock a ServicioDeLogueo.



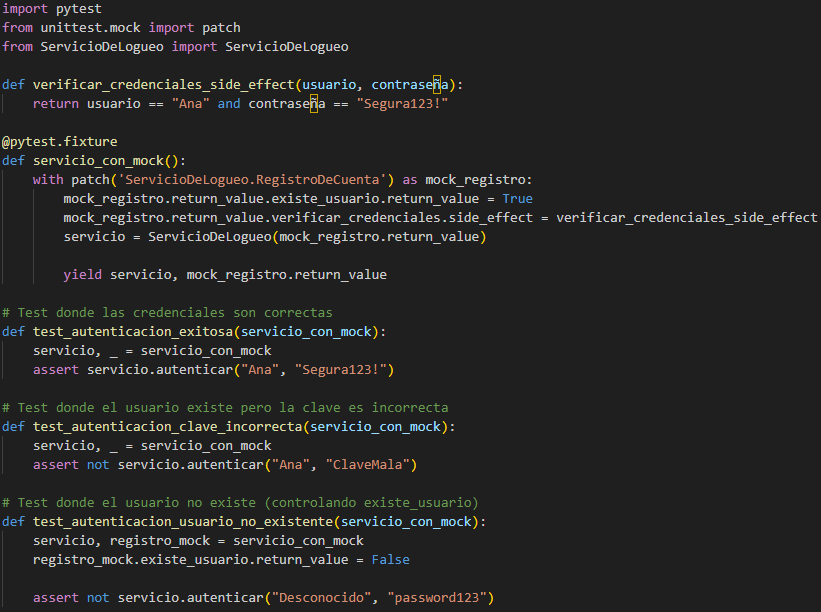
O podemos usar with patch('ServicioDeLogueo.RegistroDeCuenta') con @pytest.fixture(autouse=True) para evitar repetir código y hacerlo automatico.



Aunque estos son los metodos mas simples en Python, es posible simular lo mismo que en Java con when(...).thenAnswer(...) para unos pocos pares de datos específicos.

En lugar de devolver siempre True o False, usamos side\_effect con una función que reciba usuario y contraseña y que devuelva True o False según corresponda.

Entonces, verificar\_credenciales simula la validación real según los parámetros recibidos. Por lo tanto, ya no devuelve siempre True/False sino que se comporta como una función real y solo llamas a registro\_mock.existe\_usuario para cambiar casos particulares.



Es importante aclarar la razón de porque, a pesar de no haber importado el script de registro de cuenta, el mock, aun asi es capaz de simular sus objetos. Esto se debe a que Python utiliza **Duck Typing**, lo que significa que ServicioDeLogueo no verifica si el objeto que recibe es realmente una instancia de RegistroDeCuenta, sino que simplemente usa sus métodos.

Mientras el mock tenga los métodos que ServicioDeLogueo espera llamar, el sistema funcionará sin problemas. Para lograr esto, el comportamiento del mock se configura con return\_value o .side\_effect, permitiendo definir respuestas simuladas sin importar la clase original.

Gracias a esta característica, los mocks pueden reemplazar cualquier dependencia en pruebas sin necesidad de definir tipos específicos o importar el código real.

## Pruebas parametrizadas

Las pruebas parametrizadas permiten ejecutar la misma prueba varias veces, usando diferentes datos de entrada sin tener que duplicar el código. Esto es útil cuando queremos comprobar que una función se comporta correctamente con múltiples casos (valores válidos, extremos, errores, etc.).

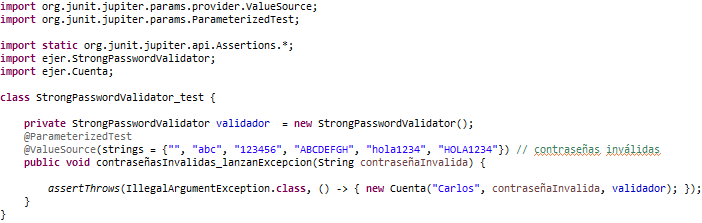
Java (Junit)

En JUnit se ocupa la anotación @ParameterizedTest luego se coloca otra anotación para proveer los datos. Algunas son:

* @EnumSource: prueba con valores de una enumeración.



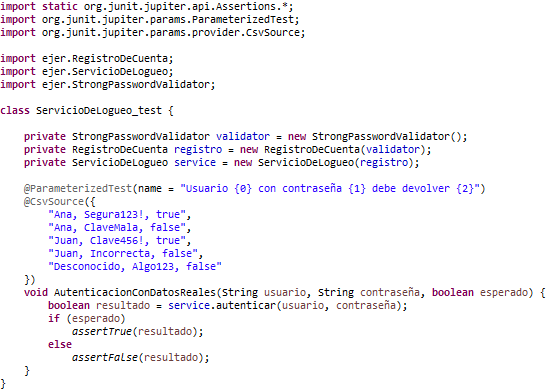
* @ValueSource: provee un solo valor por ejecución. Ejemplo:



* @MethodSource: llama a un método que retorna un stream/colección con datos.
* @CsvFileSource: carga los parámetros desde un archivo CSV externo.



* @CsvSource: provee varios parámetros, como tuplas.



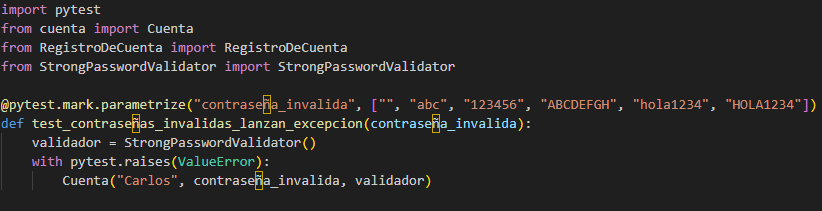
Python (Pytest)

Usamos @pytest.mark.parametrize para definir múltiples conjuntos de argumentos para una función de prueba.

Aquí simulamos lo mismo que @EnumSource, quien usaba un switch pero en Python, esto se hace con un **@pytest.mark.parametrize**, que automáticamente itera. Así también obtiene un reporte separado de los casos (igual que JUnit).



Aquí simulamos @ValueSource de java que proveía un solo valor por ejecución y aquí en Python hacemos lo mismo con **@pytest.mark.parametrize** y una lista de strings que son inyectados un valor distinto en cada ejecucion:



Ahora simulamos @MethodSource de java que llama a un método que retorna un stream/colección con datos, en Python que toma cada tupla por separado y luego la evalúa.



Tambien simulamos @CsvFileSource (que carga los parámetros desde un archivo CSV externo) en Python que lee el csv con with open, con csv.DictReader y @pytest.mark.parametrize que toma cada linea por separado y luego la evalua.



Por último, @CsvSource (que provee varios parámetros, como tuplas) puede simularse en Python de esta manera:

