

#

# DESARROLLO WEB FULLSTACK

con Python y JavaScript

polotic  
misiones

# Clase 12: Anexo 3



**TESTING**



# Testing



Una parte importante del proceso de desarrollo de software es el acto de probar el código que hemos escrito para asegurarnos de que todo funcione como esperamos. Aquí analizaremos varias formas en las que podemos mejorar la forma en que probamos nuestro código.

# Assert

Una de las formas más sencillas en las que podemos ejecutar pruebas en Python es mediante el comando `assert`. Este comando va seguido de alguna expresión que debería ser `True`. Si la expresión es `True`, no sucederá nada y si es `False`, se lanzará una excepción.

Veamos cómo podríamos incorporar un comando para probar la función cuadrado que escribimos cuando aprendimos Python por primera vez. Cuando la función se escribe correctamente, no sucede nada ya que `assert` es `True`

```
def cuadrado(x):  
    return x * x  
  
assert cuadrado(10) == 100
```

```
""" Salida:  
  
"""
```

# Assert

Y luego, cuando está escrito incorrectamente, se lanza una excepción.

```
def cuadrado(x):  
    return x + x  
  
assert cuadrado(10) == 100
```

```
""" Salida:  
Traceback (most recent call apellido):  
  File "assert.py", line 4, in <module>  
    assert square(10) == 100  
AssertionError  
"""
```



# Desarrollo basado en pruebas



A medida que comienzas a construir proyectos más grandes, es posible que desees considerar el uso del *desarrollo basado en pruebas* o *Test Driven Development*, un estilo de desarrollo en el que cada vez que corrige un error, agregas una prueba que verifica ese error a un conjunto creciente de pruebas que se ejecutan cada vez que haces cambios.

Esto te ayudará a asegurarte de que las funciones adicionales que agregues a un proyecto no interfieran con las funciones ya existentes. Ahora, veamos una función un poco más compleja y pensemos en cómo escribir pruebas puede ayudarnos a encontrar errores.

# Desarrollo basado en pruebas

Escribiremos una función llamada `es_primo` que devuelve `True` si y solo si su entrada es un número primo:

```
import math

def es_primo(n):

    # Sabemos que numeros menores a 2 no son primos
    if n < 2:
        return False

    # Verificando factores hasta sqrt(n)
    for i in range(2, int(math.sqrt(n))):

        # Si I es un factor, retornar false
        if n % i == 0:
            return False

    # Si no se encontraron factores, retornar true
    return True
```

# Desarrollo basado en pruebas



Ahora, echemos un vistazo a una función que hemos escrito para probar nuestra función principal:

```
from primos import es_primo

def test_primo(n, esperado):
    if es_primo(n) != esperado:
        print(f"ERROR en es_primo({n}), se esperaba {esperado}")
```

En este punto, podemos ir al intérprete de Python y probar algunos valores:

```
>>> import test
>>> test.test_primo(5, True)
>>> test.test_primo(10, False)
>>> test.test_primo(25, False)
ERROR en es_primo(25), se esperaba False
```



# Desarrollo basado en pruebas



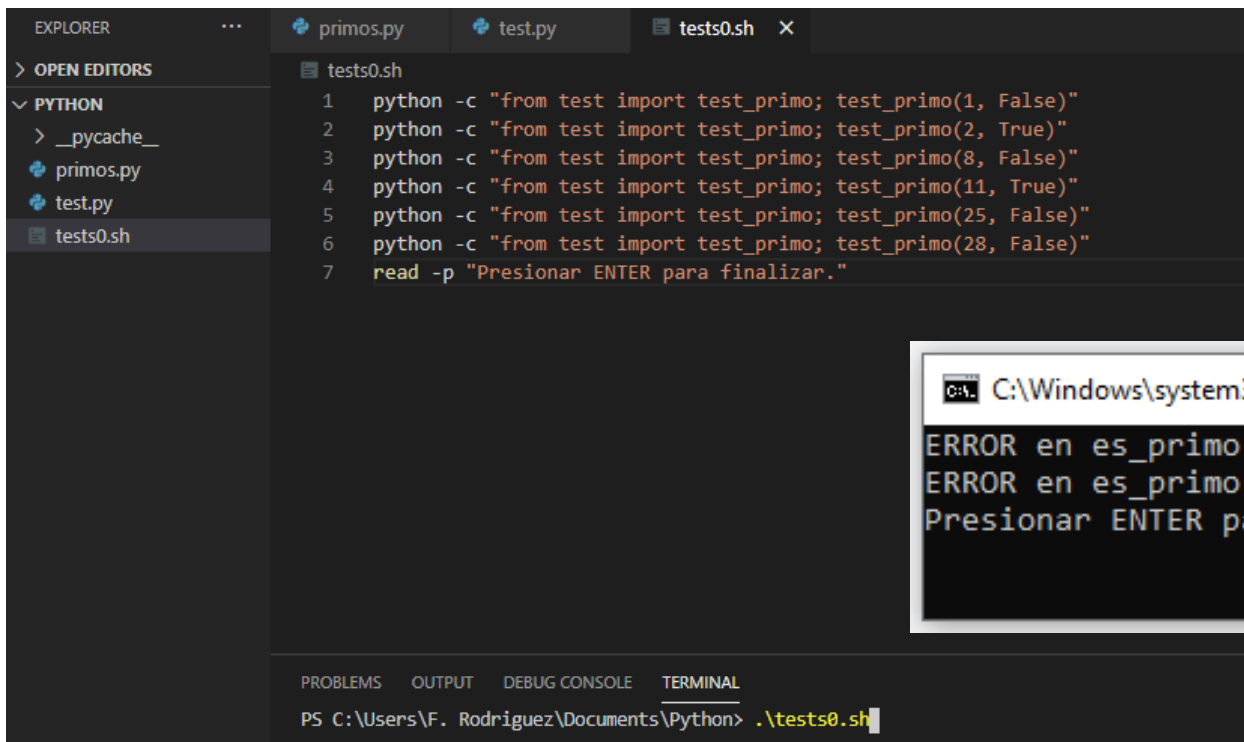
Podemos ver en el resultado anterior que 5 y 10 se identificaron correctamente como primos y no primos, pero 25 se identificó incorrectamente como primos, por lo que debe haber algo mal en nuestra función.

Sin embargo, antes de ver qué está mal con nuestra función, veamos una forma de **automatizar** nuestras pruebas. Una forma en que podemos hacer esto es creando un script de shell, o algún script que se pueda ejecutar dentro de nuestra terminal. Estos archivos requieren una extensión `.sh`, por lo que nuestro archivo se llamará `tests0.sh`.

Cada una de las líneas siguientes consta de

- ✓ Un comando `python` para especificar que estamos ejecutando comandos de Python
- ✓ Un `-c` para indicar que deseamos ejecutar un comando
- ✓ Un comando para ejecutar en formato de cadena

# Desarrollo basado en pruebas



The image shows a Visual Studio Code editor window with three files open: `primos.py`, `test.py`, and `tests0.sh`. The `tests0.sh` file contains a script that runs a series of Python commands to test a function `test_primo` with various inputs. The script is as follows:

```
1 python -c "from test import test_primo; test_primo(1, False)"
2 python -c "from test import test_primo; test_primo(2, True)"
3 python -c "from test import test_primo; test_primo(8, False)"
4 python -c "from test import test_primo; test_primo(11, True)"
5 python -c "from test import test_primo; test_primo(25, False)"
6 python -c "from test import test_primo; test_primo(28, False)"
7 read -p "Presionar ENTER para finalizar."
```

Below the editor, a terminal window is open, showing the output of the script. The terminal title is `C:\Windows\system32\cmd.exe`. The output indicates that the function `es_primo` failed for inputs 8 and 25, where `False` was expected but not received. The terminal output is:

```
ERROR en es_primo(8), se esperaba False
ERROR en es_primo(25), se esperaba False
Presionar ENTER para finalizar._
```

At the bottom of the VS Code window, the terminal tab is selected, showing the command prompt `PS C:\Users\F. Rodriguez\Documents\Python> .\tests0.sh`.



# UNIT TESTING

## TEST DE UNIDAD



# Unit Testing



Aunque pudimos ejecutar pruebas automáticamente usando el método anterior, es posible que deseemos evitar tener que escribir cada una de esas pruebas. Afortunadamente, podemos usar la biblioteca unittest de Python para facilitar un poco este proceso. Echemos un vistazo a cómo se vería un programa de prueba para nuestra función `es_primo`.

# Unit Testing

```
# Importar la librería unittest y nuestra funcion
import unittest
from primos import es_primo

# Una clase que contiene todos nuestros tests
class Tests(unittest.TestCase):

    def test_1(self):
        """Verificar que 1 no es un numero primo."""
        self.assertFalse(es_primo(1))

    def test_2(self):
        """Verificar que 2 no es un numero primo."""
        self.assertTrue(es_primo(2))

    def test_8(self):
        """Verificar que 8 no es un numero primo."""
        self.assertFalse(es_primo(8))

    def test_11(self):
        """Verificar que 11 no es un numero primo."""
        self.assertTrue(es_primo(11))

    def test_25(self):
        """Verificar que 25 no es un numero primo."""
        self.assertFalse(es_primo(25))

    def test_28(self):
        """Verificar que 28 no es un numero primo."""
        self.assertFalse(es_primo(28))

# Correr cada una de las funciones de testing
if __name__ == "__main__":
    unittest.main()
```



# Unit Testing

- ❖ Observe que cada una de las funciones dentro de nuestra clase Tests siguió un patrón: El nombre de las funciones comienza con `test_`. Esto es necesario para que las funciones se ejecuten automáticamente con la llamada a `unittest.main()`.
- ❖ Cada prueba incluye el argumento `self`. Esto es estándar cuando se escriben métodos dentro de las clases de Python.
- ❖ La primera línea de cada función contiene un **docstring** rodeado por tres comillas. Estos no son solo para la legibilidad del código. Cuando se ejecutan las pruebas, el comentario se mostrará como una descripción de la prueba si falla.
- ❖ La siguiente línea de cada una de las funciones contenía una afirmación con el formato `self.assertALGO`. Hay muchas afirmaciones diferentes que puedes hacer, incluidas `assertTrue`, `assertFalse`, `assertEqual` y `assertGreater`. Puedes encontrar estos y más consultando la [documentación](#).

# Unit Testing

```
PS C:\Users\F. Rodriguez\Documents\Python> python unit.py
...F.F
```

```
=====
FAIL: test_25 (__main__.Tests)
Verificar que 25 no es un numero primo.
```

```
-----
Traceback (most recent call last):
  File "unit.py", line 26, in test_25
    self.assertFalse(es_primo(25))
AssertionError: True is not false
```

```
=====
FAIL: test_8 (__main__.Tests)
Verificar que 8 no es un numero primo.
```

```
-----
Traceback (most recent call last):
  File "unit.py", line 18, in test_8
    self.assertFalse(es_primo(8))
AssertionError: True is not false
```

```
-----
Ran 6 tests in 0.003s
```

```
FAILED (failures=2)
```

1. Después de ejecutar las pruebas, unittest nos proporciona información útil sobre lo que encontró. En la primera línea, nos da una serie de puntos ( . ) para éxitos y efes ( F ) para fallas en el orden en que se escribieron nuestras pruebas.
2. A continuación, para cada una de las pruebas que fallaron, se nos da el nombre de la función que falló.
3. Luego figura el comentario descriptivo que proporcionamos anteriormente mas un Traceback de la excepción.
4. Y finalmente, se nos muestra cuántas pruebas se ejecutaron, cuánto tiempo tomaron y cuántas fallaron.

# Unit Testing

Ahora echemos un vistazo para corregir el error en nuestra función `es_primo(n)`. Resulta que necesitamos probar un número adicional en nuestro ciclo `for`. Por ejemplo, cuando `n` es 25, la raíz cuadrada es 5, pero cuando ese es un argumento en la función `range`, el ciclo `for` termina en el número 4. Por lo tanto, podemos simplemente cambiar el encabezado de nuestro ciclo `for` a:

```
for i in (2, (math.sqrt(n)) + 1):
```

```
PS C:\Users\F. Rodriguez\Documents\Python> python unit.py
```

```
.....
```

```
-----
```

```
Ran 6 tests in 0.001s
```

```
OK
```

```
PS C:\Users\F. Rodriguez\Documents\Python> █
```





# Unit Testing



Estas pruebas automatizadas serán aún más útiles a medida que trabajes para optimizar esta función.

Siempre que realices cambios para mejorar esta función, querrás tener la capacidad de ejecutar fácilmente las pruebas unitarias nuevamente para asegurarte de que su función aún siga siendo correcta.



**TESTING**



# Django Testing



Ahora, veamos cómo podemos aplicar las ideas de las pruebas automatizadas al crear aplicaciones Django.

Mientras trabajamos con esto, usaremos el proyecto de Vuelos que creamos cuando conocimos los modelos de Django. Primero, agregaremos un método a nuestro modelo de vuelo que verifica que un vuelo es válido al verificar dos condiciones:

1. El origen no es el mismo que el destino
2. La duración es superior a 0 minutos.

# Django Testing

Ahora, nuestro modelo podría verse así:

```
class Vuelo(models.Model):
    origen = models.ForeignKey(Aeropuerto, on_delete=models.CASCADE, related_name="salidas")
    destino = models.ForeignKey(Aeropuerto, on_delete=models.CASCADE, related_name="arribos")
    duracion = models.IntegerField()

    def __str__(self):
        return f"{self.id}: {self.origen} a {self.destino}"

    def es_valido_vuelo(self):
        return self.origen != self.destino or self.duracion > 0
```



# Django Testing



Para asegurarnos de que nuestra aplicación funcione como se espera, cada vez que creamos una nueva aplicación, automáticamente se nos da un archivo `tests.py`. Cuando abrimos este archivo por primera vez, vemos que la biblioteca `TestCase` de Django se importa automáticamente:

```
from django.test import TestCase
```

Una ventaja de usar la biblioteca `TestCase` es que cuando ejecutamos nuestras pruebas, se creará una base de datos completamente nueva solo con fines de prueba.

Esto es útil porque evitamos el riesgo de modificar o eliminar accidentalmente entradas existentes en nuestra base de datos y no tenemos que preocuparnos por eliminar entradas ficticias que creamos solo para pruebas.

# Django Testing



Para comenzar a usar esta biblioteca, primero queremos importar todos nuestros modelos:

```
from .models import Vuelo, Aeropuerto, Pasajero
```

Y luego crearemos una nueva clase que amplíe la clase `TestCase` que acabamos de importar. Dentro de esta clase, definiremos una función `setUp` que se ejecutará al inicio del proceso de prueba.

En esta función, probablemente querremos crear. Así es como se verá nuestra clase para comenzar:

```
class VueloTestCase(TestCase):  
  
    def setUp(self):  
  
        # Crear aeropuertos.  
        a1 = Aeropuerto.objects.create(codigo="AAA", ciudad="Ciudad A")  
        a2 = Aeropuerto.objects.create(codigo="BBB", ciudad="Ciudad B")  
  
        # Crear vuelos.  
        Vuelo.objects.create(origen=a1, destino=a2, duracion=100)  
        Vuelo.objects.create(origen=a1, destino=a1, duracion=200)  
        Vuelo.objects.create(origen=a1, destino=a2, duracion=-100)
```

# Django Testing



Ahora que tenemos algunas entradas en nuestra base de datos de prueba, agreguemos algunas funciones a esta clase para realizar algunas pruebas.

Primero, asegurémonos de que nuestros campos de salidas y arribos funcionen correctamente intentando contar el número de salidas (que sabemos que deberían ser 3) y arribos (que deberían ser 1) desde el aeropuerto AAA:

```
def test_salidas_count(self):
    a = Aeropuerto.objects.get(codigo="AAA")
    self.assertEqual(a.salidas.count(), 3)

def test_arribos_count(self):
    a = Aeropuerto.objects.get(codigo="AAA")
    self.assertEqual(a.arribos.count(), 1)
```

# Django Testing



También podemos probar la función `es_valido_vuelo` que agregamos a nuestro modelo de vuelo. Comenzaremos afirmando que la función devuelve `True` cuando el vuelo es válido:

```
def test_valido_vuelo(self):  
    a1 = Aeropuerto.objects.get(codigo="AAA")  
    a2 = Aeropuerto.objects.get(codigo="BBB")  
    f = Vuelo.objects.get(origen=a1, destino=a2, duracion=100)  
    self.assertTrue(f.es_valido_vuelo())
```



# Django Testing

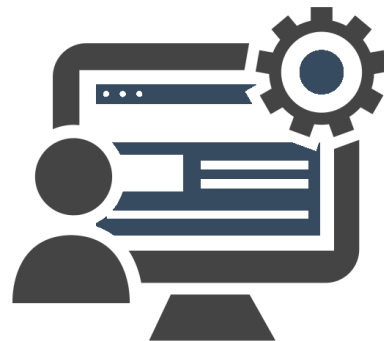


A continuación, asegurémonos de que los vuelos con destinos y duraciones no válidos devuelvan falso:

```
def test_invalido_vuelo_destino(self):
    a1 = Aeropuerto.objects.get(codigo="AAA")
    f = Vuelo.objects.get(origen=a1, destino=a1)
    self.assertFalse(f.es_valido_vuelo())

def test_invalida_vuelo_duracion(self):
    a1 = Aeropuerto.objects.get(codigo="AAA")
    a2 = Aeropuerto.objects.get(codigo="BBB")
    f = Vuelo.objects.get(origen=a1, destino=a2, duracion=-100)
    self.assertFalse(f.es_valido_vuelo())
```

Ahora, para ejecutar nuestras pruebas, ejecutaremos `python manage.py test`. El resultado de esto es casi idéntico al resultado que vimos al usar la librería `unittest` de Python, aunque también registra que está creando y destruyendo una base de datos de prueba:



# CLIENT TESTING



# Client Testing



Al crear aplicaciones web, probablemente querremos comprobar no solo si funcionan o no funciones específicas, sino también si las páginas web individuales se cargan según lo previsto. Podemos hacer esto creando un objeto Client en nuestra clase de prueba de Django y luego realizando solicitudes usando ese objeto. Para hacer esto, primero tendremos que agregar Client a nuestras importaciones:

```
from django.test import Client, TestCase
```

# Client Testing

Por ejemplo, agreguemos ahora una prueba que asegure que obtenemos un código de respuesta HTTP de 200 y que los tres vuelos se agregan al contexto de una respuesta:

```
def test_index(self):  
  
    # Configurar el cliente para hacer las requests  
    c = Client()  
  
    # Enviar request get a la pagina index y almacenar la respuesta  
    response = c.get("/vuelos/")  
  
    # Nos aseguramos de que devuelva el codigo de estado 200  
    self.assertEqual(response.status_codigo, 200)  
  
    # Nos aseguramos que los tres vuelos hayan sido retornados en el contexto  
    self.assertEqual(response.context["vuelos"].count(), 3)
```

# Client Testing

De manera similar, podemos verificar para asegurarnos de obtener un código de respuesta válido para una página de vuelo válida y un código de respuesta no válido para una página de vuelo que no existe. (Tenga en cuenta que usamos la función Max para encontrar el id máximo, al que tenemos acceso al incluir desde `django.db.models import Max` en la parte superior de nuestro archivo)

```
def test_pagina_vuelo_valida(self):
    a1 = Aeropuerto.objects.get(codigo="AAA")
    f = Vuelo.objects.get(origin=a1, destination=a1)

    c = Client()
    response = c.get(f"/vuelos/{f.id}")
    self.assertEqual(response.status_codigo, 200)

def test_pagina_vuelo_in0valida(self):
    max_id = Vuelo.objects.all().aggregate(Max("id"))["id__max"]

    c = Client()
    response = c.get(f"/vuelos/{max_id + 1}")
    self.assertEqual(response.status_codigo, 404)
```

# Client Testing

Finalmente, agreguemos algunas pruebas para asegurarnos de que las listas de pasajeros y no pasajeros se generen como se esperaba:

```
def test_pagina_vuelo_pasajeros(self):
    f = Vuelo.objects.get(pk=1)
    p = Pasajero.objects.create(nombre="Juan", apellido="Perez")
    f.pasajeros.add(p)

    c = Client()
    response = c.get(f"/vuelos/{f.id}")
    self.assertEqual(response.status_code, 200)
    self.assertEqual(response.context["pasajeros"].count(), 1)

def test_pagina_vuelo_no_pasajeros(self):
    f = Vuelo.objects.get(pk=1)
    p = Pasajero.objects.create(nombre="Juan", apellido="Perez")

    c = Client()
    response = c.get(f"/vuelos/{f.id}")
    self.assertEqual(response.status_code, 200)
    self.assertEqual(response.context["no_pasajeros"].count(), 1)
```



**SELENIUM**



# Selenium



Hasta ahora, hemos podido probar el código del lado del servidor que hemos escrito usando Python y Django, pero a medida que creamos nuestras aplicaciones, queremos tener la capacidad de crear pruebas para nuestro código también del lado del cliente.

Por ejemplo, pensemos en nuestra página contador.html y trabajemos en escribir algunas pruebas para ella.



# Selenium

Comenzaremos escribiendo una página de contador ligeramente diferente donde incluimos un botón para disminuir el recuento:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <title>Contador</title>
    <script>

      // Esperar que cargue la pagina
      document.addEventListener('DOMContentLoaded', () => {

        // Inicializar variable a 0
        let counter = 0;

        // Si se clikea el botón de aumentar. Se aumenta el contador y se cambia el innerHTML
        document.querySelector('#incrementar').onclick = () => {
          counter ++;
          document.querySelector('h1').innerHTML = counter;
        }

        // Si se clikea el botón de decrementar, disminuir el contador y cambiar el innerHTML
        document.querySelector('#decrementar').onclick = () => {
          counter --;
          document.querySelector('h1').innerHTML = counter;
        }
      })
    </script>
  </head>
  <body>
    <h1>0</h1>
    <button id="incrementar">+</button>
    <button id="decrementar">-</button>
  </body>
</html>
```



# Selenium



Ahora, si deseamos probar este código, podríamos abrir nuestro navegador web, hacer clic en los dos botones y observar lo que sucede.

Esto, sin embargo, se volvería muy tedioso a medida que escribe aplicaciones de una sola página cada vez más grandes, por lo que se han creado varios frameworks que ayudan con las pruebas en el navegador, uno de los cuales se llama [Selenium](#).



# Selenium



Con Selenium, podremos definir un archivo de prueba en Python donde podemos simular que un usuario abre un navegador web, navega a nuestra página e interactúa con él.

Nuestra principal herramienta al hacer esto se conoce como **Web Driver**, que abrirá un navegador web en tu computadora.

Veamos cómo podemos empezar a utilizar esta librería para comenzar a interactuar con las páginas.

Ten en cuenta que a continuación usamos tanto selenium como ChromeDriver. Selenium se puede instalar para python ejecutando `pip install selenium`, y ChromeDriver se puede instalar ejecutando `pip install chromedriver-py`

# Selenium



La configuración básica es la siguiente:

```
import os
import pathlib
import unittest

from selenium import webdriver

# Encuentar el URI (Uniform Resource Identifier) de un archivo
def file_uri(filename):
    return pathlib.Path(os.path.abspath(filename)).as_uri()

# Configura el webdriver usando Google chrome
driver = webdriver.Chrome()
```

Una nota sobre las primeras líneas es que para apuntar a una página específica, necesitamos el Identificador uniforme de recursos (URI) de esa página, que es una cadena única que representa ese recurso.

# Selenium

## Usando los comandos en el interprete de Python:

```
# Encontrar la URI de nuestro archivo recientemente creado
>>> uri = file_uri("contador.html")

# Usar la URI para abrir la pagina web
>>> driver.get(uri)

# Acceder al titulo de la pagina actual
>>> driver.title

# Acceder al código fuente de la página
>>> driver.page_source

# Encontrar y almacenar los botones de incremento y decremento:
>>> incrementar = driver.find_element_by_id("incrementar")
>>> decrementar = driver.find_element_by_id("decrementar")

# Simular que el usuario hizo click en los dos botones
>>> incrementar.click()
>>> incrementar.click()
>>> decrementar.click()

# Podemos incluso incluir clicks dentro de otras estructuras de Python:
>>> for i in range(25):
...     incrementar.click()
```

# Selenium

Ahora echemos un vistazo a cómo podemos usar esta simulación para crear pruebas automatizadas de nuestra página:

```
# Marco estándar de una clase de testing
class WebpageTests(unittest.TestCase):

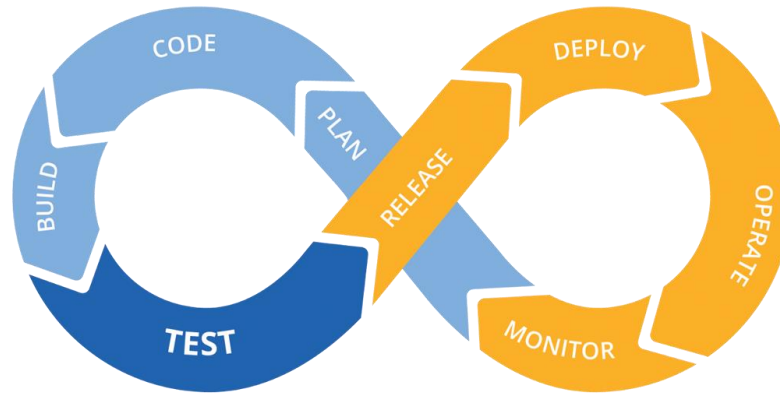
    def test_titulo(self):
        """Asegurarse de que el titulo es correcto"""
        driver.get(file_uri("contador.html"))
        self.assertEqual(driver.title, "Contador")

    def test_incrementar(self):
        """Asegurarse de que la cabecera se actualizó a 1 luego de 1 click del botón incrementar"""
        driver.get(file_uri("contador.html"))
        incrementar = driver.find_element_by_id("incrementar")
        incrementar.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "1")

    def test_decrementar(self):
        """Asegurarse que la cabecera se actualice a -1 luego de 1 click del botón incrementar"""
        driver.get(file_uri("contador.html"))
        decrementar = driver.find_element_by_id("decrementar")
        decrementar.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "-1")

    def test_multiples_incrementar(self):
        """Asegurarse de que la cabecera se actualice a 3 luego de 3 clicks del botón incrementar"""
        driver.get(file_uri("contador.html"))
        incrementar = driver.find_element_by_id("incrementar")
        for i in range(3):
            incrementar.click()
        self.assertEqual(driver.find_element_by_tag_name("h1").text, "3")

if __name__ == "__main__":
    unittest.main()
```



CI/CD



# CI/CD

**CI / CD** es un conjunto de mejores prácticas de desarrollo de software que dictan cómo un equipo de personas escribe el código y cómo ese código se entrega posteriormente a los usuarios de la aplicación. Como su nombre lo indica, este método consta de dos partes principales:

- Continuous Integration / Integración continua:
  - Fusiones frecuentes a la rama principal
  - Pruebas unitarias automatizadas con cada combinación
  
- Continuous Delivery / Entrega continua:
  - Programas de lanzamiento cortos, lo que significa que las nuevas versiones de una aplicación se lanzan con frecuencia.





# CI/CD

CI / CD se ha vuelto cada vez más popular entre los equipos de desarrollo de software por varias razones:

- Cuando diferentes miembros del equipo están trabajando en diferentes funciones, pueden surgir muchos problemas de compatibilidad cuando se combinan varias funcionalidades al mismo tiempo. La integración continua permite a los equipos abordar los pequeños conflictos a medida que surgen.
- Debido a que las pruebas unitarias se ejecutan con cada combinación, cuando una prueba falla, es más fácil aislar la parte del código que está causando el problema.
- La publicación frecuente de nuevas versiones de una aplicación permite a los desarrolladores aislar los problemas si surgen después del lanzamiento.
- La publicación de cambios pequeños e incrementales permite a los usuarios acostumbrarse lentamente a las nuevas funciones de la aplicación en lugar de sentirse abrumados con una versión completamente diferente.
- No esperar a lanzar nuevas funciones permite a las empresas mantenerse a la vanguardia en un mercado competitivo.