1. Introducción

TDD, o Desarrollo Dirigido por Pruebas (*Test-Driven Development*), es una práctica de desarrollo de software que consiste en escribir primero las pruebas (generalmente unitarias), luego escribir el código fuente que pase la prueba satisfactoriamente, y finalmente refactorizar el código escrito.

Con esta práctica se logra, entre otras cosas, un código más robusto, seguro, y mantenible, así como una mayor rapidez en el desarrollo. En contraposición, escribir mucho código y luego testearlo (o no testearlo en absoluto) retrasa significativamente el desarrollo general de cualquier sistema (habitualmente por la aparición de *bugs* y fallas) y produce software de peor calidad que suele ser difícil de mantener.

La idea principal es escribir primero la prueba, incluso sin escribir una sola línea de código (la cual fallará). Luego se escribe el mínimo código indispensable para que pase esta prueba. Como último paso, se *refactoriza* el código (optimizándolo, mejorándolo, corrigiéndolo, etc.) y se verifica que pasen todas las pruebas escritas anteriormente. Si el código satisface al desarrollador, se vuelve a empezar con un nuevo test. Siempre deben pasar todas las pruebas anteriores para asegurarse de que lo agregado como nueva funcionalidad no genere conflictos con lo que ya existe.

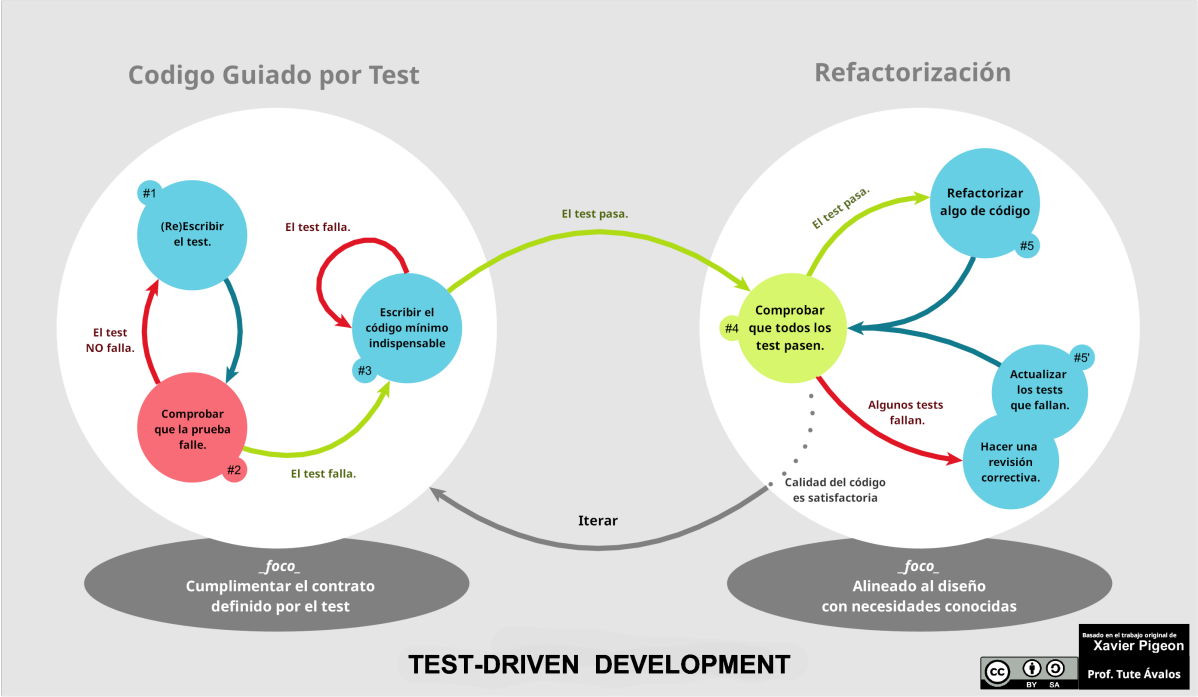
1.1. Ciclo iterativo de desarrollo

En esta metodología, hay dos etapas principales. La primera, como se ha mencionado, tiene el enfoque en generar código que cumpla con el **contrato definido por la prueba**. Para ello, se debe asegurar que este contrato esté bien definido y que falle inicialmente, ya que la nueva característica aún no existe. En la Figura 1 se observa cómo se avanza desde la creación del nuevo test `#1` hasta su verificación `#2`, mientras este no falle, lo cual sería el primer indicador de que la prueba es incorrecta.

Una vez definido el test, y después de verificar que efectivamente ha fallado, se procede a escribir el código fuente que cumpla con dicho contrato `#3`, es decir, que pase la prueba. Hasta que esto no ocurra, se debe seguir agregando o modificando código.

Después de cumplir con el contrato del test, se pasa a la segunda etapa, que tiene el enfoque en **el diseño de software** y la calidad del código.

Figura 1: Ciclo de vida general de TDD



En esta etapa se debe verificar que ningún otro test falle `#4`. Si las pruebas pasan, se puede proceder a refactorizar el código `#5`. Esto implica normalmente:

- Mover el código a la parte que lógicamente corresponda.
- Eliminar código repetido.
- Utilizar nombres autoexplicativos.
- Separar funcionalidades en unidades más pequeñas o atómicas.
- Restructurar para cumplir con los principios de diseño de software.

Es importante destacar que ningún software serio debería comenzar a codificarse sin haber realizado previamente un diseño que modele la solución del mismo. Esta segunda etapa depende de ello.

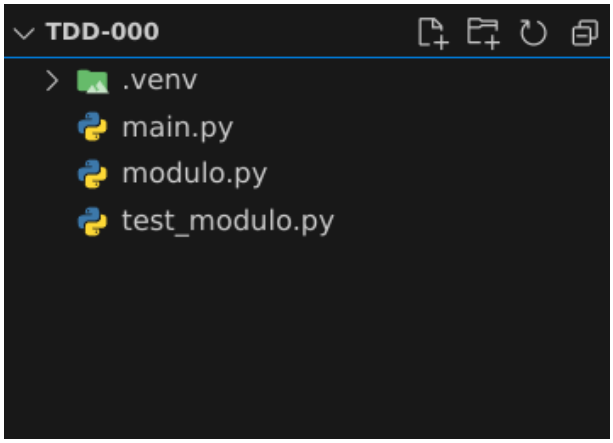
Si luego de refactorizar, por algún motivo (cambios de nombre, cambio de algoritmo, etc.), alguno de los tests falla, se debe realizar una revisión del código y/o ajustar los tests de las funciones modificadas para lograr que pasen nuevamente `#5'`. Una vez que

el software se ajusta al diseño y satisface los estándares de estilo, documentación y otros aspectos, se retoma desde el punto `#1` para agregar una nueva funcionalidad al sistema.

2. Metodología utilizando `Python` y `pytest`

Retomando lo visto en la guía anterior, se debe agregar al proyecto el módulo de `pytest`. En general, una aplicación estará compuesta de varios módulos, los cuales interactúan entre sí para hacer a la funcionalidad completa de la aplicación.

Un proyecto en general tendrá la siguiente estructura:



Cada módulo que se agregue tendrá asociado su archivo de `test_<modulo>.py` donde se agregarán las pruebas unitarias a medida que se general el código que debió ser *diseñado con anterioridad*.

2.1. Escribir el test

Supongamos que el módulo se llama `matematica.py` y tiene operaciones básicas. Este módulo debería tener las funciones `suma, resta, multi y divi`. Primero escribiremos los test que verifica la existencia de cada una de estas, empezando por `suma`, en el archivo `test_matematica.py`.

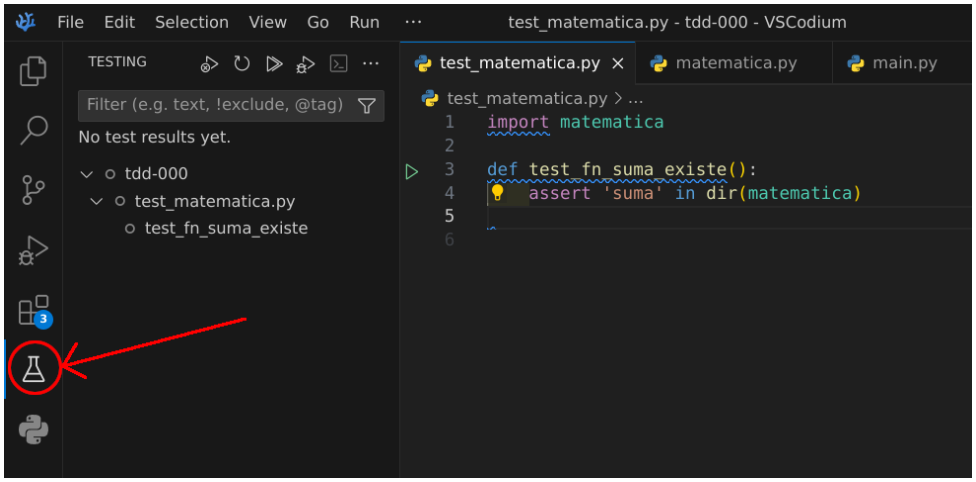
```python
import matematica

def test_fn_suma_existe():
    assert 'suma' in dir(matematica)
```

2.2. Hacer pasar el test

Debemos introducir el código mínimo e indispensable para hacer pasar el test, en este caso es simplemente crear la función dentro de `matematica.py`.

```
1 def suma():
2   pass
```

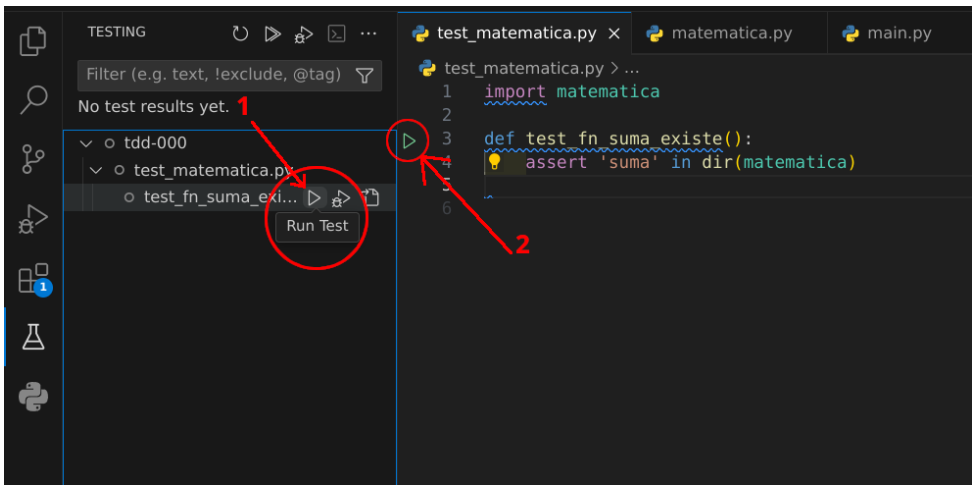
Para probar los test podemos ir en la barra lateral izquierda al ícono de **Test**.



Para correr el test tenemos dos opciones:

1. Ir al menú de test como se mostró recién y correr los test deseado haciendo clic en el play correspondiente.

2. Ir al archivo del test y hacer clic en el play verde a la izquierda de la función del test correspondiente para correrlo.

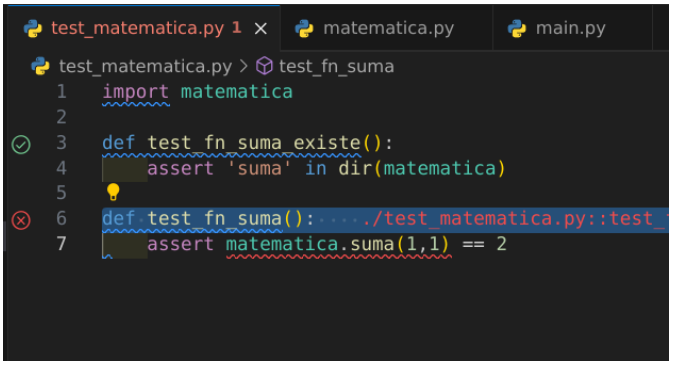


2.3. *Refactorizar* y correr los tests (de nuevo)

En el ejemplo anterior no hay mucho que probar, ya que no se probó la funcionalidad, para ello vamos a iterar sobre esta función sencilla para entender el procedimiento completo.

2.3.1. Creamos un test y lo corremos

En este ejemplo crearemos un test unitario que compruebe la suma:

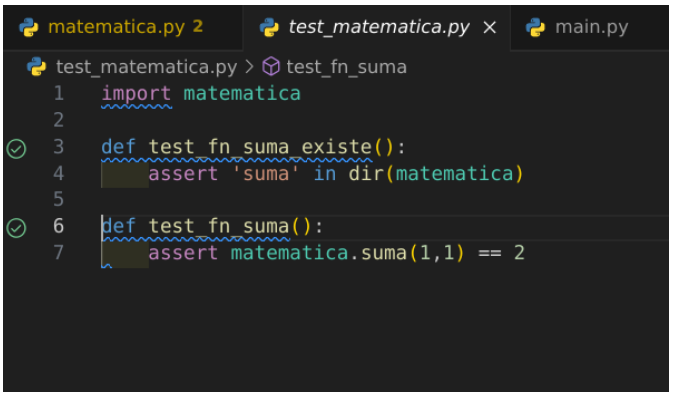


En este caso, como podemos corroborar el test falla porque `suma` no solo no recibe parámetros, sino que además no devuelve ningún valor.

2.3.2. Escribir el código mínimo para que pase la función

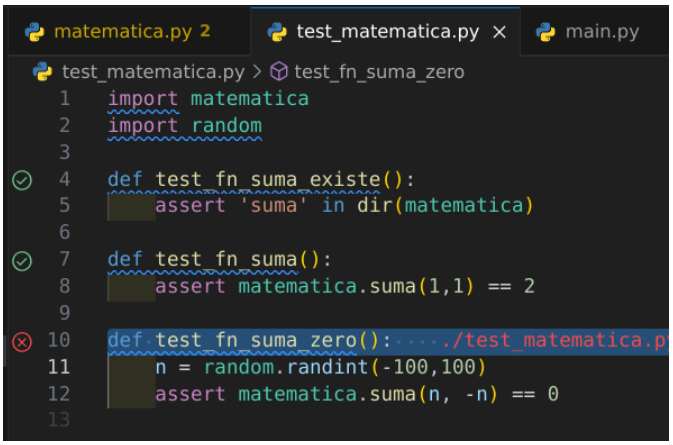
Una solución poco elegante, pero que cumple con el contrato de test es la siguiente:

```
1 def suma(a, b):
2     return 2
```



2.3.3. Segunda iteración

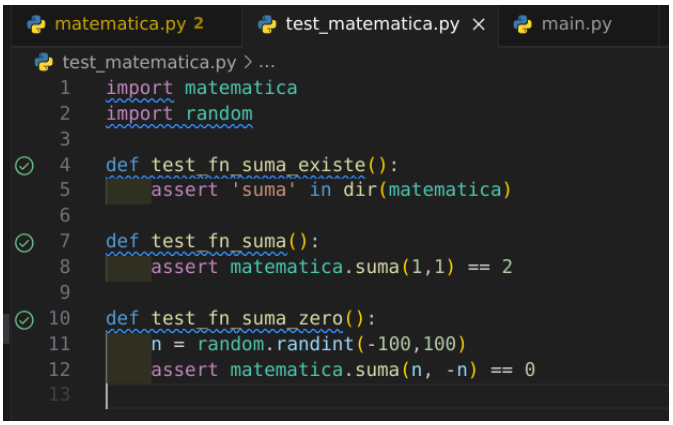
Ahora se agrega un test que suma un número al azar y su complemento, es decir, una suma que siempre de el valor cero.



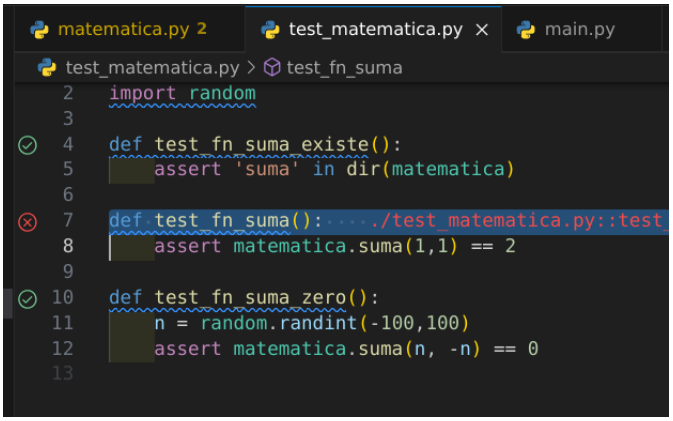
Una en este caso se opta por hacer el siguiente cambio para que el test pase:

```
1 def suma(a, b):
2     return 0
```

Y efectivamente, el test pasa:



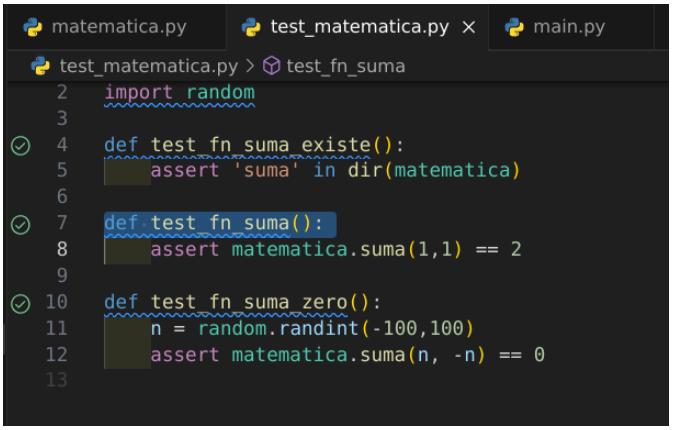
Pero como podrá advertir fácilmente, al volver a pasar todos los tests, esto hace que falle el anterior:



Entonces, es momento de refactorizar la función `suma` para poder hacerla pasar todos los tests.

```
def suma(a, b):
    return a + b
```

Una vez más corremos todos los tests.



2.3.4. tercera iteración

Siempre que se pueda debe asegurar que la función sea lo suficientemente robusta para poder operar correctamente en toda situación posible, incluso *remotamente posible*. Por ejemplo, ¿Funciona para cualquier tipo de número?¿qué ocurre si envían un `str` en lugar de un número?

```python
def test_fn_suma_float():
    assert matematica.suma(1.1, 1.1) == 2.2

def test_fn_suma_str_int():
    assert matematica.suma("1", "1") == 2

def test_fn_suma_str_float():
    assert matematica.suma("1.1", "1.1") == 2.2

# ...
```

Así podremos asegurarnos de que nuestro software tenga la calidad que requiere nuestra aplicación.