TDD y Diseño Simple - Kent Beck

Autora: Rojas Córsico, Ivana Soledad

Introducción

En el mundo del desarrollo de software, dos aspectos fundamentales se destacan: el diseño y las pruebas. Ambos son pilares esenciales para crear aplicaciones robustas y mantenibles.

En este documento, abordaremos inicialmente TDD (Test Driven Development) a fin de escribir las pruebas antes de comenzar a escribir código fuente. Esto nos ayuda a pensar en los casos límite, los escenarios de error y los comportamientos esperados antes de escribir el código fuente. Luego, abordaremos las cuatro reglas del diseño simple propuestas por Kent Beck que son, en orden de importancia: pasa las pruebas, revela intención, sin duplicación y con el menor número de elementos siendo las tres últimas las aplicadas al momento de realizar la refactorización de código.

Más allá del código.

El desarrollo de software va más allá de escribir código. Es necesario un diseño de software que defina la estructura subyacente que determine cómo se organizan los componentes, cómo interactúan entre sí y cómo se resuelven los problemas. Un buen diseño facilita la comprensión, la extensibilidad y la evolución del sistema.

El Desarrollo Guiado por Pruebas (TDD) y las pruebas unitarias son prácticas cruciales. Sin embargo, no son suficientes por sí solas. El diseño adecuado es la base sobre la cual construimos pruebas efectivas. Si el diseño es deficiente, incluso las pruebas más exhaustivas pueden no detectar problemas. Para evitar la complejidad innecesaria, debemos seguir las cuatro reglas del diseño simple propuestas por Kent Beck y los principios SOLID (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation y Dependency Inversion). El mismo, se abordará en un próximo documento.

TDD (Test Driven Development)

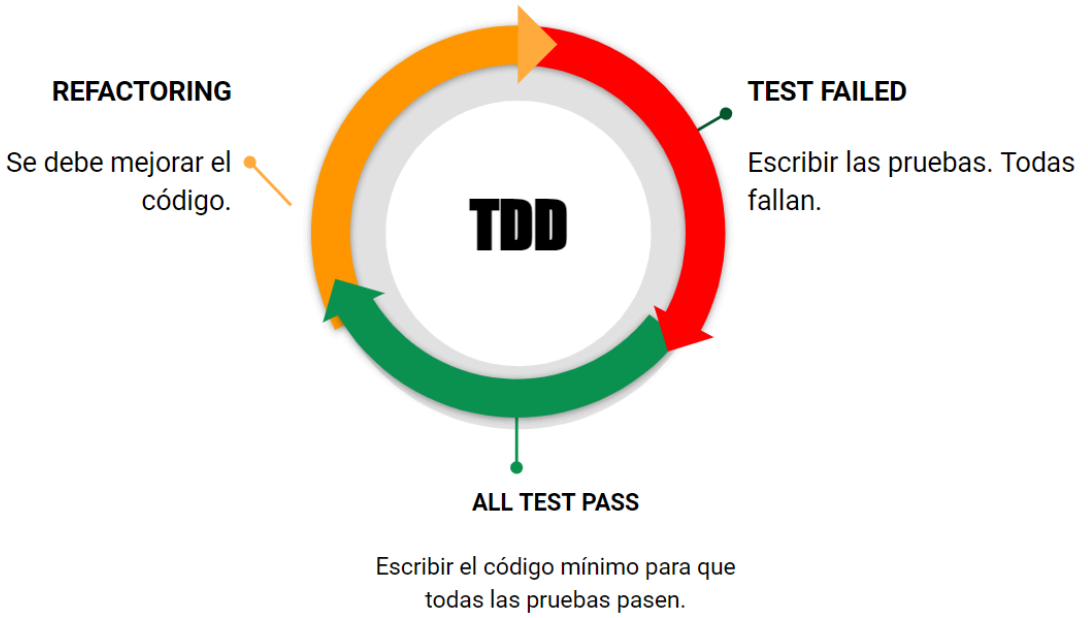
El Test Driven Development (TDD), o desarrollo guiado por pruebas, es un enfoque en el cual se crean pruebas unitarias antes de escribir el código de la funcionalidad. El mismo combina dos metodologías: *Test-first development* (escribir las pruebas primero) y *refactoring* (refactorización de código).

TDD tiene por objetivo es lograr un código limpio, robusto y simple.

El mismo fue propuesto por Kent Beck en la década de 1990 en su libro “Extreme Programming Explained”. En este punto es importante agregar que, si bien Kent Beck fue uno de los defensores de la idea de escribir pruebas de código previo a la codificación, ésta no era completamente nueva. Sin embargo fué él quien la formalizó y estructuró su enfoque.

A continuación, se enumera el proceso para realizar TDD:

1. Escribir una prueba que falle (es decir, que aún no pase).
2. Escribir el código mínimo necesario para que la prueba pase.
3. Refactorizar el código fuente para hacerlo más limpio, genérico y apropiado.



Beneficios de TDD:

- Se entiende que la función es correcta dado que las pruebas lo verifican.
- Obliga a pensar en los requisitos antes de escribir el código, lo que conduce a un diseño más limpio y modular.
- Reduce temprana de errores en el código existente.
- Las pruebas actúan como documentación activa dado que permite observar ejemplos de uso. También brinda un conjunto completo de pruebas de regresión una vez finalizado.

Herramientas para realizar una pruebas con Python

Existen varias herramientas para realizar pruebas unitarias con Python. Entre ellas, podemos nombrar:

- **unittest.** Es un módulo de pruebas integrado en Python.

Ejemplo:

```python
# mymodule.py
def sum(num1, num2):
    if not isinstance(num1, (int, float)) or not isinstance(num2, (int,
float)):
        raise TypeError("Ambos números deben ser del tipo int o float.")
    return a + b
```

```python
# test_mymodule.py
import unittest
import mymodule

class TestMyModule(unittest.TestCase):
    def test_sum(self):
        self.assertEqual(mymodule.sum(5, 7), 12)
```

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

Para más información puedes acceder a la fuente oficial. Link: https://docs.python.org/3/library/unittest.html

- **pytest**. Es una biblioteca para pruebas de Python. Debe instalarse ejecutando **Pip install pytest** en el entorno de desarrollo.

Ejemplo:

```
# test_mymodule.py
import mymodule

def test_sum():
    assert mymodule.sum(5, 7) == 12
```

Para más información, puedes acceder a la fuente oficial. Link: https://docs.pytest.org/en/8.2.x/

Resolviendo el kata FizzBuzz

Para comprenderlo mejor resolveremos un kata de código muy conocido denominado FizzBuzz.

Un **kata de código** en la industria de desarrollo de software es un ejercicio diseñado para que los programadores desarrollen y mejoren sus habilidades en programación mediante la práctica y la repetición. Es importante agregar que no se basa únicamente en aprender un concepto de programación sino acceder al conocimiento correcto, entenderlo y pulirlo.

El kata denominado FizzBuzz consiste en escribir una función que tome un número entero positivo y retorne su representación en cadena pero, respetando las siguientes reglas:

- Si el número es múltiplo de 3, imprime “Fizz” en lugar del número.
- Si el número es múltiplo de 5, imprime “Buzz” en lugar del número.
- Si el número es múltiplo de ambos 3 y 5, imprime “FizzBuzz”.

Sin seguir los principios de TDD, probablemente lleguemos a un resultado similar al siguiente:

```
def fizzbuzz(num):
     if num % 15 == 0:
        print("FizzBuzz")
     elif num  % 5 == 0:
        print("Buzz")
     elif num  % 3 == 0:
```

```
        print("Fizz")
    else:
        print(i)
```

Pero ¿es posible testear esta función de manera automática? ¿Cuál es el problema? Intenta responder antes de continuar con la lectura.

Veamos entonces, cómo resolver siguiendo los principios de TDD.

Primor Etapa

A) ROJO - Escribir la primer prueba fallida

Para comenzar, consideramos la prueba de unidad más simple. Por ej. sabemos que para el número 1, el resultado de la función fizzbuzz(n) será "1". Por lo tanto, comenzamos creando la primera prueba.

```
# test_fizzbuzz.py
from fizzbuzz import FizzBuzz

def test_fizzbuzz_1():
    my_fizzbuzz= FizzBuzz(1)
    assert my_fizzbuzz.convert_to_fizzbuzz() == "1"
```

En nuestro caso utilizaremos la librería de pruebas unitarias **pytest** y la extensión **"Pytest Runner for Visual Studio Code"**.

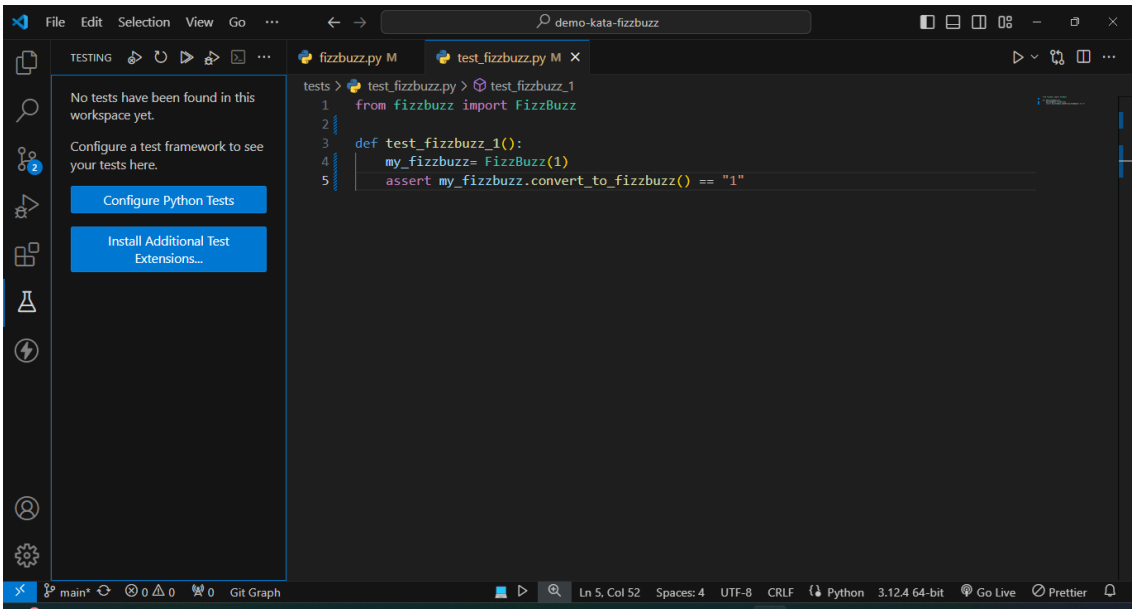
Como es sabido, esto fallará al compilar porque no existe el módulo fizzbuzz. Ahora escribimos el código fuente para que compile.

```
# fizzbuzz.py
class FizzBuzz:
    def __init__(self, num):
        self.num=num
    def convert_to_fizzbuzz(self):
        raise NotImplemented("Este método debe ser implementado")
```

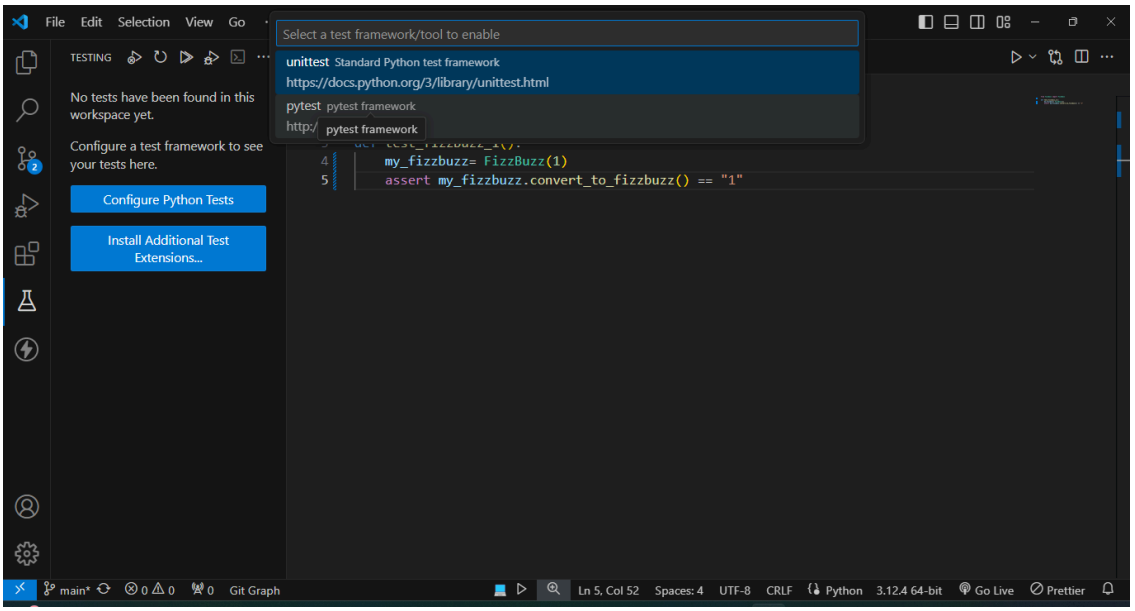
Para ejecutar las pruebas utilizando podemos:

- Ejecutar el comando **pytest test_fizzbuzz.py**
- o bien, utilizar la extensión ***"Pytest Runner for Visual Studio Code".*** Para ello, configurar las pruebas como sigue:

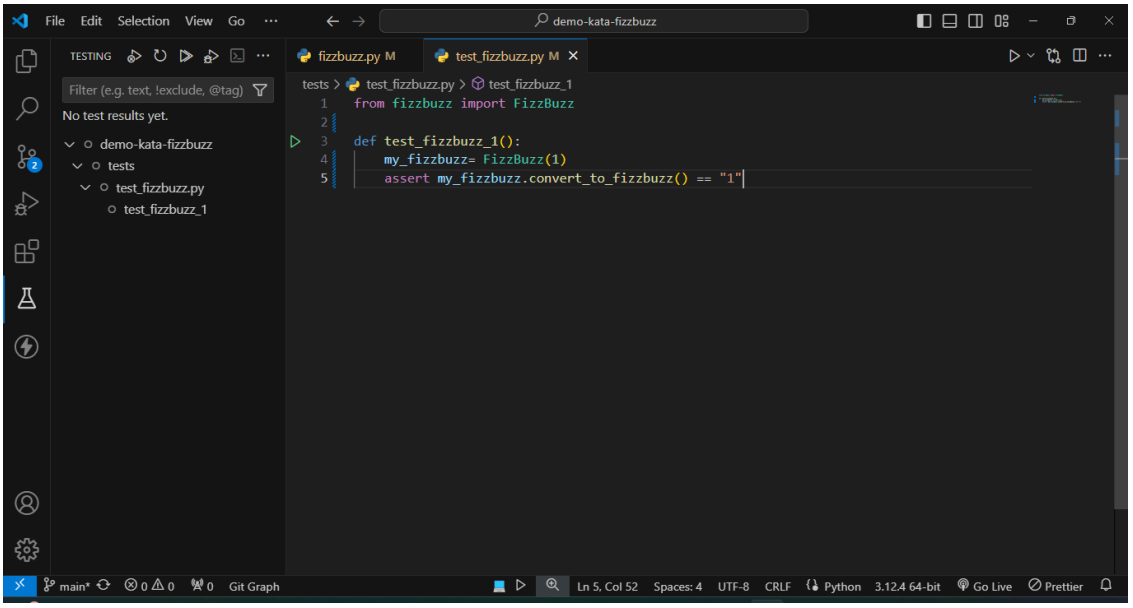
- Seleccionar el icono **Testing**, ubicado en el menú lateral izquierdo.
- Luego, hacer click en el botón **Configure Python Tests**



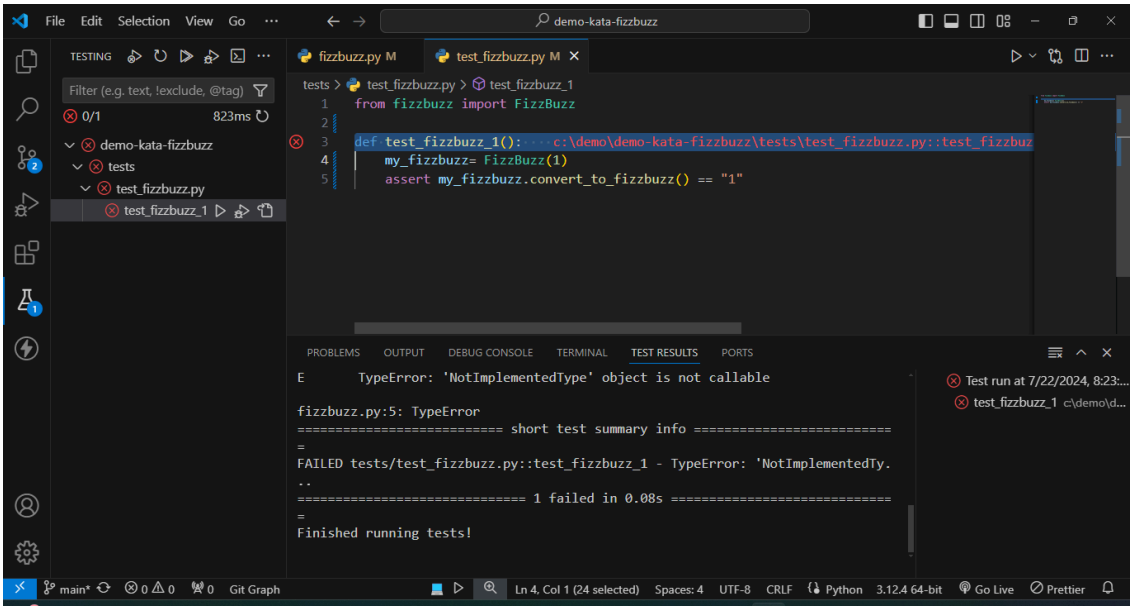
y seleccionar la opción **pytest pytest framework** como sigue:



A continuación, seleccionar la ruta dónde se encuentran las pruebas. Accederás a la siguiente ventana:



Finalmente, ejecutar la prueba haciendo click en el botón play como sigue:



Como puedes observar, la prueba falla porque no está implementado el método.

B) VERDE - **Escribir código para pasar la prueba**

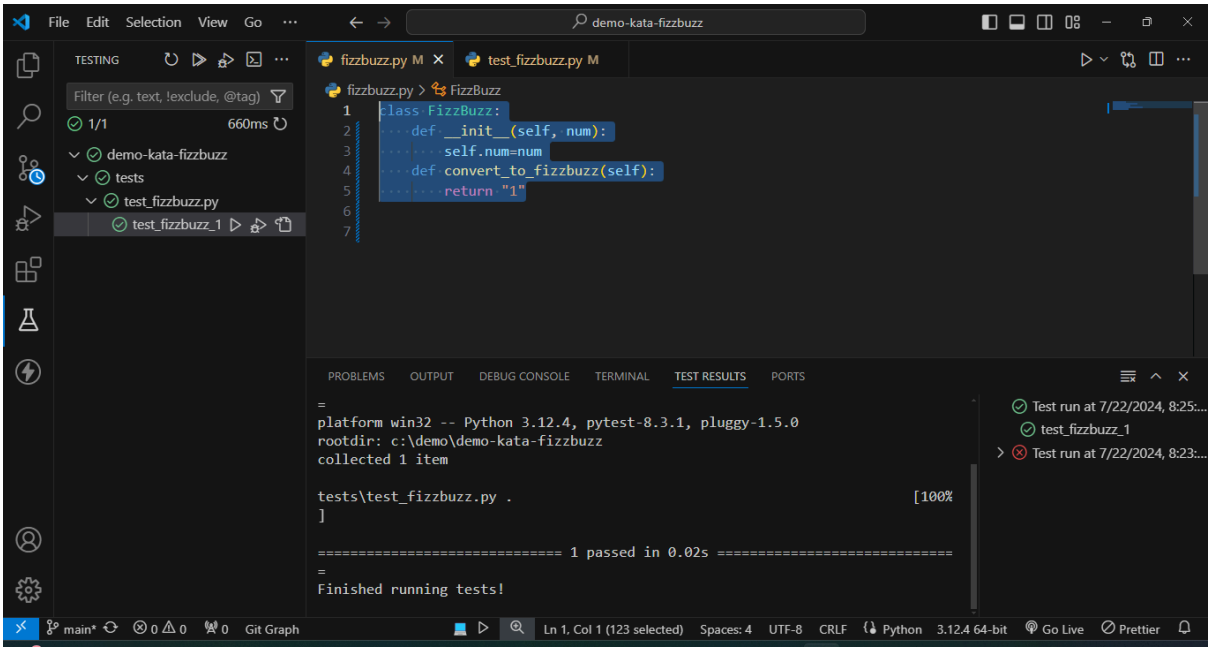
Para ello, aplicaremos el primer principio de TDD *"No debes escribir ningún código de producción a menos que sea para pasar una prueba fallida"*

Para ello, modificamos nuestro método ***convert_to_fizzbuzz*** como sigue:

```
# fizzbuzz.py
class FizzBuzz:
```

```
    def __init__(self, num):
        self.num=num
    def convert_to_fizzbuzz(self):
        return "1"
```

Ejecutando las pruebas:



Esto se conoce como *"fingir"*. Puede que te sientas desconcertado respecto a esta solución en este momento e incluso quizás sientas la necesidad de incluir más código para más pruebas, pero recuerda! TDD nos sugiere concentrarnos en la solución más sencilla posible dado que, de esta manera, surgirá una solución limpia y elegante a medida que se añaden más pruebas. Incluso Kent Beck dijo *"Fingelo hasta que lo consigas"*.

C) AMARILLO - **Refactorización**

Como podemos observar, no hay código para refactorizar. ¡Vamos a la segunda etapa!

Segunda Etapa

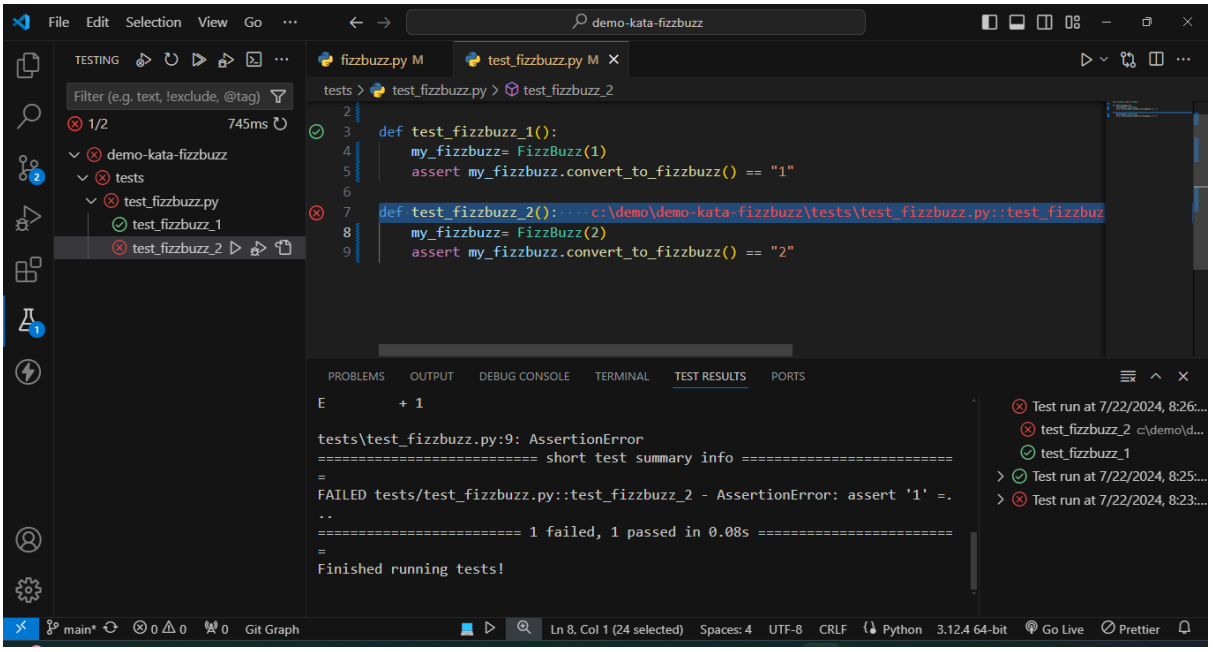
A) ROJO - **Escribir la segunda prueba fallida**

En este punto, debemos decidirnos por la siguiente prueba. Se sugiere entonces, desglosar el problema en problemas más pequeños. En este caso realizaremos una prueba que responda a: *"escribir una función que tome un número entero positivo y retorne su representación en cadena"*.

Entonces, creamos una prueba para el número 2 dónde su representación en cadena debería ser “2” como sigue:

```python
# test_fizzbuzz.py
def test_fizzbuzz_2():
    my_fizzbuzz= FizzBuzz(2)
    assert my_fizzbuzz.convert_to_fizzbuzz() == "2"
```

Luego, ejecutamos nuevamente las pruebas.



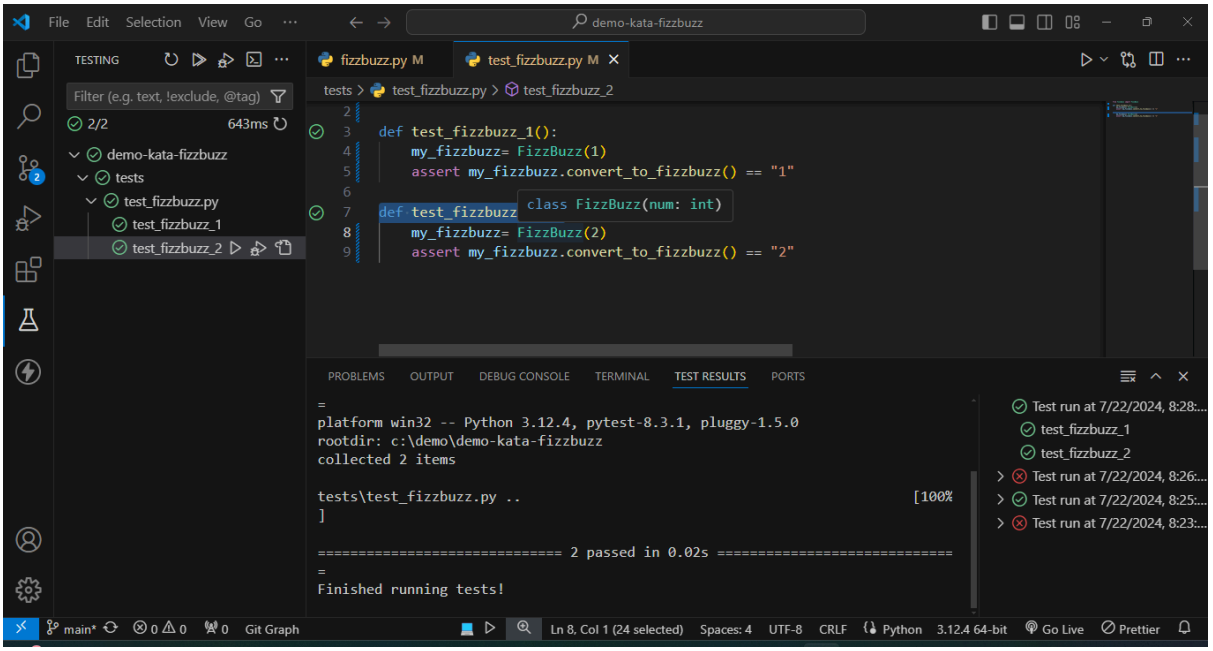
Como puedes observar, falla la segunda prueba dado que esperaba “2” en lugar de “1”.

B) VERDE - Escribir código para que pase la prueba.

Para pasar las pruebas, modificamos el código del método **convert_to_fizzbuzz** como sigue:

```python
# fizzbuzz.py
class FizzBuzz:
    def __init__(self, num):
        self.num=num
    def convert_to_fizzbuzz(self):
        return str(self.num)
```

Luego, ejecutamos nuevamente las pruebas:



Como puedes observar, ahora ambas pasan.

C) AMARILLO - **Refactorización**

Como podemos observar, puede que haya algo para mejorar, pero lo dejaremos para más adelante. ¡Vamos a la tercera etapa!

Tercer Etapa

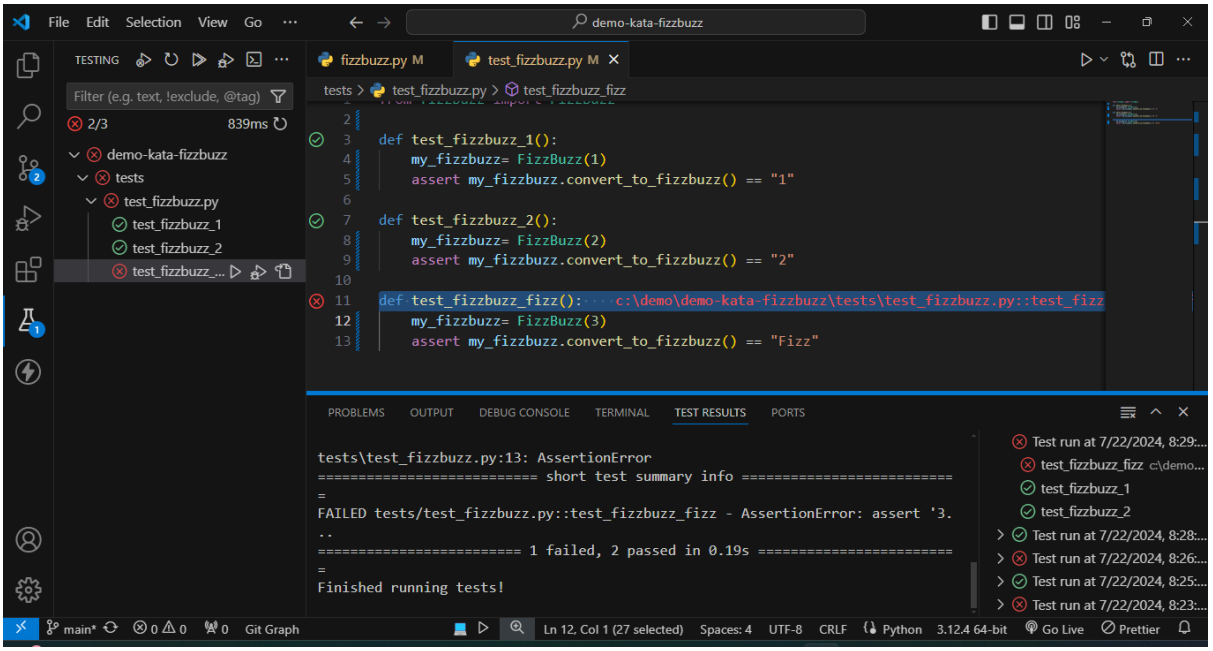
A) ROJO - **Escribir la tercera prueba fallida.**

Nuevamente nos toca decidir una nueva prueba. En este caso, vamos a crear una prueba que responda a la siguiente regla: “Si el número es múltiplo de 3, imprime “Fizz”.

Para ello, agregamos la siguiente prueba:

```
# test_fizzbuzz.py
def test_fizzbuzz_fizz():
    my_fizzbuzz= FizzBuzz(3)
    assert my_fizzbuzz.convert_to_fizzbuzz() == "Fizz"
```

Luego, ejecutamos las pruebas como sigue:



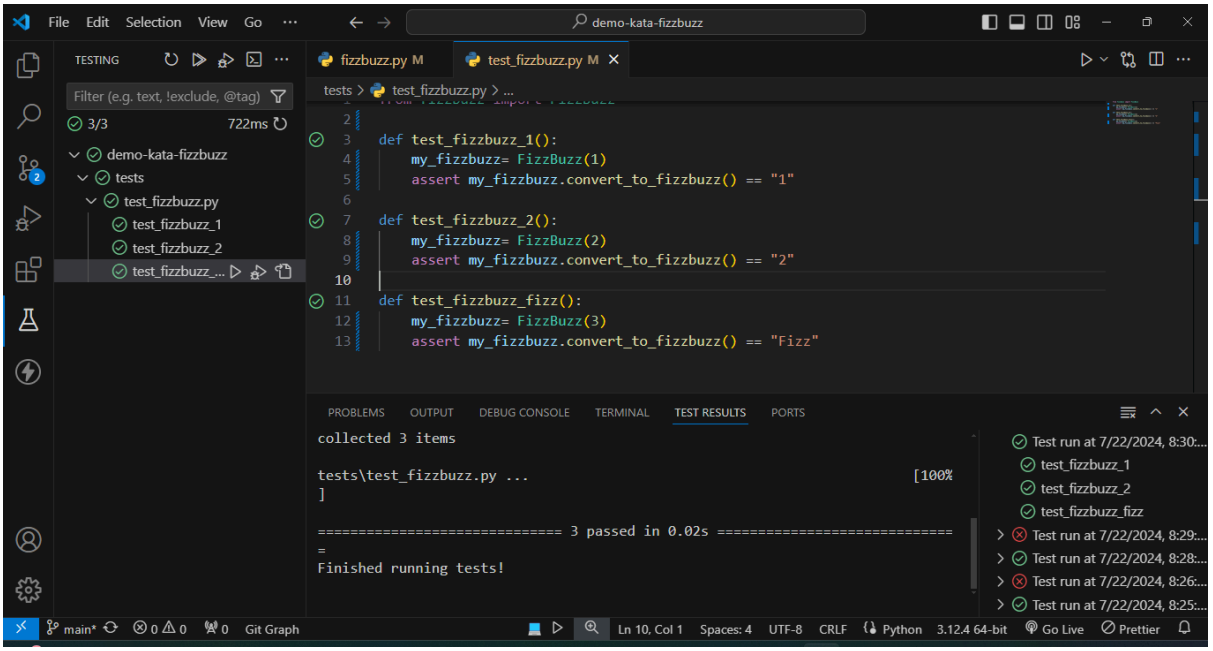
Como puedes observar, la tercera prueba falla.

B) VERDE - **Escribir código para que pase la prueba**

Nuevamente, para pasar las pruebas modificamos el código para pasar las pruebas como sigue:

```python
# fizzbuzz.py
class FizzBuzz:
    def __init__(self, num):
        self.num=num
    def convert_to_fizzbuzz(self):
        if (self.num % 3 == 0):
            return "Fizz"
        return str(self.num)
```

Luego, ejecutamos nuevamente las pruebas:



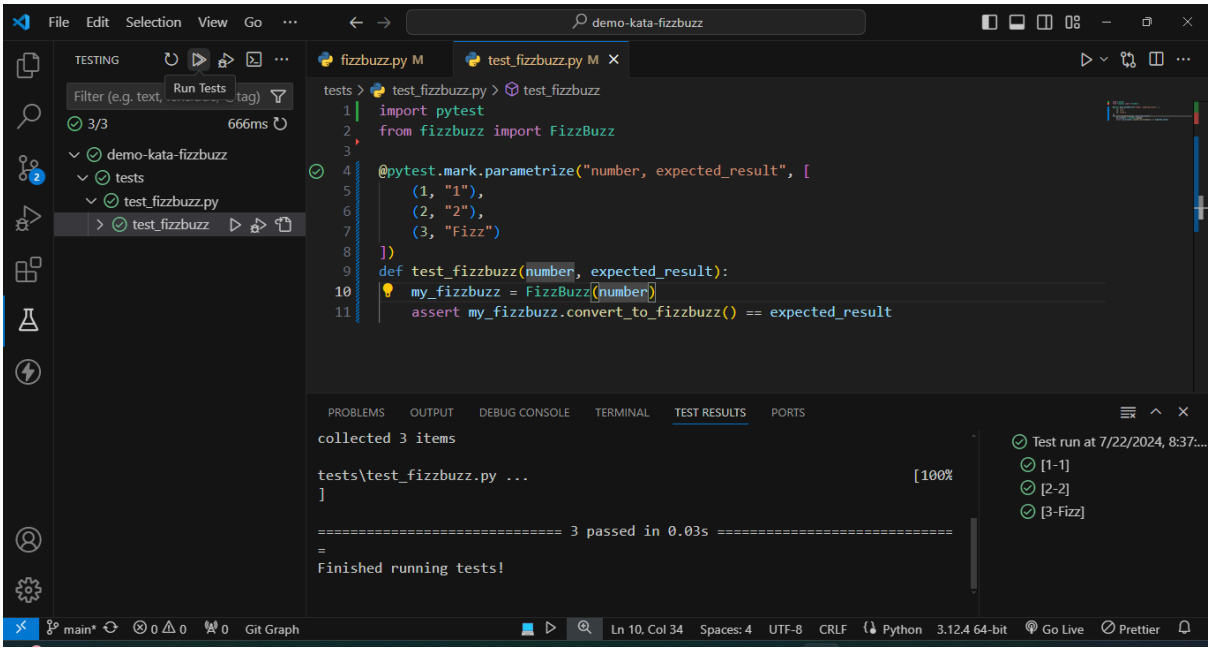
C) AMARILLO - **Refactorizar.**

En este punto podemos refactorizar evitando la repetición de las pruebas (dado que sólo cambian los valores) utilizando parametrización de pruebas que nos provee pytest como sigue:

```python
# test_fizzbuzz.py
import pytest
from fizzbuzz import FizzBuzz

@pytest.mark.parametrize("number, expected_result", [
    (1, "1"),
    (2, "2"),
    (3, "Fizz")
])

def test_fizzbuzz(number, expected_result):
    my_fizzbuzz = FizzBuzz(number)
    assert my_fizzbuzz.convert_to_fizzbuzz() == expected_result
```

Observa que estamos refactorizando el código fuente relacionado a las pruebas. En este punto es importante agregar que, es importante refactorizar las pruebas también ya que, significan un costo para el desarrollo.

Finalmente, evaluamos que todas las pruebas pasen:



¿Es posible refactorizar algo más? Si consideras que sí, realiza los cambios pertinentes antes de seguir con la lectura.

¡Recuerda! Las refactorizaciones son cambios en el código fuente que NO cambian su comportamiento.

¡Pongámonos a prueba! *Intenta completar el resto del ejercicio a fin de realizar la implementación de Buzz para números que sean múltiplos de 5 y, la implementación de FizzBuzz para números múltiplos de 3 y 5.*

Puedes descargar el código fuente implementado hasta el momento haciendo clic aquí.

En este punto, es crucial destacar que la creación de sistemas testeables nos brinda varias ventajas. Nos permite diseñar clases de tamaño reducido con baja dependencia y alta cohesión. Además, podemos construir clases que cumplan tanto el principio de Responsabilidad Única (SRP) como con el principio de Inversión de Dependencias (DIP)

¡Investiga! De qué se tratan estos principios y a continuación, verifica que el código fuente generado a través de TDD para resolver el kata kizzbuzz lo cumple. En caso de no cumplir, modifica a fin de mejorar el código fuente.

¡Más Katas de código para resolver! A continuación te invitamos a intentar resolver los siguientes katas aplicando TDD:

- String Calculator
- Leap Years (año bisiesto)
- Roman Numeral
- Bowling

Accede al siguiente link para encontrar estos y otros katas de código para resolver. Link: https://codingdojo.org/kata/

Las reglas de Diseño Simple según Kent Beck

A menudo comenzamos a escribir código fuente sin detenernos a pensar en el diseño por lo que, creamos código difícil de testear y muchas veces difícil de mantener. Para resolver el problema, Kent Beck propone las reglas del diseño simple. Las mismas consisten en simplemente dejar de escribir líneas de código y dejar más tiempo al diseño respondiendo preguntas tales cómo: ¿Podemos hacer lo mismo sin esa herencia?, ¿y sin inyección de dependencias?, ¿con menos clases?, ¿con menos métodos?, ¿con menos variables?, ¿con menos bucles?, ¿con menos condicionales?, etc.

Las reglas de diseño simple según Kent Beck

- **El código fuente pasa las pruebas.** La funcionalidad debe estar probada y funcionando correctamente. Es fundamental que nuestro código pase las pruebas unitarias y de integración.
- **Revela la intención del programador.** El código debe ser expresivo y comunicar claramente su propósito. Los nombres de variables, funciones y clases deben ser descriptivos y revelar su intención.
- **No hay duplicados**. Evita la repetición innecesaria de código. La duplicación dificulta el mantenimiento y aumenta la posibilidad de errores.
- **Tiene el menor número posible de elementos.** Busca la simplicidad eliminando lo superfluo. No añadas componentes o características innecesarias al diseño.

¡Recuerda! Las mismas están en orden de prioridad.

Estas reglas son importantes para escribir un código fuente fácil de mantener. Es decir que podremos cambiarlo, actualizarlo, añadirle o quitarle funcionalidad y por supuesto, corregir fallos. Cuanto más fácil sea mantener el mismo, más sencillo será entenderlo y por ende más rápido podremos adaptarlo a nuestras necesidades.

Por otra parte, es importante agregar que, aunque el código respete estas cuatro reglas, no garantiza que otros puedan continuar con el producto y la complejidad inherente al problema persiste. Sin embargo, es la estrategia más mantenible conocida.

El código pasa los pruebas

Un diseño debe poder permitir la implementación de un sistema que actúe de la forma prevista. Por ende, si un sistema tiene un diseño perfecto sobre el papel, pero no existe una forma sencilla de comprobar si realmente funciona de la forma prevista, entonces no debería implementarse.

Entonces, desde esta perspectiva, podemos pensar que el tiempo que lleva asegurar que las pruebas pasen, puede ser un factor significativo para poder realizar los cambios. Es decir, que mientras más rápido tengamos los resultados de las pruebas, más rápido podremos realizar las correcciones pertinentes en el caso que fallen. Es por ello que, al considerar una estrategia de testing, es recomendable inclinarse por las pruebas automatizadas.

A continuación, se enumeran algunas consideraciones al realizar pruebas:

- Incluye pruebas basadas en estados (State-Based Testing). En estas pruebas verificamos que el código sujeto a evaluación nos devuelva resultados correctos. No nos concentramos en el comportamiento ni en las propiedades y valores internos del objeto.

  Ejemplo:

  ```
  def test_es_par():
      assert es_par(2) == True
  ```

- Incluye pruebas basadas en el comportamiento (Behavior-Based Testing). En estas pruebas nos enfocamos en verificar que el código sujeto a evaluación interactúe con su entorno y responda con los comportamientos esperados. Es decir que interesa evaluar cómo se comporta en situaciones específicas. El objetivo de este tipo de pruebas es centrarse en escenarios posibles.

  Ejemplo:

  ```
  def test_dos_strike(juego):
      juego.lanzar(10)  # Strike
      juego.lanzar(10)  # Strike
      juego.lanzar(1)
      juego.lanzar(2)
  ```

```
    for turno in range(8):
        for lanzamiento in range(2):
            juego.lanzar(1)
    assert juego.puntuacion() == 51
```

- Evitar pruebas que dependan de otras pruebas. Cada prueba debe configurarse, ejecutarse de manera aislada dado que si una prueba falla, es útil observar que las demás puedan o no fallar. Esto nos dará una idea del alcance del problema causado por el cambio en el código fuente.
- Intenta cubrir la mayor cantidad de código posible. No es necesario cubrir todas las combinaciones posibles pero sí las más relevantes. No sacrifiques la eficiencia por una cobertura excesiva.
- Escribe pruebas fáciles, claras y fáciles de leer.
- Escribe pruebas para situaciones límites y extremas. Ej. ¿cómo se comporta con valores mínimos y máximos? Considera casos de entradas vacías, nulas o inesperadas.
- Automatiza las pruebas a fin de que se ejecuten en el proceso de integración continua.

Revela la intención del programador

El mantenimiento a largo plazo es el principal costo de un proyecto de software. A medida que los sistemas se vuelven más complejos, se requiere más tiempo para comprenderlos, lo que aumenta las posibilidades de errores. Para mitigar esto, es crucial que el código exprese claramente la intención del autor. La claridad en el código reduce el tiempo que otros emplean en entenderlo. Algunas formas de revelar la intención del programador son: elegir nombres adecuados para clases y funciones, reducir el tamaño de funciones y clases para facilitar su comprensión, crear pruebas que sirvan como documentación mediante ejemplos, y dedicar tiempo a mejorar la legibilidad del código en lugar de apresurarnos al siguiente problema de codificación ha resolver.

Principio de Menor Sorpresa

El Principio de la Mínima Sorpresa sugiere que el comportamiento de un programa debe ser predecible y coherente. Tanto los usuarios como otros desarrolladores no deberían verse sorprendidos por el funcionamiento de una función o método. Para aplicarlo, se sugiere:

- Asigna nombres descriptivos a las funciones y variables para que su propósito sea evidente.
- Sigue las convenciones del lenguaje y de la comunidad para evitar sorpresas inesperadas.

- Organiza el código en módulos o paquetes lógicos (quizás siguiendo algún patrón de arquitectura de software reconocido. Si estás iniciando, se sugiere comenzar con el patrón de arquitectura en capas: Presentación, Lógica de Negocio y Acceso a Datos)
- Documenta cualquier comportamiento inusual o excepcional que pueda surgir.

¿Conoces alguna convención de nombres de la comunidad de Python? Enumera y explica antes de continuar con la lectura.

Técnicas para los nombres

A continuación, listamos algunas técnicas para los nombres de elementos en Python:

- Utiliza nombres descriptivos y pronunciables. Opta por nombres claros y fáciles de pronunciar.
- Evita utilizar nombres que incluyan información técnica. No utilices la jerga técnica en los nombres.
- Evita alias. No recurras a nombres ambiguos o abreviaturas que puedan confundir a otros desarrolladores.
- Evita abreviaturas. Prioriza la claridad.
- Considera el contexto. Elige nombres que tengan sentido dentro del contexto específico en el que se utilizan.
- Distingue entre sustantivos y verbos. Utiliza sustantivos para clases y objetos y verbos para funciones y métodos.
- Respeta las sugerencias del PEP 8.

Guía PEP 8

Guido van Rossum, el creador de Python, nos propone una guía de estilos para escribir código en el lenguaje de programación Python titulada Pep 8. La misma, es ampliamente utilizada y aceptada por la comunidad de desarrolladores de Python como una forma de escribir código limpio.

Nombres de Variables y funciones

Se recomienda:

- Los nombres de las variables y funciones deben ser descriptivos, en minúsculas, separando las palabras por guión bajo.
- Evitar nombres de una sola letra, a menos que sean contadores. Ej. i en el for.

Ejemplo:

```
# Wrong
e=15

# Correct
```

```
edad=15
```

Identación

Se recomienda:

- Usar 4 espacios por nivel de identación (no tabulación).
- Evitar espacios en blanco al final de las líneas.
- Mantener la consistencia en la identación a lo largo del código.

Ejemplo:

```
# Wrong
if numero % 2 == 0:
    print("Es Par")
   print(f"El número {numero} es par")

# Correct
if numero % 2 == 0:
    print("Es par")
    print(f"El número {numero} es par")
```

Líneas en blanco y longitud de línea

Se recomienda:

- Dejar dos líneas en blanco para separar funciones.
- Dejar una línea en blanco al final de cada archivo.
- Limitar la longitud de línea de código a 79 caracteres.
- Dividir líneas de código que se consideren largas en varias líneas utilizando paréntesis o barras invertidas.

Ejemplo:

```
# Wrong
def calcular_area_circulo(radio):
    pi = 3.14159
    area = pi * radio
    ** 2
    return area

# Correct
def calcular_area_circulo(radio):
    pi = 3.14159
    area = (pi * radio
            ** 2)
    return area
```

Para más información puedes acceder a la documentación oficial. Link: https://peps.python.org/pep-0008/

Pylint

Pylint es una herramienta de análisis estático de código fuente para Python. Su función principal es mejorar la calidad de código fuente mediante la aplicación de estándares de codificación incluyendo las reglas PEP 8 y sugerencias para posibles refactorizaciones. Además es configurable por lo que, puedes agregar tus propias reglas de código.

Si deseas instalar esta herramienta, puedes hacerlo ejecutando **pip install pylint** y luego, **pylint my_module.py** para ejecutar las pruebas.

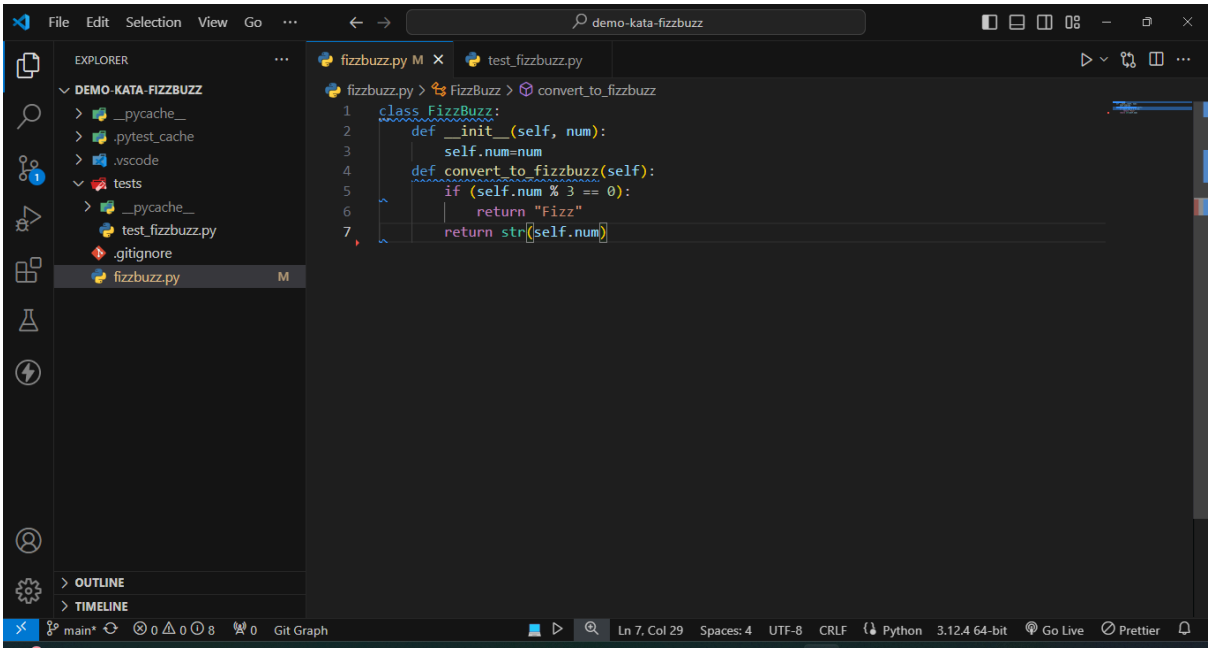
Para más información puedes acceder a la documentación oficial. Link: https://www.pylint.org/

Por otro lado, si estás trabajando en VSCode puedes instalar las extensiones: 1- Python y 2- Pylint.

Las reglas se evaluarán de manera automática mientras escribes el código.

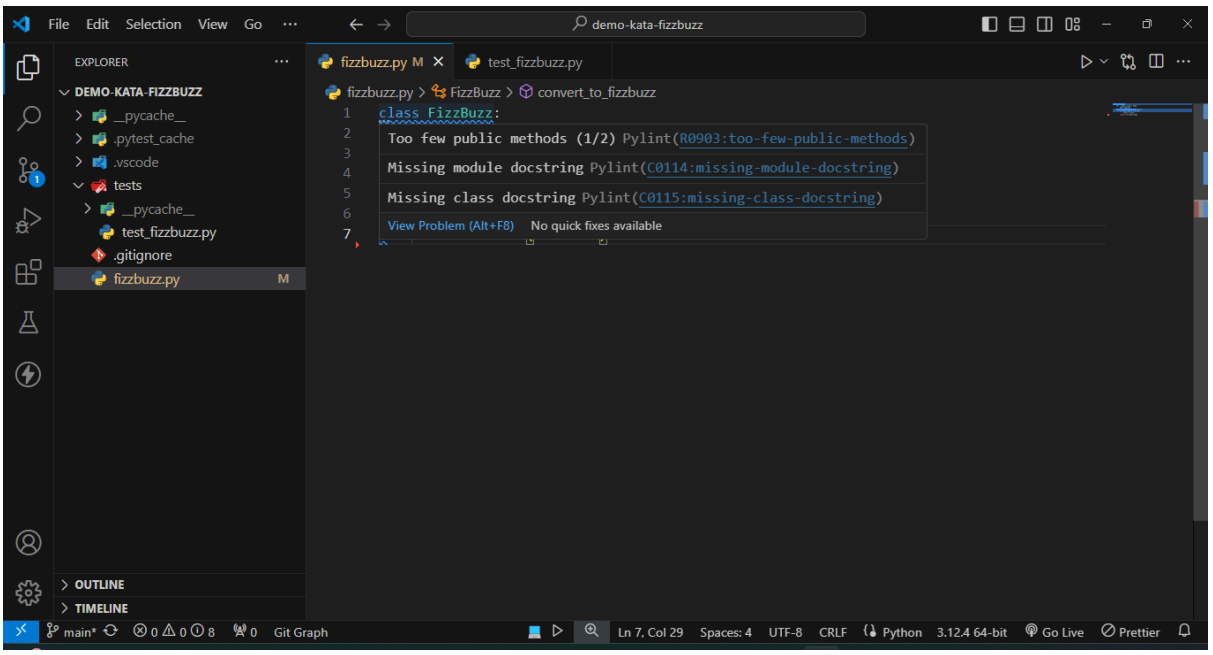
Veamos un ejemplo:

Luego de instalar ambas extensiones, abrir el proyecto **`kata fizzbuzz`** trabajado previamente con VSCode:

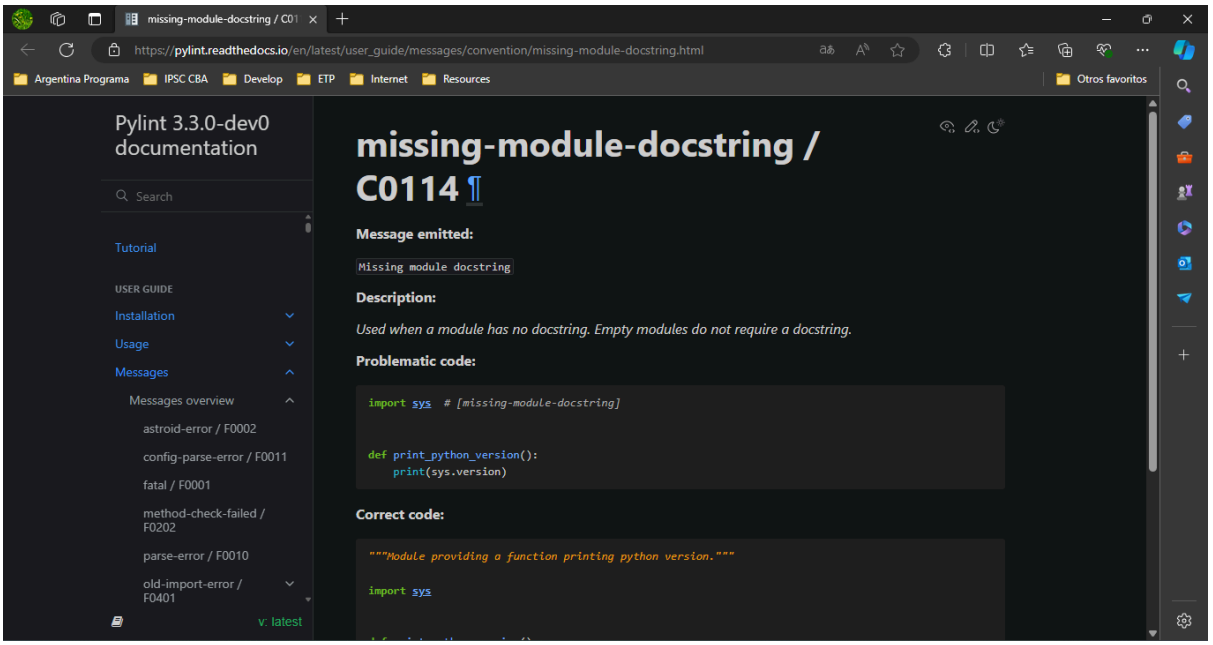


Observa que VSCode nos alerta de manera automática los fallos que tienen que ver con el incumplimiento de los estándares especificados en pylint.

Para resolver, debemos posicionar el cursor sobre el primer error. A continuación, nos aparecerá un tooltip con una una o más reglas que estamos incumpliendo como sigue:

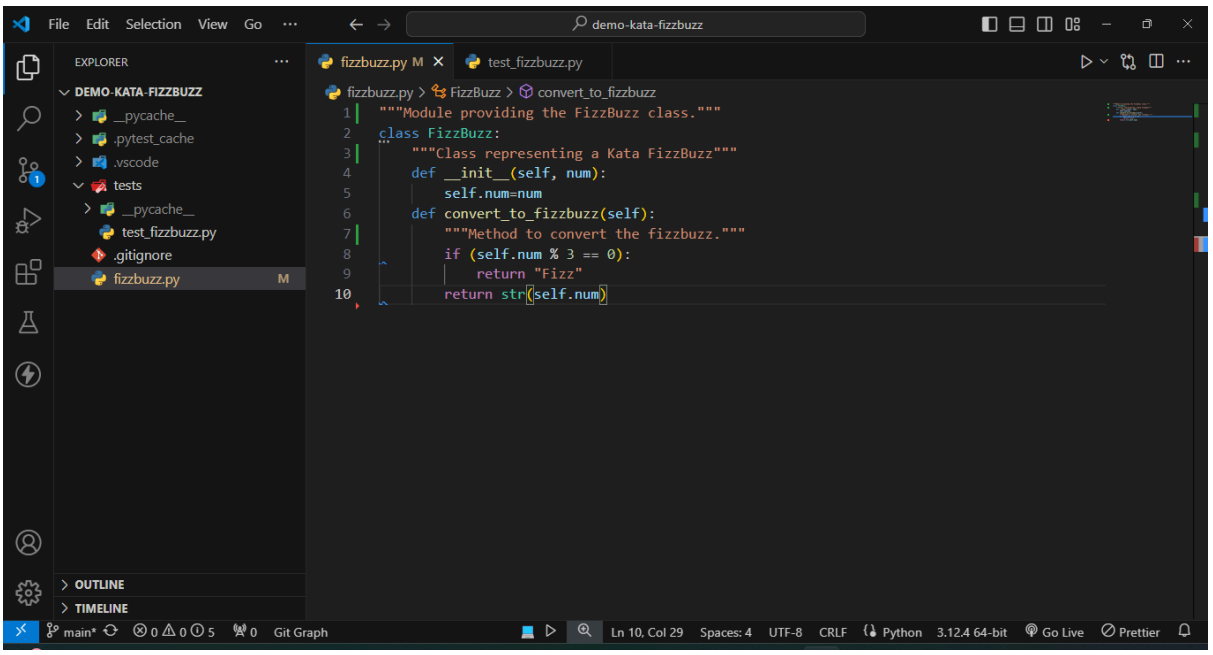


Para comprender el error y poder resolver simplemente debemos hacer click en el enlace. A continuación, Pylint nos llevará directamente a la fuente oficial con el detalle del error y su posible solución.



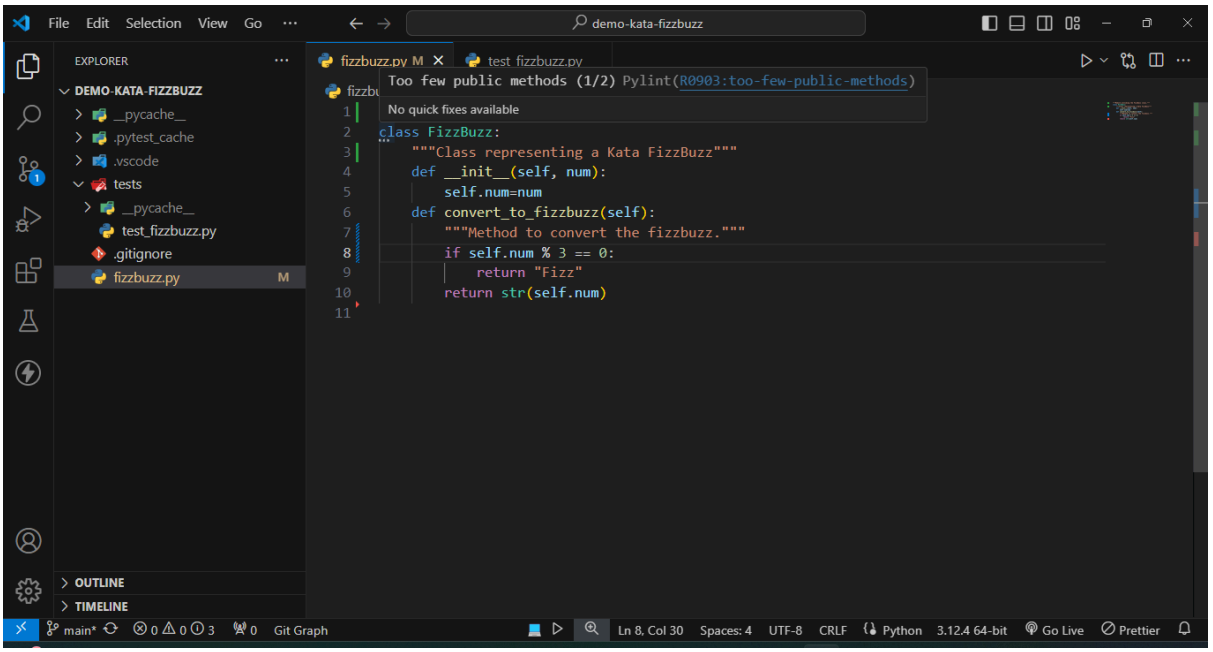
Como podemos observar, el mensaje de **"missing-module-docstring / C0114"** emitido por Pylint refiere a la falta de una docstring (cadena de documentación) en un

módulo. Lo mismo pasa con las clases y métodos. Para resolver, simplemente agregamos una cadena que lo documente como sigue:



Sin embargo, aún existen más errores por resolver. Por ejemplo, pylint nos alerta con el siguiente mensaje **"superfluous-parens / C0325"** sugiere eliminar los paréntesis para la línea 8 dónde se encuentra la estructura condicional if.

Corregido eso, ahora nos alerta con el siguiente mensaje **"too-few-public-methods / R0903"** como sigue:



En este caso, Pylint hace referencia a una clase con muy pocos métodos públicos lo que podría ser un indicio de que la clase no está proporcionando la suficiente funcionalidad o bien, podría simplificarse la funcionalidad en más métodos ayudando a revelar su intención.

En nuestro caso, no se requiere agregar más métodos para pasar la regla por lo que, vamos a deshabilitar la misma agregando un comentario justo encima de la clase dónde se marque el error como sigue:

```
"""Module providing the FizzBuzz class."""
# pylint: disable=too-few-public-methods
class FizzBuzz:
    """Class representing a Kata FizzBuzz"""
    def __init__(self, num):
        self.num=num
    def convert_to_fizzbuzz(self):
        """Method to convert the fizzbuzz."""
        if self.num % 3 == 0:
            return "Fizz"
        return str(self.num)
```

¡Recuerda! Si bien deshabilitar advertencias puede ser útil en ciertos casos, siempre es importante considerar que si estamos siguiendo las buenas prácticas de diseño, una advertencia puede señalar oportunidades para mejorar la calidad del código.

¿Y qué pasa con los comentarios?

Si bien suele ser útil realizar comentarios en el código fuente es importante evitar lo obvio. No comentes lo que es evidente por sí mismo. Por ejemplo, esto no es necesario:

```
# Incrementa el contador en 1
contador += 1
```

Hay una cita de Kent Beck que dice *"Cuando sientas la necesidad de escribir un comentario, quizás debas refactorizar el código"*. Esta cita es un recordatorio valioso dado que no debemos depender en exceso de los comentarios para explicar el código, debemos esforzarnos para escribir código limpio que se explique por sí solo.

A continuación te dejamos una lista de reflexiones a tener en cuenta:

- **Código Expresivo.** En lugar de comentar cada línea, es preferible escribir código que sea expresivo y fácil de entender.
- **Refactorización**. Cuando sientas la necesidad de escribir un comentario que explique el código fuente, considera si esa parte del código podría refactorizarse para ser más claro.
- **Docstrings y comentarios estratégicos**. Aunque debemos evitar comentarios innecesarios, hay momentos en que los comentarios pueden ser útiles como por ej. para documentar derechos de autor, explicar la intención cuando se cree que no fue

la mejor forma de resolver o bien, un TODO con tareas pendientes. Utiliza docstrings para documentar clases, módulos y métodos de manera clara y concisa. Estos se pueden extraer como documentación.

Puede que consideres un comentario útil y necesario, pero **¡recuerda!** El mejor comentario es aquel que no hay que escribir.

¡Pongámonos a prueba! Intenta completar este y los kata realizados previamente aplicando esta regla. Para ello, puedes hacerte las siguientes preguntas: ¿El código se explica por sí sólo? ¿Y en las pruebas? ¿Cumple con las sugerencias de PEP 8?

No hay duplicados

Seguramente al estar codeando te encontraste alguna vez con ese gurú del código que te susurra al oído ¡Evita la duplicación a toda costa!

Dado que esta regla supone que tenemos pruebas unitarias y éstas pasan, esto hace que perdamos el miedo a refactorizar y que resulte el código dañado por lo cual podemos trabajar en aumentar la cohesión, reducir las conexiones, modularizar, etc. gla.

DRY (Don’t Repeat Yourself)

El principio DRY (no te repitas) tiene por objetivo evitar la duplicidad y mantener el código lo más limpio y legible posible. Es decir que no debemos repetir funcionalidad innecesariamente en nuestro código. Cada concepto o fragmento de lógica debe tener una única representación en el sistema. Es por ello que, debemos buscar maneras de reutilizar y abstraer la funcionalidad repetida.

Resolviendo el kata Bowling

El kata Bowling es un reconocido ejercicio para practicar la programación y aplicar el principio DRY (Don’t Repeat Yourself). A continuación, las reglas básicas del mismo:

Estructura del Juego:

- El jugador tiene 10 turnos y dos lanzamientos por turno para derribar los diez bolos.

Reglas:

- En cada lanzamiento, el jugador tiene hasta dos lanzamientos para derribar los diez bolos.

- Si la primera bola de un turno derriba los diez bolos, se denomina "strike" y el turno termina. La puntuación del turno es 10 más el total de los bolos derribados en los dos lanzamientos siguientes.
- Si la segunda bola de un turno derriba los diez bolos, se denomina "spare" y el turno termina. La puntuación es 10 más el total de bolos derribados en el siguiente lanzamiento.
- Si, después de dos lanzamientos, todavía queda al menos uno de diez bolos en pie, la puntuación de ese cuadro es simplemente el número total de bolos derribados en ese turno.
- Si un lanzamiento extra derriba todos los bolos, el proceso no se repite. Los lanzamientos extras sólo se utilizan para calcular la puntuación final.
- La puntuación del juego es el total de las puntuaciones de todos los turnos.

Bowling Score Calculator puede ayudarte para comprender la puntuación.

Veamos a continuación, cómo aplicar el principio DRY al siguiente código referente a la clase JuegoDeBolos:

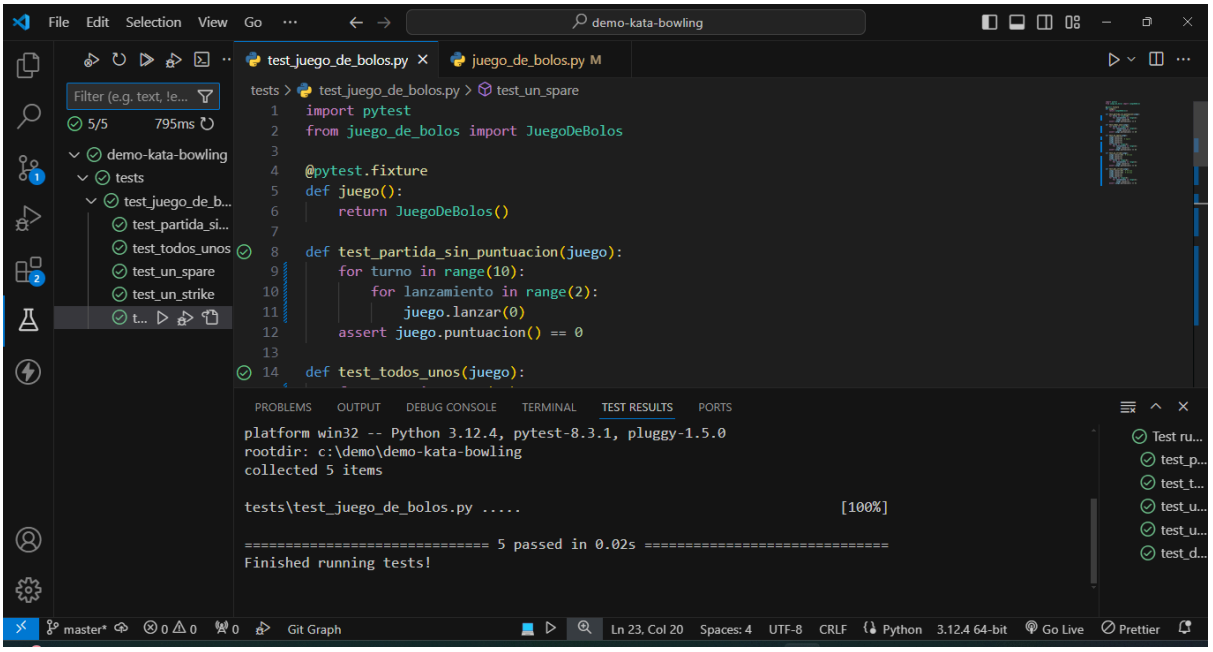
```
# juego_de_bolos.py
class JuegoDeBolos:
    def __init__(self):
        self.lanzamientos = []

    def lanzar(self, bolos_derribados):
        self.lanzamientos.append(bolos_derribados)

    def puntuacion(self):
        puntuacion_total = 0
        indice = 0

        for turno in range(10):
            if self.es_strike(indice):
                puntuacion_total += 10 +
self.lanzamientos[indice+1]+self.lanzamientos[indice+2]
                indice += 1
            elif self.es_spare(indice):
                puntuacion_total += 10 + self.lanzamientos[indice+2]
                indice += 2
            else:
                puntuacion_total += self.lanzamientos[indice] +
self.lanzamientos[indice + 1]
                indice += 2

        return puntuacion_total

    def es_strike(self, indice):
        return self.lanzamientos[indice] == 10

    def es_spare(self, indice):
        return self.lanzamientos[indice] + self.lanzamientos[indice + 1] == 10
```

```
0
```

y sus correspondientes pruebas:

```python
# test_juego_de_bolos.py
import pytest
from juego_de_bolos import JuegoDeBolos

@pytest.fixture
def juego():
    return JuegoDeBolos()

def test_partida_sin_puntuacion(juego):
    for turno in range(10):
        for lanzamiento in range(2):
            juego.lanzar(0)
    assert juego.puntuacion() == 0

def test_todos_unos(juego):
    for turno in range(10):
        for lanzamiento in range(2):
             juego.lanzar(1)
    assert juego.puntuacion() == 20

def test_un_spare(juego):
    juego.lanzar(4)
    juego.lanzar(6)  # Spare
    juego.lanzar(3)
    juego.lanzar(1)
    for turno in range(8):
        for lanzamiento in range(2):
            juego.lanzar(1)
    assert juego.puntuacion() == 33

def test_un_strike(juego):
    juego.lanzar(10)  # Strike
    juego.lanzar(1)
    juego.lanzar(2)
    for turno in range(8):
        for lanzamiento in range(2):
            juego.lanzar(1)
    assert juego.puntuacion() == 32

def test_dos_strike(juego):
    juego.lanzar(10)  # Strike
    juego.lanzar(10)  # Strike
    juego.lanzar(1)
    juego.lanzar(2)
    for turno in range(8):
        for lanzamiento in range(2):
            juego.lanzar(1)
    assert juego.puntuacion() == 51
```

A continuación, evaluamos que las pruebas pasan:



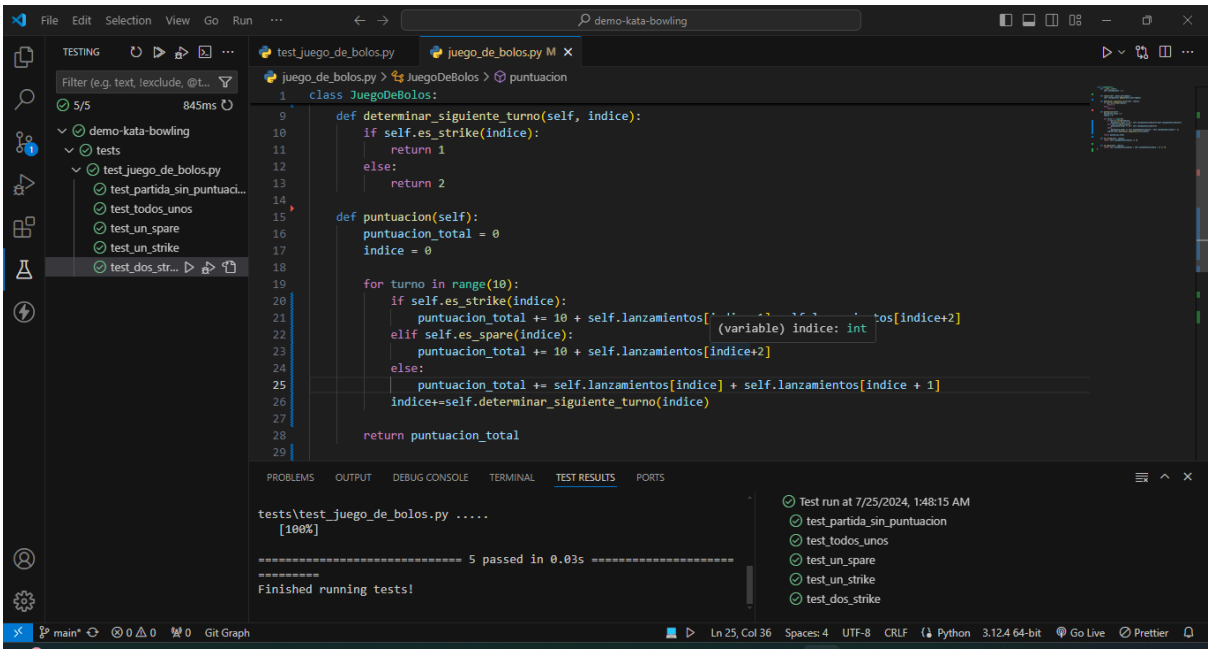
Comenzamos, evaluando la clase JuegoDeBolos, intentando identificar si se repite alguna línea. Como podemos observar, en el método **puntuacion** se repite la línea **indice += 2** que corresponde al cálculo que identifica el índice dónde comienza el siguiente turno. Dado que, para todos los casos se suman dos enteros a la variable **indice** excepto cuando tenemos un *strike* que sólo suma 1 podemos refactorizar creando un nuevo método titulado **determinar_siguiente_turno** como sigue:

```
def determinar_siguiente_turno(self, indice):
        if self.es_strike(indice):
            return 1
        else:
            return 2
```

Luego, eliminamos las líneas repetidas y modificamos el método `publicacion` como sigue:

```
    def puntuacion(self):
        puntuacion_total = 0
        indice = 0

        for turno in range(10):
            if self.es_strike(indice):
                puntuacion_total += 10 +
self.lanzamientos[indice+1]+self.lanzamientos[indice+2]
            elif self.es_spare(indice):
                puntuacion_total += 10 + self.lanzamientos[indice+2]
            else:
                puntuacion_total += self.lanzamientos[indice] +
self.lanzamientos[indice + 1]
            indice+=self.determinar_siguiente_turno(indice)

        return puntuacion_total
```

A continuación, evaluamos si las pruebas aún pasan:



Observa que las pruebas pasan. Esto significa que podemos modificar sin temor a dañar el código fuente dado que las pruebas evalúan que todo sigue funcionando correctamente.

Veamos qué más podemos encontrar repetido… Como podemos observar, el valor de la variable **puntuación_total** se actualiza en varias oportunidades por lo que, podríamos llevar todo el cálculo de puntuación por turnos a un método llamado **obtener_puntuacion_por_turno** para dejar el código más limpio como sigue:

```
    def obtener_puntuacion_por_turno(self, indice):
        if self.es_strike(indice):
            return 10 +
self.lanzamientos[indice+1]+self.lanzamientos[indice+2]
        elif self.es_spare(indice):
            return 10 + self.lanzamientos[indice+2]
        else:
            return self.lanzamientos[indice] + self.lanzamientos[indice + 1]
```

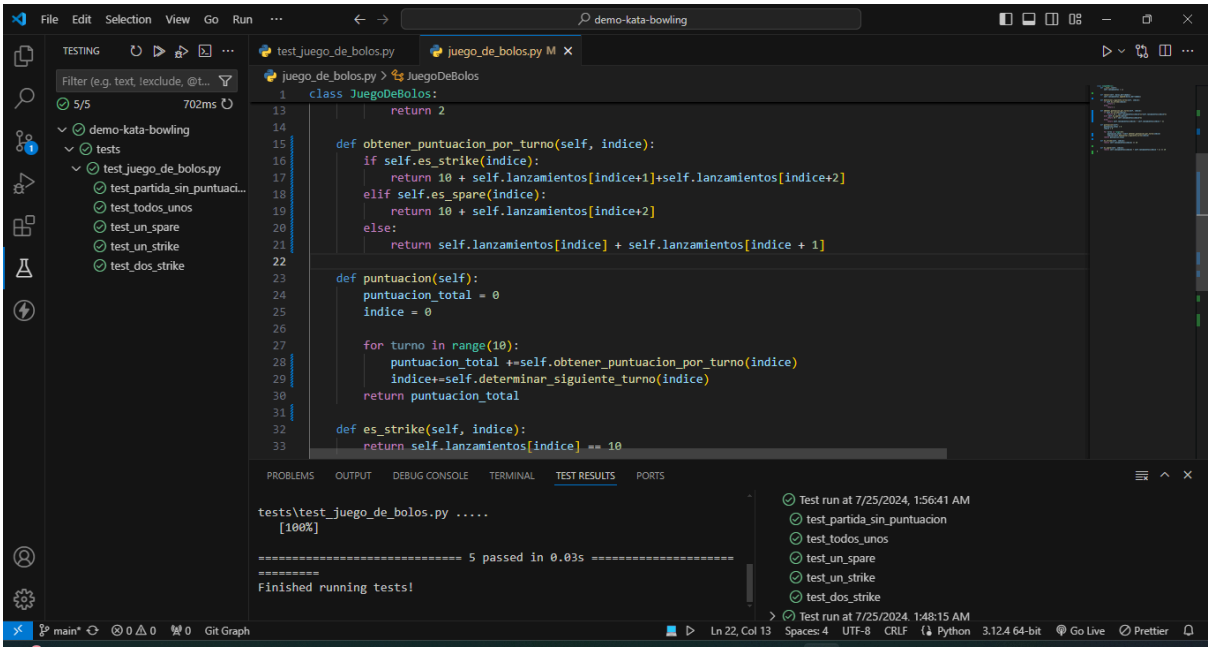
Finalmente, modificamos el método **puntuación** como sigue:

```
    def puntuacion(self):
        puntuacion_total = 0
        indice = 0

        for turno in range(10):
            puntuacion_total +=self.obtener_puntuacion_por_turno(indice)
            indice+=self.determinar_siguiente_turno(indice)
        return puntuacion_total
```

Observa que ahora, el método **`puntuación`** es mucho más fácil de leer ahora dado que expresa su intención claramente.

¡Recuerda! este principio está en tercer lugar, por lo que sí encontramos formas de expresar la intención, no es mala idea refactorizar el código!

Finalmente, ejecutamos las pruebas nuevamente para evaluar que todo siga funcionando como corresponde.



¡Pongámonos a prueba! Intenta completar este y los kata realizados previamente aplicando la regla “No hay duplicados”. Para ello, puedes hacerte las siguientes preguntas: ¿Hay más código duplicado en la clase JuegoDeBolos? ¿Y en las pruebas?.

Tiene el menor número posible de elementos

Esta regla nos sugiere minimizar la cantidad de elementos posibles a fin de reducir la complejidad y la cantidad de partes en nuestro sistema. **No se trata de minimizar la cantidad de líneas de código, sino de tener solo lo esencial para cumplir con los requisitos.**

Es decir que busca:

- La simplicidad. Sólo mantenemos los elementos que son esenciales para cumplir con los requisitos.
- Priorizar la legibilidad y la facilidad de mantenimiento sobre la complejidad. A veces, menos es más.

En este punto es importante mencionar que, la regla de minimizar el número de elementos es la regla de menor prioridad por lo cual, aunque sea importante reducir la cantidad, es más importante contar con pruebas, eliminar duplicados y que el código revele claramente su intención.

Algunos de los principios relacionados con esta regla son: KISS (Keep It Simple, Stupid) y YAGNI (You Ain't Gonna Need It).

KISS (Keep It Simple, Stupid)

Este principio busca la simplicidad en lugar de complicar innecesariamente las cosas dado que, las soluciones más sencillas suelen ser las mejores. Por ende, debemos evitar complicaciones innecesarias y buscar la simplicidad en el diseño y la implementación.

KISS tiene por objetivo reducir la complejidad y facilitar la comprensión.

Por ejemplo cuando diseñamos una clase o función, debemos optar por la solución más directa y fácil de entender. No debemos añadir capas de abstracción o características sofisticadas a menos que realmente lo necesitemos.

YAGNI (You Ain't Gonna Need It)

Este principio "no lo vas a necesitar" nos sugiere que no debemos desarrollar funcionalidades o añadir complejidad anticipándonos a futuros requisitos que aún no han surgido dado que tiene por objetivo evitar el exceso de funcionalidad y mantener el foco en lo que realmente se necesita.

Por ejemplo, supongamos que estamos construyendo una aplicación de gestión de tareas. Si alguien nos pide una función de recordatorios avanzados, no deberíamos implementarlo a menos que sea un requisito inmediato. No debemos anticiparnos a las necesidades futuras sin una razón concreta.

¡Importante! Evita la tentación de escribir código que podría ser útil en el futuro pero que no se necesita ahora mismo.

¿Cómo aplicar YAGNI?

- Si no necesitas una funcionalidad ahora mismo, no la implementes.
- Enfócate en resolver los problemas actuales y construir solo lo que es necesario para el proyecto actual.

Referencias

Corey Haines (04/06/2014), Understanding the 4 rules of simple design. cyber-dojo, a place to practice programming. Recuperado de: https://cyber-dojo.org/creator/home

Carlos Blé Jurado (07/07/2019). La simplicidad por principio. Recuperado de: https://leanmind.es/es/blog/simplicidad-por-principio/