pytest

Adrián García Nóvoa

Índice

1. Introducción

Los test automáticos son considerados una herramienta y una metodología indispensable para la producción de software de calidad. Pero ese no tiene que ser el caso cuando se utiliza pytest como framework de testeo.

El módulo estándar en Python para la realización de testeo es unittest, pero pytest tiene muchas más funcionalidades además de tener una curva de aprendizaje menos pronunciada.

1.1. ¿Por que invertir tiempo escribiendo test?

Para los programadores es común escribir fragmentos de código, seguir tutoriales, ejecutar código en una consola de programación o incluso utilizar Jupyter Notebook. Frecuentemente, esto involucra una verificación manual de los resultados de lo que se está estudiando utilizando la sentencia print. Esto es una forma fácil, natural y válida de aprender nuevas técnicas.

Esta forma de trabajo, sin embargo, no se puede utilizar en el desarrollo de software profesional. El software profesional es normalmente muy complejo, dependiendo de lo bien diseñado que esté el sistema, y pueden existir partes del software que estén entrelazadas, así con la adición de una nueva funcionalidad existe la probabilidad de que otra, aparentemente no relacionada, deje de funcionar. Como resumen; resolver un error puede causar otro.

¿Cómo nos podemos asegurar de que una nueva funcionalidad está funcionando o que un error ha sido eliminado?, ¿Cómo podemos asegurarnos, de que al arreglar o introducir una nueva funcionalidad, otra parte del sistema no deje de funcionar?.

La respuesta es teniendo un sistema de testeo automatizado, o banco de pruebas.

Un conjunto de tests, es código que realiza pruebas al código. Normalmente, estos crean los recursos necesarios y ejecutan los test. Estos luego se aseguran de que los test funcionan como se esperaba. Además de ejecutarse en la máquina del desarrollador, en entornos de trabajo profesional, estos se ejecutan de forma continua, por ejemplo cada hora o cada commit del código. Esta ejecución continua se realiza mediante sistemas automatizados como Jenkins. Debido a esto, agregar una pieza de código de test implica que esta se probará una y otra vez cada vez que una funcionalidad sea añadida o un error sea corregido.

Así, el tener un conjunto de test bien escritos, nos permitirá tener mayor confianza cuando realicemos un cambio, ya sea pequeño o grande.

1.2. ¿Por qué Pytest?

Pytest es un framework maduro y con muchas funcionalidades, desde test pequeños hasta test de gran escala como test funcionales de aplicaciones y librerías.

Pytest es simple para empezar. Para escribir un test no se requiere de clases, como con unittest. Se puede escribir una función que empiece con test y utilizar la sentencia assert que ya viene como parte de Python.

Con Pytest es más simple escribir test, además tiene muchos comandos que incrementan la productividad, como por ejemplo el solo ejecutar los test que están fallando, o ejecutar un conjunto específico de test por el nombre.

Pytest viene con fixtures, el cual nos ayuda con el manejo de recursos. La creación de recursos es un aspecto que muchas veces es pasado por alto. Los test de aplicaciones normalmente necesitan configuraciones complejas, como inicializar el uso de un recurso, registrar información en una base de datos o inicializar una interfaz de usuario. Con Pytest, el uso de estos recursos complejos puede ser manejado por medio de los fixtures.

La personalización es importante, y pytest tiene un sistema muy poderoso de plugins. Estos plugins o complementos pueden cambiar varios aspectos de los test, desde cómo se ejecutan estos, hasta proporcionar nuevas capacidades para facilitar los tests de muchos tipos de aplicaciones y frameworks. Existe un plugin que ejecuta los tests en forma aleatoria cada vez, para asegurarse de que estos test no cambien el estado global que puedan afectar a otros test, plugins que muestran los fallos tal como aparecen en lugar de solo al final de su ejecución, y plugins que ejecutan pruebas en múltiples CPU para acelerar los test. Actualmente hay más de 800 complementos para pytest.

Así mismo, pytest ejecuta los test basados en unittest y sin modificaciones, de esta forma se puede migrar nuestro set de test actuales en forma gradual de unittest a pytest.

Por estos motivos, muchos consideran pytest como la forma Pythonica de hacer testeo en Python.

2. Instalación y Configuración

Pythons: Python 3.6, 3.7, 3.8, 3.9, PyPy3
Plataformas: Linux y Windows
PyPI package: pytest
Documentacion completa en PDF (Ingles: Descargar

pytest es un framework que facilita la creación de test simples y escalables. Los test son expresivos y legibles, y además no se requiere un código repetitivo.

2.1. Instalar pytest

Por medio de pip, solo tenemos que ejecutar el siguiente comando en la terminal:

```
pip install -U pytest
```

Comprobamos que se ha instalado la versión correcta:

```
$ pytest --version
pytest 6.2.2
```

2.2. Instalar y usar plugins

La instalación o desinstalación de plugins de terceros para pytest se puede hacer de forma sencilla con pip de la siguiente manera:

```
pip install pytest-NAME
pip uninstall pytest-NAME
```

Si un plugin está instalado, pytest automaticamente lo encontrara e integrara, no es necesario activarlo.

Aquí podemos ver una pequeña lista de los plugins más útiles y populares:

pytest-pep8: usamos la opción –pep8 para habilitar la verificación del cumplimiento de las reglas PEP8.
pytest-flakes: comprueba el código con pyflakes.
pytest-spark: para ejecutar tests con soporte para pyspark (Apache Spark).
pytest-instafail: para informar de errores mientras se ejecuta el test.

Para ver la lista e información completa de todos los plugins podemos acceder a la lista completa de plugins.

2.2.1. Deactivando un plugin por nombre

Podemos evitar que un plugin se cargue por medio del comando:

```
pytest -p no:NAME
```

Esto significa que cualquier intento posterior de activar/cargar el plugin nombrado no funcionará. Si deseamos deshabilitar totalmente un complemento para un proyecto, podemos agregar esta opción al archivo pytest.ini:

```
[pytest]
addopts = -p no:NAME
```

Alternativamente, si queremos deshabilitarlo solo en ciertos entornos (por ejemplo, en un servidor CI), podemos establecer la variable de entorno PYTEST_ADDOPTS a -p no: name.

3. Usar Pytest

3.1. Creando un test

Para escribir los test es necesario escribir funciones que comiencen con el prefijo "test_". Es necesario que las llamemos así, ya que en el momento que ejecutemos pytest deberemos especificar un directorio raíz, a partir de este directorio pytest leerá todos los archivos buscando funciones que comiencen con "test_".

Para ejecutar todas las pruebas que queramos en diferentes archivos, será tan simple como ejecutar pytest con esta carpeta como argumento o desde la propia carpeta.

Crearemos un test de prueba simple con solo cuatro líneas de código que nos servirá para poder comprender el concepto:

```
# contenido del archivo test_prueba1.py
def func(x):
        return x + 1
def test_answer():
        assert func(3) == 5
```

Ahora podemos ejecutar pytest desde la terminal. Se debe ejecutar en la carpeta donde tenemos el archivo que hemos creado o indicarle la ruta como argumento.

```
$ pytest
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item


test_sample.py F [100%]

================================= FAILURES =================================
_____________________________ test_answer _____________________________

        def test_answer():
>               assert func(3) == 5
E               assert 4 == 5
E               + where 4 = func(3)

test_sample.py:6: AssertionError
========================== short test summary info ==========================
FAILED test_sample.py::test_answer - assert 4 == 5
============================ 1 failed in 0.12s =============================
```

El [100 %] se refiere al progreso general en la ejecución de todos los test. Una vez se ha finalizado la ejecución, pytest nos muestra un informe de los errores, en este caso, nos indica un error ya que no se cumple el assert $4 = 5$ como era de esperar.

3.2. Parametrizando nuestros test

En el ejemplo de apartado anterior hemos hecho un test sencillo: una entrada, una salida y sin llamadas a servicios externos.

Sin embargo, en muchos casos necesitaremos probar nuestras funciones para distintos argumentos y tendríamos que escribir métodos para cada caso, lo que seria una duplicación absurda de código. Para solucionar este problema podemos hacer uso de la parametrizacion por medio del decorador @pytest.mark.parametrize.

Haremos ahora un ejemplo utilizando este decorador para comprobar que se cumple la función conmutativa y asociativa de una función suma:

```
import pytest
from main import suma

def suma(x, y):
        return x + y

def test_suma
        assert suma(2, 2) == 4

@pytest.mark.parametrize(
        "input_a, input_b, expected",
        [
                (3, 2, 5),
                (2, 3, 5),
                (suma(3, 2), 5, 10),
                (3, suma(2, 5), 10)
        ]
)

def test_suma_multi(input_a, input_b, expected):
        assert suma(input_a, input_b) == expected
```

Este código le indica a pytest que ejecute la prueba test_suma_multi cuatro veces, cada una reemplazando input_a, input_b y expected con sus valores correspondientes especificados en nuestro parametrize.

3.3. Fixtures

Los fixtures son funciones adjuntas a los test que se ejecutan antes de que se ejecute la función de test. Son funciones que se configuran y ejecutan antes y se limpian una vez se completa la ejecución de los test que hagamos.

La funcion fixture de pytest es llamada automaticamente por el framework pytest cuando el nombre del argumento y el fixture es el mismo.

Una función se marca como fixture con el siguiente marcador:

```
@pytest.fixture
```

Un ejemplo simple pero descriptivo de esto:

```
@pytest.fixture
def fixture_func():
    return "fixture test"

def test_fixture(fixture_func):
    assert fixture_func == "fixture test"
```

Las fixtures de pytest no se quedan simplemente en una forma de comenzar los test desde un cierto estado. Una característica de gran importancia y mas en el caso de usar pytest para Pyspark es la reutilizacion del mismo código. Analizaremos esta propiedad en la siguiente sección.

4. Pytest con Pyspark

Cuando queremos testear una aplicación PySpark, nos encontramos con el problema de ejecutar nuestros test por medio de SparkSession. Por supuesto, podemos crear una SparkSession para cada función de test, pero eso ralentizará significativamente la ejecución. Esa solución puede ser aceptable cuando solo tenemos uno o dos test de PySpark dentro de una aplicación más grande, pero ¿qué pasa si queremos ejecutar un gran cantidad de test y cada uno de ellos usa PySpark? Una de las soluciones es usar las fixtures.

Existen dos formas de hacer test de código Apache Spark con pytest. Analizaremos ambas opciones:

4.1. Por medio de fixture

De esta forma, deberemos crear una instancia de SparkSession y pasársela a cada test como parámetro. Los fixtures configuran el entorno de test y se limpian después de la ejecución de este.

Para configurar un fixture, debemos crear un nuevo archivo en el directorio de test e implementar una función que devuelva el valor de ese fixture. La función debe decorarse usando el decorador pytest.fixture como ya hemos indicado anteriormente. Dentro de la función, también definiremos un finalizador que liberará los recursos asignados.

Para reutilizar la misma SparkSession en todos los test, debemos especificar el alcance del fixture y establecer su valor a "session". Los fixture se suelen incluir en un archivo con el nombre conftest.py en nuestra carpeta raiz de test. Los fixtures dentro de este archivo se compartirán entre todos los test de nuestra carpeta. Sin embargo, definir fixtures en conftest.py podría ser inútil y ralentizaría los test si dichos fixtures no se utilizan en todas o casi todos los test.

El siguiente ejemplo demostrará la forma completa de definir una SparkSession por medio de un fixture. Tenga en cuenta que el nombre de la función se utilizará como nombre del fixture. Este código ira dentro de un archivo. Es importante que el nombre sea çonftest.py":

```
# Contenido del archivo conftest.py
import pytest
from pyspark.sql import SparkSession


@pytest.fixture(scope="session")
def spark_session(request):
    spark_session = SparkSession.builder \
        .master("local[*]") \
        .appName("some-app-name") \
        .getOrCreate()

    request.addfinalizer(lambda: spark_session.sparkContext.stop())

    return spark_session
```

En los test, tendremos que declarar que fixture queremos usar dentro del archivo test. La función que crea la SparkSession se llama spark_session, por lo tanto usaremos el mismo nombre para declarar la fixture. Esto iría dentro de los archivos que queremos testear:

```
pytestmark = pytest.mark.usefixtures("spark_session")
```

Ahora, podemos añadir el parámetro spark_session a cada función de test que necesite una SparkSession:

```
def test_name(spark_session):
    ...
```

4.2. Por medio del plugin pytest-pyspark

El plugin pytest nos permite ejecutar tests con soporte para pyspark (Apache Spark).

Este plugin nos permite especificar el directorio de SPARK_HOME en pytest.ini y hará que pyspark sea importable a nuestros test para ser ejecutados por pytest.

Además, nos permite definir "spark_options."[en] pytest.ini para customizar pyspark con nuestras necesidades, incluyendo "spark.jars.packages"para cargar librerías externas (por ejemplo"com.databricks:spark-xml").

pytest-spark proporciona la fixture spark_context y spark_session para usarse en los test.

Nota: no es necesario definir SPARK_HOME si ya hemos instalado pyspark por medio de pip (pip install pyspark). En este caso no definimos SPARK_HOME en pytest.ini, -spark_home como una variable de entorno.

4.2.1. Instalación

Como hemos comentado en la sección de instalación de plugins, utilizaremos pip para instalar este plugin:

```
$ pip install pytest-spark
```

4.2.2. Uso

Para ejecutar los test con la localización de spark_home definida se puede hacer por medio de tres métodos distintos:

1. Por medio de la línea de comandos con la opción spark_home":

```
$ pytest --spark_home=/opt/spark
```

2. Añadiendo el valor de "spark_home" en el archivo pytest.ini en el directorio del proyecto:

```
[pytest]
spark_home = /opt/spark
```

3. Estableciendo la variable de entorno "SPARK_HOME".

Nota: "spark_home" será leído en el orden en el que se han descrito las distintas opciones. Es decir, podremos sustituir el valor de pytest.ini añadiéndolo en la opción de la línea de comandos que tiene más prioridad.

4.2.3. Personalizar spark_options

Simplemente tenemos que definir las "spark_options" deseadas en nuestro pytest.ini, por ejemplo:

```
[pytest]
spark_home = /opt/spark
spark_options =
    spark.app.name: my-pytest-spark-tests
    spark.executor.instances: 1
    spark.jars.packages: com.databricks:spark-xml_2.12:0.5.0
```

4.2.4. Uso de la fixture spark_context

Usaremos la fixture spark_context en nuestros tests como una fixture normal de pyspark. SparkContext se instancia una vez y será reutilizada durante todos nuestros test.

Ejemplo:

```
def test_my_case(spark_context):
    test_rdd = spark_context.parallelize([1, 2, 3, 4])
    # ...
```

4.2.5. Uso de la fixture spark_session (Spark 2.0 o superior)

Usaremos la fixture spark_session en nuestros tests como una fixture normal de pyspark. SparkSession se instancia con soporte para Hive por defecto y se reutilizará en toda nuestra sesión de tests.

Ejemplo:

```
def test_spark_session_dataframe(spark_session):
    test_df = spark_session.createDataFrame([[1,3],[2,4]], "a: int, b: int")
    # ...
```

4.2.6. Sustituyendo los parámetros por defecto de la fixture spark_session

Por defecto, la spark_session se cargará con la siguiente configuración:

```
{
    'spark.app.name': 'pytest-spark',
    'spark.default.parallelism': 1,
    'spark.dynamicAllocation.enabled': 'false',
    'spark.executor.cores': 1,
    'spark.executor.instances': 1,
    'spark.io.compression.codec': 'lz4',
    'spark.rdd.compress': 'false',
    'spark.sql.shuffle.partitions': 1,
    'spark.shuffle.compress': 'false',
    'spark.sql.catalogImplementation': 'hive',
}
```

Podemos anular o modificar estos parámetros en pytest.ini. Por ejemplo, eliminando el soporte para Hive en nuestra sesión de la siguiente manera:

```
[pytest]
spark_home = /opt/spark
spark_options =
    spark.sql.catalogImplementation: in-memory
```

5. Enlaces de interes

Página oficial de PyTest: https://docs.pytest.org/en/stable/

Página oficial del paquete pytest-spark: https://pypi.org/project/pytest-spark/

Integración de pytest en el IDE PyCharm: https://www.jetbrains.com/help/pycharm/pytest.html

Github del proyecto: https://github.com/malexer/pytest-spark

Página con información sobre testeo de código Python: https://pythontesting.net/start-here/