Ejemplo sencillo de Test Driven Development en Python

En este tutorial os voy a hablar un poco sobre el TDD: el desarrollo guiado por pruebas.

A diferencia de programar un proyecto y luego añadir pruebas, la idea del TDD es desarrollar el software a partir de las propias pruebas, dejando que éstas nos guíen durante el proceso.

De esta forma cada funcionalidad ya es concebida desde el principio con la idea de superar un test y por tanto está monitorizada para su correcto funcionamiento en el futuro.

Para hacer TDD hay que seguir un orden estricto:

1. **Escribir una prueba**, que recoja los requisitos de la funcionalidad que vamos a implementar.
2. **Ejecutar la prueba y comprobar que falla**, ya que todavía no habremos implementado la funcionalidad.
3. **Implementar la funcionalidad**, con el código mínimo necesario.
4. **Volver a ejecutar la prueba**, que en esta ocasión debería pasar correctamente, y si no es así corregir el código hasta que la pase.
5. **Refactorizar el código**, borrando redundancias e incongruencias, siempre comprobando que los tests siguen validando bien.
6. **Volver a empezar**, para implementar el siguiente requisito.

Crear una calculadora simple es el ejemplo más fácil e ilustrativo del procedimiento, basado en crear una clase calculadora con métodos para diferentes operaciones (suma, resta, división…).

Lo primero que vamos a hacer es crear un fichero llamado **test_calculator.py** en un directorio. Es importante que empiece con **test_** si queremos aprovechar la opción de autodescubrimiento de Python:

```
test_calculator.py
# Cargamos el módulo unittest
import unittest

# Creamos una clase heredando de TestCase
class TestMyCalculator(unittest.TestCase):

    # Creamos una prueba para probar un valor inicial
    def test_initial_value(self):
        calc = Calculator()
        self.assertEqual(0, calc.value)
```

Como véis es un simple test para comprobar que el valor inicial de nuestra supuesta calculadora es 0 (es el valor que muestran por defecto todas las calculadoras).

Por cierto, los tests también deben empezar con **test_**.

Ahora, desde el directorio que contiene el fichero ejecutamos los tests de autodescubrimiento:

```
C:\python-tdd-example>python -m unittest discover
E
======================================================================
ERROR: test_initial_value (test_calculator.TestMyCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\python-tdd-example\test_calculator.py", line 9, in
test_initial_value
    calc = Calculator()
NameError: name 'Calculator' is not defined
----------------------------------------------------------------------
Ran 1 test in 0.001s
FAILED (errors=1)
```

Como era obvio se encontrará un test, se ejecutará y fallará.

Para seguir correctamente la filosofía del TDD es esencial no implementar todo el código de golpe, sino simplemente resolver los errores de uno en uno para hacer que la prueba pase, **de eso se trata que nos guíen las pruebas**.

El error que tenemos ahora nos dice:

```
NameError: name 'Calculator' is not defined
```

Para solucionarlo creamos la clase **Calculator** con el mínimo código:

```
calculator.py
class Calculator:
    pass
```

Y hacemos uso de ella en nuestros módulo de pruebas:

```
test_calculator.py
import unittest

# Importamos la clase calculadora
from calculator import Calculator

class TestMyCalculator(unittest.TestCase):

    def test_initial_value(self):
        calc = Calculator()
        self.assertEqual(0, calc.value)
```

Se supone que hemos resuelto el fallo, así que vamos a probar:

```
C:\python-tdd-example>python -m unittest discover
E
```

```
======================================================================
ERROR: test_initial_value (test_calculator.TestMyCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\python-tdd-example\test_calculator.py", line 11, in
test_initial_value
    self.assertEqual(0, calc.value)
AttributeError: 'Calculator' object has no attribute 'value'
----------------------------------------------------------------------
Ran 1 test in 0.001s
FAILED (errors=1)
```

¡Vaya qué sorpresa! Hemos arreglado un error y a aparecido otro diciéndonos que no tenemos el atributo value:

```
AttributeError: 'Calculator' object has no attribute 'value'
```

Pues nada, tendremos que arreglarlo y añadir este atributo en el constructor de nuestra clase:

```
calculator.py
class Calculator:

    def __init__(self):
        self.value = 0
```

Ejecutamos de nuevo:

```
C:\python-tdd-example>python -m unittest discover
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Nuestro código es muy sencillo, pero podemos mejorar un poco el test unitario añadiendo un método setUp() que se encargue de crear la instancia automáticamente:

```
test_calculator.py
import unittest
from calculator import Calculator

class TestMyCalculator(unittest.TestCase):

    def setUp(self):
        self.calc = Calculator()

    def test_initial_value(self):
        self.assertEqual(0, self.calc.value)
```

Después de refactorizar volveremos a comprobar que el código pasa los tests, no vaya a ser que la hayamos liado:

```
C:\python-tdd-example>python -m unittest discover
.
----------------------------------------------------------------------
Ran 1 test in 0.001s
OK
```

¡Perfecto! Ya hemos desarrollado nuestro primer requisito guiándonos de las pruebas mientras hemos resuelto 2 errores.

Una calculadora sin operaciones no es una calculadora, así que vamos a añadirle por lo menos un método para sumar dos valores y guardar el resultado en el atributo value.

Recordemos sin embargo que estamos haciendo TDD, así que no podemos escribir el método directamente, primero tendremos que hacer un test que falle:

```
test_calculator.py
import unittest
from calculator import Calculator

class TestMyCalculator(unittest.TestCase):

    def setUp(self):
        self.calc = Calculator()

    def test_initial_value(self):
        self.assertEqual(0, self.calc.value)

    # Creamos un nuevo test para comprobar una suma
    def test_add_method(self):
        # Ejecutamos el método
        self.calc.add(1, 3)
        # Comprobamos si el valor es el que esperamos
        self.assertEqual(4, self.calc.value)
```

Ejecutamos el test que fallará:

```
C:\python-tdd-example>python -m unittest discover
E.
======================================================================
ERROR: test_add_method (test_calculator.TestMyCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\python-tdd-example\test_calculator.py", line 15, in
test_add_method
    self.calc.add(1, 3)
AttributeError: 'Calculator' object has no attribute 'add'
----------------------------------------------------------------------
Ran 2 tests in 0.001s
FAILED (errors=1)
```

Implementamos el método **add**:

```
calculator.py
class Calculator:

    def __init__(self):
        self.value = 0

    def add(self, a, b):
        self.value = a + b
```

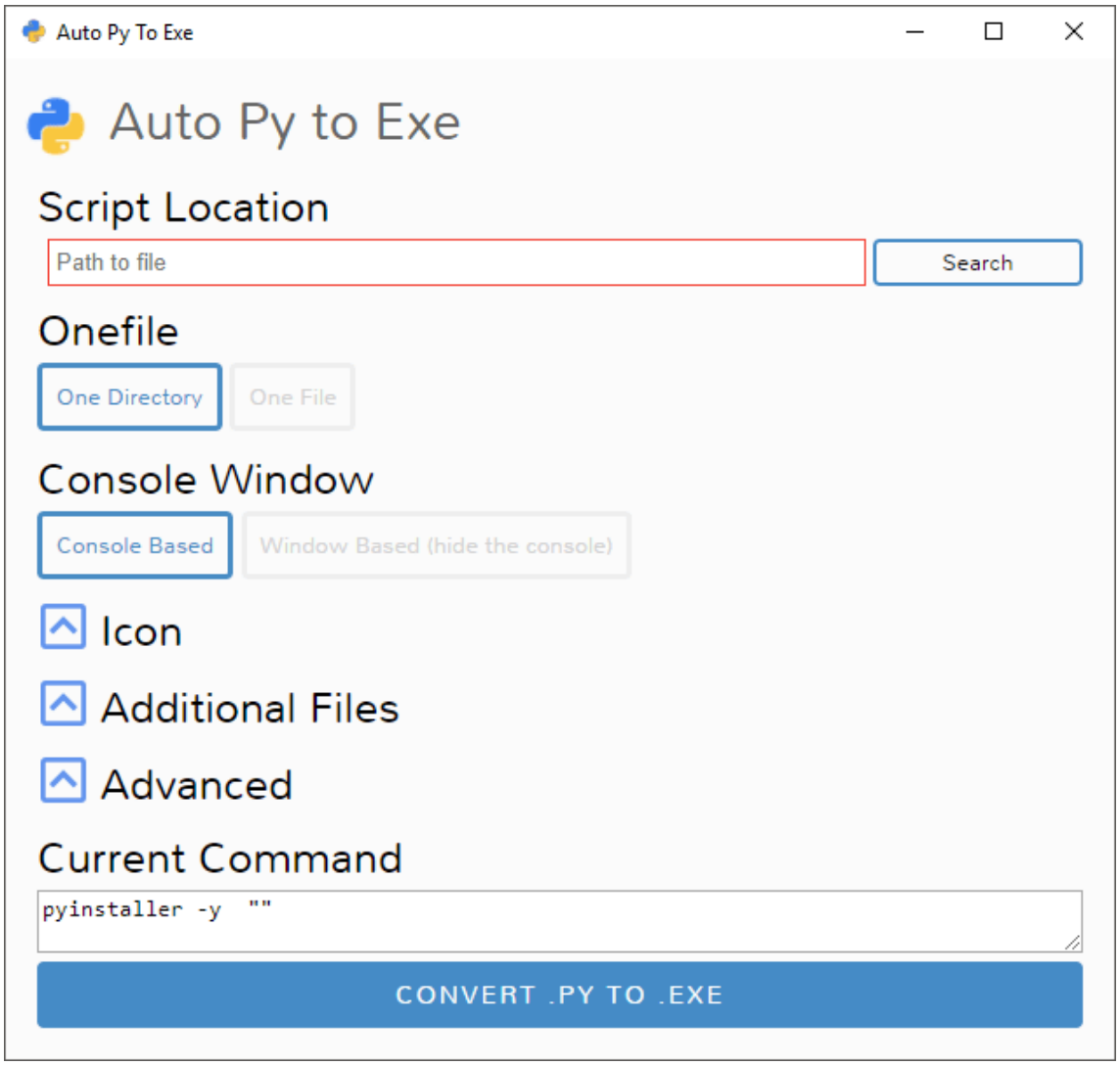
Probamos de nuevo el test:

```
C:\python-tdd-example>python -m unittest discover
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s
OK
```

¡Perfecto, ya tenemos implementada nuestra suma! Ahora sería cuestión de ir añadiendo la resta, el producto, la división, etc.

Generar ejecutables utilizando auto-py-to-exe

Esta aplicación permite generar archivos .exe de tus proyectos, ya sea un archivo .py o varios. Tiene una interfaz gráfica de usuario que se se ve así:



1) Instalación y ejecución

Instalación usando PyPI:

```
pip install auto-py-to-exe
```

Para abrir la aplicación:

```
auto-py-to-exe
```

2) Conversión

Hay algunas opciones principales:

- Seleccionar el archivo principal .py
- Seleccionar la opción "Un directorio" o "Un archivo"
- Seleccionar los archivos adicionales

2.1) Seleccionar el archivo principal .py

Si tiene varios archivos, seleccionar el que inicie el programa.

2.2) "One Directory"

Al elegir la opción **"Un directorio",** Auto PY to EXE colocará todas las dependencias en una carpeta. Puede elegir el directorio de salida en el menú "Avanzado". Si tiene archivos multimedia como iconos y fondos, no debería tener problemas para usarlos dentro de su .exe si coloca archivos / carpetas multimedia en el directorio de salida.

2.3) "One File"

Al elegir la opción **"Un archivo",** Auto PY to EXE creará un archivo .exe que contiene todas las dependencias, pero **NO LOS ARCHIVOS DE MEDIOS**. Si tu programa solo tiene la interfaz gráfica de usuario predeterminada de Windows sin iconos, fondos, archivos multimedia o si prefieres colocar la carpeta con los recursos multimedia adjunta al archivo .exe omite la siguiente explicación. Para aquellos que quieran empaquetar archivos multimedia en el archivo .exe, continua con el paso 2.4.

2.4) Elegir archivos adicionales

Hay un menú en **Auto PY to EXE** llamado "Archivos adicionales" que permite agregar archivos. Sin embargo, hay un problema. "Auto PY to EXE" usa pyinstaller que descomprime los datos en una carpeta temporal y almacena esta ruta de directorio en la variable de entorno _MEIPASS. El proyecto no encontrará los archivos necesarios porque la ruta cambió y no verá la nueva. En

otras palabras, si se eliges la opción "Un archivo", los archivos seleccionados en el menú "Archivos adicionales" no se agregarán al archivo .exe.

Para solucionar este problema, debes utilizar este código proporcionado por el desarrollador de Auto PY to EXE:

```
def resource_path(relative_path):
    """ Get absolute path to resource, works for dev and for
PyInstaller """
    try:
        # PyInstaller creates a temp folder and stores path in
_MEIPASS
        base_path = sys._MEIPASS
    except Exception:
        base_path = os.path.abspath(".")
    return os.path.join(base_path, relative_path)</pre>
```

Para usar este código en su proyecto, reemplaza los enlaces al archivo multimedia que tienes ahora. Por ejemplo:

```
setWindowIcon(QIcon('media\icons\logo.png'))
```

Lo cambiaríamos por:

```
setWindowIcon(QIcon(resource_path('logo.png'))
```

Ahora se hará referencia al enlace correctamente y los archivos elegidos se empaquetarán correctamente en un archivo .exe.

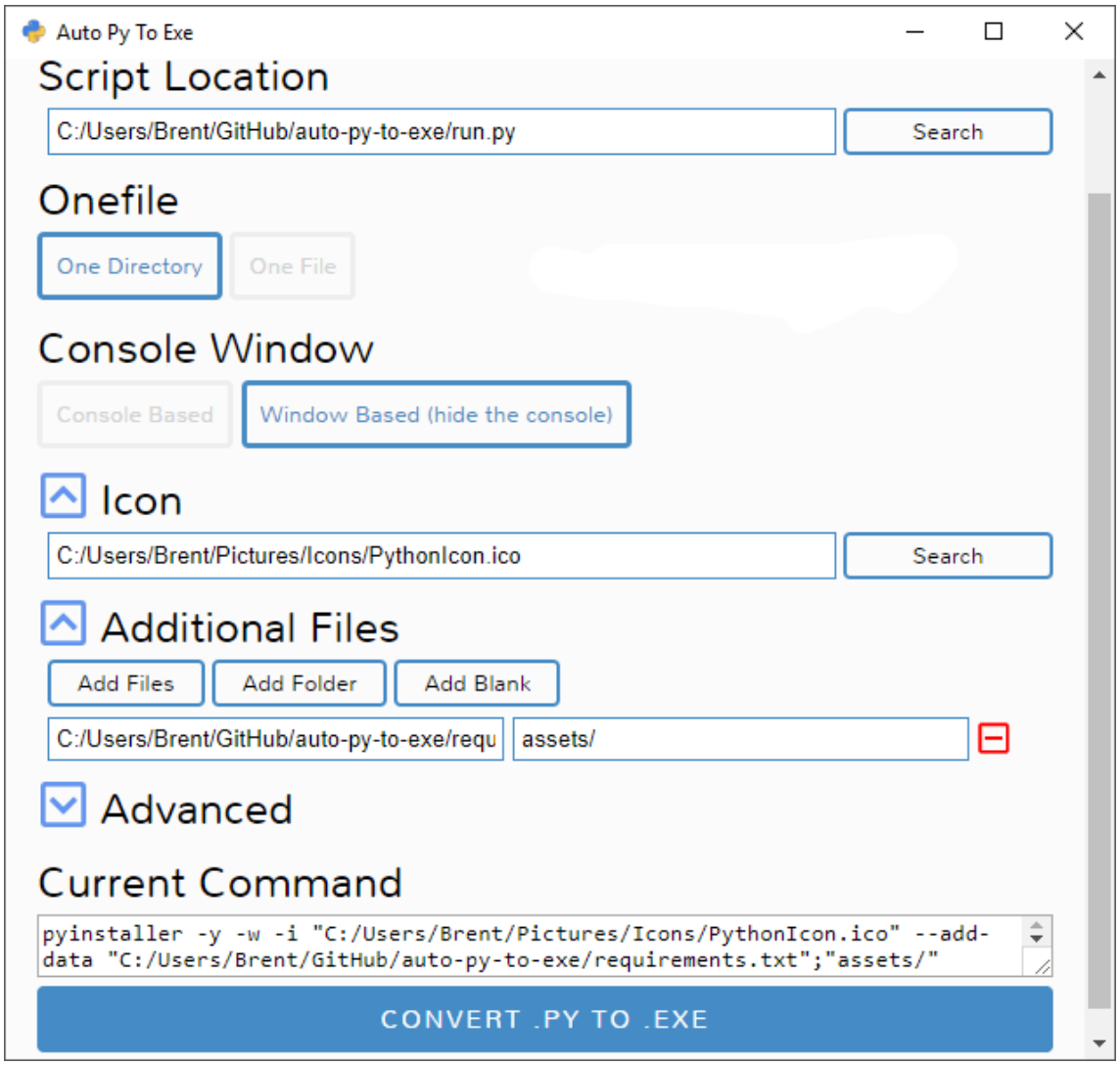
A modo de comparación, como quedaría antes:

```
"C:\Users\User\Project\media\icons\logo.png"
```

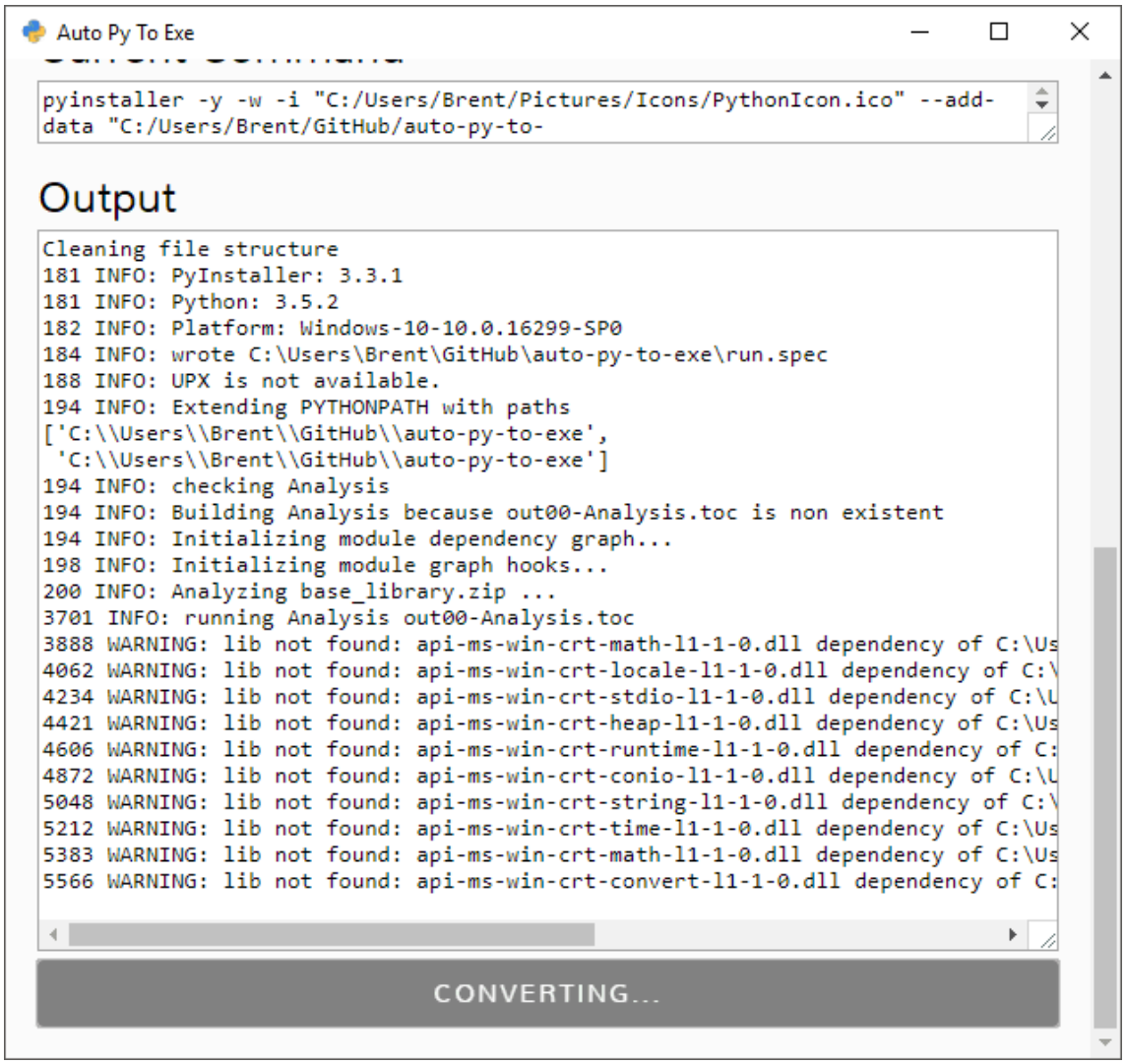
Y después de usar la función **resource_path()**:

```
"C:\Users\User\AppData\Local\Temp\\_MEI34121\logo.png"
```

Ahora presiona **CONVERT .PY TO .EXE**


Auto Py To Exe
Script Location
C:/Users/Brent/GitHub/auto-py-to-exe/run.py
Search
Onefile
One Directory
One File
Console Window
Console Based
Window Based (hide the console)
Icon
C:/Users/Brent/Pictures/Icons/PythonIcon.ico
Search
Additional Files
Add Files
Add Folder
Add Blank
C:/Users/Brent/GitHub/auto-py-to-exe/requ
assets/
Advanced
Current Command
pyinstaller -y -w -i "C:/Users/Brent/Pictures/Icons/PythonIcon.ico" --add-data "C:/Users/Brent/GitHub/auto-py-to-exe/requirements.txt";"assets/"
CONVERT .PY TO .EXE

Espera:



3) Ejecuta el programa

¡Ya está! Prueba el ejecutable y asegúrate de que todo funcione bien.

- Si creaste el ejecutable en un directorio cada archivo que necesita debe estar en un **único directorio**.
- Si creaste el ejecutable en un fichero deberías tener un solo archivo .exe. No necesitarás ningún archivo ni carpeta multimedia junto al ejecutable .exe para que se ejecute correctamente.