

# \* Práctica 1.5: Creación de entorno

## P1.5 - Creación de un entorno virtual con Pytest

### 1. Introducción

Los entornos virtuales se pueden describir como directorios de instalación aislados. Este aislamiento te permite localizar la instalación de las dependencias de tu proyecto, sin obligarte a instalarlas en todo el sistema.

Se trata de un entorno Python en el que el intérprete Python, las bibliotecas y los scripts instalados en él están aislados de los instalados en otros entornos virtuales, y (*por defecto*) cualquier biblioteca instalada en un «sistema» Python, es decir, uno que esté instalado como parte de tu sistema operativo.

Virtualenv es una herramienta que se utiliza para crear entornos Python aislados. Crea una carpeta que contiene todos los ejecutables necesarios para usar los paquetes que necesitaría un proyecto de Python.

### 2. Pasos a seguir en la práctica

**2.1. Antes de nada, debemos abrir una carpeta en Visual Studio Code, donde vamos a trabajar con nuestros programas de Python. (por ejemplo en ~/Documentos/practica5)**

Podemos hacerlo de varias formas:

- Desde una terminal de tu sistema operativo, navegando donde queramos crear el directorio y usando el comando `mkdir nombreCarpeta`. Posteriormente accederemos a la nueva carpeta (`cd`) creada y ejecutaremos el comando `code ..`
- A nivel gráfico, desde el Explorador de archivos, creando una carpeta y arrastrándola dentro de Visual Studio Code.
- También podemos hacerlo desde Visual Studio Code, en el menú `File -> Open Folder`, navegaremos a la carpeta o la crearemos, después simplemente pulsamos en el botón `Seleccionar carpeta`.

**2.2. A continuación abrimos la consola dentro del IDE:**

Teclas rápidas `Ctrl+ñ` o desde el menú `View -> Terminal`

**2.3. Instalamos virtualenv con el siguiente comando:**

```
pip install virtualenv
```

**2.4. Para comprobar su versión:**

```
virtualenv --version
```

## 2.5. Creamos ahora un entorno virtual:

```
python -m virtualenv venv
```

**2.6. El entorno virtual crea físicamente una carpeta llamada venv, donde gestionará todas las librerías que instalemos.**

## 2.7. Para utilizar el entorno virtual debemos activarlo:

```
.\venv\Scripts\activate
```

- Si se produce un error que indica que "la ejecución de scripts está deshabilitada en el sistema"... debéis abrir como **Administrador** Windows PowerShell y ejecutar el siguiente comando:

```
Set-ExecutionPolicy RemoteSigned
```

- Este error ocurre porque, en PowerShell, la ejecución de scripts está deshabilitada por motivos de seguridad. Para solucionarlo, necesitas cambiar la política de ejecución para permitir la ejecución de scripts.
- El parámetro `RemoteSigned`: Permite ejecutar scripts locales **no firmados**, pero los scripts descargados de Internet necesitarán estar firmados.
- Si en un futuro quiers volver a la configuración original, solo tienes que ejecutar el mismo comando con el parámetro `Restricted`:

```
Set-ExecutionPolicy Restricted
```

**2.8. Si queremos comprobar los paquetes instalados en el entorno virtual ejecutamos el siguiente comando:**

```
pip list
```

## 2.9. Para comprobar que tenemos la última versión de pip y actualizarla:

```
python -m pip install --upgrade pip
```

**2.10. Ya tenemos el entorno virtual preparado y activado para usarlo en nuestro proyecto. Vamos a instalar pytest, que nos van a ayudar a ejecutar las pruebas unitarias:**

```
pip install pytest
```

**2.11. Podemos volver a revisar los paquetes instalados y vemos cómo nos ha instalado pytest y otros paquetes necesarios:**

```
pip list
```

## 2.12. Para comprobar la versión que tenemos instalada de pytest:

```
pytest --version
```

### **2.13. Si necesitamos desactivar el entorno virtual**

```
deactivate
```

### **2.14. Para exportar los paquetes que tenemos instalados, por si los queremos instalarlos en otro entorno posteriormente:**

```
pip freeze > requirements.txt
```

### **2.15. Para importarlos en otro entorno virtual de otra carpeta o proyecto:**

```
pip install -r requirements.txt
```

# **\* Práctica 1.6: Uso de pytest**

## **P1.6 - Mis primeras pruebas unitarias**

### **1. Introducción**

Para realizar pruebas unitarias de forma básica debemos conocer qué es y para qué se usan las funciones en los lenguajes de programación.

### **2. Funciones**

Una función te permite definir un bloque de código reutilizable que se puede ejecutar muchas veces dentro de tu programa.

Las funciones en Python son componentes importantes en la programación que cuentan con una estructura que consta de dos principios.

- Principio de reutilización: puedes reutilizar una función varias veces y en distintos programas.
- Principio de modularización: te permite segmentar programas complejos con módulos más simples para depurar y programar con mayor facilidad.

En Python, una definición de función tiene las siguientes características:

1. La palabra clave `def`.
2. Un nombre de función
3. Paréntesis `()`, y dentro de los paréntesis los parámetros de entrada (opcionales).
4. Dos puntos `:`
5. Algún bloque de código para ejecutar
6. Una sentencia de retorno (opcional)

Un ejemplo, que además vamos a usar en nuestra práctica es el siguiente:

```
def suma(num1, num2):  
    return num1 + num2
```

En el código podemos llamarla las veces que nosotros necesitemos:

```
print(suma(3, 2))  
print("La suma de", 3, "+", 2, "es", suma(3, 2))  
tot = 100 + suma(25, 40)
```

### **3. Pruebas unitarias**

Las pruebas automáticas son consideradas una herramienta y una metodología indispensable para producir software de calidad.

Un conjunto de pruebas, es código que realiza pruebas a nuestro código. Las pruebas unitarias nos permiten verificar que nuestro código funciona como se espera.

Las pruebas unitarias son un conjunto de casos de prueba diseñados para verificar que cada “unidad” o componente de nuestro código funciona como se espera.

Cada prueba unitaria se ejecuta en un entorno aislado, lo que significa que no afecta a otras partes del código y viceversa.

Además de ejecutarse en la máquina del desarrollador, en entornos de trabajo profesional, estas se ejecutan en 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 pruebas implica que esta se probará una y otra vez cada que una funcionalidad sea agregada o un error sea corregido.

### 3.1. Pytest

Pytest es un framework con muchas funcionalidades, desde pruebas pequeñas hasta pruebas de gran escala como pruebas funcionales de aplicaciones y librerías. Ofrece la recolección automática de los tests, aserciones simples, soporte para fixtures, debugeo y mucho más...

### 3.2. Assert

La palabra `assert` en Python se refiere a un enunciado que verifica ciertas suposiciones sobre nuestro código. Si la suposición no es cierta, la afirmación falla y se genera una excepción.

Por ejemplo, si suponemos que una variable es mayor que cero, podemos usar `assert` para verificar esa suposición:

```
def funcion_ejemplo(x):  
    assert x > 0, "x no es mayor que cero"  
    return x * 2
```

En este ejemplo, si `x` es igual a 0 o menor que 0, se generará una excepción `AssertionError` con el mensaje “x no es mayor que cero”.

## 4. Creando el primer test

1. Para empezar, vamos a partir de la práctica 5, donde creamos un entorno virtual e instalamos pytest. Vamos a seguir los siguientes pasos para realizar nuestro primer test y ejecutarlo desde el terminal.
2. Abrir en Visual Studio Code nuestra carpeta de trabajo, vamos a crear una carpeta en `~/Documents/practica6`, activar el entorno virtual y comprobar que está instalado correctamente pytest.
3. Lo habitual es crear un directorio llamado `tests` que contenga los ficheros de pruebas y `src` para incluir los programas o módulos. Si hacéis esto, incluid en todas las carpetas del proyecto el fichero `__init__.py` vacío (*si lo hacéis desde el terminal: touch **init.py***).

Por ejemplo, si tenemos la siguiente estructura de nuestra carpeta o proyecto `practica6`:

```

\practica6
|
--src
| |--__init__.py
| |--main.py
--tests
| |--__init__.py
| |--test_main.py

```

4. Crear una carpeta que se llame `src`. En ella crearemos un nuevo fichero `main.py` con el siguiente contenido:

```

5. def suma(num1, num2):
6.     return num1 + num2
7.
8.
9. def main():
10.     print("La suma de 3 + 2 es {}".format(suma(3, 2)))
11.
12.     # Otras formas de hacer lo mismo:
13.     print("La suma de", 3, "+", 2, "es", suma(3, 2))
14.
15.     print("La suma de " + str(3) + " + " + str(2) + " es " +
16.           str(suma(3, 2)))
17.
18.     res = "La suma de "
19.     res += str(3)
20.     res += " + " + str(2)
21.     res += " es " + str(suma(3, 2))
22.     print(res)
23.
24.     print(f"La suma de 3 + 2 es {suma(3, 2)}")
25.
26. if __name__ == "__main__":
27.     main()

```

Si os fijáis en el código, no es más que una función que voy a llamar en mi función principal en varias ocasiones.

28. A continuación, nos creamos la carpeta `tests` y dentro de ella un fichero con el nombre `test_main.py` (*Pytest va a reconocer por defecto todos los programas que comiencen por `test_` cómo pruebas unitarias que debe realizar*). El contenido será el siguiente:

```

29. from src.main import suma
30.
31. def test_suma():
32.     assert suma(1, 1) == 2
33.     assert suma(0, 0) == 0
34.     assert suma(100, -100) == 0

```

Si os da problemas porque no encuentra los paquetes importados desde la prueba unitaria, agregamos manualmente el directorio raíz del proyecto a `sys.path`:

```

import sys
import os

# Agregar el directorio raíz del proyecto (practica6) al sys.path

```

```

sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

def test_suma():
    assert suma(1, 1) == 2
    assert suma(0, 0) == 0
    assert suma(100, -100) == 0

```

35. Para empezar, siempre debemos importar la función que deseamos probar del módulo dónde está definida.
36. Todas las pruebas serán también una función con el nombre `test_nombreFunciónAProbar`.
37. Si realizamos varias funciones para probar una misma función es recomendable añadir un texto explicativo de la prueba.
38. `assert` verifica que la expresión de la derecha es verdadera (`true`), sino generará una excepción.
39. Pytest capturará la excepción si se produce y la gestionará para mostrarnos los resultados.
40. Vamos a ejecutar las pruebas unitarias desde la terminal:
41. `> pytest`

El comando `pytest` nos muestra si las pruebas pasaron o no. Si queremos una información un poco más detallada usamos el parámetro `-v`:

```
> pytest -v
```

42. Se pueden usar las marcas para realizar múltiples pruebas sobre un determinado método (*marca `parametrize`*). En el mismo fichero `test_main.py` añadimos otra función:

```

43. import pytest
44. from src.main import suma
45.
46. def test_suma():
47.     assert suma(1, 1) == 2
48.     assert suma(0, 0) == 0
49.     assert suma(100, -100) == 0
50.
51. @pytest.mark.parametrize(
52.     "input_n1, input_n2, expected",
53.     [
54.         (1, 1, 2),
55.         (0, 0, 0),
56.         (100, -100, 0),
57.         (-15, -1, -16),
58.         (-3, 8, 5),
59.         (9, suma(-1, -2), 6)
60.     ]
61. )
62. def test_suma_params(input_n1, input_n2, expected):
63.     assert suma(input_n1, input_n2) == expected

```

Necesitamos importar las librerías de `pytest` en nuestro fichero de pruebas con `import pytest`.

64. Al volver a realizar el test obtendremos un resultado por cada tupla de parámetros probados:
65. `> pytest -v`
66. **Obligüemos a que se produzca un error**, por ejemplo modificando uno de los parámetros `expected` `(0, 0, 1)` y observemos lo que nos muestra `pytest`.
67. Pruébalo y vuelve a ejecutar los tests unitarios.