
—— Logging, Pytest, docker, metaclasses ——

LOGGING

El registro es una herramienta muy útil en la caja de herramientas de un programador. Puede ayudarlo a desarrollar una mejor comprensión del flujo de un programa y descubrir escenarios en los que quizás ni siquiera haya pensado durante el desarrollo.

Los registros brindan a los desarrolladores un par de ojos adicionales que observan constantemente el flujo por el que pasa una aplicación. Pueden almacenar información, como qué usuario o IP accedió a la aplicación. Si se produce un error, pueden proporcionar más información que un seguimiento de la pila al indicarle cuál era el estado del programa antes de que llegara a la línea de código donde se produjo el error.

Niveles de Logging

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

```
import logging

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

Shell

```
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

Config del Logging

```
import logging
```

```
logging.basicConfig(filename='app.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')  
logging.warning('This will get logged to a file')
```

Shell

```
root - ERROR - This will get logged to a file
```

¿Qué son los test?

Los tests son código que escribimos para comprobar que lo que estamos programando funciona como debería. Existen muchos tipos de test vamos a trabajar con unit test o pruebas unitarias, que se refieren a la comprobación individual de funciones y métodos.

Red-Green-Refactor

jamesshore.com/v2/blog/2005/red-green-refactor

1. **Think:** Figure out what test will best move your code towards completion. (Take as much time as you need. This is the hardest step for beginners.)
2. **Red:** Write a very small amount of test code. Only a few lines... usually no more than five. Run the tests and watch the new test fail: the test bar should turn red. (This should only take about 30 seconds.)
3. **Green:** Write a very small amount of production code. Again, usually no more than five lines of code. Don't worry about design purity or conceptual elegance. Sometimes you can just hardcode the answer. This is okay because you'll be refactoring in a moment. Run the tests and watch them pass: the test bar will turn green. (This should only take about 30 seconds, too.)
4. **Refactor:** Now that your tests are passing, you can make changes without worrying about breaking anything. Pause for a moment. Take a deep breath if you need to. Then look at the code you've written, and ask yourself if you can improve it. Look for duplication and other "code smells." If you see something that doesn't look right, but you're not sure how to fix it, that's okay. Take a look at it again after you've gone through the cycle a few more times. (Take as much time as you need on this step.) After each little refactoring, run the tests and make sure they still pass.
5. **Repeat:** Do it again. You'll repeat this cycle dozens of times in an hour. Typically, you'll run through several cycles (three to five) very quickly, then find yourself slowing down and spending more time on refactoring.

Pytest: Instalación

```
pip install pytest
```

```
pytest -h
```

¿Cómo se hacen test en Python?

Python trae por defecto la librería unittest que incluye todos los métodos y módulos necesarios para hacer las validaciones en nuestro código. A continuación te muestro un ejemplo de un test para una función que suma dos números

funciones.py > ...

```
1 def suma(numero_a, numero_b):
2     try:
3         return numero_a + numero_b
4     except Exception:
5         return None
6
7
8
9
```

tests > test_funciones_suma.py > ...

```
1 from unittest import TestCase
2 from funciones import suma
3
4
5 class TestFuncionSuma(TestCase):
6     def test_funcion_suma(self):
7         sum = suma(2, 3)
8         self.assertIsNotNone(sum)
9         self.assertEqual(sum, 5)
```


¿Cómo se hacen test en Python?

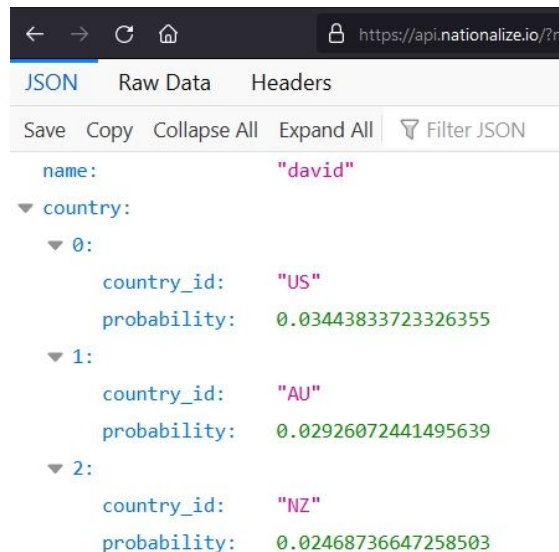
Pero, por ejemplo: ¿qué pasa si la función que queremos testear hace una consulta a una API? Nuestro test podría fallar si la API no está disponible, si falla la conexión e incluso si la API fue actualizada causando que la respuesta sea distinta.

Nuestras pruebas unitarias no deben estar ligadas a esta clase de situaciones, ya que el objetivo de esta clase de tests es comprobar que un segmento específico del código funcione correctamente.

¿Qué son los mocks?

Los mocks son objetos “dummy” (o de muestra) con los que podemos simular objetos cuyo funcionamiento es más complejo. No es recomendable utilizar el objeto real como parte de la prueba.

```
def get_nacionality_by_name(name):  
    url = f"https://api.nationalize.io/?name={name}"  
    response = requests.get(url).text  
    country_code = json.loads(response)["country"][0]["country_id"]  
    return country_code
```



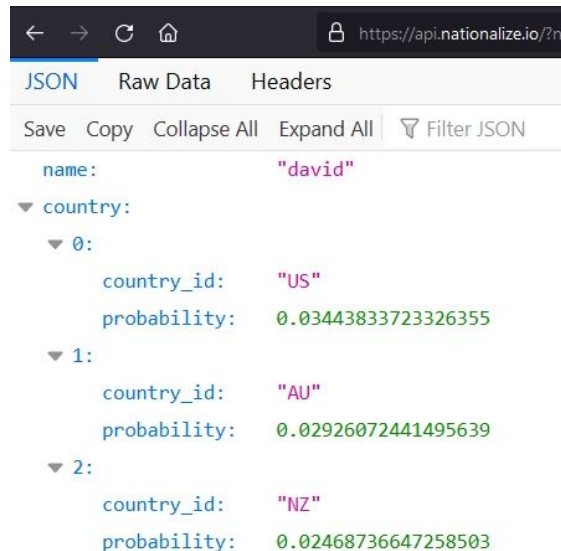
The screenshot shows a web browser window with the URL <https://api.nationalize.io/?name=david>. The browser's developer tools are open, displaying the JSON response. The response is a JSON object with a 'name' field set to 'david' and a 'country' field containing an array of three objects. Each object in the array represents a country with a 'country_id' and a 'probability'.

| country_id | probability |
|------------|---------------------|
| US | 0.03443833723326355 |
| AU | 0.02926072441495639 |
| NZ | 0.02468736647258503 |

Cómo usar un mock

Los mocks son objetos “dummy” (o de muestra) con los que podemos simular objetos cuyo funcionamiento es más complejo. No es recomendable utilizar el objeto real como parte de la prueba.

```
def get_nacionalidad_by_name(name):  
    url = f"https://api.nationalize.io/?name={name}"  
    response = requests.get(url).text  
    country_code = json.loads(response)["country"][0]["country_id"]  
    return country_code
```



Crea tu propia librería con python en 3 pasos

1. Construir el paquete
 - a. Nombre_su_libreria.py
 - b. Setup.py
 - c. Construir distribution wheel
2. Publicarlos en test.pypi.org y luego probarlo.
 - a. Crear una cuenta
 - b. Crear un API token
 - c. Pip install nombre_su_libreria
3. Importarlo.
 - a. Import nombre_su_libreria

1. Construir el paquete

- a. `nombre_su_libreria.py` <- acá consolidaremos nuestros paquetes (creatividad = límite)

```
def hola_mundo(name="Mundo"):  
    return "Librería creada e importada exitosamente. Hola, {name}".format(name = name)
```

- b. `setup.py` <- acá hacemos la magia para que python nos lo estructure

```
from setuptools import find_packages, setup  
setup(  
    name='nombre_su_libreria', ## Este nombre debe ser único y no estar indexado  
    version='0.2',  
    packages=find_packages(),  
    py_modules=['nombre_su_libreria'] ## Este debe ser el nombre del archivo *.py  
)
```

1. Construir el paquete

1. Construir el paquete *(cont.)*

- c. Construir un "Binary distribution wheel", el formato para distribuir librerías

En terminal o cmd, seguir los siguientes pasos

1. **Sugerencia:** Crear un ambiente separado en alguna carpeta donde realizaremos pruebas:
 - i. `pip install virtualenv`
 - ii. `virtualenv mypython`
 - iii. `source mypython/bin/activate`
2. Dirigirse a la carpeta donde están `setup.py` y `nombre_su_libreria.py`
 - i. `python3 -m pip install --upgrade setuptools wheel`
 - ii. `python3 setup.py sdist bdist_wheel`