BACKEND CON PYTHON

PRUEBAS Y MANEJO DEL ERROR

- Pruebas
  - *Unitarias*
  - *Integración*
- Manejo del error

PRUEBAS Y MANEJO DEL ERROR

- Pruebas
  - *Unitarias*
  - *Integración*
- Manejo del error

PRUEBAS

Las pruebas son una parte fundamental en el proceso de desarrollo de software, y desempeñan un papel crítico en el proceso de garantizar la calidad y la confiabilidad del software:

- Detección temprana de errores permiten identificar errores y defectos en el código durante las etapas iniciales del desarrollo reduciendo así los costos.
- Aseguramiento de la calidad: ayudan a garantizar que el software cumpla con los requisitos especificados y funcione según lo esperado
- Validación de requisitos: aseguran que el software esté cumpliendo todos los requisitos y requerimientos por los que fue diseñado e implementado
- Mejor mantenibilidad: al documentar las pruebas, se describe el comportamiento de cada uno de los módulos. La documentación facilita la futura actualización y modificación del código.

PRUEBAS

Hay 4 niveles de pruebas fundamentalmente:

- Unitarias
- Integración
- Sistema
- Aceptación

En este curso entraremos a ver solo las primeras 2, que se pueden trabajar con Python y desde VSCode.

PRUEBAS Y MANEJO DEL ERROR

- Pruebas
  - *Unitarias*
  - *Integración*
- Manejo del error

PRUEBAS UNITARIAS

Como su nombre lo indica, son pruebas que se centran en probar unidades individuales de software.

Por unidades individuales de software, podemos pensar en funciones y/o métodos.

Cuando estamos hablando de la Programación Orientada a Objetos, las pruebas unitarias se hacen sobre los objetos de las clases que están declaradas, idealmente una clase de prueba por cada clase de la lógica. Para asegurar una mayor cobertura de código, las pruebas unitarias sobre una clase deben tener al menos un caso de prueba por cada método.

Las pruebas unitarias son automatizadas: se ejecutan automáticamente durante el proceso de desarrollo de software.

PRUEBAS UNITARIAS

Ventajas de las pruebas unitarias:

- Detección temprana de errores
- Facilitan el mantenimiento del código
- Mejora la calidad del código

Desventajas:

- Aumentan el tiempo de desarrollo
- No garantizan la ausencia de errores

PRUEBAS UNITARIAS

Para hacer las pruebas unitarias en Python vamos a retomar el ejemplo de Perros, Gatos y la guardería.

De momento nos olvidamos de Flask y DB para centrarnos en las pruebas. Primero haremos algunas aclaraciones sobre la estructura del proyecto

La carpeta raíz se llama proyecto y tiene dos carpetas: una llamada modelos y la otra llamada pruebas. Cada carpeta, a su vez tiene un archivo llamado __init__.py, este archivo es generalmente vacío y lo usamos para indicar que esa carpeta es un paquete dentro del proyecto.

```
proyecto/
│
├── modelos/
│   ├── __init__.py
│   ├── animal.py
│   ├── perro.py
│   ├── gato.py
│   ├── guarderia.py
│
├── pruebas/
│   ├── __init__.py
│   ├── test_animal.py
│   ├── test_perro.py
│   ├── test_gato.py
│   ├── test_guarderia.py
│
└── main.py
```

PRUEBAS UNITARIAS

Tenemos 2 paquetes separados, aunque todos los archivos .py podrían estar en la misma carpeta, hemos decidido separarlo por buenas prácticas de desarrollo:
La lógica no debe mezclarse con las pruebas:

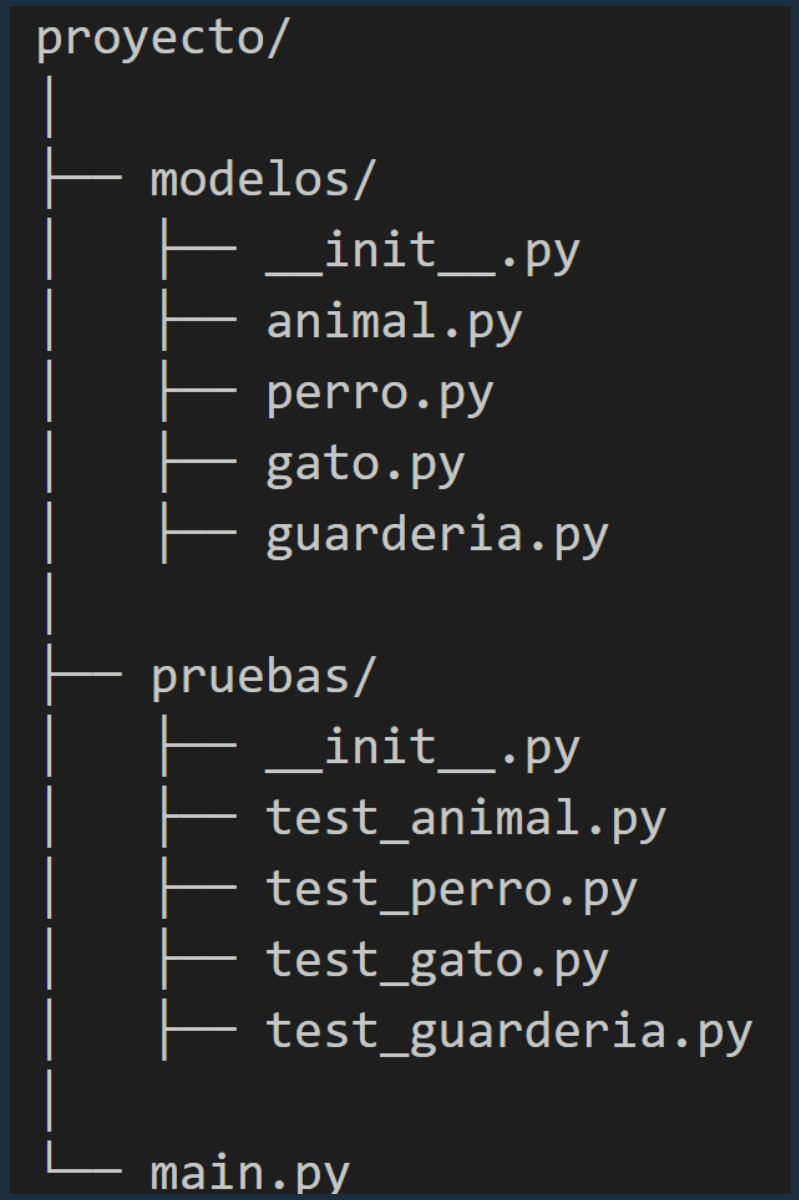
- Facilidad de modificación
- Facilidad de lectura

Mas adelante veremos que las clases de prueba, es decir los archivos que tienen un nombre que empieza por test_, necesitarán usar las clases del modelo. Para facilitar la importación, dado que están en carpetas independientes, convertimos las carpetas en paquetes poniendo el archivo __init__.py

```
proyecto/
│
├── modelos/
│   ├── __init__.py
│   ├── animal.py
│   ├── perro.py
│   ├── gato.py
│   ├── guarderia.py
│
├── pruebas/
│   ├── __init__.py
│   ├── test_animal.py
│   ├── test_perro.py
│   ├── test_gato.py
│   ├── test_guarderia.py
│
└── main.py
```

PRUEBAS UNITARIAS

Como podemos ver, por cada clase que teníamos originalmente (animal, perro, gato, guardería), vamos a crear un archivo Python con el nombre test_nombre, como se puede ver ahí test_animal, test_perro, test_gato y test_guarderia.
Todos estos test podrían estar en un único archivo Python, pero por buenas prácticas de desarrollo, generamos un archivo de test por cada archivo del modelo.
Por último, está el archivo main.py, al que le podemos pedir que ejecute las pruebas.



PRUEBAS UNITARIAS

Para realizar cada una de las pruebas, vamos a importar y utilizar la librería unittest de Python.

```python
import unittest
from modelos.gato import Gato

class TestGato(unittest.TestCase):
    def test_hacer_sonido(self):
        gato = Gato(nombre="Mittens", raza="Siamés", edad=1, peso=10)
        self.assertEqual(gato.hacer_sonido(), "¡Miau!")

    def test_modificar_edad(self):
        gato = Gato(nombre="Mittens", raza="Siamés", edad=1, peso=10)
        gato.modificar_edad(2)
        self.assertEqual(gato.obtener_edad(), 2)

    def test_modificar_peso(self):
        gato = Gato(nombre="Mittens", raza="Siamés", edad=1, peso=10)
        gato.modificar_peso(12)
        self.assertEqual(gato.obtener_peso(), 12)
```

```python
import unittest
from modelos.perro import Perro

class TestPerro(unittest.TestCase):
    def test_hacer_sonido(self):
        perro = Perro(nombre="Zeus", raza="Rottweiler", edad=3, peso=45)
        self.assertEqual(perro.hacer_sonido(), "¡Guau!")

    def test_modificar_edad(self):
        perro = Perro(nombre="Zeus", raza="Rottweiler", edad=3, peso=45)
        perro.modificar_edad(4)
        self.assertEqual(perro.obtener_edad(), 4)

    def test_modificar_peso(self):
        perro = Perro(nombre="Zeus", raza="Rottweiler", edad=3, peso=45)
        perro.modificar_peso(50)
        self.assertEqual(perro.obtener_peso(), 50)
```

A continuación, discutiremos más al detalle estas pruebas.

PRUEBAS UNITARIAS

Dado que perro y gato heredan de animal, sus pruebas unitarias serán muy similares, por lo que acá solo analizaremos test_perro.py

- Importamos unittest y perro
- Creamos un método de prueba por cada método existente en la clase a probar (perro).
- Dentro de cada método creamos un objeto para poder probar sus métodos en un ambiente controlado: conocemos todos los valores.

```python
import unittest
from modelos.perro import Perro

class TestPerro(unittest.TestCase):
    def test_hacer_sonido(self):
        perro = Perro(nombre="Zeus", raza="Rottweiler", edad=3, peso=45)
        self.assertEqual(perro.hacer_sonido(), "¡Guau!")

    def test_modificar_edad(self):
        perro = Perro(nombre="Zeus", raza="Rottweiler", edad=3, peso=45)
        perro.modificar_edad(4)
        self.assertEqual(perro.obtener_edad(), 4)

    def test_modificar_peso(self):
        perro = Perro(nombre="Zeus", raza="Rottweiler", edad=3, peso=45)
        perro.modificar_peso(50)
        self.assertEqual(perro.obtener_peso(), 50)
```

- Al conocer los valores, podemos realizar modificaciones y predecir el resultado. Comparamos el resultado obtenido con el que predecimos a través de la instrucción assertEqual, propia de unittest

PRUEBAS UNITARIAS

Test_animal, se supone que sería la clase de pruebas sobre la clase abstracta animal, pero dado que no podemos instanciar objetos de clases abstractas, no podemos crear el objeto animal para realizarle pruebas unitarias. De igual manera, los métodos de Animal serán probado desde las clases que lo heredan como perro y gato.

Test_guarderia, será una prueba de integración de la que hablaremos más adelante.

PRUEBAS UNITARIAS

Para ejecutar las pruebas, en el Terminal podemos usar el comando:

```
                                                                python -m unittest pru
ebas.test_perro
>> C:\Users\[redacted]\pruebas unitarias>
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Si queremos ejecutar todas las pruebas podemos ejecutar un comando del tipo:

```
PS C:\Users\[redacted]\pruebas unitarias> python -m unittest di
cover -p 'test_*.py'
----------------------------------------------------------------------
Ran 8 tests in 0.002s
```

Para que la instrucción discover funcione y encuentre todas las pruebas, es importante que todas tengan un nombre que empiece con test_

PRUEBAS Y MANEJO DEL ERROR

- Pruebas
  - *Unitarias*
  - *Integración*
- Manejo del error

PRUEBAS DE INTEGRACIÓN

Como su nombre lo indica, son pruebas que se centran en probar la interacción (integración) entre diferentes unidades del software.

Las pruebas de integración son automatizadas: se ejecutan automáticamente durante el proceso de desarrollo de software.

Es importante resaltar que las pruebas de integración y las unitarias son complementarias: se utilizan en conjunto para garantizar la calidad del software.

PRUEBAS DE INTEGRACIÓN

Ventajas de las pruebas de integración:

- Identifican problemas de interoperabilidad
- Validan la correcta transferencia de información (flujo de datos) entre diferentes componentes del sistema
- Da una mayor confianza en la funcionalidad del sistema

Desventajas:

- Suelen ser más costosas de implementar que las unitarias
- En ocasiones, es difícil encontrar la raíz del problema, mucho más que en las unitarias.

PRUEBAS DE INTEGRACIÓN

Para la implementación en Python, también se usa unittest, solo que en la clase test_, se importa más de 1 clase del modelo, como por ejemplo test_guarderia que importa guarderia, perro y gato.

En este ejemplo se usa el assertIn, otra función de unittest. Para conocer todas las aserciones deben revisar la documentación de la librería.

Por último, la instrucción __name__==‘main’: unittest.main()
Asegura que se ejecute únicamente cuando el programa está siendo ejecutado y no cuando está siendo importado en otro módulo

```python
import unittest
from modelos.guarderia import Guarderia
from modelos.perro import Perro
from modelos.gato import Gato

class TestGuarderia(unittest.TestCase):
    def test_agregar_animal(self):
        guarderia = Guarderia()

        perro = Perro(nombre="Zeus", raza="Rottweiler", edad=3, peso=45)
        gato = Gato(nombre="Mittens", raza="Siamés", edad=1, peso=10)

        guarderia.agregar_animal(perro)
        guarderia.agregar_animal(gato)

        self.assertIn(perro, guarderia.obtener_animales())
        self.assertIn(gato, guarderia.obtener_animales())

    def test_lista_animales(self):
        guarderia = Guarderia()

        perro = Perro(nombre="Zeus", raza="Rottweiler", edad=3, peso=45)
        gato = Gato(nombre="Mittens", raza="Siamés", edad=1, peso=10)

        guarderia.agregar_animal(perro)
        guarderia.agregar_animal(gato)

        self.assertIn(perro, guarderia.obtener_animales())
        self.assertIn(gato, guarderia.obtener_animales())

        guarderia.obtener_animales()[0].modificar_edad(4)
        self.assertEqual(guarderia.obtener_animales()[0].obtener_edad(), 4)

        guarderia.obtener_animales()[1].modificar_peso(12)
        self.assertEqual(guarderia.obtener_animales()[1].obtener_peso(), 12)

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

PRUEBAS Y MANEJO DEL ERROR

- Pruebas
  - *Unitarias*
  - *Integración*
- Manejo del error

MANEJO DEL ERROR

Ya hablamos de las pruebas y de cómo nos ayudan a encontrar errores en las aplicaciones que desarrollamos. Sin embargo, nos encontraremos en situaciones implementando el modelo, en las que tendremos que prepararnos para manejar algunos errores.

Para ejemplificar estas situaciones en las que tenemos que aprender a manejar errores utilizaremos la división por 0.

Es decir, si tenemos algunas instrucciones que realicen operaciones y en algún momento dividimos por una variable y esta puede ser cero, debemos indicarle a nuestro programa que este error puede suceder y debe prepararse para manejarlo.

Para esto existen las excepciones.

MANEJO DEL ERROR

Comencemos con el siguiente código:

```
def dividir(a, b):
    if b == 0:
        raise ValueError("No se puede dividir por cero")
    return a / b
```

La función recibe dos parámetros y divide el uno entre el otro, pero si el divisor es == 0 se generará un error en la ejecución.
Para evitar este error de ejecución, se creó el if que identifica la condición en la que se pude presentar el error, y en caso de que se presente, lanza (raise) un ValueError.
Al lanzar el ValueError, las otras funciones que invoquen la función dividir, podrán darle un manejo al error sin deter la ejecución del programa.

MANEJO DEL ERROR

Teniendo en cuenta que la función dividir(a,b), puede lanzar una expción de tipo ValueError, cada vez que haya una función que haga un llamado a dividir, debe darle un manejo a su excepción.

```python
def realizar_operacion():
    try:
        resultado = dividir(10, 0)
        print(f"Resultado de la operación: {resultado}")
    except ValueError as e:
        print(f"Error al realizar la operación: {e}")
```

Mientras que dividir arrojaba la excepción con la instrucción “raise”, la función realizar_operación captura la excepción por medio de la instrucción “except ValueError” y le asigna la variable e usando “as e”

MANEJO DEL ERROR

En este nuevo ejemplo vemos todas las partes posibles en el manejo de una excepción:

- try: contiene las instrucciones que hacen el llamado a la función que arroja excepción
- except: captura la excepción en caso de que haya sido arrojada y le da algún manejo.
- else: En caso de que no se haya arrojado la excepción indica como proceder.
- finally: se ejecuta siempre, sin importar si se arrojó la excepción o no

```python
def realizar_operacion2(a, b):
    try:
        resultado = dividir(a, b)
    except ValueError as e:
        print(f"Error: {e}")
    else:
        print(f"Resultado de la operación: {resultado}")
    finally:
        print("Este bloque siempre se ejecuta")
```

MANOS A LA OBRA

Taller 3 disponible en BloqueNeon