

# **Unit Testing, Stubs and Mocks**

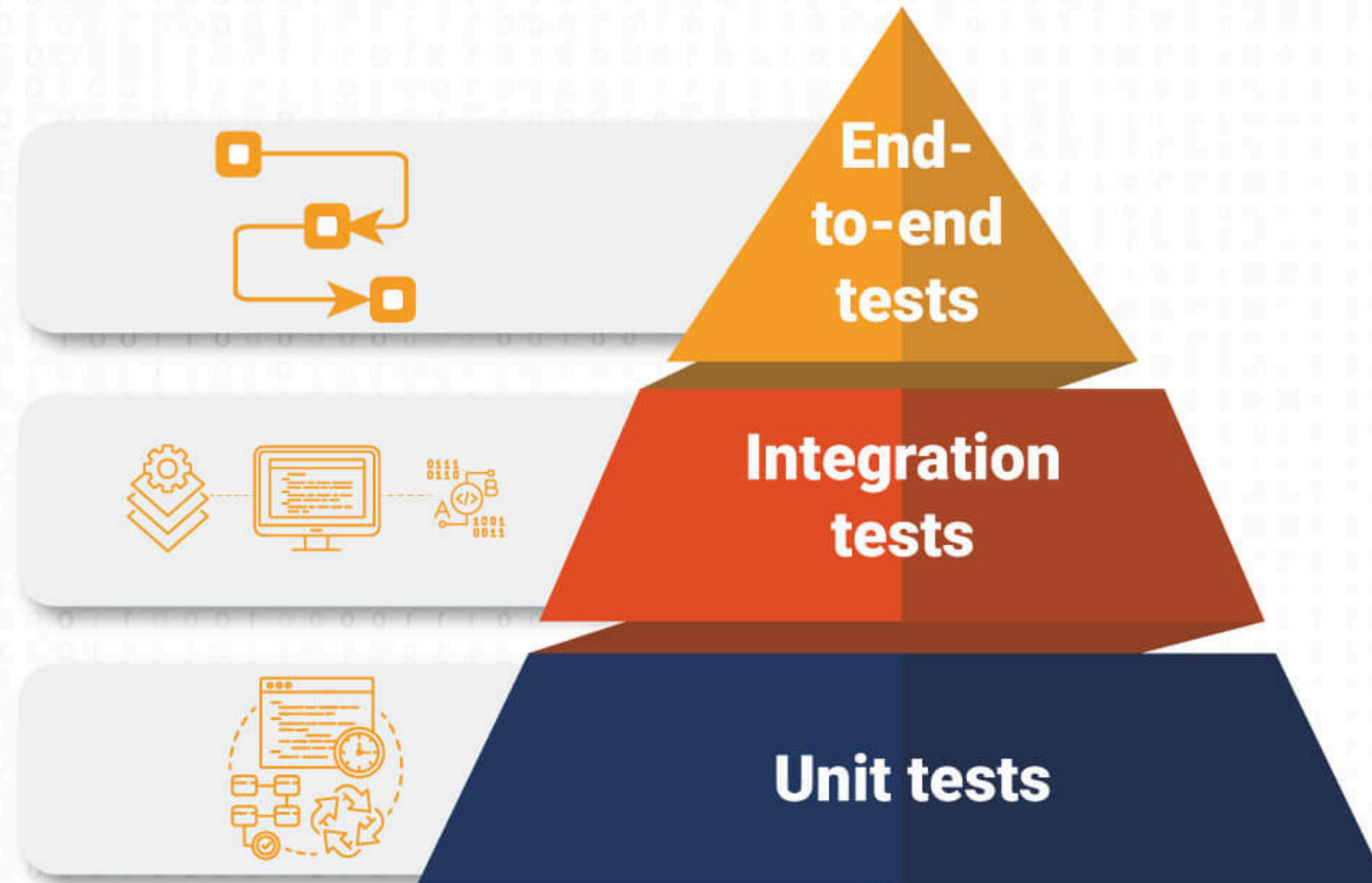
**IIC3745 - Testing**

**Maximiliano Narea Carvajal**

# Conceptos básicos

# ¿Qué tipos de tests hay?

## Conceptos básicos



# ¿Qué es un test unitario?

## Conceptos básicos

- Definición: Las pruebas unitarias son un tipo de prueba de software que verifica el funcionamiento correcto de una unidad individual de código.
- Propósito: Asegurar que cada parte del código funcione correctamente de manera aislada.

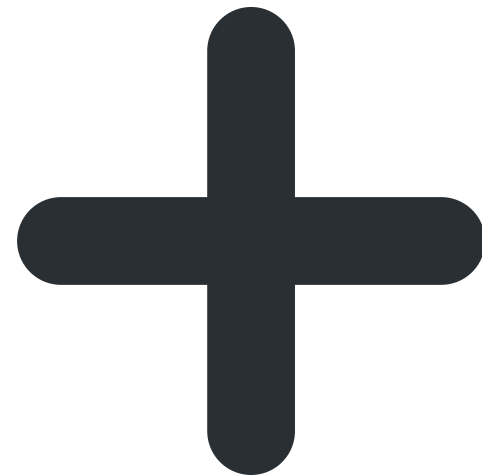
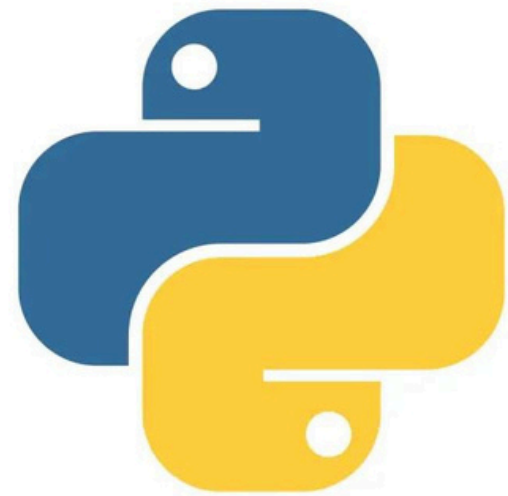
# ¿Cuáles son las ventajas de este tipo de test?

## Conceptos básicos

- Detectar errores en etapas tempranas del desarrollo.
- Facilitar el **mantenimiento y refactorización** del código.
- Proveer **documentación** sobre el **funcionamiento** del código.
- Incrementar la confianza en la calidad del software.

# Framework para practicar Tests

## Conceptos básicos



**unittest**

[Documentación Unittest](#)

# Unit Testing

# ¿Cómo se ve un test genérico?

## Unit Testing

```
1 import unittest
2
3 def suma(a, b):
4     return a + b
5
6 class TestSuma(unittest.TestCase):
7
8     def test_suma_positivos(self):
9         self.assertEqual(suma(2, 3), 5)
10
11     def test_suma_negativos(self):
12         self.assertEqual(suma(-1, -1), -2)
13
14     def test_suma_cero(self):
15         self.assertEqual(suma(0, 0), 0)
16
17     def test_suma_positivo_negativo(self):
18         self.assertEqual(suma(10, -5), 5)
19
20 if __name__ == '__main__':
21     unittest.main()
22
```



# ¿De qué partes se compone un test?

## Unit Testing

- Arrange
- Act
- Assert

```
1 import unittest
2
3 class WalletTest(unittest.TestCase):
4
5     def test_deposit(self):
6         self.wallet = Wallet("Max")
7         self.wallet.deposit(100)
8         self.wallet.deposit(50)
9         balance = self.wallet.total_balance()
10        self.assertEqual(150, balance)
11
12 unittest.main()
13
```

# Tipos de Assert

## Unit Testing

```
1
2 class TestEjemplo(unittest.TestCase):
3
4     def test_varios_asserts(self):
5         # assertEquals
6         self.assertEqual(suma(2, 3), 5, "La suma de 2 y 3 debería ser 5")
7
8         # assertTrue
9         self.assertTrue(es_par(4), "4 debería ser un número par")
10
11        # assertFalse
12        self.assertFalse(es_par(5), "5 no debería ser un número par")
13
14        # assertIn
15        self.assertIn(3, [1, 2, 3], "3 debería estar en la lista [1, 2, 3]")
16
17        # assertIsNone
18        a = None
19        self.assertIsNone(a, "a debería ser None")
20
```

# ¿Cómo ordenamos nuestros tests?

## Unit Testing

Supongamos que tenemos tests de la siguiente forma.

Es importante que podamos disponer de una forma de hacerlos más legibles y menos verbosos.

```
1 class WalletTest(unittest.TestCase):
2
3     def test_deposit(self):
4         self.wallet1 = Wallet("1")
5         self.wallet2 = Wallet("2")
6         self.wallet1.open_resources()
7         self.wallet2.open_resources()
8         self.wallet1.deposit(100)
9         self.wallet1.deposit(50)
10        balance = self.wallet1.total_balance()
11        self.assertEqual(150, balance)
12        self.wallet1.close_resources()
13        self.wallet2.close_resources()
14
15    def test_withdraw(self):
16        self.wallet1 = Wallet("1")
17        self.wallet2 = Wallet("2")
18        self.wallet1.open_resources()
19        self.wallet2.open_resources()
20        self.wallet2.deposit(100)
21        self.wallet2.withdraw(30)
22        balance = self.wallet2.total_balance()
23        self.assertEqual(70, balance)
24        self.wallet1.close_resources()
25        self.wallet2.close_resources()
```

# Los métodos setUp y tearDown

## Unit Testing

- setUp nos permite reusar el código para instanciar.
- tearDown nos permite “limpiar” recursos.

```
1 class Wallet:
2     def __init__(self, id):
3         self.id = id
4         self.balance = 0
5         self.resources_open = False
6
7     def deposit(self, amount):
8         self.balance += amount
9
10    def withdraw(self, amount):
11        self.balance -= amount
12
13    def total_balance(self):
14        return self.balance
15
16    def open_resources(self):
17        self.resources_open = True
18
19    def close_resources(self):
20        self.resources_open = False
```

```
1 class WalletTest(unittest.TestCase):
2
3     def setUp(self):
4         # Se ejecuta antes de cada prueba
5         self.wallet1 = Wallet("1")
6         self.wallet2 = Wallet("2")
7         self.wallet1.open_resources()
8         self.wallet2.open_resources()
9
10    def tearDown(self):
11        # Se ejecuta después de cada prueba
12        self.wallet1.close_resources()
13        self.wallet2.close_resources()
14
15    def test_deposit(self):
16        self.wallet1.deposit(100)
17        self.wallet1.deposit(50)
18        balance = self.wallet1.total_balance()
19        self.assertEqual(150, balance)
20
21    def test_withdraw(self):
22        self.wallet2.deposit(100)
23        self.wallet2.withdraw(30)
24        balance = self.wallet2.total_balance()
25        self.assertEqual(70, balance)
```

**F.I.R.S.T.**

# ¿Cómo hacemos buenos tests?

## F.I.R.S.T.

- F = Fast: Deben ejecutarse rápido
- I = Isolated: Único motivo del fallo, independiente de los demás tests
- R = Repetible: Se pueden ejecutar N veces y deben obtener el mismo resultado
- S = Self-Validating: Resultado booleano y automático
- T = Timely: Construido en el momento oportuno, idealmente antes de implementar el método (Test Driven Development)

# ¿Cómo hacemos buenos tests?

## F.I.R.S.T.

- El **nombre del test** debe ser **lo más descriptivo posible**, de forma que si el test falla, solo leyendo el nombre el desarrollador tenga ya una idea del posible problema.
- **No probar varios métodos al mismo tiempo**, cuando falle al desarrollador le costará más entender qué método es el que ocasionó la falla.
- No es recomendable tener **dependencias externas**.
- Debiera permitir entender el comportamiento del código en general. (Docs)



# Mocks & Stubs

# ¿En qué consisten?

## Mocks & Stubs

- Mocks: Objetos simulados que permiten verificar interacciones entre componentes. Se usan para reemplazar partes del sistema bajo prueba y verificar que se comportan correctamente.
- Stubs: Versiones simplificadas de componentes que devuelven datos predefinidos. Se usan para proporcionar respuestas controladas a dependencias del sistema bajo prueba.

# ¿Por qué usar Mocks y Stubs?

## Mocks & Stubs

- Aislar la unidad de código que se está probando.
- Simular comportamientos de dependencias externas.
- Facilitar la prueba de componentes en diferentes escenarios.
- Aumentar la cobertura de pruebas sin necesidad de configuraciones complejas.

# ¿Por qué usar Mocks y Stubs?

## Mocks & Stubs

```
1 import unittest
2 from unittest.mock import MagicMock
3
4 class MiClase:
5     def llamada_a_api(self):
6         pass
7
8     def metodo_a_probar(self):
9         return self.llamada_a_api()
10
11 class TestMiClase(unittest.TestCase):
12     def test_metodo_a_probar(self):
13         instancia = MiClase()
14         instancia.llamada_a_api = MagicMock(return_value=
15                                             '{"code': 200, 'message': 'Success!'}")
16         resultado = instancia.metodo_a_probar()
17         instancia.llamada_a_api.assert_called_once()
18         self.assertEqual(resultado, '{"code': 200, 'message': 'Success!'}",
19                           "Debería devolver el retorno de una api.")
```

# Coverage

# ¿En qué consiste?

## Coverage

- Es una medida que nos permite determinar qué tanto código hemos probado.
  - Existen diversas métricas, pero hoy vamos a revisar solo dos.
  - En general, se apunta a tener un porcentaje de coverage alto, ya que indica que hemos revisado el comportamiento de ciertas partes del código.
- (disclaimer)

# Statement Coverage

## Coverage

Probamos la mayor cantidad de líneas de código

En este ejemplo, basta con testear `open_resources()`




```
1 class Wallet:
2     def __init__(self, id):
3         self.id = id
4         self.balance = 0
5         self.resources_open = False
6
7     def deposit(self, amount):
8         self.balance += amount
9
10    def withdraw(self, amount):
11        self.balance -= amount
12
13    def total_balance(self):
14        return self.balance
15
16    def open_resources(self):
17        self.resources_open = True
18
19    def close_resources(self):
20        self.resources_open = False
```

# Branch Coverage

## Coverage

Probamos todas las condiciones posibles.

En este ejemplo se requieren por lo menos dos tests distintos



```
1 def test_function(x, y):  
2     if x > y:  
3         return true  
4     else:  
5         return false
```