



# UNIT TESTING



# Introduzione al Unit Testing

- Verifica del corretto funzionamento di singole unità di codice
- Un'unità è tipicamente una funzione, un metodo o una classe
- Isolato dalle dipendenze esterne (database, API, filesystem)
- Automatizzato e ripetibile
- Benefici:
  - Documentazione eseguibile
  - Prevenzione di regressioni
  - Miglioramento del design del codice
  - Confidenza nelle modifiche
- Framework popolari:
  - Python: `unittest`, `pytest`
  - JavaScript: `Jest`, `Mocha`





# Anatomia di un Unit Test

- **Setup:** preparazione dell'ambiente di test
- **Esecuzione:** chiamata alla funzione o al metodo da testare
- **Verifica:** controllo che i risultati siano quelli attesi
- **Teardown:** pulizia dell'ambiente dopo il test

Perché fare Unit Testing?

## Vantaggi immediati

- Individuazione precoce dei bug
- Documentazione vivente del comportamento atteso del codice
- Facilita il refactoring e la manutenzione

## Vantaggi a lungo termine

- Maggiore fiducia nel codice
- Riduzione del "debito tecnico"
- Sviluppo più rapido di nuove funzionalità
- Onboarding più semplice per nuovi sviluppatori

# Principi del buon Unit Testing

## FIRST

- **Fast:** i test devono essere veloci
- **Independent:** ogni test deve essere indipendente dagli altri
- **Repeatable:** i test devono dare lo stesso risultato ad ogni esecuzione
- **Self-Validating:** i test devono determinare autonomamente se hanno successo o no
- **Timely:** i test dovrebbero essere scritti prima o insieme al codice

## Le 3 A

- **Arrange:** preparazione dell'ambiente
- **Act:** esecuzione del codice da testare
- **Assert:** verifica dei risultati



A hand-drawn decorative border in black ink, featuring various loops, swirls, and flourishes that frame the central text. The border is irregular and artistic, with some lines extending outwards from the corners and sides.

# UNIT TESTING CON Python

# Unit Test in Python con unittest

```
1 # file: calcoli.py
2 def somma(a, b):
3     return a + b
4
5 def moltiplica(a, b):
6     return a * b
7
8 def dividi(a, b):
9     if b == 0:
10         raise ValueError("Divisione per zero non consentita")
11     return a / b
```

```
1 # file: test_calcoli.py
2 import unittest
3 from calcoli import somma, moltiplica, dividi
4
5 class TestCalcoli(unittest.TestCase):
6
7     def test_somma(self):
8         self.assertEqual(somma(5, 3), 8)
9         self.assertEqual(somma(-1, 1), 0)
10        self.assertEqual(somma(0, 0), 0)
11
12    def test_moltiplica(self):
13        self.assertEqual(moltiplica(4, 3), 12)
14        self.assertEqual(moltiplica(0, 5), 0)
15        self.assertEqual(moltiplica(-2, 3), -6)
16
17    def test_dividi(self):
18        self.assertEqual(dividi(10, 2), 5)
19        self.assertEqual(dividi(0, 5), 0)
20        self.assertAlmostEqual(dividi(5, 3), 1.6666666666666667)
21
22    def test_dividi_zero(self):
23        with self.assertRaises(ValueError):
24            dividi(10, 0)
25
26 if __name__ == "__main__":
27     unittest.main()
```

# unittest documentazione

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x)</code> is True
<code>assertFalse(x)</code>	<code>bool(x)</code> is False
<code>assertIs(a, b)</code>	<code>a</code> is <code>b</code>
<code>assertIsNot(a, b)</code>	<code>a</code> is not <code>b</code>
<code>assertIsNone(x)</code>	<code>x</code> is None
<code>assertIsNotNone(x)</code>	<code>x</code> is not None
<code>assertIn(a, b)</code>	<code>a</code> in <code>b</code>
<code>assertNotIn(a, b)</code>	<code>a</code> not in <code>b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

# Unit Test in Python con pytest

```
1 # file: calcoli.py
2 def somma(a, b):
3     return a + b
4
5 def moltiplica(a, b):
6     return a * b
7
8 def dividi(a, b):
9     if b == 0:
10         raise ValueError("Divisione per zero non consentita")
11     return a / b
```

```
1 # Non è necessario ereditare da classi specifiche!
2 # Installazione: pip install pytest
3
4 # La stessa classe Calculator definita prima
5 class Calculator:
6     def add(self, a, b):
7         return a + b
8
9     def divide(self, a, b):
10        if b == 0:
11            raise ValueError("Impossibile dividere per zero")
12        return a / b
13
14 # I test con pytest sono più semplici e leggibili
15 def test_add():
16     calc = Calculator()
17     assert calc.add(2, 3) == 5
18
19 def test_divide():
20     calc = Calculator()
21     assert calc.divide(10, 2) == 5
22
23 import pytest
24
25 def test_divide_by_zero():
26     calc = Calculator()
27     with pytest.raises(ValueError):
28         calc.divide(10, 0)
```



A hand-drawn decorative border in black ink, featuring various loops, swirls, and lines that frame the central text. The border is irregular and artistic, with some lines extending outwards from the corners.

# **UNIT TESTING CON Javascript**

# Unit Testing in JavaScript/Node.js (Jest)

```
1 // Installazione: npm install --save-dev jest
2
3 // La classe che vogliamo testare - calculator.js
4 class Calculator {
5   add(a, b) {
6     return a + b;
7   }
8
9   divide(a, b) {
10    if (b === 0) {
11      throw new Error("Impossibile dividere per zero");
12    }
13    return a / b;
14  }
15 }
16
17 module.exports = Calculator;
```

- **Avvio del testing con:**

- a. `npm run test`

```
1 // Il file di test - calculator.test.js
2 const Calculator = require('./calculator');
3
4 describe('Calculator', () => {
5   let calc;
6
7   beforeEach(() => {
8     // Questo viene eseguito prima di ogni test
9     calc = new Calculator();
10  });
11
12  test('should add two numbers correctly', () => {
13    // Arrange e Act
14    const result = calc.add(2, 3);
15
16    // Assert
17    expect(result).toBe(5);
18  });
19
20  test('should divide two numbers correctly', () => {
21    const result = calc.divide(10, 2);
22    expect(result).toBe(5);
23  });
24
25  test('should throw error when dividing by zero', () => {
26    // Verifichiamo che una funzione sollevi un'eccezione
27    expect(() => calc.divide(10, 0)).toThrow("Impossibile dividere per
    zero");
28  });
29 });
```

# Mocha (test runner) e Chai (libreria di asserzioni)

```
1 // Installazione: npm install --save-dev jest
2
3 // La classe che vogliamo testare - calculator.js
4 class Calculator {
5   add(a, b) {
6     return a + b;
7   }
8
9   divide(a, b) {
10    if (b === 0) {
11      throw new Error("Impossibile dividere per zero");
12    }
13    return a / b;
14  }
15 }
16
17 module.exports = Calculator;
```

- **Avvio del testing con:**

- a. `npm run test`

```
1 // Installazione: npm install --save-dev mocha chai
2
3 // Il file di test con Mocha e Chai
4 const Calculator = require('./calculator');
5 const expect = require('chai').expect;
6
7 describe('Calculator', function() {
8   let calc;
9
10  beforeEach(function() {
11    calc = new Calculator();
12  });
13
14  it('should add two numbers correctly', function() {
15    const result = calc.add(2, 3);
16    expect(result).to.equal(5);
17  });
18
19  it('should divide two numbers correctly', function() {
20    const result = calc.divide(10, 2);
21    expect(result).to.equal(5);
22  });
23
24  it('should throw error when dividing by zero', function() {
25    // Verifichiamo che una funzione sollevi un'eccezione
26    expect(function() {
27      calc.divide(10, 0);
28    }).to.throw("Impossibile dividere per zero");
29  });
30 });
```

A hand-drawn decorative border in black ink, featuring various loops, swirls, and flourishes that frame the central text. The border is irregular and artistic, with some elements resembling calligraphy or doodles.

# **Best Practices e altro**



# Best Practices

## Cosa testare

- Logica di business complessa
- Edge cases e gestione degli errori
- Funzionalità critiche dell'applicazione

## Cosa non testare

- Codice di terze parti
- Getter e setter semplici
- Implementazioni triviali

## Organizzazione dei test

- Struttura dei file di test (mirror della struttura del codice)
- Naming convenzionale (test\_\*.py, \*.test.js, \*\_test.go)
- Raggruppamento logico dei test




# Test-Driven Development (TDD)

## Il ciclo Red-Green-Refactor

1. **Red:** Scrivi un test che fallisce
2. **Green:** Scrivi il minimo codice necessario per far passare il test
3. **Refactor:** Migliora il codice mantenendo i test in verde

# Esempio di TDD in Python

```
1 # 1. RED - Scriviamo un test che fallisce
2 def test_validate_password():
3     assert validate_password("abc123") == False # troppo corto
4     assert validate_password("abcdefgh") == False # nessun numero
5     assert validate_password("12345678") == False # nessuna lettera
6     assert validate_password("abcd1234") == True # valida
7
8 # Il test fallirà perché la funzione non esiste ancora
9
10 # 2. GREEN - Implementiamo la funzione per far passare il test
11 def validate_password(password):
12     if len(password) < 8:
13         return False
14     if not any(c.isdigit() for c in password):
15         return False
16     if not any(c.isalpha() for c in password):
17         return False
18     return True
19
20 # 3. REFACTOR - Miglioriamo il codice mantenendo i test in verde
21 def validate_password(password):
22     has_min_length = len(password) >= 8
23     has_digit = any(c.isdigit() for c in password)
24     has_letter = any(c.isalpha() for c in password)
25
26     return has_min_length and has_digit and has_letter
```



# Conclusioni e Risorse

## Riassunto

- I test unitari sono fondamentali per la manutenzione del codice
- Python e JavaScript offrono diversi framework con approcci simili
- Il TDD è una pratica che migliora la qualità del codice

## Passaggi successivi

- Integrare i test in CI/CD
- Esplorare altre tipologie di test (integrazione, sistema, end-to-end)
- Misurare la copertura del codice

## Risorse utili

- [Documentazione ufficiale unittest](#)
- [Documentazione Jest](#)





FINE



— WWW.DRACO.ME —