

```

sVideo = ( (type == "image") || (type == "video") || (type == "audio") ) ? "image" : "video";
sUrl = ( source.indexOf("youtube.com") > -1 ) ? "youtube.com" : "vimeo.com";
sElement = ( (type == "url") || (type == "image") || (type == "video") || (type == "audio") ) ? "url" : "image";
sObject = ( (typeof subject == "string") || (typeof subject == "number") || (typeof subject == "boolean") || (typeof subject == "object") ) ? "object" : "string";

// Check if boxer is already active, if so, return;
if ($("#boxer").length > 0) {
    return;
}

// All event
Event(e);

// Cache internal data
data = $.extend({}, {
    window: $(window),
    body: $("body"),
    target: $target,
    object: $object,
    visible: false,
    resizeTimer: null,
    touchTimer: null,
    gallery: {
        active: false
    }
});

```

# Introdução ao Unittest

Unittest é um framework de teste unitário em Python que permite testar partes específicas do código para identificar possíveis erros e garantir a qualidade do software.



**by Jéssica Oliveira**

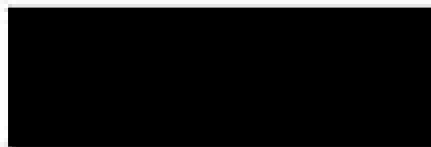
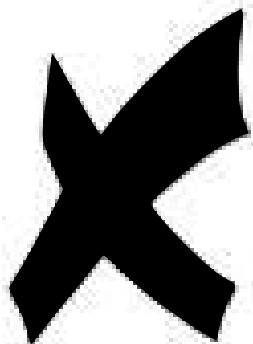
# Como escrever testes unitários

Escrever testes unitários eficazes requer conhecimento sobre as **funcionalidades** do código, criação de **casos de teste** para cada função e verificação dos **resultados esperados**.

# Estrutura de um teste unitário

Um teste unitário consiste em três etapas:

1. preparação do ambiente de teste,
2. execução do código a ser testado,
3. verificação dos resultados obtidos em relação aos resultados esperados.

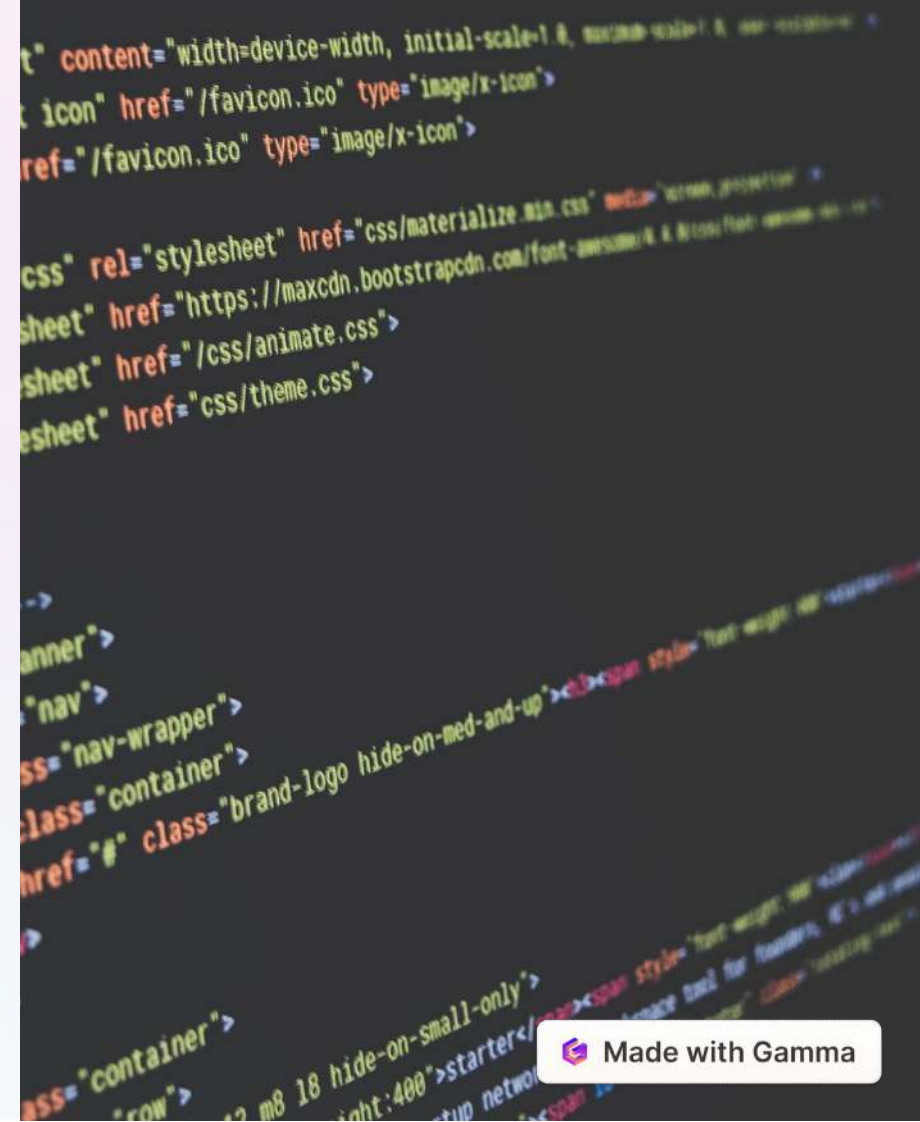


# Testes parametrizados

Os testes parametrizados permitem executar a mesma função de teste com diferentes conjuntos de dados, identificando se a função é capaz de lidar corretamente com diferentes cenários.

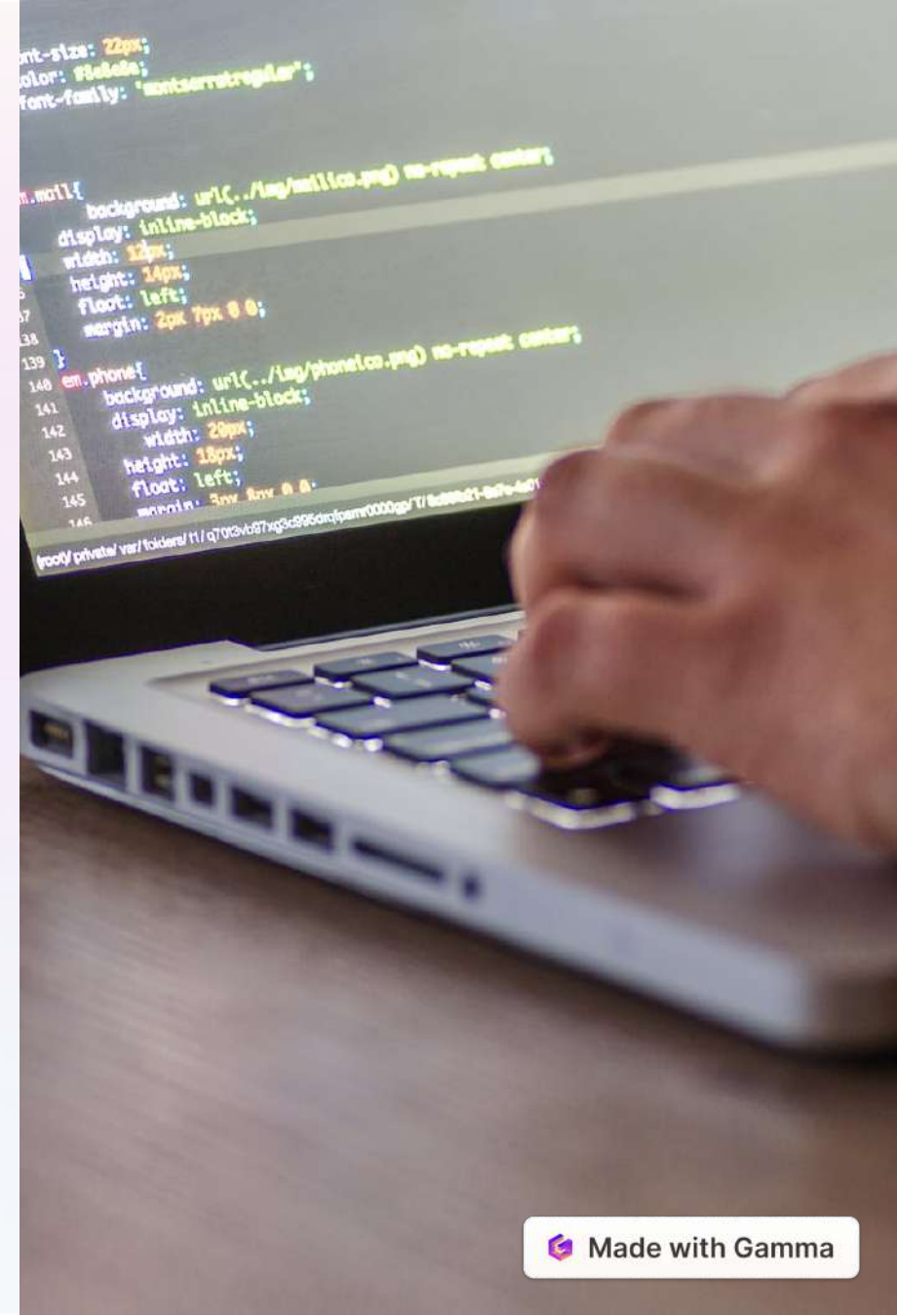
# Cobertura de código

A cobertura de código é uma medida da quantidade de código-fonte que é testada durante a execução dos testes unitários. É útil para identificar lacunas de teste e garantir uma cobertura abrangente.



# Boas práticas no uso do unittest

1. Nova funcionalidade? novos testes!
2. Manter os testes simples e independentes
3. Utilizar nomes descritivos para os testes



# passos básicos para usar a biblioteca **unittest:**

1. importe o módulo **unittest:**

```
import unittest
```



# passos básicos para usar a biblioteca **unittest**:

2. Crie uma classe de teste que herda de **unittest.TestCase**

```
class MeusTestes(unittest.TestCase):
```



# passos básicos para usar a biblioteca unittest:

## 3. Dentro da classe de teste, defina os métodos de teste.

Os métodos de teste devem começar com a palavra **"test"** e podem usar os métodos de assert para verificar se o comportamento do código está correto.

```
def test_soma(self):  
    resultado = soma(2, 3)  
    self.assertEqual(resultado, 5)
```

```
def test_subtracao(self):  
    resultado = subtracao(5, 2)  
    self.assertEqual(resultado, 3)
```

# passos básicos para usar a biblioteca **unittest**:

4. Execute os testes usando a função **unittest.main()**

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

# Exemplo Completo

```
import unittest

def soma(a, b):
    return a + b

def subtracao(a, b):
    return a - b

class MeusTestes(unittest.TestCase):

    def test_soma(self):
        resultado = soma(2, 3)
        self.assertEqual(resultado, 5)

    def test_subtracao(self):
        resultado = subtracao(5, 2)
        self.assertEqual(resultado, 3)

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

# Em resumo,

Quando você executa este arquivo, a biblioteca `unittest` descobrirá automaticamente os métodos de teste (aqueles que começam com "test") e os executará. Se todos os testes passarem, você verá uma saída indicando que todos os testes foram bem-sucedidos. Caso contrário, você receberá informações sobre os testes que falharam, o que ajuda na depuração do seu código.

# Exemplos de Assert

A biblioteca `unittest` fornece vários métodos de assert que podem ser usados para verificar diferentes condições nos seus testes. Vamos ver alguns dos tipos de assert mais comuns e exemplos de como usá-los:

# AssertEqual

`assertEqual(a, b)`: Verifica se `a` é igual a `b`.

```
import unittest

class TestAssertMethods(unittest.TestCase):

    def test_assertEqual(self):
        self.assertEqual(2 + 2, 4)
        self.assertEqual('hello', 'hello')
        self.assertEqual([1, 2, 3], [1, 2, 3])

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

# AssertNotEqual

`assertNotEqual(a, b)`: Verifica se `a` não é igual a `b`.

```
import unittest

class TestAssertMethods(unittest.TestCase):

    def test_assertNotEqual(self):
        self.assertNotEqual(3, 4)
        self.assertNotEqual('hello', 'world')
        self.assertNotEqual([1, 2, 3], [3, 2, 1])

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



# assertTrue()

`assertTrue(expr)`: Verifica se a expressão `expr` é verdadeira.

```
import unittest

class TestAssertMethods(unittest.TestCase):

    def test_assertTrue(self):
        self.assertTrue(5 > 3)
        self.assertTrue(len([1, 2, 3]) > 0)

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

# assertFalse( )

assertFalse(expr): Verifica se a expressão `expr` é falsa.

```
import unittest

class TestAssertMethods(unittest.TestCase):

    def test_assertFalse(self):
        self.assertFalse(2 > 5)

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

# AssertIsNone / AssertIsNotNone

`assertIsNone(a)`: Verifica se `a` é Nullo.

```
import unittest

class TestAssertMethods(unittest.TestCase):


    def test_assertIsNone(self):
        x = None
        y = 5

        self.assertIsNone(x)
        self.assertIsNotNone(y)

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

# como eu faço o calculo da média?

A fórmula matemática para a média (ou média aritmética) é a seguinte:

 **Média** = Soma de todos os números / Número de elementos no conjunto

**Quais cenários de Teste eu posso ter para o calculo de uma média?**