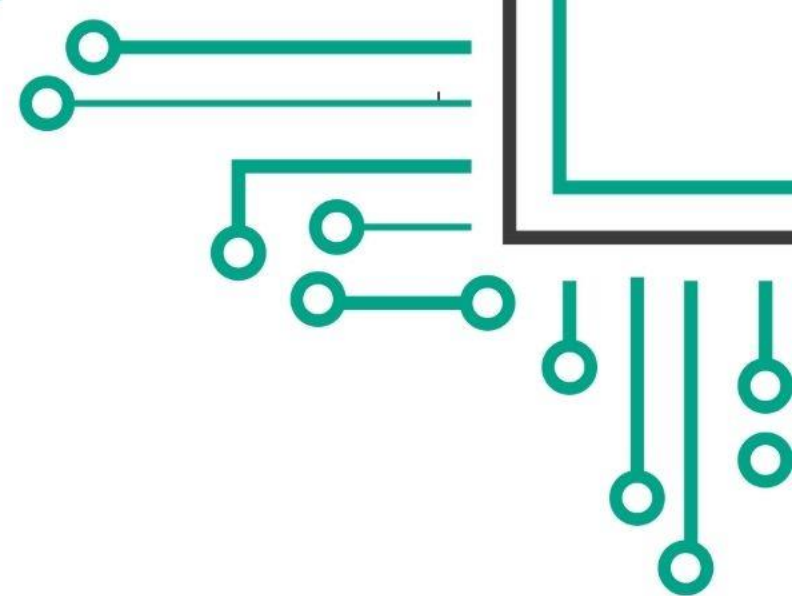
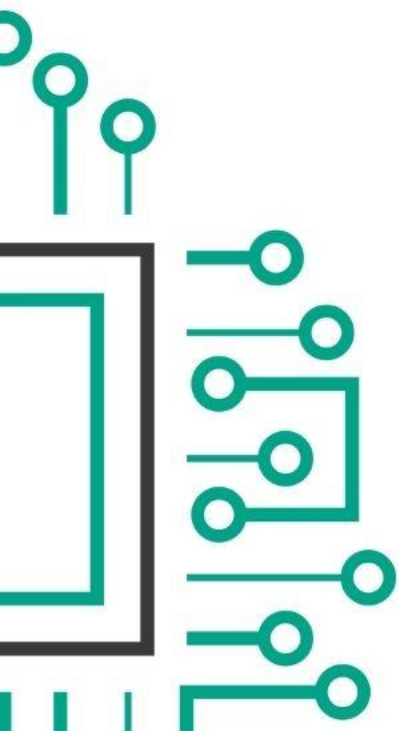


# JCAVI

## TREINAMENTOS EM TI



# TESTES COM PYTHON

Para testarmos os código utilizaremos algumas Bibliotecas de teste como o Pytest, Doctest e unittest.

Antes de utilizarmos as bibliotecas vamos entender a função assert. O Assert ou Assertions utilizamos para realizar a checagem as afirmações utilizadas no teste.

Exemplos de Asserts:

```
def testanumeros(a,b):  
    assert a > 0 and b > 0, "Numeros precisam ser maiores que zero"  
    return a + b  
  
print(testanumeros(5,0))
```

# TESTES COM PYTHON

Vamos utilizar a biblioteca de Testes PyTest. Para isso devemos instalar a biblioteca do PyTest.

Pip install pytest.

Vamos aplicar o conceito de TDD com a Biblioteca do PyTest e também em outras bibliotecas de testes.

Test Driven Development (TDD) ou em português Desenvolvimento guiado por testes é uma técnica de desenvolvimento de software que se relaciona com o conceito de verificação e validação e se baseia em um ciclo curto de repetições: Primeiramente o desenvolvedor escreve um caso de teste automatizado que define uma melhoria desejada ou uma nova funcionalidade. Então, é produzido código que possa ser validado pelo teste para posteriormente o código ser refatorado para um código sob padrões aceitáveis.

# TESTES COM PYTHON

Criando os código para testar com o Pytest.

```
def soma(x,y):  
    return x + y
```

```
def testa_soma():  
    assert 5 == soma(2,3)
```

```
divisao = 4/0  
print(divisao)
```

Para executar os testes utilizamos o terminal no diretório onde estão presentes os arquivos Python e executamos. Podemos executar no terminal da própria IDE.

```
pytest arquivo.py
```

# TESTES COM PYTHON

Testes com biblioteca Doctest.

O doctest é um módulo incluído na biblioteca padrão da linguagem de programação Python que permite a fácil geração de testes com base na saída do shell do interpretador padrão do Python, recortada e colada em docstrings.

Para utilizar o doctest devemos importar o módulo doctest e podemos ativar o modo verbose para detalhar as mensagens de saída do código.

```
import doctest
doctest.testmod(verbose=True)
```

Criando os códigos de testes em doctest e utilizando docstrings.

```
def soma(x,y):
    """
    >>> soma(2,3)
    5
    """
    return x + y
```

# TESTES COM PYTHON

```
def divide(x,z):
```

```
    """
```

```
    >>> divide(2,1)
```

```
    Traceback (most recent call last):
```

```
    ...
```

```
    ZeroDivisionError: division by zero
```

```
    """
```

```
    return x /z
```

```
def testacondicao(frase):
```

```
    """
```

```
    >>> testacondicao("Doctest Python")
```

```
    Doctest Python
```

```
    """
```

```
    print(frase)
```

Vamos validar as saídas do programa e resultados dos testes.



# TESTES COM PYTHON

## Biblioteca unittest

unittest utilizamos de forma unitária de testar o Código.

Pedaços individuais de códigos como métodos, funções, classes Que também são chamadas de TestCase.

TestCase: São casos de testes para pequenas partes de código

Para executar os testes utilizamos os assertions. No unittest temos diversos tipos de assertions.

<https://docs.python.org/3/library/unittest.html#assert-methods>

# TESTES COM PYTHON

Method	Checks that	New in
<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) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2



# TESTES COM PYTHON

Para criar os testes criamos um programa onde definimos uma classe que herda da classe `unittest.TestCase`.

Utilizamos um arquivo para declarar a classe e rodar os testes. E importamos os trechos de códigos de outros programas.

Para criarmos o arquivo de testes devemos importar o módulo `unittest` e declarar a execução dos testes.

```
import unittest
```

```
unittest.main(verbosity=2)
```

Vamos criar um outro arquivo onde vamos criar nossas funções que serão usadas para testes.

# TESTES COM PYTHON

Criando as funções para testes.

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

```
def testanumeros(a,b):  
    if a > 0 and b > 0:  
        return "OK Numeros Maiores"  
    else:  
        return "Numeros precisam ser maiores que zero"
```

```
def carrosgaragem(carros):  
    garagem = ["Fusca", "Ford KA", "Fiesta"]  
    if carros in garagem:  
        return True  
    return False
```

# TESTES COM PYTHON

Criando o arquivo de testes.

```
from prog import soma, testanumeros, carrosgaragem
```

Para criar as classes e métodos de teste temos que utilizar por convenção o test\_

```
class TesteProg(unittest.TestCase):  
    def test_soma(self):  
        self.assertEqual(soma(2,3),5)  
  
    def test_numeros(self):  
        self.assertEqual(testanumeros(2,2),"OK Numeros Maiores")  
  
    def test_carrosnagaragem(self):  
        self.assertTrue(carrosgaragem("Fusca"))
```

Vamos validar as saídas dos testes.

# EXERCÍCIO DE FIXAÇÃO

Crie um programa e algumas funções e valide os seus código utilizando o pytest.

Crie um programa e algumas funções e defina padrões de testes com o doctest.

Crie um TestCase com o unittest de algum programa criado anteriormente. Crie a classe de testes e seus assertions.