Python Fundamentals

Introdução a Teste Unitário

Anotações

Introdução a teste unitário

Objetivos da Aula

- ✔ Entender o que é teste.
- ✔ Entender a razão para testar.
- ✔ Utilizar o assert e fazer testes simples.
- ✔ Utilizar o modulo uninttest.

Introdução a teste unitário

Aprenderemos o fundamental sobre teste de software, utilizaremos assert e o módulo unnistest, para criar nossos testes.

Anotações

Testes com Python

Teste é o processo de validação da conquista dos objetivos esperados por parte do software .
Desenvolvida uma solução para determinado problema, o teste é mandatório como meio de confirmar que o software pode ser colocado em produção, com a garantia de que tudo funcionará perfeitamente.

Anotações

Por que testar?

- ✔ Garantir que as funcionalidades foram entregues.
- ✔ Assegurar que o nosso software continua funcionando a cada atualização.
- ✔ Manter proatividade na correção de falhas.
- ✔ Sustentar confiança ao alterar nossa software.

Anotações

Sintaxe assert

assert <valor> condição <valor>, <expressão>
Caso a condição seja verdadeira o teste é aprovado e nada acontece. Se a condição for falsa, o teste é reprovado e executa o Traceback.

```
>>> a = 2
>>> b = 2
>>> assert a == b, 'a é diferente de b'
>>> assert a != b, 'a é igual a b'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    assert a != b, 'a é igual a b'
AssertionError: a é igual a b
```

Sintaxe da função assert

Definimos duas variáveis ao utilizar o assert. Este, espera da sintaxe, valor, condição, valor,, semelhante a blocos de condição if. Se o resultado da condição for True, o teste foi aprovado e nada acontece, caso contrário, o teste falha, executa o parâmetro de especificação do erro junto com AssertionError.

Anotações

Testando funções e resultados esperados

```python
def somar(x, y):
    return x + y

def subtrair(x, y):
    return x - y

def dividir(x, y):
    return x / y

def multiplicar(x, y):
    return x * y

# Testar função somar
assert somar(2, 2) == 4, "Obtido:{}, Esperado:4".format(somar(2, 2))
assert somar(3, 3) == 6, "Obtido:{}, Esperado:6".format(somar(3, 3))
# Testar função subtrair
assert subtrair(2, 2) == 0, "Obtido:{}, Esperado:0 ".format(subtrair(2, 2))
assert subtrair(5, 3) == 2, "Obtido:{}, Esperado:2".format(subtrair(5, 3))
# Testar função dividir
assert dividir(4, 2) == 2, "Obtido:{}, Esperado:2 ".format(dividir(4, 2))
assert dividir(9, 3) == 3, "Obtido:{}, Esperado:3".format(dividir(9, 3))
# Testar função multiplicar
assert multiplicar(10, 2) == 20, "Obtido:{}, Esperado:20 ".format(multiplicar(10, 2))
assert multiplicar(1, 3) == 3, "Obtido:{}, Esperado:3".format(multiplicar(1, 3))
```

Testando funções e resultados esperados

Definimos algumas funções: soma, subtrai, divide e multiplica, depois estabelecemos os testes unitários, verificando se o retorno da função é igual ao valor esperado, caso falhe, será exibida uma mensagem personalizada com o valor obtido e o valor esperado.

Anotações

Uninttest

O (‘teste unitário’) consiste um módulo nativo do python. similar aos principais frameworks de teste encontrados. Suporta automação de testes, compartilhamento de código de configuração, desligamento e agregação de testes em coleções além de independência dos testes da estrutura de relatório.

Para mais informações acesse a documentação oficial: https://docs.python.org/3/library/unittest.html

Anotações

Utilizando unittest

```
1  from unittest import TestCase, main
2
3  def soma(x, y):
4      return x + y
5
6  def sub(x, y):
7      return x - y
8
9  class Testes(TestCase):
10     def test_soma(self):
11         self.assertEqual(soma(5, 5), 10) # Equivalente a ==
12         self.assertLess(soma(5,5), 11) # Equivalente a <
13     def test_sub(self):
14         self.assertEqual(sub(5,5), 0)
15         self.assertLessEqual(sub(15, 5), 10) # Equivalente a <=
16
17 if __name__ == '__main__':
18     main()
```

Utilizando o unittest

In [1]: do módulo unittest importamos TestCase e main.
In [3]: definimos duas funções para realizar os testes.
In [9]: instanciamos a classe, Testes, que herda do TestCase. Determinamos alguns métodos, como os atributos específicos que recebem dois valores, no parâmetro, para realizar a condição do teste. No próximo slide, apresentamos uma tabela com a estrutura de atributos existentes neste objeto para automação de testes.
In [17]: criamos o bloco de condição para que o teste seja executado apenas se for iniciado por ele mesmo, evitando que outros scripts o importem e executem.

Anotações

Atributos do objeto TestCase

Método	Verifica que	Novo em
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3,1
`assertIsNot(a, b)`	`a is not b`	3,1
`assertIsNone(x)`	`x is None`	3,1
`assertIsNotNone(x)`	`x is not None`	3,1
`assertIn(a, b)`	`a in b`	3,1
`assertNotIn(a, b)`	`a not in b`	3,1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3,2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3,2

Anotações

Atributos do objeto TestCase

Método	Verifica que	Novo em
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3,1
`assertIsNot(a, b)`	`a is not b`	3,1
`assertIsNone(x)`	`x is None`	3,1
`assertIsNotNone(x)`	`x is not None`	3,1
`assertIn(a, b)`	`a in b`	3,1
`assertNotIn(a, b)`	`a not in b`	3,1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3,2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3,2

Anotações

Python Fundamentals

TDD
Test Driven Development

Anotações

Objetivos da Aula

- ✔ Compreender o que é TDD.
- ✔ Conhecer suas vantagens e desvantagens.
- ✔ Aprender como utilizar esta cultura.

Desenvolvimento orientado a teste

Nesta aula aprensentaremos o conceito sobre TDD, suas vantagens e desvantagens, explicaremos como implementá-lo.

Anotações

Test Driven Development

TDD

É o desenvolvimento de software orientado a testes, em inglês, Test Driven Development. Mas mais do que simplesmente testar seu código, TDD é uma filosofia, uma cultura.
Desenvolvimento orientado a testes, criamos inicialmente os testes antes de implementar a lógica do software.

Anotações

Por que muitos não praticam?

- ✔ Dificuldade em começar.
- ✔ Curva de aprendizado.
- ✔ Tempo de desenvolvimento.
- ✔ Culturas adotadas.

Por que muitos não praticam?

Dificuldade em começar — Apesar de uma extensa e clara documentação, iniciar o desenvolvimento orientado a testes pode ser um trabalho árduo para muitos. Devido o simples fato de que geralmente, muitos iniciantes tentam praticá-lo em código já existente. Este, definitivamente não é o caminho. A principal característica do desenvolvimento orientado a testes, reside na orientação a testes. Em outras palavras, o código que realizará sua lógica, deve ser criado somente após a criação do teste. Torna-se algo de difícil aceitação, pois ainda não se produziu nada e já se planeja testar.

Curva de aprendizado — Complementando o item anterior, mais um motivo que faz programadores desistirem do desenvolvimento orientado a testes. Como qualquer nova tecnologia, para a prática de TDD, leva-se tempo em dependência, disponibilidade e principalmente, da vontade do programador.

Tempo — Engana-se quem pensa que produzirá mais código pelo simples fato de utilizar TDD. A cultura na verdade chega a desacelerar a produção de código. Nos referimos em produção de código, à quantidade de linhas escritas. Mesmo neste aspecto, há vantagens, que serão descritas mais à frente.

Cultura — Muito se fala em TDD no Brasil. Porém, ao questionarmos programadores de diversas empresas, muitos apresentam os motivos já citados, para não utilizá-lo. Há muitas empresas e programadores que levam a prática a sério, a evangelizam justamente por conhecerem as vantagens oferecidas pelo TDD.

Vantagens

- ✔ Qualidade do código.
- ✔ Raciocínio.
- ✔ Segurança.
- ✔ Trabalho em equipe.
- ✔ Documentação.

Anotações

Vantagens

Qualidade do código — Um dos principais ensinamentos, senão o principal, do TDD: se algo não é suscetível a teste, foi desenvolvido de forma errada. Parece um pouco drástico, mas, não é. Em pouco tempo utilizando testes, o programador percebe mudanças relevantes em sua forma de programar. Em suma, o uso de TDD ajuda o programador a elaborar um código com melhor qualidade, criando objetos concisos e com menos dependências.

Raciocínio — Para tornar o código mais conciso, tenha menos acoplamentos e dependências. O programador deve forçar seu raciocínio a níveis elevados. É trabalhoso criar algo que realmente tenha um bom design. Utilizando TDD o programador praticamente obriga-se a olhar seu código de outra forma, normalmente jamais vista antes. Aí está a parte legal da coisa toda.

Segurança — Importantíssimo, para qualquer software nos dias de hoje. Mas, não se engane, não se trata de segurança da informação, todavia, de segurança ao desenvolver. Pense em uma situação na qual o programador trabalha um código em construção há cerca de um ano. Como normalmente vivemos em um mundo com inúmeros softwares desenvolvidos ao longo de cada ano, torna-se muito difícil lembrar de tudo a respeito de um caso específico que merece atenção em determinado momento. Normalmente deve-se realizar um trabalho bastante cauteloso para nova implementação em um software já em produção. Toda e qualquer alteração deve ser minunciosamente testada, garantindo que não afetará os demais módulos do software. Esta implementação manual, é realmente complexa, pois até então, não se sabe, ou recorda-se ao certo quem afeta o que no sistema. Com a prática de TDD, cada pequeno passo do software, está devidamente testado. Ou seja, neste cenário o programador pode realizar qualquer alteração sem medo e sem culpa.
Como cada pequeno passo tomado pelo sistema está testado, caso qualquer módulo ou funcionalidade sofra alteração, em poucos segundos descobre-se se houveram quebras, ainda melhor, onde foram essas quebras. Assim, a correção das quebras torna-se uma tarefa simples sem frustrar o cliente e o usuário.

Trabalho em equipe — Ao prover mais segurança, o trabalho em equipe torna-se muito mais proveitoso, eliminando discussões e dúvidas desnecessárias. Ao entrar no desenvolvimento de um projeto, o novo desenvolvedor terá apenas o trabalho de compreender qual task deve ser realizada e, ler os testes das features já desenvolvidas. Ao rodar os testes pela primeira vez, descobrirá que está no caminho para dispor entregáveis mais rapidamente e com segurança. Existem empresas em que um novo programador tem entregáveis logo no primeiro dia de trabalho. Sem testes, normalmente haveria um período de adaptação, para prévio entendimento do que há no sistema no momento de seu ingresso ao time de desenvolvimento.

Documentação — Ao criar testes descritivos, estes servem como uma excelente documentação para o software. Quando qualquer programador rodar os testes, basta habilitar o modo verbose, uma “história” será contada, eliminando o árduo trabalho de documentar um software, que em meios tradicionais tende a defasagem. A documentação tradicional raramente segue o mesmo ritmo do desenvolvimento. Com os testes unitários, a “documentação” é gerada antes mesmo da nova feature ser implementada e, permanece fiel a qualquer alteração.

Criando teste

```python
from unittest import TestCase, main

def validar_par(num: int)-> bool:
    '''
    Função para validar um número par.
    Args:
        num - recebe um número do tipo inteiro
    Retorno: Booleano
    '''
    # Aqui será implementado a logica
    pass

class Testes(TestCase):
    def test_par(self):
        self.assertEqual(validar_par(4), True)
        self.assertEqual(validar_par(1000), True)
    def test_impar(self):
        self.assertEqual(validar_par(5), False)
        self.assertEqual(validar_par(1001),False)
    def test_string(self):
        self.assertEqual(validar_par('102'), True)
        self.assertEqual(validar_par('1059'), False)
        self.assertEqual(validar_par('string'), None)

if __name__ == "__main__":
    main()
```

Criando teste

In [1]: do módulo unittest importamos TestCase e main, para criar nosso teste.
In [3]: definimos uma função com o seguinte problema, validar um número par, com a seguinte descrição, receber um número inteiro e retornar um booleano.
In [13]: determinamos a nossa classe Testes, herdando de TestCase, com os seguintes métodos , testar par: caso o número recebido seja par, retornar verdadeiro, testar ímpar: caso o número seja impar, retornar falso, testar string: caso a string seja numérica, realizar a conversão para inteiro e retornar verdadeiro para par e, falso para ímpar , caso contrário retornar None.

Anotações

Solucionando o teste

```
3   def validar_par(num: int)-> bool:
4       '''
5       Função para validar um número par.
6       Args:
7           num - recebe um número do tipo inteiro
8       Retorno: Booleano
9       '''
10      if isinstance(num, int):
11          return True if num % 2 == 0 else False
12      elif isinstance(num, str):
13          if num.isnumeric():
14              return True if int(num) % 2 == 0 else False
15      else:
16          return None
```

Solucionando o teste

In [10]: criamos um bloco de condição para verificar se a instância do parâmetro é inteira, caso seja, retornar verdadeiro quando o módulo de divisão por 2 for igual a 0. Falso para qualquer outro valor. Esta condição verifica se um número é par.
In [12]: criamos um bloco de condição encadeada, verificando se a instância do parâmetro é string caso seja verdadeira. Criando outro bloco de condição, confirmando se é numérico, caso verdadeiro realizar a conversão pra inteiro e fazer a verificação se o número é par.
In [15]: se o parâmetro não passar nas condições acima, a função retorna 'None'

Anotações

Python Fundamentals

BDD
Behavior Driven Development

Anotações

Behavior Driven Development

Objetivos da Aula

- ✔ Aprender o que é BDD.
- ✔ Conhecer o framework behave.
- ✔ Conhecer sua estrutura de pasta.
- ✔ Criar steps e funcionalidade.

Behavior Driven Development

Nesta aula você aprenderá sobre BDD, conhecerá o framework Behave, compreendendo sua estrutura de pastas, criará steps e funcionalidades, acompanhando sua saída de teste.

Anotações

Desenvolvimento guiado por comportamento

Behavior Driven Development (BDD), em tradução Desenvolvimento Guiado por Comportamento, é uma técnica de desenvolvimento Ágil. Encoraja a colaboração entre desenvolvedores, setores de qualidade e pessoas não técnicas ou de negócios, num projeto de software.
Relaciona-se ao conceito de verificação e validação.
Originalmente concebida em 2003, por Dan North como uma resposta à Test Driven Development (Desenvolvimento Guiado por Testes), vem se expandido bastante nos últimos anos.

Diferenças BDD e TDD

BDD — Trabalha para definir como uma demanda chega ao desenvolvedor, integra diferentes áreas da empresa, pensa a partir do ponto de vista de comportamento esperado, pelo usuário, de uma funcionalidade. Por consequência, influencia como os testes são planejados e escritos.

TDD — Busca garantir a qualidade do código, sempre pensando em 100% de cobertura de testes, melhora o que acabou de ser realizado, nunca escreve uma linha de código sem antes pensar em como garantir que irá funcionar.

Podemos concluir que BDD não se opõe ao TDD, ambos os métodos podem ser aplicados em conjunto ou, apenas um deles. Um e outro, buscam melhorar o desenvolvimento de software, constituindo parte de cultura de excelência.

Anotações

Behave framework

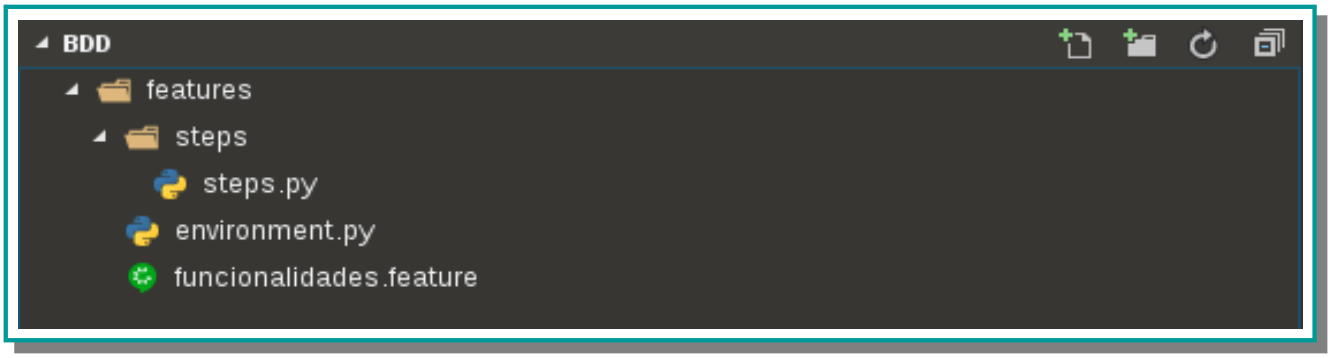
Trata-se de uma biblioteca Python que permite a escrita de testes de comportamento em linguagem humana e, a execução dos cenários através de asserts em "linguagem de programação".

1 Instale o behave

```
# pip3 install behave
```

Anotações

Estrutura de pasta



Estrutura de Pasta

Features # Pasta de estrutura do framework.
- steps # Pasta para arquivos de definição da feature.
- step.py # arquivo python de definição para feature.
- environment.py # arquivo python para setup global.
- name.feature # feature file.

Anotações

Feature

Arquivo funcionalidade.feature

```
1  # language: pt
2
3  Funcionalidade: Soma
4      Cenario: adicao basica
5          Quando somar "2" com "2"
6          Então o resultado deve ser "4"
```

Feature

In [1]: definimos a linguagem.
In [3]: determinamos o bloco funcionalidade e atribuímos nome a ele.
In [4]: estabelecemos o cenário com descrição do evento.
In [5] e [6]: definimos o bloco de condição.
Obs. : toda a feature escrita em português.

Anotações

Arquivo steps.py

```python
from behave import step

def soma(x:int, y:int) -> int:
    return x + y

@step('somar "{num1}" com "{num2}"')
def test_soma(context, num1, num2):
    context.r_soma = soma(int(num1), int(num2))

@step('o resultado deve ser "{esperado}"')
def test_soma_result(context, esperado):
    assert context.r_soma == int(esperado), "descrição do erro"
```

Behave

In [1]: do módulo behave importamos step.
In [3]: definimos a função somar, retorno a soma de dois números inteiros.
In [6]: determinamos o decorator step de 2 números, recebe a função test_soma que recebe 3 parâmetros, context é o atributo da classe de objeto do framework e num1 e num2, números inteiros.
In [10]: estabelecemos o decorator step, recebe a função de resultado que obtem dois parâmetros, contexto, este, recebeu os valores do teste acima, o esperado é um valor a ser comparado a context para ter a estrutura de condição, se o teste passou ou não.

Anotações

Executando

1 Execute a feature

```
# behave funcionalidades.feature
```

```
jonsnow@stark:~$ cd BDD/
jonsnow@stark:~/BDD$ cd features/
jonsnow@stark:~/BDD/features$ behave funcionalidades.feature
Funcionalidade: Soma # funcionalidades.feature:3

  Cenario: adicao basica            # funcionalidades.feature:4
    Quando somar "2" com "2"        # steps/steps.py:6 0.000s
    Então o resultado deve ser "4" # steps/steps.py:10 0.000s

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
2 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.000s
```

Executando

Ao executar o arquivo da feature, teremos um resultado semelhante, colorindo de vermelho os steps reprovados na condição, os aprovados serão coloridos em verde. Também especifica quantos testes foram realizados, quantos aprovados e o tempo consumido para fazer as verificações.

Anotações