**Testes automatizados: TDD com Python**

1. **Aula 1 – Por que testar?**
   1. ALT + INSERT: Atalho para criar um novo arquivo no pycharm.
   2. Quando vamos importar uma classe presente em um arquivo separado do python, podemos fazer de duas maneiras, a primeira é somente selecionar o que o python sugere em primeiro e a segunda é seguir o caminho completo das pastas desde o root até o arquivo .py:



* + 1. Como podemos ver, as subpastas são separadas por “.” E não por “/” ou “\” como estamos acostumados.
  1. CTRL + P: Atalho para saber o que colocar como parâmetro em uma função caso não lembramos o que precisa ser passado. Semelhante ao CTRL + SPACE no VScode e no próprio Pycharm, mas para funções, atributos e afins:



* 1. @property: Já vimos antes, responsável por retornar uma propriedade privada como uma normal para visualização da mesma. Não permite alterar. A visualização é de uma segunda variável que possuí o valor da privada, permitindo que o usuário a acesse como se fosse o atributo:



* 1. Quando vamos implementar um sistema de avaliador para saber qual foi o menor e qual foi o maior lance dado em um leilão, por exemplo, podemos utilizar uma biblioteca chamada sys.
     1. Como precisamos criar duas variáveis, uma que receberá o maior e outra o menor valor, o valor de início delas precisa ser, respectivamente, o menor e o maior valor existente, pois, dessa maneira, qualquer valor que for muito alto sobrescreverá o menor valor da variável maior valor, e o contrário também é verdade para o menor valor.
     2. Dessa forma, utilizamos a biblioteca sys para que ela nos forneça esses dados a partir do sistema, sem que nós mesmos precisemos determinar isso:



* + 1. O que estamos dizendo é que queremos pegar o menor valor das informações de números flutuantes, ou seja, com “.”/”,”, existentes na biblioteca sys. O contrário é válido para o maior valor.
  1. Quando estamos criando um método, podemos passar como argumento o nome dele seguido de “:” e uma classe, ou então uma frase, indicando o que se espera receber como argumento. Chamamos esse tipo de estratégia de anotação:





* + 1. Basicamente estamos dizendo que como argumento para esse método esperamos que seja passado o objeto Leilao.
    2. Para fazer com que o maior e menor lance sejam atribuídas as suas respectivas variáveis, precisamos fazer um esquema de validação com *ifs*:



* + 1. Basicamente dizendo que para cada lance nos lances do leilão, deverá ser feita essa verificação: Se o valor do lance for maior que o maior lance (inicialmente determinado pelo sys, como visto acima), então o maior lance recebe esse novo valor do lance. O contrário também é verdade.
  1. ALT + ENTER: Caso esteja tentando utilizar uma classe que ainda não foi importada, ao invés de subir todo o código e digitar o import dela, pode simplesmente utilizar esse atalho e selecionar a primeira opção:



* 1. Testamos nosso código e descobrimos que existe um bug nele ao adicionar outro lance com um valor maior do que o que já havia sido definido antes. Com esse exemplo determinamos o quão importante é testar nosso código, pois se esse programa fosse de um hospital, pacientes poderiam ter recebido as doses erradas de um medicamento.
     1. Considerando isso, muitos desenvolvedores criaram frameworks que possibilitam fazer testes em programas.
     2. Caso queiramos fazer o teste em uma classe específica, podemos deixar o cursor em cima do nome da classe e apertar CTRL + SHIFT + T e ENTER:





* + 1. O arquivo que ele vai criar é este:



* + 1. Onde ele cria uma classe teste e herda de *TesCase*.
    2. O que vamos fazer é apagar esse *sel.fail* (que serve justamente para dar erro no nosso teste) e colocar o que já tínhamos no nosso principal:



* + 1. Para confirmar nosso teste podemos utilizar um método do *TestCase* chamado *assertEqual*(valor\_esperado, valor\_programa), dessa forma ele faz a comparação se o valor esperado é igual ou não ao valor recebido, reportando no console depois:







* + 1. Para corrigir esse bug é simples, basta ir lá no nosso domínio e remover duas letras da classe Avaliador:





* + 1. Se deixássemos um *elif* no método avalia ele nunca passaria pela segunda verificação se passasse logo na primeira, fazendo com que ocorresse o bug, mas ao deixar apenas como if, o código é obrigado a passar por ambas as verificações evitando o bug de ocorrer:



* + 1. Podemos também rodar os testes pelo terminal, para isso, podemos pedir para o Python rodar o arquivo test\_avaliador.py utilizando o módulo de testes:



* 1. O que aprendemos:
     1. O porquê testar;
     2. Como começar a testar com Python;
     3. Conhecemos a biblioteca *unittest*;
     4. Como escrever e rodar um teste com o Pycharm.

1. **Aula 2 – Boas Práticas e Mais Alguns Testes:**
   1. Quando estamos rodando testes, não podemos criar métodos de testes com nomes parecidos como test\_avalia1,2,3...., pois dessa forma fica muito confuso e de difícil leitura para alguém novo na equipe que não conhece o código e pegue para ver os testes.
      1. Portanto, é uma boa prática fazermos testes com nomes coerentes e condizentes com o que está sendo testado, assim:

def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_crescente(self):

Def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_decrescente(self):

* + 1. Dessa forma deixa muito mais claro o que cada teste deve fazer, deixando o entendimento mais fácil e rápido.
    2. Existem vários tipos de nomenclaturas que podem ser usadas e ela será determinada em cada projeto. O importante é sempre usar a mesma nomenclatura sempre para evitar deixar os testes confusos.
  1. Após criar uma quantidade de testes, como para leilões, não precisamos testar até 1000 lances para ver se o código continua funcionando.
     1. Isso porque existe algo chamado teste de equivalência, onde temos a confiança de que nosso código funcionará independente do número de lances e se eles foram adicionados em ordem crescente ou decrescente, uma vez que já fizemos esses testes com um número menor de lances.
     2. Portanto, se já fizemos testes com 1, 2, 3 lances, em ordem crescente e decrescente, já podemos confiar que funcionará para inúmeros lances devido ao teste de equivalência. A menos que exista uma condição diferente em algum determinado lance ou que faça sentido ser testado, aí, nesse caso, seria interessante fazer os testes necessários.
     3. Considerando isso é importante sempre levarmos em conta essas classes de equivalência quando formos escrever os testes, para que não percamos tempo escrevendo testes que irão apenas testar mais do mesmo.
  2. Sempre que temos muito copia e cola é interessante isolarmos esse código e somente chamar a classe usada depois, dessa forma se precisarmos alterar algo, basta alterar em um único local, sem ter que fazer isso em diversos pontos e evitando quebra de código.
     1. Geralmente utilizamos para criar cenários, uma vez que para rodar um teste um cenário precisa ser criado logo no seu início toda vez.
     2. Contudo, se algo é muito comum de se fazer, podemos supor que já exista um framework que faça isso para nós e facilite nossa vida.
     3. Então, ao invés de criar os usuários e lances toda vez que criarmos um teste, basta criar um *def criaCenário* e colocar isso dentro dele, apenas invocando a classe nas subsequentes que a utilizam, assim:

def criaCenario(self):  
 self.gui = Usuario('Gui')  
 self.yuri = Usuario('Yuri')  
 self.lance\_do\_yuri = Lance(self.yuri, 100.0)  
 self.lance\_do\_gui = Lance(self.gui, 150.0)  
 self.leilao = Leilao('Celular')

def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_crescente(self):  
 self.criaCenario()  
 self.leilao.lances.append(self.lance\_do\_yuri)  
 self.leilao.lances.append(self.lance\_do\_gui)  
  
 avaliador = Avaliador()  
 avaliador.avalia(self.leilao)  
  
 menor\_valor\_esperado = 100.0  
 maior\_valor\_esperado = 150.0  
  
 self.assertEqual(menor\_valor\_esperado, avaliador.menor\_lance)  
 self.assertEqual(maior\_valor\_esperado, avaliador.maior\_lance)

* + 1. Sempre colocando o self antes de tudo para indicar que é interno e fazer funcionar.
    2. Contudo, ao invés de criar esse *criaCenario*, podemos utilizar o framework da própria biblioteca que já importamos:

from unittest import TestCase

* + 1. Método esse chamado de *setUp*(self). O próprio python já chama esse método em todos os métodos de teste para podermos utilizar, sem temor que fazer manualmente como no *criaCenario*, portanto, ele vai automaticamente criar o cenário e invocar esse método no início do teste:

def setUp(self):  
 self.gui = Usuario('Gui')  
 self.yuri = Usuario('Yuri')  
 self.lance\_do\_yuri = Lance(self.yuri, 100.0)  
 self.lance\_do\_gui = Lance(self.gui, 150.0)  
 self.leilao = Leilao('Celular')  
  
  
def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_crescente(self):  
 self.leilao.lances.append(self.lance\_do\_yuri)  
 self.leilao.lances.append(self.lance\_do\_gui)  
  
 avaliador = Avaliador()  
 avaliador.avalia(self.leilao)  
  
 menor\_valor\_esperado = 100.0  
 maior\_valor\_esperado = 150.0  
  
 self.assertEqual(menor\_valor\_esperado, avaliador.menor\_lance)  
 self.assertEqual(maior\_valor\_esperado, avaliador.maior\_lance)

* + 1. Porém temos um dilema aqui, o usuário Yuri não é utilizado em todos os testes, mas, independente disso, ele é criado toda vez que rodamos o teste, o que pode prejudicar a performance do nosso programa, uma vez que sempre que vamos rodar um teste nosso código tem que criar o usuário Yuri e o lance do Yuri, ou seja, 2 objetos que algumas vezes não é utilizado.
    2. Diferentemente do usuário gui que aparece em todos os testes, portanto, faz sentido sua criação no isolamento.
    3. Considerando isso o ideal seria tirar ele e deixar no setup tudo e somente aquilo que for necessário para que todos os testes sejam executados. Essa seria a boa prática.
    4. Como o usuário Yuri não é um objeto grande a ser criado e somente um dos testes não o utiliza, até podemos deixar, mas caso surgissem outros testes que ele não faria parte, o ideal seria retirar ele do setup.
    5. Apesar de continuar sendo um copia e cola, deixa nosso teste mais performático.
  1. Outros métodos da TestCase:
     1. Vimos na aula o método *setUp* que é executado antes de cada método de testes. Este não é o único método que herdamos da classe *TestCase*, além deste, temos outros métodos como o *tearDown*, *setUpClass* e o *tearDownClass*.
     2. Vamos imaginar que estamos realizando um teste que faz a conexão com um serviço externo, envia um e-mail ou então faz uma modificação no banco de dados.
     3. Nesses casos, precisamos abrir uma conexão, executar os testes e fechar a conexão em seguida. Fechar a conexão é algo em comum com esses tipos de testes.
     4. Podemos então declarar o método *tearDown* que é executado logo após a execução do teste. Ou seja, caso abrimos uma conexão, podemos utilizar o método *tearDown* para fechá-la após os testes.
     5. Analogamente ao método *setUp* e *tearDown*, temos o *setUpClass* e o *tearDownClass*. Ambos funcionam de uma forma parecida com o *setUp* e *tearDown*, porém, ao invés de serem executados antes de cada método, são executados apenas uma vez - o *setUpClass* no momento que a classe é criada e o *tearDownClass* após o último teste da classe ser rodado.
     6. Os métodos *tearDown* e *tearDownClass* são muito utilizados em testes que integram duas partes do sistema - banco de dados, sistemas externos, ou então desejamos fechar uma conexão que foi aberta.
     7. Esses tipos de testes, que verificam como duas partes do sistema se integram, são chamados de testes de integração.
  2. Link de um artigo que mostra outras formas de nomenclaturas para os testes: <https://dzone.com/articles/7-popular-unit-test-naming>.
  3. O que aprendemos:
     1. Formas de nomear os testes;
     2. Porque é importante dar nomes semânticos aos testes;
     3. Como isolar o cenário com o método *setUp*.

1. **Aula 3 – Remodelando as Classes de Domínio:**
   1. *Tell, don’t ask*: Propõem algo para o python e ele faz não interessa como.
      1. Se trata de criar uma classe que faça algo que evite quebrar o encapsulamento. Estávamos pedindo para o python fazer um acréscimo/*append* nos lances do leilão, acessando o atributo privado diretamente e alterando ele, mesmo que através de uma *@property*:

self.leilao.lances.append(lance\_do\_yuri)  
self.leilao.lances.append(self.lance\_do\_gui)

* + 1. Então criamos um método, chamado propõem, que faz isso e agora simplesmente falamos para o python propor nossos lances ao invés de pedir como estava antes, o que deixa o código muito mais amarrado:

def propoem(self, lance: Lance):  
 self.\_\_lances.append(lance)

self.leilao.propoem(lance\_do\_yuri)  
self.leilao.propoem(self.lance\_do\_gui)

* + 1. Quando criamos um método novo, não queremos que os outros que estavam utilizando o atributo diretamente, continuem fazendo isso, mas, ao rodar o teste sem alterar os outros métodos colocando nosso novo jeito de propor lances, eles ainda irão rodar. Isso acontece porque temos uma @property que nos dá acesso direto à referência da lista original, permitindo sua alteração:

@property  
def lances(self):  
 return self.\_\_lances

* + 1. Para evitar que isso aconteça precisamos criar uma cópia dessa lista, e para isso basta criar um índice com “:” dentro:

@property  
def lances(self):  
 return self.\_\_lances[:]

* + 1. Existem várias formas de criar cópias de uma lista e essa é uma delas, chamada cópia rasa de lista.
    2. Ao testar o código agr, vemos que apenas um passou e os outros 3 falharam, uma vez que alteramos o acesso direto do atributo \_\_lances para uma cópia da lista, evitando que os testes que estavam fazendo a implementação diretamente falhassem e aquele que fizemos a alteração para o método propõem continuou funcionando.
    3. Fazendo esse tipo de alteração estamos aumentando o encapsulamento do nosso código. Chamamos isso de programação defensiva, é uma técnica utilizada para evitar efeitos colaterais no nosso código.
  1. Com relação a criar objetos diferentes para saber qual é o maior e menor lance ou deixar com que o próprio leilão faça isso, vai depender muito do que queremos, do tamanho do código e das responsabilidades que o objeto leilão já tem, dentre outras coisas.
     1. Conforme nosso código for aumentando, é válido sim ir fazendo alterações e ir o quebrando em vários métodos e objetos. Lembre-se que programação não é cravada em pedra e podemos fazer alterações para deixar nosso código mais responsivo. Com melhor leitura e assim por diante.
     2. No nosso caso, como o leilão não possuía muitos métodos, passamos essa responsabilidade para ele por fazer mais sentido, então ao invés de ter as classes leilão e avaliador:

class Leilao:  
 def \_\_init\_\_(self, descricao):  
 self.descricao = descricao  
 self.\_\_lances = []  
  
 def propoem(self, lance: Lance):  
 self.\_\_lances.append(lance)  
  
 @property  
 def lances(self):  
 return self.\_\_lances[:]  
  
  
class Avaliador:  
 def \_\_init\_\_(self):  
 self.maior\_lance = sys.float\_info.min  
 self.menor\_lance = sys.float\_info.max  
  
 def avalia(self, leilao: Leilao):  
 for lance in leilao.lances:  
 if lance.valor > self.maior\_lance:  
 self.maior\_lance = lance.valor  
 if lance.valor < self.menor\_lance:  
 self.menor\_lance = lance.valor

* + 1. Ficamos somente com a leilão:

class Leilao:  
 def \_\_init\_\_(self, descricao):  
 self.descricao = descricao  
 self.\_\_lances = []  
 self.maior\_lance = sys.float\_info.min  
 self.menor\_lance = sys.float\_info.max  
  
 def propoem(self, lance: Lance):  
 if lance.valor > self.maior\_lance:  
 self.maior\_lance = lance.valor  
 if lance.valor < self.menor\_lance:  
 self.menor\_lance = lance.valor  
 self.\_\_lances.append(lance)  
  
 @property  
 def lances(self):  
 return self.\_\_lances[:]

* + 1. E o nosso teste ficou de assim:

from unittest import TestCase  
from src.leilao.dominio import Usuario, Lance, Leilao, Avaliador  
  
  
class TestAvaliador(TestCase):  
 def setUp(self):  
 self.gui = Usuario('Gui')  
 self.lance\_do\_gui = Lance(self.gui, 150.0)  
 self.leilao = Leilao('Celular')  
  
  
 def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_crescente(self):  
 yuri = Usuario('Yuri')  
 lance\_do\_yuri = Lance(yuri, 100.0)  
  
 self.leilao.propoem(lance\_do\_yuri)  
 self.leilao.propoem(self.lance\_do\_gui)  
  
 avaliador = Avaliador()  
 avaliador.avalia(self.leilao)  
  
 menor\_valor\_esperado = 100.0  
 maior\_valor\_esperado = 150.0  
  
 self.assertEqual(menor\_valor\_esperado, avaliador.menor\_lance)  
 self.assertEqual(maior\_valor\_esperado, avaliador.maior\_lance)  
  
 def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_decrescente(self):  
 yuri = Usuario('Yuri')  
 lance\_do\_yuri = Lance(yuri, 100.0)  
  
 self.leilao.propoem(self.lance\_do\_gui)  
 self.leilao.propoem(lance\_do\_yuri)  
  
 avaliador = Avaliador()  
 avaliador.avalia(self.leilao)  
  
 menor\_valor\_esperado = 100.0  
 maior\_valor\_esperado = 150.0  
  
 self.assertEqual(menor\_valor\_esperado, avaliador.menor\_lance)  
 self.assertEqual(maior\_valor\_esperado, avaliador.maior\_lance)  
  
 def test\_deve\_retornar\_o\_mesmo\_valor\_para\_o\_maior\_e\_menor\_lance\_quando\_o\_leilao\_tiver\_um\_lance(self):  
 self.leilao.propoem(self.lance\_do\_gui)  
  
 avaliador = Avaliador()  
 avaliador.avalia(self.leilao)  
  
 self.assertEqual(150.0, avaliador.menor\_lance)  
 self.assertEqual(150.0, avaliador.maior\_lance)  
  
 def test\_deve\_retornar\_o\_maior\_e\_menor\_valor\_quando\_o\_leilao\_tiver\_tres\_lances(self):  
 yuri = Usuario('Yuri')  
 bru = Usuario('Bru')  
  
 lance\_do\_yuri = Lance(yuri, 100.0)  
 lance\_do\_bru = Lance(bru, 200.0)  
  
 self.leilao.propoem(self.lance\_do\_gui)  
 self.leilao.propoem(lance\_do\_yuri)  
 self.leilao.propoem(lance\_do\_bru)  
  
 avaliador = Avaliador()  
 avaliador.avalia(self.leilao)  
  
 menor\_valor\_esperado = 100.0  
 maior\_valor\_esperado = 200.0  
  
 self.assertEqual(menor\_valor\_esperado, avaliador.menor\_lance)  
 self.assertEqual(maior\_valor\_esperado, avaliador.maior\_lance)

* + 1. Para assim:

from unittest import TestCase  
from src.leilao.dominio import Usuario, Lance, Leilao  
  
  
class TestAvaliador(TestCase):  
 def setUp(self):  
 self.gui = Usuario('Gui')  
 self.lance\_do\_gui = Lance(self.gui, 150.0)  
 self.leilao = Leilao('Celular')  
  
  
 def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_crescente(self):  
 yuri = Usuario('Yuri')  
 lance\_do\_yuri = Lance(yuri, 100.0)  
  
 self.leilao.propoem(lance\_do\_yuri)  
 self.leilao.propoem(self.lance\_do\_gui)  
  
 menor\_valor\_esperado = 100.0  
 maior\_valor\_esperado = 150.0  
  
 self.assertEqual(menor\_valor\_esperado, self.leilao.menor\_lance)  
 self.assertEqual(maior\_valor\_esperado, self.leilao.maior\_lance)  
   
  
 def test\_deve\_retornar\_o\_maior\_e\_o\_menor\_valor\_de\_um\_lance\_quando\_adicionados\_em\_ordem\_decrescente(self):  
 yuri = Usuario('Yuri')  
 lance\_do\_yuri = Lance(yuri, 100.0)  
  
 self.leilao.propoem(self.lance\_do\_gui)  
 self.leilao.propoem(lance\_do\_yuri)  
  
 menor\_valor\_esperado = 100.0  
 maior\_valor\_esperado = 150.0  
  
 self.assertEqual(menor\_valor\_esperado, self.leilao.menor\_lance)  
 self.assertEqual(maior\_valor\_esperado, self.leilao.maior\_lance)  
  
 def test\_deve\_retornar\_o\_mesmo\_valor\_para\_o\_maior\_e\_menor\_lance\_quando\_o\_leilao\_tiver\_um\_lance(self):  
 self.leilao.propoem(self.lance\_do\_gui)  
  
 self.assertEqual(150.0, self.leilao.menor\_lance)  
 self.assertEqual(150.0, self.leilao.maior\_lance)  
  
 def test\_deve\_retornar\_o\_maior\_e\_menor\_valor\_quando\_o\_leilao\_tiver\_tres\_lances(self):  
 yuri = Usuario('Yuri')  
 bru = Usuario('Bru')  
  
 lance\_do\_yuri = Lance(yuri, 100.0)  
 lance\_do\_bru = Lance(bru, 200.0)  
  
 self.leilao.propoem(self.lance\_do\_gui)  
 self.leilao.propoem(lance\_do\_yuri)  
 self.leilao.propoem(lance\_do\_bru)  
  
 menor\_valor\_esperado = 100.0  
 maior\_valor\_esperado = 200.0  
  
 self.assertEqual(menor\_valor\_esperado, self.leilao.menor\_lance)  
 self.assertEqual(maior\_valor\_esperado, self.leilao.maior\_lance)

* + 1. Onde retiramos a importação de Avaliadores, deletamos todos os avaliadores de cada teste, e substituímos cada avaliador por *self.leilão*, já que colocamos a classe avaliador dentro de leilão.
  1. Lei de Demeter, ou o Princípio do menor conhecimento:
     1. Nesta classe, existe uma lista de lances. Ou seja, uma composição com uma lista. O que esperamos que ocorra com ela?
     2. Esperamos que sejam adicionados os lances pertencentes ao leilão. Algo como *leilao.lances.append(lance)* é uma má prática, já que estamos muito acoplados com a implementação da classe *Leilao*.
     3. Além de ferir o princípio do Diga, não pergunte (o *Tell don't ask*), estamos ferindo outro princípio chamado Lei de Demeter, ou o Princípio do menor conhecimento (*Principle of least knowledge*).
     4. Esse princípio diz que devemos ter o menor conhecimento sobre a implementação da classe. Dessa forma, evitamos o acoplamento entre as classes do sistema.
     5. Claro, o acoplamento sempre existirá. Quando estamos utilizando uma lista, string, dicionário, estamos acoplados com essa classe. Porém, isso é chamado de "acoplamento bom". A chance de uma dessas classes mudarem e afetarem o código são muito pequenas.
     6. Acoplar com classes estáveis é muito melhor do que com classes instáveis
     7. Então só devemos nos acoplar com classes do sistema?
     8. Não! No código que escrevemos existem classes mais estáveis do que outras. São essas que devemos optar por acoplar.
     9. Seguindo esses princípios como a Lei de Demeter e o Diga, não pergunte, escrevemos um código mais simples de ser alterado e escalado.
     10. Quem quiser ver um pouco mais sobre a Lei de Demeter, deixo esse texto (<https://en.wikipedia.org/wiki/Law_of_Demeter>), em inglês, da Wikipédia.
  2. Cópia profunda de listas:
     1. Quando utilizamos uma cópia rasa, apenas a referência da lista, neste caso, é diferente. Todos os outros objetos dentro dessa lista compartilham a mesma referência. Ou seja, apesar da lista de lances ser uma cópia, os lances dentro da lista copiada são os mesmos lances do leilão.
     2. Para que os lances sejam diferentes, precisamos copiar a lista profundamente. Por isso os termos cópia rasa e cópia profunda.
     3. Link para saber mais sobre: <http://blog.alura.com.br/como-fazer-copia-de-lista-python/>.
  3. O que aprendemos:
     1. Como realizar uma cópia rasa de lista;
     2. Um pouco de programação defensiva;
     3. Conceitos como encapsulamento e comportamento;
     4. Como os testes ajudam na modelagem das classes.

1. **Aula 4 – Novas Regras de Negócio e Testando Exceções:**
   1. SHIFT + F6: atalho para renomear arquivos no pycharm.
   2. Refatorar: Não serve somente para classes e métodos, o nome também é dado quando vamos encapsular uma linha de código que estava sendo usada diretamente, em uma variável, por exemplo:

def test\_deve\_permitir\_propor\_lance\_caso\_o\_leilao\_não\_tenha\_lances(self):  
 self.leilao.propoem(self.lance\_do\_gui)  
 self.assertEqual(1, len(self.leilao.lances))

def test\_deve\_permitir\_propor\_lance\_caso\_o\_leilao\_não\_tenha\_lances(self):  
 self.leilao.propoem(self.lance\_do\_gui)

quantidade\_de\_lances\_recebido = len(self.leilao.lances)   
 self.assertEqual(1, quantidade\_de\_lances\_recebido)

* + 1. Isso ajuda e facilita a leitura e entendimento do teste.
  1. Colocar [-1] indica que quer pegar o último de uma lista/dicionário:

def propoem(self, lance: Lance):  
 if self.\_\_lances[-1].usuario != lance.usuario:  
 if lance.valor > self.maior\_lance:  
 self.maior\_lance = lance.valor  
 if lance.valor < self.menor\_lance:  
 self.menor\_lance = lance.valor  
 self.\_\_lances.append(lance)

* + 1. No primeiro if temos que: se o último usuário de um lance for diferente de usuário de lance, então executa o código.
    2. Porém, se tentarmos executar o código acima, nosso teste dará erro, pois se a lista está vazia, não tem como pegar o último usuário para fazer a validação.
    3. Para concertar isso podemos criar outra condição dentro dessa:

def propoem(self, lance: Lance):  
 if len(self.\_\_lances) == 0 or self.\_\_lances[-1].usuario != lance.usuario:  
 if lance.valor > self.maior\_lance:  
 self.maior\_lance = lance.valor  
 if lance.valor < self.menor\_lance:  
 self.menor\_lance = lance.valor  
 self.\_\_lances.append(lance)

* + 1. Dessa forma dizemos que, se o tamanho de lances, ou seja, a quantidade de lances existentes for igual a 0, ou se o último usuário de lances for diferente o usuário de lance atual, pode executar o bloco de código seguinte.
    2. Porém, essa forma de abordagem e escrita de código não é pythonica, ou seja, não seria o ideal. Para fazer de uma forma mais coerente fazemos assim:

def propoem(self, lance: Lance):  
 if not self.\_\_lances or self.\_\_lances[-1].usuario != lance.usuario:  
 if lance.valor > self.maior\_lance:  
 self.maior\_lance = lance.valor  
 if lance.valor < self.menor\_lance:  
 self.menor\_lance = lance.valor  
 self.\_\_lances.append(lance)

* + 1. Ou seja, se o *self.\_\_lances* não estiver vazio (se deixasse sem o *not*, ele daria como False (porque a lista estaria vazia) e precisamos do True quando ela está vazia), ou se o último usuário de lances for diferente do atual usuário de lance, rode o código.
  1. *Try* e *Except*: Dois comandos de validação. O primeiro sugere ao python fazer o bloco de código seguinte, se não for possível executar, então faça essa exceção:

try:  
 self.leilao.propoem(self.lance\_do\_gui)  
 self.leilao.propoem(lance\_do\_gui200)  
except ValueError:  
 quantidade\_de\_lances\_recebidos = len(self.leilao.lances)  
 self.assertEqual(1, quantidade\_de\_lances\_recebidos)

* + 1. O *ValueError* é o que está sendo esperado pelo *except*.
    2. Ele está basicamente dizendo para o python tentar propor ambos os lances no leilão, mas se não der e receber um *ValueError*, então execute a verificação se a quantidade de lances recebidos é igual a 1.
    3. É dessa maneira que tratamos uma exceção, evitando a parada do código.
    4. Caso não tenhamos colocado a exceção (*else* do if no código principal, não do teste) que lançaria a mensagem de erro para executar a exceção do *try*, podemos pedir para que o teste falhe exibindo essa mensagem:

try:  
 self.leilao.propoem(self.lance\_do\_gui)  
 self.leilao.propoem(lance\_do\_gui200)  
 self.fail(msg='Não lançou excessão')  
except ValueError:  
 quantidade\_de\_lances\_recebidos = len(self.leilao.lances)  
 self.assertEqual(1, quantidade\_de\_lances\_recebidos)

* + 1. Porém, se o *else* no código principal existir, ele roda e passa o teste normalmente.
  1. Outra maneira de lançar erros e trabalhar com exceções é utilizando o with:

with self.assertRaises(ValueError):  
 self.leilao.propoem(self.lance\_do\_gui)  
 self.leilao.propoem(lance\_do\_gui200)

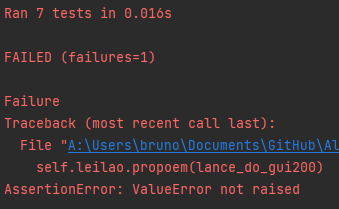
* + 1. Nesse caso ele está dizendo que só executará o código seguinte se o código principal tive subido o erro *ValueError*, aí nesse caso, podemos apagar as duas últimas linhas anteriores pois já temos o *assert* no *with*.
    2. O teste passa se o *else* estiver lá no código principal, mas falha se ele for deletado:

def propoem(self, lance: Lance):  
 if not self.\_\_lances or self.\_\_lances[-1].usuario != lance.usuario:  
 if lance.valor > self.maior\_lance:  
 self.maior\_lance = lance.valor  
 if lance.valor < self.menor\_lance:  
 self.menor\_lance = lance.valor  
 self.\_\_lances.append(lance)  
 else:  
 raise ValueError('O mesmo usuário não pode propor 2 lances seguidos!!')

* + 1. Com o *else*, zero falhas. Sem o *else*:

def propoem(self, lance: Lance):  
 if not self.\_\_lances or self.\_\_lances[-1].usuario != lance.usuario:  
 if lance.valor > self.maior\_lance:  
 self.maior\_lance = lance.valor  
 if lance.valor < self.menor\_lance:  
 self.menor\_lance = lance.valor  
 self.\_\_lances.append(lance)

* + 1. Recebemos o erro de que o *ValueError* não subiu:

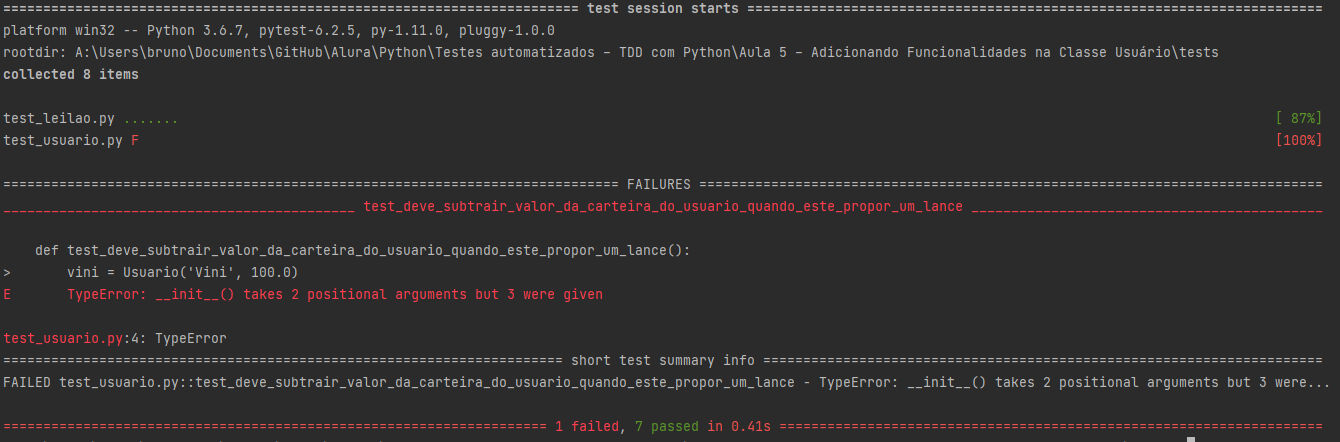


* 1. Tudo o que estamos fazendo, programando com testes, não é nada menos do que a chamada: TDD – desenvolvimento orientado a testes.
  2. Teste de regressão: é o TDD. Basicamente a vantagem de fazer TDD é que sempre que implementamos uma nova funcionalidade no nosso código, conseguimos garantir que ele não irá quebrar em outras partes, pois já temos os testes das partes que já existiam antes e eles continuam rodando normalmente, garantindo que aquelas que já funcionavam, continuam funcionando.
  3. Baby steps: Basicamente ir passo a posso, de pouquinho a pouquinho no nosso código, ajudando a entender melhor o nosso próprio código a codar uma classe.
     1. Contudo, apesar de ser uma boa prática, se você já conhece a regra de negócio que vai implementar, não existe necessidade de ir em baby steps novamente, pode ir implementar o código direto e depois fazer os testes para garantir.
     2. Vimos uma abordagem de escrita de testes, os baby steps, ou passos de bebê. Nessa abordagem, começamos primeiro pelos testes e fazemos o mínimo possível para o teste passar. Quando ele passa, refatoramos o teste e o código e vamos mantendo esse ciclo no desenvolvimento.
     3. Apesar dos baby steps ajudarem na escrita do código de testes e no entendimento das regras de negócio, não precisamos fazê-lo sempre.
     4. Quando já temos conhecimento claro das regras de negócio ou então quando já conhecemos o problema, não precisamos realizar baby steps.
     5. Existe um debate na comunidade de como praticar o TDD realizando os baby steps e escrevendo primeiro os testes ajudam a escrever um código mais simples, com um melhor algoritmo. Porém, isso também é relativo.
     6. A verdade é que escrever o código é algo muito pessoal. Existem pessoas que começam pelo código e pessoas que começam pelos testes. Existem pessoas que fazem baby steps, outras que não. O importante é o código estar funcionando sem erros quando for solicitado.
  4. O que aprendemos:
     1. Como testar exceções com a *unittest*;
     2. O que são baby steps;
     3. Vimos que testes também são refatorados para atender as novas regras de negócio.

1. **Aula 5 – Adicionando Funcionalidades na Classe Usuário:**
   1. Pytest: Outra biblioteca de testes espetacular. Além da unitteste existem outras bibliotecas de testes no mercado e essa promete deixar ainda mais fácil escrever os testes. Link para a documentação oficial: <https://docs.pytest.org/en/6.2.x/>.
      1. Essa biblioteca sugere, na documentação, que nós criemos uma pasta separada da aplicação para fazermos os testes, dessa maneira, caso precisemos enviar nossa aplicação para algum lugar, os testes não vão junto. Ela considera isso uma boa prática.
      2. A menos que você queira disponibilizar os testes junto, nesse caso, ela sugere que criemos uma pasta para os testes dentro da aplicação, ao lado do projeto.
   2. A praticidade do pytest chega no nível de não termos que importar a biblioteca, criar uma classe do teste que queremos fazer e herdar a biblioteca nela.
      1. A simplicidade chega a tanto que basta criar uma def e executar o teste para ver se o código está funcionando como o esperado.
      2. Nós podemos fazer tudo isso se quisermos, mas não há a necessidade.
      3. Diferente do unittest, não conseguimos usar o self.assertEqual() para verificar o que queremos. No caso do pytest basta colocar a palavra reservada assert do python e a condição na frente que ele funciona da mesma forma:

from src.leilao.dominio import Usuario, Leilao  
  
def test\_deve\_subtrair\_valor\_da\_carteira\_do\_usuario\_quando\_este\_propor\_um\_lance():  
 vini = Usuario('Vini')  
  
 leilao = Leilao('Celular')  
  
 lance = vini.propoem\_lance(leilao, 50.0)  
  
 assert vini.carteira == 50.0

* + 1. O jeito de fazer os testes com o pytest é diferente do unittest, com o py basta abrir o terminal na pasta onde os testes estão e comandar pytest. Ele irá executar todos os testes/funções que tenham o prefixo ou sufixo *test*.
    2. Por isso é tão importante manter esse padrão de nomenclatura.
    3. A partir do momento que executamos o pytest essa é a tela que ele exibirá:



* + 1. Ele diz a porcentagem de testes rodados em cada arquivo de testes e quais deles falharam, além de separar na parte do meio quais foram os testes que falharam e o tipo de erro.
    2. Na parte inferior ele mostra um breve resumo dos erros.
    3. No nosso caso atual ele está dizendo que o nosso erro foi ter passado 3 argumentos/parâmetros para o \_\_init\_\_ (construtor da classe usuários) quando deveríamos ter passado 2: o self (que o python quem passa) e o nome do usuário criado. Além desses 2, tentamos passar um valor na carteira.
    4. Para corrigir esse problema precisamos ir no código do domínio, classe usuário, e implementar a carteira:

class Usuario:  
 def \_\_init\_\_(self, nome, carteira):  
 self.\_\_nome = nome  
 self.\_\_carteira = carteira  
  
 @property  
 def nome(self):  
 return self.\_\_nome  
  
 @property  
 def carteira(self):  
 return self.\_\_carteira

* + 1. Ao executar os testes, percebemos que TODOS os testes falharam, aí entra o conceito de teste de regressão, que basicamente consiste de executar todos os testes já feitos para garantir que o que estava funcionando antes, continue funcionando.
    2. No nosso caso deu erro em todos pois adicionamos um novo parâmetro no construtor de usuário, portanto, todos os usuários criados nos outros testes, cujo não tem esse parâmetro novo da carteira pois foram criados antes dessa nova feature, não rodaram e deram erro.
    3. Essa é a grande importância dos testes, se não fosse por isso, levaríamos bem mais tempo para descobrir que o resto do código todo daria erro.
    4. Após concertar todos os testes passando o novo parâmetro, agora apenas o último que fizemos deu outro erro: o método propõem\_lance() não existe.
    5. Criamos o teste:

def propoem\_lance(self, leilao, valor):  
 lance = Lance(self, valor)  
 leilao.propoem(lance)  
  
 self.\_\_carteira -= valor

* + 1. Temos que passar o leilão que queremos que o usuário que está propondo quer participar e qual o valor que quer propor no lance, portanto, lance recebe o construtor do objeto lance, que está recebendo o self por estar no mesmo código que a classe dele em si, e o valor proposto pelo usuário, então o leilão vai propor o lance dado e a carteira subtrai o valor que ela tinha menos o valor proposto pelo usuário, concertando o erro.
  1. Criamos um novo teste para ter certeza que nosso código não permitiria que um usuário desse um lance maior do que o valor da carteira, mas ele estava falhando:

def test\_nao\_deve\_permitir\_propor\_lance\_com\_valor\_maior\_que\_o\_da\_carteira():  
 vini = Usuario('Vini', 100.0)  
  
 leilao = Leilao('Celular')  
  
 vini.propoem\_lance(leilao, 150.0)  
  
 assert vini.carteira == 100.0

* + 1. Nesse caso, o que fizemos para concertar foi importar a biblioteca pytest, criar uma exceção no nosso código app que lançasse um erro caso tentassem propor um lance maior que o valor da carteira e substituímos o assert por um with da biblioteca pyteste que funciona semelhante à do unittest:
       1. Código do app:

def propoem\_lance(self, leilao, valor):  
 if valor > self.\_\_carteira:  
 raise ValueError('Não pode propor um lance com valor maior que o valor da carteira!')  
 else:  
 lance = Lance(self, valor)  
 leilao.propoem(lance)  
  
 self.\_\_carteira -= valor

* + - 1. Código atual do teste:

def test\_nao\_deve\_permitir\_propor\_lance\_com\_valor\_maior\_que\_o\_da\_carteira():  
 with pytest.raises(ValueError):  
 vini = Usuario('Vini', 100.0)  
  
 leilao = Leilao('Celular')  
  
 vini.propoem\_lance(leilao, 150.0)

* + 1. Dessa forma consertamos a falha no teste.
    2. Caso tivéssemos colocado o with, mas não tivéssemos lançado a exceção no app, o teste falharia novamente justamente dizendo que não possui a exceção.
  1. Nos nossos testes criamos muita repetição de cenário que poderia ser substituído facilmente por um setup, caso estivéssemos utilizando o unittest. Como não estamos, temos outra forma de lidar com esse copia e cola usando o pytest.fixture:

@pytest.fixture  
def vini():  
 return Usuario('Vini', 100.0)

* + 1. Essa decoração basicamente diz para o python que essa função é uma fixture e queremos que toda vez que ela seja passada como parâmetro em outra função, ele entenda isso e instancie nosso objeto usuário vini como primeira atividade em todos os testes.
    2. Desse modo, podemos passar o vini como parâmetro em todos os nossos testes e deletar ele do código, assim:
       1. Vini no código:

def test\_deve\_subtrair\_valor\_da\_carteira\_do\_usuario\_quando\_este\_propor\_um\_lance():  
 vini = Usuario('Vini', 100.0)  
  
 leilao = Leilao('Celular')  
  
 vini.propoem\_lance(leilao, 50.0)  
  
 assert vini.carteira == 50.0

* + - 1. Vini como parâmetro:

def test\_deve\_subtrair\_valor\_da\_carteira\_do\_usuario\_quando\_este\_propor\_um\_lance(vini):  
 leilao = Leilao('Celular')  
  
 vini.propoem\_lance(leilao, 50.0)  
  
 assert vini.carteira == 50.0

* + 1. Das duas formas o teste continua passando, mas na segunda vini funciona como o setup do unittest.
    2. Outra vantagem de se utilizar essa técnica é que nos testes que não utilizamos o vini, basta não passar como parâmetro. Diferentemente do unittest que tudo tinha que ir pro setup e precisávamos escolher o que passar para ele ou não para evitar problemas de performance.
    3. Com o pytest, podemos deixar ele com a decoração e somente passar como parâmetro para os testes que utilizam ele, desse modo, nos testes que não utilizam, ele não será instanciado, evitando problemas de performance por criação de objetos não utilizados para todos os testes.
    4. Podemos utilizar e fazer essa técnica para todos os objetos que criamos várias vezes para vários testes diferentes:

@pytest.fixture  
def vini():  
 return Usuario('Vini', 100.0)  
  
  
@pytest.fixture  
def leilao():  
 return Leilao('Celular')

* + 1. Para passar os próximos fixtures, basta fazer igual ao vini e passar como parâmetro separando por “,”. Da mesma forma que se usa herança.