**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.