**Python 3 – Orientações a Objetos**

**Curso 1 – Introdução a Orientação a Objetos:**

1. **Aula 1 – Aprenda o Paradigma OO com Python:**
   1. Ao criar uma função, para contas de um banco, por exemplo, podemos colocar como parâmetros o que desejamos que essa conta tenha, como titular, número, saldo, dentre outros.
      1. Dentro da função, criamos uma variável conta que recebe um dicionário: conta = {“titular”: titular, “numero”: numero, por exemplo, e retorna a conta no final.
   2. Para criar uma nova conta, criamos uma variável “conta” que recebe a função “cria\_conta(titular, numero, saldo...)”, com os parâmetros já definidos, desse modo, quando chamarmos essa “conta[“titular”]”, por exemplo, receberemos o nome que foi dado como parâmetro “titular” logo aqui acima.
   3. A ideia central do paradigma orientação a objeto é: Dado e funcionalidade (comportamentos) andam juntos.
   4. O que aprendemos:
      1. Dicionário;
      2. Funções;
      3. Encapsulamento de código.
2. **Aula 2 – Classes e Objetos:**
   1. Classes são como receitas de bolo, ou formas, que servem para que o projeto final seja um só.
   2. Class: Determina uma classe.
   3. Nomenclatura em classes: NomeDeUmaClasseSeEscreveAssim.
   4. Pass: Faz com que uma classe ou função passe mesmo que não tenha nada escrito ainda. Python entende que anda será escrito algo lá e não dá erro durante a execução do código.
   5. Para chamar uma classe no console, é só colocar o nome dela (com a inicial Maiúscula) seguida de (), como uma função.
   6. Quando guardamos uma classe dentro de uma variável (ex.: conta = Conta()), chamamos essa variável de referência. Ele guarda o endereço da memória onde essa classe foi alocada para saber onde encontrar esse objeto.
   7. O parâmetro *self* é o que indica a referência, a localização daquela classe.
   8. Def \_\_init\_\_(self):: É uma função construtora, responsável por criar um objeto.
      1. Funciona basicamente como a função *cria\_conta* que falamos lá acima.
      2. Logo depois do *self* colocamos outros parâmetros, como: numero, titular, saldo, dentre outros.
      3. Utilizamos o self para nos referir a locação da referência, ao local onde esse objeto está armazenado.
      4. O código final fica assim:



* + 1. Colocando os parâmetros ao invés de os números e nomes direto, torna esse molde interativo, fazendo com que possamos criar várias contas utilizando o mesmo construtor, apenas com parâmetros diferentes.
  1. Podemos criar atributos pré definidos. Caso queiramos que o limite de todas as contas seja o mesmo, menos para contas especiais, basta colocar, junto com os atributos, limite = 1000.0, por exemplo. Nesse caso, durante a criação das contas só colocamos o atributo limite naquelas que terão limites especiais, caso contrário, só precisamos atribuir até o saldo, pois o limite será adicionado automaticamente. Ex.:



* 1. Através da referência podemos acessar diretamente os atributos do objeto criado. Ex.:



* + 1. Primeira linha: variável conta recebe a classe Conta com os atributos.
    2. Segunda e Terceira linha: chama a referência e diz que a variável conta é um objeto no endereço 0x00...
    3. Quarta e Quinta linha: acesso direto ao atributo saldo da conta através da referência/variável conta e exibição do mesmo.
  1. O que aprendemos:
     1. Classes;
     2. Objetos;
     3. Função construtora;
     4. Endereço e referência de objetos;
     5. Atributos de classe;
     6. Acesso aos atributos através do objeto.

1. **Aula 3 – Implementando Métodos:**
   1. Métodos são comportamentos que os objetos tem.
   2. Métodos são funções que criamos dentro de uma classe para executar coisas.
   3. Passando como parâmetro o *self* no início da função, sempre estaremos nos referindo à referência que escolhemos no console, seja conta 1, 2, 3... Ex.:



* 1. Para utilizar/chamar o método, precisamos colocar a referência “.” E o método que quer executar. Ex.:



* 1. Caso queiramos depositar ou sacar, a lógica é a mesma para as funções criadas lá no começo, com a diferença de que, ao invés de passar o parâmetro “conta” na hora de criar o método, passamos o parâmetro “self”, para que a modificação seja feita na conta em que estamos mexendo, seguido do parâmetro “valor”, que será adicionado ou tirado. Ex.:



* 1. No console fica assim:



* 1. No console não precisamos passar a referência nos () do método pois ela já foi chamada no início.
  2. Por enquanto estamos fazendo métodos de apenas algumas linhas, mas esses métodos podem conter mais 100 linhas de código, o que para o usuário não importa, pois está chamando o método que quer usar e pronto.
  3. Quando você cria um objeto conta dentro de uma variável conta, você está dizendo que sempre que mencionar a variável conta, está se referindo ao objeto conta, agora, se você criar um outro objeto conta dentro da mesma variável conta que já tinha um objeto conta, a nossa variável conta irá começar a se referir e ir somente para esse novo objeto conta criado, o antigo objeto conta será totalmente desvinculado apesar de ainda existir, ou seja, ele ainda ocupa memória, mas se torna inalcançável.
  4. No python existe um processo que recicla esses objetos perdidos/inutilizados, eliminando-os e dando espaço para novos objetos ocuparem.
  5. None: faz com que uma referência que antes apontava para algo, deixar de apontar, ou seja, se eu criar uma variável chamada outraRef que recebe a variável conta (outraRef = conta) agora temos 2 variáveis referências que nos levam para o mesmo endereço, o objeto conta. Caso eu queira que a variável/referência outraRef deixe de apontar para esse objeto conta, preciso fazer com que ela receba *none* (outraRef = none).
  6. O que aprendemos:
     1. Métodos, que definem o comportamento de uma classe;
     2. Criação de métodos;
     3. Como chamar métodos através do objeto;
     4. Acesso aos atributos através do self;
     5. Garbage Collector;
     6. O tipo None.

1. **Aula 4 – Encapsulamento:**
   1. Para deixar um atributo privado, basta colocar dois “\_\_” antes de cada atributo (em python. Outras linguagens usam palavras como “private”). Ex.:



* + 1. Quando fazemos isso esses atributos “somem” quando tentamos acessar os métodos e atributos. Ex.:
       1. Sem ser privado:



* + - 1. Sendo privado:



* + 1. Quando está como privado, as primeiras coisas que aparecem são os métodos, em seguida os atributos, ao contrário de quando não estava privado. Porém, nos atributos quando privados, podemos ver que sua aparência está diferente de quando eram públicos, aparecendo *“\_Conta\_\_saldo”*, *“\_Conta\_\_limite”*, *“\_Conta\_\_numero”*...
    2. Apesar de o nome ter mudado, ainda temos acesso direto aos atributos se selecionarmos eles, porém, como um desenvolvedor experiente você sabe que isso não é uma boa prática e se trata de um aviso do python, dizendo que apesar de poder, você não deve utilizar o acesso direto, sempre pelos métodos.
  1. Podemos criar métodos que utilizem outros métodos. Caso queiramos transferir dinheiro de uma conta para a outra, precisamos sacar um valor da origem e depositar no destino, certo!? Tendo isso em mente, podemos criar um método transfere() que faça isso automaticamente. Ex.:



* + 1. Nesse caso, o self é a origem e não precisa ser declarado novamente, uma vez que o método será chamado a partir da conta que desejamos fazer a transferência, ficando assim no console:



* 1. Isso se chama encapsulamento, pois colocamos dois métodos encapsulados dentro de um outro, facilitando/encurtando o caminho para a execução do que queremos, desse modo, ao invés de ter que usar 2 métodos separadamente, um para sacar o dinheiro de uma conta e outro para depositar esse dinheiro em outra conta, criamos um método que encapsula esses dois e faz esse processo todo automaticamente.
  2. Nossos códigos precisam ser coesos, ou seja, precisamos colocar métodos dentro de classes que fazem sentido eles estarem, não faz sentido, por exemplo, criar um método que verifica se o cliente é inadimplente ou não, na classe Conta(), esse método deveria estar na classe Cliente(). Precisamos ficar atentos pois esse tipo de coisa acontece o tempo todo e é normal, já que faz parte do dia a dia modificarmos códigos para que fiquem cada vez melhores e fazendo mais sentido.
  3. Uma classe deve ter responsabilidade única, quando isso não ocorre, chamamos ela de sem coesão, pois está fazendo mais coisas do que deveria.
     1. Esse é o princípio de *responsabilidade única*.
  4. Além desse princípio, no início dos anos 2000 Robert C. Martin definiu outros princípios que foram denominados **SOLID:**
     1. **S** - Single responsibility principle;
     2. **O** - Open/closed principle;
     3. **L** - Liskov substitution principle;
     4. **I** - Interface segregation principle;
     5. **D** - Dependency inversion principle.
  5. O que aprendemos:
     1. Atributos privados;
     2. Encapsulamento de código;
     3. Encapsulamento da manipulação dos atributos nos métodos;
     4. Coesão do código.

1. **Aula 5 – Usando Propriedades:**
   1. Getter: basicamente uma nomenclatura para métodos que só devolvem informações, como somente o número do saldo (sem o extrato), somente o titular, limite etc. Quando quiser um método que faça essas coisas, basta nomear ele como get\_saldo(), get\_titular(). Ex.:



* 1. Setters: basicamente nomenclatura dada para métodos que definem coisas, como por exemplo, um novo limite. Setters nunca retornam nada, apenas settam coisas novas. Ex.:





* 1. Sempre usar getters e setters só quando forem necessários/realmente fazerem sentido.
  2. @property: Quando colocado antes de um método, mascara ele e faz com que se pareça com um atributo na hr de chamar no console. Ex.:



* + 1. Usando isso podemos fazer com que ao invés de acessar o atributo direto, o usuário esteja tendo acesso à um método getter, mas sem o nome get\_nome, que não precisa usar nem mesmo os () para ser executado. Ex. no console:



* + 1. Outra vantagem, como podemos ver, é que o nome já retorna tratado, ou seja, se na hora do cadastro você digitar o nome tudo em minúsculas, na hora de acessar o “atributo” (que na verdade é a nossa propriedade) ele retorna o nome tratado em *title()*.



* + 1. Na imagem acima podemos ver a propriedade “nome” aparecendo em primeiro e o atributo real logo abaixo como “\_Cliente\_\_nome”, como sugestão do próprio python.
  1. Podemos fazer a mesma coisa com os setters, mas utilizando uma sintax um pouco diferente:



* + 1. Como podemos ver, ao invés de usar o @property, colocamos o nome que queremos dar para o método seguido de um “.” E a função que ele executará, sendo, nesse caso, um setter.
    2. O resto do programa é basicamente o método que faz a alteração do que queremos, assim como foi feito no limite da conta lá em cima.
    3. No console fica assim:



* + 1. Se quiser chamar o getter, basta dar enter ao colocar .propriedade, se quiser chamar o setter, faça uma nova atribuição de valor, como se estivesse fazendo acesso direto ao atributo.
  1. O que aprendemos:
     1. Métodos de leitura dos atributos, os getters;
     2. Métodos de modifição dos atributos, os setters;
     3. Propriedades.

1. **Aula 6 – Métodos Privados e Estáticos:**
   1. Podemos criar condições usando métodos.
   2. É possível retornar uma condição de um método direto (true/false), sem ter a necessidade de criar ela em uma variável antes do retorno.
   3. Para indicar que um método é privado e que não foi feito para ser utilizado fora da classe, colocamos “\_\_” antes do nome do método, assim como é feito com os atributos.
   4. Métodos estáticos são os métodos da classe.
      1. Digamos que a gente queira que o código do banco aparece mesma que não tenhamos criado uma conta, ao tentar ver qual é o código isso não funciona pois o python automaticamente vai criar esse objeto e ainda assim não retornará o código do banco, para isso temos que criar esses métodos estáticos.
      2. @staticmethod: diz para o python que é um método estático. Usa da mesma maneira que os outros @’s.



* + 1. No console:



* + 1. Lembrando que para isso precisamos chamar a classe Conta antes do “.”.
  1. Podemos retornar dicionários diretamente, sem a necessidade de criar uma variável/atributo para colocá-lo dentro:

