Métodos POO

Class = é algo básico

Objeto = algo mais especifico

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def apresentar(self):
        print(f"Olá meu nome é {self.nome}, e tenho {self.idade} anos."

    def __str__(self):
        return f"{self.nome}"
```

Herança

Herança é quando uma nova classe herda parâmetros e funções de uma classe anterior

```
class Pessoa:
    def __init__(self, nome, idade, genero):
        self.nome = nome
        self.idade = idade
        self.genero = genero

    def apresentacao(self):
        print(f"Nome: {self.nome}\nIdade: {self.idade}\nGênero: {self.genero}.")

class Funcionario(Pessoa):
    def __init__(self, nome, idade, genero, cargo):
        super().__init__(nome, idade, genero)
        self.cargo = cargo
```

```
pessoa1 = Funcionario("Beatriz", 18, "Feminino", "Aprendiz")
pessoa1.apresentacao()
```

Polimorfismo

Polimorfismo é quando uma classe herda os parâmetros e funções de outra classe a adapta de acordo com a sua necessidade

```
class Pessoa:
    def __init__(self, nome, idade, genero):
        self.nome = nome
        self.idade = idade
        self.genero = genero

    def apresentacao(self):
        print(f"Nome: {self.nome}\nIdade: {self.idade}\nGênero: {self.genero}.")

class Funcionario(Pessoa):
    def __init__(self, nome, idade, genero, cargo):
        super().__init__(nome, idade, genero)
        self.cargo = cargo

    def apresentacao(self):
        print(f"Nome: {self.nome}\nIdade: {self.idade}\nGênero: {self.genero}\nC

pessoa1 = Funcionario("Beatriz", 18, "Feminino", "Aprendiz")
pessoa1.apresentacao()
```

Encapsulamento

Encapsulamento é quando definimos quando uma variável irá ser pública ou não, e definimos métodos para edita-la.

Por exemplo, estamos desenvolvendo um código para um banco, onde as principais informações do cliente é Nome do titular da conta e o saldo dela. Até então, tudo bem que mostrar o nome do cliente, não se trata de um dado

sensível, mas em questão a saldo disponível na conta é perigoso deixar que qualquer pessoa possa visualizar.

Pensando nisso usamos o método de encapsulamento, para esconder dada informação usamos o “__.” na frente da variável.

```
class ContaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular #Variavel pública
        self.__saldo = saldo   #Variável privada

    def get_saldo(self):
        return self.__saldo


conta = ContaBancaria("Beatriz", 1900)

print(conta.get_saldo())
```

Primeiro definimos as variáveis normalmente no começo da função, mas dentro dela definimos a variável “saldo” como privada. Caso for preciso visualizar a informação contida em “saldo” precisamos chamar o método “get”.

Quando declaramos que uma variável é privada permitimos que somente dentro da classe poderá haver alterações, então quando for tentando modificar o valor dela por fora do método será totalmente em vão. Exemplo:

```
class ContaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular
        self.__saldo = saldo

    def apresentacao(self):
        print(f"Titular: {self.titular}\nSaldo: {self.__saldo}")

conta = ContaBancaria("Beatriz", 1900)
```

```
conta.saldo = 1908
conta.apresentacao()
```

Ao rodar o código nenhum problema irá aparecer, mas quando chamamos a apresentação dos dados é notável que o valor do saldo não foi alterado. Isso aconteceu pois tornamos a variável saldo privada e ao tentarmos mudar seu valor criamos uma variável nova ao invés de alterar seu valor

Agora para realmente alterarmos o valor da variável precisamos criar o método set, na qual iria iniciar o novo valor para aquela variável.

```
class ContaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular
        self.__saldo = saldo

    def get_saldo(self):
        return self.__saldo

    def set_saldo(self, saldo):
        self.__saldo = saldo



    def apresentacao(self):
        print(f"Titular: {self.titular}\nSaldo: {self.__saldo}")

conta = ContaBancaria("Beatriz", 1900)
conta.set_saldo(1747)

conta.apresentacao()
```

Resumo:

O método get permite a visualização do valor dentro da variável. Pensando na possibilidade de modificação desse valor, precisamos utilizar o método “set”.

```
class ContaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular #Atributo público
        self.__saldo = saldo

    @property
    def saldo(self):
        return  self.__saldo

    @saldo.setter
    def saldo(self,valor):
        if valor < 0:
            print("Saldo não pode ser negativo")
    #def get_saldo(self):
    #    return  self.__saldo
    #def set_saldo(self, valor):
    #    if valor < 0:
    #        raise   ValueError("Saldo nao pode ser negativo")

minhaConta = ContaBancaria("Beatriz", 700)
#minhaConta.saldo(500) #modificando o valor da conta bancária
#print(minhaConta.saldo())
```

Abstração

Podemos compreender a abstração como a a definição doque deve ser tratado dentro do código. Por exemplo, pense que estamos desenvolvendo um programa que realizara envio de dinheiro, para qualquer um dos meios que o cliente optar para realizar isso (PIX ou Depósito) precisará ter validações.

Pensando de maneiras de diminuir falhas no desenvolvimento do projeto, com foco nos meios de validações, utilização o método de abstração, no qual podemos definir funções obrigatórias para desenvolvimento:

```
from abc import ABC, abstractmethod

class Pagamento(ABC):

    @abstractmethod
    def autorizar(self, valor):
        pass

    @abstractmethod
    def estornar(self, valor):
        pass

class Pix(Pagamento):
    def autorizar(self, valor):
        print(f"Transferindo R$ {valor} via PIX")

    def estornar(self, valor):
        print(f"Devolvendo R$ {valor} via PIX")
```

Caso não tivéssemos definido dentro da classe PIX as funções autorizar e estornar, ao rodar o código iria ser exibido alguma mensagem de erro falando que as funções não foram definidas.

Código da aula