**4 pilares da Orientação a Objetos em Python**

A linguagem Python possibilita que o desenvolvedor programe utilizando diferentes tipos de programação como processual, funcional, imperativa e orientação a objetos (sendo uma linguagem multiparadigma).

A programação orientada a objetos é um dos paradigmas mais utilizados atualmente, porque através dela é possível organizar a arquitetura do programa trazendo uma perspectiva mais próxima do mundo real, considerando as coisas (abstratas ou concretas) do mundo real como objetos no sistema, que interagem entre si e dá vida ao fluxo do programa.

E para garantir uma programação orientada a objetos eficientes, existem alguns pilares desse paradigma que podemos utilizar em programas escritos com Python.

**1º Pilar: Abstração**

Na programação orientada a objetos, a construção dos objetos é baseada em uma Classe que representa as características daquele objeto.

|  |
| --- |
| from abc importABCMeta abstractmethod  class Conta(metaclass=ABCMeta):  \_numero = “00000”  \_titular = “root”  saldo = 0  @abstractmethod def \_\_init\_\_ (self, numero: str, titular: str, saldo: float):  self\_numero = numero  self\_titular = titular  self\_saldo = saldo  @abstractmethod def sacar(self, value: float):  pass  @abstractmethod def depositar(self, value: float):  pass  @abstractmethod def exibir\_saldo(self):  pass |

Abstração é o princípio de criar uma classe que contenha atributos e métodos que sejam comuns a outras classes e que possam servir como base para serem herdados.

Você abstrai características comuns a N classes e fornece uma classe abstrata que pode ser herdada e servir de base para os demais.

Nesse conceito, podemos mencionar também a questão de fornecer uma classe abstrata como um contrato para que quando herdada garanta que as classes filhas irão implementar os métodos necessários, dessa forma, proteger o nosso código dando a certeza da existência e implementação do método.

Para ilustrar a abstração, imagine um sistema bancário, onde uma **Conta** bancária possa ser de diversos tipos, como, Conta poupança, Conta, Conta PJ, etc.

Cada tipo de conta, ainda é uma **Conta**, e as características comuns a todos os tipos de contas podem ser **Abstratas** em uma classe abstrata. Por exemplo:

Dessa forma eu crio uma abstração do conceito de **Conta** e disponibilizo uma classe para ser herdada por outras classes que são do tipo Conta.

Obs.: Uma classe abstrata não deve ser utilizada diretamente, deve ser vista e utilizada como uma base para outras classes como ContaCorrente, ContaPoupanca, etc.

**2º Pilar: Encapsulamento**

O princípio de encapsulamento consiste em "esconder" a parte funcional dos objetos de forma que quem utiliza não tenha que conhecer mais do que o necessário para utilizá-lo.

Por exemplo, ao dirigir um carro, se quisermos que o carro **pare de andar** , não é preciso, conheça toda a mecânica do funcionamento por dentro do veículo para que possamos **frear** . Basta pisar no freio, o freio é o encapsulamento do comportamento de frear de um carro.

Na orientação a objetos, estruturamos as classes de forma a encapsular toda regra de negócio e parte funcional relacionada a classe dentro de métodos e atributos, de forma que quem utiliza, apenas diga o que quer fazer.

Por exemplo:

|  |
| --- |
| class Biblioteca:  def \_\_init\_\_ (self, livros\_disponiveis 🡪 None);  self.livros\_disponiveis = livros\_disponiveis  def exibir\_livros(self):  for livro in self.livros\_disponiveis:  print(livro)  def emprestar\_livro(self, livro):  print(f’Você escolheu o livro: {livro}’)  if livro in self.livros\_disponiveis:  self.livros\_disponiveis\_remove(livro)  else:  print(‘Desculpe o livro não está disponível!’)  def devolver\_livro(self, livro):  self.livros\_disponiveis.append(livro)  print(f‘Obrigado por devolver o livro: {livro}’) |

Na biblioteca de classes, temos os métodos que fornecem as regras de negócio para que o programa que você utiliza não precisa conhecer o que está acontecendo ali dentro e sim pode simplesmente dizer:

|  |
| --- |
| // Biblioteca me mostre os livros: biblioteca.exibir\_livros() |

**3º Pilar: Herança**

A herança consiste em determinar que uma classe existe por si mesma, porém, ela é uma outra classe em sua essência... Por exemplo, uma ContaCorrente existe mas ela é uma Conta.

Quando a herança é utilizada, a classe que herda, automaticamente, possui os atributos e métodos definidos na classe de qual herdou.

Por exemplo, imagine que o sistema bancário tenha uma regra geral para depósito e exibição de saldo para todos os tipos de conta somente o saque muda de conta para conta. Nesse caso, podemos criar uma classe **Conta** com as características comuns das outras Contas, e as outras contas, irão herdar essas características:

|  |
| --- |
| class Conta():  \_numero = “00000”  \_titular = “root”  saldo = 0  def \_\_init\_\_(self, numero: str, titular: str, saldo: float):  self.\_numero = numero  self.\_titular = titular  self.saldo = saldo  def depositar(self, value: float):  # Regra para fazer o depósito...  def exibir\_saldo(self):  # Exiba o saldo...  # Para aplicar a herança em Python, basta referenciar a classe que se quer herdar entre parênteses: class ContaPoupanca(Conta):  def \_\_init\_\_(self, numero: str, titular: str, saldo: float):  super().\_\_init\_\_(numero, titular, saldo)  def sacar(self, value: float):  if value <= self.\_saldo:  self.\_saldo -= value  return True  return False  def titular(self):  return self.\_titular  def numero(self):  return self.\_numero |

A classe ContaPoupaca herda tudo de Conta, ou seja, possui os métodos depositar e exibir\_saldo e os atributos \_numero, \_titular e saldo e ainda pode implementar outros métodos e definir outros atributos que sejam só dela.

**4º Pilar: Polimorfismo**

O conceito do polimorfismo é permitir que comportamentos comuns a N tipos de classes possam ser definidos de forma específica para cada classe.

Imagine que um sistema bancário, defina que uma ContaCorrente e uma ContaPoupaca deve ter saldo, titular, número da conta, um método para sacar, depositar e exibir o saldo, mas, cada classe tem suas próprias regras de negócio, por exemplo, na ContaCorrente , o saque vai ter algum tipo de desconto dependendo de onde for sacado e a Conta Poupança o depósito vai ser salvo em outro tipo de saldo interno, etc.

A questão é: possuem comportamentos diferentes, mas devem possuir os comportamentos porque ambas são Contas.

Por exemplo, eu posso criar uma classe abstrata chamada Conta e a partir dela definir quais são os atributos e comportamentos necessários para quem herdar dela mas as regras de negócio cada classe implementa a sua maneira, como o caso aqui em que ContaPoupanca herda de Conta ( classe abstrata) e implementa suas próprias regras de negócio (gerando o comportamento polimórfico das Contas):

from abc import ABCMeta, abstractmethod

class Conta(metaclass = ABCMeta):  
 \_numero = “00000”  
 \_titular = “root”  
 \_saldo = 0

@abstractmethod  
 def \_\_init\_\_(self, numero: str, titular: str, saldo: float):  
 self.\_numero = numero  
 self.\_titular = titular  
 self.\_saldo = saldo

@abstractmethod  
 def sacar(self, value: float):  
 pass

@abstractmethod  
 def depositar(self, value: float):  
 pass

@abstractmethod  
 def exibir\_saldo(self):  
 pass

class ContaPoupanca(Conta):

def \_\_init\_\_(self, numero: str, titular: str, saldo: float):  
 super().\_\_init\_\_(numero, titular, saldo)

def sacar(self, value: float):  
 if value <= self.\_saldo:  
 self.\_saldo -= value  
 return True  
 return False

def depositar(self, value: float):

if value > 0:

self.\_saldo += value

return True

return False

def exibir\_saldo(self):

return self.\_saldo

def titular(self):

return self.\_titular

def numero(self):

return self.\_numero

class BankingSystem:

\_\_contas = []

def \_\_gerar\_numero\_conta(self):

numero = len(self.\_\_contas) + 1

return f"{numero:05}"

def \_\_checar\_valor\_positivo(self, value):

if value <= 0:

return False

return True

def criar\_conta\_poupanca(self, deposito\_inicial: float, nome\_titular: str):

if self.\_\_checar\_valor\_positivo(deposito\_inicial) == False:

print("Deposito Inicial deve ser maior que Zero!")

return

conta = Savingsconta(self.\_\_gerar\_numero\_conta(), nome\_titular, deposito\_inicial)

self.\_\_contas.append(conta)

print("Conta criada com sucesso!")

print("Titular: ", conta.titular())

print("Número da conta: ", conta.numero())

print("Saldo: ", conta.exibir\_saldo())

def total\_of\_contas(self):

print("contas: ", str(len(self.\_\_contas)))

def acessar\_conta(self, nome\_titular: str, conta\_numero: str):

contas = [acc for acc in self.\_\_contas

if acc.titular() == nome\_titular and acc.numero() == conta\_numero]

if len(contas) == 0:

print("Conta não existe!")

return

conta = contas[0]

print("Digite 1 para sacar")

print("Digite 2 para depositar")

print("Digite 3 para exibir saldo")

escolha\_usuario = (int(input()))

if escolha\_usuario == 1:

print("sacar")

print("Digite o valor a sacar")

value = float(input())

if conta.sacar(value):

print("Saque realizado com sucesso!")

else:

print("Problema ao sacar, verifique o saldo!")

elif escolha\_usuario == 2:

print("DEPOSITAR")

print("Digite o valor do depósito")

value = float(input())

if conta.depositar(value):

print("Deposito realizado com sucesso!")

else:

print("Problema ao depositar, valor não permitido!")

elif escolha\_usuario == 3:

print(conta.exibir\_saldo())

else:

print("Escolha inválida!")

banking\_system = BankingSystem()

while True:

print("Digite 1 para criar uma conta poupança")

print("Digite 2 para acessar a conta")

print("Digite 3 para exibir o total de contas")

print("Digite 0 para sair")

escolha\_usuario = (int(input()))

if escolha\_usuario == 1:

print("CREATE")

print("Digite o nome do titular da conta")

nome\_titular = input()

print("Digite o deposito inicial")

initial\_deposit = float(input())

banking\_system.criar\_conta\_poupanca(initial\_deposit, nome\_titular)

elif escolha\_usuario == 2:

print("ACCESS")

print("Digite o name do titular da conta")

nome\_titular = input()

print("Digite o conta numero")

conta\_numero = input()

banking\_system.access\_conta(nome\_titular, conta\_numero)

elif escolha\_usuario == 3:

banking\_system.total\_of\_contas()

elif escolha\_usuario == 0:

quit()

else:

print("Escolha inválida!")

**Definimos Polimorfismo como um princípio a partir do qual as classes derivadas de uma única classe base são capazes de invocar os métodos que, embora apresentem a mesma assinatura, comportam-se de maneira diferente para cada uma das classes derivadas**.

**O Polimorfismo é um mecanismo por meio do qual selecionamos as funcionalidades utilizadas de forma dinâmica por um programa no decorrer de sua execução**.

Com o **Polimorfismo**, os mesmos atributos e objetos podem ser utilizados em objetos distintos, porém, com implementações lógicas diferentes.

Por exemplo: podemos assumir que uma bola de futebol e uma camisa da seleção brasileira são artigos esportivos, mais que o cálculo deles em uma venda é calculado de formas diferentes.

Outro exemplo: podemos dizer que uma classe chamada Vendedor e outra chamada Diretor podem ter como base uma classe chamada Pessoa, com um método chamado CalcularVendas. Se este método (definido na classe base) se comportar de maneira diferente para as chamadas feitas a partir de uma instância de Vendedor e para as chamadas feitas a partir de uma instância de Diretor, ele será considerado um método polimórfico, ou seja, um método de várias formas.

Assim podemos ter na classe base o método CalcularVendas:

public decimal CalcularVendas()

{

decimal valorUnitario = decimal.MinValue;

decimal produtosVendidos = decimal.MinValue;

return valorUnitario \* produtosVendidos;

}

Na classe Vendedor temos o mesmo método, mais com a codificação diferente:

public decimal CalcularVendas()

{

decimal valorUnitario = 50;

decimal produtosVendidos = 1500;

return valorUnitario \* produtosVendidos;

}

O mesmo ocorre na classe Diretor:

public decimal CalcularVendas()

{

decimal valorUnitario = 150;

decimal produtosVendidos = 3800;

decimal taxaAdicional = 100;

return taxaAdicional + (valorUnitario \* produtosVendidos);

}

Polimorfismo

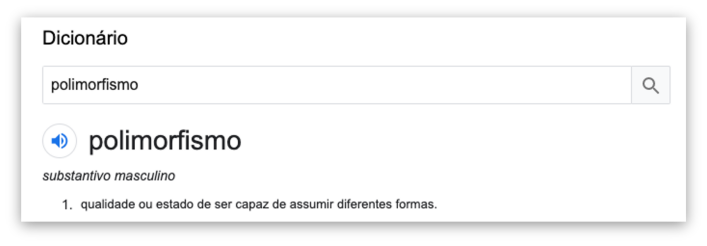
O polimorfismo deriva da palavra polimorfo, que significa multiforme, ou que pode variar a forma. Para a POO, polimorfismo é a habilidade de objetos de classes diferentes responderem a mesma mensagem de diferentes maneiras. Ou seja, várias formas de responder à mesma mensagem. Veja a figura a seguir para entender onde se localiza o pilar do polimorfismo dentro da Programação Orientada a Objetos.

**Figura 1** - Polimorfismo



Antes de aplicarmos esse conceito ao conceito de Programação Orientada a Objetos, segue na figura abaixo o conceito extraído de <https://dicionario.priberam.org/polimorfismo>:

**Figura 2** - Conceito de polimorfismo



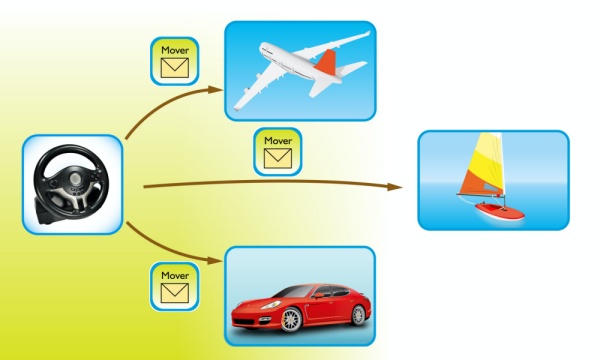
No contexto da POO, o polimorfismo permite que diferentes objetos respondam a uma mesma mensagem de formas diferentes (voar, navegar e correr).

## Exemplo

Um dono de uma fábrica de brinquedos solicitou que seus engenheiros criassem um mesmo controle remoto para todos os brinquedos de sua fábrica. A única restrição era que cada brinquedo atendesse aos comandos específicos definidos pelo controle.

O controle remoto teria vários botões, sendo que todos eles seriam úteis para todos os brinquedos. Assim, quando o usuário clicasse no botão mover, o controle enviaria o sinal MOVER para todos os brinquedos que estivessem no raio de dois metros. A figura abaixo ilustra tal situação:

**Figura 3** - Exemplo de polimorfismo



Assim, quando o brinquedo recebe o sinal MOVER, ele se move de acordo com a sua função. Para o avião, mover significa VOAR, para o barco significa NAVEGAR, e para o automóvel, CORRER. Observe que os brinquedos respondem ao mesmo sinal de formas diferentes. Temos aqui então um caso de polimorfismo.

<https://materialpublic.imd.ufrn.br/curso/disciplina/5/76/8/6>

Para saber mais entre no link acima.