PYTHON ACADEMY

INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS NO PYTHON

Guia completo de POO em Python: classes, objetos, herança, polimorfismo, encapsulamento, abstração. Casos reais (conta bancária, sistema escolar), quando usar OOP vs funcional.

PYTHONACADEMY.COM.BR

Crie ebooks profissionais incríveis em minutos com IA

Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

E deixe que nossa IA faça o trabalho pesado!

- Capas gerados por IA
- Infográficos feitos por IA
- Edite em Markdown em Tempo Real
- Adicione Banners Promocionais

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS

> ✅ ***Atualizado para Python 3.13*** *(Dezembro 2025) Conteúdo enriquecido com casos de uso reais e comparação OOP vs programação funcional.*

Olá Pythonista!

Programação Orientada a Objetos (POO) organiza código em **classes** e **objetos**, modelando entidades do mundo real. É ideal para sistemas complexos com estados mutáveis e relações entre entidades.

Neste guia, você vai aprender:

- ✅ **Pilares da OOP** - Abstração, Encapsulamento, Herança, Polimorfismo
- ✅ **Classes e Objetos** - Criação e uso
- ✅ **Casos reais** - Sistema bancário, escolar
- ✅ **OOP vs Funcional** - Quando usar cada paradigma

Python já nasceu sendo uma linguagem de programação **multi-paradigma**, isto é: é possível programar em Python de maneira Imperativa, Funcional e também no paradigma que será abordado nesse post, utilizando conceitos da **Programação Orientada a Objetos**!

Criar e usar Classes e Objetos é muito fácil em Python e esse post vai ter ajudar a se tornar um **especialista** no uso da Programação Orientada a Objetos em Python!

Então, **VAMOS NESSA**!

Programação Orientada a Objetos: Introdução

A **P**rogramação **O**rientada a **O**bjetos (**POO**) é um paradigma de programação baseado no conceito de **Classes** e **Objetos**.

Classes podem conter dados e código:

- **Dados** na forma de campos (também chamamos de **atributos** ou propriedades); e
- **Código**, na forma de procedimentos (frequentemente conhecido como **métodos**).

Uma importante característica dos objetos é que seus próprios métodos podem acessar e frequentemente modificar seus campos de dados: objetos mantém uma referência para si mesmo, o atributo `self` no Python.

Na POO, os programas são projetados a partir de objetos que interagem uns com os outros.

Esse paradigma se concentra nos objetos que os desenvolvedores desejam manipular, ao invés da lógica necessária para manipulá-los.

Essa abordagem de programação é adequada para programas grandes, complexos e ativamente atualizados ou mantidos.

Classes, Objetos, Métodos e Atributos

Esses conceitos são os pilares da Programação Orientada a Objetos então é muito importante que você os **DOMINE**:

- As **Classes** são tipos de dados definidos pelo desenvolvedor que atuam como um modelo para objetos. ***Pra não esquecer mais: Classes são fôrmas de bolo e bolos são objetos*** 😉
- **Objetos** são instâncias de uma Classe. Objetos podem modelar entidades do mundo real (Carro, Pessoa, Usuário) ou entidades abstratas (Temperatura, Umidade, Medição, Configuração).
- **Métodos** são funções definidas dentro de uma classe que descreve os comportamentos de um objeto. Em Python, o primeiro parâmetro dos métodos é sempre uma referência ao próprio objeto.
- Os **Atributos** são definidos na Classe e representam o estado de um objeto. Os objetos terão dados armazenados nos campos de atributos. Também existe o conceito de atributos de classe, mas veremos isso mais pra frente.

Princípios Básicos de POO

A programação orientada a objetos é baseada nos seguintes princípios:

Encapsulamento

Usamos esse princípio para juntar, ou **encapsular**, dados e comportamentos relacionados em entidades únicas, que chamamos de objetos.

Por exemplo, se quisermos modelar uma entidade do mundo real, por exemplo **Computador**.

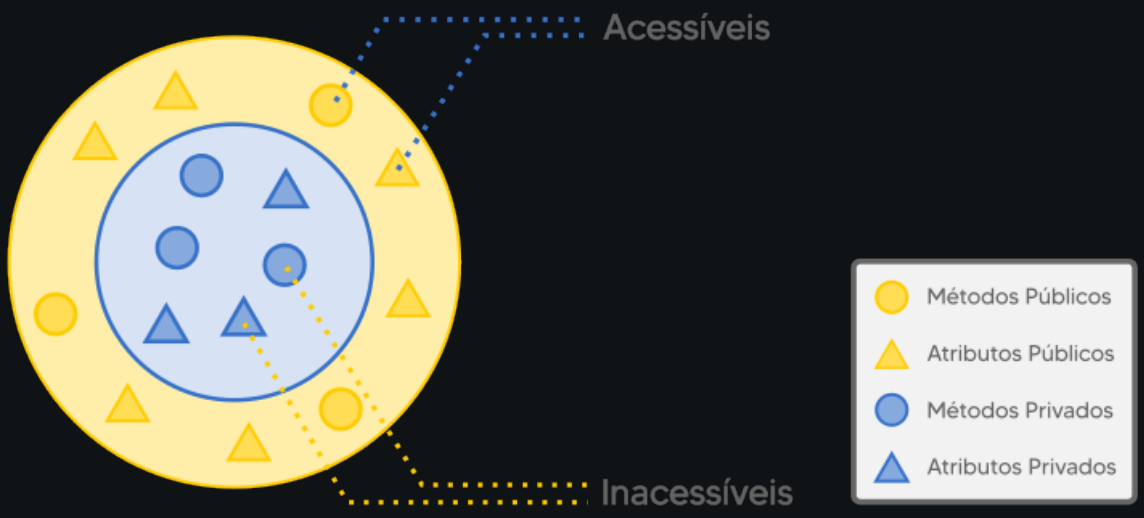
Encapsular é **agregar** todos os **atributos e comportamentos** referentes à essa Entidade dentro de sua Classe.

Dessa forma, o mundo exterior não precisa saber como um Computador liga e desliga, ou como ele realiza cálculos matemáticos!

Basta instanciar um objeto da Classe Computador, e utilizá-lo! 😉

O princípio do Encapsulamento também afirma que informações importantes devem ser contidas dentro do objeto de maneira privada e apenas informações selecionadas devem ser expostas publicamente.

Veja a imagem abaixo que exemplifica a relação entre atributos e métodos públicos e privados:



A implementação e o estado de cada objeto são mantidos de forma **privada** dentro da definição da Classe.

Outros objetos não têm acesso a esta classe ou autoridade para fazer alterações.

Eles só podem chamar uma lista de funções ou métodos **públicos**.

Essa característica de ocultação de dados fornece maior segurança ao programa e evita corrupção de dados não intencional.

Abstração

Pense em um Tocador de DVD.

Ele tem uma placa lógica bastante complexa com diversos circuitos, transistores, capacitores e etc do lado de dentro e apenas alguns botões do lado de fora.

Você apenas clica no botão de “Play” e não se importa com o que acontece lá dentro: o tocador apenas… **Toca**.

Ou seja, a complexidade foi “escondida” de você: isto é **Abstração** na prática!

O Tocador de DVD **abstraiu** toda a lógica de como tocar o DVD, expondo apenas botões de controle para o usuário.

O mesmo se aplica à Classes e Objetos: nós podemos esconder atributos e métodos do mundo exterior. E isso nos traz alguns benefícios!

Primeiro, a interface para utilização desses objetos é **muito mais simples**, basta saber quais “botões” utilizar.

Também reduz o que chamamos de “Impacto da mudança”, isto é: ao se alterar as propriedades internas da classes, nada será alterado no mundo exterior, já que a interface já foi definida e deve ser respeitada.

Herança

Herança é a característica da POO que possibilita a **reutilização de código comum** em uma relação de hierarquia entre Classes.

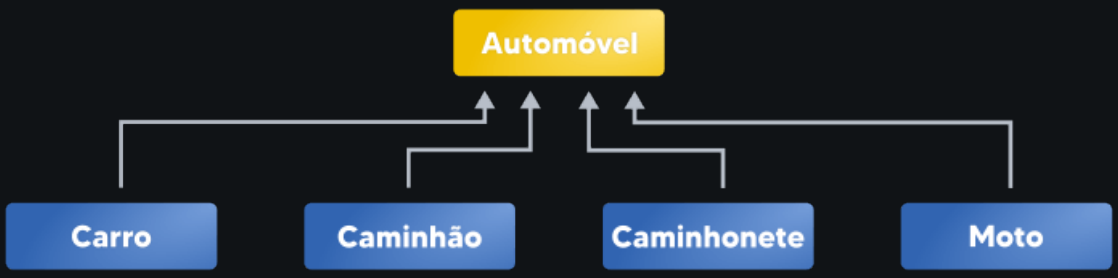
Vamos utilizar a entidade **Carro** como exemplo.

Agora imagine uma Caminhonete, um Caminhão e uma Moto.

Todos eles são **Automóveis**, correto? Todos possuem característica semelhantes, não é mesmo?

Podemos pensar que Automóveis aceleram, freiam, possuem mecanismo de acionamento de faróis, entre outros.

Uma relação de hierarquia entre as classes poderia ser pensada da seguinte forma:



Dessa forma podemos modelar os comportamentos semelhantes em uma Classe "pai" **Automóvel** que conterá os atributos e comportamentos comuns.

Através da Herança, as Classes filhas de **Automóvel** vão herdar esses atributos e comportamentos, sem precisar reescrevê-los reduzindo assim o tempo de desenvolvimento!

Polimorfismo

Quando utilizamos **Herança**, teremos Classes filhas utilizando código comum da Classe acima, ou Classe pai.

Ou seja, as Classes vão compartilhar atributos e comportamentos (herdados da Classe acima).

Assim, Objetos de Classes diferentes, terão métodos e atributos compartilhados que podem ter implementações diferentes, ou seja, um método pode possuir várias formas e atributos podem adquirir valores diferentes.

Daí o nome: **Poli** (muitas) **morfismo** (formas).

Para entendermos melhor, vamos utilizar novamente o exemplo da entidade **Carro** que herda de **Automóvel**.

Suponha agora que **Automóvel** possua a definição do método `acelerar()`.

Por conta do conceito de Polimorfismo, objetos da Classe **Moto** terão uma implementação do método `acelerar()` que será diferente da implementação desse métodos em instâncias da Classe **Carro**!

> 💡 *Estou construindo o **Ebookr.ai**, uma plataforma onde você cria ebooks profissionais com IA sobre qualquer assunto — do zero ao PDF pronto, com capas e infográficos gerados automaticamente. Dá uma olhada!*

Programação Orientada a Objetos no Python

Agora vamos finalmente juntar a Programação Orientada a Objetos com o Python!

Python possui palavras reservadas (*keywords*) para criarmos **Classes** e **Objetos**.

👉 Primeiro, temos a *keyword* `class` que utilizamos para criar uma classe.

👉 Também temos a *keyword* `self`, utilizada para guardar a referência ao próprio objeto.

Uma observação importante, caso você venha de outra linguagem de programação: Python não utiliza a *keyword* `new` para instanciar novos objetos!

Vamos logo para o código! Tudo ficará mais claro 😉

Vamos criar uma classe que representa uma entidade do tipo **Pessoa**!

Ela deve ter os seguintes campos:

- Nome como String;
- Idade como Inteiro;
- Altura como Decimal.

Também deve ter métodos para:

- Dizer “Olá”;
- Cozinhar;
- Andar.

Utilizando POO e Python, podemos modelar a entidade `Pessoa` da seguinte forma:

```python
class Pessoa:
  def __init__(self, nome: str, idade: int, altura: float):
    self.nome = nome
    self.idade = idade
    self.altura = altura

  def dizer_ola(self):
    print(f'Olá, meu nome é {self.nome}. Tenho {self.idade} '
          f'anos e minha altura é {self.altura}m.')

  def cozinhar(self, receita: str):
    print(f'Estou cozinhando um(a): {receita}')

  def andar(self, distancia: float):
    print(f'Saí para andar. Volto quando completar {distancia} metros')
```

Agora vamos explicar “tintim por tintim”:

- Temos a definição da Classe na primeira linha com `class Pessoa`. Isso diz ao Python que vamos criar a definição de uma nova classe.
- Em seguida, temos o método `__init__`. Ele é muito importante e é chamado de **Construtor**. Ele é chamado ao se instanciar objetos e é nele que geralmente setamos os atributos do objeto.
- Em seguida temos a definição dos métodos `dizer_ola()`, `cozinhar()` e `andar()`.
- Perceba que no método `dizer_ola()` referenciamos os atributos do próprio objeto com o argumento `self`: `self.nome`, `self.idade` e `self.altura`.

Agora veja como podemos instanciar e interagir com objetos dessa Classe:

```
# Instancia um objeto da Classe "Pessoa"
pessoa = Pessoa(nome='João', idade=25, altura=1.88)

# Chama os métodos de "Pessoa"
pessoa.dizer_ola()
pessoa.cozinhar('Spaghetti')
pessoa.andar(750.5)
```

Se lembra do **Construtor**?

Então, ele entrou em ação na linha 2 do código acima!

Quando escrevemos `pessoa = Pessoa()`, chamamos o método `__init__` da classe `Pessoa`, passando os parâmetros `nome`, `idade` e `altura`.

A saída dessas linhas de código será:

```
Olá, meu nome é João. Tenho 25 anos e minha altura é 1.88m.
Estou cozinhando um(a): Spaghetti
Saí para andar. Volto quando completar 750.5 metros
```

Casos de Uso Reais

1. Sistema de Contas Bancárias

```python
class ContaBancaria:
    def __init__(self, titular, saldo_inicial=0):
        self.titular = titular
        self.__saldo = saldo_inicial  # Encapsulamento (privado)

    def depositar(self, valor):
        if valor > 0:
            self.__saldo += valor
            return True
        return False

    def sacar(self, valor):
        if 0 < valor <= self.__saldo:
            self.__saldo -= valor
            return True
        return False

    def consultar_saldo(self):
        return self.__saldo

# Herança
class ContaCorrente(ContaBancaria):
    def __init__(self, titular, saldo_inicial=0, limite=1000):
        super().__init__(titular, saldo_inicial)
        self.limite = limite

    def sacar(self, valor):  # Polimorfismo
        if valor <= self.consultar_saldo() + self.limite:
            return super().sacar(valor)
        return False

conta = ContaCorrente("Alice", 500, limite=200)
conta.sacar(600)  # Pode sacar até 700 (saldo + limite)
```

2. Sistema Escolar com Herança

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

    def apresentar(self):
        return f"{self.nome}, {self.idade} anos"

class Aluno(Pessoa):
    def __init__(self, nome, idade, matricula):
        super().__init__(nome, idade)
        self.matricula = matricula
        self.notas = []

    def adicionar_nota(self, nota):
        self.notas.append(nota)

    def media(self):
        return sum(self.notas) / len(self.notas) if self.notas else 0

class Professor(Pessoa):
    def __init__(self, nome, idade, disciplina):
        super().__init__(nome, idade)
        self.disciplina = disciplina

aluno = Aluno("Bob", 16, "2025001")
aluno.adicionar_nota(8.5)
aluno.adicionar_nota(9.0)
print(f"Média: {aluno.media()}")  # 8.75
```

OOP vs Programação Funcional

Quando Usar OOP?

✅ **Use OOP quando:**

- Sistema tem **estados mutáveis** (conta bancária, carrinho)
- Entidades com **comportamentos próprios** (usuário, produto)
- Precisa **herança/polimorfismo** (veículos, formas geométricas)
- **Modelar mundo real** (pessoas, empresas, animais)
- Código **grande e complexo** (frameworks, APIs)

Quando Usar Funcional?

✅ **Use Funcional quando:**

- **Transformações de dados** (map, filter, reduce)
- **Funções puras** sem side effects
- **Pipelines** de processamento
- Scripts **simples** e diretos
- **Imutabilidade** é importante

Exemplo Comparativo

```python
# OOP: Estado mutável
class Carrinho:
    def __init__(self):
        self.items = []

    def adicionar(self, item):
        self.items.append(item)

    def total(self):
        return sum(item['preco'] for item in self.items)

carrinho = Carrinho()
carrinho.adicionar({'nome': 'Livro', 'preco': 30})

# Funcional: Imutável
def adicionar_item(carrinho, item):
    return carrinho + [item]  # Nova lista

def calcular_total(carrinho):
    return sum(item['preco'] for item in carrinho)

carrinho = []
carrinho = adicionar_item(carrinho, {'nome': 'Livro', 'preco': 30})
```

Conclusão

Neste guia de **Programação Orientada a Objetos**, você aprendeu:

✅ **Pilares** - Abstração, Encapsulamento, Herança, Polimorfismo ✅ **Classes e Objetos** - Estrutura e instâncias ✅ **Casos práticos** - Contas bancárias, sistema escolar ✅ **OOP vs Funcional** - Quando usar cada paradigma

Principais lições:

- OOP modela **entidades do mundo real**
- **Encapsulamento** protege dados internos (`__atributo`)
- **Herança** reutiliza código (DRY)
- **Polimorfismo** permite comportamentos diferentes
- Use OOP para **estados mutáveis** e Funcional para **transformações**

Próximos passos:

- Pratique Classes e Objetos
- Aprenda @property para getters/setters
- Explore Dataclasses
- Estude Design Patterns em Python

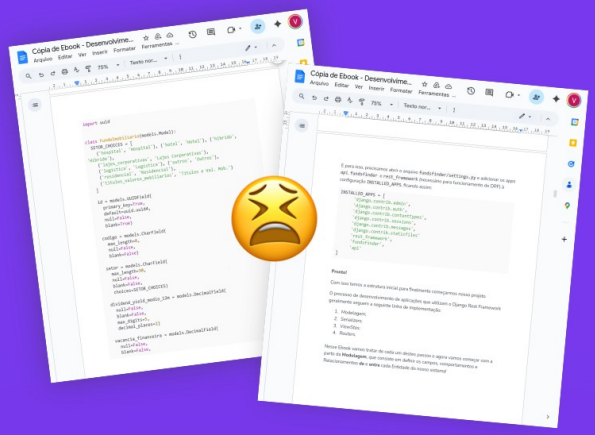
Este foi um post introdutório sobre a tão importante **Programação Orientada a Objetos**!

Se ficou com alguma dúvida, fique à vontade para deixar um comentário no box aqui embaixo! Será um prazer te responder! 😉

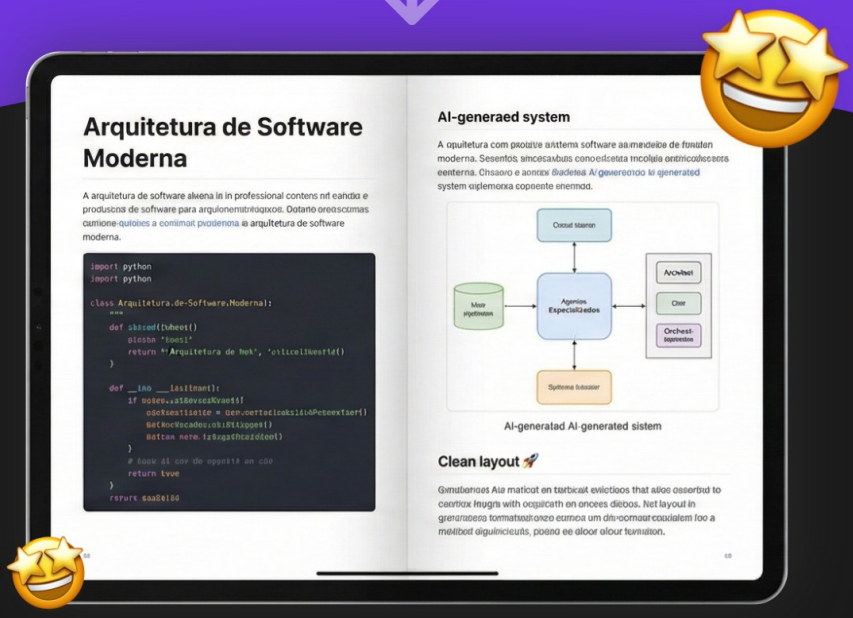
Não se esqueça de conferir!

Ebookr

Crie Ebooks profissionais em minutos com IA



Chega de formatar código no Google Docs ou Word



Capas gerados por IA

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS