Apostila de Programação Orientada a Objetos em Python

Versão completa (com exemplos, exercícios, projetos e apêndices)

Autor(a): Você + Assistente IA

Esta apostila foi organizada a partir de um material-base fornecido por você, e expandida com explicações, exemplos práticos e atividades para estudo guiado.

Sumário

1 Como usar esta apostila
2 1. Introdução à POO
3 2. Classes e Objetos
4 3. Atributos: instância e classe
5 4. Métodos: instância, classe e estático
6 5. Construtores e outros métodos especiais
7 6. Encapsulamento (público, protegido, privado)
8 7. Getters, Setters e @property
9 8. Herança (simples e múltipla) e MRO
10 9. super() na prática
11 10. Polimorfismo e Duck Typing
12 11. Composição x Herança
13 12. Classes Abstratas e Interfaces (abc)
14 13. Métodos mágicos (dunder) úteis
15 14. Iteradores, Iteráveis e Protocolos
16 15. Exceções e hierarquia de erros
17 16. dataclasses e slots
18 17. Tipagem estática (typing, Protocol)
19 18. Padrões de projeto (Strategy, Factory, Observer)
20 19. Boas práticas e princípios (SOLID, coesão, acoplamento)
21 20. Testes unitários em POO
22 21. Serialização e persistência simples
23 22. Projeto guiado 1: Sistema de Frotas (Veículos)
24 23. Projeto guiado 2: Banco simples (Contas)
25 24. Exercícios práticos (lista)
26 25. Gabarito selecionado
27 26. Glossário rápido de POO
28 27. Checklist de revisão
29 Apêndice A: Trechos do material-base
30 Apêndice B: Tabela de funções/builtins úteis
31 Referências e próximos passos

Como usar esta apostila

Cada capítulo apresenta teoria objetiva, seguida de exemplos comentados e exercícios. Execute os códigos, altere parâmetros e observe os efeitos. A prática é a chave para dominar POO.

Os exemplos usam Python 3 e seguem convenções do PEP 8. Quando fizer sentido, apontamos alternativas idiomáticas.

1. Introdução à POO

Programação Orientada a Objetos (POO) é um paradigma baseado em **objetos** que encapsulam dados (atributos) e comportamentos (métodos). No Python, a POO convive com outros estilos (funcional e estruturado), oferecendo flexibilidade para modelar problemas do mundo real.

Ideias centrais: abstração (esconder detalhes), encapsulamento (proteger dados), herança (reuso) e polimorfismo (interfaces comuns, comportamentos diferentes).

2. Classes e Objetos

Uma classe é um molde; um objeto é uma instância viva desse molde. Você define atributos e métodos dentro da classe para descrever estado e comportamento.

Em Python, tudo é objeto: até funções e classes. Isso facilita compor funcionalidades e passar comportamentos como valores.

```
class Veiculo:
    def movimentar(self):
        print("Sou um veículo e me desloco")

meu_veiculo = Veiculo()
meu_veiculo.movimentar()
```

3. Atributos: instância e classe

Atributos de instância pertencem a cada objeto; atributos de classe pertencem à classe e são compartilhados por instâncias.

Use atributos de classe para valores constantes ou contadores globais; evite usá-los para estado mutável específico de instância.

```
class Pedido:
    _contador = 0  # atributo de classe
    def __init__(self, cliente):
        Pedido._contador += 1
        self.id = Pedido._contador
        self.cliente = cliente
```

4. Métodos: instância, classe e estático

Métodos de instância recebem 'self'; métodos de classe recebem 'cls' e manipulam estado da classe; métodos estáticos são funções agrupadas logicamente na classe.

```
class Conversor:
    fator = 2.54

    def __init__(self, cm):
        self.cm = cm

    @classmethod
    def de_polegadas(cls, pol):
        return cls(pol * cls.fator)

    @staticmethod
    def eh_positivo(x):
        return x > 0
```

5. Construtores e outros métodos especiais

__init__ inicializa o objeto. __repr__ e __str__ ajudam no debug e apresentação. Implemente comparação (__eq__/__lt__) quando fizer sentido semanticamente.

```
class Moeda:
    def __init__(self, valor):
        self.valor = float(valor)

    def __repr__(self):
        return f"Moeda({self.valor:.2f})"

    def __str__(self):
        return f"R$ {self.valor:.2f}"

    def __eq__(self, other):
        return isinstance(other, Moeda) and self.valor == other.valor
```

6. Encapsulamento (público, protegido, privado)

Em Python, privacidade é por convenção: '_' indica 'protegido'; '__' aciona name-mangling (mais difícil de acessar acidentalmente).

Encapsule para proteger invariantes e controlar acesso; exponha somente o necessário.

```
class Conta:
    def __init__(self, titular, saldo):
        self._titular = titular       # 'protegido' por convenção
        self.__saldo = float(saldo)   # 'privado' com name-mangling

    def _pode_debitar(self, valor):   # método interno
        return valor <= self.__saldo
```

7. Getters, Setters e @property

Prefira @property para fornecer acesso controlado, mantendo uma API simples (atributos parecem públicos, mas têm validação interna).

```
class Produto:
    def __init__(self, nome, preco):
        self._nome = nome
        self._preco = 0.0
        self.preco = preco  # usa setter

    @property
    def preco(self):
        return self._preco

    @preco.setter
    def preco(self, valor):
        if valor < 0:
            raise ValueError("Preço não pode ser negativo")
        self._preco = float(valor)
```

8. Herança (simples e múltipla) e MRO

Herança permite reuso estrutural. A herança múltipla existe no Python; em caso de conflitos, a ordem de resolução (MRO) define prioridade.

```
class Logavel:
    def log(self, msg): print(f"[LOG] {msg}")

class Repositorio:
    def salvar(self): print("salvando...")

class RepoLog(Logavel, Repositorio):
    pass

print(RepoLog.__mro__)
```

9. super() na prática

Use super() para delegar à superclasse sem acoplar ao nome dela, respeitando o MRO. É crucial em herança múltipla para cooperar com outras classes na hierarquia.

```
class Veiculo:
    def __init__(self, fabricante, modelo):
        self._fabricante = fabricante
        self._modelo = modelo

class Aviao(Veiculo):
    def __init__(self, fabricante, modelo, categoria):
        super().__init__(fabricante, modelo)  # delega
        self._categoria = categoria
```

10. Polimorfismo e Duck Typing

Polimorfismo: objetos diferentes respondem a uma interface comum. Em Python, prática de 'if it quacks like a duck' (duck typing) é usual.

```
class Carro:
    def movimentar(self): print("Ando pelas ruas")

class Motocicleta:
    def movimentar(self): print("Corro muito!")

def mover(veiculo):
    veiculo.movimentar()

for v in (Carro(), Motocicleta()):
    mover(v)
```

11. Composição x Herança

Composição favorece reutilização flexível ao acoplar objetos internamente. Prefira herdar apenas quando há uma relação 'é-um' inequívoca e você precisa de polimorfismo.

```
class Motor:
    def ligar(self): print("vrummm")

class Carro:
    def __init__(self):
        self.motor = Motor()  # composição
    def dirigir(self):
        self.motor.ligar()
        print("Dirigindo...")
```

12. Classes Abstratas e Interfaces (abc)

abc.ABC e @abstractmethod permitem definir contratos. Subclasses devem implementar métodos abstratos.

```
from abc import ABC, abstractmethod

class Autenticavel(ABC):
    @abstractmethod
    def autenticar(self, senha): ...
```

13. Métodos mágicos (dunder) úteis

Implemente __len__, __iter__, __contains__, __enter__/__exit__ para tornar seus objetos idiomáticos.

```
class Colecao:
    def __init__(self, itens):
        self._itens = list(itens)
    def __len__(self): return len(self._itens)
    def __iter__(self): return iter(self._itens)
    def __contains__(self, x): return x in self._itens
```

14. Iteradores, Iteráveis e Protocolos

Para criar iteradores personalizados, implemente __iter__ e __next__. Com typing.Protocol (PEP 544), você descreve interfaces estruturais (duck typing com tipos).

```
class Contador:
    def __init__(self, maximo):
        self.atual = 0
        self.maximo = maximo
    def __iter__(self): return self
    def __next__(self):
        if self.atual >= self.maximo:
            raise StopIteration
        self.atual += 1
        return self.atual
```

15. Exceções e hierarquia de erros

Crie exceções específicas para domínios diferentes. Forneça mensagens claras e documente causas comuns.

```
class SaldoInsuficiente(Exception):
    pass

def debitar(saldo, valor):
    if valor > saldo:
        raise SaldoInsuficiente("Saldo insuficiente")
    return saldo - valor
```

16. dataclasses e slots

dataclasses reduzem boilerplate, gerando __init__/__repr__/__eq__. Com slots=True (Python 3.10+), há economia de memória e proibição dinâmica de atributos desconhecidos.

```
from dataclasses import dataclass

@dataclass(slots=True)
class Ponto:
    x: float
    y: float
```

17. Tipagem estática (typing, Protocol)

Anotações de tipo ajudam ferramentas (mypy/pyright) a detectar erros cedo. Protocol descreve interfaces por comportamento, e @runtime_checkable permite isinstance/issubclass em tempo de execução.

```
from typing import Protocol, runtime_checkable

@runtime_checkable
class Exportavel(Protocol):
    def exportar(self) -> str: ...

class CSV(Exportavel):
    def exportar(self) -> str: return "a,b,c"
```

18. Padrões de projeto (Strategy, Factory, Observer)

Padrões catalisam reutilização e clareza. Strategy encapsula algoritmos; Factory centraliza criação; Observer notifica interessados.

```
# Strategy
class RegraPreco:
    def calcular(self, valor): return valor

class Desconto10(RegraPreco):
    def calcular(self, valor): return valor*0.9

class Pedido:
    def __init__(self, total, regra=RegraPreco()):
        self.total = total
        self.regra = regra
    def total_final(self):
        return self.regra.calcular(self.total)
```

19. Boas práticas e princípios (SOLID, coesão, acoplamento)

Mantenha classes pequenas e focadas; injete dependências; programe para interfaces; favoreça imutabilidade quando possível.

20. Testes unitários em POO

Testes garantem que alterações não quebrem contratos. Use dublês (stubs/mocks) para isolar unidades.

```
import unittest

class TestMoeda(unittest.TestCase):
    def test_repr(self):
        self.assertEqual(repr(Moeda(2)), "Moeda(2.00)")
```

21. Serialização e persistência simples

Objetos podem ser serializados em JSON (quando compostos de tipos básicos) ou com pickle (cuidado com segurança).

```
import json

class Cliente:
    def __init__(self, nome):
        self.nome = nome

c = Cliente("Ana")
print(json.dumps({"nome": c.nome}))
```

22. Projeto guiado 1: Sistema de Frotas (Veículos)

Projeto inspirado no material-base: modele Veiculo, Carro, Motocicleta e Aviao. Implemente polimorfismo em movimentar(), use super() no Aviao para estender __init__, e proteja atributos com @property.

Requisitos: cadastro, listagem, busca por fabricante/modelo, e relatório por categoria (aviões).

23. Projeto guiado 2: Banco simples (Contas)

Modele Conta, ContaCorrente e ContaPoupanca com herança. Implemente exceções para saldo insuficiente, taxas diferentes por tipo e extrato textual. Demonstre composição: Conta usa uma classe Extrato para registrar operações.

Exercícios práticos

Os exercícios a seguir foram pensados em progressão. Resolva no seu editor de código e rode os testes manualmente. Dica: prefira compor classes antes de herdar, a menos que você precise explicitamente de polimorfismo ou reutilização estrutural.

01. Crie uma classe simples com __init__, __repr__ e um método de instância.
02. Crie uma classe simples com __init__, __repr__ e um método de instância.
03. Crie uma classe simples com __init__, __repr__ e um método de instância.
04. Crie uma classe simples com __init__, __repr__ e um método de instância.
05. Crie uma classe simples com __init__, __repr__ e um método de instância.
06. Crie uma classe simples com __init__, __repr__ e um método de instância.
07. Crie uma classe simples com __init__, __repr__ e um método de instância.
08. Crie uma classe simples com __init__, __repr__ e um método de instância.
09. Crie uma classe simples com __init__, __repr__ e um método de instância.
10. Crie uma classe simples com __init__, __repr__ e um método de instância.
11. Modele herança e sobrescreva um método. Mostre o polimorfismo em ação.
12. Modele herança e sobrescreva um método. Mostre o polimorfismo em ação.
13. Modele herança e sobrescreva um método. Mostre o polimorfismo em ação.
14. Modele herança e sobrescreva um método. Mostre o polimorfismo em ação.
15. Modele herança e sobrescreva um método. Mostre o polimorfismo em ação.
16. Modele herança e sobrescreva um método. Mostre o polimorfismo em ação.
17. Modele herança e sobrescreva um método. Mostre o polimorfismo em ação.
18. Modele herança e sobrescreva um método. Mostre o polimorfismo em ação.
19. Modele herança e sobrescreva um método. Mostre o polimorfismo em ação.
20. Modele herança e sobrescreva um método. Mostre o polimorfismo em ação.
21. Implemente @property com validação, e lance exceções personalizadas quando necessário.
22. Implemente @property com validação, e lance exceções personalizadas quando necessário.
23. Implemente @property com validação, e lance exceções personalizadas quando necessário.
24. Implemente @property com validação, e lance exceções personalizadas quando necessário.
25. Implemente @property com validação, e lance exceções personalizadas quando necessário.
26. Implemente @property com validação, e lance exceções personalizadas quando necessário.
27. Implemente @property com validação, e lance exceções personalizadas quando necessário.
28. Implemente @property com validação, e lance exceções personalizadas quando necessário.
29. Implemente @property com validação, e lance exceções personalizadas quando necessário.
30. Implemente @property com validação, e lance exceções personalizadas quando necessário.
31. Implemente um iterável/iterador e escreva testes unitários para ele.
32. Implemente um iterável/iterador e escreva testes unitários para ele.

33. Implemente um iterável/iterador e escreva testes unitários para ele.
34. Implemente um iterável/iterador e escreva testes unitários para ele.
35. Implemente um iterável/iterador e escreva testes unitários para ele.
36. Implemente um iterável/iterador e escreva testes unitários para ele.
37. Implemente um iterável/iterador e escreva testes unitários para ele.
38. Implemente um iterável/iterador e escreva testes unitários para ele.
39. Implemente um iterável/iterador e escreva testes unitários para ele.
40. Implemente um iterável/iterador e escreva testes unitários para ele.
41. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
42. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
43. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
44. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
45. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
46. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
47. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
48. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
49. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
50. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
51. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
52. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
53. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
54. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
55. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
56. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
57. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
58. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
59. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.
60. Aplique um padrão de projeto (Strategy/Factory/Observer) em uma mini-funcionalidade.

Gabarito selecionado (exercícios)

A seguir, disponibilizamos resoluções comentadas para uma seleção de exercícios. Use-as para conferir ideias, não como única resposta possível.

Exercício 01

```
class Produto:
    def __init__(self, nome, preco):
        self._nome = nome
        self._preco = float(preco)

    def aplicar_desconto(self, pct):
        self._preco *= (1 - pct/100)

    def __repr__(self):
        return f"Produto(nome={self._nome!r}, preco={self._preco:.2f})"
```

Exercício 02

```
class Produto:
    def __init__(self, nome, preco):
        self._nome = nome
        self._preco = float(preco)

    def aplicar_desconto(self, pct):
        self._preco *= (1 - pct/100)

    def __repr__(self):
        return f"Produto(nome={self._nome!r}, preco={self._preco:.2f})"
```

Exercício 03

```
class Produto:
    def __init__(self, nome, preco):
        self._nome = nome
        self._preco = float(preco)

    def aplicar_desconto(self, pct):
        self._preco *= (1 - pct/100)

    def __repr__(self):
        return f"Produto(nome={self._nome!r}, preco={self._preco:.2f})"
```

Exercício 04

```
class Produto:
    def __init__(self, nome, preco):
        self._nome = nome
        self._preco = float(preco)

    def aplicar_desconto(self, pct):
        self._preco *= (1 - pct/100)

    def __repr__(self):
        return f"Produto(nome={self._nome!r}, preco={self._preco:.2f})"
```

Exercício 05

```
class Animal:
```

```
    def falar(self): return "???"

class Gato(Animal):
    def falar(self): return "miau"

class Cachorro(Animal):
    def falar(self): return "au au"

def coro(animals):
    for a in animals:
        print(a.falar())

coro([Gato(), Cachorro(), Gato()])
```

Exercício 06

```
class Animal:
    def falar(self): return "???"

class Gato(Animal):
    def falar(self): return "miau"

class Cachorro(Animal):
    def falar(self): return "au au"

def coro(animals):
    for a in animals:
        print(a.falar())

coro([Gato(), Cachorro(), Gato()])
```

Exercício 07

```
class Animal:
    def falar(self): return "???"

class Gato(Animal):
    def falar(self): return "miau"

class Cachorro(Animal):
    def falar(self): return "au au"

def coro(animals):
    for a in animals:
        print(a.falar())

coro([Gato(), Cachorro(), Gato()])
```

Exercício 08

```
class Animal:
    def falar(self): return "???"

class Gato(Animal):
    def falar(self): return "miau"

class Cachorro(Animal):
    def falar(self): return "au au"

def coro(animals):
```

```
    for a in animals:
        print(a.falar())

coro([Gato(), Cachorro(), Gato()])
```

Exercício 09

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 10

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 11

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
```

```
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 12

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 13

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 14

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 15

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 16

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso
```

```
class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 17

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 18

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 19

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
```

```
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

Exercício 20

```
from abc import ABC, abstractmethod

class RegraFrete(ABC):
    @abstractmethod
    def calcular(self, peso): ...

class FreteFixo(RegraFrete):
    def calcular(self, peso): return 20.0

class FretePorPeso(RegraFrete):
    def calcular(self, peso): return 12.0 + 3.0*peso

class CalculadoraFrete:
    def __init__(self, regra: RegraFrete):
        self.regra = regra
    def calcular(self, peso):
        return self.regra.calcular(peso)

calc = CalculadoraFrete(FretePorPeso())
print(calc.calcular(5))
```

26. Glossário rápido de POO

Classe: molde que define atributos e métodos.

Objeto: instância concreta de uma classe.

Encapsulamento: ocultar detalhes internos e expor uma interface.

Herança: reutilizar e especializar comportamentos.

Polimorfismo: mesma interface, múltiplas implementações.

Composição: montar objetos por meio de outros objetos.

27. Checklist de revisão

■ Minha classe tem uma responsabilidade clara? ■ A API pública está coesa e minimalista? ■ Estou usando @property para validar estado? ■ Preciso mesmo de herança ou composição resolveria melhor? ■ Testes cobrem casos de erro e de sucesso? ■ Docstrings e __repr__ ajudam no debug?

Apêndice A: Trechos do material-base

Classes, Objetos, Métodos, Atributos, Herança - POO em Python
7:03:32
Python vamos tratar agora de forma introdutória de um assunto bem interessante que são as classes e
7:03:38
objetos em Python mas precisamente orientação a objetos básica em Python
7:03:44
Vamos criar um arquiv para isto que eu vou chamar de
7:03:49
p.py Poo programação orientada a objetos que que é esse negócio de orientação a
7:03:55
objetos orientação a
7:04:01
objetos isso aqui é um paradigma de programação que é um paradigma de
7:04:07
programação é um estilo uma forma de você programar utilizando determinadas
7:04:13
estruturas tipos de dados e funcionalidades o Python é uma linguagem multiparadigma significando qu
7:04:19
suporta vários paradigmas distintos paradigma estruturado que é o que a gente vem usando na maior p
7:04:26
vídeos paradigma funcional que a gente também já utilizou e o paradigma orientado objetos que na ve
7:04:32
também já utilizou porque já usamos muitos métodos da linguagem os métodos fazem parte da programaç
7:04:38
objetos a ideia aqui é a gente trabalhar com um conceito de classes e
7:04:46
objetos as classes são modelos abstratos que vão apresentar itens do mundo real
7:04:53
dentro do software e os objetos são as ocorrências dessas classes são essas classes carregadas na m
7:04:59
efetivamente existindo para valer então a grande

ideia aqui é a seguinte como é que eu represento dentro de um software
7:05:06
uma entidade do mundo real quando eu preciso realizar algum tipo de programação como é que eu repre
7:05:11
pessoa como é que eu represento um animal uma conta corrente dentro do software para isso o paradig
7:05:17
orientação objetos é bastante útil e aqui a gente vai ver uma introdução a assunto porque ele é mui
7:05:24
verdade o Python não é uma linguagem com suporte 100% a orientação a objetos é uma linguagem multip
7:05:30
citei mas é importante conhecer alguns conceitos e saber como criar uma classe instanciar um objeto
7:05:37
porque isso sim pode ser bastante útil na hora de criar os seus scripts vamos lá então para demonst
7:05:44
eu vou criar uma classe usando a palavra-chave Class eu vou criar uma
7:05:49
classe para modelar vamos supor V veículos então eu quero trabalhar num programa que consegue opera
7:05:56
veículos então eu vou chamar essa classe de veículo e a ideia é sempre que as
7:06:02
classes tenham letra maiúscula por padrão então a gente escreve classe veículo dois pontos e aí a g
7:06:09
a programar essa classe que que a gente vai colocar dentro dessa classe ela vai ter basicamente dua
7:06:16

atributos e métodos os atributos são propriedades da classe Ares meio de
7

:06:22
longe a variáveis ou seja estruturas que guardam valores e os métodos são as
7:06:29
funcionalidades ou ações que a classe pode realizar ou sofrer são similares a
7:06:34
funções por exemplo todo veículo se movimenta então a gente pode criar uma ação de movimentar-se pa
7:06:42
E para isso a gente Define um método que eu vou chamar de movimentar você pode
7:06:47
chamar do que você quiser o método Claro e aí com os dentro dos parênteses a gente vai
7:06:54
colocar a palavra self e essa palavrinha aqui ela é importante porque ela tá aqui
7:07:01
para dizer que quando eu criar um objeto baseado nessa classe esse movimento se refere aquele objet
7:07:08
a outro objeto derivado da mesma classe porque a gente cria a classe para ser uma espécie de modelo
7:07:14
planta e essa classe pode ser usada para criar vários objetos diferentes ou seja
7:07:19
eu programo uma vez essa classe mas na na minha aplicação eu posso usá-la diversas vezes para criar
7:07:26
veículos diferentes Então essa é a grande sacada da orientação objetos porque ela também permite re
7:07:32
código então o meu veículo ele vai se movimentar eu vou colocar dois pontos e Vou definir essa funç
7:07:39
dentro de uma classe A gente chama de método aqui eu vou fazer de forma bem simples eu vou colocar
7:07:45
mensagenz

inha do tipo sou um veículo e
7:07:50
me looco só para demonstrar a funcionalidade então este método
7:07:57
movimentar imprime na tela a frase sou um veículo e me desloco existe um tipo
7:08:03
especial de método que geralmente a gente cria para as classes que é o método que aqui no P A gente
7:08:08
método init e que em outras linguagens de programação a gente vai chamar de método Construtor que é
7:08:14
inicializa a classe Ou seja quando a gente criar um objeto baseado nessa classe quando a gente fori
7:08:20
ter um veículo na aplicação a gente vai fornecer alguns dados alguns parâmetros para que esse veícu
7:08:27
com algumas informações então Vamos definir o método init especial esse
7:08:32
método é definido assim und und init und und e aí dentro dos parênteses
7:08:41
a gente vai colocar algumas informações a gente sempre vai colocar o primeiro parâmetro self que é
7:08:47
se refere ao próprio objeto quando ele for criado na na prática não precisa ser a palavra self pode
7:08:53
palavra qualquer que você queira usar desde que seja a primeira palavra eu vou usar self porque é p
7:09:00
vírgula e agora a gente vai colocar os argumentos ou parâmetros que vão inicializar efetivamente o
7:09:07

o veículo por exemplo ele tem um fabricante e um modelo então eu

posso dizer que para inicializar esse objeto
7:09:13
eu vou fornecer dados de fabricante e de modelo então são os três
7:09:19
parâmetros que vai ter pode ter quantos parâmetros você precisar dois pontos e agora eu vou program
7:09:27
e agora eu vou criar um atributo interno Esse atributo é uma espécie de variável interna da classe
7:09:33
valor E para isso a gente usa de novo a palavra self a mesma palavrinha que a gente usou
7:09:39
aqui em cima e esse self ponto a gente vai dar um nome pro atributo interno
7:09:44
pode ser fabricante fabricante vai receber o que for passado
7:09:50
para esse este argumento fabricante aqui de fora então a gente escreve simplesmente fabricante de r
7:09:57
precisa ser o mesmo nome tá pode ser um nome diferente o importante é esse fabricante aqui é o é o
7:10:03
sendo passado e esse aqui é o atributo interno da classe que vai guardar o valor efetivamente é mui
7:10:09
função né Poderia ter uma variável guardando o valor que foi passado como como argumento e vou faze
7:10:16
self pro modelo então self ponto modelo
7:10:21
recebe modelo a gente também pode criar eh outros atributos aqui dentro internos
7:10:27
que não necessariamente são passados na hora da inicialização mas que já ficam disponíveis na memór
7:10:34
veículo pode ter um núm

ero de registro dependendo do tipo do veículo então a gente pode criar um self ponto número de
7:10:42
registro como esse número de registro não vai receber um valor eh fornecido
7:10:48
externamente na hora de de instanciar eh essa essa Classe A gente tem que atribuir um valor para el
7:10:54
aqui a gente vai atribuir o valor n para dizer ó não tem nada por enquanto mas depois a gente vai c
7:11:01
fornecer esse número de registro beleza Esse é o básico do básico de uma classe então aqui embaixo
7:11:08
seguinte eu vou criar o meu programa principal if
7:11:15
name igual M beleza e aqui agora eu vou
7:11:20
criar um objeto baseado nessa classe então a classe ficou criadinha aqui ela é um modelo apenas tá
7:11:27
objeto que significa pega esta classe e Monte na memória do computador esta estrutura especial aqui
7:11:35
trabalhar com veículos eu vou chamar isso aqui de meu veículo esse aqui é o
7:11:41
nome do objeto igual e para instanciar instanciar significa tornar a classe que
7:11:47
é um modelo real na memória do computador então a gente chama o nome da
7:11:53
classe veículo e dentro dos parênteses a gente passa os parâmetros necessários
7:12:00

que inclusive o próprio Idea já me disse fabricante modelo eu tenho que passar então eu vou passar
7:12:06
supor GM e o model

o vamos supor
7:12:12
cadlac Escalade Isso aqui vai criar um objeto na memória com essas duas informações já
7:12:19
dentro dos atributos corres entes E aí após criar o objeto eu posso invocar
7:12:24
seus métodos por exemplo ele tem o método movimentar não tem então meu veículo ponto já temos aqui
7:12:32
disponível no Intel sense movimentar então movimentar com os parênteses como
7:12:38
esse método não recebe nada além do selfie eu não preciso digitar nada aqui dentro é só executar o
7:12:44
vamos executar para ver o que aparece aqui a princípio tá lá apareceu a frase sou um veículo e me d
7:12:51
justamente o que a gente programou Aqui dentro deste método movimentar Olha só então quando eu invo
7:12:57
esse código aqui foi executado certo bem tranquilo uma outra coisa que a gente pode fazer é acessar
7:13:04
internos então por exemplo eu posso dar um print e aqui no print colocar meu
7:13:11
veículo ponto e puxar o fabricante ou modelo que já aparecem aqui inclusive
7:13:16
Então e o número do registro também aparece mas não tem nada gravado lá dentro então eu vou chamar
7:13:22
que é um atributo por isso não vai ter parênteses para ele especificamente tá ele é um atributo é c
7:13:28
variável E aí se eu executo esse código Olha só vai aparecer aqui a palavrinha
7:13:34
GM depois ali do

su veículo e me desloco eu tenho aqui GM que é o nome que eu
7:13:39
passei na hora que eu criei e a Instância dessa classe esse objeto meu veículo passei GM aqui tá de
7:13:46
fabricante agora isso aqui não é o ideal a gente conseguia acessar esses valores
7:13:52
internos da classe não é o ideal por conta de questões de segurança por exemplo e de funcionalidade
7:13:59
que esses atributos aqui só sejam acessíveis por meio de métodos especiais no momento esses atribut
7:14:06
públicos públicos significa eu consigo acessá-lo diretamente usando esse formato aqui nome do objet
7:14:14
do atributo Mas eu posso transformar esses atributos em atributos privados que não são acessíveis d
7:14:21
isso o que a gente faz é o seguinte aqui na hora de criar o atributo logo depois
7:14:27
do selfie ponto e antes do nome do atributo a gente coloca dois undor und
7:14:33
und e a gente vai fazer a mesma coisa pro modelo und und e a mesma coisa pro
7:14:40
número do registro Pronto agora os três atributos são privados ou seja eles não
7:14:46
são acessíveis mais diretamente isso aqui implementa um esquema eh de
7:14:51

orientação objeto chamado de encapsulamento o dado está encapsulado aqui dentro e você não consegue
7:14:57
enxergá-lo diretamente e o que acontece se eu rodar agora o programa Olha só vou rodar e ele vai di

le
7:15:04
vai dar um erro para mim erro de atributo então ele executou o primeiro método beleza sou um veícul
7:15:10
desloco mas na hora de tentar acessar esse atributo ele vai dizer o objeto veículo não tem um atrib
7:15:17
erro de atributo Mas e se eu colocasse aqui embaixo o and und aí nesse caso ele
7:15:23
vai continuar dizendo que não tem o atributo olha só então você achou que i aparecer o valor não ap
7:15:31
und é para dizer o atributo é privado ele não está acessível fora da classe ou
7:15:36
diretamente fora da classe bom quer dizer que eu não consigo mais usar o fabricante ou o modelo não
7:15:43
que funciona a gente consegue só que a gente vai ter que criar um método especial para isso esse mé
7:15:49
ele existe tem nome Inclusive a gente vai criar para poder acessar um elemento
7:15:57
a gente usa um método especial chamado de getter então o que que é um getter é um método especial u
7:16:03
permite acessar os atributos de dentro da classe ou acessar outros elementos dentro da classe Então
7:16:10
um getter eu vou definir esse getter que é uma função um método eu vou chamar de
7:16:16
get fabricante modelo por exemplo vamos supor que eu queira é só fabricante e o
7:16:21
modelo aqui e ele vai sempre receber o selfie para poder fazer a referência ao próprio objeto tá do

pontos e aí aqui
7:16:29
dentro a gente coloca a informação que a gente precisa por exemplo vamos supor que eu queira só imp
7:16:35
do fabricante e seu modelo então eu boto um print e aqui a gente coloca uma mensagem do tipo
7:16:43
modelo self ponto esse aqui é o modelo
7:16:48
beleza e o facante será o self P fabricante que tá
7:16:57
aqui beleza agora eu vou conseguir ter acesso a essas informações aqui certinho então deixa eu colo
7:17:03
de linha e a gente vai testar de novo só que agora não preciso mais desse print aqui embaixo pra ge
7:17:11
o que a gente vai fazer é simplesmente executar esse método Ou seja eu tiro
7:17:17
esse print chamo meu veículo ponto e aí
7:17:22
get Fabri modelo com os parênteses porque isso é um método vamos ver o que aparece agora vou rodar
7:17:29
sou um veículo e me desloco modelo cadlac Escalade fabricante GM perfeito
7:17:35
agora a gente conseguiu acessar os atributos internos só que usando um método Qual que é a vantagem
7:17:40
gente pode programar esse método de forma que os dados sejam acessíveis somente da forma que eu des
7:17:47

preservando assim a integridade desses dados tá isso importante então recomendo
7:17:52
sempree utilizar atributos privados com o und und e criar métodos getter para
7:17:57
você acessar esses atributos dentro de uma cla

sse e o número do registro número do registro a gente tem que gravar eu
7:18:04
tem que fornecer esse dado só que ele também é privado Então também não vou conseguir acessá-lo dir
7:18:11
pra gente poder acessar esse esse atributo interno e gravar um dado nele eu vou precisar de um outr
7:18:19
Setter o Setter é um método que permite gravar um dado dentro de um elemento na
7:18:25
dentro da classe dentro do objeto no caso Então a gente vai definir um set número do
7:18:32
registro que vai receber e self e aqui a gente precisa dizer que ele vai ter um
7:18:38
um atributo registro também porque eu tô fornecendo um dado para ele externamente tá E aí eu vou di
7:18:45
self ponto e eu vou procurar o número do registro tá aqui vai receber o
7:18:54
registro que é o nome desse desse atributo aqui o valor interno vai ficar
7:19:00
armazenado aqui então quando eu executar esse método passando o valor do registro o número de regis
7:19:06
vai vai ficar guardado aqui e se eu quiser ler esse valor posteriormente eu posso criar um getter p
7:19:12
então Def get num registro esse aqui recebe só o self
7:19:19
porque não vai modificar nada e aí a gente pode dizer o seguinte vamos supor eu não quero imprimir
7:19:25
retornar esse valor para ser usado em outro processo Então em vez

de escrever um print eu boto um return e mando
7:19:31
retornar o self ponto e o número do registro que tá gravado lá dentro
7:19:37
Prontinho agora eu tenho um getter pro número do registro e um Setter para ele também e agora a ger
7:19:43
aqui aqui embaixo depois de pegar o fabricante modelo eu vou fazer o seguinte meu veículo ponto S r
7:19:51
registro e vamos passar um número de registro vamos supor que o esse meu veículo tem o registro um
7:19:57
qualquer aleatório 4903 1 2 tracinho um pronto certo então esse valor vai ficar
7:20:03
gravado dentro daquele atributo aí eu consigo depois acessar isso aqui por exemplo
7:20:09
print e aí eu vou dizer o seguinte registro dois pontos e aqui eu vou
7:20:16
colocar meu veículo ponto get num registro os parênteses porque isso aqui
7:20:24
se trata de um método beleza posso até colocar uma quebrinha de linha depois então aqui eu vou eu v
7:20:30
valor para esse para esse atributo interno e depois eu acesso esse valor utilizando o getter Então
7:20:36
gravar e o getter para obter o valor quando eu executo esse código ele vai mostrar registro 490
7:20:43

320 a gente conseguiu gravar o valor lá dentro e depois fazer a leitura deste valor tranquilo aqui
7:20:52
valor que eu passei Ah o que ele fez aqui foi fazer um cálculo ele fez isso menos isso e gravou o v

r lá dentro
7:20:58
Então nesse caso como tem esse tracinho aqui eu preciso dizer que é uma string
7:21:03
tá aí quando eu executar Aí sim vem o registro direitinho então aqui a gente tem o tipo de dado faz
7:21:11
importante mas tá funcionando tá tá funcionando tranquilamente tanto o getter quanto o Setter pro r
7:21:17
então beleza A gente pode dizer que a nossa classe aqui tá pronta Por que a gente tem para fazer ac
7:21:24
tem outras coisas que a gente pode ver e um exemplo é o conceito de herança Qual
7:21:29
é a ideia da herança A ideia é a seguinte essa minha classe ela implementa um veículo mas um veícul
7:21:35
pode ser qualquer coisa na verdade não necessariamente um carro uma motocicleta é um veículo um cam
7:21:41
avião é um veículo uma espaçonave é um veículo então a gente pode ter categorias diferentes de veíc
7:21:46
um deles vai ter dados distintos vai ter métodos distintos por exemplo um avião
7:21:52
voa um carro anda uma bicicleta não necessariamente vai ter um número de registro então para cada t
7:21:58
a gente vai ter atributos ou métodos distintos E aí que que a gente tem que
7:22:04
fazer para poder trabalhar de forma adequada com esses casos a gente vai criar classes específicas
7:22:10
deles só que essas classes que a gente vai criar elas elas têm muito em

comum
7:22:15
com os veículos porque são veículos então eu não vou precisar na verdade
7:22:20
definir tudo isso aqui de novo para cada classe que eu criar que seja um veículo o que eu vou fazer
7:22:27
chamado de herança no qual eu reaproveito o que já foi codificado numa
7:22:32
classe para criar uma outra e depois só modifico aquilo que realmente for diferente de uma para out
7:22:38
gente implementa a herança bom vamos criar uma outra classe aqui vou criar uma classe carro por exe
7:22:45
falar especificamente sobre carros então fora dessa classe fora ela eu vou criar
7:22:51
uma nova classe que eu vou chamar de carro e aqui dentro dos parênteses a
7:22:56
gente vai usar parênteses agora a gente vai escrever o nome da classe da qual ela vai herdar as car
7:23:02
caso é a classe veículo dois pontos e agora a gente codifica essa classe Então esse essa
7:23:09
palavra veículo dentro dos parênteses tá dizendo Olha a classe carro é um tipo de veículo então tud
7:23:14
dentro da classe veículo também tá disponível aqui então por exemplo eu não vou criar um método ini
7:23:20
essa classe eu só vou deixar um comentário dizendo que o método que o
7:23:25

método init será herdado pronto não preciso programar
7:23:33
esse método mas eu vou criar um método movimentar distinto para

esse carro então eu vou criar um novo método
7:23:38
movimentar com o mesmo nome do método anterior isso é importante senão você vai acabar com dois mét
7:23:44
então eu vou criar um novo método movimentar e esse método aqui é não posso esquecer do self ele va
7:23:51
implementar uma mensagem diferente por exemplo print a gente pode escrever sou um carro
7:23:58
e ando pelas ruas então quando eu criar um objeto do tipo carro e invocar o
7:24:05
método movimentar o que vai aparecer é isso aqui mas o resto é igual a da classe veículo vamos test
7:24:13
aqui no meu programa principal vou pular uma linha e vou criar um objeto carro vou chamar ele de me
7:24:21
Então o meu carro é o objeto da classe carro e eu preciso passar dois valores
7:24:27
para ele o fabricante e o modelo porque isso tá tá codificado na minha classe veículo e o carro her
7:24:34
Então vamos colocar Um fabricante por exemplo Volkswagen e o modelo Polo por exemplo
7:24:44
beleza criei meu carro aí se eu chamar meu carro ponto movimentar ele vai exibir a
7:24:52
mensagem de movimento do carro e eu também posso chamar meu carro ponto e o
7:24:57
método getter fabricante modelo vamos testar Então vou executar o código tá lá
7:25:03
sou um carro e ando pelas ruas modelo Polo fabricante Volkswagen Então temos um objeto diferente ac
7:25:09
comentar es

se pedaço de cima do meu veículo para não atrapalhar os nossos testes Então tá aí meu carro tá cria
7:25:16
eu posso criar quantos objetos eu quiser usando a mesma classe por exemplo eu vou criar um segundo
7:25:22
seu carro agora também da classe carro e recebendo
7:25:27
outras informações por exemplo fabricante Audi e é Volkswagen também certo
7:25:32
e modelo pode ser um A5
7:25:38
Sportback beleza e a a gente pode chamar o seu carro ponto movimentar que é o
7:25:44
mesmo método e também posso chamar o seu carro Modelo E aí quando eu executo esse
7:25:52
código olha só que legal Aqui tá o meu carro em cima sou um carro indo pelas ruas modelo Polo fabri
7:25:59
sou um carro andando pelas ruas modelo A5 Sportback fabricante Audi tenho dois carros na memória do
7:26:04
representados agora Então veja o poder da orientação objetos com uma classe eu posso criar múltiplo
7:26:10
entre si com seus próprios dados suas próprias estruturas e interagir com cada um deles separadamen
7:26:18
uma classez inha com heran eu vou criar agora aqui em cima volto para fora do meu if name e vou cri
7:26:26

classe vamos supor motocicleta também vaiar de
7:26:33
veículo e aí para essa classe motocicleta também vou heredar o método init e vou criar um método mo
7:26:40
diferente para ela en

tão Def movimentar self Não esque da palavra
7:26:45
self para fazer a referência ao próprio objeto ISO isso que vai diferenciar objeto do outro quando
7:26:51
deles então é importante ter essa palavra self E aí vou colocar uma mensagenzinha diferente do tipo
7:26:57
muito então quando eu chamar movimentar pra motocicleta vai aparecer corro muito isso aqui é um exe
7:27:04
conceito de orientação objetos chamado de polimorfismo no qual o mesmo item no
7:27:11
caso aqui um método se comporta de forma diferente de acordo com a forma como ele
7:27:16
é usado então se eu uso movimentar para esse carro que é um tipo de veículo aparece uma mensagem se
7:27:22
motocicleta aparece uma mensagem diferente e a gente pode aqui embaixo agora criar uma moto então m
7:27:31
igual motocicleta aí eu tenho que passar os o os dados de fabricante e modelo
7:27:38
então pode ser uma Harley Davidson
7:27:43
Davidson beleza e o modelo dessa Harley pode ser uma night
7:27:51
Special beleza e aí é só chamar moto. movimentar pra gente ver a mensagem o
7:27:58
polimorfismo em ação e obter o fabricante modelo dela se a gente quiser
7:28:03
fazer essa consulta Então vamos executar para ver deixa eu puxar aqui o terminal para cima rodando
7:28:11
nightster Special fabricante Harley Davidson bom para finalizar eu vou criar
7:28:16
mai

s uma classez distinta também usando do conceito de herança a gente vem encapsulamento vimos polimo
7:28:24
herança Então vamos lá eu vou criar uma classe agora avião bem diferente um tipo
7:28:29
de veículo bem diferente desses outros também herdando de veículo certo e aqui pro avião eu vou
7:28:38
fazer uma alteração no método init dele porque ao criar o objeto avião além do
7:28:44
fabricante modelo eu também vou passar a categoria do avião então para isso eu tenho que ter o méto
7:28:50
então eu vou fazer um override em cima desse método init então Def aí eu
7:28:57
programa de novo init self sempre a primeira palavra vai
7:29:04
continuar tendo fabricante o modelo só que agora vai ter a categoria também
7:29:10
certo e aí eu vou dizer que self pon vou chamar de Cat para abreviar é igual a
7:29:16
categoria ou seja o elemento interno Cat vai receber o que está em categoria E
7:29:22

aqui a gente pode Claro deixá-lo privado com und und se for o caso passei a
7:29:28
categoria tenho que passar também o fabricante o modelo E aí o seguinte eu poderia simplesmente cop
7:29:35
aqui self fabricante fabricante modelo modelo mas é desnecessário fazer isso mesmo porque eu posso
7:29:41
itens aqui para serem copiados o método n ele pode ser e executado para fazer um monte de coisas e
7:29:49
desnecessária de códig

o então o que eu vou falar o seguinte eu vou usar um codozinho aqui que é para dizer que além
7:29:56
desse self é diferente o resto é igual E para isso eu uso a palavra super e abre
7:30:03
e fecha o parênteses super significa superclasse significa a classe da qual o
7:30:09
avião herda portanto é uma referência direta à classe veículo Então eu quero
7:30:14
que lá ó ponto aí eu vou chamar o meu init
7:30:20
super init e vou dizer que eu quero herdar o
7:30:26
fabricante e o modelo então esses dois itens vem da vem Obrigatoriamente de
7:30:32
veículo e o cat a categoria implementada dentro do avião assim eu não preciso
7:30:38
criar aquelas linhas de código para capturar os dados de fabricante modelo de novo e além disso eu
7:30:43
um método getter se eu quiser consultar a categoria do avião depois então eu vou criar um método ge
7:30:50
capturar a categoria do avião eu vou simplesmente copiar isso aqui Vou definir esse método eu vou
7:30:58
chamar get categoria get
7:31:03
categoria return self Cat então quando ele for executado ele retorna a
7:31:08
categoria que foi fornecida na hora que eu criei o avião e se eu quiser saber o fabricante o modelo
7:31:14
Ele foi herdado de veículo e é esse método aqui get fabricante modelo então quando executa esse mét
7:31:20
programado eu não preciso fazer ele de novo lá emba

ixo porque é o conceito de herança então ele já tá disponível pro
7:31:26
meu avião Então vamos testar agora eu vou criar um avião que eu vou chamar de meu avião que é o
7:31:32
objeto avião é o nome da classe e aí eu vou passar eh Os dados aqui no caso
7:31:39
agora são três parâmetros fabricante Boeing vírgula eh o modelo pode ser um
7:31:47
747 vírgula e categoria é um avião para voos comerciais então comercial isso vai
7:31:55
criar o meu avião Note que eu tô dando exemplos de com Strings Mas você poderia ter números aqui nú
7:32:01
inteiro até mesmo valores booleanos podem ser passados dependendo da funcionalidade que você quer i
7:32:07
eu vou chamar o meu avião pon movimentar ponto movimentar eu vou chamar o meu avião PG fabricante m
7:32:16

o fabricante modelo e eu vou chamar o meu avião ponto get número de registro
7:32:21
não categoria isso categoria vamos ver se tá tudo certo aqui ou se falta a gente fazer
7:32:28
alguma coisa então deixa eu puxar isso aqui para cima vou comentar Esses códigos dos carros e das m
7:32:34
não nos atrapalhar então deixa eu selecionar tudo cont control KC comentou
7:32:40
Então vamos lá vamos ver esse avião que será que aparece quando eu executar sou um veículo e me des
7:32:47
fabricante Boeing beleza que que tá faltando aqui vamos lá falt algumas coisinhas que fa

lta aqui é
7:32:54
o seguinte quando eu dei esse G categoria o get categoria que ele
7:32:59
retorna O self C retornou para quem tá na memória da máquina não apareceu aqui
7:33:05
na saída então neste caso eu vou dar um print Antes desse meu
7:33:11
avião e aqui usando uma f stringz eu vou pedir para retornar este valor aqui a
7:33:17
categoria dele vamos colocar categoria do avião e aí dentro das chaves eu
7:33:23
coloco a minha categoria agora deve aparecer a categoria dele outro problema
7:33:28
foi foi o movimento sou um veículo el me desloco Ele usou exatamente a mensagem que tá na minha cla
7:33:36
superclasse só que o avião ele tem um movimento específico então a gente pode implementar de novo r
7:33:42
polimorfismo um novo método pro movimento do avião que é o método
7:33:47
movimentar só que na são avião agora então esse método aqui ele vai imprimir
7:33:53
na tela uma mensagem do tipo eu vo alto pronto tá ótimo já é suficiente então
7:34:02
agora se eu executar olha só o que vai acontecer eu vou alto mudou né a
7:34:07
mensagem por causa do polimorfismo modelo e fabricante aqui e a categoria
7:34:13
comercial por conta do nosso método get a gente conseguiu ler esse parâmetro que foi passado durant
7:34:19
objeto a criação desse objeto a partir da classe avião e o resto tudo er dado da classe veículo Ent

qui
7:34:26
foi um primer de orientação objetos em Python tem bastante coisa eu sei na
7:34:31
verdade a disciplina de orientação objetos é muito mais extensa que isso dá para estudar um semestr
7:34:36
disso na na faculdade por exemplo mas com isso aqui a gente já consegue começar a criar aplicações
7:34:42
conceitos nos nossos scripts né Por exemplo quando você precisar modelar algo do mundo real dentro
7:34:49
apliação e trabalhar com esse algo é muito mais amigável trabalhar usando esse conceito de orientaç
7:34:54
porque você dá nomes PR as coisas pras ações e pros atributos da mesma forma que você usa no mundo
7:35:01

de serem itens físicos ou itens abstratos Independente de ser um veículo ou uma conta corrente de u
7:35:07
exemplo Então tá aí nessa aula vimos o conceito de orientação objetos classe objeto método e atribu
7:35:14
eit setters getters e os conceitos de encapsulamento polimorfismo e herança em
7:35:21
oo tente criar outras classes para representar eh outras entidades do mundo
7:35:26
real para você poder treinar e consolidar esse conhecimento na sua mente vamos abordar Nesta aula c

Apêndice B: Tabela de funções/builtins úteis

dir(x): lista atributos e métodos disponíveis em x.
hasattr(x, 'nome'): verifica se x possui atributo.
getattr(x, 'nome', padrão): obtém atributo dinamicamente.
setattr(x, 'nome', valor): define atributo dinamicamente.
isinstance(x, Classe) / **issubclass(C, Base)**: checam tipos em runtime.
vars(x): dicionário de atributos (quando disponível).

```
class Demo:
    def __init__(self): self.valor = 42

d = Demo()
print(dir(d))
print(hasattr(d, "valor"))
print(getattr(d, "valor"))
setattr(d, "valor", 99)
print(d.valor)
```

Referências e próximos passos

Pesquise PEPs relacionadas (PEP 8, PEP 484, PEP 544) e documentação oficial do Python sobre classes, dataclasses e abc. Pratique convertendo scripts estruturados em pequenos domínios de objetos.