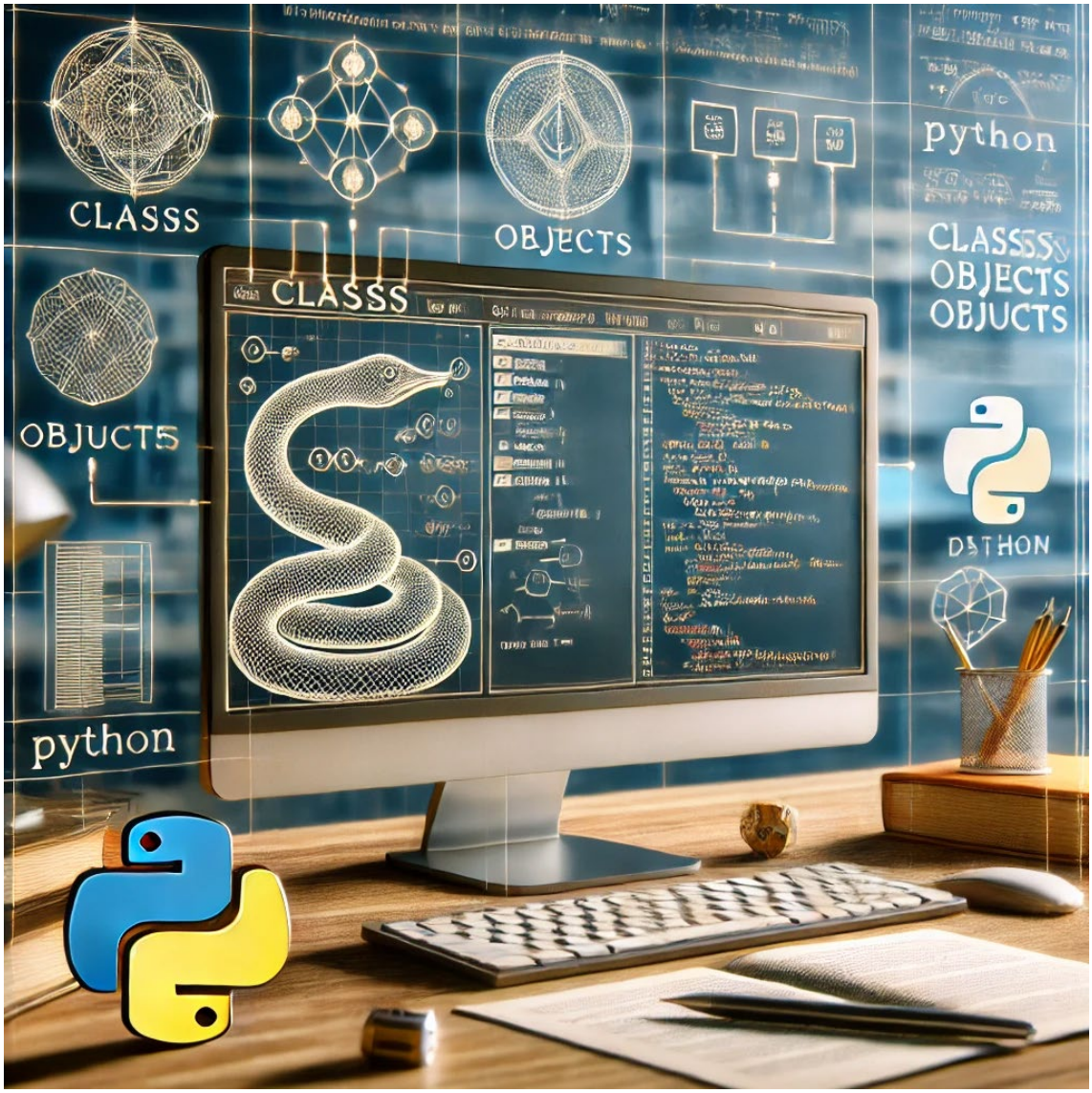
Digital Transformation Academy

CONCEITOS DE ORIENTAÇÃO A OBJETO COM PYTHON

PROF. EMMERSON SANTA RITA DA SILVA

LG inova POLO DE INOVAÇÃO IFAM INSTITUTO FEDERAL Amazonas Campus Manaus Zona Leste FAEPI

SUMÁRIO


CLASSS
OBJECTS
python
CLASSSS
OBJECTS
OBJUCTS
CLASSS
OBJUCTS
DSTHON
python

APRESENTAÇÃO

Caro aprendiz,

Seja bem-vindo ao curso **Conceitos de Orientação a Objeto com Python** do ZL Academy – Digital Transformation.

Este curso surgiu da necessidade de desenvolver, com os alunos e os parceiros do ZL Academy, conhecimentos e competências relacionadas Orientação a Objetos.

O objetivo deste curso é proporcionar uma compreensão sólida dos conceitos de Programação Orientada a Objetos (POO) utilizando a linguagem Python, uma das mais populares e versáteis no desenvolvimento de software atualmente. Para tanto, serão desenvolvidas/aprimoradas as seguintes competências:

- Entender e aplicar os conceitos fundamentais de POO, incluindo classes, objetos, herança, encapsulamento e polimorfismo.
- Desenvolver, testar e manter aplicativos em Python, utilizando práticas recomendadas de programação.
- Modelar e solucionar problemas complexos através da criação de estruturas de dados e algoritmos eficientes utilizando POO.
- Criar aplicativos modulares e escaláveis que seguem os princípios da POO, facilitando a manutenção e a expansão futura.

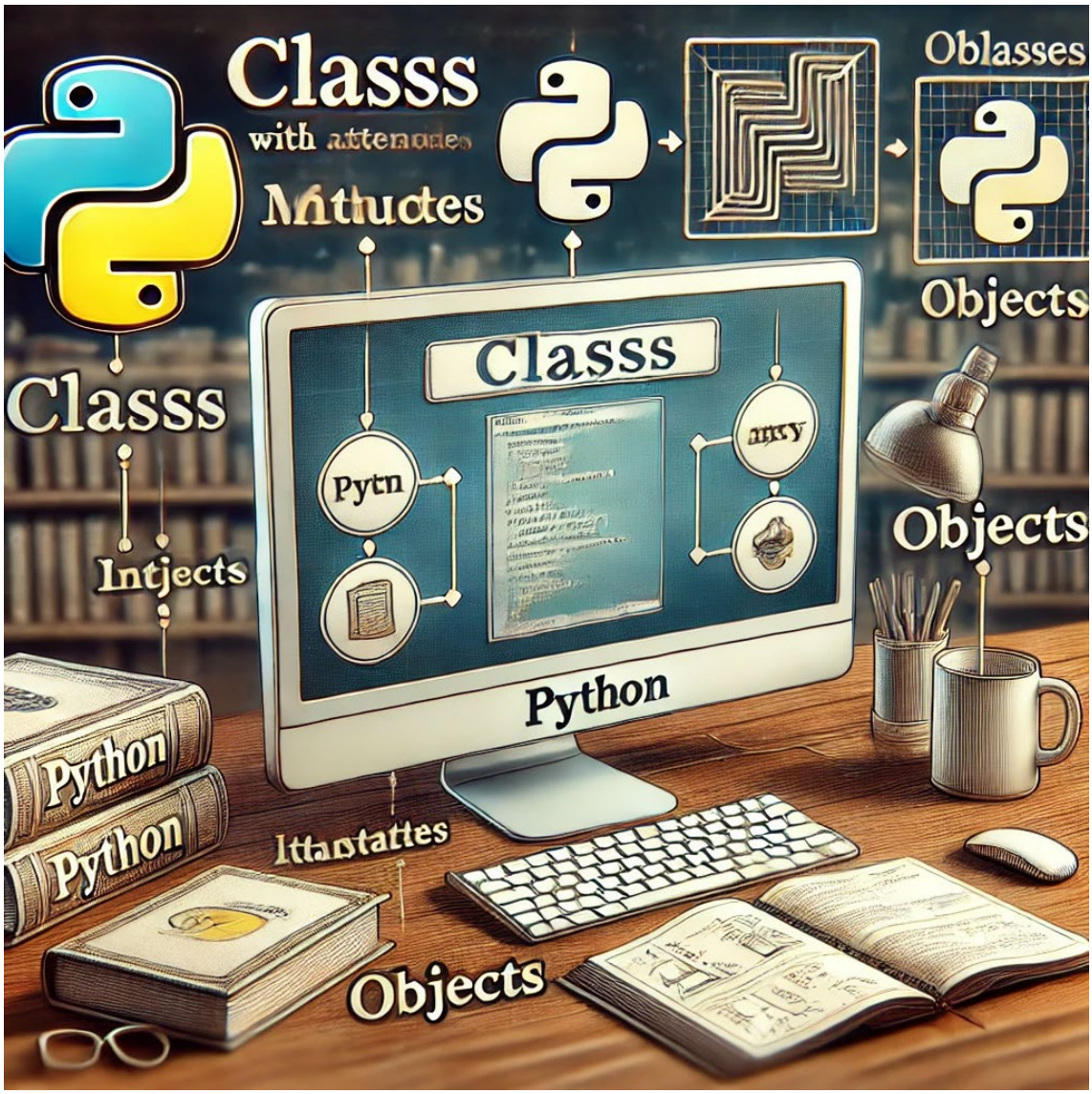
O Que Você Vai Aprender?

- **Fundamentos da POO**: Entenda os princípios básicos da Programação Orientada a Objetos e como aplicá-los em Python.
- **Classes e Objetos**: Aprenda a criar e manipular classes e objetos, os blocos de construção fundamentais da POO.
- **Encapsulamento e Herança**: Descubra como proteger dados e reutilizar código de forma eficiente.
- **Polimorfismo**: Explore a flexibilidade e a modularidade oferecidas pela POO.

- **Projetos Práticos**: Aplique os conceitos aprendidos em projetos práticos que simulam situações do mundo real.

Ao término do curso, os alunos não apenas dominarão a Programação Orientada a Objetos com Python, mas também estarão preparados para enfrentar desafios reais no desenvolvimento de software, prontos para contribuir de forma significativa em equipes de desenvolvimento e projetos de tecnologia.

Bons estudos!


Classs
Oblasses
Objects
Classs
Classs
Objects
Intjects
Python
Python
Python
Objects

Introdução à Programação Orientada a Objetos

DEFINIÇÃO E IMPORTÂNCIA

A Programação Orientada a Objetos (POO) é um paradigma de programação que utiliza "objetos" - entidades que combinam dados e comportamentos - como unidades fundamentais de programação. Os objetos são instâncias de "classes", que podem ser consideradas como moldes ou *templates* que definem as características (atributos) e comportamentos (métodos) dos objetos.

A POO oferece várias vantagens que a tornam um paradigma popular e amplamente utilizado no desenvolvimento de software. Sua importância pode ser destacada nos seguintes aspectos:

1. **Modularidade e Reutilização de Código**: A POO permite que o código seja organizado em módulos independentes (classes e objetos), facilitando a manutenção e a reutilização. Classes e objetos bem definidos podem ser reutilizados em diferentes partes do programa ou em diferentes projetos.
2. **Facilidade de Manutenção**: A estrutura modular e encapsulada da POO facilita a manutenção do código. Alterações e correções podem ser feitas em classes específicas sem afetar outras partes do programa, reduzindo o risco de introduzir novos bugs.
3. **Abstração e Modelagem de Problemas**: A POO permite a modelagem de problemas do mundo real de forma mais intuitiva e natural. Conceitos como herança e polimorfismo permitem que os desenvolvedores criem hierarquias e relações entre classes que refletem a complexidade do mundo real.
4. **Flexibilidade e Extensibilidade**: A herança e o polimorfismo fornecem uma base para a criação de sistemas extensíveis. Novas funcionalidades podem ser adicionadas criando novas classes que

herdam de classes existentes, promovendo a evolução contínua do sistema.

5. **Facilitação do Trabalho em Equipe**: A POO facilita a divisão do trabalho entre equipes de desenvolvimento. Cada desenvolvedor ou equipe pode trabalhar em classes e módulos específicos, permitindo um desenvolvimento paralelo e mais eficiente.

6. **Desenvolvimento de Software Complexo**: A POO é especialmente útil no desenvolvimento de sistemas complexos e de grande escala, onde a organização e a clareza do código são cruciais. Ela fornece uma estrutura clara para o desenvolvimento e a manutenção de tais sistemas.

HISTÓRIA DA POO

O termo Programação Orientada a Objetos foi criado por Alan Kay, autor da linguagem de programação Smalltalk. Mas mesmo antes da criação do Smalltalk, algumas das ideias da POO já eram aplicadas, sendo que a primeira linguagem a realmente utilizar estas ideias foi a linguagem simula 67, criada por Ole Johan Dahl e Kristen Nygaard em 1967. O infográfico a seguir apresenta uma linha do tempo detalhada sobre a evolução da Programação Orientada a Objetos (POO), destacando os principais marcos e autores que contribuíram para o desenvolvimento desse paradigma de programação.

FIGURA 1 - INFOGRÁFICO HISTÓRIA DA POO

Fonte: imagem gerada por IA

- **1960s - Simula:** Kristen Nygaard e Ole-Johan Dahl criaram a linguagem simula, considerada a primeira linguagem de programação orientada a objetos. Desenvolvida na Noruega, a Simula introduziu conceitos fundamentais como classes e objetos, que formam a base da POO.
- **1970s - Smalltalk:** Alan Kay, juntamente com sua equipe na Xerox PARC, desenvolveu a linguagem Smalltalk. Alan Kay é amplamente creditado por cunhar o termo "orientação a objetos". Smalltalk popularizou muitos dos conceitos de POO, incluindo herança e polimorfismo, e teve uma grande influência no design de muitas linguagens de programação modernas.
- **1980s - C++:** Bjarne Stroustrup criou o C++, uma linguagem que estendeu o C com funcionalidades de POO. O C++ tornou-se extremamente popular por permitir a programação de sistemas e aplicações com um alto nível de eficiência e controle, mantendo a capacidade de modelar objetos do mundo real.
- **1980s - Objective-C:** Brad Cox e Tom Love desenvolveram o Objective-C, combinando a simplicidade do C com os conceitos de POO do Smalltalk. Objective-C foi amplamente adotado pela Apple e se tornou a linguagem principal para o desenvolvimento de aplicativos para Mac OS e iOS até a introdução do Swift.
- **1990s - Java:** James Gosling e sua equipe na Sun Microsystems lançaram o Java, uma linguagem projetada para ser portátil, segura e orientada a objetos. Java trouxe o conceito de "write once, run anywhere" (escreva uma vez, execute em qualquer lugar), sendo amplamente utilizada no desenvolvimento de aplicativos web e corporativos.
- **2000s - C#:** Desenvolvida pela Microsoft, o C# foi introduzido como parte da plataforma .NET. C# é uma linguagem orientada a objetos que incorpora muitas características do C++ e Java, proporcionando uma forte integração com a infraestrutura .NET e sendo amplamente utilizada no desenvolvimento de aplicativos para Windows.
- **2000s - Python:** Embora Python tenha sido criado por Guido van Rossum nos anos 1980, sua popularidade como uma linguagem orientada a objetos cresceu significativamente nos anos 2000. Python é conhecida por sua

simplicidade e legibilidade, tornando-se uma escolha popular tanto para iniciantes quanto para desenvolvedores experientes em diversas áreas, incluindo ciência de dados, desenvolvimento web e automação.

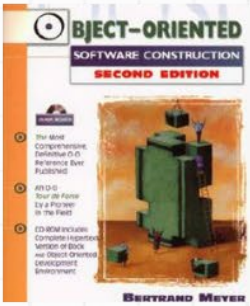
Outros autores

01

Larry Constantine: Embora nos anos 60 Constantine não tenha feito coisa alguma dirigida ao tópico da "orientação a objeto", ele realmente pesquisou de forma intensiva os critérios fundamentais para um bom desenho de software. Na realidade, ele foi uma das primeiras pessoas até mesmo a sugerir que o software poderia ser desenhado antes de programado. Muitas das noções de Constantine provaram sua aplicabilidade no mundo atual da orientação a objetos.

02

Dijkstra, em "The Conscience of Software Correcteness", tem nos causado uma culpa constante. Em seu primeiro trabalho, ele propôs algumas idéias sobre como construir software em camadas de abstração com uma estrita separação semântica entre camadas sucessivas. Isso representa uma forma de encapsulamento muito forte, fato esse central à orientação a objeto.

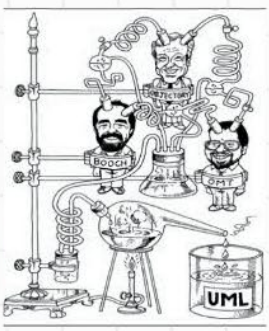


03

Bertrand Meyer: A força de Meyer é a mistura das melhores idéias da informática com as melhores idéias da orientação a objeto. O resultado: uma linguagem e um ambiente chamados de Eiffel. Eiffel é uma raridade no mundo do software, por atrai simultaneamente acadêmicos, engenheiros de software e aquele pessoal nas trincheiras que precisa de um código robusto. Não importa a linguagem orientada a objeto de sua preferência, definitivamente você deve aprender os conceitos que existem por detrás da linguagem Eiffel se quiser tornar-se um verdadeiro profissional orientado a objeto.

04

Grandy Booch, Ivar Jacobson e Jim Rumbaugh: Esses três personagens são conhecidos coletivamente pelo apelido de "Os Três Amigos". Muito embora cada um deles tenha seu próprio direito à fama devido a seus trabalhos individuais orientados a objeto, como um grupo esse direito surge de suas colaborações no final dos anos 90 no tocante à racionalização da notação orientada a objeto. O resultado do trabalho deles foi a Unified Modeling Language, uma linguagem de modelagem unificada gráfica contendo uma forma de expressão visual e uma escora semântica perfeita.



Você sabia?

Sabia que o conceito de "classe" e "objeto" na Programação Orientada a Objetos (POO) foi inspirado pela ideia de "objetos" do mundo real?

Kristen Nygaard e Ole-Johan Dahl, os criadores da primeira linguagem de POO, Simula, imaginaram que programar poderia ser como descrever o mundo real, onde cada objeto tem suas próprias características e comportamentos. Assim, um "carro" em POO pode ser uma classe com atributos como cor e modelo, e métodos como acelerar e frear. Essa abordagem revolucionou a forma como desenvolvemos software, tornando-o mais intuitivo e modular!


Python
CLASSS
& OBJECTS
ATTABBUTES
& METHODS
ENIJACTES
& MOBHOTS
ATTABBUTES
AND METHONS
ENCCAPLATION

Fundamentos da POO em Python

O conceito de orientação a objetos é fundamentado nessa estrutura, pois possui quatro pilares mestres, que são: classe, objeto, atributos e métodos, os quais serão rapidamente expostos em seguida, juntamente com conceitos que norteiam seu uso.

CLASSES E OBJETOS

A resolução de um problema passa pela análise de uma determinada situação real, tendo-se por objetivo a construção de um modelo (Figura 2) que represente esta situação real. Este modelo deverá considerar os objetos (entidades) que integram o problema. Pode ser considerada como objeto, qualquer "coisa" que tenha algum significado dentro do contexto do problema, seja ela concreta ou abstrata.

> "*Classe é o conjunto de objetos que se define pelo fato de tais objetos, e só eles, terem uma ou mais características em comum*".

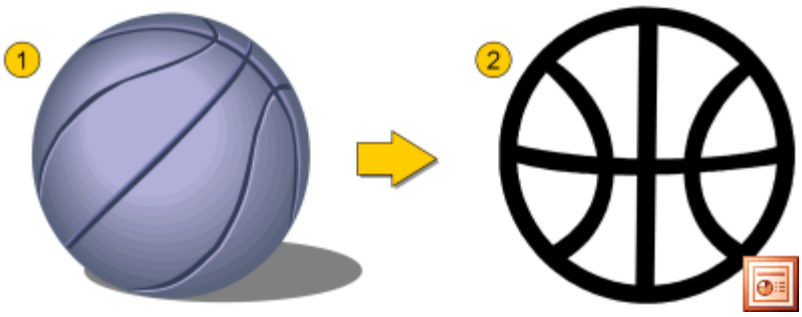


FIGURA 2: REPRESENTAÇÃO DO MUNDO REAL: SOMENTE OS ASPECTOS MAIS RELEVANTES SÃO CONSIDERADOS.

Pode-se observar que um objeto pode ser qualquer coisa que apresente alguma utilidade ou sirva a algum propósito. Cada objeto consiste em uma

entidade com identidade própria. Assim, mesmo que a partir de uma observação simples se diga que dois objetos são iguais, têm-se dois objetos distintos. Se analisarmos dois exemplares deste texto, por exemplo, veremos que se constituem em dois objetos distintos, apesar de terem a mesma quantidade de páginas, mesmo conteúdo, etc.

"*Objeto é uma ocorrência específica de uma classe, ou seja, uma instância de uma classe ou um elemento específico do mundo real*".

Em termos de programação podemos definir um objeto como sendo a abstração de uma entidade do mundo real, que apresenta sua própria existência, identificação, características de composição e que tem alguma utilidade, isto é, pode executar determinados serviços quando solicitado.

ATRIBUTOS E MÉTODOS

Atributo

É o conjunto de características específicas de um objeto. Em sua obra (AMBLER, 1997), define o atributo comparando-o a um elemento de dados definido em um registro.

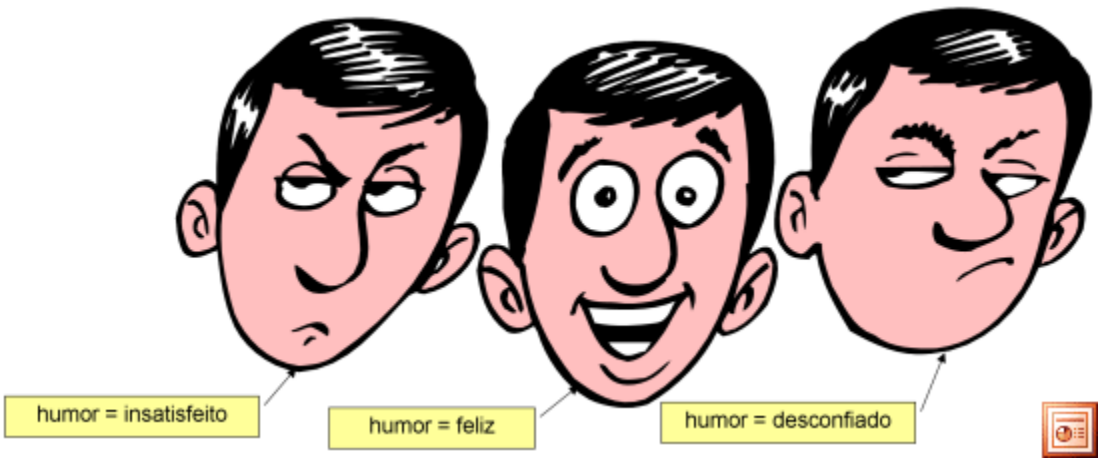


FIGURA 3: ATRIBUTO DE ESTADO "HUMOR".

O conteúdo de um atributo pode ser público ou privado. Quando privado, ocorre o efeito de ocultamento de informações de um atributo de uma determinada classe ou objeto, o qual não será visualizado ou utilizado na forma de acesso público.

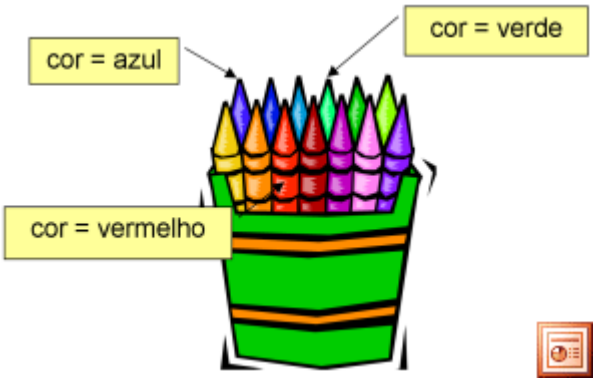


FIGURA 4: ATRIBUTO DE CARACTERÍSTICA "COR".

Métodos

O conceito de método está associado à forma como um determinado atributo será alterado, ou seja, método é a característica que possibilita alterar a funcionalidade de um atributo. O conceito de método num nível mais amplo possibilita efetuar o controle lógico que refletirá uma determinada ação (designar um comportamento) sobre o objeto e, por conseguinte, a sua classe também, ou melhor, sua **operação**.

Em sua obra (AMBLER, 1997), afirma que um método pode ser visto como uma função (funcionalidade) de um objeto, pois é através dos métodos que se torna possível modificar os atributos de um objeto, ou seja, um método é algo que estabelece o que realmente um objeto faz. As operações são acessadas por mensagens.

Diferença entre Métodos de Classe e Métodos de Instância em Python

Em Python, as classes podem definir diferentes tipos de métodos, cada um com comportamentos e propósitos distintos. Os dois tipos principais são métodos de instância e métodos de classe.

Métodos de Instância

Métodos de instância são os métodos mais comuns em Python. Eles operam em instâncias de uma classe e podem acessar e modificar os atributos dessas instâncias.

Características dos Métodos de Instância:

- Recebem a instância da classe (self) como primeiro argumento.
- Podem acessar e modificar o estado da instância (atributos de instância).
- Podem chamar outros métodos de instância.

Exemplo de Método de Instância:

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

    def apresentar(self):
        print(f"Meu nome é {self.nome} e eu tenho {self.idade} anos.")

# Criando uma instância da classe Pessoa
pessoa = Pessoa("Alice", 30)

# Chamando o método de instância
pessoa.apresentar()
```

CÓDIGO-FONTE 1: CLASSE PESSOA – EXEMPLO DE MÉTODO DE INSTÂNCIA

Explicação:

apresentar(self): É um método de instância que recebe a instância da classe self e pode acessar os atributos nome e idade dessa instância.

Métodos de Classe

Métodos de classe operam na classe em si, em vez de em uma instância específica. Eles podem acessar e modificar atributos de classe, que são compartilhados entre todas as instâncias da classe.

Características dos Métodos de Classe:

- Recebem a classe (cls) como primeiro argumento.
- Podem acessar e modificar o estado da classe (atributos de classe).
- São decorados com @classmethod.

Exemplo de Método de Classe:

```python
class Pessoa:
    numero_de_pessoas = 0  # Atributo de classe

    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
        Pessoa.numero_de_pessoas += 1

    @classmethod
    def total_de_pessoas(cls):
        print(f"Total de pessoas: {cls.numero_de_pessoas}")

# Criando instâncias da classe Pessoa
pessoa1 = Pessoa("Alice", 30)
pessoa2 = Pessoa("Bob", 25)

# Chamando o método de classe
Pessoa.total_de_pessoas()
```

CÓDIGO-FONTE 2 – CLASSE PESSOA – EXEMPLO DE MÉTODO DE CLASSE

Explicação:

total_de_pessoas(cls): É um método de classe que recebe a classe cls e pode acessar o atributo de classe numero_de_pessoas.

Comparação

Métodos de Instância:

- Recebem a instância (self) como primeiro argumento.
- Operam em atributos e métodos específicos da instância.
- Usados para comportamento específico de objetos individuais.

Métodos de Classe:

- Recebem a classe (cls) como primeiro argumento.
- Operam em atributos e métodos da classe como um todo.
- Usados para comportamento que se aplica a todas as instâncias da classe.

Exemplo Comparativo:

```python
class Exemplo:
    contador = 0  # Atributo de classe

    def __init__(self, valor):
        self.valor = valor  # Atributo de instância
        Exemplo.contador += 1

    def metodo_instancia(self):
        # Acessa o atributo de instância
        print(f"Valor da instância: {self.valor}")

    @classmethod
    def metodo_classe(cls):
        # Acessa o atributo de classe
        print(f"Contador da classe: {cls.contador}")

# Criando instâncias da classe Exemplo
obj1 = Exemplo(10)
obj2 = Exemplo(20)

# Chamando métodos de instância
obj1.metodo_instancia()
obj2.metodo_instancia()

# Chamando método de classe
Exemplo.metodo_classe()
```

CÓDIGO-FONTE 3: COMPARAÇÃO ENTRE MÉTODOS DE INSTÂNCIA E DE CLASSE

Explicação:

metodo_instancia(self): Acessa o atributo de instância valor e imprime seu valor.

metodo_classe(cls): Acessa o atributo de classe contador e imprime seu valor.

Método construtor

O método construtor em Python é um método especial que é automaticamente chamado quando um novo objeto de uma classe é criado. Este método é utilizado para inicializar os atributos do objeto e executar qualquer configuração inicial que a classe precise. Em Python, o método construtor é definido pelo método __init__.

Características do Método Construtor:

- **Nome Especial**: O nome __init__ é uma convenção especial em Python que indica que o método é um construtor.
- **Inicialização de Atributos**: O __init__ é usado para definir e inicializar os atributos de instância de uma classe.
- **Parâmetro self**: O primeiro parâmetro do __init__ deve ser sempre self, que é uma referência à instância que está sendo criada. Outros parâmetros podem ser adicionados para inicializar os atributos da instância.
- **Automático**: O __init__ é automaticamente chamado toda vez que um novo objeto da classe é instanciado.

Encapsulamento

Encapsulamento é um dos quatro princípios fundamentais da Programação Orientada a Objetos (POO). Ele se refere à prática de esconder os detalhes internos de um objeto e expor apenas o necessário para a interação com esse objeto. Isso significa que os atributos (dados) de um objeto são protegidos contra acesso direto do código externo, e a manipulação desses dados é feita através de métodos (funções) definidos na classe.

Benefícios do Encapsulamento

- **Proteção de Dados**: Evita que os dados internos de um objeto sejam corrompidos por acesso ou modificações não controladas.
- **Facilidade de Manutenção**: Facilita a alteração da implementação interna de uma classe sem afetar o código que a utiliza.
- **Modularidade**: Permite que diferentes partes de um programa sejam desenvolvidas e entendidas de forma independente.

Exemplos de Encapsulamento em Python

Em Python, o encapsulamento é implementado usando diferentes níveis de visibilidade para os atributos e métodos. Python utiliza uma convenção de nomenclatura para indicar diferentes níveis de acesso:

Público: Atributos e métodos que podem ser acessados de fora da classe.

Protegido: Atributos e métodos que devem ser acessados apenas dentro da classe e das subclasses.

Privado: Atributos e métodos que devem ser acessados apenas dentro da própria classe.

Exemplo Completo com Classes, Objetos, Atributos, Métodos e Encapsulamento em Python

Vamos criar um exemplo que demonstre o uso de classes, objetos, atributos, métodos, encapsulamento e construtores de classe.

```python
class ContaBancaria:
    # Atributo de classe compartilhado por todas as instâncias
    taxa_juros = 0.02

    def __init__(self, titular, saldo):
        # Atributos privados de instância
        self.__titular = titular
        self.__saldo = saldo

    # Método de instância para depositar dinheiro
    def depositar(self, valor):
        if valor > 0:
            self.__saldo += valor
            print(f"Depósito de R${valor:.2f} realizado com sucesso.")
        else:
            print("O valor do depósito deve ser positivo.")

    # Método de instância para sacar dinheiro
    def sacar(self, valor):
        if valor > 0 and valor <= self.__saldo:
            self.__saldo -= valor
            print(f"Saque de R${valor:.2f} realizado com sucesso.")
        else:
            print("Saldo insuficiente ou valor de saque inválido.")
```

CÓDIGO-FONTE 4: EXEMPLO CONSTRUTOR (PARTE 1)

```python
    # Método de instância para exibir o saldo
    def mostrar_saldo(self):
        print(f"Saldo de {self.__titular}: R${self.__saldo:.2f}")

    # Método de classe como construtor alternativo
    @classmethod
    def conta_com_juros(cls, titular, saldo_inicial):
        saldo_com_juros = saldo_inicial * (1 + cls.taxa_juros)
        return cls(titular, saldo_com_juros)

# Criando objetos da classe ContaBancaria usando o construtor padrão
conta1 = ContaBancaria("Alice", 1000)

# Criando objetos da classe ContaBancaria usando o construtor de classe
conta2 = ContaBancaria.conta_com_juros("Bob", 1000)

# Usando métodos da classe ContaBancaria
conta1.mostrar_saldo()
conta1.depositar(200)
conta1.sacar(150)
conta1.mostrar_saldo()

conta2.mostrar_saldo()
conta2.depositar(500)
conta2.sacar(300)
conta2.mostrar_saldo()
```

CÓDIGO-FONTE 5: EXEMPLOS CONSTRUTOR (PARTE 2)

Perguntas para Reflexão:

1. O que acontece se você tentar acessar diretamente o atributo `__saldo` fora da classe?
2. Como o método `__init__` ajuda a inicializar os atributos da classe?
3. Por que é importante usar encapsulamento em uma classe como `ContaBancaria`?
4. Como você pode adicionar métodos adicionais para aumentar a funcionalidade da classe `ContaBancaria`?

Getters e Setters em Python

Getters e setters são métodos usados para acessar (get) e modificar (set) os atributos de uma classe. Em Python, é comum usar propriedades (@property) para implementar getters e setters, permitindo o encapsulamento e a validação dos dados.

Usando @property para Getters e Setters

Com o decorador @property, você pode definir um método que se comporta como um atributo de uma classe. Para definir um setter, você usa @<nome_do_atributo>.setter.

Exemplo de Getters e Setters com Validação

Vamos criar uma classe Pessoa que valida o valor de um atributo idade para garantir que seja sempre um número positivo.

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade  # Usa o setter para validação inicial

    @property
    def idade(self):
        return self._idade

    @idade.setter
    def idade(self, valor):
        if valor < 0:
            raise ValueError("A idade não pode ser negativa.")
        self._idade = valor

    @property
    def nome(self):
        return self._nome

    @nome.sette
    def nome(self, valor):
        if not valor:
```

CÓDIGO-FONTE 6: EXEMPLO DE MÉTODOS GETTERS E SETTERS (PARTE 1)

```
            raise ValueError("O nome não pode ser vazio.")
        self._nome = valor

# Criando uma instância da classe Pessoa
try:
    pessoa = Pessoa("Alice", 30)
    print(pessoa.idade)  # Saída: 30
    pessoa.idade = -5  # Isso levantará um ValueError
except ValueError as e:
    print(e)  # Saída: A idade não pode ser negativa.

# Verificando a validação do nome
try:
    pessoa.nome = ""  # Isso levantará um ValueError
except ValueError as e:
    print(e)  # Saída: O nome não pode ser vazio.
```

CÓDIGO-FONTE 7: EXEMPLO DE MÉTODOS GETTERS E SETTERS (PARTE 2)

Explicação do Exemplo:

- **Definição da Classe Pessoa:**
  - O método __init__ inicializa os atributos nome e idade.
  - O atributo idade é definido usando o setter, permitindo a validação inicial.
- **Getter e Setter para idade:**
  - O método idade decorado com @property é o getter que retorna o valor de _idade.
  - O método idade decorado com @idade.setter é o setter que valida o valor antes de atribuí-lo a _idade.
  - Se o valor for negativo, um ValueError é levantado.
- **Getter e Setter para nome:**
  - O método nome decorado com @property é o getter que retorna o valor de _nome.

- O método nome decorado com @nome.setter é o setter que valida o valor antes de atribuí-lo a _nome.
- Se o valor for vazio, um ValueError é levantado.

- **Uso da Classe Pessoa:**
  - Ao criar uma instância de Pessoa com idade 30, a validação ocorre e o valor é aceito.
  - Ao tentar definir idade como -5, um ValueError é levantado, indicando que a idade não pode ser negativa.
  - Ao tentar definir nome como uma string vazia, um ValueError é levantado, indicando que o nome não pode ser vazio.

Getters e setters em Python, implementados com o decorador @property, permitem encapsular e validar os atributos de uma classe de maneira elegante. Eles ajudam a garantir que os dados dos atributos estejam sempre em um estado válido, aumentando a robustez e a confiabilidade do código.

Vamos Codificar

Estes exercícios têm como objetivo ajudar o aluno a compreender e aplicar os conceitos de classe, objeto, métodos e encapsulamento em Python.

Exercício 1. O aluno deverá criar uma classe que modele um sistema de biblioteca, com métodos para adicionar livros, emprestar livros e verificar o inventário:

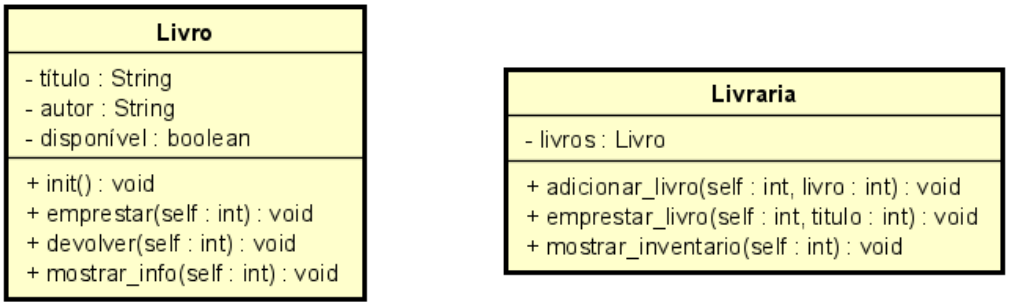


DIAGRAMA UML 1 – CLASSES LIVRO E LIVRARIA

Vamos implementar essa classe e entender os conceitos iniciais apresentados até aqui.

Exercício 2. (Individual): Criar uma classe simples no Python que represente um produto, c.om atributos (nome, preço, quantidade) e métodos para exibir e atualizar informações. Automatizar a criação de um objeto produto com BotCity, utilizando um formulário simples em uma página web de exemplo.

Você sabia?

Em Programação Orientada a Objetos, **encapsulamento** é como um cofre que guarda os segredos de um objeto! **Atributos** são os tesouros guardados dentro do cofre, e **métodos** são as chaves que permitem acessar e modificar esses tesouros de maneira controlada. Isso mantém o "mundo exterior" longe das partes sensíveis, garantindo que tudo funcione de maneira segura e previsível!

Herança

É o compartilhamento de atributos e operações entre classes com base em um relacionamento hierárquico. Uma classe pode ser definida de forma abrangente e depois refinada em sucessivas subclasses mais definidas.

Cada subclasse incorpora todas as propriedades de sua superclasse e acrescenta suas próprias exclusivas características.

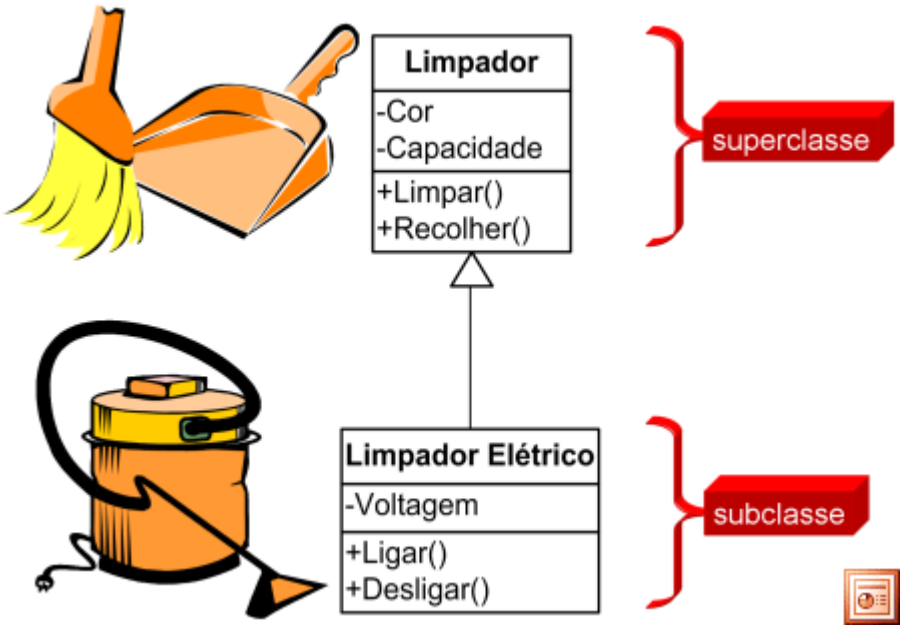


FIGURA 5 EXEMPLO DE HERANÇA. AS PROPRIEDADES DA SUPERCLASSE SÃO "ADOTADAS" PELA SUBCLASSE.

A herança representa outro caminho muito importante no qual a orientação a objeto diverge das abordagens dos sistemas convencionais. Ela efetivamente permite a construção de forma incrementada, de acordo com as instruções a seguir:

1. Construir as classes para lidar com o caso mais geral.
2. Em seguida, a fim de tratar com os casos especiais, acrescente classes mais especializadas, herdadas da primeira classe. Essas novas classes estarão habilitadas à utilização de todas as operações e atributos da classe original: tanto operações e atributos de classe, como operações e atributos de instância.

Tipos de Herança

1. **Herança Simples**: Uma subclasse herda de uma única superclasse.
2. **Herança Múltipla**: Uma subclasse herda de mais de uma superclasse.
3. **Herança Multinível**: Uma classe herda de uma subclasse que já herda de outra superclasse.
4. **Herança Hierárquica**: Uma superclasse serve como base para várias subclasses.
5. **Herança Híbrida**: Combinação de dois ou mais tipos de herança acima.

Exemplo em Python:

```python
# Superclasse
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

# Subclasse
class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

# Outra Subclasse
class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

# Instanciando objetos
dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.speak())  # Output: Buddy says Woof!
print(cat.speak())  # Output: Whiskers says Meow!
```

CÓDIGO-FONTE 8: EXEMPLO DE HERANÇA

Sobrecarga de Métodos

Sobrecarga de métodos refere-se à capacidade de uma classe ter múltiplos métodos com o mesmo nome, mas com diferentes assinaturas (tipos e/ou número de parâmetros). Em muitas linguagens de programação, como Java ou C++, a sobrecarga de métodos é um recurso comum. No entanto, Python não suporta sobrecarga de métodos diretamente como essas outras linguagens.

Sobrecarga de Métodos em Python

Em Python, você não pode definir múltiplos métodos com o mesmo nome e diferentes parâmetros diretamente na mesma classe. Em vez disso, você pode usar argumentos padrão ou *args e **kwargs para permitir que um único método lide com diferentes números de argumentos.

```python
class Calculadora:
    def adicionar(self, a, b=0, c=0):
        return a + b + c

calc = Calculadora()
print(calc.adicionar(5))        # Saída: 5
print(calc.adicionar(5, 10))    # Saída: 15
print(calc.adicionar(5, 10, 15)) # Saída: 30
```

CÓDIGO-FONTE 9: EXEMPLO DE SOBRECARGA USANDO ARGUMENTOS PADRÃO

Sobrecarga de Métodos vs. Herança

Sobrecarga de métodos não é um tipo de herança. Enquanto a sobrecarga de métodos envolve definir múltiplos métodos com o mesmo nome e diferentes assinaturas, a herança é um conceito onde uma classe (subclasse) herda atributos e métodos de outra classe (superclasse). A herança permite que as subclasses utilizem ou modifiquem o comportamento definido nas superclasses.

Sobreposição de Métodos (Override)

Sobreposição de métodos, ou sobrescrita, refere-se à prática de definir um método em uma subclasse com o mesmo nome e assinatura de um método na superclasse, substituindo o comportamento da superclasse.

```python
class Animal:
    def fazer_som(self):
        print("O animal faz um som")

class Cachorro(Animal):
    def fazer_som(self):
        print("O cachorro late")

animal = Animal()
cachorro = Cachorro()

animal.fazer_som()   # Saída: O animal faz um som
cachorro.fazer_som() # Saída: O cachorro late
```

CÓDIGO-FONTE 10: EXEMPLO DE SOBREPOSIÇÃO DE MÉTODOS

Diferença entre Sobrecarga e Sobreposição de Métodos

Sobrecarga de Métodos:

- **Definição**: Múltiplos métodos com o mesmo nome, mas diferentes assinaturas (número e/ou tipo de parâmetros).
- **Implementação em Python**: Não suportada diretamente; pode ser simulada usando argumentos padrão ou *args e **kwargs.
- **Uso**: Permitir diferentes formas de chamar um método com diferentes conjuntos de argumentos.

Sobreposição de Métodos:

- **Definição**: Um método em uma subclasse tem o mesmo nome e assinatura que um método na superclasse, substituindo o comportamento da superclasse.
- **Implementação em Python**: Suportada diretamente usando herança.
- **Uso**: Permitir que uma subclasse forneça uma implementação específica de um método que já é definido na sua superclasse.

Exemplo Comparativo:

```python
class Calculadora:
    def adicionar(self, *args):
        return sum(args)

calc = Calculadora()
print(calc.adicionar(5))          # Saída: 5
print(calc.adicionar(5, 10))      # Saída: 15
print(calc.adicionar(5, 10, 15))  # Saída: 30
```

CÓDIGO-FONTE 11: SOBRECARGA (SIMULADA) EM PYTHON

```python
class Animal:
    def fazer_som(self):
        print("O animal faz um som")

class Gato(Animal):
    def fazer_som(self):
        print("O gato mia")

animal = Animal()
gato = Gato()

animal.fazer_som()  # Saída: O animal faz um som
gato.fazer_som()    # Saída: O gato mia
```

CÓDIGO-FONTE 12: SOBREPOSIÇÃO DE MÉTODOS

Sobrecarga de Métodos: Em Python, você pode simular sobrecarga de métodos usando argumentos padrão ou ***args** e ****kwargs** para permitir que um método aceite diferentes números de argumentos.

Sobreposição de Métodos: É suportada diretamente em Python através da herança, onde uma subclasse pode redefinir métodos da superclasse para fornecer uma implementação específica.

Exemplo de Aplicação com Herança Múltipla em Python

A herança múltipla permite que uma classe herde atributos e métodos de mais de uma classe. Isso pode ser útil em diversas situações onde uma classe precisa combinar funcionalidades de várias classes base.

Exemplo: Sistema de Transporte

Vamos criar um exemplo onde temos várias classes representando diferentes tipos de veículos e funcionalidades adicionais como navegação e capacidade de carregar carga. A classe final VeiculoHibrido combinará essas funcionalidades.

```python
# Classe base para veículos
class Veiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def info_veiculo(self):
        print(f"Veículo: {self.marca} {self.modelo}")

# Classe base para navegação
class Navegacao:
    def __init__(self, gps_marca):
        self.gps_marca = gps_marca

    def info_navegacao(self):
        print(f"Sistema de navegação: {self.gps_marca}")

# Classe base para carga
class Carga:
    def __init__(self, capacidade):
        self.capacidade = capacidade
```

CÓDIGO-FONTE 13: EXEMPLO DE HERANÇA MÚLTIPLA (PARTE 1)

```
    def info_carga(self):
        print(f"Capacidade de carga: {self.capacidade} kg")

# Classe que herda de Veiculo, Navegacao e Carga
class VeiculoHibrido(Veiculo, Navegacao, Carga):
    def __init__(self, marca, modelo, gps_marca, capacidade):
        Veiculo.__init__(self, marca, modelo)
        Navegacao.__init__(self, gps_marca)
        Carga.__init__(self, capacidade)

    def info_completo(self):
        self.info_veiculo()
        self.info_navegacao()
        self.info_carga()

# Criando um objeto da classe VeiculoHibrido
veiculo_hibrido = VeiculoHibrido("Toyota", "Highlander", "Garmin", 500)

# Usando os métodos da classe VeiculoHibrido
veiculo_hibrido.info_completo()
```

CÓDIGO-FONTE 14: EXEMPLO DE HERANÇA MÚLTIPLA (PARTE 2)

Explicação do Exemplo:

- **Classe Veiculo:**
  - Representa características básicas de um veículo.
  - Método info_veiculo exibe informações sobre o veículo.
- **Classe Navegacao:**
  - Representa características relacionadas ao sistema de navegação.
  - Método info_navegacao exibe informações sobre o sistema de navegação.
- **Classe Carga:**
  - Representa características relacionadas à capacidade de carga.
  - Método info_carga exibe informações sobre a capacidade de carga.
- **Classe VeiculoHibrido:**

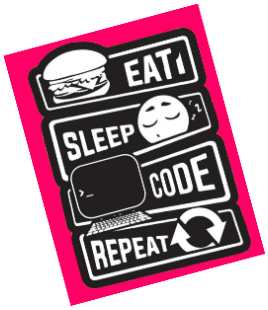
- Herda de Veiculo, Navegacao e Carga.
- O construtor __init__ inicializa as características das três classes base.
- Método info_completo chama os métodos das três classes base para exibir informações completas sobre o veículo híbrido.

- **Instanciação e Uso:**
  - Um objeto da classe VeiculoHibrido é criado com as características combinadas das três classes base.
  - O método info_completo é chamado para exibir todas as informações combinadas.

Saída do código:

```
Veículo: Toyota Highlander
Sistema de navegação: Garmin
Capacidade de carga: 500 kg
```

A herança múltipla em Python permite que uma classe combine funcionalidades de várias classes base. No exemplo, a classe VeiculoHibrido herda de Veiculo, Navegacao e Carga, combinando suas funcionalidades e permitindo a criação de um objeto com características de todas essas classes. A herança múltipla é poderosa, mas deve ser usada com cuidado para evitar complexidade excessiva e ambiguidades na resolução de métodos.



Vamos Codificar

Exercício 3. Crie uma classe Veiculo com os atributos marca e modelo, e um método informacao() que imprime esses atributos. Em seguida, crie uma classe Carro que herda de Veículo e adiciona o atributo numero_portas. Adicione um método na classe Carro que imprime todas as informações, incluindo o número de portas.

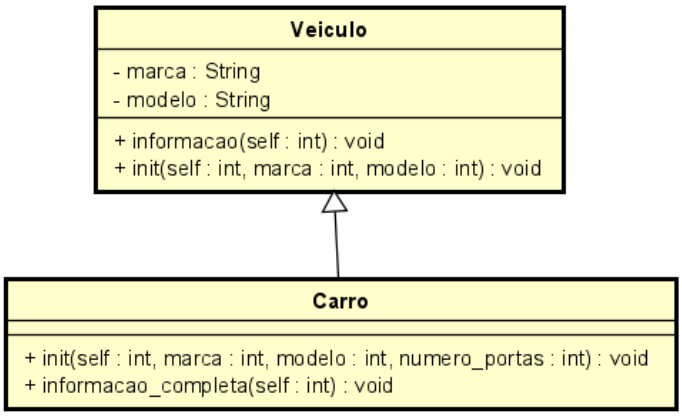


DIAGRAMA UML 2 – CLASSES VEÍCULO E CARRO

Exercício 4 (Individual): Criar uma hierarquia de classes que representem diferentes tipos de formulários (FormBase, FormularioContato, FormularioLogin). Utilizar BotCity para preencher automaticamente diferentes tipos de formulários em uma página web.

Você sabia?

Em Python, todas as classes herdam, direta ou indiretamente, da classe base object. Isso significa que mesmo se você não especificar explicitamente uma superclasse, a classe que você cria ainda terá os métodos e propriedades da classe object. Além disso, você pode usar a função super() para chamar métodos da superclasse, facilitando a reutilização de código e a extensão de funcionalidades.

Composição em Python

Composição é um conceito na Programação Orientada a Objetos onde uma classe é composta de uma ou mais instâncias de outras classes. Em outras palavras, uma classe é formada por objetos de outras classes, indicando uma relação "tem um" (has-a). Composição permite criar relações complexas entre objetos e construir objetos maiores e mais complexos a partir de componentes menores e mais simples.

DIFERENÇA ENTRE COMPOSIÇÃO E HERANÇA

- **Herança**:
    - **Relação "é um" (is-a)**: A herança representa uma relação onde uma classe (subclasse) herda atributos e métodos de outra classe (superclasse).
    - **Reutilização de Código**: Permite reutilizar o código da superclasse na subclasse.
    - **Extensibilidade**: A subclasse pode adicionar ou modificar o comportamento da superclasse.
- **Composição**:
    - **Relação "tem um" (has-a)**: A composição representa uma relação onde uma classe é composta por uma ou mais instâncias de outras classes.
    - **Modularidade**: Permite construir objetos complexos a partir de componentes menores e mais simples.
    - **Encapsulamento**: Facilita o encapsulamento, pois os objetos compostos podem gerenciar seus próprios componentes.

Exemplo de Composição em Python

Vamos criar um exemplo que demonstra a composição, onde uma classe Carro é composta de uma classe Motor e uma classe Roda.

Definição das Classes Componentes:

```python
class Motor:
    def __init__(self, potencia):
        self.potencia = potencia

    def ligar(self):
        print(f"Motor de {self.potencia} cavalos ligado.")

class Roda:
    def __init__(self, tamanho):
        self.tamanho = tamanho

    def girar(self):
        print(f"Roda de tamanho {self.tamanho} polegadas girando.")
```

CÓDIGO-FONTE 15: EXEMPLO DE CLASSE COMPONENTE

Definição da Classe Composta:

```python
class Carro:
    def __init__(self, marca, modelo, potencia_motor, tamanho_rodas):
        self.marca = marca
        self.modelo = modelo
        self.motor = Motor(potencia_motor)  # Composição: Carro tem um Motor
        self.rodas = [Roda(tamanho_rodas) for _ in range(4)]  # Composição: Carro tem 4 Rc

    def ligar(self):
        print(f"Carro {self.marca} {self.modelo} ligado.")
        self.motor.ligar()
        for roda in self.rodas:
            roda.girar()

# Criando um objeto da classe Carro
meu_carro = Carro("Toyota", "Corolla", 150, 16)

# Usando métodos da classe Carro
meu_carro.ligar()
```

CÓDIGO-FONTE 16: EXEMPLO DE CLASSE COMPOSTA

Explicação do Exemplo:

1. **Classes Componentes:**
    - **class Motor**: Define uma classe Motor com um atributo **potencia** e um método **ligar**.
    - **class Roda**: Define uma classe Roda com um atributo **tamanho** e um método **girar**.
2. **Classe Composta:**
    - **class Carro**: Define uma classe Carro que é composta de um Motor e quatro Rodas.
    - **self.motor = Motor(potencia_motor)**: O carro tem um motor, que é uma instância da classe Motor.
    - **self.rodas = [Roda(tamanho_rodas) for _ in range(4)]**: O carro tem quatro rodas, cada uma sendo uma instância da classe Roda.
    - **def ligar(self)**: Método que liga o carro, ligando o motor e girando as rodas.

Comparação: Composição vs. Herança

Herança:

```python
class Veiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def ligar(self):
        print(f"Veículo {self.marca} {self.modelo} ligado.")

class Carro(Veiculo):
    def __init__(self, marca, modelo, portas):
        super().__init__(marca, modelo)
        self.portas = portas

    def ligar(self):
        super().ligar()
        print(f"Carro {self.marca} {self.modelo} com {self.portas} portas ligado.")

# Criando um objeto da classe Carro
meu_carro = Carro("Toyota", "Corolla", 4)

# Usando métodos da classe Carro
meu_carro.ligar()
```

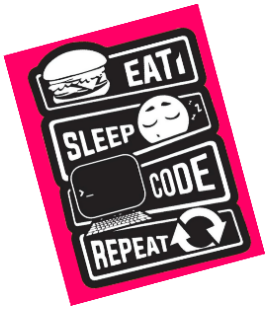
CÓDIGO-FONTE 17: EXEMPLO DE HERANÇA

Composição:

```python
class Motor:
    def __init__(self, potencia):
        self.potencia = potencia

    def ligar(self):
        print(f"Motor de {self.potencia} cavalos ligado.")

class Carro:
    def __init__(self, marca, modelo, potencia_motor):
        self.marca = marca
        self.modelo = modelo
        self.motor = Motor(potencia_motor)  # Composição: Carro tem um Motor

    def ligar(self):
        print(f"Carro {self.marca} {self.modelo} ligado.")
        self.motor.ligar()

# Criando um objeto da classe Carro
meu_carro = Carro("Toyota", "Corolla", 150)

# Usando métodos da classe Carro
meu_carro.ligar()
```

CÓDIGO-FONTE 18: EXEMPLO DE COMPOSIÇÃO

Dessa forma, podemos concluir que herança é utilizada para criar uma hierarquia de classes, reutilizando e estendendo o comportamento da superclasse. Enquanto, a composição e utilizada para construir objetos complexos a partir de componentes menores e mais simples, facilitando a modularidade e o encapsulamento. Ambas as técnicas são úteis e a escolha entre elas depende do problema específico e do design do sistema que você está desenvolvendo.



Vamos Codificar

Exercício 5. Este exercício tem como objetivo ajudar os alunos a compreenderem e aplicar o conceito de composição na Programação Orientada a Objetos em Python. O aluno deverá criar um sistema de gestão escolar utilizando composição para modelar as relações entre as classes.

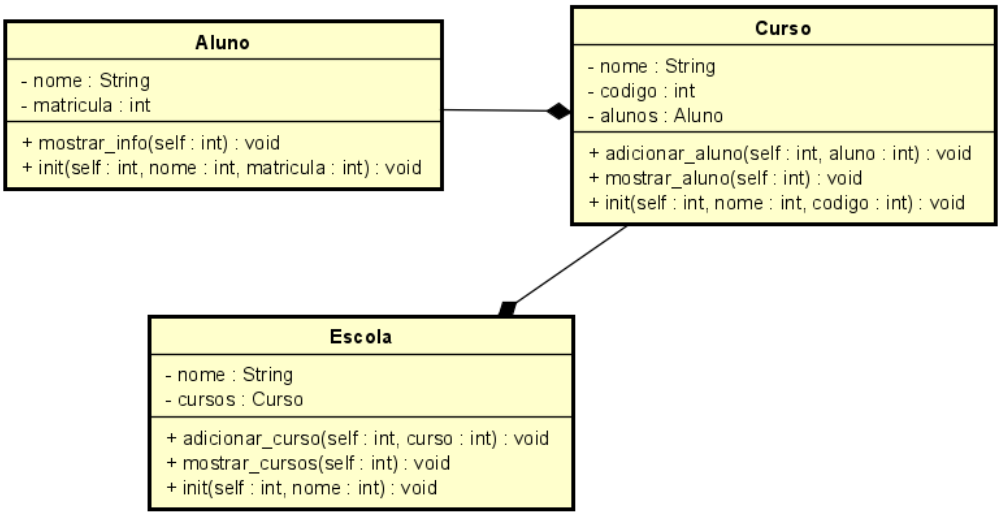


DIAGRAMA UML 3: DIAGRAMA UML DO SISTEMA DE GESTÃO ESCOLAR

Instruções:

1. Crie uma classe chamada Aluno:
    - A classe deve ter os seguintes atributos:
        - nome (nome do aluno)
        - matricula (número de matrícula do aluno)
    - A classe deve ter um método:
        - mostrar_info(self): Este método deve imprimir o nome e o número de matrícula do aluno.
2. Crie uma classe chamada Curso:
    - A classe deve ter os seguintes atributos:
        - nome (nome do curso)
        - codigo (código do curso)
        - alunos (lista de alunos matriculados no curso)
    - A classe deve ter os seguintes métodos:

- adicionar_aluno(self, aluno): Este método deve adicionar um aluno à lista de alunos.
- mostrar_alunos(self): Este método deve listar todos os alunos matriculados no curso.

3. Crie uma classe chamada Escola:
   - A classe deve ter os seguintes atributos:
     - nome (nome da escola)
     - cursos (lista de cursos oferecidos pela escola)
   - A classe deve ter os seguintes métodos:
     - adicionar_curso(self, curso): Este método deve adicionar um curso à lista de cursos.
     - mostrar_cursos(self): Este método deve listar todos os cursos oferecidos pela escola e os alunos matriculados em cada curso.
4. Crie objetos das classes Aluno, Curso e Escola e teste os métodos:
   - Crie vários alunos.
   - Crie vários cursos e adicione alunos a esses cursos.
   - Crie uma escola e adicione os cursos à escola.
   - Use os métodos implementados para exibir as informações da escola, cursos e alunos.

Tarefas para os Alunos:

- Implementar as classes Aluno, Curso e Escola conforme as instruções.
- Criar objetos das classes Aluno, Curso e Escola e usar os métodos para adicionar alunos aos cursos e cursos à escola.
- Usar os métodos mostrar_info, mostrar_alunos e mostrar_cursos para exibir as informações de alunos, cursos e da escola.
- Adicionar novos cursos e alunos para verificar a flexibilidade do sistema.

Perguntas para Reflexão:

1. Como a composição facilita a criação de relações complexas entre objetos?

2. Qual é a vantagem de usar composição em vez de herança neste exercício?
3. Como o encapsulamento é utilizado nas classes Aluno, Curso e Escola?
4. Como você pode estender este sistema para incluir novas funcionalidades, como notas dos alunos e professores para cada curso?

Você sabia?

Acoplamento é um conceito crucial na Programação Orientada a Objetos que impacta diretamente a facilidade de manutenção, reusabilidade, flexibilidade, extensibilidade, teste e depuração de um sistema. Manter um baixo acoplamento entre classes e módulos é fundamental para criar sistemas robustos e adaptáveis. A composição é frequentemente preferida sobre a herança para atingir um baixo acoplamento, promovendo uma maior modularidade e independência entre os componentes do sistema.

Polimorfismo

A palavra polimorfismo origina-se de duas palavras gregas que significam, respectivamente, muitas e forma. Algo que é polimórfico, portanto, apresenta a propriedade de assumir muitas formas. Voltando para a questão de software, temos as seguintes definições:

1. É a habilidade pela qual uma única operação ou nome de atributo pode ser definido em mais de uma classe e assumir implementações diferentes em cada uma dessas classes.
2. É a propriedade por meio da qual um atributo ou variável pode apontar para (ou manter o identificador de) objetos de diferentes classes em horas diferentes.

Exemplo:

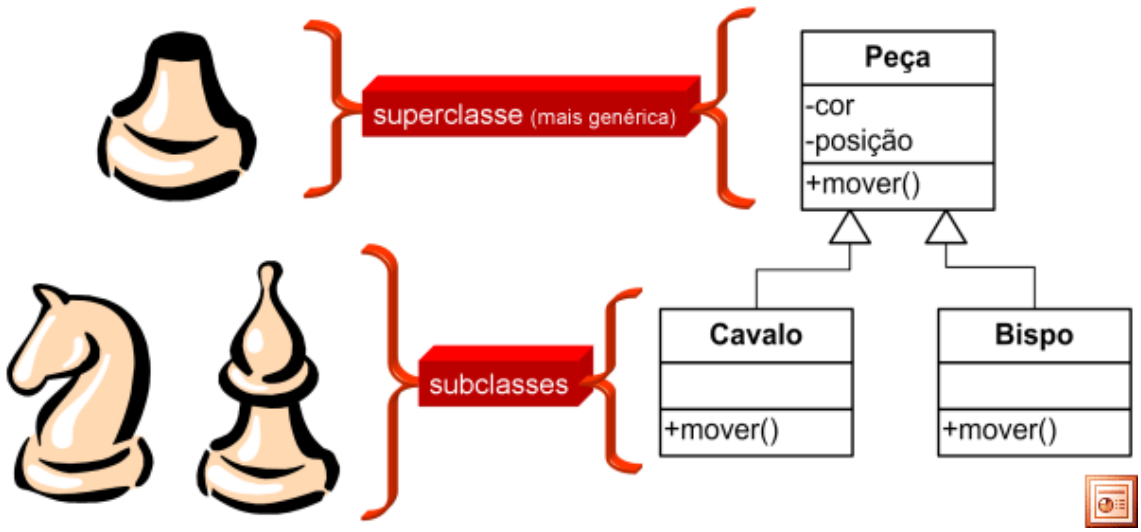


Figura VII.2.4.A

FIGURA 6 : DIAGRAMA DE CLASSES REPRESENTANDO A ESTRUTURA DE PEÇAS DO JOGO DE XADREZ.

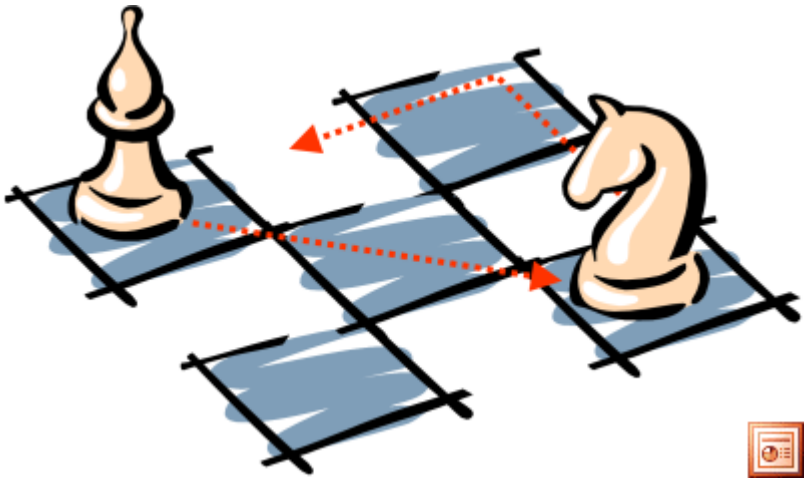


Figura VII.2.4.B

FIGURA 7: COMPORTAMENTO POLIMÓRFICO DO MÉTODO `MOVER()` APLICADO A DOIS OBJETOS DE CLASSES DIFERENTES.

O método **mover**(), por exemplo, pode atuar de forma diferente nas classes **Bispo** e **Cavalo**. Uma **Operação** é uma ação ou transformação que um objeto executa ou a que ele está sujeito. Uma implementação específica de uma operação por uma determinada classe é chamada de **Método**.

Implementação de Polimorfismo em Python

Em Python, o polimorfismo é implementado através de métodos que são comuns a diferentes classes. Mesmo que esses métodos sejam implementados de maneiras diferentes, eles podem ser chamados de maneira uniforme.

Exemplo de Polimorfismo

Vamos considerar um exemplo com classes Animal, Cachorro e Gato. Cada uma dessas classes terá um método **fazer_som**, mas com implementações diferentes.

```python
class Animal:
    def fazer_som(self):
        pass

class Cachorro(Animal):
    def fazer_som(self):
        return "O cachorro late."

class Gato(Animal):
    def fazer_som(self):
        return "O gato mia."

def descrever_som(animal):
    print(animal.fazer_som())

# Criando instâncias de Cachorro e Gato
rex = Cachorro()
mimi = Gato()

# Usando o polimorfismo
descrever_som(rex)   # Saída: O cachorro late.
descrever_som(mimi)  # Saída: O gato mia.
```

CÓDIGO-FONTE 19: EXEMPLO DE POLIMORFISMO

Explicação do Exemplo:

Definição da Classe Base Animal:

- **class Animal**: Define uma classe base chamada Animal.
- **def fazer_som(self)**: Define um método vazio fazer_som que será sobrescrito pelas subclasses.

Definição das Subclasses Cachorro e Gato:

- **class Cachorro(Animal)**: Define uma classe Cachorro que herda da classe Animal.
- d**ef fazer_som(self)**: Sobrescreve o método fazer_som para retornar "O cachorro late."
- **class Gato(Animal)**: Define uma classe Gato que herda da classe Animal.

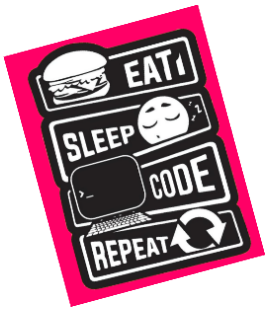
- **def fazer_som(self)**: Sobrescreve o método fazer_som para retornar "O gato mia."

Função descrever_som:

- **def descrever_som(animal)**: Define uma função que aceita um objeto animal e chama seu método fazer_som.
- **print(animal.fazer_som())**: Imprime o som que o animal faz, usando o método específico da classe do objeto passado.

Criação de Objetos e Uso do Polimorfismo:

- rex = Cachorro(): Cria uma instância da classe Cachorro.
- mimi = Gato(): Cria uma instância da classe Gato.
- descrever_som(rex): Chama a função descrever_som com a instância rex do tipo Cachorro, que imprime "O cachorro late."
- descrever_som(mimi): Chama a função descrever_som com a instância mimi do tipo Gato, que imprime "O gato mia."



Vamos Codificar

Exercício 6. Este exercício tem como objetivo ajudar os alunos a compreender e aplicar o conceito de polimorfismo na Programação Orientada a Objetos em Python. O aluno deverá criar um sistema de pagamentos que utiliza polimorfismo para processar diferentes tipos de pagamentos.

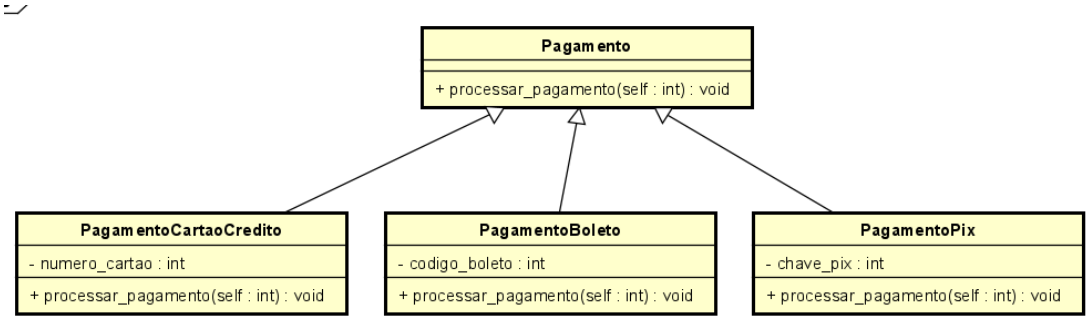


DIAGRAMA UML 4: DIAGRAMA DE CLASSE DO SISTEMA DE PAGAMENTO

Instruções:

1. Crie uma classe base chamada **Pagamento**:

   - A classe deve ter um método chamado **processar_pagamento** que não faz nada (pass).

2. Crie subclasses de Pagamento para diferentes métodos de pagamento:

   - **PagamentoCartaoCredito**
     - Deve ter um atributo **numero_cartao** para armazenar o número do cartão.
     - O método **processar_pagamento** deve exibir uma mensagem indicando que o pagamento com cartão de crédito foi processado.
   - **PagamentoBoleto**
     - Deve ter um atributo **codigo_boleto** para armazenar o código do boleto.
     - O método **processar_pagamento** deve exibir uma mensagem indicando que o pagamento com boleto foi processado.
   - **PagamentoPix**
     - Deve ter um atributo **chave_pix** para armazenar a chave Pix.
     - O método **processar_pagamento** deve exibir uma mensagem indicando que o pagamento via Pix foi processado.

3. Crie uma função processar que aceita um objeto do tipo **Pagamento** e chama o método **processar_pagamento** desse objeto:

- A função deve ser capaz de lidar com qualquer tipo de pagamento que herda da classe Pagamento.

4. Crie objetos das subclasses **PagamentoCartaoCredito, PagamentoBoleto e PagamentoPix** e teste o polimorfismo:

- Crie pelo menos um objeto de cada subclasse e passe-os para a função processar para verificar se o polimorfismo está funcionando corretamente.

Perguntas para Reflexão:

1. O que acontece se você adicionar um novo método de pagamento sem modificar a função **processar**?
2. Como o polimorfismo ajuda a manter o código flexível e extensível?
3. Qual é a diferença entre a função **processar** e os métodos **processar_pagamento** nas subclasses?
4. Como você pode garantir que todos os métodos de pagamento implementem o método **processar_pagamento** corretamente?

Exercício 7 (Em Equipe): Criar uma estrutura de classes que representem diferentes tipos de bot (BotBase, BotCadastro, BotAtualizacao). Cada bot deve ser capaz de preencher um tipo de formulário específico usando BotCity. Implementar polimorfismo para que um método genérico de preenchimento funcione para todos os tipos de bot.

Você sabia?

Sabia que o polimorfismo é como falar várias línguas?

No mundo da programação, polimorfismo permite que o mesmo método funcione de maneiras diferentes em classes distintas. Assim como uma palavra pode ter significados diferentes em várias línguas, métodos polimórficos adaptam seu comportamento conforme a classe, tornando o código mais flexível e poderoso!

Classes Abstratas em Python

Classes Abstratas são classes que não podem ser instanciadas diretamente e servem como um modelo para outras classes. Elas podem definir métodos que devem ser implementados por qualquer classe derivada (subclasse). Em Python, classes abstratas são implementadas usando o módulo abc (*Abstract Base Classes*).

Uma classe abstrata pode ter:

- **Métodos abstratos**: Métodos declarados, mas que não possuem implementação e devem ser implementados nas subclasses.
- **Métodos concretos**: Métodos que possuem implementação e podem ser usados ou sobrescritos nas subclasses.

Por que usar Classes Abstratas?

Classes abstratas são úteis quando você quer garantir que certas funcionalidades sejam implementadas em todas as subclasses. Elas fornecem uma maneira de definir uma interface comum para um conjunto de subclasses.

Exemplo de Implementação de Classe Abstrata

Passo a Passo:

1. **Importar o módulo abc:** Usamos ABC e abstractmethod do módulo abc.
2. **Definir a Classe Abstrata:** A classe deve herdar de ABC.
3. **Definir Métodos Abstratos:** Usamos o decorador @abstractmethod para definir métodos que devem ser implementados nas subclasses.

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def fazer_som(self):
        pass

    def dormir(self):
        print("Este animal está dormindo.")

class Cachorro(Animal):
    def fazer_som(self):
        print("O cachorro late.")

class Gato(Animal):
    def fazer_som(self):
        print("O gato mia.")

# Não é possível instanciar a classe abstrata diretamente
# animal = Animal()  # Isso causaria um erro

# Instanciando as subclasses
cachorro = Cachorro()
gato = Gato()

# Usando os métodos das subclasses
cachorro.fazer_som()  # Saída: O cachorro late.
gato.fazer_som()      # Saída: O gato mia.

# Usando o método concreto da classe abstrata
cachorro.dormir()     # Saída: Este animal está dormindo.
gato.dormir()         # Saída: Este animal está dormindo.
```

Explicação do Exemplo:

Definição da Classe Abstrata Animal:

- A classe Animal herda de ABC, tornando-a uma classe abstrata.
- O método fazer_som é decorado com @abstractmethod, indicando que ele deve ser implementado em qualquer subclasse de Animal.
- O método dormir é um método concreto, com uma implementação padrão.

Definição das Subclasses Cachorro e Gato:

- Ambas as subclasses herdam de Animal.
- Ambas as subclasses implementam o método fazer_som.

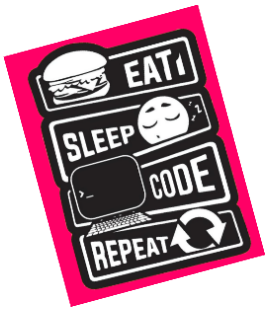
Instanciação das Subclasses:

- Instâncias das subclasses Cachorro e Gato são criadas.
- Os métodos fazer_som e dormir são chamados nas instâncias das subclasses.

Observações Importantes:

- **Instanciação Direta**: Tentar instanciar diretamente a classe Animal causará um erro, pois ela é uma classe abstrata e não pode ser instanciada.
- **Obrigatoriedade de Implementação**: Qualquer subclasse de Animal deve implementar o método fazer_som, caso contrário, um erro será gerado ao tentar instanciá-la.

Classes abstratas são uma maneira poderosa de definir interfaces comuns e garantir que as subclasses implementem métodos específicos. Elas ajudam a manter o design de código limpo e estruturado, promovendo a reutilização e a extensibilidade.



Vamos Codificar

O objetivo deste exercício é entender e aplicar o conceito de classes abstratas em Python, garantindo que suas subclasses implementem métodos essenciais para um sistema de controle de pagamentos.

Descrição:

1. **Defina uma classe abstrata**
   - Pagamento que inclui:
     - Um método abstrato processar_pagamento() que deve ser implementado por todas as subclasses.
     - Um método concreto detalhes_pagamento() que imprime "Processando pagamento via [Método]".
2. **Crie subclasses PagamentoCartao e PagamentoBoleto que herdam de Pagamento**:
   - Cada subclasse deve implementar o método processar_pagamento().
     - Para PagamentoCartao, processar_pagamento() deve imprimir "Pagamento processado com cartão."
     - Para PagamentoBoleto, processar_pagamento() deve imprimir "Pagamento processado com boleto."
3. **Crie uma função testar_pagamentos() que**:
   - Crie instâncias de PagamentoCartao e PagamentoBoleto.
   - Chame os métodos processar_pagamento() e detalhes_pagamento() para cada instância.

Instruções:

1. Complete a definição das classes PagamentoCartao e PagamentoBoleto conforme descrito.
2. Implemente a função testar_pagamentos() para criar instâncias de PagamentoCartao e PagamentoBoleto, e chamar os métodos processar_pagamento() e detalhes_pagamento() para cada instância.
3. Execute o código e verifique se as saídas são como esperado.

Você sabia?

Você sabia que classes abstratas são como contratos na programação? Elas garantem que todas as subclasses sigam um conjunto de regras, obrigando a implementação de métodos essenciais. Pense nelas como um manual de instruções que todas as subclasses devem seguir. Essa prática não só organiza o código, mas também assegura que funcionalidades críticas não sejam esquecidas! 🚀🔧

RESUMO MÓDULO

Neste módulo, exploramos os fundamentos da Programação Orientada a Objetos (POO) utilizando a linguagem Python. Começamos com a definição e a importância da POO, compreendendo como ela utiliza objetos — entidades que combinam dados e comportamentos — como unidades fundamentais de programação. Através da história da POO, vimos a evolução desde a linguagem Simula até o Python, que hoje é amplamente usado para diversas aplicações.

Avançamos para os pilares fundamentais da POO: classes, objetos, atributos e métodos. Entendemos como as classes definem a estrutura e o comportamento dos objetos, e como os objetos são instâncias dessas classes. Discutimos a diferença entre métodos de instância, métodos de classe e métodos estáticos, além de detalhar o método construtor __init__ usado para inicializar objetos.

O encapsulamento foi um ponto crucial, mostrando como proteger os dados internos de um objeto e permitir a interação controlada através de métodos. Exemplificamos o uso de getters e setters para validar e controlar o acesso aos atributos.

A herança permitiu-nos ver como criar novas classes baseadas em classes existentes, promovendo a reutilização de código. Exploramos diferentes tipos de herança e a sobrecarga de métodos, diferenciando-a da sobreposição de métodos. A composição foi apresentada como uma alternativa à herança, mostrando como criar objetos complexos a partir de componentes menores e mais simples.

Estudamos também o polimorfismo, que permite que uma única interface possa ser utilizada por diferentes classes, cada uma implementando comportamentos específicos. Abordamos o uso de classes abstratas para definir interfaces comuns e garantir a implementação de métodos específicos em subclasses.

Por fim, destacamos a importância do Method Resolution Order (MRO) na resolução de métodos em hierarquias complexas e como o tratamento de exceções é essencial para criar programas robustos.

Com o conhecimento adquirido neste módulo, você está bem preparado para avançar para o próximo tema. No próximo módulo, mergulharemos no universo de Data Science e Business Intelligence. Exploraremos como a análise de dados e a inteligência de negócios podem transformar informações em insights valiosos, complementando sua formação em programação e preparando você para desafios ainda maiores.

ATIVIDADES

EXERCÍCIOS INDIVIDUAIS

Exercício 1: Definição de Classes e Objetos

Objetivo: Praticar a definição de classes e a criação de objetos.

Enunciado: Crie uma classe Pessoa que tenha os atributos nome e idade. Implemente um método apresentar que imprima uma mensagem apresentando a pessoa. Crie dois objetos da classe Pessoa e chame o método apresentar para cada um.

Exercício 2: Encapsulamento

Objetivo: Entender e aplicar o conceito de encapsulamento, usando getters e setters para validação.

Enunciado: Crie uma classe ContaBancaria com os atributos privados saldo e titular. Implemente métodos getters e setters para o atributo saldo, garantindo que não seja possível definir um saldo negativo.

Exercício 3: Herança

Objetivo: Implementar a herança entre classes e entender como reutilizar código.

Enunciado: Crie uma classe base Animal com um método fazer_som que levanta uma exceção NotImplementedError. Crie duas subclasses Cachorro e Gato que herdam de Animal e implementam o método fazer_som.

Exercício 4: Polimorfismo

Objetivo: Demonstrar o polimorfismo, permitindo que diferentes classes implementem a mesma interface.

Enunciado: Crie uma função fazer_animais_falarem que aceita uma lista de objetos da classe Animal e chama o método fazer_som para cada um. Teste a função com uma lista contendo objetos de Cachorro e Gato.

Exercício 5: Classes Abstratas

Objetivo: Utilizar classes abstratas para definir interfaces que devem ser implementadas pelas subclasses.

Enunciado: Crie uma classe abstrata Forma com um método abstrato area e um método concreto mostrar_area que imprime a área calculada pela subclasse. Crie duas subclasses Quadrado e Circulo que implementam o método area.

Exercício 6: Composição

Objetivo: Demonstrar o uso da composição para construir objetos complexos a partir de componentes mais simples.

Enunciado: Crie uma classe Motor e uma classe Roda. Em seguida, crie uma classe Carro que tenha instâncias de Motor e Roda como atributos. Implemente métodos na classe Carro para ligar o motor e girar as rodas.

Exercício 7: Tratamento de Exceções

Objetivo: Praticar o tratamento de exceções para garantir que o programa lide com erros de forma controlada.

Enunciado: Crie uma função dividir que recebe dois números e retorna o resultado da divisão. Implemente o tratamento de exceções para lidar com divisão por zero e entrada de dados inválida.

DESAFIOS EM EQUIPE

Todos os projetos serão realizados em equipes e utilizando a metodologia ágil Scrum, conforme segue:

Parte 1: Organização do Time em SCRUM

1. **Papéis no Time:**

- **Scrum Master**: Um aluno que será responsável por facilitar as reuniões e garantir que a metodologia ágil seja seguida.
- **Product Owner**: Um aluno que representará o cliente e definirá as prioridades das funcionalidades.
- **Time de Desenvolvimento**: Restante dos alunos que serão responsáveis por desenvolver o código.

2. **Cerimônias do SCRUM:**

- **Reunião de Planejamento** (Sprint Planning): Definir quais funcionalidades serão implementadas na Sprint (ciclo de desenvolvimento).
- **Reuniões diárias** (Daily Scrum): Reunião curta (máximo de 15 minutos) onde cada membro do time irá responder:

    - O que foi feito ontem?
    - O que será feito hoje?
    - Algum impedimento?
- **Revisão da Sprint (Sprint Review):** No final da sprint, o time irá demonstrar o que foi desenvolvido e ajustar o backlog para a próxima sprint.
- **Retrospectiva da Sprint (Sprint Retrospective):** O time refletirá sobre o processo de trabalho e discutirá melhorias para a próxima sprint.

Projeto 1: Sistema de Gerenciamento de Biblioteca

Objetivo: Desenvolver um sistema de gerenciamento de biblioteca que permita o cadastro, consulta, empréstimo e devolução de livros. O projeto deve utilizar conceitos de Programação Orientada a Objetos (POO) e banco de dados.

Descrição do Projeto:

1. **Cadastro de Livros:**
    - Criação de uma classe Livro com atributos como título, autor, ISBN e status (disponível/emprestado).
    - Métodos para adicionar, editar e remover livros.
2. **Gestão de Usuários:**
    - Criação de uma classe Usuario com atributos como nome, número de usuário e histórico de empréstimos.
    - Métodos para adicionar, editar e remover usuários.
3. **Empréstimo e Devolução:**
    - Implementar métodos para registrar o empréstimo e a devolução de livros, atualizando o status do livro e o histórico do usuário.
4. **Banco de Dados:**
    - Utilizar um banco de dados SQLite para armazenar as informações dos livros e dos usuários.
    - Implementar operações CRUD (Create, Read, Update, Delete) para a interação com o banco de dados.

Desafio:

Cada equipe deve projetar e implementar o sistema, garantindo a integridade dos dados e a usabilidade do sistema.

Projeto 2: Sistema de Gerenciamento de Tarefas

Objetivo: Desenvolver um sistema de gerenciamento de tarefas que permita a criação, edição e conclusão de tarefas. O projeto deve utilizar conceitos de POO, manipulação de arquivos e estruturas de dados.

Descrição do Projeto:

1. **Cadastro de Tarefas:**
    - Criação de uma classe Tarefa com atributos como descrição, data de criação, prazo e status (pendente/concluída).
    - Métodos para adicionar, editar e remover tarefas.
2. **Gestão de Usuários:**
    - Criação de uma classe Usuario com atributos como nome e lista de tarefas.
    - Métodos para adicionar, editar e remover usuários.
3. **Prioridade e Notificação:**
    - Implementar um sistema de prioridade para as tarefas.
    - Implementar um método que notifique o usuário sobre tarefas pendentes ou próximas do prazo.
4. **Manipulação de Arquivos:**
    - Utilizar arquivos JSON para armazenar e carregar dados de tarefas e usuários.
    - Implementar métodos para salvar e carregar dados dos arquivos.

Desafio:

Cada equipe deve projetar e implementar o sistema, garantindo a funcionalidade e a integridade dos dados armazenados em arquivos.

Projeto 3: Sistema de Análise de Vendas

Objetivo: Desenvolver um sistema de análise de vendas que permita a importação de dados, análise de tendências e geração de relatórios. O projeto deve utilizar conceitos de POO, manipulação de arquivos e estruturas de dados.

Descrição do Projeto:

1. **Importação de Dados:**
    - Criação de uma classe Venda com atributos como data, produto, quantidade e valor.
    - Métodos para importar dados de vendas a partir de arquivos CSV.
2. **Análise de Dados:**
    - Implementar métodos para calcular métricas como total de vendas, produtos mais vendidos e receita total.
    - Utilizar estruturas de dados como listas e dicionários para armazenar e processar os dados de vendas.
3. **Geração de Relatórios:**
    - Implementar métodos para gerar relatórios em formato CSV ou PDF.
    - Relatórios devem incluir análises e gráficos de tendências de vendas.
4. **Interface de Usuário:**
    - Desenvolver uma interface de usuário simples para permitir a interação com o sistema.
    - Implementar funcionalidades para importar dados, visualizar análises e gerar relatórios.

Desafio:

Cada equipe deve projetar e implementar o sistema, garantindo a precisão das análises e a usabilidade da interface.

Projeto 4: Sistema de Controle de Estoque

Objetivo: Desenvolver um sistema de controle de estoque que permita a adição, remoção e atualização de itens de estoque. O projeto deve utilizar conceitos de POO, banco de dados e estruturas de dados.

Descrição do Projeto:

1. **Gestão de Itens:**
   - Criação de uma classe Item com atributos como nome, quantidade e preço.
   - Métodos para adicionar, editar e remover itens do estoque.
2. **Gestão de Categorias:**
   - Criação de uma classe Categoria com atributos como nome e lista de itens.
   - Métodos para adicionar, editar e remover categorias.
3. **Banco de Dados:**
   - Utilizar um banco de dados SQLite para armazenar as informações dos itens e das categorias.
   - Implementar operações CRUD para a interação com o banco de dados.
4. **Relatórios de Estoque:**
   - Implementar métodos para gerar relatórios de estoque, incluindo itens em falta e itens com baixa quantidade.
   - Relatórios devem ser exportáveis em formato CSV.

Desafio:

Cada equipe deve projetar e implementar o sistema, garantindo a integridade dos dados e a eficiência do controle de estoque.

Projeto 5: Sistema de Gestão de Projetos

Objetivo da Atividade:

Os alunos irão desenvolver um Sistema de Gestão de Projetos robusto utilizando Programação Orientada a Objetos (POO) em Python. A atividade deve seguir a metodologia ágil, utilizando o Scrum para organizar o desenvolvimento em sprints com entregas contínuas e incrementais. A ideia é integrar conceitos avançados de POO, como design patterns, herança múltipla, e polimorfismo, enquanto se implementa um ciclo de vida completo de gestão de projetos.

Contexto: Sistema de Gestão de Projetos

O sistema será utilizado por uma empresa para gerenciar diferentes projetos, alocando recursos, definindo tarefas, e monitorando o progresso dos projetos. A plataforma permitirá:

- Cadastro de Projetos, incluindo detalhes como descrição, data de início e data de fim.
- Criação de Tarefas associadas aos projetos, com prazos e responsáveis.
- Atribuição de Recursos Humanos (colaboradores) a tarefas específicas.
- Controle de Status e Progresso de tarefas e projetos.
- Relatórios Dinâmicos sobre o andamento dos projetos e eficiência da equipe.

Parte 1: Requisitos do Sistema

Os requisitos funcionais do sistema de gestão de projetos estarão organizados em um backlog, com as user stories priorizadas:

Funcionalidades principais (Backlog):

1. Como gerente, quero criar um projeto para gerenciar as atividades de uma iniciativa específica.
2. Como gerente, quero adicionar tarefas a um projeto para que as etapas do projeto possam ser organizadas.
3. Como gerente, quero atribuir colaboradores a tarefas para distribuir o trabalho entre a equipe.
4. Como colaborador, quero atualizar o status da minha tarefa para manter o gerente informado sobre o progresso.
5. Como gerente, quero gerar relatórios de progresso do projeto para acompanhar o desempenho da equipe e as entregas.

Funcionalidades avançadas:

1. Como gerente, quero calcular o percentual de conclusão do projeto com base nas tarefas concluídas.
2. Como gerente, quero visualizar gráficos de Gantt para acompanhar visualmente o cronograma do projeto.
3. Como gerente, quero um sistema de notificações automáticas para avisar sobre prazos e tarefas atrasadas.
4. Como administrador, quero gerenciar usuários e permissões para controlar quem pode visualizar e editar os projetos.

Parte 2: Estrutura do Sistema com POO (Python)

Design Patterns e Conceitos Avançados

A estrutura do sistema será projetada utilizando padrões de design como Factory Method, Observer, e Strategy para criar uma arquitetura escalável e de fácil manutenção.

1. Classes e Atributos Avançados

A seguir um exemplo de Diagrama de Classes Simplificado, a equipe deverá implementar novas funcionalidades:

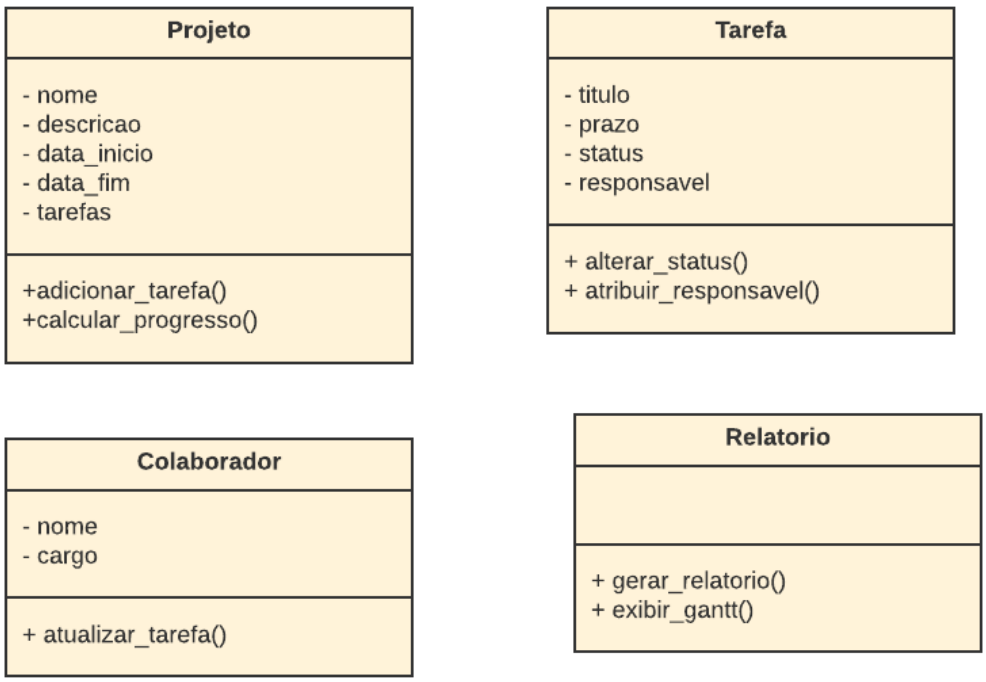


FIGURA 8 – CLASSES DO PROJETO

2. Implementação de Padrões de Design

1. **Factory Method** para criar diferentes tipos de projetos (por exemplo, Projetos de TI, Projetos de Marketing).
2. **Observer Pattern** para notificações automáticas quando uma tarefa for atualizada ou estiver perto do prazo.
3. **Strategy Pattern** para implementar diferentes formas de calcular o progresso do projeto (por exemplo, peso de tarefas, prazo).

Projeto 6: Sistema de e-commerce para vendas de produtos

Objetivo da Atividade:

Os alunos deverão implementar um sistema simples de e-commerce utilizando os princípios de Programação Orientada a Objetos (POO) em Python, enquanto seguem uma abordagem de desenvolvimento ágil baseada no Scrum. O objetivo é aprender a estruturar um sistema com classes e métodos, além de experimentar a entrega contínua e o trabalho colaborativo de equipes.

Contexto: Sistema de E-commerce para Vendas de Produtos

A aplicação permitirá que usuários:

- Cadastrarem produtos.
- Busquem produtos.
- Adicionem produtos ao carrinho.
- Finalizem a compra.

A equipe trabalhará em Sprints para implementar as funcionalidades gradualmente, com entregas incrementais de software funcional.

Parte 1: Requisitos do Sistema de E-commerce

Funcionalidades principais (Backlog):

1. Como cliente, quero adicionar um produto ao catálogo, para disponibilizá-lo para venda.
2. Como cliente, quero pesquisar produtos por nome ou categoria, para encontrar o que estou procurando.
3. Como cliente, quero adicionar produtos ao carrinho, para que eu possa preparar minha compra.
4. Como cliente, quero ver o preço total do meu carrinho, para saber quanto vou gastar antes de finalizar a compra.
5. Como cliente, quero remover produtos do carrinho, para ajustar minha compra antes do pagamento.
6. Como cliente, quero finalizar a compra, para receber a confirmação e o resumo da compra.

Parte 2: Implementação com Programação Orientada a Objetos (POO)

1. Classes e Atributos

A seguir um exemplo de Diagrama de Classes Simplificado, a equipe deverá implementar novas funcionalidades:

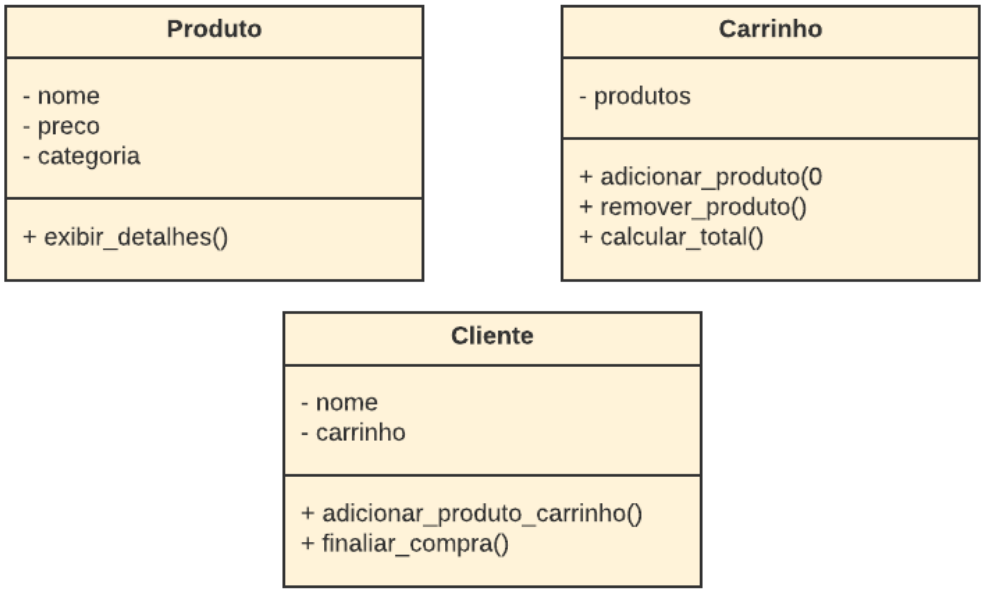


FIGURA 9 – DIGRAMAS DE CLASSE PROJETO

Parte 4: Desafios Extras

Algumas funcionalidades extras podem ser adicionadas:

- Sistema de Estoque: Adicionar controle de estoque para cada produto.
- Descontos e Cupons: Implementar descontos em produtos ou cupons para o carrinho.
- Pagamentos: Simular diferentes métodos de pagamento.

DESAFIOS DE AUTOMAÇÃO COM POO

Desafio 1. Desenvolver um sistema de cadastro e atualização de produtos. Criar uma classe Produto e uma classe GestorProdutos para gerenciar o estoque. Utilizar o BotCity para automatizar o preenchimento e a atualização dos dados de produtos em um sistema online.

Desafio 2. Analisar um código fornecido de um bot automatizador e propor melhorias usando POO (dividir responsabilidades em classes, adicionar encapsulamento, etc.). Refatorar o código e explicar as mudanças.

Desafio 3. Criar uma classe LogBot com métodos estáticos para registrar a execução de bots. Usar essa classe para registrar todas as atividades de um bot que automatiza o preenchimento de um formulário.

Desafio 4. Planejar e desenvolver um sistema automatizado completo utilizando BotCity e POO. O projeto deve incluir uma hierarquia de classes, encapsulamento e polimorfismo, além de funções de logging e tratamento de erros.

Desafio 5. Cada aluno deve escrever um relatório refletindo sobre o projeto desenvolvido, os conceitos de POO aplicados e como a automação com BotCity facilitou a resolução de problemas.

ANEXO

DECORATOR

Decorators em Python são uma maneira poderosa e flexível de modificar o comportamento de funções ou métodos. Eles permitem que você "envolva" uma função ou método com outra função, adicionando funcionalidade antes ou depois da execução da função original, sem modificar diretamente o código da função original.

Conceito de Decorator

Um decorator é essencialmente uma função que recebe outra função como argumento e retorna uma nova função que geralmente estende o comportamento da função original.

Exemplos de Decorators em Python

Exemplo 1: Decorator Simples

Vamos começar com um exemplo básico de um decorator que imprime uma mensagem antes e depois de chamar a função original.

```python
def meu_decorator(func):
    def wrapper():
        print("Algo antes da função.")
        func()
        print("Algo depois da função.")
    return wrapper

@meu_decorator
def saudacao():
    print("Olá, mundo!")

# Chamando a função decorada
saudacao()
```

Explicação:

1. **Definindo o Decorator:**

- `meu_decorator(func)`: Define um decorator que recebe uma função `func` como argumento.
- `def wrapper()`: Define uma função `wrapper` dentro do decorator que adiciona comportamento adicional.
- `print("Algo antes da função.")`: Imprime uma mensagem antes de chamar a função original.
- `func()`: Chama a função original.
- `print("Algo depois da função.")`: Imprime uma mensagem depois de chamar a função original.
- `return wrapper`: Retorna a função `wrapper`.

2. **Usando o Decorator:**
   - `@meu_decorator`: Aplica o decorator `meu_decorator` à função `saudacao`.
3. **Resultado:**
   - Quando `saudacao()` é chamada, a saída será:

```
Algo antes da função.
Olá, mundo!
Algo depois da função.
```

Exemplo 2: Decorator com Argumentos

Decorators também podem ser usados para modificar funções que aceitam argumentos.

```
def meu_decorator(func):
    def wrapper(*args, **kwargs):
        print("Algo antes da função.")
        resultado = func(*args, **kwargs)
        print("Algo depois da função.")
        return resultado
    return wrapper

@meu_decorator
def saudacao(nome):
    print(f"Olá, {nome}!")

# Chamando a função decorada
saudacao("Alice")
```

Explicação:

1. **Definindo o Decorator:**
   - `def wrapper(*args, **kwargs)`: Usa `*args` e `**kwargs` para aceitar qualquer número de argumentos posicionais e nomeados.

- `resultado = func(*args, **kwargs)`: Chama a função original com os argumentos recebidos e armazena o resultado.
- `return resultado`: Retorna o resultado da função original.

2. **Resultado:**
   - Quando `saudacao("Alice")` é chamada, a saída será:

```
Algo antes da função.
Olá, Alice!
Algo depois da função.
```

Exemplo 3: Decorator com Métodos de Classe

Decorators podem ser aplicados a métodos de classe usando `@classmethod` e `@staticmethod`.

```python
class MinhaClasse:
    @staticmethod
    def metodo_estatico():
        print("Este é um método estático.")

    @classmethod
    def metodo_de_classe(cls):
        print(f"Este é um método de classe da classe {cls.__name__}.")

# Chamando métodos da classe
MinhaClasse.metodo_estatico()
MinhaClasse.metodo_de_classe()
```

Explicação:

1. **`@staticmethod`:**
   - `@staticmethod` define um método estático que não recebe implicitamente o primeiro argumento (que seria a instância da classe).
   - `metodo_estatico()`: Método que pode ser chamado diretamente pela classe sem criar uma instância.
2. **`@classmethod`:**
   - `@classmethod` define um método de classe que recebe a própria classe como primeiro argumento.
   - `metodo_de_classe(cls)`: Método que pode ser chamado diretamente pela classe e recebe a classe como argumento (`cls`).

Decorators Comuns em Python

1. **`@property`:**

- Transforma o método de uma classe em um atributo que pode ser acessado como se fosse um atributo comum, mas com a lógica de um método.

2. **`@staticmethod` e `@classmethod`:**
   - `@staticmethod`: Define um método estático.
   - `@classmethod`: Define um método de classe.
3. **`@functools.wraps`:**
   - É um decorator usado para preservar as informações da função original quando um decorator é aplicado.

```python
from functools import wraps

def meu_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Algo antes da função.")
        resultado = func(*args, **kwargs)
        print("Algo depois da função.")
        return resultado
    return wrapper
```

A utilização de Decorators tem as seguintes vantagens:

- **Reusabilidade de Código**: Decoradores permitem reutilizar lógica comum a várias funções, mantendo o código DRY (Don't Repeat Yourself).
- **Separação de Preocupações**: Eles ajudam a separar as preocupações, mantendo a lógica de negócios separada da lógica de controle ou de monitoramento.
- **Sintaxe Clara e Expressiva**: A sintaxe dos decoradores (@) torna o código mais legível e expressivo, facilitando o entendimento de que uma função está sendo modificada ou aprimorada.

Decorators são uma ferramenta fantástica em Python, oferecendo uma forma poderosa de adicionar funcionalidades às funções de maneira limpa e reutilizável. Seja para adicionar logs, verificar permissões, ou simplesmente modificar o comportamento de uma função sem alterar seu código original. Com um pouco de prática, você pode começar a tirar proveito dos benefícios que eles oferecem, tornando seu código mais modular, eficiente e fácil de manter.

MRO (METHOD RESOLUTION ORDER)

Method Resolution Order (MRO) é o mecanismo que Python usa para determinar a ordem na qual os métodos são chamados em uma hierarquia de herança. Em outras palavras, MRO é a ordem em que as classes são verificadas para encontrar o método ou atributo que está sendo acessado. O MRO é particularmente importante em herança múltipla, onde uma classe pode herdar de várias outras classes.

Uso do MRO em Python

O MRO é utilizado para resolver conflitos em herança múltipla, garantindo que cada método ou atributo seja chamado na ordem correta. Python utiliza o algoritmo C3 Linearization (ou C3 superclass linearization) para calcular o MRO de uma classe.

Como Verificar o MRO

Você pode usar o método __mro__ ou a função mro() para verificar o MRO de uma classe. Exemplo de Uso do MRO:

```
class A:
    def who_am_i(self):
        print("Eu sou a classe A")

class B(A):
    def who_am_i(self):
        print("Eu sou a classe B")

class C(A):
    def who_am_i(self):
        print("Eu sou a classe C")

class D(B, C):
    pass

d = D()
d.who_am_i()

print(D.__mro__)
print(D.mro())
```

CÓDIGO-FONTE 20: EXEMPLO SIMPLES DE USO DO MRO

Explicação do Exemplo:

- **Definição das Classes:**
  - class A: Define a classe base A com o método who_am_i.
  - class B(A): Herda de A e sobrescreve o método who_am_i.
  - class C(A): Herda de A e sobrescreve o método who_am_i.
  - class D(B, C): Herda de B e C, criando uma herança múltipla.
- **Instanciação e Chamada de Método:**
  - d = D(): Cria uma instância da classe D.
  - d.who_am_i(): Chama o método who_am_i na instância d. De acordo com o MRO, ele segue a ordem B -> C -> A, então o método da classe B é chamado, resultando em "Eu sou a classe B".
- **Verificação do MRO:**
  - print(D.__mro__): Exibe o MRO da classe D como uma tupla.
  - print(D.mro()): Exibe o MRO da classe D como uma lista.

Saída do código:

```
Eu sou a classe B
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>,
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>,
```

O MRO é uma característica essencial do Python que garante que os métodos e atributos sejam resolvidos na ordem correta em uma hierarquia de herança múltipla. Utilizando o algoritmo C3 Linearization, Python assegura uma ordem consistente e lógica para a resolução de métodos, evitando ambiguidades e conflitos em hierarquias complexas de herança. Entender o MRO é crucial para trabalhar eficazmente com herança múltipla em Python.

Tratamento de Exceções em Python

O tratamento de exceções em Python permite que você lide com erros e condições excepcionais em seu código de forma controlada. Em vez de seu programa terminar abruptamente quando encontra um erro, você pode capturar e tratar o erro, fornecendo uma resposta adequada.

Sintaxe Básica

A estrutura básica para o tratamento de exceções em Python envolve as palavras-chave try, except, else e finally:

1. **try**: O bloco de código onde você coloca as instruções que podem gerar uma exceção.
2. **except**: O bloco de código que é executado se uma exceção for levantada no bloco try.
3. **else**: (Opcional) O bloco de código que é executado se nenhuma exceção for levantada no bloco try.
4. **finally**: (Opcional) O bloco de código que é sempre executado, quer uma exceção tenha sido levantada ou não.

Exemplo de Implementação

```
try:
    x = int(input("Digite um número: "))
    resultado = 10 / x
    print(f"O resultado é {resultado}")
except ZeroDivisionError:
    print("Erro: Divisão por zero não é permitida.")
except ValueError:
    print("Erro: Entrada inválida. Por favor, digite um número inteiro.")
else:
    print("Nenhuma exceção foi levantada.")
finally:
    print("Bloco finally executado.")
```

CÓDIGO-FONTE 21: EXEMPLO SIMPLES DE TRATAMENTO DE EXCEÇÃO

Explicação do Exemplo:

- **Bloco try:**
  - O código dentro do bloco try tenta converter a entrada do usuário para um número inteiro e, em seguida, realiza uma divisão.
- **Bloco except:**
  - O primeiro bloco except captura a exceção ZeroDivisionError, que ocorre se o usuário digitar zero.
  - O segundo bloco except captura a exceção ValueError, que ocorre se o usuário digitar algo que não possa ser convertido para um número inteiro.
- **Bloco else:**
  - O bloco else é executado apenas se nenhum erro for levantado no bloco try.
- **Bloco finally:**
  - O bloco finally é sempre executado, independentemente de uma exceção ter sido levantada ou não. Ele é útil para liberar recursos ou realizar ações de limpeza.

Tratamento de Múltiplas Exceções

Você pode capturar várias exceções específicas separando-as com vírgulas:

```
try:
    y = int(input("Digite outro número: "))
    resultado = 10 / y
    print(f"O resultado é {resultado}")
except (ZeroDivisionError, ValueError) as e:
    print(f"Erro: {e}")
```

CÓDIGO-FONTE 22:EXEMPLO DE TRATAMENTO DE MÚLTIPLAS EXCEÇÕES

Levantando Exceções

Você pode levantar (lançar) exceções usando a palavra-chave raise:

```
def verificar_numero_positivo(num):
    if num < 0:
        raise ValueError("O número deve ser positivo")
    return num

try:
    numero = verificar_numero_positivo(-5)
except ValueError as e:
    print(f"Erro: {e}")
```

CÓDIGO-FONTE 23: EXEMPLO DE LEVANTAMENTO DE EXCEÇÕES

Exemplo com Classes Personalizadas

Você também pode definir suas próprias exceções personalizadas criando classes que herdam de Exception:

```
class MinhaExcecaoPersonalizada(Exception):
    pass

def verificar_numero_par(num):
    if num % 2 != 0:
        raise MinhaExcecaoPersonalizada("O número deve ser par")
    return num

try:
    numero = verificar_numero_par(3)
except MinhaExcecaoPersonalizada as e:
    print(f"Erro: {e}")
```

CÓDIGO-FONTE 24: EXEMPLO DE CLASSES PERSONALIZADAS

Por Que Usar Tratamento de Exceções?

- **Robustez do Código:**

- Capturar e tratar exceções permite que seu programa lide com erros de maneira controlada, em vez de falhar inesperadamente.
- Exemplos incluem divisão por zero, erros de tipo, erros de entrada/saída, entre outros.

- **Melhor Experiência do Usuário:**
  - Fornecer mensagens de erro claras e opções para correção melhora a experiência do usuário.
  - Em um ambiente de produção, o tratamento de exceções pode prevenir a exposição de mensagens de erro técnicas para o usuário final.
- **Manutenção e Debugging:**
  - Facilita a localização de bugs, já que você pode registrar ou exibir mensagens de erro detalhadas.
  - Permite que desenvolvedores tratem situações inesperadas sem interromper o fluxo do programa.

O tratamento de exceções em Python é uma ferramenta poderosa para lidar com erros e situações inesperadas de maneira controlada. Usando try, except, else e finally, você pode garantir que seu código lide com exceções de forma robusta, melhorando a confiabilidade e a robustez de seus programas.

REFERÊNCIAS BIBLIOGRÁFICAS

Bibliografia Básica:

RAMALHO, Luciano. **Python Fluente: Programação clara, concisa e eficaz***. São Paulo: Novatec, 2015.*

MARTINS, Luiz Eduardo. **Python: Programação Orientada a Objetos***. 2ª ed. São Paulo: Novatec, 2016.*

NILO, Fernando Anselmo. **Python para Desenvolvedores: Um guia de referência para a linguagem Python***. São Paulo: Novatec, 2009.*

Bibliografia Complementar:

SEVERANCE, Charles. **Python for Everybody: Exploring Data in Python 3***. Charles Severance, 2016.*

JÚNIOR, Osvaldo Santana. ***Desenvolvimento Web com Python e Django****. São Paulo: Novatec, 2019.*

MASSIRONI, Ricardo. ***Python e Django: Desenvolvimento Web Moderno e Ágil****. São Paulo: Novatec, 2020.*

SOUZA, Allan; NUNES, Igor. ***Introdução à Programação com Python: Algoritmos e Lógica de Programação para Iniciantes****. São Paulo: Novatec, 2020.*

MENEZES, Daniel de Oliveira. ***Python para Engenharia: Programação Orientada a Objetos com Aplicações em Engenharia****. São Paulo: LTC, 2019.*

GLOSSÁRIO

Agregação: Um tipo especial de associação que representa uma relação "tem um" mas com uma vida independente das classes associadas.

Atributo: Variáveis que pertencem a uma classe e são usadas para armazenar o estado de um objeto.

Classe Abstrata: Uma classe que não pode ser instanciada diretamente e serve como um modelo para outras classes.

Classe: Um modelo ou blueprint que define a estrutura e comportamento de objetos. Uma classe encapsula dados (atributos) e métodos que operam sobre esses dados.

Composição: Um conceito onde uma classe é composta de uma ou mais instâncias de outras classes. Indica uma relação "tem um" (has-a).

Construtor: Um método especial usado para inicializar objetos de uma classe. Em Python, é o método __init__.

Decorator: São, essencialmente, funções que adicionam funcionalidades a outras funções ou métodos sem alterá-los permanentemente. Eles "embrulham" a função original, permitindo executar código antes ou depois dela, e então retornam uma nova função com comportamento modificado.

Destrutor: Um método especial chamado quando um objeto é destruído. Em Python, é o método __del__.

Encapsulamento: O princípio de esconder os detalhes internos de uma classe e expor apenas o necessário através de métodos públicos.

Herança: O mecanismo pelo qual uma classe (subclasse) pode herdar atributos e métodos de outra classe (superclasse).

Interface: Um conjunto de métodos que uma classe deve implementar. Python não tem suporte direto a interfaces como outras linguagens, mas pode usar classes abstratas.

Método de Classe: Um método que recebe a classe como primeiro argumento em vez da instância. Usa o decorador @classmethod.

Método Estático: Um método que pertence à classe em vez de uma instância e não pode acessar ou modificar o estado da instância.

Método: Funções definidas dentro de uma classe que descrevem os comportamentos dos objetos da classe.

MRO (Method Resolution Order): A ordem na qual Python resolve métodos e atributos em uma hierarquia de herança. É especialmente importante em herança múltipla.

Namespace: Um espaço de nomes onde os nomes são mapeados para objetos. Em Python, cada função, classe e módulo define seu próprio namespace.

Objeto: Uma instância de uma classe. Um objeto é criado a partir de uma classe e pode ter seu próprio estado e comportamento.

Polimorfismo: A habilidade de diferentes classes utilizarem a mesma interface. Um método pode se comportar de diferentes maneiras dependendo do objeto que o invoca.

Sobrecarga de Métodos: A capacidade de definir múltiplos métodos com o mesmo nome, mas diferentes assinaturas (número e/ou tipos de parâmetros). Não é diretamente suportada em Python.

Sobrescrita de Métodos (Override): Quando uma subclasse fornece uma implementação específica de um método que já é definido em sua superclasse.