

RODRIGO VIANNINI

POSTECH

MACHINE LEARNING ENGINEERING

PYTHON PARA ML E IA

AULA 03

SUMÁRIO

O QUE VEM POR AÍ?	3
HANDS ON	4
SAIBA MAIS.....	6
O QUE VOCÊ VIU NESTA AULA?	15
REFERÊNCIAS.....	16

EMSE

O QUE VEM POR AÍ?

É hora de avançar nos conceitos fundamentais: vamos falar sobre programação orientada à objeto (POO).

Nesta aula iremos entender os conceitos dos quatro pilares da POO, abstração, herança, encapsulamento e polimorfismo, aplicados no código através de classes, objetos, atributos e métodos, entre outros.

EXEMPLO

HANDS ON

Nesta aula vamos direto à prática; mas, antes, falaremos dos conceitos fundamentais da programação orientada a objeto (POO): abstração, herança, encapsulamento e polimorfismo. Eles serão aplicados de maneira tangível, utilizando classes, objetos, atributos e métodos para construir soluções eficientes e reutilizáveis.

Abstração

Ao modelar objetos da vida real, focamos apenas nos detalhes relevantes para o contexto, ignorando complexidades desnecessárias. A abstração permite a representação eficiente das entidades, simplificando o desenvolvimento e facilitando a compreensão do código.

Herança

Permite criar classes aproveitando características das classes existentes.

Encapsulamento

Protege os detalhes internos de uma classe, limitando o acesso a certos atributos ou métodos.

Polimorfismo

Permite que objetos diferentes sejam tratados de maneira uniforme.

Exemplo:

Carro.py

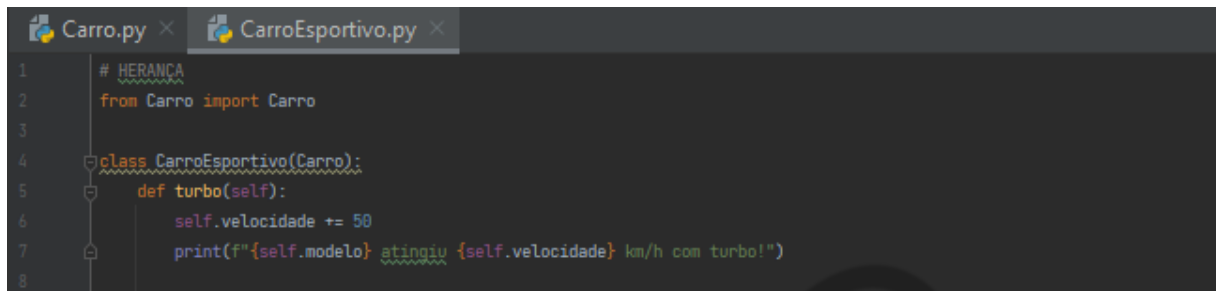


```
1 # ABSTRAÇÃO | ENCAPSULAMENTO
2 class Carro:
3     def __init__(self, modelo, cor):
4         self.modelo = modelo
5         self.cor = cor
6         self.velocidade = 0
7
8     def acelerar(self, incremento):
9         self.velocidade += incremento
10        print(f"{self.modelo} acelerou para {self.velocidade} km/h.")
11
```

Figura 1 – Exemplo de código-fonte Python (abstração e encapsulamento)

Fonte: Elaborado pelo autor (2024)

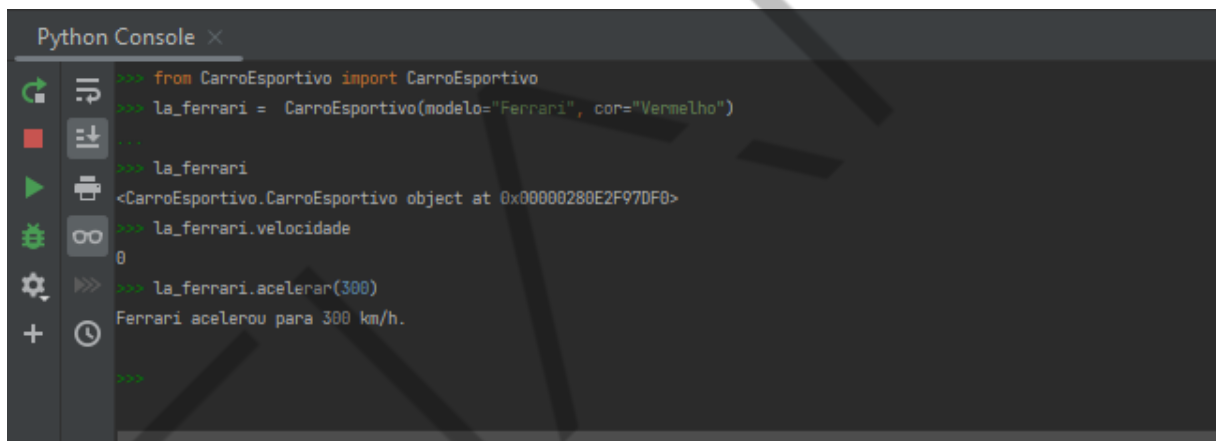
CarroEsportivo.py



```
1 # HERANCA
2 from Carro import Carro
3
4 class CarroEsportivo(Carro):
5     def turbo(self):
6         self.velocidade += 50
7         print(f"{self.modelo} atingiu {self.velocidade} km/h com turbo!")
8
```

Figura 2 – Exemplo de código-fonte Python (herança)
Fonte: Elaborado pelo autor (2024)

Execução do código



```
Python Console x
>>> from CarroEsportivo import CarroEsportivo
>>> la_ferrari = CarroEsportivo(modelo="Ferrari", cor="Vermelho")
>>> la_ferrari
<CarroEsportivo.CarroEsportivo object at 0x00000280E2F97DF0>
>>> la_ferrari.velocidade
0
>>> la_ferrari.acelerar(300)
Ferrari acelerou para 300 km/h.
>>>
```

Figura 3 – Exemplo de código-fonte Python (Python console)
Fonte: Elaborado pelo autor (2024)

SAIBA MAIS

Lembrando que, antes de nos aprofundarmos em POO, temos que garantir que os conceitos fundamentais foram absorvidos; para isso, precisamos praticar! Evite a simples replicação de código, busque a compreensão profunda de cada linha.

A seguir iremos explorar conceitos avançado de POO, proporcionando uma compreensão mais profunda e prática desta metodologia.

Métodos especiais (magic methods)

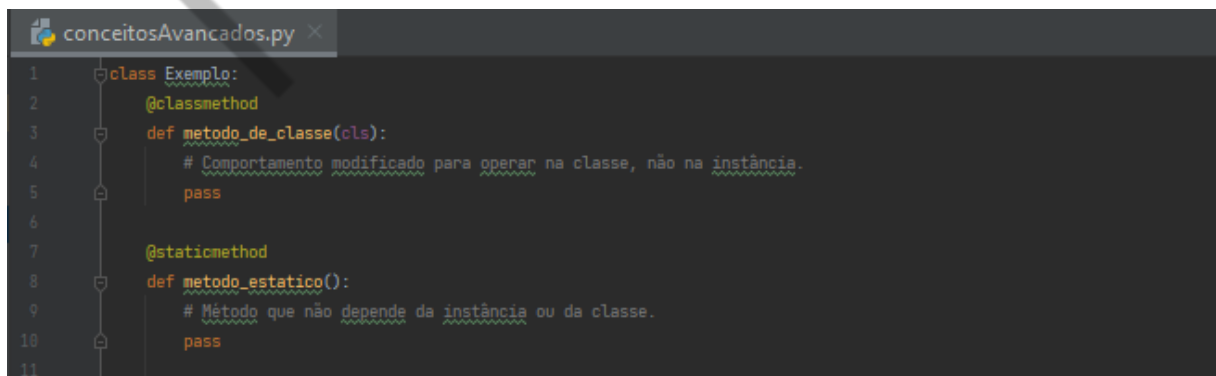
Os métodos especiais, também conhecidos como “dunder” (double underscore), proporcionam uma maneira poderosa de personalizar o comportamento das classes. Notem a sintaxe e suas respectivas funções:

<code>__init__:</code>	Inicialização de objetos.
<code>__str__</code> e <code>__repr__:</code>	Representação de texto e debug.
<code>__getitem__</code> e <code>__setitem__:</code>	Manipulação de índices como em listas.

Decoradores em Métodos (Decorators)

Os decoradores podem ser aplicados a métodos de classes para modificar seu comportamento.

Exemplo:



```
1 class Exemplo:
2     @classmethod
3     def metodo_de_classe(cls):
4         # Comportamento modificado para operar na classe, não na instância.
5         pass
6
7     @staticmethod
8     def metodo_estatico():
9         # Método que não depende da instância ou da classe.
10        pass
11
```

Figura 4 – Exemplo de código-fonte Python (classmethod e staticmethod)
Fonte: Elaborado pelo autor (2024)

Composição e agregação

A composição é quando um objeto faz parte de outro e não há sentido em existir independentemente. Se o objeto pai for excluído, todos os seus filhos devem ser excluídos; esta relação pode ser considerada forte e íntima.

Exemplo:



```
1 class Motor:
2     def __init__(self, tipo):
3         self.tipo = tipo
4
5 class Carro:
6     def __init__(self):
7         # O carro possui um motor
8         self.motor = Motor(tipo="V8")
9
10    def ligar(self):
11        print("Carro ligado com um motor", self.motor.tipo)
12
13    # Exemplo
14    meu_carro = Carro()
15    meu_carro.ligar()
16
```

Figura 5 – Exemplo de código-fonte Python (composição de classes)
Fonte: Elaborado pelo autor (2024)

Neste exemplo, temos a classe “Carro” que possui um objeto “Motor”. O motor é criado quando o carro é criado; se o carro deixar de existir, o motor também deixará de existir.

Agregação é quando os objetos são independentes, ou seja, podem existir fora do objeto pai. Eles se relacionam, mas cada um pode existir independentemente do outro, ou seja: se o objeto pai deixar de existir, os objetos filhos ainda podem existir.

Exemplo:



```
conceitosAvancados.py
4 class Turma:
5     def __init__(self):
6         self.alunos = []
7
8     def adicionar_aluno(self, aluno):
9         self.alunos.append(aluno)
10
11 # Exemplo de uso
12 aluno1 = Aluno(nome="João")
13 aluno2 = Aluno(nome="Maria")
14
15 turma = Turma()
16 turma.adicionar_aluno(aluno1)
17 turma.adicionar_aluno(aluno2)
18
```

Figura 6 – Exemplo de código-fonte Python (input de dados)
Fonte: Elaborado pelo autor (2024)

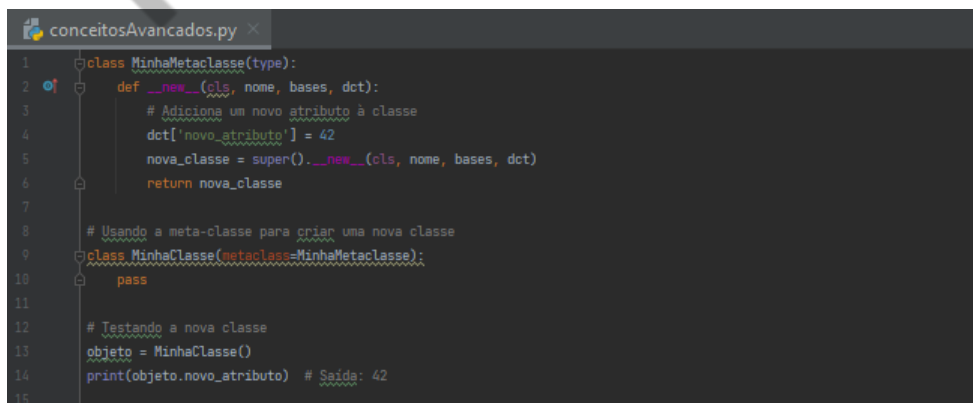
Neste exemplo, temos a classe “Turma” e uma lista de alunos. Desta forma, se a turma deixar de existir os alunos continuarão existindo.

A composição e a agregação oferecem maneiras diferentes de organizar a estrutura dos objetos de uma aplicação. A composição é mais forte e está intimamente relacionada, já a agregação é mais fraca. A escolha entre composição e agregação depende dos objetivos da pessoa programadora.

Meta-classe

As meta-classes são classes especiais que definem como as outras classes devem ser criadas. Elas oferecem um nível de controle mais profundo sobre o processo de desenvolvimento de classes, sendo úteis para personalizar o comportamento e introduzir uma lógica especializada.

Exemplo:



```
conceitosAvancados.py
1 class MinhaMetaclasse(type):
2     def __new__(cls, nome, bases, dct):
3         # Adiciona um novo atributo à classe
4         dct['novo_atributo'] = 42
5         nova_classe = super().__new__(cls, nome, bases, dct)
6         return nova_classe
7
8 # Usando a meta-classe para criar uma nova classe
9 class MinhaClasse(metaclass=MinhaMetaclasse):
10     pass
11
12 # Testando a nova classe
13 objeto = MinhaClasse()
14 print(objeto.novo_atributo) # Saída: 42
15
```

Figura 7 – Exemplo de código-fonte Python (meta-classe)
Fonte: Elaborado pelo autor (2024)

Neste exemplo, temos a classe “MinhaMetaClasse” que herda de “type” a função “__new__”. Este é um método especial chamado durante a criação da classe e pode realizar ações personalizadas, como adicionar atributos ou métodos à classe. Quando a “MinhaClasse” é criada, a meta-classe “MinhaMetaClasse” adiciona o atributo “novo_atributo” à classe.

Uso prático de Meta-classes

Validação durante a criação da classe

Podemos usar meta-classes para validar se uma classe está sendo definida corretamente, garantindo que ela tenha os atributos necessários.

Modificação de atributos ou métodos

Podemos modificar dinamicamente os atributos ou métodos de uma classe antes que ela seja criada.

Registrar classes automaticamente

Podemos usar para criar um registro automático de todas as classes em um módulo ou pacote.

Implementação de padrões de design

Podemos usar para implementar padrões de design automaticamente durante a criação de classes.

Controle de acesso

Podemos usar para impor regras específicas de acesso a membros da classe.

O uso de meta-classes pode ser avançado e, em muitos casos, o código pode ser mais claro sem elas. No entanto, em situações em que seja necessário um controle mais profundo sobre a criação de classes, as meta-classes oferecem uma ferramenta poderosa.

Design Patterns

Podemos aplicar padrões de projetos como Singleton, Observer e Strategy, compreendendo e aplicando esses padrões que contribuem para a construção de sistemas mais robustos e flexíveis.

Singleton

O padrão Singleton é um padrão de design que garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso para ela. Isso é útil em situações em que você deseja controlar estritamente o acesso a uma única instância de uma classe e garantir que não haja múltiplas instâncias da mesma classe em execução.

Exemplo:

A screenshot of a code editor window titled 'conceitosAvancados.py'. The code defines a Singleton class. It has a class attribute '_instance' set to None. The __new__ method checks if _instance is None; if so, it calls super(Singleton, cls).__new__(cls) to create a new instance and assigns it to _instance. Otherwise, it returns the existing _instance. Below the class definition, there is a test section where two instances, s1 and s2, are created and then compared with 'print(s1 is s2)'. The output is shown as 'Saída: True'.

```
1 class Singleton:
2     _instance = None
3
4     def __new__(cls):
5         if cls._instance is None:
6             cls._instance = super(Singleton, cls).__new__(cls)
7         return cls._instance
8
9 # Testando o Singleton
10 s1 = Singleton()
11 s2 = Singleton()
12
13 print(s1 is s2) # Saída: True
14
```

Figura 8 – Exemplo de código-fonte Python (classe singleton)

Fonte: Elaborado pelo autor (2024)

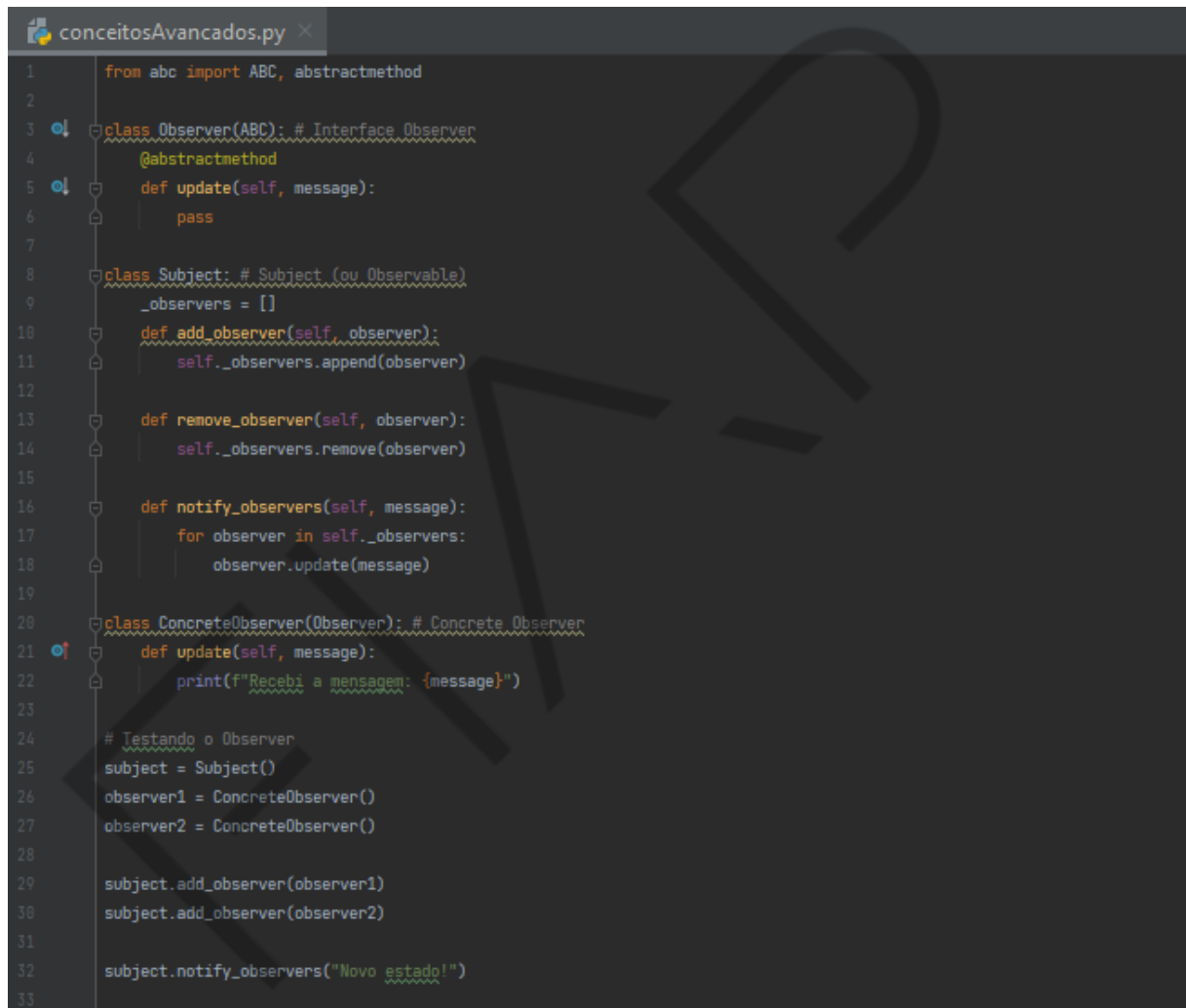
O método “__new__” é responsável por criar uma nova instância da classe. No entanto, antes de criar, ele verifica se a variável de classe “_instance” é nula. Se for, ele cria uma nova instância chamando o construtor da classe base “(super(Singleton, cls).__new__(cls))” e atribui essa instância a “_instance”. Se não for, ele simplesmente retorna a instância existente.

No teste do Singleton com “s1” e “s2”, você cria duas instâncias da classe Singleton. No entanto, como o padrão Singleton garante que haja apenas uma instância, “s1” e “s2” acabam referenciando exatamente a mesma instância. Portanto, a saída do teste “print(s1 is s2)” será “True”, indicando que “s1” e “s2” referenciam o mesmo objeto Singleton.

Observer

O padrão Observer é um padrão de design comportamental que estabelece uma relação de dependência um para muitos entre objetos, de modo que quando um objeto (o "subject" ou "observable") muda de estado, todos os seus dependentes (os "observers") são notificados e atualizados automaticamente.

Exemplo:

A screenshot of a code editor window titled 'conceitosAvancados.py'. The code implements the Observer design pattern in Python. It starts with an import from 'abc' for 'ABC' and 'abstractmethod'. Then, it defines an abstract class 'Observer' that inherits from 'ABC' and has an abstract method 'update(self, message)'. Next, it defines a 'Subject' class that has a list '_observers' and methods 'add_observer', 'remove_observer', and 'notify_observers'. Finally, it defines a 'ConcreteObserver' class that inherits from 'Observer' and implements the 'update' method by printing a message. At the bottom, there is a test section that creates a 'Subject' instance, adds two 'ConcreteObserver' instances, and calls 'notify_observers' with the message 'Novo estado!'.

```
1 from abc import ABC, abstractmethod
2
3 class Observer(ABC): # Interface Observer
4     @abstractmethod
5     def update(self, message):
6         pass
7
8 class Subject: # Subject (ou Observable)
9     _observers = []
10    def add_observer(self, observer):
11        self._observers.append(observer)
12
13    def remove_observer(self, observer):
14        self._observers.remove(observer)
15
16    def notify_observers(self, message):
17        for observer in self._observers:
18            observer.update(message)
19
20 class ConcreteObserver(Observer): # Concrete Observer
21    def update(self, message):
22        print(f"Recebi a mensagem: {message}")
23
24 # Testando o Observer
25 subject = Subject()
26 observer1 = ConcreteObserver()
27 observer2 = ConcreteObserver()
28
29 subject.add_observer(observer1)
30 subject.add_observer(observer2)
31
32 subject.notify_observers("Novo estado!")
33
```

Figura 9 – Exemplo de código-fonte Python (design comportamental - observer)
Fonte: Elaborado pelo autor (2024)

Neste exemplo, temos as seguintes considerações:

Observer

É uma interface que define o método "update", que será chamado quando o estado do subject for alterado. Qualquer classe que queira ser um observer deve implementar essa interface.

Subject

É a classe que é observada. Ela mantém uma lista de observadores “_observers”. Os métodos “add_observer” e “remove_observer” são usados para adicionar e remover observadores da lista. Já o método “notify_observers” percorre a lista de observadores e chama o método update em cada um, passando uma mensagem que indica a mudança de estado.

ConcreteObserver

É uma implementação concreta da interface “Observer”. Neste exemplo, quando o método “update” é chamado ele simplesmente imprime a mensagem recebida.

Teste do Observer

No teste do Observer, é criado um objeto “subject”, dois objetos “observer1” e “observer2” e os observadores são registrados no subject usando “add_observer”. Em seguida, chama-se “notify_observers” no subject, indicando que houve uma mudança de estado. Como resultado, os “observers” (“observer1” e “observer2”) são notificados e seus métodos “update” são chamados, resultando na impressão da mensagem “Novo estado!” para cada observer.

Strategy

O padrão Strategy é um padrão de design comportamental que permite definir uma família de algoritmos, encapsular cada um deles e torná-los intercambiáveis. Isso permite que o cliente escolha dinamicamente um algoritmo específico de uma família de algoritmos sem alterar sua estrutura.

Exemplo:

```
conceitosAvancados.py x
1  from abc import ABC, abstractmethod
2
3  class Strategy(ABC): # Interface Strategy
4      @abstractmethod
5      def execute(self):
6          pass
7
8  class ConcreteStrategyA(Strategy): # Concrete Strategies
9      def execute(self):
10         print("Executando estratégia A")
11
12  class ConcreteStrategyB(Strategy):
13      def execute(self):
14         print("Executando estratégia B")
15
16  class Context: # Context
17      def __init__(self, strategy):
18         self._strategy = strategy
19      def set_strategy(self, strategy):
20         self._strategy = strategy
21      def execute_strategy(self):
22         self._strategy.execute()
23
24  # Testando o Strategy
25  strategy_a = ConcreteStrategyA()
26  strategy_b = ConcreteStrategyB()
27
28  context = Context(strategy_a)
29  context.execute_strategy() # Saída: Executando estratégia A
30
31  context.set_strategy(strategy_b)
32  context.execute_strategy() # Saída: Executando estratégia B
33
```

Figura 10 – Exemplo de código-fonte Python (design comportamental - strategy)

Fonte: Elaborado pelo autor (2024)

Neste exemplo, temos as seguintes considerações:

Strategy

É uma interface que define o método “execute”. Qualquer classe que implemente essa interface pode ser considerada uma estratégia.

ConcreteStrategyA e ConcreteStrategyB

São implementações concretas da interface “Strategy”. Cada uma delas fornece uma implementação específica do método “execute”.

Context

É a classe que possui uma referência para uma estratégia (“_strategy”). O método “set_strategy” permite que o cliente altere dinamicamente a estratégia

associada ao contexto. O método "execute_strategy" chama o método "execute" da estratégia atual.

Teste do strategy

No teste do Strategy são criados objetos para as estratégias A e B. Um objeto "context" é inicializado com a estratégia A e então a estratégia é executada, resultando na saída "Executando estratégia A".

Em seguida, a estratégia é trocada para B usando "set_strategy" e a estratégia é executada novamente, resultando na saída "Executando estratégia B".

Isso demonstra a flexibilidade do padrão Strategy, permitindo que diferentes estratégias sejam usadas dinamicamente.

O QUE VOCÊ VIU NESTA AULA?

Nessa aula, exploramos os princípios essenciais da programação orientada a objetos, focalizando especialmente na linguagem Python e suas aplicações práticas.

Essa representa nossa introdução inicial a uma estrutura mais sólida de programação, direcionada para a reutilização eficiente de códigos e embasada nos quatro pilares da orientação a objetos.

A obtenção de proficiência em POO demanda prática consistente, envolvendo a análise metódica de cada linha de código e uma compreensão abrangente de todos os conceitos envolvidos.

REFERÊNCIAS

LUTZ, M. **Aprendendo Python**. [s.l.]: Novatec, 2014.

RAMALHO, L. **Python fluente**: programação clara, concisa e eficaz. [s.l.]: Novatec, 2015.

SANTOS, N. N. C. M. **Programação orientada a objetos com python**. [s.l.]: Novatec, 2015.

VANDERPLAS, J. **Aprendendo Python**: do Iniciante ao Avançado. [s.l.]: Novatec, 2019.

EXEMPLO

PALAVRAS-CHAVE

Palavras-chave: Python. Programação. Programação Orientada a Objeto.

EMSE



POSTECH