semana 10 - aula 01

Pilares da Programação Orientada a Objetos

Implementação de classes e objetos e conceituação de encapsulamento

Código da aula: [SIS] ANO1C3B2S10A1

Objetivos da Aula:

Conhecer as aplicações práticas acerca da implementação de classes, seus objetos e sua conceituação de encapsulamento.

Exposição:

A aula 01 da semana 10 introduz os fundamentos essenciais da Programação Orientada a Objetos (POO) em Python, focando na implementação de classes e objetos, e na conceituação de encapsulamento. Compreender classes e objetos é apresentado como fundamental para o desenvolvimento de aplicações mais complexas e de fácil manutenção. Uma classe em Python é definida como uma estrutura de código que permite a criação de objetos, os quais podem conter dados (atributos) e comportamentos (métodos). A implementação de classes e objetos é um pilar fundamental da POO. O encapsulamento é introduzido como um princípio da POO que visa esconder os detalhes internos de uma classe, expondo apenas o necessário. Em Python, embora não existam modificadores de acesso como private ou public, utiliza-se a convenção de prefixar um atributo com sublinhado (_) para indicar que ele deve ser tratado como interno à classe. A aula também sugere a prática do desenvolvimento de classes dentro de projetos de software e a utilização de atributos e métodos ao definir classes.

- Fundamentos de classes e objetos em Python: Este tema aborda a base da Programação Orientada a Objetos (POO) na linguagem Python. É apresentado como essencial para o desenvolvimento de aplicações mais complexas e manuteníveis. Inclui a prática do desenvolvimento de classes e a utilização de atributos e métodos.
- Conceitos fundamentais - Classes e objetos: Em Python, uma classe é uma estrutura de código que permite a criação de objetos. Esses objetos podem conter tanto dados (atributos) quanto comportamentos (métodos). A implementação de classes e objetos é um pilar fundamental da Programação Orientada a Objetos (POO).
- Conceitos fundamentais - Encapsulamento: Encapsulamento é um princípio da Programação Orientada a Objetos que visa esconder os detalhes internos de uma classe e expor apenas o necessário. Em Python, a convenção de

prefixar um atributo com sublinhado (_) é utilizada para indicar que ele deve ser tratado como um detalhe interno da classe.

semana 10 - aula 02

Pilares da Programação Orientada a Objetos

Implementação de classes e objetos e conceituação de encapsulamento

Código da aula: [SIS]ANO1C3B2S10A2

Objetivos da Aula:

Praticar a atribuição de classes dentro do contexto de orientação e seus objetos, alinhados à conceituação de encapsulamento.

Exposição:

A aula 02 aprofunda o conceito de encapsulamento em Python, enfatizando sua importância para a segurança e integridade dos dados, permitindo que os objetos controlem o acesso a seus atributos e métodos internos. O encapsulamento é reafirmado como um dos princípios fundamentais da POO, referindo-se à prática de ocultar os detalhes internos de como uma classe funciona, expondo apenas o necessário para uso externo. Isso garante que o estado interno de um objeto (seus atributos) seja mantido privado, com acesso realizado apenas por meio de métodos públicos. A importância do encapsulamento é destacada por razões como segurança (previne alteração inesperada do estado interno), manutenção (facilita mudanças internas sem afetar outras partes do código) e abstração (permite focar no uso do objeto sem necessidade de entender detalhes de implementação). A aula apresenta Getters (acessores) e Setters (modificadores) como métodos que permitem ler e alterar o valor de atributos privados de forma controlada, sendo fundamentais para manter o encapsulamento. A prática do encapsulamento dentro de projetos de software e a utilização de atributos e métodos são incentivadas.

- Por que é importante?: A importância do encapsulamento reside em três aspectos principais: Segurança (previne alteração inesperada ou inadequada do estado interno do objeto), Manutenção (facilita a manutenção do código, pois mudanças internas não afetam outras partes que utilizam a classe) e Abstração (permite focar em como usar um objeto sem a necessidade de entender detalhes complexos de sua implementação interna).

- ❖ Métodos para acessar e modificar dados (Getters e Setters): Getters e Setters são métodos que permitem ler (Getters) e alterar (Setters) o valor de atributos privados de uma classe de maneira controlada. São fundamentais para manter o princípio de encapsulamento, garantindo acesso e modificação seguros e controlados dos dados internos.

semana 10 - aula 03

Pilares da Programação Orientada a Objetos

Implementação de classes e objetos e conceituação de encapsulamento

Código da aula: [SIS]ANO1C3B2S10A3

Objetivos da Aula:

Conhecer as estruturas de objetos dentro de diferentes contextos e aplicar o conceito de classes dentro da visão.

Exposição:

A aula 03 foca na aplicação prática dos conceitos de classes, objetos e encapsulamento, incentivando a aplicação do conhecimento teórico em situações reais de programação. Utilizar classes e objetos é apresentado como uma maneira eficiente de organizar e estruturar o código em programas maiores, agrupando dados e funcionalidades relacionadas para facilitar a compreensão e manutenção. O encapsulamento em cenários reais envolve proteger os dados dentro de uma classe para garantir que sejam acessados e modificados apenas de maneira apropriada, o que é crucial para manter a integridade dos dados e a segurança da aplicação. A aula também aborda boas práticas em POO, incluindo a reutilização de código, a criação de interfaces claras para as classes e a manutenção de um acoplamento fraco entre diferentes partes do código, o que facilita a manutenção e expansão. A prática da Programação Orientada a Objetos e a construção de classes e objetos em Python são reforçadas.

Espero que este roteiro detalhado e os materiais adicionais sejam úteis para a preparação das suas aulas da semana 10! Se precisar de mais alguma coisa, é só perguntar.

Com certeza! Apresento exemplos práticos em Python para cada um dos temas abordados nas aulas da semana 10, seguindo a divisão das aulas:

Exemplos dos Temas - Semana 10

Aula 1: Implementação de Classes e Objetos e Conceituação de Encapsulamento

- **Fundamentos de classes e objetos em Python / Conceitos fundamentais - Classes e objetos:**
    - **Exemplo:** Criando uma classe `Carro` e um objeto dessa classe.
- <!-- end list -->

- Python

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

    def exibir_informacoes(self):
        print(f"Marca: {self.marca}, Modelo: {self.modelo}")

# Criando um objeto (instância) da classe Carro
meu_carro = Carro("Toyota", "Corolla")
meu_carro.exibir_informacoes()
```

-
    - **Explicação:** Definimos uma classe `Carro` com atributos (`marca`, `modelo`) e um método (`exibir_informacoes`). Em seguida, criamos um objeto `meu_carro` a partir dessa classe.
- **Conceitos fundamentais - Encapsulamento:**
    - **Exemplo:** Usando a convenção de sublinhado para indicar um atributo "privado".
- <!-- end list -->

- Python

```
class ContaBancaria:
    def __init__(self, saldo_inicial):
        self._saldo = saldo_inicial # Atributo "encapsulado" pela convenção

    def depositar(self, valor):
        if valor > 0:
            self._saldo += valor
            print("Depósito realizado com sucesso.")
        else:
```

```
            print("Valor de depósito inválido.")

    def exibir_saldo(self):
        print(f"Saldo atual: {self._saldo}")

minha_conta = ContaBancaria(1000)
minha_conta.depositar(500)
minha_conta.exibir_saldo()
# Acessar diretamente _saldo é desencorajado pela convenção, embora possível em Python
# print(minha_conta._saldo)
```

-
    - **Explicação:** O atributo `_saldo` é "encapsulado" usando a convenção de sublinhado. O acesso e modificação são feitos através dos métodos `depositar` e `exibir_saldo`.

Aula 2: Prática de Atribuição de Classes e Encapsulamento

- **Encapsulamento em Python: protegendo os dados / O que é encapsulamento? / Por que é importante?:**
    - **Exemplo:** Demonstração da proteção de dados e controle de acesso.
- <!-- end list -->

- Python

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

    def get_preco(self): # Getter
        return self._preco

    def set_preco(self, novo_preco): # Setter
        if novo_preco > 0:
            self._preco = novo_preco
            print("Preço atualizado.")
        else:
            print("Preço inválido. Não atualizado.")

tenis = Produto("Tênis Esportivo", 250.00)
print(f"Preço inicial: {tenis.get_preco()}")
tenis.set_preco(280.00)
print(f"Novo preço: {tenis.get_preco()}")
tenis.set_preco(-50.00) # Tentativa de definir um preço inválido
print(f"Preço após tentativa inválida: {tenis.get_preco()}")
```

-
    - **Explicação:** O preço (`_preco`) é protegido e só pode ser alterado através do método `set_preco`, que inclui uma validação para garantir que o novo preço seja positivo. Isso demonstra a segurança e integridade dos dados proporcionadas pelo encapsulamento.
- **Métodos para acessar e modificar dados (Getters e Setters):**
    - **Exemplo:** (Já mostrado no exemplo anterior) Os métodos `get_preco()` e `set_preco()` são exemplos clássicos de Getters e Setters, respectivamente. Eles fornecem uma interface controlada para acessar e modificar o atributo `_preco`.

Aula 3: Aplicação Prática de Classes, Objetos e Encapsulamento

- **Aplicação prática de classes, objetos e encapsulamento / Exemplos práticos de classes e objetos / Encapsulamento em cenários reais:**
    - **Exemplo:** Um sistema simples de gerenciamento de funcionários.
- <!-- end list -->

- Python

```python
class Funcionario:
    def __init__(self, nome, cargo, salario):
        self.nome = nome
        self.cargo = cargo
        self._salario = salario # Salário é encapsulado

    def get_salario(self):
        # Poderia adicionar lógica de cálculo de bônus, impostos, etc.
        return self._salario

    def set_salario(self, novo_salario):
        if novo_salario > self._salario:
            self._salario = novo_salario
            print(f"Salário de {self.nome} atualizado para {self._salario}")
        else:
            print(f"Novo salário para {self.nome} deve ser maior que o atual.")

    def exibir_detalhes(self):
        print(f"Nome: {self.nome}, Cargo: {self.cargo}, Salário: {self.get_salario()}")

func1 = Funcionario("Ana Clara", "Desenvolvedora", 5000)
func1.exibir_detalhes()
func1.set_salario(5500)
```

```
func1.exibir_detalhes()
func1.set_salario(5200) # Tentativa de diminuir ou manter salário
```

-
    - **Explicação:** A classe `Funcionario` modela um funcionário, encapsulando o atributo `_salario` e controlando sua modificação via `set_salario`. `get_salario` permite acessar o valor.
- **Boas práticas com POO:**
    - **Exemplo:** Reutilização de código e interfaces claras.
- <!-- end list -->

- Python

```
# Reutilização: A classe Carro poderia ser usada em diferentes partes de um sistema (vendas, manutenção, etc.)

# Interfaces Claras: O método exibir_informacoes() na classe Carro
# ou get_salario() na classe Funcionario são exemplos de interfaces claras,
# indicando o que o método faz sem expor detalhes internos.

class Animal: # Classe base para reutilização
    def __init__(self, nome):
        self.nome = nome

    def comunicar(self):
        pass # Método a ser implementado pelas subclasses

class Cachorro(Animal): # Reutiliza a estrutura de Animal
    def comunicar(self):
        print(f"{self.nome} late!")

class Gato(Animal): # Reutiliza a estrutura de Animal
    def comunicar(self):
        print(f"{self.nome} mia!")

# Interface clara: Chamar o método comunicar()
rex = Cachorro("Rex")
felix = Gato("Felix")
rex.comunicar()
felix.comunicar()
```

-
    - **Explicação:** A classe `Animal` serve como base para `Cachorro` e `Gato`, demonstrando reutilização. O método `comunicar()` é uma interface clara para a ação de comunicação de diferentes animais.