Aula 03 - POO em Python: Encapsulamento

Introdução ao tema

Encapsulamento se refere ao agrupamento de dados com os mecanismos ou métodos que operam nos dados. É o processo de manter detalhes da informação oculta do usuário.

Nessa aula será abordado o encapsulamento e como implementá-lo em Python. Será trabalho:

- Encapsulamento em Python.
- Motivação para o encapsulamento.
- Ocultação de dados com os modificadores de acesso.
- Métodos Getter e Setter.

Encapsulamento em Python

Encapsulamento em Python descreve o conceito de agrupar dados e métodos em uma única unidade, ou seja, ao criar uma classe está sendo implementado o encapsulamento.

Uma classe é um exemplo de encapsulamento., pois vincula todos os membros (atributos/variáveis de classe e instância) e métodos em uma única unidade.

Exemplo Real

```
class Employee:
    # Construtor
    def __init__(self, name, salary, project):
        # Atributos: Membros de Dados
        self.name = name
        self.age = age
        self.salary = salary
        self.project = project

    # Métodos: Ações
    def show(self):
        print(f'Nome: {self.name}\tIdade: {self.age}\tSalário: R$ {self.salary}')

    def work(self):
        print(f'{self.name} no momento está atuando no projeto {self.project}')

# Instânciando objeto da classe
julio_cesar = Employee('Júlio Cesar', 23, 7.500, 'PLN')

# Chamando os métodos públicos
julio_cesar.show()
julio_cesar.work()
```

Saída

```
Nome do Funcionário: Júlio Cesar    Idade: 23   Salário: R$ 7.500

Júlio Cesar no momento está atuando no projeto PLN
```

- No exemplo anterior foi criada a classe Employee (Empregado) definindo atributos de funcionário, como `name`, `age`, `salary` e `project`, e foi implementado o comportamento atráves dos métodos `work()` e `show()`.

Usando encapsulamento, podemos esconder a representação interna de um objeto do exterior. Isso é chamado de ocultação de informação.

Além disso, o encapsulamento nos permite restringir o acesso direto a atributos e métodos e evitar modificações acidentais de dados criando membros de dados e métodos privados dentro de uma classe.

Modificadores de Acesso

O encapsulamento pode ser alcançado declarando os membros de dados e métodos de uma classe como private ou protected.

Em Python, não temos modificadores de acesso direto como `public`, `private` e `protected` como presentes na linguagem Java. No entanto, podemos aplicar os modificadores de acesso utilizando ***underscore simples*** "_" e ***underscores duplos*** "__".

Modificadores de acesso limitam o acesso aos atributos e métodos de uma classe. Python fornece três tipos de modificadores de acesso: `private`, `public` e `protected`.

1. **Membros Públicos**: Acessível de dentro ou fora da classe.
2. **Membros Privado**: Acessível somente dentro da classe.
3. **Membros Protegidos**: Acessível dentro da classe e de suas subclasses.

```
class Employee:
    def __init__(self, name, salary, project):
        # Atributos: Membros de Dados
        self.name = name # Público
        self.__salary = salary # Privado
        self._project = project # Protegido
```

1. Membros Públicos

Membros de dados públicos são acessíveis dentro e fora de uma classe. Todas as variáveis de membro da classe são públicas por padrão.

Exemplo Real

```
# Criando a classe pessoa
class Person:
  def __init__(self, name, profession):
    self.name = name
    self.profession = profession

  def show(self):
    print(self.name, "\t", self.profession)

# Criando objetos da classe pessoa
reinaldo = Person('Reinaldo Carlos Mendes', 'Cientista de Dados')

# Acessando os atributos públicos
print(reinaldo.name, '\t', reinaldo.profession)

# Chamando método público
reinaldo.show()
```

Saída

```
Reinaldo Carlos Mendes  Cientista de Dados
Reinaldo Carlos Mendes  Cientista de Dados
```

2. Membro Privado

Membros privados são acessíveis somente dentro da classe, e não podemos acessá-los diretamente dos objetos de classe.

Para definir uma variável privada, adicione dois underscores `__` como prefixo no início do nome de uma variável.

Exemplo Real

```python
class Employee:
    # construtor
    def __init__(self, name, salary):
        # Membro de dado público
        self.name = name
        # Membro de dado privado
        self.__salary = salary

# criando um objeto da classe
empregado_duda = Employee('Duda', 10.000)

# acessando membro de dados privado
print('Salário:', emp.__salary)
```

Saída

```
AttributeError: 'Employee' object has no attribute '__salary'
```

No exemplo anterior, o `__salary` é um atributo privada. Não é possível acessar um atributo privada de fora dessa classe.

3. Membro Protegido

Membros protegidos são acessíveis dentro da classe e também disponíveis para suas subclasses.

Para definir um membro protegido, prefixe o nome do membro com um único sublinhado `_`.

- *Membros de dados protegidos são usados quando você implementa **herança** e deseja permitir que somente classes filhas acessem os membros de dados.*

Exemplo Real

```python
# Classe Base
class Company:
    def __init__(self):
        # Protected member
        self._project = "MFA - App Call Center"

# Classe Filha
class Employee(Company):
    def __init__(self, name):
        self.name = name
        Company.__init__(self)

    def show(self):
        print("Nome Funcionário :", self.name)
        # Accessing protected member in child class
        print("Trabalhando no projeto :", self._project)

empregado_aristela = Employee("Aristela")
empregado_aristela.show()

# Acesso direto a um membro de dados protegido.
print('Projeto:', empregado_aristela._project)
```

Sáida

```
Nome funcionário: Jessa
Trabalhando no projeto: MFA - App Call Center
Projeto: MFA - App Call Center
```

Acessando Membros de Dados Privados e Protegidos.

É importante destacar que a sintaxe de sublinhado duplo ou simples ***é apenas uma convenção*** e ***não é uma regra estrita***.

Podemos acessar ***membros privados*** de fora de uma classe usando as seguintes abordagens.

1. Usar a ***manipulação*** de nomes.
2. Criar um ***método público*** para acessar ***membros privados***.

1. Manipulação de nomes para acessar membros de dados privados ou protegidos

É possível acessar diretamente atributos privadas e protegidas de fora de uma classe por meio de ***name mangling*** (***mutilação do nome***).

O *name mangling* é criado em um identificador adicionando dois sublinhados iniciais e um sublinhado final, como `_classname__dataMember`, onde `classname` é a ***classe atual***, e `dataMember` é o nome da ***atributo/variável privada***.

Exemplo Real

```
class Employee:
    def __init__(self, name, salary):
        # atributo público
        self.name = name
        # atributo privado
        self.__salary = salary

funcionario = Employee('André', 10.000)

print('Nome:', funcionario.name)

# acesso direto para atributos privado usando "name mangling"
print('Salário:', funcionario._Employee__salary)
```

Saída

```
Nome: André
Salário: 10.000
```

2. Criando Métodos Públicos para acesar membros privados ou protegidos

É possível acessar atributos Privados e Protegidos das classes atráves da criação de métodos públicos que dentro da classe realizam o acesso aos atributos.

Uma abordagem muito comumn no encapsulamento é criar métodos chamados de getter e setter para o acesso aos atributos protegidos ou privados.

Use o método getter para acessar membros de dados e os métodos setter para modificar os membros de dados.

Métodos Getter e Setter

Em Python, variáveis privadas não são campos ocultos como em outras linguagens de programação. Os métodos getters e setters são frequentemente usados quando:

- Quando queremos evitar o acesso direto a variáveis privadas.
- Para adicionar lógica de validação para definir um valor.

Exemplo Real

```
class Student:
    def __init__(self, name, age):
        self.name = name
        # membro privado
        self.__age = age

    # método get
    def get_age(self):
        return self.__age

    # método set
    def set_age(self, age):
        self.__age = age

aluno = Student('Luís', 14)

# recuperando a idade usando get
print('Nome:', aluno.name, aluno.get_age())

# Alterando a idade usando o método set
aluno.set_age(16)

# recuperando a idade usando get
print('Name:', aluno.name, aluno.get_age())
```

Saída

```
Nome: Luís 14
Nome: Luís 16
```

No exemplo em questão, a variável/atributo `name` é um membro de dados público e acessada de forma direta pela chama do objeto (`object.member`), no entanto, a variável `age` é um membro de dados privado, e para realizar o acesso e modificação, fazemos o uso da estrategia dos metódos públicos, `get_age()` para recuperar a informação e `set_age()` para definir um novo valor.

O exemplo a seguir demonstra como usar o encapsulamento para implementar a ocultação de informações e aplicar validação adicional antes de alterar os valores dos atributos do objeto.

Exemplo Real

```
class Student:
    def __init__(self, name, age):
        self.name = name

        # membros privado para restringir o acesso e evidar a moficaçao direta.
        self.__roll_no = 0
        self.__age = age

    def show(self):
        print('Informações do Estudante:', self.name, self.__roll_no)

    # método get
    def get_roll_no(self):
        return self.__roll_no

    # método set. Validação para realizar a modificação
    def set_roll_no(self, number):
        if number > 50:
            print('Matrícula Ínvalida. Por favor, entre com um número de matrícula correto.')
        else:
            self.__roll_no = number
```

```
student_leonardo = Student('Leonardo', 15)

# Antes da moficação
student_leonardo.show()

# Tentativa de definir matrícula com método set
student_leonardo.set_roll_no(120)


student_leonardo.set_roll_no(25)
student_leonardo.show()
```

Saída

```
Informações do Estudante: Leonardo, 0

Matrícula Ínvalida. Por favor, entre com um número de matrícula correto.

Informações do Estudante: Leonardo, 25
```

Benéficios do Encapsulamento

- **Segurança**: A principal vantagem de usar encapsulamento é a segurança dos dados. O encapsulamento protege um objeto de acesso não autorizado. Ele permite níveis de acesso privado e protegido para evitar modificação acidental de dados.
- **Simplicidade**: Simplifica a manutenção do aplicativo mantendo as classes separadas e evitando que elas se acoplem fortemente umas às outras.

Benéficios do Encapsulamento

- **Ocultação de Dados**: O usuário não saberia o que está acontecendo nos bastidores. Ele saberia apenas que para modificar um membro de dados, chame o método setter. Para ler um membro de dados, chame o método getter. O que esses métodos setter e getter estão fazendo é oculto deles.
- **Estética**: Agrupar dados e métodos em uma classe torna o código mais legível e sustentável.