

ENCAPSULAMENTO DE DADOS

Uma das principais vantagens de orientação a objetos é a utilização de estruturas sem necessidade de conhecer como elas foram implementadas, e por isso o conceito de encapsulamento de dados se torna essencial.

O encapsulamento de dados envolve a proteção dos atributos ou métodos de uma classe e vem com a premissa que somente a classe onde as declarações foram feitas tenham acesso.

Esse conceito garante a integridade das informações e facilita a utilização das implementações.

Diferente da maioria das linguagens como Java, PHP e C#, o Python mantém todos os atributos e métodos públicos, o que não significa que todas as funções de uma classe podem ser chamadas por outras ou que todos os atributos podem ser lidos e alterados sem cuidado.

Para isso, na linguagem Python tem um método chamado de convenção.

MODIFICADORES DE ACESSO

Diferente de outras linguagens que utilizam palavras reservadas para alterar a visibilidade dos atributos de uma classe, no Python é utilizado o símbolo *underscore/underline*.

Dentro da orientação a objetos temos os modificadores:

- **Public:** Ele permite acesso tanto de dentro, quanto de fora de uma classe e sua implementação é por meio do *underline* na frente do nome.
- **Protected:** O modificador protegido faz com que apenas a suas *classes* e *subclasses* tenham acesso ao atributo ou método e sua implementação é por meio do *underline* antes do nome
- **Private:** É o modificador mais restrito da orientação a objetos, permitindo apenas que somente a sua classe tenha acesso a um determinado atributo ou método e sua implementação é através do *underline duplo* antes do nome.

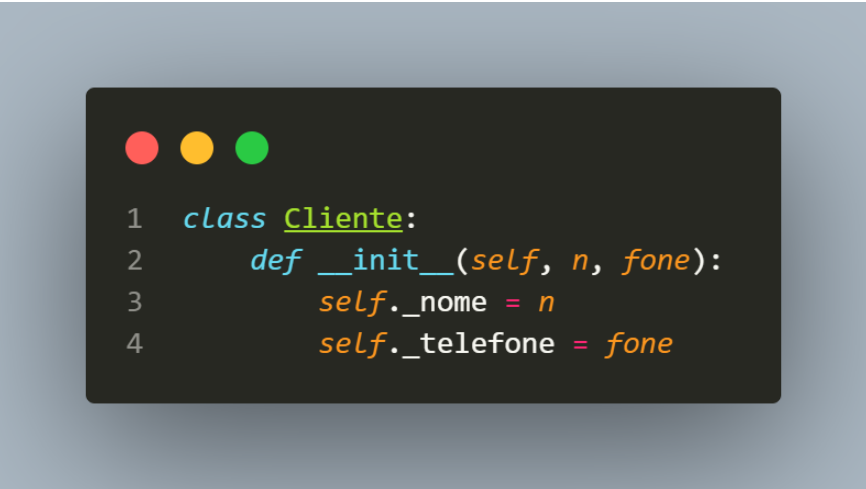
VISIBILIDADE DOS MEMBROS

Um dos recursos mais importantes da orientação a objetos é o de restringir o acesso a variáveis de um objeto e a alguns métodos, evitando que variáveis internas sem acessadas e recebam valores diretamente, garantindo a integridade das informações.

Modificando os atributos:

Para iniciar o processo de encapsulamento, vamos modificar os atributos das classes de forma que fiquem privados, começando pela classe Cliente

Observe o underline criado antes da definição do nome do atributo



```
1 class Cliente:
2     def __init__(self, n, fone):
3         self._nome = n
4         self._telefone = fone
```

Após a alteração do nome do atributo, a classe Main não vai mais visualizar o atributo diretamente.

MÉTODOS DE ACESSO (GET e SET)

Getters e **Setters** são usados na maioria das linguagens de programação orientada a objetos com o objetivo de garantir o princípio de encapsulamento de dados.

Os métodos são utilizados para implementações que alteram os valores internos da classe ou que retornam valores dela.

GET

- Sempre retornam valores.

O método Get é utilizado para ler os valores internos do objeto e enviá-los como valor de retorno da função.

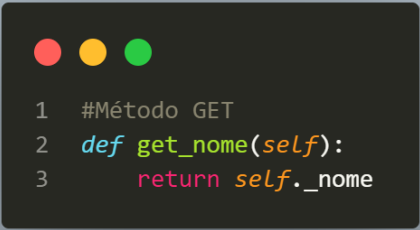
SET

- Recebem valores por parâmetros.

Os métodos Set recebem argumentos que são atribuídos a membros internos do objeto.

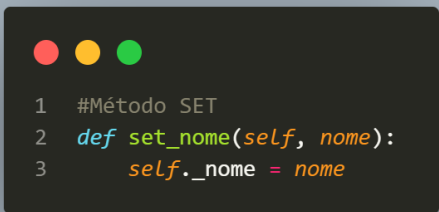
Sintaxe dos métodos de acesso:

- **Get:**



```
1  #Método GET
2  def get_nome(self):
3      return self._nome
```

- **Set:**



```
1  #Método SET
2  def set_nome(self, nome):
3      self._nome = nome
```

FORMA “PYTHÔNICA” DE ENCAPSULAR DADOS

O conjunto de métodos públicos de uma classe é conhecido como interface da classe, sendo a única maneira de comunicação com os objetos da classe.

Em Python, o *underline* antes do atributo não impede o acesso dele em outra classe, ou seja, ele não fica privado.

Em alguns atributos é muito importante preservar o valor iniciado na classe, não sendo possível realizar a inserção de qualquer valor no atributo, a não ser por meio de métodos.

Praticando com a classe: Conta

Para o saldo não ser negativo, a utilização do método setter é justificável, ficando do seguinte modo:



```
1  class Conta:
2      def __init__(self, titular, saldo, numero):
3          self.titular = titular
4          self.saldo = 0
5          self.numero = numero
6
7      def get_saldo(self):
8          return self._saldo
9
10     def set_saldo(self, saldo):
11         if (saldo < 0):
12             print("O saldo não pode ser negativo!")
13         else:
14             self._saldo = saldo
```

RESUMINDO OS GETTERS E SETTERS...

Em Python, o conceito de “público e privado” não existe na sintaxe da linguagem, o que temos é a convenção de estilo que diz que nomes de atributos, métodos e funções iniciados com underline não devem ser usados por usuários de uma classe, só pelos próprios implementadores e que o funcionamento desses métodos e funções pode mudar sem aviso prévio.

Portanto, não é considerado errado deixar os atributos simplesmente como atributos de instância de forma simples, onde qualquer usuário da classe pode ler ou alterar, sem depender de nenhum outro mecanismo.

PROTOCOLO DE DESCRITORES - DECORATOR

Um decorator é um padrão de projeto de software que permite adicionar comportamento a um objeto já existente, em tempo de execução, ou seja, agrega, de forma dinâmica, responsabilidades adicionais a um objeto.

Na prática, o decorator permite que atributos de uma classe tenham responsabilidades.

Um decorator é um objeto invocável, uma função que aceita outra função como parâmetro (a função decorada).

@PROPERTY

A linguagem Python traz uma outra solução para manter os atributos privados, conhecida como Property.

A função Property é um Decorator e é utilizada para obter um valor de um atributo.

Basicamente, a função Property permite que você declare uma função para obter o valor de um atributo.

```
1  class Conta:
2      def __init__(self, titular, saldo, numero):
3          self.titular = titular
4          self.saldo = 0
5          self.numero = numero
6
7      @property
8      def saldo(self):
9          return self._saldo
10
11     @saldo.setter
12     def saldo(self, saldo):
13         if (saldo < 0):
14             print("O saldo não pode ser negativo!")
15         else:
16             self._saldo = saldo
```