Programação Orientada a Objetos em Python

Instrutor: Tarik Ponciano

Senac Fecomércio Sesc

Links da Disciplina

1. Discord: https://discord.gg/wt5CVZZWJs

2. Drive: https://drive.google.com/drive/folders/1hOl0DaPeAor7gnhKBUIlZ5n8lLRDNvUY?usp=sharing

3. Github: https://github.com/TarikPonciano/Programador-de-Sistema-SENAC

Encapsulamento

Usamos esse princípio para juntar, ou encapsular, dados e comportamentos relacionados em entidades únicas, que chamamos de objetos.

Por exemplo, se quisermos modelar uma entidade do mundo real, por exemplo Computador.

Encapsular é agregar todos os atributos e comportamentos referentes à essa Entidade dentro de sua Classe.

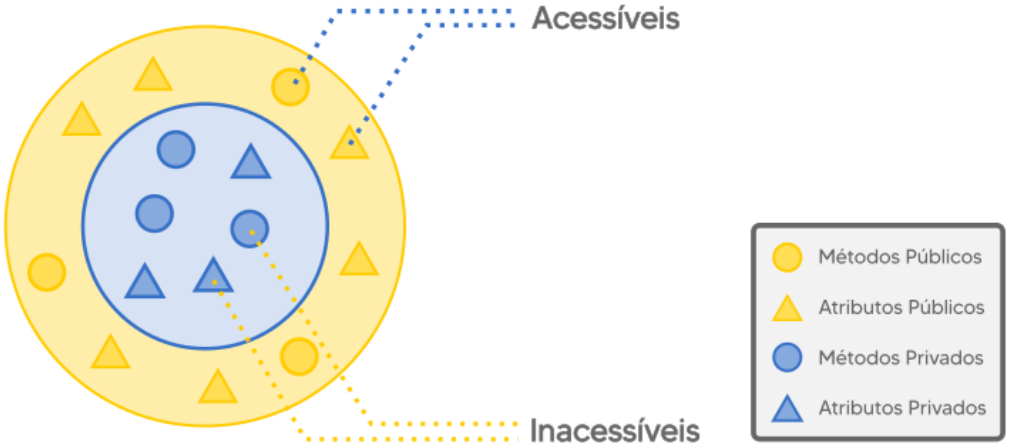
Dessa forma, o mundo exterior não precisa saber como um Computador liga e desliga, ou como ele realiza cálculos matemáticos!

Basta instanciar um objeto da Classe Computador, e utilizá-lo!

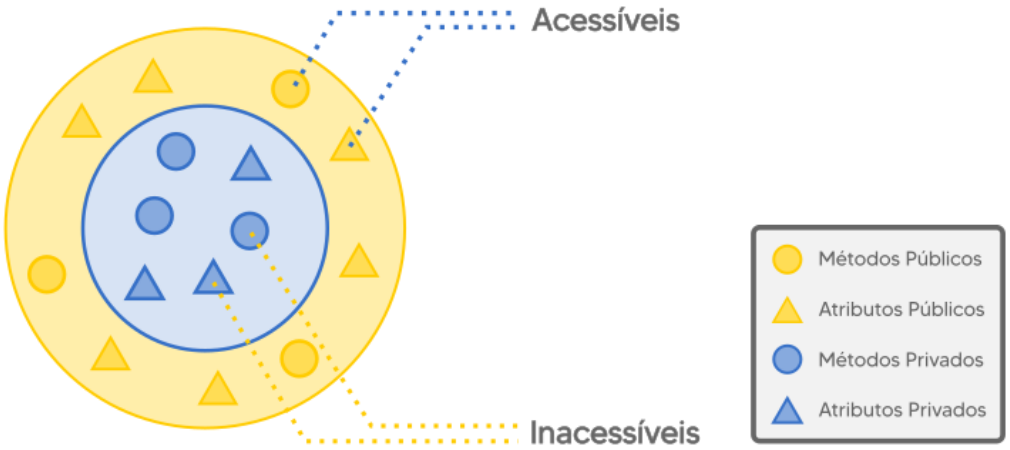
Encapsulamento

O princípio do Encapsulamento também afirma que informações importantes devem ser contidas dentro do objeto de maneira privada e apenas informações selecionadas devem ser expostas publicamente.

Veja a imagem abaixo que exemplifica a relação entre atributos e métodos públicos e privados:



Encapsulamento



A implementação e o estado de cada objeto são mantidos de forma privada dentro da definição da Classe. Outros objetos não têm acesso a esta classe ou autoridade para fazer alterações.

Eles só podem chamar uma lista de funções ou métodos públicos. Essa característica de ocultação de dados fornece maior segurança ao programa e evita corrupção de dados não intencional.

Encapsulamento

```
class Conta:

    def __init__(self, numero, titular, saldo, limite=1000.0):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite

    # outros métodos

    def saca(self, valor):
        self.saldo -= valor
```

Vamos testar nosso código:

```
minha_conta = Conta('123-4', 'João', 1000.0, 2000.0)
minha_conta.saca(500000)
```

Encapsulamento

Em linguagens como Java e C#, basta declarar que os atributos não possam ser acessados de fora da classe utilizando a palavra chave private. Em orientação a objetos, é prática quase que obrigatória proteger seus atributos com private. Cada classe é responsável por controlar seus atributos, portanto ela deve julgar se aquele novo valor é válido ou não. E esta validação não deve ser controlada por quem está usando a classe, e sim por ela mesma, centralizando essa responsabilidade e facilitando futuras mudanças no sistema.

O Python não utiliza o termo private, que é um modificador de acesso e também chamado de modificador de visibilidade. No Python, inserimos dois underscores ('__') ao atributo para adicionarmos esta característica:

```
class Pessoa:

    def __init__(self, idade):
        self.__idade = idade
```

Senac
Fecomércio
Sesc

Encapsulamento

```
class Pessoa:

    def __init__(self, idade):
        self.__idade = idade
```

```
pessoa = Pessoa(20)
pessoa.idade
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Pessoa' object has no attribute 'idade'
```

Senac
Fecomércio
Sesc

Encapsulamento

```
class Conta:

    def __init__(self, titular, saldo):

        self._titular = titular
        self._saldo = saldo

    def get_saldo(self):
        return self._saldo

    def set_saldo(self, saldo):
        self._saldo = saldo

    def get_titular(self):
        return self._titular

    def set_titular(self, titular):
        self._titular = titular
```

Encapsulamento

Getters e setters são usados em muitas linguagens de programação orientada a objetos para garantir o princípio do encapsulamento de dados. O encapsulamento de dados é visto como o agrupamento de dados com os métodos que operam nesses dados. Esses métodos são, obviamente, o getter para recuperar os dados e o setter para alterar os dados. De acordo com esse princípio, os atributos de uma classe são tornados privados para ocultá-los e protegê-los de outro código.

```
class Conta:

    def __init__(self, saldo):
        self._saldo = saldo

    def get_saldo(self):
        return self._saldo

    def set_saldo(self, saldo):
        self._saldo = saldo
```

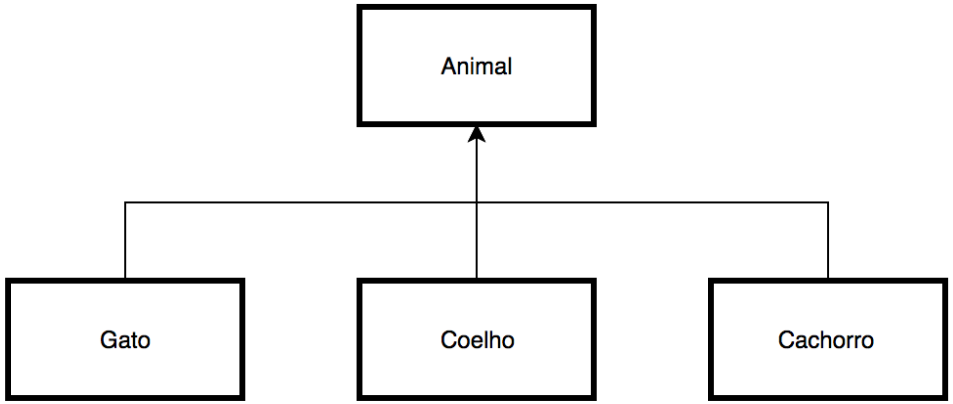
Encapsulamento

```
class Conta:

    def __init__(self, saldo):
        self._saldo = saldo

    def get_saldo(self):
        return self._saldo

    def set_saldo(self, saldo):
        self._saldo = saldo

conta1 = Conta(200.0)
conta2 = Conta(300.0)
conta3 = Conta(-100.0)
conta1.get_saldo()
#200.0
conta2.get_saldo()
#300.0
conta3.set_saldo(conta1.get_saldo() + conta2.get_saldo())
    conta3.get_saldo()
#500.0
```

Herança

A Herança é um conceito do paradigma da orientação à objetos que determina que uma classe (filha) pode herdar atributos e métodos de uma outra classe (pai) e, assim, evitar que haja muita repetição de código.

Um exemplo do conceito de herança pode ser visto no diagrama a seguir:



Herança

Para utilizar a herança no Python é bem simples. Assim com vimos no diagrama anterior, vamos criar quatro classes para representar as entidades Animal, Gato, Cachorro e Coelho.

```
class Animal():
    def __init__(self, nome, cor):
        self.__nome = nome
        self.__cor = cor

    def comer(self):
        print(f"O {self.__nome} está comendo")
```

No código acima definimos a classe pai que irá possuir todos os atributos e métodos comuns às classes filhas (Gato, Cachorro e Coelho). Nela, criamos apenas o construtor que irá receber o nome e a cor do animal, além do método comer que vai exibir a mensagem com o nome do animal que está comendo.

Herança

Após isso, criamos as três classes “filhas” da classe Animal. Para definir que estas classes são herdeiras da classe Animal, declaramos o nome da classe pai nos parênteses logo após definir o nome da classe, como podemos ver abaixo:

```
import animal

class Gato(animal.Animal):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)
```

```
import animal

class Cachorro(animal.Animal):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)
```

```
import animal

class Coelho(animal.Animal):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)
```

Herança

Agora, por herdar da classe Animal, as classes Gato, Cachorro e Coelho podem, sem nenhuma alteração, utilizar o método comer(), definido na classe Animal pois elas herdam dessa classe, logo elas possuem a permissão de invocar este método:

```
import gato, cachorro, coelho

gato = gato.Gato("Bichano", "Branco")
cachorro = cachorro.Cachorro("Totó", "Preto")
coelho = coelho.Coelho("Pernalonga", "Cinza")

gato.comer()
cachorro.comer()
coelho.comer()
```

```
O Bichano está comendo
O Totó está comendo
O Pernalonga está comendo
```

Herança

```
class Funcionario:

    def __init__(self, nome, cpf, salario):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario

    # outros métodos e propriedades
```

Herança

```
class Gerente:

    def __init__(self, nome, cpf, salario, senha, qtd_gerenciados):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario
        self._senha = senha
        self._qtd_gerenciados = qtd_gerenciados

    def autentica(self, senha):
        if self._senha == senha:
            print("acesso permitido")
            return True
        else:
           print("acesso negado")
           return False

    # outros métodos (comuns a um Funcionario)
```

Senac Fecomércio Sesc

Herança

```
class Gerente(Funcionario):

    def __init__(self, senha, qtd_funcionarios):
        self._senha = senha
        self._qtd_funcionarios = qtd_funcionarios

    def autentica(self, senha):
        if self._senha == senha:
            print("acesso permitido")
            return True
        else:
            print("acesso negado")
            return False
```

Herança

```
class Gerente(Funcionario):

    def __init__(self, senha, qtd_funcionarios):
        Funcionario.__init__(nome, cpf, salario)
        self._senha = senha
        self._qtd_funcionarios = qtd_funcionarios

    def autentica(self, senha):
        if self._senha == senha:
            print("acesso permitido")
            return True
        else:
            print("acesso negado")
            return False
```

Utilize o conceito de herança para criar a Classe Pokemon e as subclasses de tipo de Pokemon

ex: PokemonAquatico, PokemonFogo, PokemonEletrico

Polimorfismo

Quando utilizamos Herança, teremos Classes filhas utilizando código comum da Classe acima, ou Classe pai. Ou seja, as Classes vão compartilhar atributos e comportamentos (herdados da Classe acima).

Assim, Objetos de Classes diferentes, terão métodos e atributos compartilhados que podem ter implementações diferentes, ou seja, um método pode possuir várias formas e atributos podem adquirir valores diferentes. Daí o nome: Poli (muitas) morfismo (formas).

Para entendermos melhor, vamos utilizar o exemplo da entidade Cachorro que herda de Animal.

Suponha agora que Animal possua a definição do método correr().

Por conta do conceito de Polimorfismo, objetos da Classe Coelho terão uma implementação do método correr() que será diferente da implementação desse métodos em instâncias da Classe Cachorro!

Polimorfismo

```
class Funcionario:

    def __init__(self, nome, cpf, salario):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario

    # outros métodos e properties

    def get_bonificacao(self):
        return self._salario * 0.10
```

Polimorfismo

```
class Gerente(Funcionario):

    def __init__(self, nome, cpf, salario, senha, qtd_gerenciaveis):
        super().__init__(nome, cpf, salario)
        self._senha = senha
        self._qtd_gerenciaveis = qtd_gerenciaveis

    def get_bonificacao(self):
        return self._salario * 0.15

    # metodos e properties
```

Senac
Fecomércio
Sesc

Utilize o conceito de polimorfismo para criar um método checarVantagem na Classe Pokemon e modifique esse método nas subclasses de acordo com as vantagens e desvantagens daquele tipo.

Referências

1. https://www.treinaweb.com.br/blog/orientacao-a-objetos-em-python
2. https://www.caelum.com.br/apostila/apostila-python-orientacao-a-objetos.pdf
3. http://www.estruturas.ufpr.br/disciplinas/pos-graduacao/introducao-a-computacao-cientifica-com-python/introducao-python/1-9-programacao-orientada-a-objeto-poo/
4. https://www.treinaweb.com.br/blog/utilizando-heranca-no-python
5. https://pythonacademy.com.br/blog/introducao-a-programacao-orientada-a-objetos-no-python