



Orientação a Objetos (OO)

Prof. Gustavo Molina

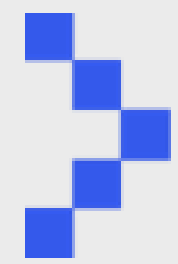
< thefutureisblue.me />



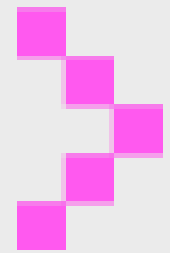
Orientação a objetos



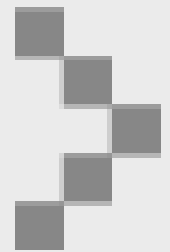
Paradigma Orientado a Objetos - Definições



Paradigma para desenvolvimento de *software* que baseia-se na utilização de componentes individuais (objetos) que colaboram para construir sistemas mais complexos.

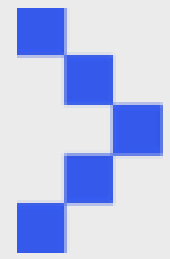


A colaboração entre objetos é feita através do envio de mensagens.

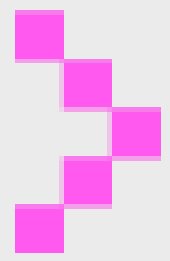


Um paradigma é um conjunto de regras que estabelecem fronteiras e descrevem como resolver problemas dentro dessa fronteira.

Vantagens da Orientação a Objetos



Facilita a reutilização do código.



Pequenas mudanças nos requisitos não implicam em grandes alterações no sistema em desenvolvimento.



Orientação a Objetos (OO) nos aproxima do mundo real.

Os 4 pilares da Orientação a Objetos

1 Abstração

2 Encapsulamento

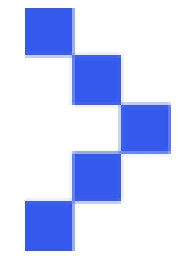
3 Herança

4 Polimorfismo

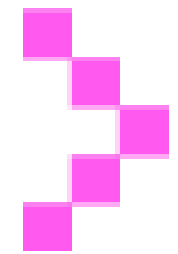


Abstração

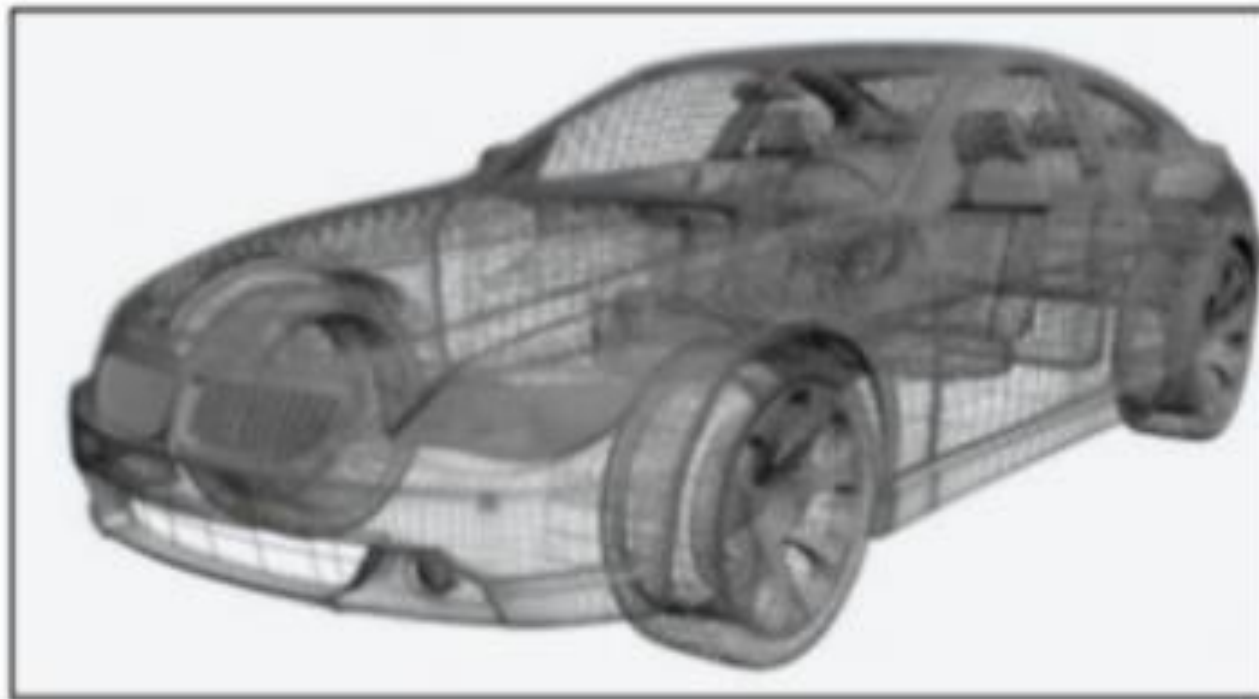
Classes



Estrutura fundamental para definir novos objetos.



Uma classe é definida em código-fonte.

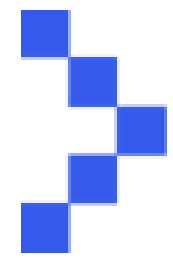


Classe



Objeto

Classes em Python



`class nome_da_classe:`

```
class Conta:  
    número = 0000000  
    saldo = 0.0
```



Instância

- Uma instância é um objeto criado com base em uma classe definida;
- Classe é apenas uma estrutura, que especifica objetos, mas que não pode ser utilizada diretamente;
- Instância representa o objeto criado por uma classe e apresenta um ciclo de vida:


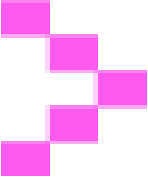




Instância em Python

 `variável = Classe()`

```
if __name__ == '__main__':  
    conta = Conta()  
    conta.saldo = 20  
    conta.número = "13131-2"  
    print(conta.saldo)  
    print(conta.número)
```

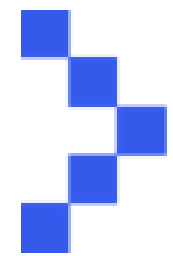
Métodos

-  Representam o comportamento de uma classe;
-  Permitem acesso a atributos, tanto para recuperar os valores, como para alterá-los caso necessário;
-  Podem ou não retornar algum valor;
-  Podem ou não possuir parâmetros.

Métodos em Python



Obrigatório



def nome_do_método (self, parâmetros)

```
def depósito (self, valor):  
    self.saldo += valor  
  
def saque(self, valor):  
    if (self.saldo > 0):  
        self.saldo -= valor  
    else:  
        print("Saldo Insuficiente")
```

Método Construtor

- Determina que ações devem ser executadas quando um objeto é criado;
- Podem ou não possuir parâmetros.
- `def __init__(self, parâmetros)`

Implementando nossa Conta Corrente

```
1 class Conta:
2
3     def __init__(self, titular, agência, número, saldo=500):
4         self.titular = titular
5         self.agência = agência
6         self.número = número
7         self.saldo = saldo
8
9
10    def depósito (self, valor):
11        self.saldo += valor
12
13    def saque(self, valor):
14        self.saldo -= valor
15
16
17    conta = Conta('Titular: Gustavo Molina', 'Agência: 739', 'Número: 1234-5')
18    print(conta.titular, conta.agência, conta.número, conta.saldo)
19    print(conta.depositó(1000), conta.saldo)
20    print (conta.saque(800), conta.saldo)
```



Encapsulamento

Classes VS Instâncias



Instâncias

Classe

CONTA1
AGENCIA = "3133-1"
CONTA = "21312-0"

CONTA2
AGENCIA = "2132-1"
CONTA = "91212-0"

CONTABANCARIA

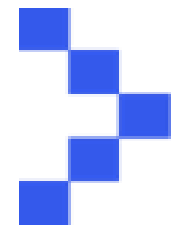
- AGENCIA
- CONTA
- DEPOSITO(VALOR)
- SAQUE(VALOR)
- MOSTRA_SALDO()

{ **Atributos**

{ **Métodos**



Encapsulamento



Encapsulamento é um dos pilares da programação OO, segundo o qual procuramos **esconder de clientes** (usuários de uma classe) todas as **informações que não são necessárias** ao uso da classe.

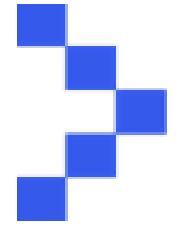
Encapsulamento – Cálculo de Salário

```
class Funcionario:
    def __init__(self, nome, cargo, valor_hora_trabalhada):
        self.nome = nome
        self.cargo = cargo
        self.valor_hora_trabalhada = valor_hora_trabalhada
        self.horas_trabalhadas = 0
        self.salario = 0

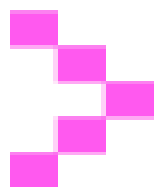
    def registra_hora_trabalhada(self):
        self.horas_trabalhadas += 1

    def calcula_salario(self):
        self.salario = self.horas_trabalhadas * self.valor_hora_trabalhada
```

Encapsulamento – Cálculo de Salário



Na classe anterior, o salário de um funcionário é calculado com base no valor por hora trabalhada e na quantidade de horas trabalhadas. A classe é razoável, mas possui alguns problemas. Informações sigilosas de funcionários, como o salário, são expostas a clientes da classe, o que nem sempre é desejável. Além disso, é possível alterar o salário final de um funcionário sem utilizar a função calcula salário.



Na implementação feita, nada impede que um cliente digite: `f.salario = 1000000`, o que alteraria o salário final do funcionário sem que este seja atrelado ao número de horas trabalhadas. O mesmo problema acontece com a variável `horas_trabalhadas`.

Encapsulamento – Cálculo de Salário

```
class Funcionario:
    def __init__(self, nome, cargo, valor_hora_trabalhada):
        self.nome = nome
        self.cargo = cargo
        self.valor_hora_trabalhada = valor_hora_trabalhada
        self.__horas_trabalhadas = 0 ❶
        self.__salario = 0 ❷

    def registra_hora_trabalhada(self):
        self.horas_trabalhadas += 1

    def calcula_salario(self):
        self.__salario = self.__horas_trabalhadas * self.valor_hora_trabalhada
```

- ❶ Mudamos o nome da variável `horas_trabalhadas` para começar com `__`.
- ❷ Fazemos o mesmo com a variável `salario`.

Encapsulamento

- Em Python, existe uma convenção (“acordo”) de que os dados ou métodos cujo nome começa com `__` (dois *underscores*) não deveriam ser acessados fora da classe.
- Porém dado que essa forma de encapsulamento é somente um indicativo de que dados e métodos cujo nome começa com `__` não devem (mas podem) ser acessados, ainda assim podemos alterar a variável salário, conforme mostrado no exemplo abaixo.

```
pedro = Funcionario('Pedro', 'Gerente de Vendas', 50)
pedro.__salario = 100000
print(pedro.__salario)
```

Encapsulamento

```
class Funcionario:
    def __init__(self, nome, cargo, valor_hora_trabalhada):
        self.nome = nome
        self.cargo = cargo
        self.valor_hora_trabalhada = valor_hora_trabalhada
        self.__salario = 0
        self.__horas_trabalhadas = 0

    @property
    def salario(self): ❶
        return self.__salario

    @salario.setter
    def salario(self, novo_salario): ❷
        raise ValueError("Impossivel alterar salario diretamente. Use a funcao calcula_salario().")

    def registra_hora_trabalhada(self):
        self.__horas_trabalhadas += 1

    def calcula_salario(self):
        self.__salario = self.__horas_trabalhadas * self.valor_hora_trabalhada

pedro = Funcionario('Pedro', 'Gerente de Vendas', 50)
pedro.salario = 100000 ❸
```

1- Criou-se a propriedade salário

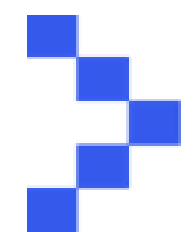
2- Restringiu-se o acesso à propriedade salário e instruímos os clientes a alterarem o valor da variável salario usando a função calcula_salario().

3- Uma tentativa de alterar a propriedade salario sem usar a função calcula_salario() resulta em erro.

Encapsulamento

- ❏ O comportamento padrão em Python é que as variáveis e métodos são completamente visíveis a clientes de uma classe (**método público**).
- ❏ Por meio de convenções de nome (uso do `__`) e de outros recursos como os decoradores (`@property`) é possível restringir a visibilidade de variáveis e métodos de uma classe (**método privado**).
- ❏ Além dos métodos públicos e privados, temos ainda uma 3ª opção que consiste no **método protected**, cujo nome começa com `_` (um *underscore*), e consiste em deixar os métodos visíveis apenas na classe base e nas classes derivadas.

Praticando



Vamos implementar a nossa Conta Corrente.



Exercício de Fixação 1 – Classe Bomba de Combustível



- a. Crie uma classe chamada `bombaCombustível`, com no mínimo esses atributos:
 - i. `tipoCombustivel`.
 - ii. `valorLitro`.
 - iii. `quantidadeCombustivel`.
 - b. A classe deve possuir no mínimo esses métodos:
 - i. `abastecerPorValor()` - método onde é informado o valor a ser abastecido e mostra a quantidade de litros que foi colocada no veículo.
 - ii. `abastecerPorLitro()` - método onde é informado a quantidade em litros de combustível e mostra o valor a ser pago pelo cliente.
 - iii. `alterarValor()` - altera o valor do litro do combustível.
 - iv. `alterarCombustivel()` - altera o tipo do combustível.
 - v. `alterarQuantidadeCombustivel()` - altera a quantidade de combustível restante na bomba.
- OBS: Sempre que acontecer um abastecimento é necessário atualizar a quantidade de combustível total na bomba.

Exercício de Fixação 2 - Tv

Faça um programa que simule um televisor criando-o como um objeto.

O usuário deve ser capaz de informar o número do canal e aumentar ou diminuir o volume.

Certifique-se de que o número do canal e o nível do volume permanecem dentro de faixas válidas.



Por hoje é só!
Obrigado! =)

Prof. Gustavo Molina

gmolina@thefutureisblue.me