



# Algoritmos Computacionais

**Conteúdo:** Introdução à programação orientada a objetos


Prof. Dsc. Giomar Sequeiros  
[giomar@eng.uerj.br](mailto:giomar@eng.uerj.br)

# Encapsulamento

# Encapsulamento em python

- Em Python existem **somente public** e **private** e eles são **definidos no próprio nome** do atributo ou método.
- Atributos ou métodos iniciado por **dois sublinhados** (underline) são **privados** e todas as **outras formas** são **públicas**
- Exemplo:

```
class Teste:  
    a = 1 # atributo público  
    __b = 2 # atributo privado
```



Testando o programa:

```
>>> t1 = Teste()  
>>> t1.a  
1  
>>> t1.__b  
Traceback (most recent call last):  
  File "<input>", line 1, in <module>  
AttributeError: 'Teste' object has no attribute '__b'
```

# Propriedades

---

- Para manipular **atributos privados** é comum utilizar métodos públicos com prefixo **get** e **set** em linguagens como Java
- Python possui o uso do **decorator @property** para trabalhar com atributos, pois:
  - A sintaxe usada para **definir propriedades** é muito concisa e legível.
  - Podemos **acessar** os **atributos** da instância exatamente como se **fossem atributos**
  - Usando **@property**, podemos "reutilizar" o nome de uma propriedade para evitar a criação de novos nomes para os **getters**, **setters** e **deleters**.

# Propriedades: Exemplo

- Consideremos a classe **Circulo**, adicionamos uma propriedade para recuperar, modificar e remover o atributo raio.

```
class Circulo:
    def __init__(self, raio):
        self.__raio = raio

    @property
    def raio(self):
        print("Get raio")
        return self.__raio

    @raio.setter
    def raio(self, value):
        print("Set raio")
        self.__raio = value

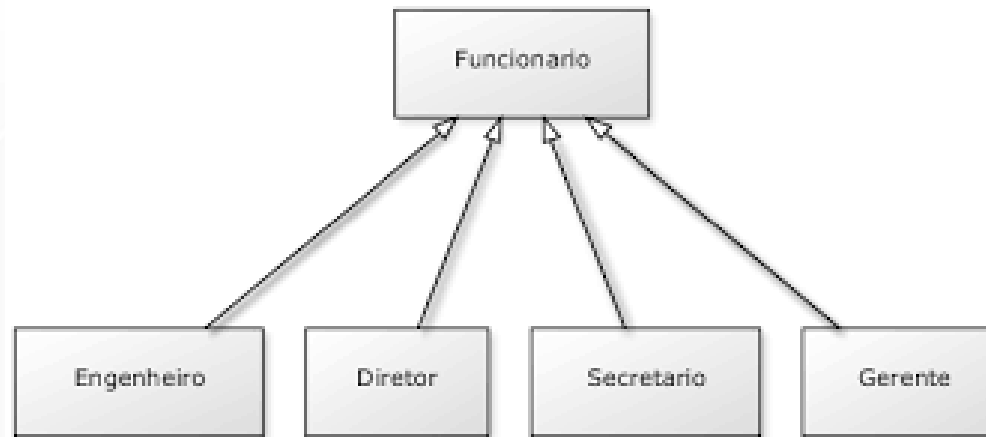
    @raio.deleter
    def raio(self):
        print("Delete raio")
        del self.__raio
```

Testando o programa:

```
>>> circ = Circulo(20)
>>> circ.raio
Get raio
20
>>> circ.raio = 100.0
Set raio
>>> circ.raio
Get raio
100.0
>>> del circ.raio
Delete raio
>>> circ.raio
Get raio
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 9, in raio
AttributeError: 'Circulo' object has no attribute '__raio'
```

# Conceitos básicos: Herança

- Herança é o mecanismo pelo qual uma **classe obtém as características e métodos de outra para expandi-la ou especializá-la**, ou seja, uma classe pode “herdar” características, métodos e atributos de outras classes.
- A herança constitui um mecanismo de **aproveitar código**.



- Existem 2 tipos de herança:
  - **Herança simples:** uma classe **herda** características de **apenas uma superclasse**.
  - **Herança múltipla:** uma classe **herda** características de **duas ou mais superclasses**.



# Herança simples em Python

- Considere o seguinte exemplo:

```
class Pai():  
    def __init__(self):  
        print("Construindo a classe Pai")
```

```
class Filha(Pai):  
    def __init__(self):  
        super(Filha, self).__init__()
```

Classe Pai

- O **super()** é utilizado entre **heranças** de classes, ele nos proporciona **extender/subscrever métodos** de uma super classe (classe pai) para uma sub classe (classe filha), através dele definimos um novo comportamento para um determinado método construído na classe pai e herdado pela classe filha.

# Herança simples em Python: Exemplo 1

- Exemplo: Consideremos a classe **Retangulo** e a classe **Quadrado** derivado dela.

```
class Retangulo:
    def __init__(self, ladoA, ladoB):
        self.ladoA = ladoA
        self.ladoB = ladoB

    def area(self):
        return self.ladoA * self.ladoB

    def perimetro(self):
        return 2*self.ladoA + 2*self.ladoB
```

Testando o programa:

```
>>> r = Retangulo(2,3)
>>> r.area()
6
>>> r.perimetro()
10
```

```
class Quadrado(Retangulo):
    def __init__(self, lado):
        super(Quadrado, self).__init__(lado, lado)
```

Testando o programa:

```
>>> c = Quadrado(5)
>>> c.area()
25
>>> c.perimetro()
20
```



# Herança simples em Python: Exemplo 2

- Criamos a classe **ContaEspecial** derivada da classe **Conta**

```
class ContaEspecial(Conta):  
    def __init__(self, cliente, numero, saldo = 0, limite = 0):  
        Conta.__init__(self, cliente, numero, saldo)  
        self.limite = limite  
  
    def saque(self, valor):  
        if self.saldo + self.limite >= valor:  
            self.saldo -= valor  
            self.operacoes.append(["Saque", valor])
```

- Teste:

```
maria = Cliente("Maria da Silva", "(21)99654-3210")  
conta2 = ContaEspecial(maria, 2, 500, 1000)  
conta2.deposito(300)  
conta2.saque(1500)  
conta2.extrato()
```

Saída:

```
Extrato Conta N° 2  
Deposito    500.00  
Deposito    300.00  
Saque      1500.00  
Saldo:     -700.00
```

# Herança múltipla em Python

A herança é denominada múltipla quando uma classe **herda** características de **duas ou mais superclasses**.

Exemplo:

```
class Veiculo:
    def __init__(self, cor, modelo):
        self.cor = cor
        self.modelo = modelo
```

```
class Dispositivo:
    def __init__(self):
        self._voltagem = 220
```

```
class Carro(Veiculo, Dispositivo):
    def __init__(self, cor, modelo, ano):
        Veiculo.__init__(self, cor, modelo)
        Dispositivo.__init__(self)
        self.ano = ano
```

A classe **Carro** é derivada da classe **Veiculo** e **Dispositivo**

# Exercícios Encapsulamento de Herança

# Exemplo 1

Crie a classe Tempo com atributos hora, minuto e segundo conforme mostrado a seguir

```
class Tempo:
    '''Classe tempo com propriedades de leitura/escrita'''
    def __init__(self, hora=0, minuto=0, segundo=0):
        '''Inicializa cada atributo'''
        {
            self.__hora = hora
            self.__minuto = minuto
            self.__segundo = segundo
        }
```

Atributos privados (encapsulamento)

Documentação da classe e do construtor

# Exemplo 1

- Dentro da classe Tempo acrescente as propriedades (**@property**) para o atributo privado **\_\_hora**

**decorator** usado para dar uma funcionalidade "especial" a certos métodos para que ajam como getters, setters

```
@property
def hora(self):
    '''Retorna o atributo hora'''
    return self.__hora

@hora.setter
def hora(self, hora):
    '''Modifica o atributo hora'''
    if not(0 <= hora < 24):
        print(f'Hora ({hora}) deve estar entre 0-23')
    self.__hora = hora
```

Permite alterar o atributo hora  
(método especial setter)

# Exemplo 1

- Dentro da classe Tempo acrescente as propriedades (property) para o atributo privado **\_\_minuto**

```
@property
def minuto(self):
    '''Retorna o atributo minuto'''
    return self.__minuto

@minuto.setter
def minuto(self, minuto):
    '''Modifica o atributo hora'''
    if not(0 <= minuto < 60):
        print(f'Minuto ({minuto}) deve estar entre 0-59')
    self.__minuto = minuto
```

# Exemplo 1

- Dentro da classe Tempo acrescente as propriedades (property) para o atributo privado `__segundo`

```
@property
def segundo(self):
    '''Retorna o atributo segundo'''
    return self.__segundo

@segundo.setter
def segundo(self, segundo):
    '''Modifica o atributo segundo'''
    if not(0 <= segundo < 60):
        print(f'Segundo ({segundo}) deve estar entre 0-59')
    self.__segundo = segundo
```



# Exemplo 1

- Adicionamos o **método especial** `__str__` para retornar uma string de um objeto Time

```
def __str__(self):  
    '''método especial para retornar uma representação de  
    string de um objeto.'''  
    return '%.2d:%.2d:%.2d' % (self.hora, self.minuto, self.segundo)
```

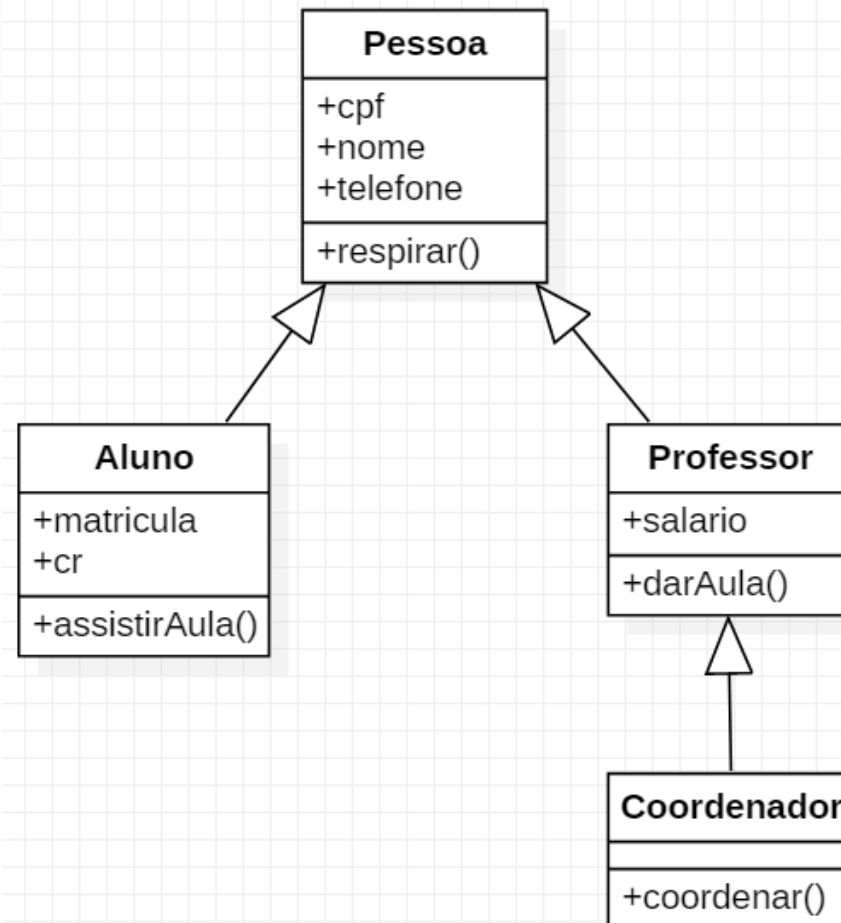
# Exemplo 1

- Criamos o teste para a classe Tempo

```
if __name__ == '__main__':  
    # Criamos um objeto t1  
    t1 = Tempo()  
    print('Tempo 1:', t1)  
    t1.hora = 13  
    t1.minuto = 25  
    t1.segundo = 42  
    print('Tempo 1 atualizado:', t1)  
  
    # Criamos um objeto t2  
    t2 = Tempo(9, 40, 14)  
    print('Tempo 2:', t2)
```

## Exemplo 2

Considere o diagrama de classes abaixo, implemente as classes em Python



# Exemplo 2

---

- Criamos a classe Pessoa

```
class Pessoa:
    def __init__(self, cpf, nome, telefone):
        self.cpf = cpf
        self.nome = nome
        self.telefone = telefone

    def respirar(self):
        print('Respirando...')
```

## Exemplo 2

- Criamos a classe Aluno derivada de Pessoa

A classe Aluno herda de Pessoa

```
class Aluno(Pessoa):  
    def __init__(self, cpf, nome, telefone, matricula, cr):  
        Pessoa.__init__(self, cpf, nome, telefone)  
        self.matricula = matricula  
        self.cr = cr  
  
    def assistirAula(self):  
        print('Assistindo aula...')
```

Chamamos o construtor da classe pai (Pessoa), também podemos usar a função **super()**

## Exemplo 2

- Criamos a classe Professor derivado de Pessoa e classe Coordenador derivado de Professor

```
class Professor(Pessoa):  
    def __init__(self, cpf, nome, telefone, salario):  
        Pessoa.__init__(self, cpf, nome, telefone)  
        self.salario = salario  
  
    def darAula(self):  
        print('Dando aula...')
```

```
class Coordenador(Professor):  
    def __init__(self, cpf, nome, telefone, salario):  
        Professor.__init__(self, cpf, nome, telefone)  
  
    def coordenar(self):  
        print('Coordenado...')
```

# Exemplo 2

- Adicione o teste

```
if __name__ == '__main__':  
    p = Pessoa('12345678901', 'João da Silva', '(21)987-654321')  
    p.respirar()  
    a = Aluno('12345678901', 'Ana Claudia', '(21)9123-45678', '1234', 7)  
    a.respirar()  
    a.assistirAula()  
    # crie objetos da classe professor e coordenador...
```

Método respirar() da classe Pessoa

Crie objetos da classe professor e coordenador...



## Exemplo 3

Considere a classe `Animal` com atributos `nome` e `cor`, adicione o método `comer` conforme mostrado e salve no arquivo **`animal.py`**

```
class Animal():  
    def __init__(self, nome, cor):  
        self.__nome = nome  
        self.__cor = cor  
  
    def comer(self):  
        print(f"O {self.__nome} está comendo")
```

## Exemplo 3

Crie as classes Gato e Cachorro derivados de Animal e salve respectivamente como **gato.py** e **cachorro.py**

```
from animal import Animal

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

```
from animal import Animal

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

# Exemplo 3

Adicione o teste no arquivo **teste.py**

```
import gato, cachorro

if __name__ == '__main__':
    gato = gato.Gato('Filomeno', 'Preto')
    cachorro = cachorro.Cachorro('Mika', 'Branco')

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

Saída:

```
O Filomeno está comendo
O Mika está comendo
```

## Exemplo 4

Crie a classe Programa com atributos nome, ano e likes, sendo que nome e likes são privados

```
class Programa:
    def __init__(self, nome, ano):
        self.__nome = nome.title()
        self.ano = ano
        self.__likes = 0
```

# Exemplo 4

Na classe Programa acrescente **getters** para nome e like e um **setter** para nome

```
@property
def nome(self):
    return self.__nome

@property
def likes(self):
    return self.__likes

@nome.setter
def nome(self, novoNome):
    self.__nome = novoNome
```

# Exemplo 4

Na classe Programa acrescente os métodos darLike() e \_\_str\_\_

```
def darLike(self):  
    self.__likes += 1  
  
def __str__(self):  
    return f'{self.__nome} de {self.ano} - {self.__likes} likes'
```

# Exemplo 4

Crie a classe Filme derivada de programa Programa acrescente o método `__str__`

```
class Filme(Programa):  
    def __init__(self, nome, ano, duracao):  
        super().__init__(nome, ano)  
        self.duracao = duracao  
  
    def __str__(self):  
        return super().__str__() + f' - duração - {self.duracao} minutos'
```

Chamamos os métodos da classe pai Programa



## Exemplo 4

Crie a classe Serie derivada de programa Programa acrescente o método `__str__`

```
class Serie(Programa):  
    def __init__(self, nome, ano, temporadas):  
        super().__init__(nome, ano)  
        self.temporadas = temporadas  
  
    def __str__(self):  
        return super().__str__() + f' - {self.temporadas} temporadas'
```

# Exemplo 4

Crie um teste

```
if __name__ == '__main__':  
    vingadores = Filme('vingadores: ultimato', 2019, 182)  
    vingadores.darLike()  
    vingadores.darLike()  
    vingadores.darLike()  
  
    euphoria = Serie('euphoria', 2019, 2)  
    euphoria.darLike()  
    euphoria.darLike()  
  
    playlist = [vingadores, euphoria]  
  
    for programa in playlist:  
        print(programa)
```

# Exercício

Considere os seguintes objetos geométricos: retângulo, caixa, círculo e cilindro. Cada objeto geométrico deve possuir métodos para obter seu perímetro (figuras 2D), sua área (externa, no caso das 3D) e volume (figuras 3D). Crie um diagrama de classes que modele objetos geométricos de forma a aproveitar ao máximo suas características comuns. Depois implemente em python o modelo criado. As classes devem possuir construtores parametrizados, os quais devem ser utilizados para inicialização dos atributos dos objetos. Considere as informações abaixo para o cálculo do perímetro, área e volume dos objetos.

- Retângulo:
  - Área =  $\text{base} * \text{altura}$ ,
  - Perímetro =  $2 * \text{base} + 2 * \text{altura}$
- Caixa:
  - Volume =  $\text{base1} * \text{base2} * \text{altura}$ ,
  - Área =  $2 * (\text{base1} * \text{base2} + \text{base1} * \text{altura} + \text{base2} * \text{altura})$
- Círculo:
  - Área =  $3.14 * (\text{raio})^2$ ,
  - Perímetro =  $2 * 3.14 * \text{raio}$
- Cilindro:
  - Volume =  $3.14 * (\text{raio})^2 * \text{altura}$ ,
  - Área =  $2 * 3.14 * (\text{raio})^2 + 2 * 3.14 * \text{raio} * \text{altura}$

# Exercício

---

Crie duas classes: Funcionario e Gerente

- Gerente deve ser classe filha de Funcionario
- Os atributos de Funcionario são: nome, CPF, salário e departamento
- A classe Funcionario tem um método para bonificar o funcionário, acrescentando 10% ao salário dele
- A classe Gerente deve ter atributos adicionais de senha e número de funcionários gerenciados
- A classe Gerente tem dois métodos a mais:
  - Um método para autenticar a senha, que apenas compara a senha passada como argumento com o valor do atributo senha
  - Um método para bonificar o gerente, com o valor de 15%
- Crie objetos e teste suas classes