



Paradigmas de linguagens de programação em python

Professores:

Sebastião Rogério feat. Kayo Monteiro

Agenda

01

Programação Orientada a
objeto - Básico

03

Programação
Orientada a objeto -
Overriding

02

Programação Orientada
a objeto - Herança e
Polimorfismo

04

Programação
Orientada a objeto -
Encapsulamento



01

Programação Orientada a objeto – Básico

Declaração de Classes



```
class Pessoa():  
    # Atributos e métodos da classe  
class PessoaFisica():  
    # Atributos e métodos da classe
```

Criando construtor da classe


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

Uma classe é representada por **atributos** e **métodos**. Os atributos de uma classe representam as características que esta classe possui, já os métodos representam o comportamento da classe. É preciso definir o método especial **`__init__`** (duplo **`_`**) para inicializar os atributos de cada instância; O **`self`** refere-se a um instância da classe

Self



Python sempre passa o objeto (**self**) como primeiro argumento de todos os métodos, **self** refere-se a um instância da classe. Ao referir-se a um instância da classe um objeto, ele é usado para:

- indicar que os atributos ou métodos que estão sendo acessados pertencem ao próprio objeto;
 - diferir os atributos da classe em relação aos parâmetros e variáveis locais de um método;
- 

Atributos

Os atributos conferem às classes, às quais pertencem, as suas respectivas características. Vimos que a classe Pessoa possui alguns atributos:

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

Atributos



- Os atributos são acessados diretamente ou através de métodos **GET** e **SET**
- Aprendemos que o padrão é acessá-los via **GET** e **SET**, mas podemos fazer esse acesso diretamente, através do "." (ponto)
- Isso vai depender da visibilidade do atributo e de onde estamos tentando acessá-lo



Acessando Atributos:




Para acessar atributos de um objeto, utiliza-se o nome do objeto (e não o da classe), seguido do caractere ponto (.), mais o nome do atributo ou método que deseja-se acessar

`objeto.nomeAtributo`



Acessando Atributos:



```
peessoa1 = Pessoa("João", "M", "123456")  
print(peessoa1.nome)
```

Declarando Métodos:



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



```
def desativar(self):
    self.ativo = False
    print("A pessoa foi desativada com sucesso")
```

Métodos



Para criarmos este “comportamento” na classe Pessoa, utilizamos a palavra reservada **def**, que indica que estamos criando um método da classe, além do nome do método e seus atributos, caso possuam.

Depois disso, é só definir o comportamento que este método irá realizar. Neste caso, o método vai alterar o valor do atributo ativo de “True” para “False” e imprimir a mensagem “A pessoa foi desativada com sucesso”.

Métodos

Para criarmos este método, usamos a palavra reservada `def` dentro da classe, além do nome do método.

Depois disso, é só definir o que o método vai realizar. Neste caso, vamos definir o método `desativar` para "False" e imprimir "sucesso".

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

    def desativar(self):
        self.ativo = False
        print("A pessoa foi desativada com sucesso")

pessoa1 = Pessoa("João", "M", "123456", True)
pessoa1.desativar()
```



Métodos

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

    def desativar(self):
        self.ativo = False
        print("A pessoa foi desativada com sucesso")

pessoa1 = Pessoa("João", "M", "123456", True)
print(pessoa1.ativo)
pessoa1.desativar()
print(pessoa1.ativo)
```

Encapsulamento

O atributo “ativo”: ele é acessível para todo mundo. Ou seja, mesmo possuindo o método “desativar”, é possível alterar o valor do atributo “ativo” sem qualquer problema.

Este comportamento do nosso programa dá brechas para que um usuário possa ser ativado ou desativado sem passar pelo método responsável por isso. Por isso, para corrigir este problema, devemos recorrer a um pilar importantíssimo da Orientação à Objetos: o encapsulamento.

**VEREMOS NOVAMENTE
MAIS A FRENTE**

Métodos [Revisão]



Vimos que o estado da classe é definido pelos valores dos atributos

- O comportamento da classe, porém, é definido pelos métodos
- Os métodos permitem que o programador modularize um programa, separando suas tarefas em unidades autocontidas
- Métodos bem programados ajudam na reutilização de código


Exercício



Crie uma classe que recebe o número de uma conta corrente, o construtor deverá receber o número da conta e o saldo inicial. A conta deverá ter os métodos de debitar e creditar o valor no saldo.

Na célula seguinte realize os testes dos métodos de debitar, creditar e consulte o saldo em conta.

Possível Solução



```
class ContaCorrente:
    def __init__(self, numero, saldo):
        self.numero = numero
        self.saldo = saldo
    def debitar(self, valor):
        self.saldo = self.saldo-valor
    def creditar(self, valor):
        self.saldo = self.saldo+valor
```

```
c = ContaCorrente("123123123", 1000)
c.debitar(500)
c.saldo
c.creditar(4000)
c.saldo
```

Exercício



Escreva uma Classe que simule o funcionamento de uma calculadora, a classe deve ter dois atributos e possuir pelo menos 4 métodos (soma, subtração, multiplicação e divisão)

Na célula seguinte crie um objeto e testes todos os métodos...

Possível Solução

```
class Calculadora:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def soma (self):
        valor = self.x + self.y
        print("O valor da soma: ", valor)
    def sub (self):
        valor = self.x - self.y
        print("O valor da subtracao: ", valor)
    def mul (self):
        valor = self.x * self.y
        print("O valor da multiplicacao: ", valor)
    def div (self):
        valor = self.x / self.y
        print("O valor da divisao: ", valor)
```



02

Herança e Polimorfismo

O que são Heranças?

- Herança é uma forma de reutilização de software, onde uma nova classe é criada absorvendo dados de uma classe existente
- Esta nova classe pode ter características mais específicas ou modificadas em comparação com a classe antiga/absorvida
- Com a herança, o tempo de desenvolvimento de um software é reduzido e a depuração é facilitada

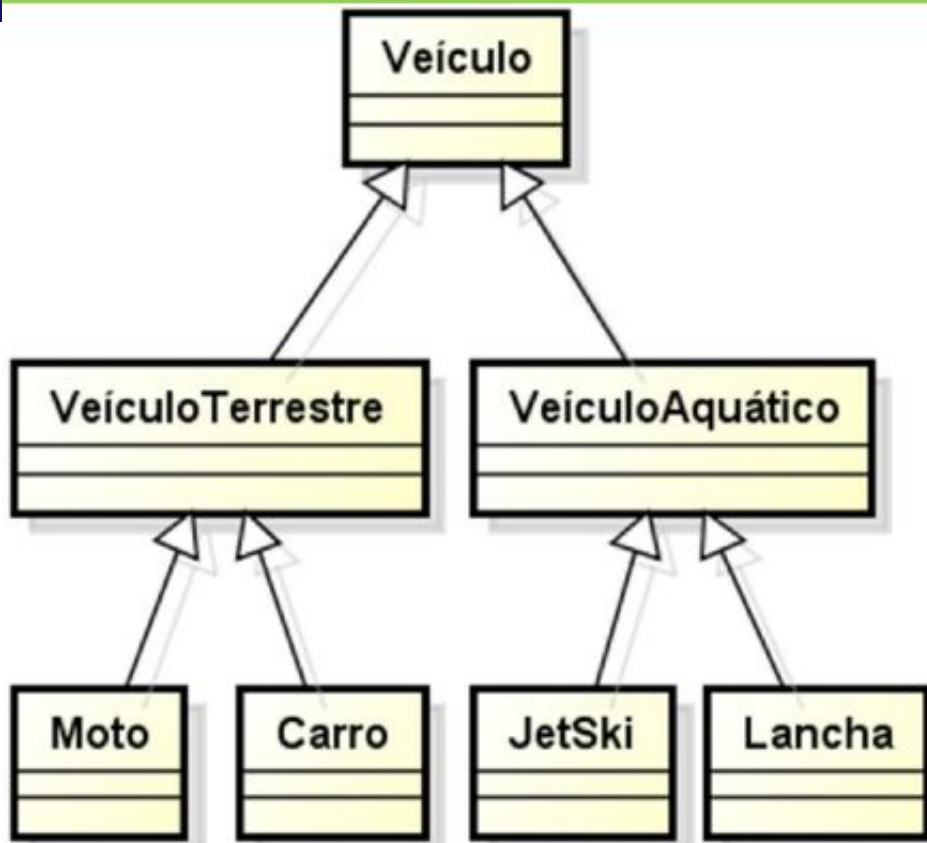
O que são Heranças?

A classe nova é chamada de **SUBCLASSE**, já a classe antiga, que é absorvida pela nova, é chamada de **SUPERCLASSE**

A herança pode se dar em vários níveis, formando uma hierarquia

- A classe imediatamente superior é uma **SUPERCLASSE direta**
- Uma classe que não seja imediatamente superior é uma **SUPERCLASSE indireta**

Herança



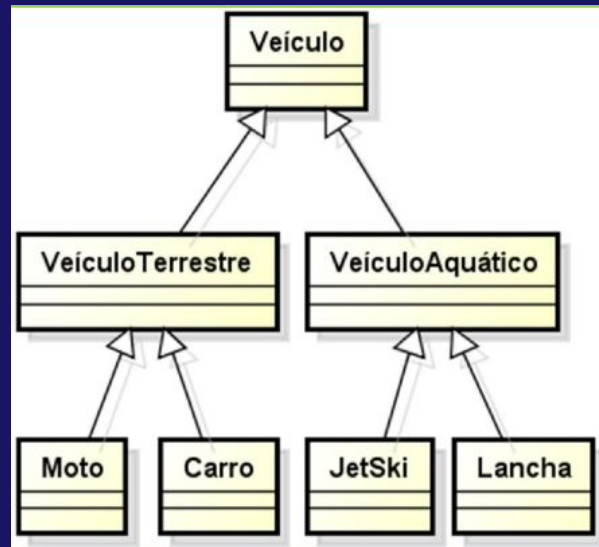
Heranças

1-A Moto é um VeículoTerrestre, mas também é um Veículo

2-A Lancha é um Veículo, porém não é VeículoTerrestre

3-Moto e JetSki são Veículos, porém um é VeículoTerrestre e o outro é VeículoAquático

4-VeículoAquático e VeículoTerrestre compartilham dados da classe veículo



Outro Exemplo

SUPERCLASSE

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

Outro Exemplo

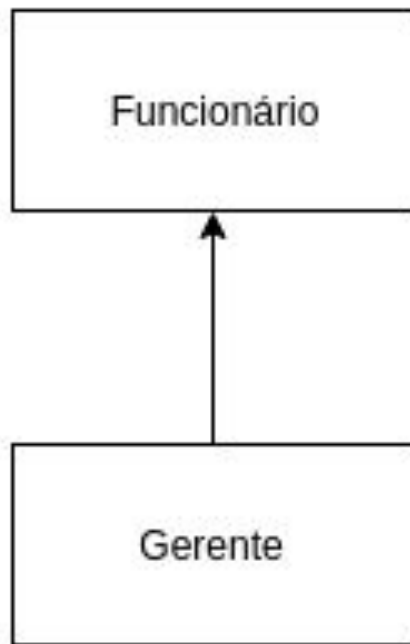


```
class Gerente(Funcionario):  
    def __init__(self, nome, cpf, salario, idade, equipe):  
        super().__init__(nome, cpf, salario)  
        self.idade = idade  
        self.equipe = equipe
```

Outro Exemplo



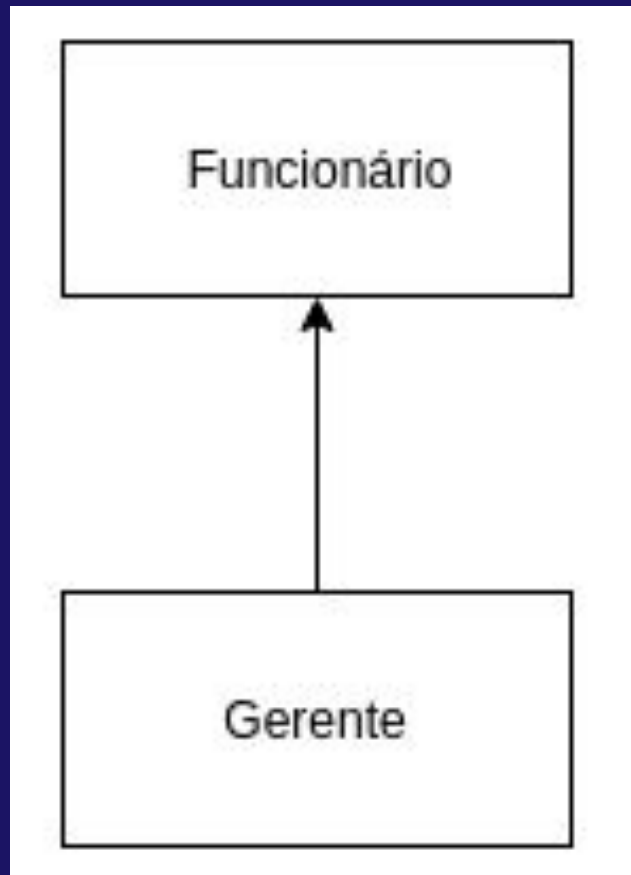
```
class Gerente(Funcionario):  
    def __init__(self, nome, idade, equipe, salario):  
        super().__init__(nome, idade, equipe)  
        self.idade = idade  
        self.equipe = equipe
```



```
    def __init__(self, nome, idade, equipe):  
        super().__init__(nome, idade, equipe)  
        self.idade = idade  
        self.equipe = equipe
```

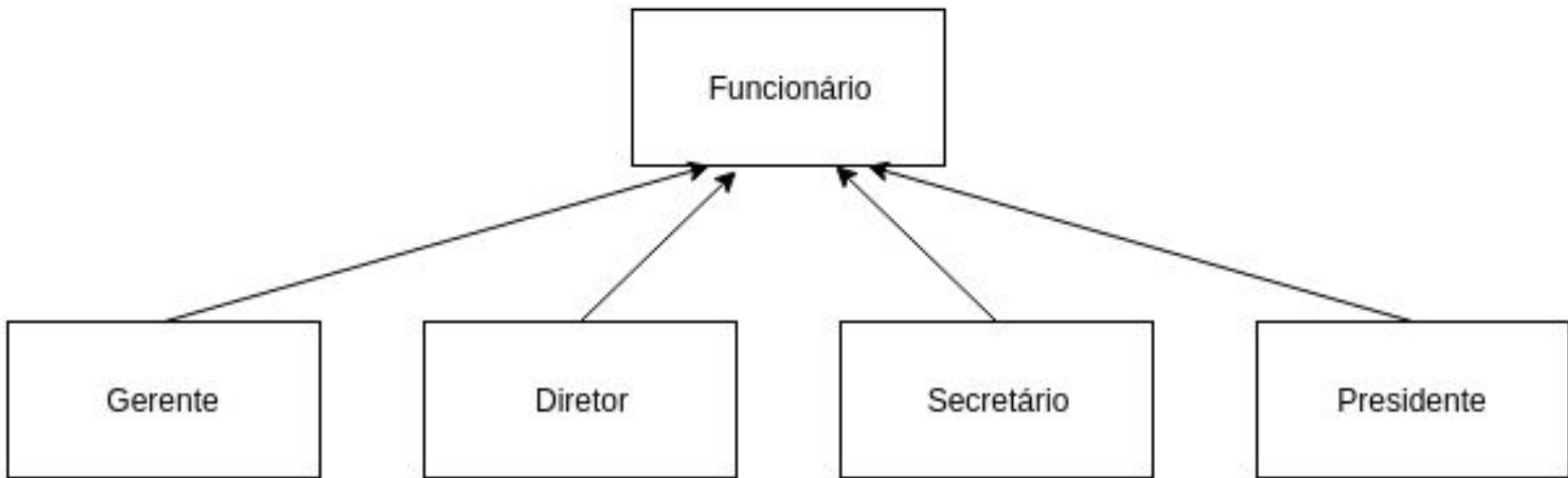
Herança

A nomenclatura mais encontrada é que Funcionario é a **superclasse** de Gerente, e Gerente é a **subclasse** de Funcionario. Dizemos também que todo Gerente é um Funcionario. Outra forma é dizer que Funcionario é a classe mãe de Gerente e Gerente é a classe filha de Funcionario.



Herança – Exercício

Com base na relação anterior, vamos criar outras 3 subclasses de **Funcionário**. Os funcionários comuns recebem 10% do valor do salário, e os gerentes e diretores, 15%.



Solução parcial

```
class Funcionario():  
    def __init__(self, nome, cpf, salario):  
        self._nome = nome  
        self._cpf = cpf  
        self._salario = salario  
    def bonitificacao(self):  
        return ((self._salario*0.10)+self._salario)
```

```
class Gerente(Funcionario):  
    def __init__(self, nome, cpf, salario, idade, equipe):  
        super().__init__(nome, cpf, salario)  
        self.idade = idade  
        self.equipe = equipe  
    def bonitificacao(self):  
        return ((self._salario*0.15)+self._salario)
```



Polimofirsmo

Polimorfismo

O polimorfismo é mais um princípio fundamental da orientação a objetos

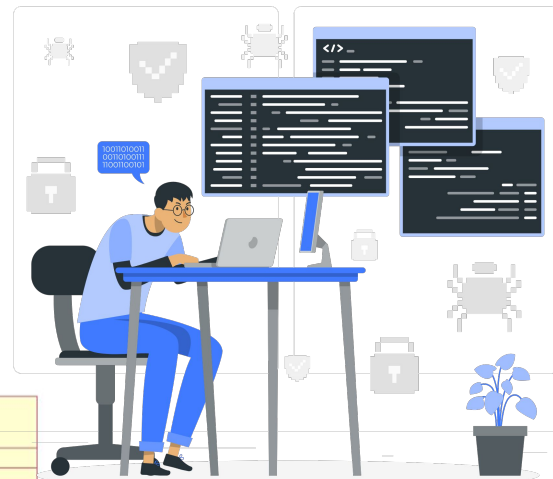
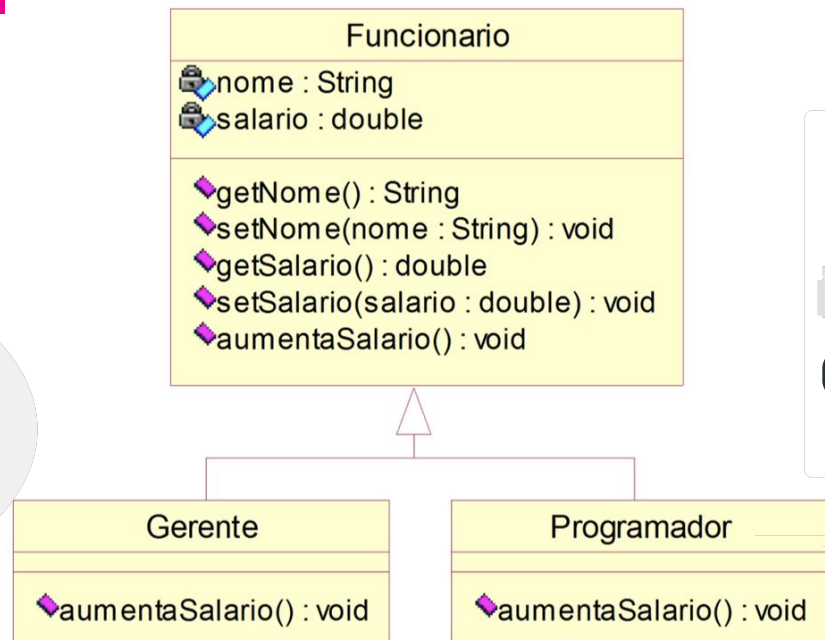
- Significa, ao pé da letra, “várias formas”
- Não confundir com sobrecarga (overload "veremos no futuro")
- Ele permite que classes pertencentes a uma mesma linha de herança possuam comportamentos diferentes para o mesmo método



03

Programação Orientada a objeto – Overriding

Vamos a um exemplo



Sobreposição de Métodos

Contextualização

- Imagine que o método *umentaSalario()* da classe *Funcionario* no exercício anterior possui uma implementação, ou seja, o método não é mais abstrato.
- E que as subclasses *Gerente* e *Programador* mantêm suas implementações para o método *umentaSalario()*.

Entendendo melhor

- Quando uma subclasse declara um método com o mesmo nome, mesmo tipo de retorno e mesma lista de parâmetros de um método da sua superclasse, dizemos que ocorreu uma sobreposição ou redefinição de método (overriding).
- Um método redefinido em uma subclasse oculta o método da classe ancestral a partir da subclasse.
- Métodos privados não podem ser sobrepostos.

Entendendo melhor

- O nome e a lista de parâmetros de um método é chamado de assinatura do método.

Sobreposição x Sobrecarga

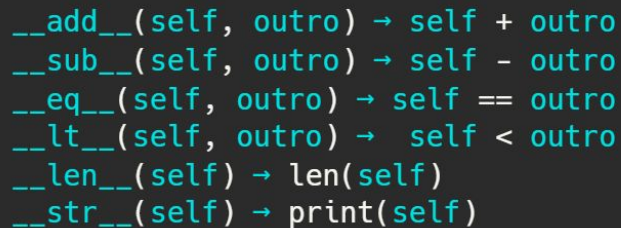
- Sobrecarga (*overloading*) significa que pode-se ter métodos de mesmo nome, mas que difiram na lista de parâmetros.
- Ou seja, métodos sobrecarregados não possuem mesma assinatura.

Sobreposição de Métodos

Dynamic Binding

- A ligação entre a assinatura de um método e o método por ela designado efetua-se em tempo de execução.
- Este mecanismo é conhecido como ligação dinâmica (*dynamic binding*).

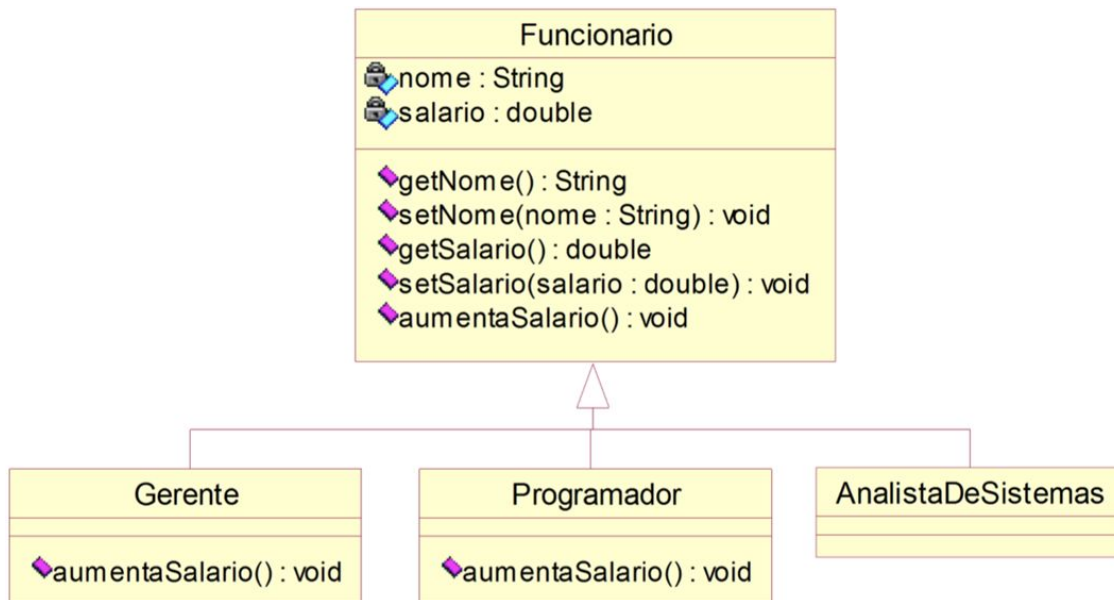
Relembrando..



```
__add__(self, outro) → self + outro  
__sub__(self, outro) → self - outro  
__eq__(self, outro) → self == outro  
__lt__(self, outro) → self < outro  
__len__(self) → len(self)  
__str__(self) → print(self)
```

Os métodos especiais **sobrescrevem** os métodos e operações originais, ou sejam, **fazem *overriding***

Exercício 1



Exercício 1

Considerações

- A classe Funcionario, Gerente e Programador!
- Uma chamada ao *umentaSalario()* do Funcionario aumenta seu salário em 5%.
- Uma chamada ao *umentaSalario()* do Gerente aumenta seu salário em 10%.
- Uma chamada ao *umentaSalario()* do Programador aumenta seu salário em 20%.

Exercício 1

Considerações

Implemente uma aplicação que declara três variáveis do tipo **Funcionario** e cria três objetos um do tipo **Gerente**, outro do tipo **Programador** e o terceiro do tipo **AnalistaDeSistemas**. Em seguida, o programa deve oferecer um menu para o usuário com as seguintes opções:

- **Imprimir dados** – O usuário deverá informar se ele deseja imprimir os dados do Gerente, do Programador ou do AnalistaDeSistemas.
- **Aumentar salário** – O usuário deverá informar se ele deseja aumentar o salário do Gerente, do Programador ou do AnalistaDeSistemas.

Exercício 2

- Implemente uma classe Conta que contenha os atributos nome do cliente, número da conta, saldo e limite. Estes valores deverão ser informados no construtor, sendo que o limite não poderá ser maior que o valor do salário mensal do cliente.
- Implemente também um método depósito e outro método saque. O método saque retorna um booleano indicando se o saque pôde ser efetuado ou não.
- Implemente uma classe ContaEspecial que funciona da mesma forma que a classe Conta, mas que aceita um limite de até 3 vezes o valor do salário do cliente.

04

Programação Orientada a objeto – Encapsulamento

Vamos entender

- Encapsulamento é o que se faz quando se restringe o acesso aos dados (atributos) de uma classe (*information hiding*);
- A ideia é fazer da classe uma cápsula, onde seus atributos só poderão ser acessados por determinados métodos;
- Pode-se alcançar o encapsulamento de dados configurando os chamados modificadores de visibilidade para tornar os atributos privados (encapsulados) e os métodos públicos.

Quais benefícios?

- Proteção dos atributos da classe de acessos indevidos ou acidentais.
- Possibilidade de definir regras para alteração dos valores dos atributos da classe

Modificadores de visibilidade

- Especificam quais classes têm acesso aos elementos (classe, atributos, métodos e construtores) de uma determinada classe.
 - `public`
 - Classe pode ser instanciada por qualquer outra classe.
 - Atributos e métodos são acessíveis (leitura e escrita) por objetos de qualquer classe

Modificadores de visibilidade

- Especificam quais classes têm acesso aos elementos (classe, atributos, métodos e construtores) de uma determinada classe.
 - `private`
 - Atributos só podem ser acessados pelos métodos dos objetos da mesma classe
 - Métodos só podem ser chamados por métodos da própria classe

Modificadores de visibilidade

- Especificam quais classes têm acesso aos elementos (classe, atributos, métodos e construtores) de uma determinada classe.
 - `protected`
 - Atributos e métodos são acessíveis dentro da própria classe, das subclasses e das classes que fazem parte do mesmo pacote

Como isso funciona em Python?

- Diferentemente de outras linguagens como C++ e Java, Python não tem as palavras reservadas *public*, *protected* ou *private* para definir as regras de acesso.
- Sendo assim, podemos considerar que, por padrão, todos os atributos são públicos
- Apesar da ausência de palavras reservadas próprias para encapsulamento, Python permite utilizar esses conceitos através do uso do "_" (*underline*) na frente das variáveis e funções

```
class Encapsulamento:
    def __init__(self, a, b, c):
        self.public = a
        self._protected = b
        self.__private = c

    def metodo_public(self):
        print("publico")

    def _metodo_protected(self):
        print("protected")

    def __metodo_private(self):
        print("private")

encapsulamento = Encapsulamento(1, 2, 3)

print(encapsulamento.public)
print(encapsulamento._protected)
print(encapsulamento.__private)
```

```
1
2
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-2-0534a7944b40> in <module>()
     18 print(encapsulamento.public)
     19 print(encapsulamento._protected)
--> 20 print(encapsulamento.__private)

AttributeError: 'Encapsulamento' object has no attribute '__private'
```

```
class Encapsulamento:
    def __init__(self, a, b, c):
        self.public = a
        self._protected = b
        self.__private = c

    def metodo_public(self):
        print("publico")

    def _metodo_protected(self):
        print("protected")

    def __metodo_private(self):
        print("private")

encapsulamento = Encapsulamento(1, 2, 3)

encapsulamento.metodo_public()
encapsulamento._metodo_protected()
encapsulamento.__metodo_private()
```

```
publico
protected
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-3-f6d725ce1373> in <module>()
      22 encapsulamento.metodo_public()
      23 encapsulamento._metodo_protected()
----> 24 encapsulamento.__metodo_private()
```

```
AttributeError: 'Encapsulamento' object has no attribute '__metodo_private'
```

```
class Encapsulamento:
    def __init__(self, a, b, c):
        self.public = a
        self._protected = b
        self.__private = c

    def metodo_public(self):
        print("publico")

    def _metodo_protected(self):
        print("protected")

    def __metodo_private(self):
        print("private")

encapsulamento = Encapsulamento(1, 2, 3)

encapsulamento.metodo_public()
encapsulamento._metodo_protected()
```

public
private
protected

Métodos e atributos privados só
são acessíveis dentro da própria
classe

```
class Encapsulamento:
    def __init__(self, a, b, c):
        self.public = a
        self._protected = b
        self.__private = c

    def metodo_public(self):
        print("publico")

    def _metodo_protected(self):
        print("protected")

    def __metodo_private(self):
        print("private")

encapsulamento = Encapsulamento(1, 2, 3)

print(encapsulamento.public)
print(encapsulamento._protected)
print(encapsulamento.get_private())

encapsulamento.metodo_public()
encapsulamento._metodo_protected()
```

1

2

3

public

private

protected

**Métodos e atributos privados só
são acessíveis dentro da própria
classe**

Getters & Setters

- De maneira geral, utilizamos a nomenclatura e getters e setters para acessar e gerenciar acesso aos atributos encapsulados:
 - get: utilizado para acessar o valor mantido por um atributo.
 - set: utilizado para alterar o valor mantido por um atributo.

The background is a dark navy blue. It features several horizontal bars of different colors: a cyan bar at the top, a magenta bar below it, a white bar to the right, and another magenta bar at the bottom right. On the left, there is a white bar and a cyan bar. A large magenta rectangle serves as a backdrop for the word 'Resumo'.

Resumo

Referências

- Materiais baseados no material do Prof. Pedro Braga: <https://phbraga.com/plp-python/>

To be...



CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics & images by Freepik.