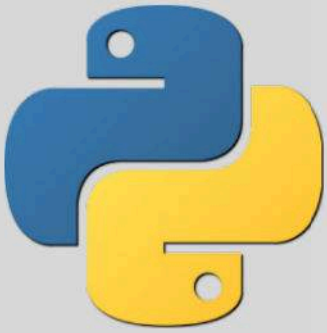
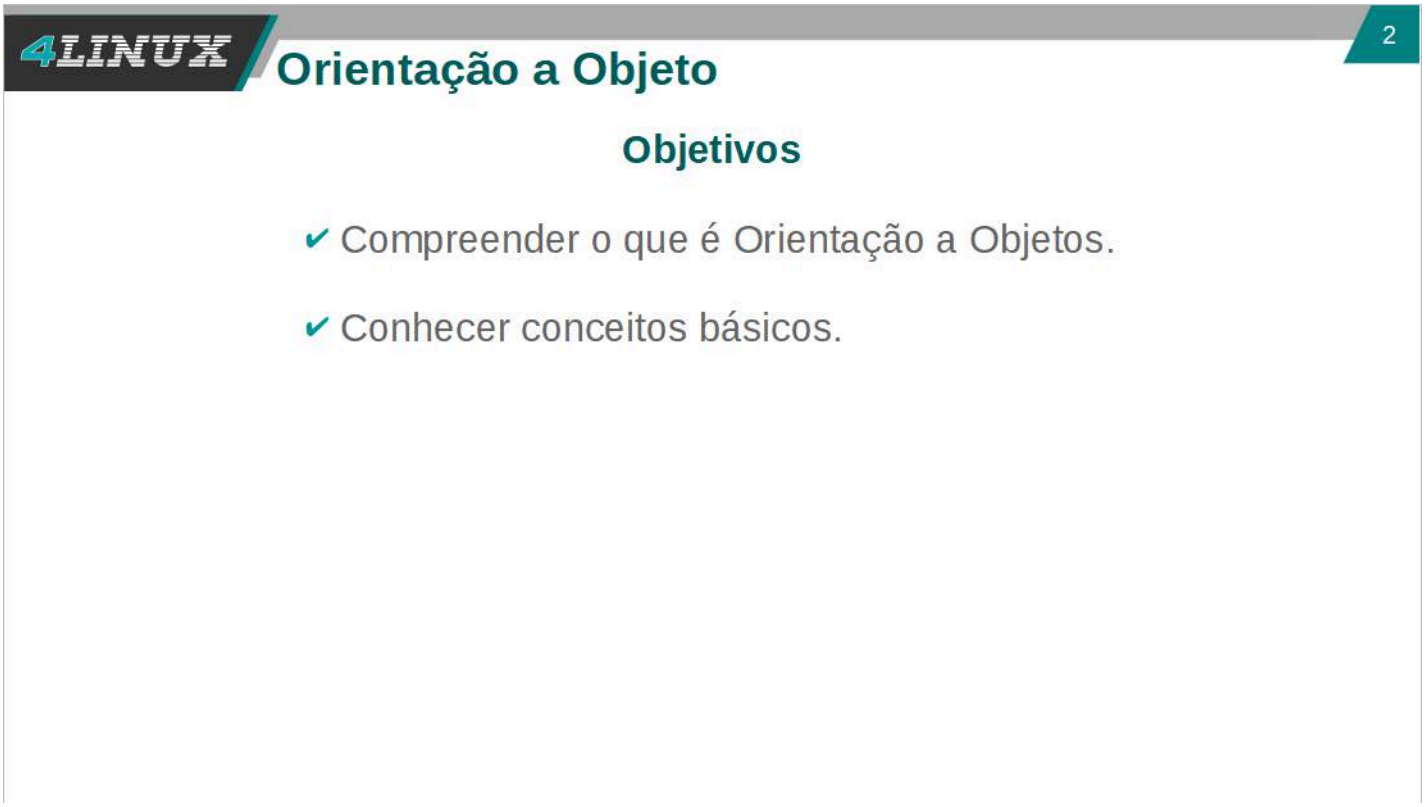
# Python Fundamentals



## Orientação a Objeto

**6.1**

Anotações




**Orientação a Objeto**

Nesta aula abordaremos o conceito de Orientação a Objeto, porque é importante implementar POO (Programação Orientada a Objeto) em nossos projetos.

**Anotações**

## Orientação a Objeto

✔ Orientação a objetos é um paradigma de programação, busca abstrair a programação para coisas do mundo real.

✔ Python é uma linguagem que aceita diversos paradigmas, no escopo de paradigmas, possui suporte maduro para desenvolvimento orientado a objetos.

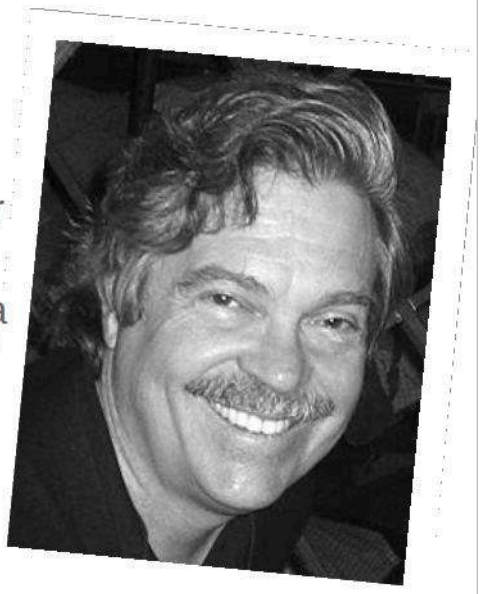
✔ Em orientação a objetos, um software não é composto por um grande bloco de funcionalidades específicas, mas sim, por vários blocos com funcionalidades distintas e independentes que juntas formam um sistema.

**Anotações**

# Orientação a Objeto

## Origem

- ✔ Um dos criadores desse conceito foi Alan Kay.
- ✔ Possuía conhecimento em biologia, acreditava que um programa de computador poderia trabalhar como o corpo humano:
- ✔ células independentes que juntas trabalham para alcançar um objetivo.
- ✔ uma das primeiras linguagens orientadas a objetos foi o SmallTalk, tornou possível a criação da interface gráfica para os computadores.



**Anotações**


**Características**

**Abstração**: utilizada para a definição de entidades do mundo real, onde são criadas as classes. Essas entidades são consideradas tudo que é real, levando em consideração suas características e ações.

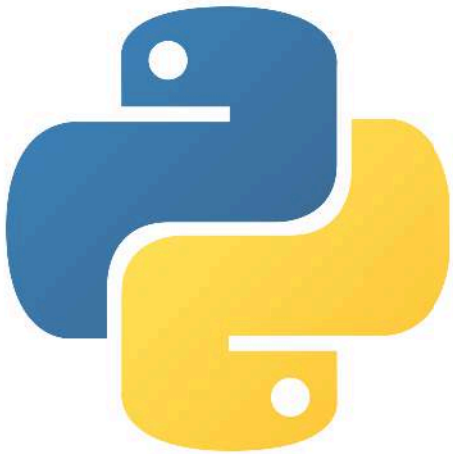
**Encapsulamento**: técnica utilizada para esconder uma ideia, não expondo detalhes internos para o usuário. Torna as partes do sistema o mais independentes possível. Por exemplo, quando um controle remoto estraga, somente o controle é trocado ou consertado e não a televisão inteira. Neste exemplo, demonstra a forma clássica de encapsulamento, quando o usuário muda de canal, ao efetuar tal ação, não sabe que programação acontece entre o televisor e o controle.

**Herança**: possui o mesmo significado que no mundo real. Assim como um filho pode herdar alguma característica do pai, na Orientação a Objetos é permitido que uma classe herde atributos e métodos da outra. Há apenas uma restrição para a herança, os modificadores de acessos das classes, métodos e atributos, devem ter visibilidade public e protected para serem herdados.

**Polimorfismo**: princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse, podem invocar métodos que têm a mesma identificação, assinatura, porém, comportamentos distintos, especializados para cada classe derivada, usando para tanto, uma referência a um objeto do tipo superclasse.

## Orientação a Objetos

A linguagem Python foi criada e conformidade ao conceito de orientação a objetos, possibilitando utilizar vários desses recursos nativamente.



A principal função da Orientação a Objetos consiste o reuso do código, traz ainda outros benefícios como facilitar a organização do código e a sua legibilidade.

**Anotações**

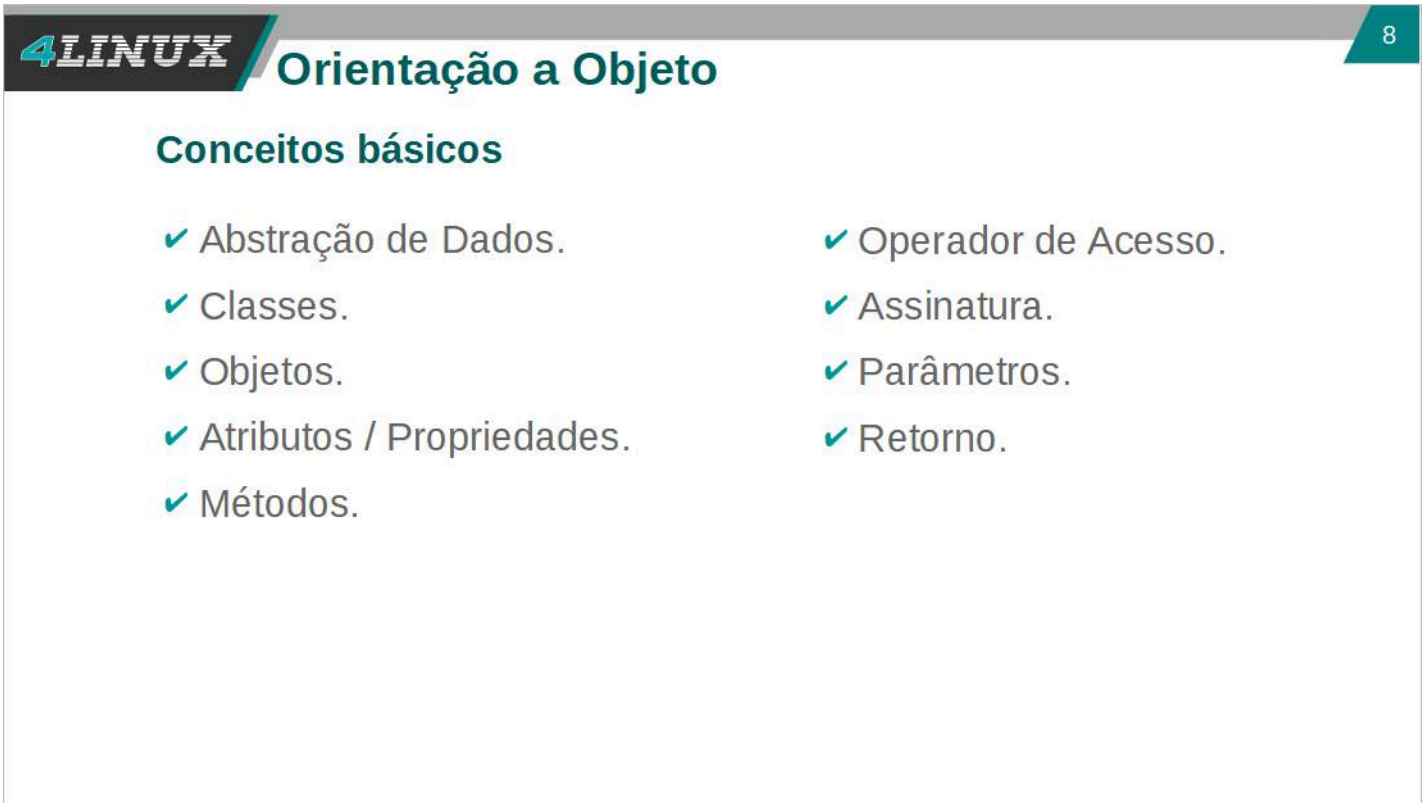
# Orientação a Objetos

## Vantagens

Entre as vantagens da orientação a objetos, destacam-se:

- ✔ Reutilização de código poupando tempo, tanto programando quanto no debug.
- ✔ Escalabilidade, adicionar código mais facilmente.
- ✔ Manutenibilidade, código mais fácil de manter.
- ✔ Agilidade de desenvolvimento.

Anotações




**Conceitos Básicos de Orientação a Objeto**

**Classes**: Constitui a abstração de alguma coisa do mundo real, em forma de código. Exemplo: quando pensamos no mundo DevOps, temos em mente um servidor. Este, assemelha-se a nossa classe Principal.

**Objetos**: consiste uma instância dessa classe. Exemplo: um Servidor tem suas características como: endereço IP, usuários, serviços, etc.

**Atributos/Propriedades**: são as características de um determinado objeto. São descritos na classe e, cada objeto possui seus próprios valores para essas propriedades. Exemplo: no servidor nós temos o endereço IP, serviços sendo executados, memória, disco, CPU etc.

**Anotações**

**Métodos**: métodos são ações ou comportamentos que cada objeto possui. Exemplo: na classe servidor, podemos realizar acesso, alterar endereço Ip, criar usuários, instalar serviços, apagar arquivos, etc.

**Operador de Acesso**: para acessar os métodos e atributos de um objeto usamos o ponto (.), empregado na maioria das linguagens de programação. Os atributos e métodos de uma classe são realizados da mesma maneira, a única diferença é que nos métodos temos () parênteses no final.

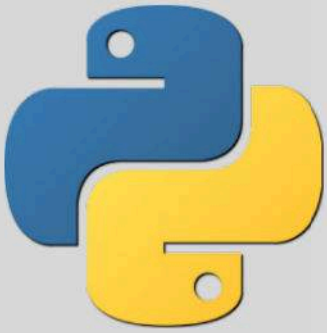
**Assinatura**: significa o retorno do método. Exemplo: uma função que faz a soma de dois valores deve
retornar um valor numérico, então a assinatura do método pode ser float, double ou integer.

**Parâmetros**: os métodos, assim como funções comuns, podem receber parâmetros. Os parâmetros podem não ser obrigatórios (endereco = None).

**Retorno**: quando queremos resgatar e trazer alguma informação. Usamos a palavra reservada *return* para resgatar o valor esperado.

**Anotações**

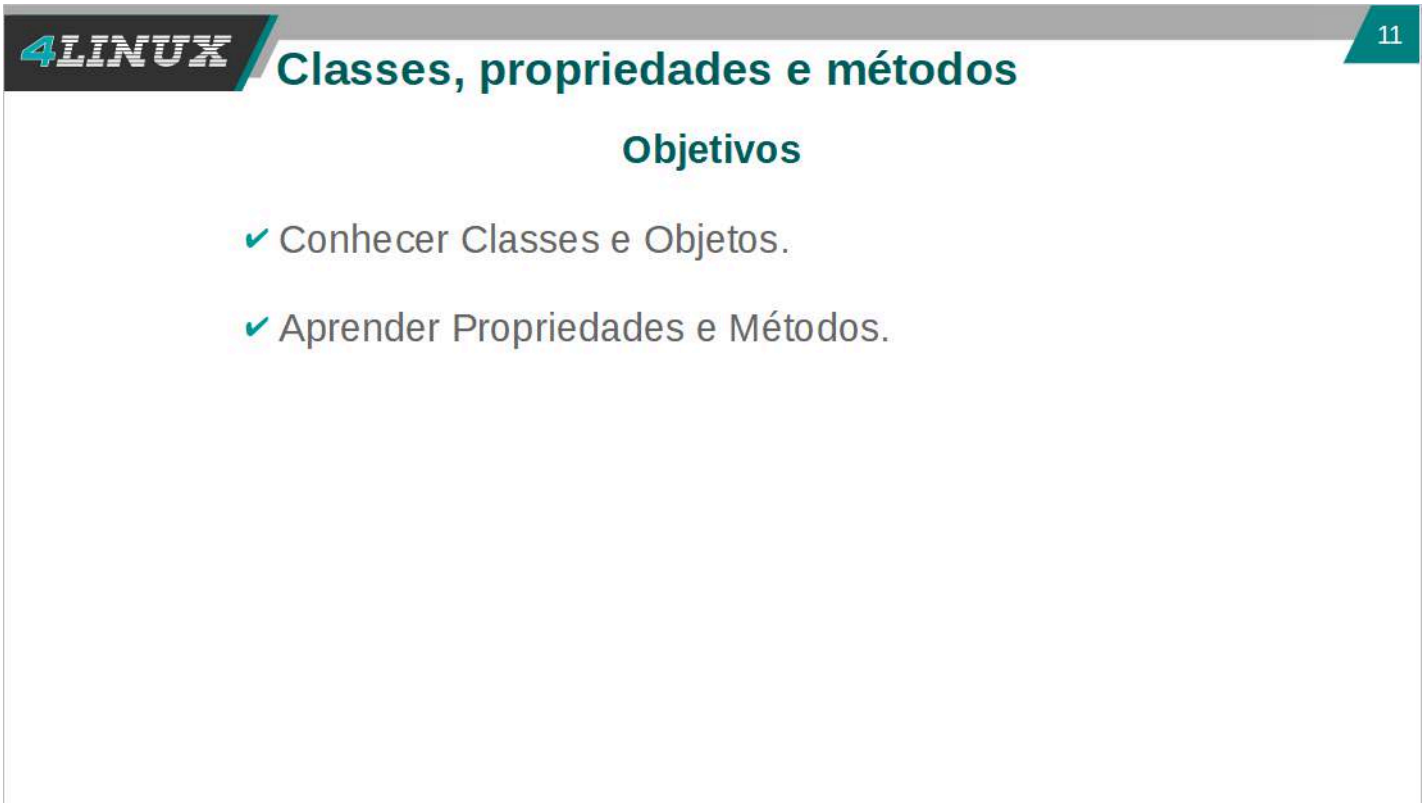
Python Fundamentals



# Classes Propriedades e Método

6.2

Anotações




**Classes, propriedades e métodos**

Nesta aula abordaremos o conceito de Classes, Objetos e trabalharemos com as definições de propriedades e métodos.

**Anotações**

# Classes, propriedades e métodos

## Entender Classes e Objetos

Antes de iniciar a prática com Orientação a Objetos, é preciso entender algo que confunde os iniciantes.

Classes definem características e o comportamento dos seus objetos. Cada característica é representada por um atributo e, cada comportamento é estabelecido por um método. Porém, essêncial saber que uma classe **NÃO É** um objeto!

### Entendendo o que são Classes e Objetos

Objetos são **entidades** que possuem um determinado **papel** num sistema. Na criação de um programa orientado à objetos, o primeiro passo, é determinar **quais** objetos existirão.

Mas como determino isso? Dependerá do sistema! Não existe uma receita, cada programador tem a sua visão e experiência. Dessa forma, julgam de modo diferente acerca dos objetos de um determinado sistema. Quanto mais complexo o sistema, mais difícil e diferente será essa análise. Observamos que em sistemas mais simples o juízo é mais uniforme.

Veja o exemplo de um programa para clínicas médicas:

*"O sistema deve cadastrar **pacientes**, **médicos**, e **consultas**. Será possível anexar uma ou mais **receita**s na consulta e, também possibilitará relacionar pacientes com **exame**s e seus **resultado**s."*

As palavras em negrito no texto, fazem referencia às entidades observadas no sistema. Note, que mesmo em casos bem simplificados há divergências: resultado do exame, é ou não um objeto? Dependerá do programador.

Se observar que o resultado do exame desempenha um papel importante no sistema e/ou que por si só, possui atributos vinculados e comportamento separado, podemos estruturá-lo como um objeto.

Considerando que seja meramente uma propriedade do objeto exame, então poderá ficar fora.

## Classes, propriedades e métodos

✔ Classes são como a planta baixa de uma casa, a especificação técnica de uma cadeira ou o projeto de um veículo.

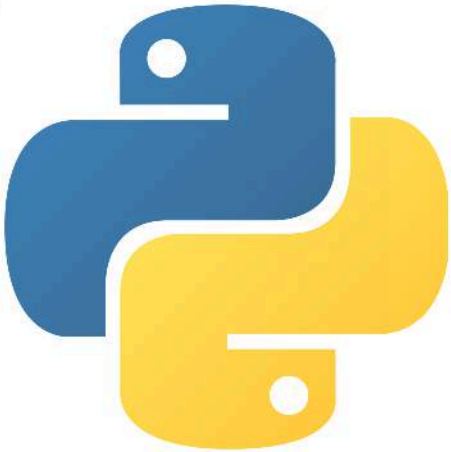
✔ A classe apenas estipula como será o objeto, mas não é o objeto em si.

Podemos construir 15 casas a partir de uma mesma planta baixa, Correto


**Anotações**

## Classes, propriedades e métodos

- ✔ Reconhecemos então, que todas essas “casas” são objetos produzidos a partir de uma classe.
- ✔ Dito de outra forma, os objetos = “casas”, possuem todas as características e comportamentos definidos na especificação da classe = “planta”.



**Anotações**

## Classes, propriedades e métodos

Portanto, uma classe NÃO é um objeto mas sim uma abstração de sua estrutura, na qual podemos definir as características que vão compor um objeto.

**Anotações**

# Classes, propriedades e métodos

## Estrutura de uma classe

Para criar classes em Python, existe palavra reservada “class”:

```
class NomedaClasse:
    def metodo(self):
        codigo
```

Anotações

# Classes, propriedades e métodos

## Método construtor

Permite executar algo quando instanciamos uma classe. Exemplo: setar atributos obrigatórios e defaults ou, exibir alguma mensagem na tela.

```
class NomedaClasse:
    def __init__(self):
        atributos
```

Anotações

# Classes, propriedades e métodos

## Atributos e métodos

Uma classe é composta de atributos e métodos, juntos criam a funcionalidade de um objeto.

- ✔ Atributos e métodos possuem o que chamamos de visibilidade.
- ✔ Métodos podem ou não, receber parâmetros que podem ou não ser obrigatórios.
- ✔ Métodos podem ou não retornar informação.

As características de um objeto, comumente chamadas de atributos ou propriedades, devem ser descritas na classe:

```
#!/usr/bin/python3

class Servidor:
     def __init_(self):
          self.cpu = None
          self.memoria = None
          self.disco = None
```

Definimos os atributos dentro do método construtor, referenciando o parâmetro self que alude a própria classe.

**Anotações**

# Classes, propriedades e métodos

## Criando classe com atributos

```
class NomedaClasse:

    atributo = "valor"

    def metodo(self):

        pass
```

Anotações

## Classes, propriedades e métodos

### Acessando Métodos e Atributos

- ✔ Em Python, utilizamos o ponto ( . ) para acessar atributos e métodos.
- ✔ Para acessar um método ou atributo, sempre usaremos uma instância da classe:

```
objeto = Classe()
```

A sintaxe para criação de atributos é a mesma para criação de variáveis.

```
#!/usr/bin/python3

class Servidor:
    memoria = None
    disco = None
    cpu = None


dns = Servidor()

dns.memoria = 2048
dns.disco = 50
dns.cpu = 2

print("O servidor tem as seguintes\
         configuracoes: CPU {},
         Memoria: {},
         Disco {} GB ".format(dns.cpu,dns.memoria,dns.disco))
```

Nas linhas destacadas, conferimos valores aos atributos, de modo que podemos resgatar seus valores posteriormente.

A saída deste código é: “o servidor tem as seguintes configurações:
CPU 2, Memoria: 2048, Disco 50 GB ".

Lembre-se, os valores atribuídos à essas propriedades, podem ser de qualquer tipo, semelhante ao praticado com variáveis normais. Podemos atribuir um array, uma string, um número, etc.

# Classes, propriedades e métodos

## Acessando Métodos e Atributos

```
class NomeClasse:
      def __init__(self):
          print("Acessando método construtor")
      def metodo(self):
          print("Acessando método")
classe = NomeClasse()
classe.metodo()
```

Anotações

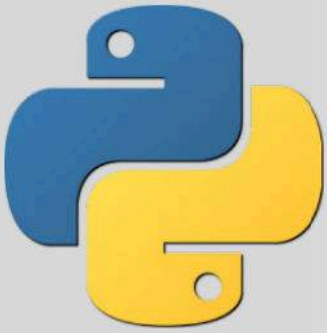
Além dos atributos de um objeto, devemos colocar nessa estrutura, as ações que aquele objeto executará, seus Métodos.
Tais ações, diferentemente dos atributos, precisam ser declaradas e implementadas. Precisamos enumerar todas as ações do nosso objeto e codificar o que cada uma realizará, há exceção que será explicada posteriormente, em Métodos Abstratos.

```
#!/usr/bin/python3

class Servidor:
    memoria = 1024
    disco = 50
    cpu = 1

    def contratarMemoria(self,memoria):
        self.memoria += memoria

    def contratarCpu(self,cpu):
        self.cpu += cpu

    def contratarDisco(self,disco):
        self.disco += disco

dns = Servidor()

dns.contratarMemoria(1024)
dns.contratarCpu(3)
dns.contratarDisco(50)

print ("O servidor tem as seguintes configuracoes:
       CPU {}, Memoria: {}, Disco {} GB ".format(
dns.cpu,dns.memoria,dns.disco))
```

Os métodos, são chamados como as funções, porém, necessitamos, do nome do objeto na frente seguido por ponto. O primeiro parâmetro **self** é obrigatório, somente com ele o python consegue diferenciar métodos de funções .

**Anotações**

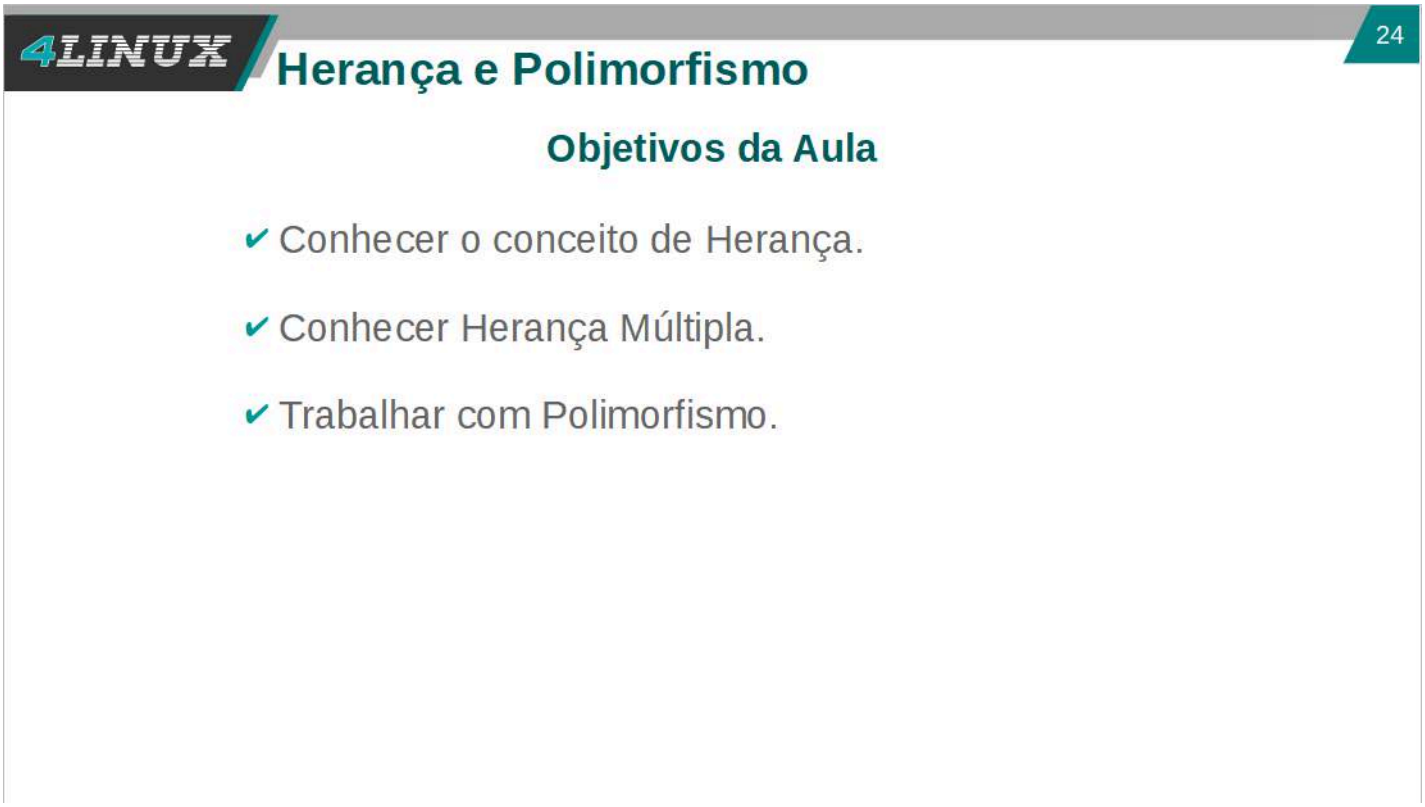
Python Fundamentals



# Herança e Polimorfismo

6.3

Anotações



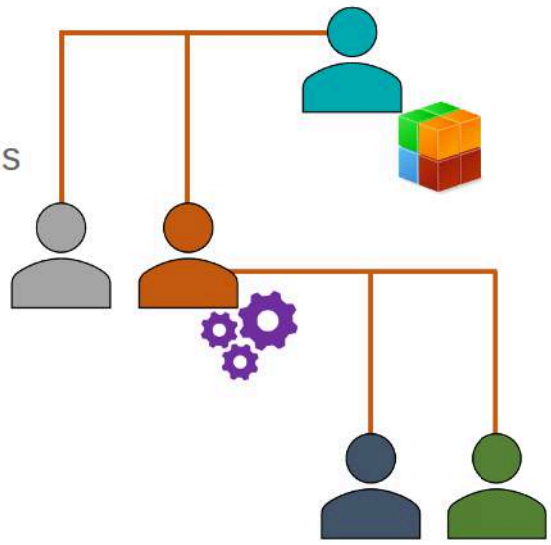

**Herança e Polimorfismo**

Nesta aula você aprenderá os conceitos de Herança e Polimorfismo, com exemplos e prática, criará uma aplicação.

**Anotações**

## Herança e Polimorfismo

- Herança de classes, é um dos conceitos fundamentais da orientação a objetos.
- Possibilita implementar uma série de outros conceitos, tornando mais claro o reúso de código.
- Permite que uma classe estenda outra.
- Assim, a classe filha, herdará todos os atributos e métodos da classe pai.
- Poderá então, possuir novos métodos e atributos, como reescrever métodos já existentes.



**Entendendo Herança**

Herança é um dos recursos mais interessantes da OO. Imagine que você está modelando seu sistema, observa que alguns objetos possuem comportamentos parecidos ou, até mesmo iguais. Algo a fazer para que a manutenção nesse código, parecido ,não seja muito complicada, seria centralizar tal código num único lugar. Facilitando assim, que a manutenção seja feita, em um único ponto.

Este é o conceito de generalização. Suponha a criação de um objeto Professor e um objeto Aluno. Se a modelagem tiver o básico de informações, notará que ambos possuem certas características e comportamentos iguais. Dito isto, podemos colocar tudo que for igual em uma nova classe, digamos Pessoa.

Deste modo, o funcionamento do sistema permanecerá o mesmo, em ambas as classes, Aluno e Professor, neste caso, será necessário colocar uma nova notação, indicando que ambas possuem como base a classe Pessoa.

**Anotações**

# Herança e Polimorfismo

Define-se a Herança colocando o nome da classe pai como parâmetro da classe filha.

```
class NomedaClasse(ClassePai):
```

- ✔ Todos os métodos e atributos estão na classe filha.
- ✔ O Python possui herança múltipla.
- ✔ É possível estender a partir de uma classe filha.

**Anotações**

## Herança e Polimorfismo

```
class ClassePai:

    def metodo(self):
        print("Acessando a Classe Pai")


class ClasseFilho(ClassePai):

    def __init__(self):
        print ("Acessando a Classe Filho")

classe = ClasseFilho()
classe.metodo()
```

**Entendendo Herança**

A classe filho, herda todos os métodos e atributos da classe pai. Porém a classe filho, pode se estender obtendo seus próprios métodos e atributos que diferem da classe pai.
Criamos um objeto da instância filho que herda da pai. O método construtor foi definido na classe filho e logo exibe uma mensagem, também acessamos o método herdado da classe pai.

**Anotações**

## Herança e Polimorfismo

**Polimorfismo**: significa muitas formas. Com esse recurso conseguimos ter métodos com nomes iguais, porém, executando coisas diferentes.

```
1   class ClassePai:
2       def metodo(self):
3           print("Metodo da classe pai")
4
5
6   class ClasseFilho(ClassePai):
7       def metodo(self):
8           print("Sobrescrevendo o metodo da classe pai.")
9
10
11  classe = ClasseFilho()
12  classe.metodo()
```

**Polimorfismo**

In [1]: definimos a instância da classe pai.
In [2]: estabelecemos o método da classe pai.
In [6]: determinamos a instância da classe filho, herdando da classe pai.
In [7]: sobrescrevemos o método herdado da classe pai realizando o polimorfismo, o qual as classes detém métodos com nomes iguais, fazendo coisas diferentes.
In [11]: criamos o objeto da instância filho.
In [12]: Verificamos o método no qual foi realizado o polimorfismo.

**Anotações**

# Herança e Polimorfismo

## Herança Múltipla

Utilizada quando uma classe precisa herdar os atributos e características de mais de uma classe.

```
1   class ClassePai:
2       a = 'caracteristica da classe pai'
3
4
5   class ClasseMae:
6       b = 'caracteristica da classe mae'
7
8
9   class ClasseFilho(ClassePai, ClasseMae):
10      c = 'caracterisca da classe filho'
11
12
13  classe = ClasseFilho()
14  print(classe.a, classe.b, classe.c)
```

**Herança Múltipla**

In [1]: definimos a instância da classe pai.
In [5]: determinamos a instância da classe mãe.
In [9]: estabelecemos a instância da classe filho, herdando da classe pai e mãe, realizando a herança múltipla.
In [13]: criamos o objeto da instância filho.
In [14]: verificamos a herança múltipla que herda de atributos de duas classes.

**Anotações**