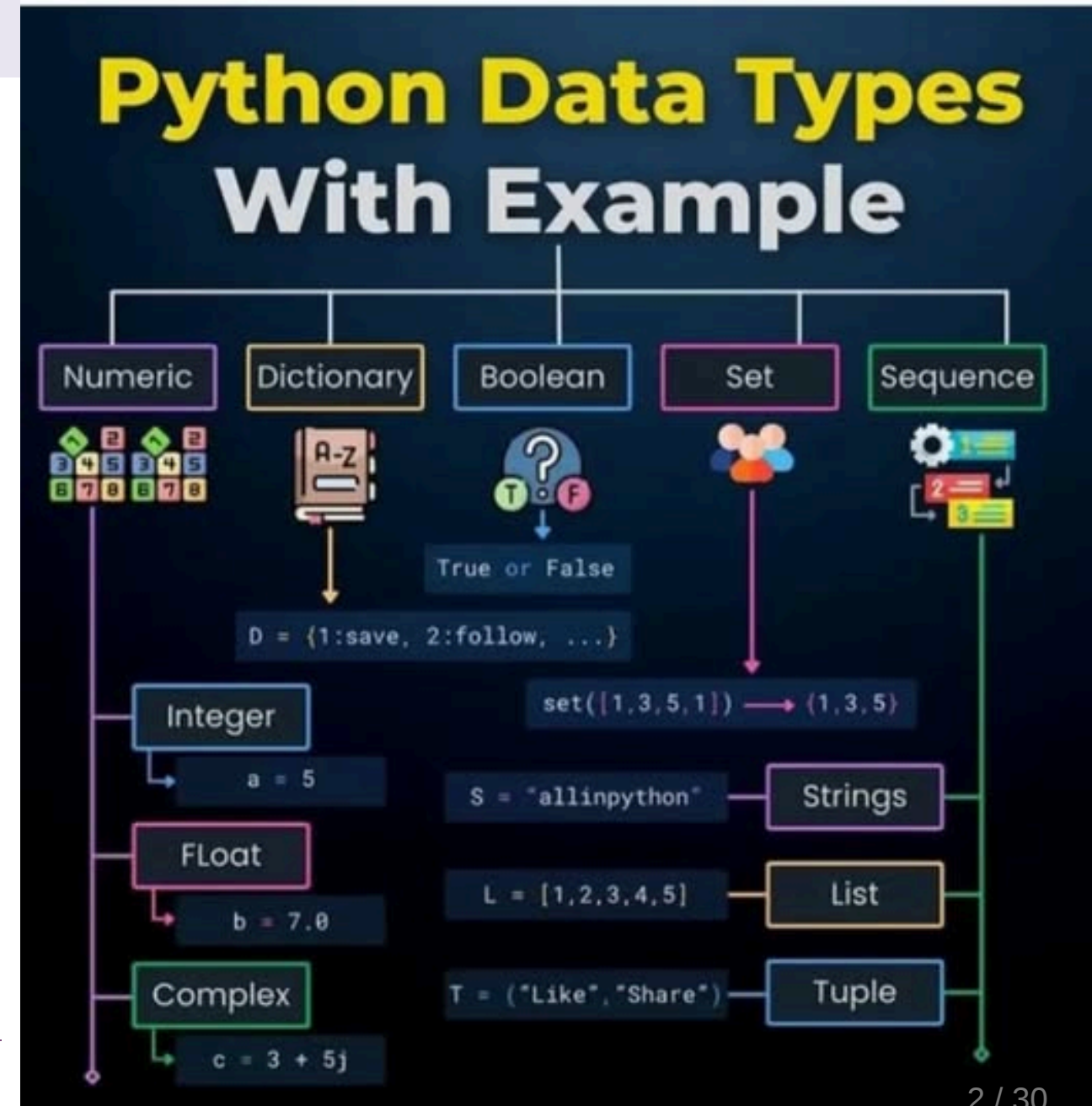


TÓPICO 03: CLASSES E OBJETOS

Estrutura de Dados - Professor Ramon Venson - SATC 2026.2

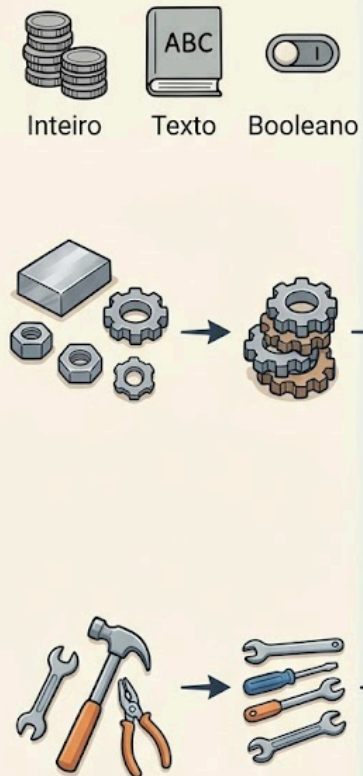
Tipos de Dados

Iniciamos o aprendizado em linguagem de programação com tipos de dados primitivos, como números, caracteres e booleanos.



SIMPLIFICANDO: PRIMITIVOS vs. CLASSES

PRIMITIVOS



CLASSE: ROBÔ



ATRIBUTOS
Nome: Toby (Texto)
Idade: 3 (Inteiro)
Carregado: True (Booleano)
Andar()
Latir()



OBJETO: ROBÔ-CÃO (Toby)



OBJETOS DERIVADOS

Tópico 03 - Classes e Objetos

Dados Estruturados

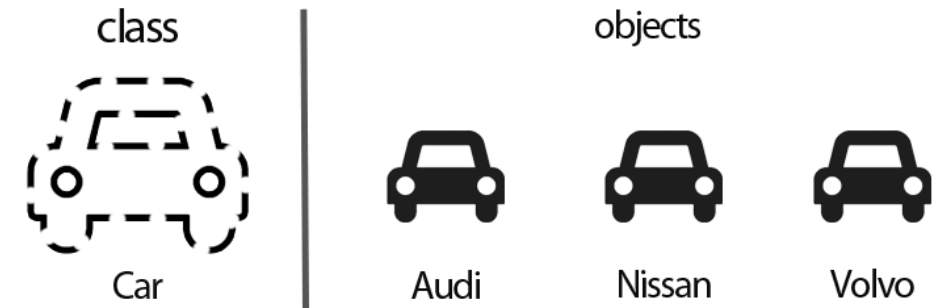
No entanto, para lidar com informações mais complexas, precisamos de estruturas que possam armazenar **múltiplos valores relacionados**.

```
nome = "Fulano"  
sobrenome = "da Silva"  
idade = 30
```

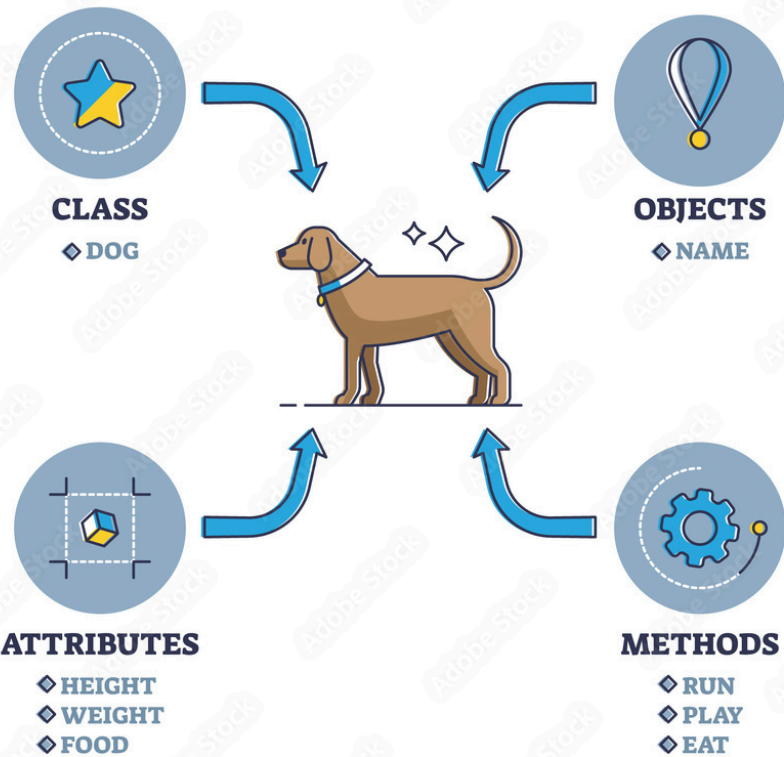
```
pessoa = Pessoa(nome, sobrenome, idade)  
pessoa.apresentar()  
# Olá, meu nome é Fulano da Silva e tenho 30 anos.
```

Orientação à Objeto

Orientação à Objeto (*OOP, Oriented Object Programming*) é um conceito de programação que tem como princípio abstrair o mundo real em objetos com dados e comportamentos.



OBJECT ORIENTED PROGRAMMING



Na Orientação à Objetos, todos os componentes de um sistema possuem **atributos** que o descrevem e **métodos** que descrevem funcionalidades.

Em uma analogia com o mundo real, podemos utilizar qualquer tipo de entidade (um cachorro por exemplo) e descrever seus atributos e funcionalidades

Um cachorro pode ser descrito (*atributos*) como:

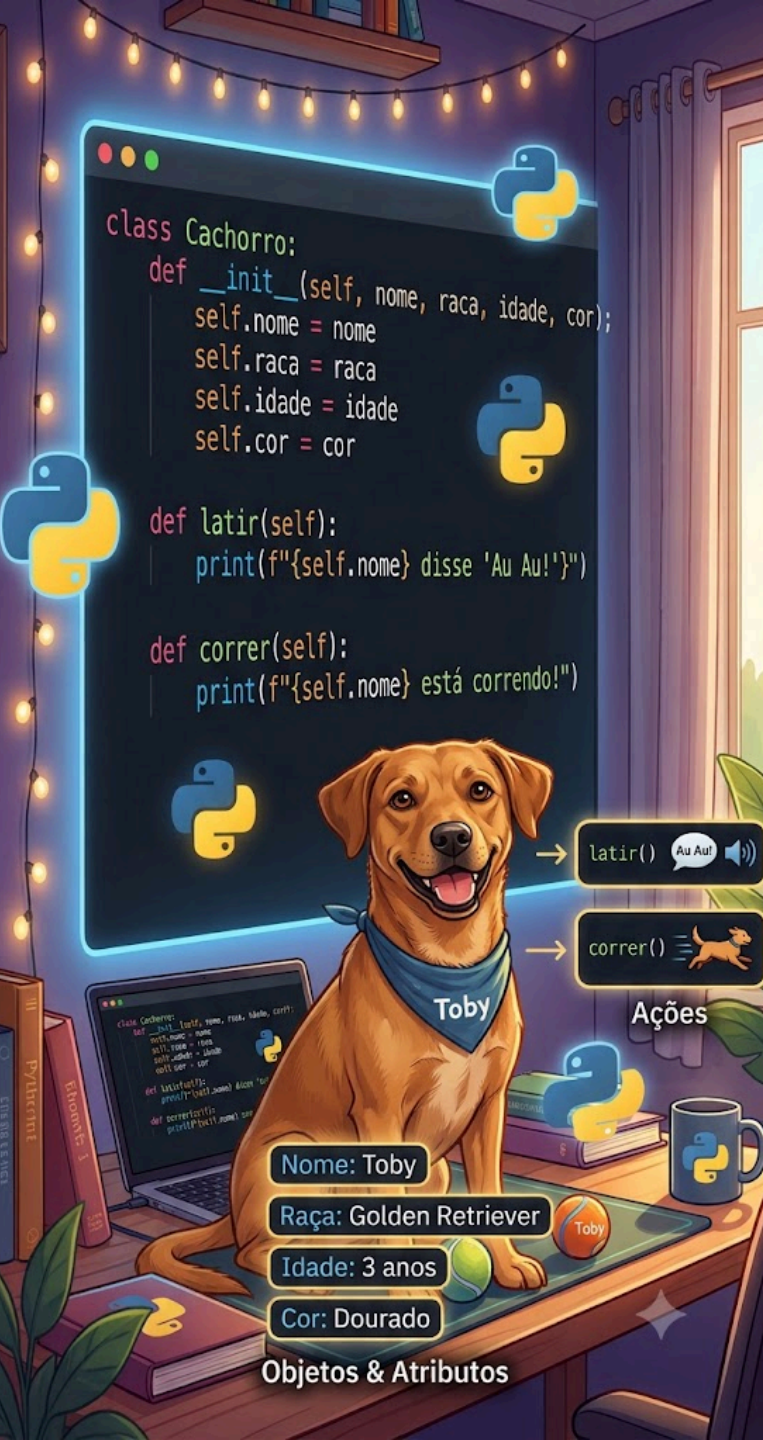
- Cor
- Raça
- Tamanho
- Peso





E suas funcionalidades (*métodos*):

- Correr
- Dormir
- Comer
- Brincar



Classes

O Python é uma linguagem de programação com suporte à orientação à objetos, o que significa que podemos criar **classes** e **objetos**.

A **classe** representa um modelo de um **objeto**.

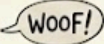


class Cachorro:

doguinho = Cachorro("Toby", "Caramelo")

outro_doguinho = Cachorro("Rex", "Preto")



instância 'doguinho' (Toby, Caramelo)
doguinho.latir() 
 # HOUH! Toby está latindo!



instância 'outro_doguinho' (Rex, Preto)
outro_doguinho.latir() 
 # HOUH! Rex está latindo!

Usando a mesma analogia do cachorro do mundo real, uma **classe** é a estrutura que usamos pra descrever um cachorro.

Os diferentes cachorros existentes são os **objetos** criados a partir dessa estrutura.

Criando uma nova classe

Para criar uma classe utilizamos a seguinte sintaxe:

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

    def latir(self):
        print("HOUH!")
```

Repare que dentro da declaração da classe estamos declarando diferentes variáveis (aqui chamadas de **atributos**) e funções (aqui chamadas de **métodos**).

Criando um novo objeto a partir de uma classe

Com a classe criada, vamos criar um novo objeto a partir dela:

```
doguinho = Cachorro("Toby", "Caramelo")  
  
print(doguinho.nome) # Toby  
print(doguinho.cor)  # Caramelo  
doguinho.latir()     # HOUH!
```

A variável `doguinho` nesse caso será do tipo `Cachorro` e conterá todos os atributos e métodos descritos na classe.

Criando múltiplos objetos

Podemos criar diferentes objetos a partir da mesma classe, e cada um terá seu próprio nome e cor:

```
doguinho = Cachorro("Toby", "Caramelo")
outro_doguinho = Cachorro("Rex", "Preto")

doguinho.latir()          # HOUH!
outro_doguinho.latir()   # HOUH!

print(doguinho.nome)      # Toby
print(outro_doguinho.nome) # Rex
```

Repare que o método `latir` foi chamado em dois objetos diferentes, e cada um deles imprimiu o nome que foi atribuído a ele no momento da criação.

Vantagens da Orientação à Objetos

- Permite modelar o mundo real de forma mais intuitiva.
- Facilita a reutilização de código através da criação de classes e objetos.
- Promove a organização do código, tornando-o mais legível e fácil de manter.

Construtor

Um **construtor** é um método especial de uma classe que é usado para inicializar os atributos de um objeto quando ele é criado.

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

O construtor é definido pelo método `__init__`, que é chamado automaticamente quando um novo objeto é criado a partir da classe.



self

`self` é um atributo especial de uma classe, que representa um objeto dentro dele mesmo.

Na linguagem Python, o `self` é utilizado como parâmetro em métodos de uma classe para referenciar o próprio objeto que está chamando o método.

Quando usamos a palavra `self` dentro de um método, estamos referenciando o próprio objeto que estiver sendo manipulado no momento.

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

    def latir(self):
        print("HOUH!")
        print(self.nome + " está latindo!")
```

No exemplo acima, o método `latir` utiliza o `self` para acessar o atributo `nome` do objeto que estiver chamando o método. O nome vai variar de acordo com o objeto que estiver chamando o método.

Nesse caso, quando chamamos o método `latir` em diferentes objetos, o valor do atributo `nome` será diferente para cada um deles:

```
doguinho = Cachorro("Toby", "Caramelo")
outro_doguinho = Cachorro("Rex", "Preto")

doguinho.latir()          # HOUH! Toby está latindo!
outro_doguinho.latir()    # HOUH! Rex está latindo!
```

Repare que o método `latir` foi chamado em dois objetos diferentes, e cada um deles imprimiu o nome que foi atribuído a ele no momento da criação.

CONCEITO DE ENCAPSULAMENTO

ANTES
- Sem Encapsulamento -



DADOS EXPOSTOS

DEPOIS
- Com Encapsulamento -



DADOS PROTEGIDOS E
ACESSO VIA INTERFACE

Encapsulamento

O encapsulamento é um conceito da orientação à objetos que consiste em **proteger os dados de um objeto** de serem acessados diretamente por outros objetos.

Problema dos atributos públicos

Quando os atributos de uma classe são públicos, qualquer objeto pode acessá-los e modificá-los diretamente, o que pode levar a inconsistências e erros no programa.

```
doguinho = Cachorro("Toby", "Caramelo")  
print(doguinho.nome) # Toby  
doguinho.nome = "Rex"  
print(doguinho.nome) # Rex
```

O que parece simples e prático, pode gerar problemas no futuro, pois qualquer objeto pode alterar os atributos de outro objeto sem restrições.

Getters

Para isolar o acesso aos atributos de uma classe, vamos utilizar o padrão **getters**, que são métodos que permitem acessar os atributos de um objeto de forma controlada.

A primeira etapa é tornar os atributos privados, ou seja, não acessíveis diretamente por outros objetos, usando o prefixo `__` antes do nome do atributo.

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

O uso do prefixo `__` antes do nome do atributo indica que ele é privado e não deve ser acessado diretamente por outros objetos.

```
doguinho = Cachorro("Toby", "Caramelo")  
print(doguinho.__nome) # Erro
```

A segunda etapa é criar os métodos getters, que permitem acessar os valores dos atributos privados de forma controlada.

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

    def get_nome(self):
        return self.__nome

    def get_cor(self):
        return self.__cor
```

Os métodos começam com o prefixo `get_` seguido do nome do atributo. Sua função é simplesmente retornar o valor do atributo privado correspondente.

Dessa forma, sempre que quisermos acessar o valor de um atributo privado, devemos utilizar o método getter correspondente:

```
doguinho = Cachorro("Toby", "Caramelo")  
print(doguinho.get_nome()) # Toby  
print(doguinho.get_cor())  # Caramelo
```

Setters

Sem possibilidade de alterar os atributos diretamente, os setters são métodos que permitem modificar os valores dos atributos de forma controlada.

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

    def set_nome(self, nome):
        self.__nome = nome

    def set_cor(self, cor):
        self.__cor = cor
```

Vamos utilizar os métodos `set_nome` e `set_cor` para alterar os valores dos atributos de forma controlada:

```
doguinho = Cachorro("Toby", "Caramelo")
print(doguinho.get_nome()) # Toby
doguinho.set_nome("Rex")
print(doguinho.get_nome()) # Rex
```

Roteiro para Encapsulamento

- **Atributos privados** : vamos iniciar colocando o prefixo `__` antes do nome do atributo para torná-lo privado.
- **Getters** : vamos criar métodos para acessar os valores dos atributos privados.
- **Setters** : vamos criar métodos para modificar os valores dos atributos privados.

Bônus: @property

Um *decorator* é uma função que modifica o comportamento de outra função ou método. No Python, podemos utilizar o *decorator* `@property` para criar métodos getters e setters de forma mais elegante.

```
class Cachorro:
    def __init__(self, nome):
        self.__nome = nome

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

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

Para utilizar os métodos getters e setters com o *decorator* `@property` , podemos acessar os atributos como se fossem públicos, mas na verdade estamos utilizando os métodos getters e setters por trás dos panos.

```
doguinho = Cachorro("Toby")  
print(doguinho.nome) # Toby  
doguinho.nome = "Rex"  
print(doguinho.nome) # Rex
```

Resumo

- **Orientação à Objetos:** Paradigma de programação que permite modelar o mundo real em objetos com atributos e métodos.
- **Classe:** Estrutura que define os atributos e métodos de um objeto.
- **Objeto:** Instância de uma classe, que possui seus próprios valores.
- **Construtor:** Método que inicializa os atributos de um objeto quando ele é criado.
- **self**: Referência ao próprio objeto dentro de um método.

- **Encapsulamento:** Técnica de proteger os dados de uma classe, tornando-os privados e acessíveis apenas por métodos específicos.
- **Getters:** Métodos que permitem acessar os valores dos atributos privados de forma controlada.
- **Setters:** Métodos que permitem modificar os valores dos atributos privados de forma controlada.
- **@property** : Decorator que permite criar métodos getters e setters de forma mais elegante.