# FIAP



FIAP

ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

DISRUPTIVE ARCHITECTURES: IOT, IOB & IA

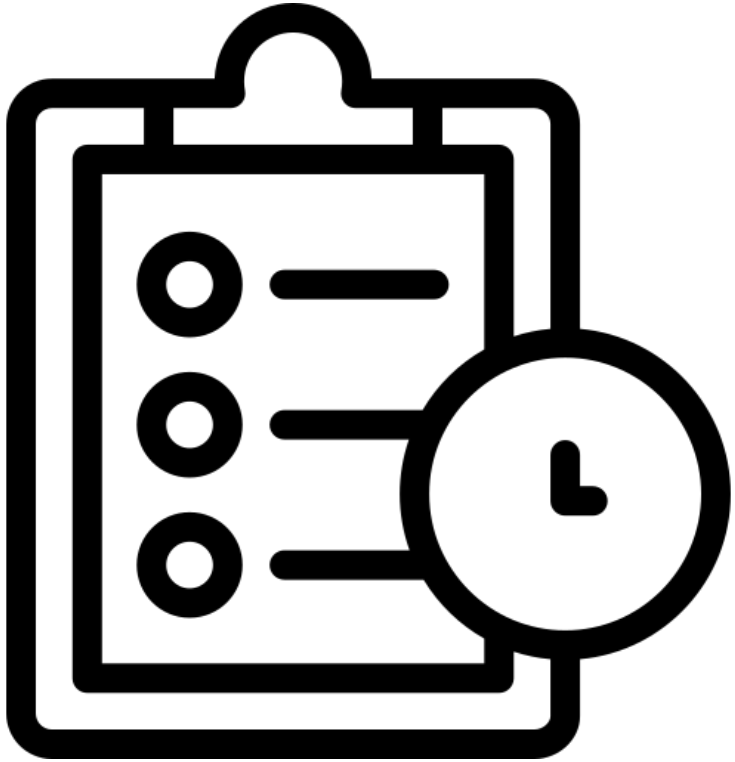
# 02 – Orientação a Objetos com Python


Prof. Airton Y. C. Toyofuku

profairton.toyofuku@fiap.com.br

# Agenda

- Introdução;
- Criando Classes e Objetos
- Construtores e Destrutores;
- Encapsulamento;
- Propriedades;
- Herança e Polimorfismo;
- Métodos Especiais;
- Aplicações em IA;
- Exercicios;



# Introdução

FIAP

## ❖ O que é orientação a objetos?

- ❑ Paradigma de programação que visa modelar o mundo real com objetos que possuam caracteristicas e comportamentos;

## ❖ Por que usar orientação a objetos em Python?

- ❑ Organização do código para manter clareza, promover reutilização e manutenibilidade;
- ❑ Utilizar bibliotecas da comunidade;

# Introdução

FIAP

## ❖ Mas como assim??????

- ❑ Vamos voltar um pouco… no Paradigma Procedural (ou como a gente fazia antes…)

```
[2]: pokemonNome = "Pikachu"
     pokemonAtaque = "Choque do Trovão"

     print (pokemonNome)
     print (pokemonAtaque)
```

```
Pikachu
Choque do Trovão
```

# Introdução

FIAP

- Outro jeito de fazer...

```
[1]: pokemon = { 'nome':"", 'ataque':""}

     pokemon['nome'] = "Pikachu"
     pokemon['ataque'] = "Choque do Trovão"

     print(pokemon['nome'])
     print(pokemon['ataque'])
```

```
Pikachu
Choque do Trovão
```

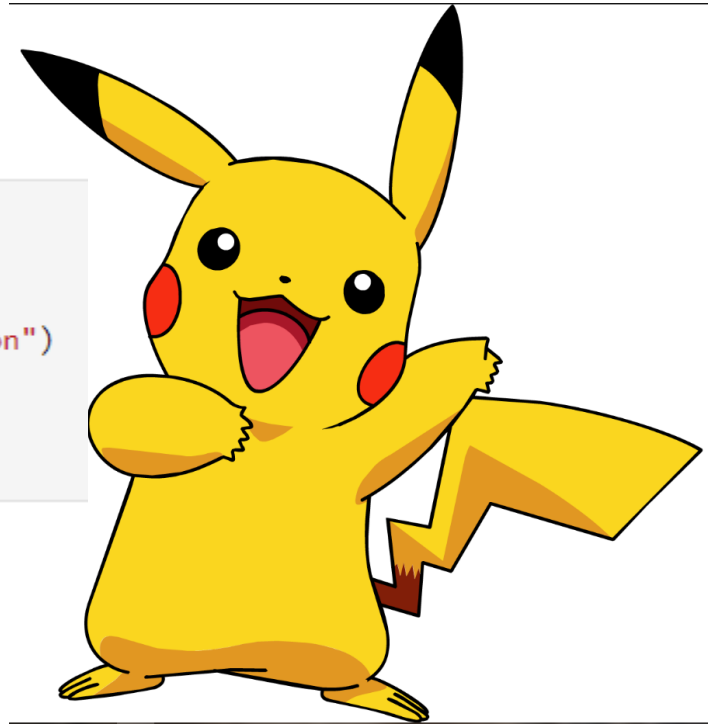
# Introdução

FIAP

- Mais um jeito de fazer...

```
pokemon = { 'nome':"", 'ataque':""}

for chave in pokemon.keys():
    pokemon[chave] = input("Digite o " + str(chave) + " do pokemon")

print(pokemon['nome'])
print(pokemon['ataque'])
```

```
Digite o nome do pokemon Pikachu
Digite o ataque do pokemon Choque do Trovão
 Pikachu
 Choque do Trovão
```



# Introdução

FIAP

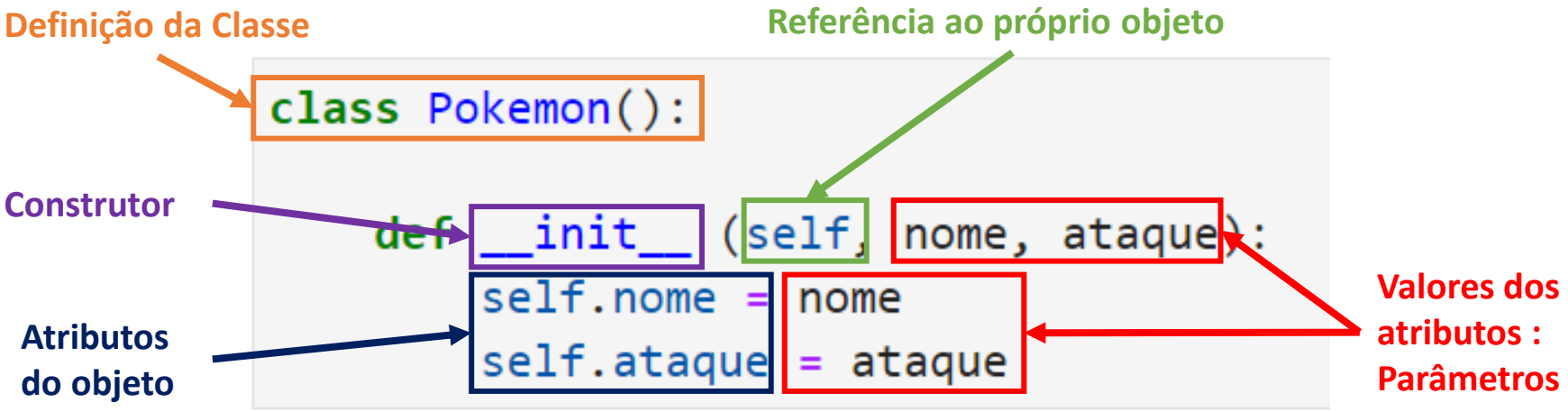
- Agora com funções!!!

```
pokemon = { 'nome':"", 'ataque':""}

def cadastra_pokemon(nome, ataque):
    pokemon['nome'] = nome
    pokemon['ataque'] = ataque

def imprime_pokemon():
    for chave, valor in pokemon.items():
        print(chave , valor)

cadastra_pokemon("Pikachu", "Choque do Trovão")
imprime_pokemon()
```

```
nome Pikachu
ataque Choque do Trovão
```

# Introdução

❖ Agora vamos fazer isso com CLASSE !!!!



# Introdução

FIAP

- Agora vamos fazer isso com CLASSE !!!!

Parâmetros para criação do objeto

Objeto

Classe

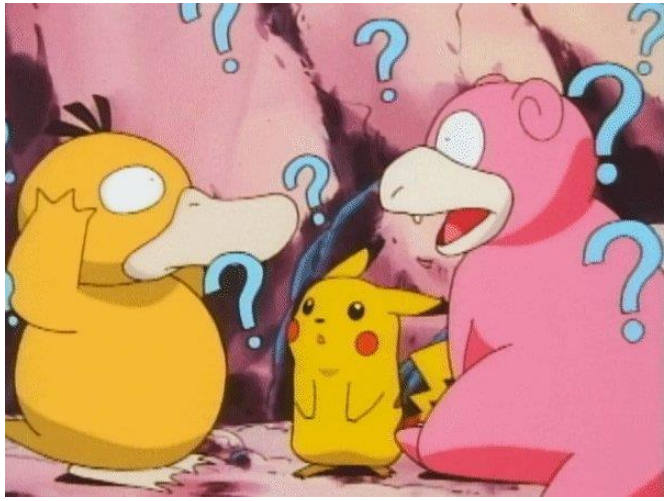
```
meuPokemon = Pokemon("Pikachu", "Choque do Trovão")
print(meuPokemon.nome + " " + meuPokemon.ataque + "!!!!!")
```

```
Pikachu Choque do Trovão!!!!!
```

# Criando Classes e Objetos

FIAP

- O que é uma CLASSE?
  - Estrutura de dados que define um modelo para criar objetos com atributos e métodos em comum;
- O que são ATRIBUTOS?
  - São características ou propriedades de um objeto;
- O que são MÉTODOS?
  - São as ações que um objeto pode realizar;
- Mas afinal, o que é um OBJETO?
  - É uma instância de uma classe!
  - É o que torna uma representação genérica em algo **EXPECÍFICO**!



# Criando Classes e Objetos

FIAP

```python
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def saudacao(self):
        print(f"Olá, meu nome é {self.nome} e tenho {self.idade} anos")
```

Classe

```python
pessoa1 = Pessoa("João", 30)
pessoa2 = Pessoa("Maria", 25)

pessoa1.saudacao()
pessoa2.saudacao()
```

```
Olá, meu nome é João e tenho 30 anos
Olá, meu nome é Maria e tenho 25 anos
```

Especificação do Objeto

# Construtores e Destrutores

## ❖ O Construtor “__init__”

- ✓ Método especial para criar um objeto a partir de uma classe;
- ✓ Usado também para inicializar atributos;
- ✓ É chamado automáticamente quando um objeto é criado;

## ❖ O Destrutor “__del__”

- ✓ Método especial para destruir um objeto;
- ✓ Quando chamado, apaga o objeto e libera a memória ocupada por ele;
- ✓ É chamado automaticamente quando o objeto é apagado;

# Construtores e Destrutores

FIAP

```python
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
        print(f"Objeto {self.nome} criado.")

    def __del__(self):
        print(f"Objeto {self.nome} apagado.")
```

Exemplo de uma Classe genérica de “Pessoa”, com seu construtor e destrutor.

```python
pessoa1 = Pessoa("João", 30)
pessoa2 = Pessoa("Maria", 25)
del pessoa1
del pessoa2
```

Exemplo dos usos do construtor e destrutor da Classe Pessoa.

```
Objeto João criado.
Objeto Maria criado.
Objeto João apagado.
Objeto Maria apagado.
```

Resultado.

# Construtores e Destrutores

```
pessoa1 = Pessoa("João", 30)
```

```
Objeto João criado.
```

```
pessoa1
```

```
<__main__.Pessoa at 0x196f068>
```

```
del pessoa1
```

```
pessoa1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[20], line 1
----> 1 pessoa1

NameError: name 'pessoa1' is not defined
```

# Encapsulamento

FIAP

- O que é ENCAPSULAMENTO?
  - Permite esconder a implementação detalhada de uma classe;
  - Fornece uma interface pública para trabalhar com a classe;
  - Ajuda a garantir integridade e manter a classe segura de modificações;
- Como usar Encapsulamento em Python?
  - Convenção de nomenclatura. Se o método ou atributo começarem com um sublinhado (__), ele é considerador **PRIVADO!**

```python
class Pokemon:
    def __init__(self, nome, tipo):
        self.__nome = nome
        self.__tipo = tipo

    def get_nome(self):
        return self.__nome

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

    def get_tipo(self):
        return self.__tipo

    def set_tipo(self, tipo):
        self.__tipo = tipo
```

# Encapsulamento

Os atributos “__nome” e “_tipo” são considerados **PRIVADOS**!

O acesso é realizado através dos métodos “get_nome” e “get_tipo”;

É possível modificar os parametros através dos métodos “set_nome” e “set_tipo”;

```
class Pokemon:
    def __init__(self, nome, tipo):
        self.__nome = nome
        self.__tipo = tipo

    def get_nome(self):
        return self.__nome

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

    def get_tipo(self):
        return self.__tipo

    def set_tipo(self, tipo):
        self.__tipo = tipo
```

# Encapsulamento

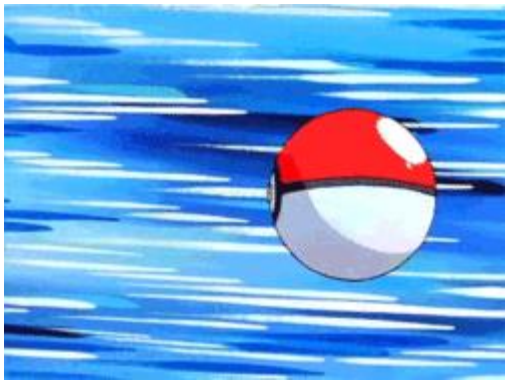
FIAP

```
meuPokemon = Pokemon("Squirtle", "Agua")

print(meuPokemon.get_nome())
print(meuPokemon.get_tipo())
```

```
Squirtle
Agua
```

```
meuPokemon.set_nome("Squirtle da Silva")
meuPokemon.set_tipo("Aquoso")
print(meuPokemon.get_nome())
print(meuPokemon.get_tipo())
```

```
Squirtle da Silva
Aquoso
```



São os famosos GETTERS e SETTERS!!!!!

# Encapsulamento

FIAP

## E os modificadores de acesso?

- NÃO EXISTE o modificador de acesso **public**!
- NÃO EXISTE o modificador de acesso **private**!
- NÃO EXISTE o modificador de acesso **protected**!

## É uma convenção!

- Se o atributo ou método começarem com sublinhado (__), ele é considerado privado e **NÃO DEVE SER ACESSADO DIRETAMENTE!!!!**

# Propriedades

FIAP

- O que são e para que servem?
  - As propriedades são um recurso em Python que permite acessar os atributos de uma classe de maneira mais controlada e segura;
- Mas já não temos os Getters e Setters do Encapsulamento?
  - Criar métodos com nomes de Get e Set em Python não é muito elegante, e também polui o código...
- Então, como resolver isso?
  - Utilizando a propriedade **@property** para um método de Get e **@atributo.setter** para o Set!

```python
class Pokemon:
    def __init__(self, nome, tipo):
        self.__nome = nome
        self.__tipo = tipo

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

    @property
    def tipo(self):
        return self.__tipo

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

    @tipo.setter
    def tipo(self, tipo):
        self.__tipo = tipo
```

# Propriedades

```
meuPokemon = Pokemon("Pikachu", "Elétrico")
print(meuPokemon.nome)
print(meuPokemon.tipo)
```

```
Pikachu
Elétrico
```

```
meuPokemon.nome = "Pikachu"
meuPokemon.tipo = "Normal e Elétrico"
print(meuPokemon.nome)
print(meuPokemon.tipo)
```

```
Pikachu
Normal e Elétrico
```

```
class Pokemon:
    def __init__(self, nome, tipo):
        self.__nome = nome
        self.__tipo = tipo

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

    @property
    def tipo(self):
        return self.__tipo

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

    @tipo.setter
    def tipo(self, tipo):
        self.__tipo = tipo
```

# Herança e Polimorfismo

## O que é Herança?

- Característica mais importante da orientação a objetos;
- Permite a criação de novas classes, baseadas em outras classes!
- Herda todos os atribuitos e métodos da classe base (classe mãe);

## Superclasse:

- Classe Base ou Classe Mãe;
- Contém os métodos e atributos basicos;

## Subclasse:

- Classe Derivada ou Classe Filhas;
- Herda todas as caracteristicas da Classe Mãe;
- Pode adicionar novos atributos e métodos;
- Pode sobrescrever atributos e métodos herdades;

# Herança e Polimorfismo

FIAP

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

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

    @property
    def cor(self):
        return self.__cor

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

    @cor.setter
    def nome(self, cor):
        self.__cor = cor

    def miar(self):
        print("Miau!!")
```

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

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

    @property
    def cor(self):
        return self.__cor

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

    @cor.setter
    def nome(self, cor):
        self.__cor = cor

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

# Herança e Polimorfismo

FIAP

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

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

    @property
    def cor(self):
        return self.__cor

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

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

```python
class Gato (Animal):
    def miar(self):
        print("Miau!!")
```

```python
class Cachorro (Animal):
    def latir(self):
        print("Au Au!!")
```

- ✓ Gato e Cachorro são Animais!
- ✓ Animal é a Superclasse;
- ✓ Gato e Cachorro são as Subclasses

- ❖ **Gato e Cachorro HERDAM as características de Animal!**

# Herança e Polimorfismo

```python
class Animal:
    def __init__(self, nome, tipo):
        self.nome = nome
        self.tipo = tipo

    def falar(self):
        print(f"{self.nome} fala.")
```

Exemplo de uma Classe genérica de “Animal”, com seus atributos e métodos. É uma Superclasse!

```python
class Cachorro(Animal):
    def falar(self):
        print(f"{self.nome} late.")

class Gato(Animal):
    def falar(self):
        print(f"{self.nome} miado.")
```

Exemplos de classes herdando a classe “Animal”. São Subclasses!

```python
cachorro = Cachorro("Buddy", "Cachorro")
gato = Gato("Mimi", "Gato")
cachorro.falar()
gato.falar()
```

Criando objetos de acordo com as subclasses e invocando seus métodos.

```
Buddy late.
Mimi miado.
```

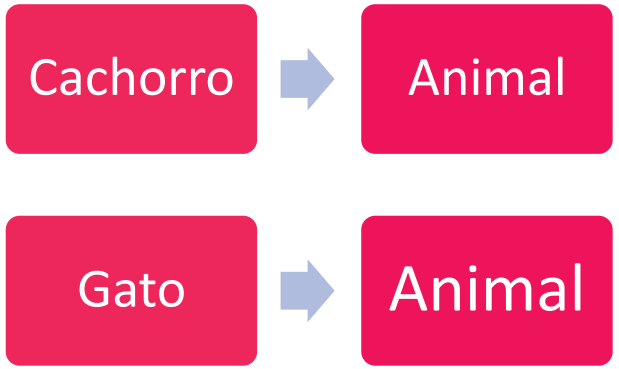
Resultado!

# Herança e Polimorfismo

FIAP

## O que é POLIMORFISMO?

- É a capacidade de um objeto de várias classes diferentes ser tratado como um objeto de uma classe base comum;




## Mas isso não é Herança?

- Sim e Não…
- Herança é quando a subclasse herda todas as características da Superclasse…
- Polimorfimos é quando a Subclasse redefine alguma característica da Superclasse!

# Herança e Polimorfismo

FIAP

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

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

    @property
    def cor(self):
        return self.__cor

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

    @cor.setter
    def nome(self, cor):
        self.__cor = cor

    def falar(self):
        print("Eu sou um animal!")
```

```python
class Gato (Animal):
    def falar(self):
        print("Miau!!")

class Cachorro (Animal):
    def falar(self):
        print("Au Au!!")
```

```python
animal = Animal("Unicornio", "Todas!")
gatinho = Gato("Mingau", "Branco")
doguinho = Cachorro("Bidu", "Azul")
```

```python
animal.falar()
gatinho.falar()
doguinho.falar()
```

```
Eu sou um animal!
Miau!!
Au Au!!
```

# Herança e Polimorfismo

## ❖ Sobrecarga de Operadores

- ❖ Permite a definição de comportamentos personalizdos para os operadores de uma classe;
- ❖ Altera o comportamento de operações básicas, como adição, subtração, multiplicação e divisão;

## ❖ Exemplo: Classe de números complexos

```
class Complexo:
    def __init__(self, real, imag):
        self.real = real;
        self.imag = imag;

    def __add__ (self, other):
        return Complexo(self.real + other.real, self.imag + other.imag)
```

- ➢ O método especial "__add__" é usado para sobrecarregar a operação de adição do objeto do tipo Complexo

```
c1 = Complexo(1,2)
c2 = Complexo(3,4)
c3 = c1 + c2
print(c3.real , c3.imag)
```

```
4 6
```

- ➢ A soma entre os números C1 (1 + 2i) e C2 (3 + 4i) resulta em C3 (4+6i);
- ➢ O operador de adição "+" foi sobrecarregado na classe Complexo!

# Métodos Especiais

| | |
|---|---|
| __init__ | Chamado automaticamente ao se criar um objeto. É o construtor da classe e responsável por inicializar os atributos; |
| __str__ | Chamado quando você tenta imprimir o objeto, retornando uma representação na forma de string do objeto; |
| __eq__ | Usado para comparar dois objetos usando o operador '=='. Retorna True se os objetos são iguais e False caso contrário; |
| __lt__ | Usado para comparar dois objetos usando o perador '<'. Retorna True ou False; |
| __le__ | Usado para comprar dois objetos usando o operador '<='. Retorna True ou False; |
| __gt__ | Usado para comprar dois objetos usando o operador '>'. Retorna True ou False; |
| __ge__ | Usado para comprar dois objetos usando o operador '>='. Retorna True ou False; |

# Métodos Especiais

FIAP

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

```
joao = Pessoa("João", "30")
print(joao)
```

```
<__main__.Pessoa object at 0x22e5360>
```

# Métodos Especiais

FIAP

```python
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def __str__(self):
        return f"Meu nome é {self.nome}, e tenho {self.idade} anos"
```

```python
joao = Pessoa("João", "30")
print(joao)
```

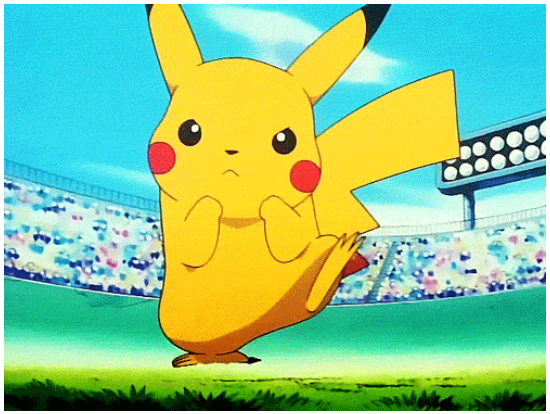
```
Meu nome é João, e tenho 30 anos
```

# Aplicações em IA

- **Sistemas de reconhecimento de imagem:**
  - OpenCV;
  - TensorFlow;
  - PyTorch;
- **Chatbots:**
  - ChatterBot;
  - BotStar;
  - Rasa
- **Sistemas de aprendizado de máquina:**
  - Keras;
  - TensorFlow;
  - PyTorch;
- **Sistemas de processamento de linguagem natural:**
  - textBlob;
  - Natural Language Toolkit - NLTK;
  - spaCy;

# Exercícios

1. Crie uma classe "Pokemon" que possua atributos como "nome", "tipo", "ataque" e "defesa". Adicione métodos para retornar esses atributos e sobrecarregue o operador de comparação para permitir a comparação de dois pokemons com base em seu ataque.
2. Crie uma classe "Treinador" que herde da classe "Pessoa" e adicione um atributo "time" que será uma lista de pokemons. Adicione métodos para adicionar e remover pokemons do time.
3. Crie uma classe "Batalha" que possua dois treinadores e determine o vencedor da batalha com base nos pokemons de seus times.
4. Crie uma classe "Pokebola" que possua um pokemon capturado. Adicione métodos para liberar o pokemon e trocar o pokemon capturado.
5. Crie uma classe "Loja Pokemon" que venda itens para treinadores, como poções e pokebolas. Adicione métodos para adicionar e remover itens da loja e para realizar compras.
6. Crie uma classe "Pokedex" que possua informações sobre os pokemons, como ataques, defesas e tipos. Adicione métodos para adicionar pokemons na pokedex e pesquisar informações sobre um pokemon específico.



This presentation has been designed using images from Flaticon.com
Images from Pokémon: The Pokémon Company