Orientação a Objetos em Python

Prof. Lucas Boaventura

Histórico do Python

- Python foi criado no final dos anos 1980 por Guido van Rossum no Centro de Ciência da Computação na Holanda.
- Lançado pela primeira vez em 1991.

- Diferente do Java, Python não foi inicialmente projetado para ser puramente Orientado a Objetos, mas incorporou o paradigma de forma nativa e poderosa.
- Foco em legibilidade e simplicidade de código.

Características do Python

- **Linguagem Orientada a Objetos (Multipardigma):** Suporta completamente o paradigma de OO, mas também permite programação procedural e funcional.

- **Gerenciamento Automático de Memória (Garbage Collection):** Assim como Java, Python gerencia automaticamente a alocação e desalocação de memória.

Características do Python

- **Tipagem Dinâmica e Forte:** Variáveis não têm um tipo fixo, mas os valores sim (dinâmica). Tipos são verificados em tempo de execução e conversões implícitas são limitadas (forte).

- **Sintaxe Clara e Concisa:** Utiliza indentação para definir blocos de código, o que favorece a legibilidade.

- **Grande Biblioteca Padrão:** Oferece uma vasta coleção de módulos e pacotes para diversas finalidades.

Características do Python

- **Linguagem Interpretada:** Não há um passo de compilação explícito como em Java (.java para .class). O código Python é executado diretamente por um interpretador.

Python vs Java

ASPECTO	PYTHON	JAVA
Tipagem	Dinâmica	Estática
Sintaxe	Concisa, Legível	Verbosa
Execução	Interpretada	Compilada com Bytecode via JVM
Herança	Suporte à Herança Múltipla	Apenas Herança Simples
Bibliotecas	Rico ecossistema de pacotes	Padrão com bibliotecas robustas

Sintaxe do Python

- **Nomes de Classes começam com letras maíusculas**
- **Nomes de variáveis/atributos, funções e métodos usam snake_case**
- **Para executar:** $ python3 meu_programa.py

```python
# meu_programa.py

class MeuPrograma:
    def saudar(self):
        print("Olá Mundo!!!")

if __name__ == "__main__":
    programa = MeuPrograma()
    programa.saudar()
```

Tipos de Dados em Python

- Em Python, tudo é um objeto. Não há "tipos primitivos" no sentido de Java.

Tipo de Dado (Objeto)	Definição	Exemplo
int	Inteiros	idade = 18 ou idade = int(18)
float	Ponto Flutuante	altura = 1.78 ou altura = float(1.78)
str	Strings	nome = “Lucas” ou nome = str(“Lucas”)
list	Listas/Arrays/Vetores	pares = [120, 26, 18]
dict	Dicionários	pessoa = {"nome": "Arthur", "idade": 23}
bool	Booleanos	eh_verdade = True ou test = False

Exemplo: Classe String (str)

- Métodos são chamados usando a sintaxe objeto.metodo().
- Python possui funções built-in como len() que operam em diferentes tipos de dados.

```python
frase_completa = "Olá!"
# Em Python, strings são objetos imutáveis
# frase_completa = str("Olá!") # Também possível, mas menos comum

tamanho = len(frase_completa) # Função len() funciona para muitos tipos
print(f"Tamanho da frase: {tamanho}")

# Métodos comuns da classe str:
print(frase_completa.upper())        # OLÁ!
print(frase_completa.lower())        # olá!
print(frase_completa.replace("!", "?")) # Olá?
print(frase_completa.startswith("O"))  # True
```

Criando suas Classes e Objetos

```python
class Cachorro:
    def __init__(self, nome, raca): # Construtor
        self.nome = nome
        self.raca = raca

    def latir(self):
        return f"{self.nome} da raça {self.raca} está latindo!"

# Criando objetos (instâncias da classe Cachorro)
meu_cachorro = Cachorro("Retrivéu", "Golden Retriever")
outro_cachorro = Cachorro("Thor", "Pastor Alemão")

# Acessando atributos e chamando métodos
print(meu_cachorro.nome)  # Retrivéu
print(outro_cachorro.latir()) # Thor da raça Pastor Alemão está latindo!
```

Conceitos

- **class:** Palavra-chave para definir uma classe.
- **__init__**: Método construtor, chamado ao criar um novo objeto.
- **self**: Referência à própria instância do objeto (primeiro parâmetro de todos os métodos de instância).

- **Atributos**: Variáveis associadas a um objeto (**self.nome**, **self.raca**).

- **Métodos**: Funções associadas a um objeto (**latir**).

```python
class Cachorro:
    def __init__(self, nome, raca): # Construtor
        self.nome = nome
        self.raca = raca

    def latir(self):
        return f"{self.nome} da raça {self.raca} está latindo!"

# Criando objetos (instâncias da classe Cachorro)
meu_cachorro = Cachorro("Retrivéu", "Golden Retriever")
outro_cachorro = Cachorro("Thor", "Pastor Alemão")

# Acessando atributos e chamando métodos
print(meu_cachorro.nome)  # Retrivéu
print(outro_cachorro.latir()) # Thor da raça Pastor Alemão está latindo!
```

Herança em Python

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

    def comer(self):
        return f"{self.nome} está comendo."

class Gato(Animal): # Gato herda de Animal
    def __init__(self, nome, cor):
        super().__init__(nome) # Chama o construtor da classe pai
        self.cor = cor

    def miar(self):
        return f"{self.nome} ({self.cor}) está miando."

meu_gato = Gato("Charlotte", "Branco")
print(meu_gato.comer()) # Charlotte está comendo. (Método herdado)
print(meu_gato.miar())  # Charlotte (Branco) está miando.
```

Herança em Python - Conceitos

- **Classe Pai (Superclasse):** Animal
- **Classe Filha (Subclasse):** Gato
- **super().__init__()**: Utilizado para chamar o construtor da classe pai.

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

    def comer(self):
        return f"{self.nome} está comendo."

class Gato(Animal): # Gato herda de Animal
    def __init__(self, nome, cor):
        super().__init__(nome) # Chama o construtor da classe pai
        self.cor = cor

    def miar(self):
        return f"{self.nome} ({self.cor}) está miando."

meu_gato = Gato("Charlotte", "Branco")
print(meu_gato.comer()) # Charlotte está comendo. (Método herdado)
print(meu_gato.miar())  # Charlotte (Branco) está miando.
```

Encapsulamento - Público

- Python não possui modificadores de acesso (public, private, protected) como Java.
- O encapsulamento é feito por convenção.
- **Atributos Públicos:** Acessíveis diretamente de fora da classe.

```python
class Carro:
    def __init__(self, marca):
        self.marca = marca # Atributo público
```

Encapsulamento - Protegido

- **Atributos Protegidos (Convenção):** Prefixo com um único underscore (_). Sugere que o atributo não deve ser acessado diretamente de fora.

```python
class ContaBancaria:
    def __init__(self, saldo_inicial):
        self._saldo = saldo_inicial # Atributo protegido (por convenção)
```

Encapsulamento - Privado

- **Atributos Privados (Name Mangling):** Prefixo com dois underscores (__). Python "muda o nome" do atributo para evitar colisões em subclasses. Ainda é acessível, mas não é uma prática acessar diretamente.

```python
class Pessoa:
    def __init__(self, nome):
        self.__nome_completo = nome # Atributo "privado"
```

Recursos Adicionais em Python

- **Propriedades (@property):** Permitem criar métodos para acessar e modificar atributos como se fossem atributos diretos, adicionando lógica. (Link exemplo github)
- **Métodos de Classe (@classmethod):** Métodos que operam na classe, não em instâncias. (Link exemplo github)
- **Métodos Estáticos (@staticmethod):** Funções utilitárias que pertencem à classe, mas não acessam a instância ou a classe. (Link exemplo github)
- **Métodos Mágicos/Dunder Methods (__str__, __len__, etc.):** Métodos com nomes especiais (começando e terminando com dois underscores) que permitem que objetos se comportem como tipos built-in. (Link exemplo github)

Documentações Python

- Documentação Oficial do Python: https://docs.python.org/3/
- PEP 8 Style Guide: https://peps.python.org/pep-0008/
- Real Python Tutorials: https://realpython.com/

Orientação a Objetos

Dúvidas?