PYTHON

Tags	BACK-END	CONCEITOS	FRONT-END

Conceitos 📚

▼ Classes, Métodos e Instância em Python

> Em Python, classes são estruturas que definem propriedades (atributos) e comportamentos (métodos).

- `__init__` é um método especial usado para inicializar objetos.
- Instâncias são objetos criados a partir de uma classe.

ex:

```
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.i
```

```
# Criando uma instância da classe Pessoa
pessoa1 = Pessoa("João", 25)
pessoa1.saudacao()
```

▼ Herança em Python

- Permite que uma classe (subclasse) herde propriedades e métodos de outra classe (superclasse).
- A subclasse pode adicionar ou modificar comportamentos.

ex:

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

    def som(self):
        print(f"{self.nome} faz um som.")

# Classe que herda de Animal
class Cachorro(Animal):
    def latir(self):
        print(f"{self.nome} está latindo!")

# Criando uma instância da classe Cachorro
cachorro = Cachorro("Buddy")
cachorro.som()  # Chama o método da classe Animal
cachorro.latir()  # Chama o método específico da classe Cac
```

▼ Polimorfismo em Python

- Objetos de diferentes classes podem responder ao mesmo método de maneiras diferentes.

- Promove a flexibilidade e reutilização de código

ex:

```
class Forma:
    def calcular_area(self):
        pass

# Classes que implementam a mesma interface
class Retangulo(Forma):
    def __init__(self, altura, largura):
        self.altura = altura
        self.largura = largura

    def calcular_area(self):
        return self.altura * self.largura

class Circulo(Forma):
    def __init__(self, raio):
        self.raio = raio

    def calcular_area(self):
        return 3.14 * self.raio ** 2

# Função que usa polimorfismo
def calcular_e_mostrar_area(forma):
    print(f"A área é: {forma.calcular_area()}")

# Criando instâncias das classes
retangulo = Retangulo(5, 10)
circulo = Circulo(3)

# Chamando a função com diferentes tipos de objetos
calcular_e_mostrar_area(retangulo)
calcular_e_mostrar_area(circulo)
```

▼ Exceções em Python

- Exceções são eventos excepcionais que podem ocorrer durante a execução do programa.
- Podem ser tratadas com blocos `try/except` para lidar com erros de forma controlada.

ex:

```
# Exemplo de exceção personalizada
class SaldoInsuficiente(Exception):
    pass

class ContaBancaria:
    def __init__(self, saldo):
        self.saldo = saldo

    def sacar(self, valor):
        if valor > self.saldo:
            raise SaldoInsuficiente("Saldo insuficiente par
        else:
            self.saldo -= valor
            print(f"Saque de {valor} realizado. Novo saldo:

# Uso da exceção
conta = ContaBancaria(1000)

try:
    conta.sacar(1200)
except SaldoInsuficiente as e:
    print(e)
```

- Além do bloco `try/except`, Python oferece o bloco `finally` que é executado independentemente se uma exceção é lançada ou não. É útil para ações que devem ser realizadas, independentemente do resultado do bloco `try`.

ex:

```
try:
    # Código que pode gerar uma exceção
    resultado = 10 / 0
except ZeroDivisionError:
    # Tratando uma exceção específica
    print("Erro: Divisão por zero!")
except Exception as e:
    # Tratando exceções genéricas
    print(f"Erro: {e}")
finally:
    # Bloco executado sempre
    print("Finalizando o bloco try/except.")
```

▼ Setter e Getters em Python

`@property`:

- O decorador `@property` é usado para criar métodos getter. Ele permite acessar o valor de um atributo como se fosse um atributo de instância, sem a necessidade de chamar um método explicitamente.
- Isso ajuda a encapsular a lógica interna do atributo e fornece uma interface mais limpa para o acesso.

`@atributo.setter`:

- O decorador `@atributo.setter` é usado para criar métodos setter. Ele permite definir valores para atributos com validações ou lógica específica antes da atribuição.
- Isso ajuda a controlar como os valores são atribuídos aos atributos, garantindo que certas condições sejam atendidas.

ex:

```
class Pessoa:
    def __init__(self, nome, idade):
        self._nome = nome  # Atributo protegido
        self._idade = idade
```

```
    @property
    def nome(self):
        return self._nome

    @property
    def idade(self):
        return self._idade

    @nome.setter
    def nome(self, novo_nome):
        if len(novo_nome) > 0:
            self._nome = novo_nome
        else:
            print("Nome inválido. Deve ter pelo menos um ca

    @idade.setter
    def idade(self, nova_idade):
        if nova_idade >= 0:
            self._idade = nova_idade
        else:
            print("Idade inválida. Deve ser um valor positi

# Criando uma instância da classe Pessoa
pessoa = Pessoa("Maria", 30)

# Usando os getters
print(f"Nome: {pessoa.nome}")
print(f"Idade: {pessoa.idade}")

# Usando os setters
pessoa.nome = "Joana"
pessoa.idade = 25
```

▼ Estruturas de dados do Python

Objetos vazios: Se toda classe herda de `object`, entao automaticamente podemos instanciar um `object` diretamente. Porém nao podemos incrementar atributos nesse objeto instanciado.

Agora se voce instancia um um objeto de uma classe vazia vai dar certo, nesse caso voce pode sim incrementar novos atributos

ex:

```
class MeuObjeto(): pass

obj = MeuObjeto()
m.nome = 'Pedro Samuel'
## Se printar {m.nome vou receber o valor passado, mesmo se
## objeto da classe está vazio
```

Tuplas:

> São listas imutaiveis, ou seja, diferente das listas comuns [“banana”, “maça”, “abacaxi”].
>
> Essas Listas imutaveis não podem ter seu valores alterados, como também não pode excluir e nem alterar os objetos dentro dela.

```
tupla = ("FB", 177.46, 178.67, 175.79) # imutavel
lista = ["FB", 177.46, 178.67, 175.79] # mutavel
```

Dataclasses:

> É uma abordagem simples para gerar classes com tipagens. Podendo ser uma alternativa para armazenar dados.

```
from dataclasses import dataclass

@dataclass(order=True)
class Movie:
    title: str
    release_year: int
    release_month: int = None
    release_day: int = None
    rating: float = None

    def better_than(self, other):
        return self.rating > other.rating


movie1 = Movie("Ocean's Eleven", 2001, 12, rating=7.7)
movie2 = Movie("Ocean's Eleven", 1960, rating=6.5)

print(movie1)
print(movie2)
print(movie1 < movie2) # Exemplo de verificação
print(movie1.better_than(movie2))
```

- `@dataclass(order=True)` é uma decoração especial para definir a classe como tipada, *order= True* permite criar verificações de instancias como no exemplo acima ⬆

Dicionários:

> Dicionário é uma forma de estruturar seus dados com chave e valores, podendo ser usado para armazenar e manipular seus dados de forma eficiente, especialmente quando você precisa acessar seus valores de uma lista com base em uma chave, sem precisar usar index.
>
> - Um dicionário é definido usando chaves `{}`

ex:

```
stocks = {
"AAPL": (150.0, 140.0, 160.0),
"GOOG": (1235.20, 1242.54, 1231.06),
"MSFT": (110.41, 110.45, 109.84),
"FB": (177.46, 178.67, 175.79),
}
```

Esse exemplo é mostrado um dicionario com chaves representando ações de bolsas de valores e os valores inseridos dentro dessas chaves são tuplas imutaveis, mostrando o valor final, o menor e o mair valor daquela ação

Usando `defaultdict`

`defaultdict` é uma subclasse do dicionário embutido. Ele substitui um valor padrão para chaves que ainda não foram definidas. Isso pode ser útil quando você está manipulando dados e não quer lidar com erros de chaves ausentes.

ex:

```
from collections import defaultdict

# Cria um defaultdict com inteiros como o tipo padrão
contagens = defaultdict(int)

# Adiciona valores a algumas chaves
contagens['maçãs'] += 5
contagens['bananas'] += 2

# Acessa uma chave que ainda não foi definida
print(contagens['peras'])  # Retorna 0, o valor padrão p
ara inteiros.
```

Outro ex:

```
from collections import defaultdict

# Configurando o defaultdict com valor padrão None
```

```
stocks = defaultdict(lambda: None)

# Adicionando valores aos stocks
stocks["AAPL"] = (150.0, 140.0, 160.0)
stocks["GOOG"] = (1235.20, 1242.54, 1231.06)

# Tentando acessar uma chave inexistente
print(stocks["AMZN"])  # Saída: None

# Acessando as chaves do defaultdict
print(stocks.keys())  # Saída: dict_keys(['AAPL', 'GOOG'])
```