E-BOOK
DO ESTUDANTE

Programação Orientada a Objetos em Python

Python diferente!

RESILIA

SUMÁRIO

Programação Orientada a Objetos em Python

Python diferente!

Python é uma linguagem de programação que suporta totalmente a Programação Orientada a Objetos (POO), um paradigma que se baseia na ideia de que os programas são compostos por objetos que interagem entre si para realizar tarefas. Na POO, o código é organizado em classes, que são modelos para criar objetos com atributos e métodos. Em Python, as classes são criadas usando a palavra-chave "class", e os objetos são criados a partir dessas classes usando a sintaxe de instanciação.

A POO em Python permite que desenvolvedores criem classes que representem objetos do mundo real ou conceitos abstratos, como contas bancárias, carros, animais, entre outros. Além disso, o Python oferece recursos poderosos, como herança, polimorfismo, encapsulamento e abstração, que permitem aos desenvolvedores escreverem códigos mais limpos, reutilizáveis e fáceis de manter. A POO em Python é amplamente usada em diversos campos, como desenvolvimento web, científico, de jogos, entre outros.

Neste e-book, estudaremos os fundamentos da Programação Orientada a Objetos utilizando a linguagem Python. Veremos conceitos importantes como classes, objetos, métodos e atributos, além de abstração, herança e polimorfismo.

CONTEXTUALIZANDO

A **Programação Orientada a Objetos** (POO) em Python é uma habilidade fundamental para os profissionais de análise de dados. A POO permite que os desenvolvedores criem classes que representam objetos do mundo real ou conceitos abstratos, como clientes, produtos ou transações, permitindo que as empresas criem soluções de análise de dados mais eficientes e escaláveis, com uma arquitetura mais organizada e reutilizável.

Além disso, o **Python** é uma das linguagens de programação mais usadas em análise de dados e ciência de dados. A linguagem é conhecida por sua sintaxe limpa e clara, além de possuir uma vasta biblioteca de funções e pacotes que facilitam a análise de dados e o aprendizado de máquina. A POO em Python permite que os desenvolvedores criem soluções personalizadas e flexíveis, adaptadas às necessidades específicas de cada empresa e setor.

Portanto, a POO em **Python** é uma habilidade altamente valorizada no mercado de trabalho em análise de dados. Os profissionais que dominam essa técnica podem criar soluções mais robustas e escaláveis, além de integrá-las a outras ferramentas e tecnologias de análise de dados. Com o crescente aumento da demanda por soluções de análise de dados, a habilidade em POO em Python é uma vantagem competitiva para os profissionais que desejam se destacar nesse mercado.

CLASSES, OBJETOS, MÉTODOS E ATRIBUTOS

A **programação orientada a objetos** (POO) é um paradigma de programação que utiliza classes e objetos para organizar e estruturar o código. Em Python, classes, objetos, métodos e atributos são elementos fundamentais da POO e são usados para criar estruturas de dados e executar tarefas específicas. Neste texto, vamos explorar cada um desses conceitos em detalhes.

Uma **classe** é um modelo ou uma definição para a criação de objetos em Python. Ela é definida utilizando a palavra-chave "class" seguida pelo nome da classe. Por exemplo, vamos criar uma classe chamada "Carro":

```
class Carro:
    pass
```

Essa é a forma mais simples de definir uma classe. O bloco de código "pass" é usado para indicar que não há implementação específica para a classe ainda. Uma classe pode conter atributos (variáveis) e métodos (funções) que serão usados pelos objetos criados a partir dela.

Um **objeto** é uma instância de uma classe. Ele é criado a partir de uma classe e possui todos os atributos e métodos definidos pela classe. Para criar um objeto, usamos a sintaxe a seguir:

```
meu_carro = Carro()
```

Aqui, "meu_carro" é um objeto da classe "Carro". Podemos criar quantos objetos quisermos a partir da mesma classe.

Um **método** é uma função definida dentro de uma classe, que pode ser chamada em um objeto daquela classe. Ele pode ser usado para modificar os atributos do objeto ou executar alguma ação específica. Por exemplo, vamos definir um método "ligar" para a classe "Carro":

```
class Carro:
    def ligar(self):
        print("O carro está ligado.")
```

Nesse exemplo, definimos um método chamado "ligar" que imprime uma mensagem na tela. O parâmetro "self" é uma referência ao objeto que chamou o método e é usado para acessar os atributos do objeto.

Para chamar o método "ligar" em um objeto, fazemos o seguinte:

```
meu_carro = Carro()
meu_carro.ligar()
```

Isso imprimirá a mensagem "O carro está ligado." na tela.

Um **atributo** é uma variável definida dentro de uma classe que armazena informações sobre um objeto. Ele pode ser acessado e modificado por meio de métodos ou diretamente por um objeto. Por exemplo, vamos definir um atributo "cor" para a classe "Carro":

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

    def ligar(self):
        print("O carro está ligado.")
```

Aqui, definimos um método especial chamado "init", que é executado quando um objeto é criado a partir da classe. Ele recebe o parâmetro "cor" e define o atributo "cor" do objeto com esse valor.

Para acessar ou modificar um atributo de um objeto, usamos a sintaxe a seguir:

```
meu_carro = Carro("azul")
print(meu_carro.cor)
meu_carro.cor = "vermelho"
print(meu_carro.cor)
```

Nesse exemplo, criamos um objeto "meu_carro" com a implementação anterior, adicionando mais atributos e métodos.

```
class Carro:
    def __init__(self, cor, marca, modelo, ano):
        self.cor = cor
        self.marca = marca
        self.modelo = modelo
        self.ano = ano
        self.ligado = False
        self.velocidade = 0

    def ligar(self):
        self.ligado = True
        print("O carro está ligado.")

    def desligar(self):
        self.ligado = False
        print("O carro está desligado.")

    def acelerar(self, velocidade):
        if self.ligado:
            self.velocidade += velocidade
            print(f"O carro acelerou para {self.velocidade} km/h.")
        else:
            print("Não é possível acelerar o carro, pois ele está desligado.")

    def frear(self, velocidade):
        if self.velocidade >= velocidade:
            self.velocidade -= velocidade
            print(f"O carro reduziu para {self.velocidade} km/h.")
        else:
            print("Não é possível frear o carro, pois ele está parado.")
```

Aqui, adicionamos mais atributos à classe "Carro", como marca, modelo e ano. Também adicionamos dois novos métodos: "desligar", que desliga o carro, e "frear", que reduz a velocidade do carro.

Para usar esses novos métodos e atributos, podemos criar um objeto da classe "Carro" e chamá-los:

```
meu_carro = Carro("azul", "Toyota", "Corolla", 2020)
meu_carro.ligar()
meu_carro.acelerar(50)
meu_carro.frear(20)
meu_carro.desligar()
```

Até aqui, vimos os conceitos fundamentais da programação orientada a objetos em Python: classes, objetos, métodos e atributos. As classes são usadas para definir modelos para a criação de objetos. Os objetos são instâncias de classes e contêm atributos e métodos definidos pela classe. Já os métodos são funções definidas em uma classe que podem ser chamadas em um objeto. Por fim, os atributos são variáveis definidas em uma classe que armazenam informações sobre um objeto.

Com esses conceitos em mente, podemos criar estruturas de dados complexas e executar tarefas específicas em nossos programas Python. No entanto, caso seja necessário criar estruturas modulares, usaremos a abstração. Vamos entender sobre isso a seguir.

ABSTRAÇÃO

A **abstração** é um conceito essencial em programação orientada a objetos e é amplamente utilizado em Python, especialmente na análise de dados. A abstração permite que sejam criados modelos de objetos que podem ser reutilizados e adaptados a diferentes situações, tornando o código mais modular, flexível e fácil de manter.

Na **análise de dados**, a abstração é particularmente útil para representar entidades do mundo real em termos de seus atributos e comportamentos. Por exemplo, podemos criar uma classe "Cliente", que representa um cliente e contém seus atributos, como nome, endereço e idade, bem como métodos que representam suas ações, como fazer um pedido ou fazer uma compra.

Aqui está um exemplo de código em Python que ilustra a abstração na criação de uma classe "Cliente":

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

    def fazer_pedido(self, pedido):
        # código para fazer um pedido
        pass

    def fazer_compra(self, carrinho):
        # código para fazer uma compra
        pass
```

Nesse exemplo, a classe "Cliente" define um construtor que inicializa os atributos de nome, endereço e idade. Também possui métodos para fazer um pedido e fazer uma compra, mas os detalhes desses métodos não são importantes para a classe como um todo. Em vez disso, eles representam comportamentos abstratos que podem ser adaptados a diferentes cenários.

Além disso, a abstração em Python também é útil na criação de classes abstratas e métodos abstratos. Uma **classe abstrata** é uma classe que não pode ser instanciada, mas define uma estrutura geral para outras classes que a implementam. Um **método abstrato** é um método que não tem implementação e é definido apenas na classe abstrata. As subclasses que herdam da classe abstrata devem implementar esses métodos.

Aqui está um exemplo de código em Python que ilustra a criação de uma classe abstrata "ModeloDeRegressao", com um método abstrato "calcular":

```
from abc import ABC, abstractmethod

class ModeloDeRegressao(ABC):
    @abstractmethod
    def calcular(self, dados):
        pass

class RegressaoLinear(ModeloDeRegressao):
    def calcular(self, dados):
        # código para calcular a regressão linear
        pass

class RegressaoLogistica(ModeloDeRegressao):
    def calcular(self, dados):
        # código para calcular a regressão logística
        pass
```

Nesse exemplo, a classe "ModeloDeRegressao" é uma classe abstrata que define um método abstrato "calcular", que deve ser implementado pelas subclasses. As subclasses "RegressaoLinear" e "RegressaoLogistica" herdam da classe "ModeloDeRegressao" e implementam o método "calcular" de maneira diferente, dependendo do tipo de regressão que está sendo realizada.

Em resumo, a abstração é um conceito importante em Python e é amplamente utilizado na análise de dados para representar entidades do mundo real em termos de seus atributos e comportamentos. A abstração também é útil na criação de classes abstratas e métodos abstratos, permitindo que desenvolvedores criem modelos abstratos que podem ser adaptados a diferentes cenários. Esses conceitos possibilitam que um código seja criado de forma mais modular, flexível e fácil de manter, além de fornecer uma maneira de criar modelos de objetos que podem ser reutilizados em diferentes projetos.

Por exemplo, uma pessoa profissional de Python que trabalha com análise de dados pode criar uma classe abstrata "ModeloDeClassificacao", que define um método abstrato "treinar" e um método abstrato "classificar". As subclasses que herdam dessa classe abstrata podem implementar esses métodos de acordo com o tipo específico de classificação que está sendo realizado, como a classificação binária ou a classificação multiclasse.

Além disso, a abstração também é útil na criação de funções genéricas que podem ser usadas com diferentes tipos de objetos. Por exemplo, podemos criar uma função que calcula a média de uma lista de objetos, independentemente do tipo dos objetos. Aqui está um exemplo de código em Python que ilustra essa ideia:

```
def media(lista):
    soma = 0
    for item in lista:
        soma += item
    return soma / len(lista)
```

Nesse exemplo, a função "media" calcula a média de uma lista de objetos, independentemente do tipo desses objetos. Isso permite que a função seja reutilizada em diferentes cenários, tornando o código mais modular e fácil de manter.

Em resumo, a abstração é um conceito essencial em Python e é amplamente utilizada na análise de dados para representar entidades do mundo real em termos de seus atributos e comportamentos. A abstração permite que sejam criados modelos de objetos que podem ser reutilizados e adaptados a diferentes situações, tornando o código de fácil entendimento, sendo ele criado em módulos.

A criação de classes abstratas e métodos abstratos permite que sejam criados modelos abstratos que podem ser implementados de maneiras diferentes, dependendo do contexto. A abstração também permite que sejam criadas funções genéricas que podem ser usadas com diferentes tipos de objetos, tornando o código com melhor legibilidade e mais facilmente ajustável.

HERENÇA

A **herança** é um conceito fundamental da programação orientada a objetos que permite que as classes compartilhem atributos e métodos com outras classes. Em Python, a herança é implementada usando a palavra-chave "class" seguida pelo nome da classe que está sendo criada e, entre parênteses, o nome da classe da qual ela está herdando.

Um exemplo de uso da herança na análise de dados é quando se tem várias classes que representam diferentes tipos de dados, mas compartilham alguns atributos e métodos em comum. Nesse caso, é possível criar uma classe-base que contém os atributos e métodos comuns e, em seguida, criar subclasses que herdam esses atributos e métodos. Aqui está um exemplo de código em Python que ilustra essa ideia:

```
class Dado:
    def __init__(self, nome, data, valor):
        self.nome = nome
        self.data = data
        self.valor = valor

class DadoFinanceiro(Dado):
    def __init__(self, nome, data, valor, categoria):
        super().__init__(nome, data, valor)
        self.categoria = categoria

class DadoDemografico(Dado):
    def __init__(self, nome, data, valor, genero):
        super().__init__(nome, data, valor)
        self.genero = genero
```

Nesse exemplo, a classe "Dado" é a classe-base que contém os atributos comuns a todos os tipos de dados, como o nome, a data e o valor. Em seguida, são criadas duas subclasses, "DadoFinanceiro" e "DadoDemografico", que herdam esses atributos da classe "Dado" e adicionam atributos específicos para cada tipo de dado, como a categoria para os dados financeiros e o gênero para os dados demográficos.

Essa estrutura de herança permite que as subclasses herdem os atributos e métodos da classe-base, tornando o código de fácil manutenção com uma estrutura em módulos. Além disso, a herança também permite que as subclasses adicionem novos atributos e métodos específicos sem afetar a classe-base ou outras subclasses.

Em resumo, a herança é um recurso poderoso da programação orientada a objetos que permite criar classes com atributos e métodos comuns, tornando o código mais modular e fácil de manter. Na análise de dados, a herança pode ser usada para criar classes de diferentes tipos de dados que compartilham atributos e métodos em comum, bem como para criar classes para a leitura de arquivos de diferentes formatos. O uso adequado da herança pode tornar o código mais organizado, facilitando o trabalho de quem trabalha com Python em análise de dados.

POLIMORFISMO

A **herança** é um conceito fundamental da programação orientada a objetos que permite que as classes compartilhem atributos e métodos com outras classes. Em Python, a herança é implementada usando a palavra-chave "class" seguida pelo nome da classe que está sendo criada e, entre parênteses, o nome da classe da qual ela está herdando.

Um exemplo de uso da herança na análise de dados é quando se tem várias classes que representam diferentes tipos de dados, mas compartilham alguns atributos e métodos em comum. Nesse caso, é possível criar uma classe-base que contém os atributos e métodos comuns e, em seguida, criar subclasses que herdam esses atributos e métodos. Aqui está um exemplo de código em Python que ilustra essa ideia:

```
class Dado:
    def __init__(self, nome, data, valor):
        self.nome = nome
        self.data = data
        self.valor = valor

class DadoFinanceiro(Dado):
    def __init__(self, nome, data, valor, categoria):
        super().__init__(nome, data, valor)
        self.categoria = categoria

class DadoDemografico(Dado):
    def __init__(self, nome, data, valor, genero):
        super().__init__(nome, data, valor)
        self.genero = genero
```

Nesse exemplo, a classe "Dado" é a classe-base que contém os atributos comuns a todos os tipos de dados, como o nome, a data e o valor. Em seguida, são criadas duas subclasses, "DadoFinanceiro" e "DadoDemografico", que herdam esses atributos da classe "Dado" e adicionam atributos específicos para cada tipo de dado, como a categoria para os dados financeiros e o gênero para os dados demográficos.

Essa estrutura de herança permite que as subclasses herdem os atributos e métodos da classe-base, tornando o código de fácil manutenção com uma estrutura em módulos. Além disso, a herança também permite que as subclasses adicionem novos atributos e métodos específicos sem afetar a classe-base ou outras subclasses.

Em resumo, a herança é um recurso poderoso da programação orientada a objetos que permite criar classes com atributos e métodos comuns, tornando o código mais modular e fácil de manter. Na análise de dados, a herança pode ser usada para criar classes de diferentes tipos de dados que compartilham atributos e métodos em comum, bem como para criar classes para a leitura de arquivos de diferentes formatos. O uso adequado da herança pode tornar o código mais organizado, facilitando o trabalho de quem trabalha com Python em análise de dados.

Para refletir

- Quais são as principais características da programação orientada a objetos em Python e como elas se diferenciam de outras abordagens de programação?
- Como a herança pode ser utilizada para melhorar a organização e a manutenção de código em um projeto de análise de dados em Python?
- Como o polimorfismo pode ser utilizado para simplificar o código e torná-lo mais flexível em um projeto de análise de dados em Python?

Atividade: Beba mais da fonte

Imagine que precisamos implementar uma classe Pessoa em Python com os atributos "nome", "idade" e "email". A classe deve ter um método "mostrar_dados", que exibe os valores de todos os atributos. Para isso, você deve:

- Criar uma classe chamada "Pessoa";
- Definir os atributos "nome", "idade" e "email" no método init;
- Criar um método "mostrar_dados" que exiba os valores de todos os atributos.

Para ir além

- Vimos que POO significa Programação Orientada a Objetos, um paradigma de programação que se baseia na ideia de que os programas são compostos por objetos que interagem entre si para realizar tarefas. No site abaixo, você encontra formas de implementação de POO em Python: <https://www.devmedia.com.br/introducao-ao-desenvolvimento-web-com-python/6552>

- Estudamos que uma classe é um modelo ou uma definição para criar objetos que possuem atributos e métodos, e que um objeto é uma instância de uma classe. Veja, no site abaixo, exemplos de códigos para desenvolvimento de classes e objetos em Python: <https://www.learnpython.org/pt/Classes_e_Objetos>

- Vimos que Herança em Python é um conceito da programação orientada a objetos (POO) que permite criar novas classes que herdam propriedades e métodos de uma classe existente, conhecida como classe-base ou superclasse. No site abaixo, você encontra exemplos de implementações de herança em Python: <https://programadoresbrasil.com.br/2021/04/heranca-de-classe-em-python/>

- Estudamos que Polimorfismo em Python é um conceito da programação orientada a objetos (POO) que permite que objetos de diferentes classes sejam tratados de forma semelhante. No site abaixo, confira exemplos de polimorfismos usando Python:

  <https://acervolima.com/maneiras-de-implementar-polimorfismo-em-python/>

RESUMO DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Classes, Objetos, Métodos e Atributos

A programação orientada a objetos (POO) é um paradigma de programação que utiliza classes e objetos para organizar e estruturar o código.

Definição de uma Classe

```
class Carro:
    pass
```

Definição de um objeto

```
meu_carro = Carro()
```

Método

```
class Carro:
    def ligar(self):
        print("O carro está ligado.")
```

Chamando Método

```
meu_carro = Carro()
meu_carro.ligar()
```

Atributo

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

    def ligar(self):
        print("O carro está ligado.")
```

Abstração

A abstração é um recurso que permite a criação de modelos de objetos que podem ser reutilizados e adaptados a diferentes situaçõess.

Exemplo de Abstração

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

    def fazer_pedido(self, pedido):
        # código para fazer um pedido
        pass

    def fazer_compra(self, carrinho):
        # código para fazer uma compra
        pass
```

Classe Abstrata

```
from abc import ABC, abstractmethod

class ModeloDeRegresão(ABC):
    @abstractmethod
    def calcular(self, dados):
        pass

class RegressãoLinear(ModeloDeRegresão):
    def calcular(self, dados):
        # código para calcular a regressão
linear
        pass

class RegressãoLogística(ModeloDeRegresão):
    def calcular(self, dados):
        # código para calcular a regressão
logística
        pass
```

Herança

A herança é um conceito fundamental da programação orientada a objetos que permite que as classes compartilhem atributos e métodos com outras classes.

Exemplo de Herança

```
class Dado:
    def __init__(self, nome, data, valor):
        self.nome = nome
        self.data = data
        self.valor = valor

class DadoFinanceiro(Dado):
    def __init__(self, nome, data, valor,
categoria):
        super().__init__(nome, data, valor)
        self.categoria = categoria

class DadoDemografico(Dado):
    def __init__(self, nome, data, valor,
genero):
        super().__init__(nome, data, valor)
        self.genero = genero
```

Polimorfismo

O polimorfismo é um conceito importante da programação orientada a objetos que permite que objetos de diferentes classes sejam tratados de maneira semelhante, de forma transparente para quem programa.

Exemplo de Polimorfismo

```
# Exemplo de classe abstrata e método abstrato
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def falar(self):
        pass

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

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

def fazer_animal_falar(animal):
    animal.falar()

cachorro = Cachorro()
gato = Gato()

fazer_animal_falar(cachorro) # Resultado: "Au
au!"
fazer_animal_falar(gato) # Resultado: "Miau!
```

Até a próxima e
#confianoprocesso

RESILIA