POO e PF

Programação orientada à objetos e Programação Funcional

Fatec 2025

Paradigmas de programção

- POO e PF são paradigmas de programação baseados em princípios básicos da ciência da computação
  - Java = POO
  - Python POO= estilo geral e PF= uso ocasional
- Oferecem um estilo para dividir seu código (manutenibilidade)
- Maioria de programas modernos combinam os dois

POO

- Em Ciência de dados, objetos comuns pode ser um Dataframe do pandas, um array do NumPy, uma figura do Matplotlib
- Um objeto pode:
  - Armazenar dados (Dataframe)
  - Ações associadas (renomear colunas em um dataframe)
  - Interagir com outros objetos (um dataframe pode interagir com um objeto Series do pandas adicionando essa série como uma nova coluna no dataframe)

Classes, métodos e atributos

- Uma classe define um objeto e podemos considera-la como um modelo para criar mais objetos desta variedade
- Métodos são ações que podemos realizar nos objetos dessa classe. Definem o comportamento dessse objeto e podem modificar seus atributos
- Atributos são variáveis que são alguma propriedade dessa classe, e cada objeto pode ter dados diferentes armazenados nesses atributos

Classes, métodos e atributos

- o livro “Cem anos de solidão” é um objeto da classe Livro
- Um dos atributos desse objeto é o número de páginas, outro é Autor
- Um método para chamar esse objeto é Ler
- A classe livro possui muitas instâncias, mas todas tem número de páginas e o método Ler()

Convenções

- Em Python uma classe usa a convenção de nomenclatura CamleCase
- `MyClass`
- Procurar um atributo: `class_instance.atribute`
  - `My_dict = { “coluna_1”:[1,2], “coluna_2”:[“a”,”b”]}`
- Chamada de um método : `class_instance.method()`
  - `df = pd.DataFrame(data=my_dict)`
- Métodos recebem argumentos, atributos não
- Procurando atributos:
  - `df.columns`
  - `df.shape`

Exemplo

- Treinamento de um modelo de ML em 2 arrays, onde x_train contenha as features de treinamento e y_train contenha os rótulos de treinamento:

```
from sklearn.linear_model import logisticRegression
clf = LogisticRegression()
clf.fit(x_train, y_train)
```

- Inicializa-se um novo objeto classificador `logisticRegression` e chamando o método `fit()` nele

Exemplo

- Quais são os objetos e classes?
- Onde está a inicialização dos objetos?
- Onde está a chamada dos métodos?

```python
import matplotlib.pyplot as plt
import numpy as np

n = np.linspace(1, 10, 1000)
line_names = [
    "Constante",
    "Linear",
    "Quadratico",
    "Exponential",
    "Logaritmico",
    "n log n",
]

big_o = [np.ones(n.shape), n, n**2, 2**n, np.log(n), n * np.log(n)]

fig, ax = plt.subplots()
fig.set_facecolor("white")

ax.set_ylim(0, 50)
for i in range(len(big_o)):
    ax.plot(n, big_o[i], label=line_names[i])
ax.set_ylabel("Tempo de execução")
ax.set_xlabel("Tamanho da entrada de dados")
ax.legend()

save_path = "big_o_plot.png"
fig.savefig(save_path, bbox_inches="tight")
```

Definindo as próprias classes

- Instrução class
  - `class RepeatText()`
- Método init é usado para criar atributos

```
def __init__(self, n_repeats):
    self.n_repeats = n_repeats
```

- `Self` = instância nova do objeto
- `Self.n_repeats = n_repeats` , cada instância nova de um objeto `RepeatText` tem um atributo `n_repeats`, que deve ser fornecido sempre que um objeto novo é inicializado

Definindo as próprias classes

```
repeat_twice = RepeatText(n_repeats=2)
```

- Para acessar o atributo:

```
print(repeat_twice.n_repeats)
```

- Para definir outro método:

```
Def multiply_text( self, some_text)
    print((some_text + " ")* self.n_repeats)
```

Exercício: analise este código sob o ponto de vista da POO

```python
import numpy as np

class Goal5Data():
    def __init__(self, name, population, women_in_parliament):
        self.name = name
        self.population = population
        self.women_in_parliament = women_in_parliament

    def print_summary(self):
        null_women_in_parliament = len(self.women_in_parliament) - np.count_nonzero(self.women_in_parliament)
        print(f"Há {len(self.women_in_parliament)} registros de dados")
        print("para o indicador 5.5.1, 'Proporção de assentos ocupados por mulheres nos parlamentos nacionais'.")
        print(f"{null_women_in_parliament} são nulos.")

# Exemplo de dados simulados (24 valores, alguns podem ser zero para simular 'nulos')
women_data = [13.33, 14.02, 14.02, 15.0, 15.5, 16.0, 16.8, 17.0, 18.2, 18.5,
              19.0, 19.3, 20.0, 20.2, 21.0, 21.5, 22.0, 22.5, 23.0, 23.2,
              24.0, 24.5, 0.0, 0.0]  # os zeros simulam dados ausentes

usa = Goal5Data(name="USA", population=336262544, women_in_parliament=women_data)

usa.print_summary()
```

Princípios da POO: Herança

- Significa que podemos estender uma classe criando outra classe que se baseia nela
- Ajuda a reduzir a repetição
- Pode-se criar classes ou usar de uma biblioteca externa
- Para identificar uma classe que usa herança:
  - `Class NovaClasse( ClasseOriginal)`

Herança

```
class Goal5TimeSeries(Goal5Data):
    def __init__(self, name, population, women_in_parliament, timestamps):
        super().__init__(name, population, women_in_parliament)
        self.timestamps = timestamps
```

Método __init__ está um diferente
Super() chama a classe mãe e inicializa os atributos name, population e women_in_parliament

Adicionando um novo método que seja relevante somente para a classe filha

```
    def fit_trendline(self):
            result = linregress(self.timestamps,
    self.women_in_parliament)
            slope = round(result.slope, 3)
            r_squared = round(result.rvalue**2, 3)
            return slope, r_squared, result
```

Herança

```python
import matplotlib.pyplot as plt
from scipy.stats import linregress
# Subclasse com herança
class Goal5TimeSeries(Goal5Data):
    def __init__(self, name, population, women_in_parliament, timestamps):
        super().__init__(name, population, women_in_parliament)
        self.timestamps = timestamps

    def fit_trendline(self):
        result = linregress(self.timestamps, self.women_in_parliament)
        slope = round(result.slope, 3)
        r_squared = round(result.rvalue**2, 3)
        return slope, r_squared, result

    def plot_trendline(self):
        slope, r_squared, result = self.fit_trendline()
        plt.figure(figsize=(10, 6))
        plt.plot(self.timestamps, self.women_in_parliament, 'o', label="Dados reais")
        plt.plot(self.timestamps, result.intercept + result.slope * np.array(self.timestamps), '-', label=f"Tendência (R²={r_squared})")
        plt.title(f"Proporção de mulheres no parlamento ({self.name})")
        plt.xlabel("Ano")
        plt.ylabel("Percentual (%)")
        plt.legend()
        plt.grid(True)
        plt.show()
# Dados simulados
timestamps = list(range(2000, 2024))  # 24 anos
women_data_india = [8.0, 8.2, 8.4, 8.5, 9.0, 9.5, 9.8, 10.0, 10.5, 11.0,
                    11.2, 11.3, 11.5, 12.0, 12.3, 12.5, 12.8, 13.0, 13.2, 13.5,
                    13.8, 14.0, 14.2, 14.5]

india = Goal5TimeSeries("India", 1400000000, women_data_india, timestamps)

# Chamando métodos
india.print_summary()
india.plot_trendline()
```

Encapsulamento

- A classe oculta seus detalhes de acesso externo
- Podemos ver apenas a interface
  - **Interface é o conjunto de métodos e propriedades públicas que um objeto expõe para ser usado por outros objetos**
- Pandas usa encapsulamento fornecendo métodos e atributos que possibilitam interagir com os dados enquanto mantém os detalhes da implementação ocultos
- Um objeto DataFrame encapsula dados e fornece vários métodos para acessá-los, filtrá-los e transformá-los
- Pandas usa numpy nos bastidores, mas a gente não precisa saber!

Abstração

- Vinculada ao encapsulamento
- Deve-se lidar com uma classe com o nível adequado de detalhes
- Podemos escolher por manter detalhes de algum cálculo em um método ou permitir que ele seja acessado por meio da interface
- Menos comum em python

Exemplo

```python
from abc import ABC, abstractmethod # Classe abstrata - só define o que deve ser feito
class ContaBancaria(ABC):
    def __init__(self, titular, saldo=0):
        self.titular = titular
        self._saldo = saldo  # atributo protegido

    def ver_saldo(self):
        print(f"Saldo atual: R$ {self._saldo:.2f}")
    @abstractmethod
    def sacar(self, valor):
        pass
    @abstractmethod
    def depositar(self, valor):
        pass
# Implementação concreta - esconde os detalhes
class ContaCorrente(ContaBancaria):

    def sacar(self, valor):
        if valor > self._saldo:
            print("Saldo insuficiente.")
        else:
            self._saldo -= valor
            print(f"Saque de R$ {valor:.2f} realizado com sucesso.")

    def depositar(self, valor):
        self._saldo += valor
        print(f"Depósito de R$ {valor:.2f} realizado com sucesso.")
# Uso
conta = ContaCorrente("João", saldo=500)

conta.ver_saldo()     # Mostra o saldo
conta.sacar(200)      # Faz um saque
conta.ver_saldo()     # Verifica novamente
```

- ABC = classe base abstrata
  - Declarar **métodos que devem ser implementados** por qualquer classe filha
  - **Evitar que a classe base seja instanciada diretamente**
  - Aplicar o conceito de **abstração** com clareza

- **abstractmethod:**
  É um **decorador** que marca um método como **obrigatório**. Se a subclasse não implementá-lo, o Python gera erro

- Garante que todas as subclasses **sigam a mesma interface**

- Ajuda a aplicar os princípios da **orientação a objetos**, como **abstração** e **polimorfismo**

- Modelos ou contratos que definem métodos obrigatórios para as subclasses implementarem

Polimorfismo

- Podemos ter a mesma interface para classes diferentes
  - simplifica o código e reduz repetição
- Duas classes podem ter um método com o mesmo nome que gera um resultado semelhante, porém os mecanismos internos são diferentes
- As duas classes podem ter a mesma classe mãe e uma classe filha ou podem não estar relacionadas

Polimorfisco

- Na scikit-learn todos os classificadores tem o mesmo método fit para serem treinados com alguns dados, ainda que sejam definidos em classes diferentes

```
from sklearn.linear_model impor LogisticRegression
from sklearn.ensemble import RandomForestClassifier
lr_clf = LogistiRegression()
lr_clf.fit(x_train, y_train)
rf_clf = RandomForestClassifier()
rf_clf.fit(x_train, y_train)
```

Exemplo: polimorfismo

```python
class AlunoRegular:
    def entregar(self):
        print("Aluno regular entregou a tarefa pelo portal da escola.")

class AlunoEAD:
    def entregar(self):
        print("Aluno EAD entregou a tarefa via sistema online.")

class Professor:
    def entregar(self):
        print("Professor não entrega tarefa, ele corrige!")

# Função polimórfica usando duck typing
def entregar_tarefa(participante):
    participante.entregar()  # Não importa a classe, importa ter o método!

# Testando
participantes = [AlunoRegular(), AlunoEAD(), Professor()]

for p in participantes:
    entregar_tarefa(p)
```

Programação funcional

- Apesar de suportar, não é comum escrever
- Linguagem Scala é mais usual e apropriada
- Foco: funções que não mudam
- Funções não devem alterar nenhum dado existente fora delas
- Funções são imutáveis, puras e isentas de efeitos colaterais
- Não afetam nada além do que é refletido no valor que retornam
- Rigorosamente: em PF um programa consiste apenas em avaliar funções
- Funções podem ser aninhadas ou podem ser passadas como argumentos para outras funções

Vantagens da programação funcional

- Fácil de testar porque uma função sempre retorna a mesma saída para uma determinada entrada. Nada fora da função é modificada
- Fácil de paralelizar porque os dados não são modificados
- Obra a escrita de um código modular
- Poder mais conciso e eficiente
- Em python:
  - Funções lambda
  - Funções built-in map e filter

Funções lambda e map()

- Funções lambda são funções pequenas e anônimas usadas para tarefas esporádicas e rápidas
- Anônimas= porque não são definidas como uma função normal

`lambda arguments: expression`

  - Pode receber quantos argumentos quisermos, mas pode ter apenas uma expressão
  - Já que aceitam funções como argumentos, podem aplicar a função a todos os elementos de um iterável (lista)

```
# Lista com percentuais de participação feminina no governo dos EUA
usa_govt_percentages = [13.33, 14.02, 14.02, 14.25, 14.7, 15.1]

# Convertendo para proporções decimais usando programação funcional (map + lambda)
usa_govt_proportions = list(map(lambda x: x / 100, usa_govt_percentages))

# Exibindo os resultados lado a lado
print("Percentual  ->  Proporção")
print("-" * 30)
for perc, prop in zip(usa_govt_percentages, usa_govt_proportions):
    print(f"{perc:>9.2f}%  ->  {prop:.4f}")
```

Função Map

```python
# Lista de notas dos alunos
notas = [75, 40, 90, 55, 62, 30]

# Função para classificar a nota
def classifica_nota(nota):
    return "Aprovado" if nota >= 60 else "Reprovado"

# Usando map() para aplicar a função a cada item da lista
resultados = list(map(classifica_nota, notas))

# Exibindo os resultados
for nota, status in zip(notas, resultados):
    print(f"Nota: {nota} → {status}")
```

Aplicando funções a DataFrames

- Aplicando função lambda a uma coluna em um DataFrame

```
import pandas as pd

# Criando um DataFrame com dados simulados
df = pd.DataFrame({
    "Ano": [2000, 2001, 2002, 2003, 2004, 2005],
    "Estados Unidos": [13.33, 14.02, 14.02, 14.25, 14.7, 15.1]
})

# Usando apply + lambda (programação funcional) para categorizar
df["USA_processed"] = df["Estados Unidos"].apply(
    lambda x: "Mostly male" if x < 50 else "Mostly female"
)

# Exibindo a tabela resultante
print(df)
```

Usando função nomeada x lambda

```python
import pandas as pd

# Dados simulados
df = pd.DataFrame({
    "Ano": [2000, 2001, 2002, 2003, 2004, 2005],
    "Estados Unidos": [13.33, 14.02, 50.02, 55.5, 14.7, 51.0]
})

# Função nomeada (preferida)
def binary_labels(women_in_govt):
    if women_in_govt < 50:
        return "Mostly male"
    else:
        return "Mostly female"

# Usando apply com lambda
df["USA_lambda"] = df["Estados Unidos"].apply(lambda x: "Mostly male" if x < 50 else "Mostly female")

# Usando apply com função nomeada
df["USA_func"] = df["Estados Unidos"].apply(binary_labels)

print(df)
```

Qual paradigma devo usar em Ciência de dados?

- Pequenos projetos:
  - Não se preocupar
  - Scripts funcionam? Tudo bem.
- Projetos grandes:
  - POO para lidar com um dataset que necessitam de muitas ações
  - É possível transformar a natureza do problema em instâncias que precisam ter comportamentos semelhantes, mas atributos ou dados diferentes
  - Escrever classes quando tiver muitas instâncias

Resumo

- POO e PF são paradigmas que serão encontrados quando você ler algum código
- POOfoca os objetos, que são estruturas de dados personalizadas
- PF foca em funções que não alteram os dados
- POO:
  - uma classe define objetos novos que podem ter atributos e métodos
  - Define-se classes, de modo que os métodos e dados associados fiquem juntos, ótimo para ser aplicados em instâncias semelhantes
  - Herança serve para evitar repetição de códigos e polimorfismo para manter as interfaces padronizadas
- PF:
  - O ideal é que tudo fique dentro da função
  - Útil quando temos dados que não mudam e queremos fazer diferentes operações com eles, ou paralisar
  - Funções lambda