In [6]:
```
import pandas as pd
import numpy as np
# Módulo para referenciar tipagens
from typing import List, Any, Tuple
import os

if (os.system("pip install multipledispatch") == 0):
    print("multipledispatch instalado com sucesso!")
else:
    print("Não foi possível instalar o multipledispatch.")
```

```
multipledispatch instalado com sucesso!
```

In [7]:
```
# Módulo externo para criar sobrecargas de funções
from multipledispatch import dispatch
```

In [8]:
```python
class ExtratorDeProbabilidades:
    # Letra a
    def __init__(self, path: str) -> None:
        # Seta o path como um atributo da classe; não será usado
        self.path = path
        try:
            # Cria o dataframe a partir do path passado
            self.df = pd.read_csv(path, sep=',', encoding="iso-8859-1")
        except:
            # Caso não consiga criar, irá simplesmente criar um dataframe vazio
            self.df = pd.DataFrame({})
            print("Ocorreu um erro ao carregar o arquivo.")

    # Letra b
    def carregar_colunas(self, lista_colunas: List[str], quantidade: int) -> pd.DataFrame:
        try:
            # Tenta carregar as colunas baseado no limite que o usuário desejar;
            # Lembrando que na indexação de python, podemos usar da seguinte maneira:
            # x = [1, 2, 3]; x[:2] -> [1, 2]
            # Ou seja, os dois primeiros elementos

            # Nesse caso, retorna a lista de colunas desejada pelo usuário e a quantidade de dados a serem
            # mostrados
            return self.df[lista_colunas][:quantidade]
        except:
            print("Ocoreu um erro ao carregar as colunas")
    # Letra c
    def descarregar_colunas(self) -> None:
        self.df = pd.DataFrame({})

    # Letra d
    def probabilidade_apriori(self, caracteristica: str, valor: Any) -> float or None:
        try:
            # Filtra o dataframe pela coluna desejada da tabela e calcula o tamanho
            num_espec = len(self.df[self.df[caracteristica] == valor])
            # Retorna o tamanho do tabela filtrada dividido pela tabela total
            return num_espec/len(self.df)
        except:
            print("Ocorreu um erro ao calcular a porcentagem.")

    # Letra e
    @dispatch(str, tuple) # Decorator usado para utilizar a sobrecarga; como definido na questão, a função deve po
ssuir mesmo nome
    def probabilidade_apriori_intervalo(self, caracteristica: str, intervalo: Tuple[int, int]) -> float or None:
        try:
            # Extrai o inicio e o fim do intervalo passado
            inicio, fim = intervalo
            # Recebe o tamanho da coluna desejada limitada pelo início e pelo fim
            num_espec = len(self.df[caracteristica][inicio:fim])
            # Retorna o tamanho obtido acima divido pelo tamanho total
            return num_espec / len(self.df[caracteristica])
        except:
            print("Ocorreu um erro ao calcular a porcentagem.")

    # Letra f
    def probabilidade_condicional(self, cond_a: Tuple[str, str], cond_b: Tuple[str, str]) -> float or None:
        try:
            # Recebe duas colunas e valores a serem filtrados
            carac_a, val_a = cond_a
            carac_b, val_b = cond_b
            # Cria duas condições que serão utilizadas para gerar o filtro no dataframe
            filter_a = self.df[carac_a] == val_a
            filter_b = self.df[carac_b] == val_b
            # Retorna o tamanho das duas condições a e b aplicadas no dataframe dividido
            # pelo tamanho do dataframe aplicado à condição b
            return len(self.df[filter_a & filter_b])/len(self.df[filter_b])
        except:
            print("Ocorreu um erro ao calcular a porcentagem.")

    # Letra g
    @dispatch(tuple, tuple)
    def probabilidade_apriori_intervalo(self, a: Tuple[str, str], b: Tuple[str, Tuple[int, int]]):
        try:
            # Recebe os valores das tuplas, como especificado na questão
            carac, valor = a
            # Ignora a coluna passada, pois ela será igual à variável 'carac' (?)
            _, (inicio, fim) = b

            try:
                # Busca os valores que satisfazem a condição de filtro passada dentro do [inicio, fim]
                # -> conta os valores únicos que serão gerados pela tabela e verifica caso exista algum valor
                # que satisfaça a condição; eles serão referenciados no índice 'True'
                qtd_filtro = (self.df[carac][inicio:fim] == valor).value_counts()[True]
            except KeyError:
                # Caso não exista valor que satisfaça a indexação da linha acima, retorna zero
                qtd_filtro = 0

            # Tamanho da coluna com o filtro aplicado
            qtd_total = len(self.df[carac] == valor)
            # Retorna a quantidade do filtro limitada acima junto à quantidade total da linha anterior
            return qtd_filtro/qtd_total
```

```
            except ZeroDivisionError:
                # Ocorre quando qtd_total é zero; nesse caso apenas retorna zero para evitar a divisão por zero
                return 0


    # Desafios
    def desafio_a(self):
        # Filtra o modelo de cambio para manual e com preço menor que 40000
        # -> Normaliza os valores para representar a a porcentagem total relativa aos valores
        # da tabela -> Cria um filtro para verifica quais valores contidos possuem valor maior que 0.9
        # ou seja, valores que possuem mais de 90% de participação
        modelos = self.df[(self.df['transmission'] == 'manual') & (self.df['price'] < 40000)]['model'].value_count
s(normalize=True) > 0.9
        # Cria um conjunto que irá guardar o valor dos carros
        carros = set()
        # Itera sobre os valores verdade da tabela criada pelo filtro, junto aos índices, que nesse caso
        # são os nomes dos carros
        for modelo, valor in zip(modelos.index, modelos):
            # Caso o valor seja true, adiciona o modelo do carro no conjunto
            if valor == True: carros.add(modelo)
        # Retorna o conjunto de carros caso exista pelo menos um, caso contrário retorna "{}"
        return carros if carros else "{}"

    def desafio_b(self, modelo):
        # Gera um dataframe com os filtros passados
        valores = self.df[(self.df['transmission'] == 'manual') & (self.df['model'] == modelo)]
        # Para que o valor gere um limite, iremos receber os índices dos valores
        # Caso o índice seja maior que 2, iremos receber o primeiro e último índice
        if len(valores.index) >= 2:
            # Retorna o primeiro e o último índice, representando os limites
            return (valores.index[0], valores.index[-1])
        # Caso haja apenas um valor, não faz sentido querer enviar um range, pois qualquer range contendo o valor
        # satisfaz a condição; nesse caso enviamos apenas o índice do valor que satisfaz a condição
        elif len(valores.index) == 1:
            return {valores.index[0]}
        else:
            # Caso não exista, retorna None
            return None

def menu(self):
    print("1 - Carregar Arquivo")
    print("2 - Mostrar DataFrame")
    print("3 - Carregar Colunas")
    print("4 - Probabilidade à Priori")
    print("5 - Probabilidade à Priori - Intervalo")
    print("6 - Probabilidade Condicional")
    print("7 - Probabilidade à Priori - Intervalo (Tipo 2)")

    print("9 - Descarregar Arquivo")
    print("0 - Sair")
    opt = int(input("Insira o número relativo à operação desejada "))
    if (opt == 1):
        filename = input("Insira o filename: ")
        self.__init__(filename)
        return True
    elif (opt == 2):
        print(self.df)
        return True
    elif (opt == 3):
        colunas = input("Insira o nome das colunas que deseja verificar, separadas por espaço: ").split()
        lim = int(input("Insira a quantidade de elementos desejada: "))
        print(self.carregar_colunas(colunas, lim))
        return True
    elif (opt == 4):
        carac = input("Insira a coluna: ")
        val = input("Insira o valor")
        print(self.probabilidade_apriori(carac, val))
        return True
    elif (opt == 5):
        carac = input("Insira a coluna desejada")
        inf = input("Insira o limite inferior: ")
        sup = input("Insira o limite superior: ")
        print(self.probabilidade_apriori_intervalo(carac, (inf, sup)))
        return True
    elif (opt == 6):
        cond_a, valor_a = input("Insira a coluna e o valor seguidos por espaço: ").split()
        cond_b, valor_b = input("Insira a segunda coluna e o valor seguidos por espaço: ").split()
        print(self.probabilidade_condicional((cond_a, valor_a), (cond_b, valor_b)))
        return True
    elif (opt == 7):
        cond, valor = input("Insira a coluna e o valor seguidos por espaço: ").split()
        inf = input("Insira o limite inferior: ")
        sup = input("Insira o limite superior: ")
        print(self.probabilidade_apriori_intervalo((cond, valor), (cond, (inf, sup))))
        return True

    elif (opt == 9):
        self.descarregar_colunas()
        return True
    elif (opt == 0):
        print("O programa foi encerrado com sucesso.")
        return
```

Testes

In [9]:
```
t = ExtratorDeProbabilidades('vehicles-light.csv')
t.probabilidade_apriori(caracteristica='region', valor='birmingham')
```

Out[9]: 0.1507537688442211

In [10]:
```
t.probabilidade_apriori_intervalo('region', (1, 10))
```

Out[10]: 0.04522613065326633

In [11]:
```
t.probabilidade_condicional(cond_a=('region', 'birmingham'), cond_b=('transmission', 'other'))
```

Out[11]: 0.018018018018018018

In [12]:
```
t.probabilidade_apriori_intervalo(("transmission", "manual"), ("transmission", (29, 150)))
```

Out[12]: 0.020100502512562814

In [13]:
```
t.desafio_b("f450")
```

Out[13]: {65}

Main Loop

In [14]:
```
while t.menu() != None:
    continue
```

```
1 - Carregar Arquivo
2 - Mostrar DataFrame
3 - Carregar Colunas
4 - Probabilidade à Priori
5 - Probabilidade à Priori - Intervalo
6 - Probabilidade Condicional
7 - Probabilidade à Priori - Intervalo (Tipo 2)
9 - Descarregar Arquivo
0 - Sair
Insira o número relativo à operação desejada 0
O programa foi encerrado com sucesso.
```

In [14]:
```

```