Pandas

O pandas é com toda certeza a biblioteca para análise de dados mais utilizada atualmente, sua versatilidade e praticidade a colocar

Vamos dar início a um guia de como usá-lo:

```
import pandas as pd
```

1. Leitura do banco de dados

Devemos atribuir uma variável ao nosso banco do seguindo modo:

```
var1= pd.read_formato('nome', sep='pode ser de vários tipos')

# Excel:
# x1 = pd.ExcelFile(file) # carrega o arquivo
# df = x1.parse('Sheet1') # carrega a aba

# pd.read_csv("fifa19.csv", dtype={"Potential":"int", "SlidingTackle":"float64"})
```

Visualizando o seus dados em forma de tabela:

```
var1.head()
```

O .head() mostra apenas as primeiras linhas, e podemos colocar dentro do parêntese até que linha gostaríamos de ler, sempre pens

Podemos também visualizar o final do dataset com:

```
var1.tail()
```

Carregando as informações do dataset

```
var1.info()
```

Mostrará o tipo de dados da tabela e seus respectivos nomes

```
var1.shape()
```

Retorna a organização matricial

```
var1.columns()
```

Retorna o nome das colunas

2. Podemos aplicar funções ao nosso dataset

```
def low_values_red(value):
    color = 'red' if value < 92 else 'black'
    return 'color: %s'%color # 'color: red' ou 'color: black'

var1[['c1','c2']].head(10).style.applymap(low_values_red)
```

Nesse caso é uma função que mudará a cor para os valores que forem inferiores a 92.

É interessante saber disso pois pode facilitar nossas análises de diversas formas possíveis, mas é necessário entender melhor sobr

applymap()

Essa função é característica do pandas e ela basicamente aplica uma função já criada por você ao nosso conjunto. Basta colocar a

Outro ponto, na maioria das vezes não é prática ou é indesejado aplicar o nosso def() a todos os dados. Para isso, precisamos delin

```
var1[['c1', 'c2', 'c...']]
```

Isso identifica ao nosso código em qual parte desejamos aplicar nossa função.

apply()

O apply funciona de maneira diferente do applymap, sua verificação ocorre em toda a série do pandas

> Pandas.apply permite que os usuários passem uma função e a apliquem em cada valor da série Panda

Exemplo:

```
def highlight_max(s):
    is_max = s == s.max()
checa esse booleano s == s.max(),
e depois guarda em is_maxreturn ['background-color: yellow' if v else '' for v in is_max]

var1.head(10).style.apply(highlight_max, subset=['c1','c2', 'c3'])
```

Mais sobre formatação em: https://pandas.pydata.org/pandas-docs/stable/user_guide/style.html

3. Filtro e seleções

Usaremos *.loc e .illoc. Vejamos a seguir cada um deles e suas vantegens*

.loc

Essa função no permite selecionar valores a partir do index da tabela. Mas nós sabemos que podemos fazer isso sem precisar de u

Qual seria a vantagem do .loc, afinal?

```
var1.loc[0:5, 'c1']
```

Essa função nos permite selecionar apenas uma coluna e com o tamanho que quisermos, a partir da sintaxe acima

.iloc

É de maneira semelhante mas não se pode selecionar a partir do nome da coluna, apenas com os números referentes a sua posiçã

```
var1.iloc[0:5, 0:3]
```

Irá retornar 6 posições de linha e 4 de coluna.

3.1 Como filtrar valores?

Podemos usar o .loc com a seguinte sintaxe:

```
var1.loc[(var1['c1']>x1) & (var1['c2']>=x2)]
```

Isso irá retornar apenas os dados em que c1 seja maior que x1, em conjunto com os dados em que c2 seja maior ou igual que x2.

Saber desse tipo de filtro é extremamente útil se pensarmos em acoplar o valor retornado a uma variável, dessa maneira podem ter

Há também a função .filter()

```
var1.filter(['c1','c2','c3'...])
```

Retorna somente as colunas e seus valores.

Aumentando a qualidade das nossas seleções

— .isin

Essa função é bastante prática, pois, irá mostrar apenas os valores que estão dentro da lista que configuramos.

```
var1_consulta = var1[var1[c1].isin(['l1', 'l2', 'l3',...])]
var1_consulta.head()
```

Desse modo, iremos visualizar apenas o dataframe que possui os valores de l1, l2, l3… correspondentes a nossa consulta. Também

```
var1_consulta = var1[~var1[c1].isin(['l1', 'l2', 'l3',...])]
var1_consulta.head()
```

Mostra apenas os valores que não correspondem.

— include

Podemos criar uma lista com os tipos de dados presentes no dataframe e realizar uma consulta.

Ex: Tipos numéricos

```
numerics = ['int16', 'float64', 'float32']
df.select_dtypes(include=numerics)
```

.sort_values

O .sort_values serve para os visualizar valores selecionados do dataframe de maneira ascendente ou descendente sendo defininos

```
var1.sort_values(by='c1', ascending=True).head()
```

Mostrará o dataframe com os valores localizados na coluna c1 de maneira ascendente

Estatística descritiva

→ Descrição dos principais dados estatísticos:

```
var1.describe()
```

→ Descrição por coluna

```
var1['c1'].describe(percentiles=[0.1, 0.2, 0.8, 0.99])
```

→ Função de agregação para comparação

```
var1.agg({'c1':['min', 'max', 'skew', 'avg'],
          'c2': ['min', 'max', 'skew', 'avg']})
```

→ Groupby

A função de agrupamento é bastante útil, iremos destrinchar um pouco mais de sua sintaxe nesse momento. Vejamos:

```
var1.groupby('coloque aqui a coluna que irá parear')
```

```
var1.groupby('coloque aqui a coluna que irá parear')['c1', 'c2'].funçãoqualquer()
```

c1 e c2 serão as colunas que iremos comparar, sempre em relação a primeira coluna.

→ Criação de tabela pivô com **.pivot_table**

A tabela pivô ocorre da mesma forma que no excel, nós usaremos valores de uma coluna para parear em relação as outras.

```
pd.pivot_table(var1, index='Coluna de pareamento', aggfunc= 'função qualquer')
```

Com a mesma função podemos trocar as colunas por valores de colunas, localizados nas linhas:

```
pd.pivot_table(var1, index=['Coluna de pareamento'], columns=['c2'],
               aggfunc= 'função qualquer')
```

→ E se quisermos saber qual a representação percentual de um determinado item?

```
var1['c1'].value_counts(normalize=True)
```

Mostrará a representação percentual de acordo com a coluna.

E se forem duas colunas, exemplo: Representação percentual de cada cliente por localidade

```
var1.groupby('Localidade')['Cliente'].value_counts(normalize=True)
```

Normalize é o essencial aqui

.nlargest() e .nsmallest()

Podemos imaginar como sendo um .sort().max() ou .sort.min() de forma acoplada, cumprindo o papel dessas funções com apena

SQL no Pandas

Podemos realizar consultas sql no pandas, com a instalação de

```
import pandasql as ps
```

Obedece a mesma sintaxe de SQL, Select, From, Groupby, Join…

```
query = """ SELECT c1, c2, c3 FROM var1 where c2>33"""
ps.sqldf(query, locals())
```

Preenchendo valores nulos

Primeiro devemos saber onde se encontram os valores nulos e N/A, existem duas funções diferentes para isso.

pd.isna()

Retorna os valores 'nan' do dataframe

.isnull()

Retorna valores nulos e é importante dizer que dados descritivos, funções estão disponíveis para elas.

.notna()

Retorna apenas os valores não nulos

.fillna()

Podemos inserir um valor dentro dos parênteses, geralmente a média dos valores gerais, não nulos.

.dropna()

Dropa os valores nulos

.replace(value, new_value)

Imagine que em um conjunto de dados temos uma coluna em que os valores dela estejam com um ou mais dados em outro tipo,

Criação de uma nova coluna para o filtro:

```
var1['new_column']= np.where(var1[c1]>x1, 'name', 'name se não atender')
```

Extra: Boas práticas com pandas

Se desejarmos contar quantos dados tem de forma agrupada:

```
df.col.value_counts(dropna= False)
```

Ou, também podemos fazer com o uso de groupby, caso o que queremos obter deve estar acoplado com outra categoria

```
df[cols].groupby('col').col2.nunique()
```