**Python Pandas**

**Curso 1 – Tratando e Analisando Dados:**

1. **Aula 1 – Conhecendo Jupyter:**
   1. O primeiro passo é baixar o Anaconda. Basta digitar no google e baixar, ou acesse o link: <https://www.anaconda.com>.
   2. Ambientes virtuais são muito importantes para setar e manter a versão e funcionalidades de todas as bibliotecas, linguagens etc, garantindo que caso queira rodar um código que você escreveu há um ano atrás, ele continue funcionando normalmente.
   3. Pip: gerenciador de pacotes padrão do python.
   4. Conda: gerenciador de pacotes padrão do Anaconda.
   5. Para criar esse ambiente virtual vamos utilizar o seguinte comando no cmd:

conda create --name alura\_pandas python=3.8.5 pandas=1.1.3

* + 1. Podemos alterar o nome, a versão do python e do pandas para quais quisermos.
    2. Caso o cmd esteja devolvendo um erro de HTTP, sugiro que utilize o cmd do próprio anaconda. Se fizer isso, vai criar o ambiente normalmente.
    3. Após o carregamento e criação do ambiente, ao entrar na parte de ambientes do anaconda veremos nosso ambiente lá:



* + 1. Para ativar o nosso ambiente e rolar ele no próprio cmd, precisamos colocar activate e o nome da pasta do nosso ambiente.
    2. Para desativar, basta digitar deactivate.
    3. Para visualizar os ambientes que temos usamos o comando “conda info –envs. Cmd retorna todos os ambientes que temos criados.
    4. Para destruir o ambiente: conda remove –name alura\_pandas –all.
  1. O que aprendemos:
     1. Anaconda é a principal distribuição para cientistas de dados que usam Python;
     2. Jupyter é a nossa ferramenta para executar código Python e visualizar os dados;
     3. Ambientes virtuais ajudam a isolar um projeto para definir a versão das bibliotecas e do próprio Python.

1. **Aula 2 – Importando Dados:**
   1. Estamos utilizando o notebook do jupyter para fazer nossos códigos agora, portanto os atalhos e comandos são meio diferentes.
      1. O jupyter é esse “app” disponível no anacondas:



* + 1. Ele executa e abre diretamente no navegador:



* + 1. A partir dele podemos criar e acessar pastas, arquivos, dentre outras coisas, como o explorer.
    2. Criamos uma pasta de projetos -> curso pandas -> dados, e fizemos o upload dos dados “alugueis” que vamos trabalhar.
    3. Criamos um notebook com o python 3 e começamos importando o pandas e lendo o arquivo:





* + 1. Caso tenha alguma dúvida do que passar como parâmetro e ou quais métodos/funções uma biblioteca tem, pode apertar SHIFT + TAB para ele exibir a ajuda.
  1. Atalhos jupyter:
     1. M: muda a célula selecionada para tipo markdown;
     2. Y: muda a célula para tipo código;
     3. A: cria uma célula acima da que está;
     4. B: cria uma célula abaixo da que está;
     5. SHIFT + ENTER: roda o código da clélula.
  2. Podemos criar um DataFrame a partir dos tipos de dados:



* + 1. Podemos passar outro parâmetro para o DataFrame fazendo ele dar nome às colunas:

pd.DataFrame(dados.dtypes, columns=['Tipos de Dados'])



* + 1. Outro jeito de fazer isso, mas na primeira coluna dessa vez:

tipos\_de\_dados.columns.name = 'Variáveis'



* 1. Extras:
     1. Aula extra de como importar dados de diversos tipos de extensão de arquivos. O primeiro foi json.
     2. Podemos abrir um arquivo e mandar printar a leitura dele para termos uma noção dos dados:



* + 1. Para importar os dados json é igual ao csv:



* + 1. Abertura e leitura de dados funcionam igual para todas as extensões:



* + 1. Para importar, usamos o table:



* + 1. No caso de arquivos excel não conseguimos ler, mas o import é assim:



* + 1. Pegar informação de html:



* + - 1. Note que ela nos devolve uma lista, para devolver bonitinho colocamos index 0:



* + 1. Caso não tenha salvo a pag web, basta passar o url dela:



* + - 1. Quando a página tem mais de uma tabela, recebemos todas elas no nosso DataFrame:



* + - 1. Nesse caso, para acessar cada uma das tabelas é só indexar de 0 (primeira) até a última delas, visualizando todas.
  1. O que aprendemos:
     1. Como importar a biblioteca (import pandas as pd);
     2. Como ler fontes de dados diferentes:
        1. Uma base CSV (pd.read\_csv(...));
        2. Uma base JSON (pd.read\_json(...));
        3. Uma base TXT (pd.read\_table(...));
        4. Um arquivo EXCEL (pd.read\_excel(...));
        5. Uma página HTML (pd.read\_html(...)).
     3. Vários métodos e atributos úteis de dataframes, como:
        1. info();
        2. head();
        3. dtypes;
        4. columns;
        5. shape.
     4. E sobre Jupyter, vimos como:
        1. Criar diferentes tipos de células dentro do Jupyter;
        2. Acessar a documentação;
        3. Como reexecutar todas as células.

1. **Aula 3 – Series e Index:**
   1. Drop\_duplicates(): Elimina todas as duplicatas de uma series/DataFrame mantendo o primeiro:





* + 1. Também possui o parâmetro inplace.
  1. Podemos mostrar todos os index de uma variável em uma lista:



* + 1. Se quisermos alterar e colocar em ordem crescente, podemos fazer com que o index da series receba o range do tamanho dela:



* 1. Extra: Criando DataFrames:
     1. Podemos criar series de maneira bem simples:



* + 1. Existe o parâmetro índice que podemos utilizar para colocar rótulos personalizados ao invés de numérico como default:



* + 1. Podemos usar a mesma estrutura acima para criar um dicionário:



* + 1. E com isso facilitar a criação do índice:



* + 1. Podemos fazer operações matemáticas com as series cujo elas alteram seus valores:



* 1. Extra 2: Concatenando DataFrames
     1. Aprendemos a criar DataFrames com listas, dicts, tuples e tudo mais.
     2. Nada que já não saibamos fazer, mas caso tenha alguma dúvida e/ou queira consultar: <http://localhost:8891/notebooks/Projetos%20Python/Curso%20Pandas/extras/Criando%20Estrutura%20de%20Dados.ipynb>.
     3. Podemos criar condições no index e mandar receber um valor diferente do que existe para substituir caso seja verdadeiro:

 

* + 1. Concat([dataframes]): Concatena listas de DataFrames a partir das colunas:



* + - 1. Se alterarmos o eixo pra 1 ele inverte:



* 1. O que aprendemos:
     1. Como selecionar uma variável do dataframe (por exemplo, dados['Tipo'] ou dados.Tipo);
     2. Que um dataframe é composto de vários Series;
     3. Como eliminar duplicatas (pelo método drop\_duplicates());
     4. Como redefinir o index de um dataframe e series (atributo index);
     5. Como concatenar dataframes (lembrando do axis);
     6. Como criar novos dataframes baseados em estruturas de dados Python (lista, dicionários ou tuples).

1. **Aula 4 – Filtrando Dados:**
   1. Df.isin(lista,...): Devolve uma series com verdadeiro ou falso de acordo com a lista passada:

import pandas as pd

dados = pd.read\_csv('/aluguel.csv', sep=';')

list(dados['Tipo'].drop\_duplicates())

residencial = ['Quitinete',

 'Casa',

 'Apartamento',

 'Casa de Condomínio',

 'Casa Comercial',

 'Casa de Vila']

dados['Tipo'].isin(residencial)



* + 1. Ou seja, se o tipo não residencial, considerado o que está na lista, ele retorna false, mas, se por acaso se enquadra, então retorna true.
    2. Para remover do nosso df tudo que não for residencial, usamos isso como base e ainda fazemos uma nova indexação:

selecao = dados['Tipo'].isin(residencial)

dados\_residencial = dados[selecao]

dados\_residencial.index = range(dados\_residencial.shape[0])

dados\_residencial



* 1. Aprendemos a importar com o método read. Para exportar vamos utilizar o método to.
     1. Da mesma forma que o read precisa da extensão do tipo de arquivo, o to também precisa, ou seja, se quiser exportar em csv, usamos to\_csv(‘local\_e\_nome\_do\_arquivo.extensão’, sep=’;’ ~~(se houver separador no arquivo original, caso contrário, não precisa)~~):

dados\_residencial.to\_csv('aluguel\_residencial.csv', sep=';')

* + 1. Caso esteja utilizando o colab do google (desisti do jupyter por ser horrível e travar meu pc, voltei pro colab), quando utilizar o comando acima ele não irá salvar na pasta em que o arquivo do colab está. Para garantir que ele vá para a pasta no drive que você espera que ele vá, entre na parte das pastas e upload de arquivos e monte o seu drive:



* + 1. Caso queira fazer a montagem da unidade via código, use isso:

from google.colab import drive

drive.mount('drive')

df.to\_csv('data.csv')

!mv data.csv "drive/MyDrive/..."

* + 1. Se você entende um pouco de bash, vai saber alterar de acordo com sua necessidade.
       1. !cp para copiar.
       2. !mv para mover.
       3. Se não encontrar o data.csv, tente \*.csv, irá pegar todos os arquivos que tenham essa extensão.
       4. Digite todo o caminho até a pasta que quer deixar o arquivo salvo.
    2. O problema de fazer essa exportação é que ela vem como padrão a exportação do índice junto, portanto, ao visualizar nosso df exportado, vemos isso:



* + - 1. 2 índices. Para concertar, exportamos passando como parâmetro index=False:

dados\_residencial.to\_csv('aluguel\_residencial.csv', sep=';', index=False)

* 1. Extra: Sort\_index()
     1. Com esse método podemos organizar de forma crescente ou alfabética o index tanto das linhas quanto colunas. Por padrão ele faz das linhas, mas podemos alterar o axis e fazer com as colunas também:

import pandas as pd

data = [[1,2,3], [4,5,6], [7,8,9]]

ln = list('321')

cl = list('ZYX')

df = pd.DataFrame(data, index = ln, columns = cl)

df



df.sort\_index(inplace=True)

df.sort\_index(inplace=True, axis=1)

df



* + 1. Sort\_values(by=’’, axis=…,inplace=…):
       1. Podemos organizer a sequência dos valores da mesma forma que os índex:



df.sort\_values(by = 'X', inplace = True)

df.sort\_values(by = '3', axis=1, inplace = True)

df



* + - 1. No final voltou a ser o que era no começo.
      2. Podemos passar listas no by:

df.sort\_values(by = ['X', 'Y'], inplace = True)



* 1. O que aprendemos:
     1. Criar uma Series booleana usando o método isin(..) a partir do dataframe;
     2. Filtrar os dados de um dataframe baseado na Series booleana;
     3. Exportar e gravar os dados do dataframe (método to\_csv());
     4. Ordenar os dados de um dataframe (métodos sort\_values() e sort\_index()).

1. **Aula 5 – Frequência de Imóveis:**
   1. Podemos simplificar a seleção de vários dados em uma variável simplesmente fazendo uma verificação bool:

selecao = dados['Tipo'] == 'Apartamento'

n1 = dados[selecao]

n1



* + 1. A partir de agora nossa n1 está com todos os dados do tipo apartamento nela. Sem precisar fazer nem um if!
    2. Também podemos fazer condições passando operadores lógicos, como o ou, que nesse caso é a ‘|’:

selecao = (dados['Tipo'] == 'Casa') | (dados['Tipo'] == 'Casa de Condomínio') | (dados['Tipo'] == 'Casa de Vila')

n2 = dados[selecao].shape[0]

n2



* + - 1. Como pedimos o shape[0] ele irá exibir somente a quantidade de linhas que nossos dados com esse índice possuí.
      2. Precisamos colcoar os () dentro de cada verificação, caso contrário ele dará um erro.
    1. Utilizando operador de E, nesse caso sendo o &:

selecao = (dados['Area'] >= 60) & (dados['Area'] <= 100)

n3 = dados[selecao].shape[0]

n3



* + 1. Operado &, mas com 2 variáveis:

selecao = (dados['Quartos'] >= 4) & (dados['Valor'] < 2000.0)

n4 = dados[selecao].shape[0]

n4



* 1. Extra: Formas de seleção:
     1. Split(): separa uma str criando uma lista com cada elemento. Por padrão faz a quebra em espaços, mas pode ser alterado para fazer a quebra com qualquer caracter:

'l1 l2 l3 l4'.split()



* + 1. Quando queremos selecionar mais de 1 coluna de uma vez colocamos [[]], como já vimos:

df[['c3', 'c1']]



* + 1. Para selecionar linhas, usamos números e ‘:’:

df[:]



df[1:3]



* + 1. Linhas e colunas:

df[1:][['c3', 'c1']]



* + - 1. Dessa maneira que estamos fazendo só conseguimos selecionar as linhas por números e não por rótulos como as colunas.
    1. Loc[]: Permite indexar as linhas por rótulo:

df.loc['l3']



* + - 1. Criou uma series pois usamos só 1 par de []. Se usar 2 ele cria um DataFrame:

df.loc[['l3']]



* + - 1. Passando mais de uma linha:

df.loc[['l3', 'l2']]



* + - 1. Ponto em comum: quando queremos pegar o valor do encontro entre uma linha e uma coluna:

df.loc['l1', 'c2']



* + 1. Iloc[]: Faz exatamente a mesma coisa que o loc, mas utilizando índices numéricos ao invés de rótulos:

df.iloc[0, 1]



* 1. O que aprendemos:
     1. Formas de seleção e frequências:
        1. Seleção com a condição OR (|);
        2. Seleção com a condição AND (&);
     2. Como criar um Index com split();
     3. Seleção por linha e coluna em um dataframe:
        1. Utilizando os índices numéricos e os rótulos das linhas.

1. **Aula 6 – Tratando de Dados Faltantes:**
   1. Df.isnull(): Retorna um DataFrame bool com dizem false para os dados que não sejam nulos e True para os dados que são nulos:

dados.isnull()



* + 1. Df.notnull(): Literalmente o oposto do acima:

dados.notnull()



* + 1. Se queremos receber todos os dados cujo variável valor é nula, fazemos a seguinte indexação:

dados[dados['Valor'].isnull()]

* + - 1. Dados quer selecionar somente as linhas “Valor” dos dados cujo isnull() retorne True:



* + 1. Df.dropna(subset=[‘nome\_da\_variável/coluna’], inplace=True): Elimina todos os dados que possuem valor nulo da coluna/variável passada. Inplace é pra salvar as alterações no df, se não ele serve apenas para exibição:

dados.dropna(subset=['Valor'], inplace=True)

dados[dados['Valor'].isnull()]



* + - 1. Retornou somente as colunas, pois já não existe nada com o Valor null.
  1. ~: inverte a seleção em um index:

selecao = (dados['Tipo']  == 'Apartamento') & (dados['Condominio'].isnull())

dados = dados[~selecao]

* + - 1. Queremos pegar somente o que NÃO é do tipo apartamento e tenham a variável condomínio nula, uma vez que não faz sentido que isso tenha acontecido já que todo apartamento possui condomínio.
      2. Nesse caso podemos fazer uma seleção de exatamente o que não queremos pegar e colocar um ‘~’ na hora de fazer a seleção dos dados.
      3. Com esse código eliminamos do nosso df todos os dados que eram apartamento e tinham o condomínio com valor nulo.
    1. Df.fillna(n, inplace=True): Preenche tudo que tiver valor nulo pelo valor passado em n.
       1. Podemos ainda usar o fillna com um dicionário, podendo passar valores diferentes para variáveis diferentes, caso necessário:

dados = dados.fillna({'Condominio': 0, 'IPTU': 0})

dados[dados['Condominio'].isnull()].shape[0]



* + - 1. Os dados que ainda possuíam condomínio com valor 0 eram apenas aqueles que não eram apartamento, ou seja, casas e afins.
      2. Nesse caso usamos o dicionário e passamos o valor 0 para ambos. Poderíamos ter simplificado apenas passando o n=0 e inplace=True, onde ele já preencheria tudo automaticamente. Contudo, fizemos desse jeito para exemplificar que podemos fazer com dicionários e passando valores diferentes.
      3. Agora o retorno de valores nulos no condomínio foi 0.
  1. Métodos de interpolação:
     1. Df.fillna(method=’’): O fillna possui vários métodos de preenchimento, não somente com valores que você determina.
        1. Ffill: Preenche todos os valores nulos com uma cópia do valor anterior a ele:





* + - 1. Bfill: A mesma coisa que o ffill, mas de baixo pra cima:



* + - 1. Poderíamos ainda passar como parâmetro o próprio df.mean(), assim seria preenchido com a média do df:



* + - 1. Podemos ainda passar o limit=n como parâmetro, limitando a quantidade de vezes que o método será executado.
  1. O que aprendemos:
     1. Tratamento de dados faltantes;
     2. Como identificar valores nulos (missing values):
        1. O método isnull() indica se os valores são nulos;
        2. O método notnull() retorna o contrário do método isnull();
        3. O método info() também é uma forma de se verificar a presença de valores faltantes.
     3. Como remover valores nulos com o método dropna();
     4. Tratamento condicional;
     5. Inversão de valores booleanos com ~;
     6. Como substituir os missing values com o método fillna();
     7. Métodos de interpolação: ffill, bfill e mean().

1. **Aula 7 – Novas Variáveis:**
   1. Para criar novas variáveis, ou seja colunas, simplesmente criamos um novo índex e atribuímos valor. Como qualquer outra lista normal:

dados['Valor Bruto'] = dados['Valor'] + dados['Condominio'] + dados['IPTU']



* + 1. Valor por m2:

dados['Valor m2'] = dados['Valor'] / dados['Area']



* + - 1. Note que ficaram com MUITAS casas decimais. Podemos concertar com o round(n), responsável por “arredondar” casas decimais:

dados['Valor m2'] = dados['Valor m2'].round(2)



* + - 1. Podemos inclusive passar isso direto na atribuição da variável:

dados['Valor Bruto m2'] = (dados['Valor Bruto'] / dados['Area']).round(2)



* + 1. Para juntar 3 tipos de imóveis em uma única variável utilizamos o apply(condição):

casa = ['Casa', 'Casa de Condomínio', 'Casa de Vila']

dados['Tipo Agregado'] = dados['Tipo'].apply(lambda x: 'Casa' if x in casa else 'Apartamento')



* + - 1. Basicamente criamos uma nova variável que está recebendo todos os tipos de imóveis e separando em tipo ‘Casa’ se x está na nossa lista, se não, entra na classificação ‘Apartamento’.
  1. Podemos criar um DataFrame a partir de outro DataFrame, mas pegando somente algumas variáveis dela, para isso, basta passar o índice delas:

dados\_aux = pd.DataFrame(dados[['Tipo Agregado', 'Valor m2', 'Valor Bruto', 'Valor Bruto m2']])

dados\_aux.head(10)



* + 1. Para excluir uma variável, basta usar o del e passar o índice dela do df:

del dados\_aux['Valor Bruto']

dados\_aux.head(10)



* + 1. Também podemos fazer isso utilizando o pop(variável):

dados\_aux.pop('Valor Bruto m2')

dados\_aux.head()



* + 1. Apesar de tudo, a melhor forma é utilizando o drop([lista\_variáveis]), onde passamos uma lista de variáveis que queremos deletar e ele elimina:
       1. O drop funciona tanto nas linhas quanto colunas e, como queremos eliminar as colunas, precisamos passar o eixo 1:

dados.drop(['Valor Bruto', 'Valor Bruto m2'], axis = 1, inplace=True)

dados.head(10)



* 1. Aula extra: Contadores
     1. Df.unique(): Retorna um Array com todos os elementos que um Series possuí sem repetições:

s = pd.Series(list('asdasfasfdasfasdasdasdasdas'))

s.unique()



* + 1. Df.value\_counts(): Mostra a contagem, quantas vezes cada elemento apareceu em um series:

s.value\_counts()



* + 1. Podemos utilizar isso no tipo dos nossos imóveis, por exemplo:

dados.Tipo.unique()



dados.Tipo.value\_counts()



* 1. O que aprendemos:
     1. Como criar variáveis;
     2. Como excluir variáveis utilizando del, pop() e drop();
     3. Contadores, através do value\_counts().

1. **Aula 8 – Estatísticas Descritiva:**
   1. S.mean(): Retorna a média de uma series:

dados['Valor'].mean()



* + 1. Df.groupby(‘variável’): agrupa todos os dados, ou seja, se tivermos 7 imóveis diferentes espalhados pela nossa planilha, ele agrupa em um único bairro em comum entre eles.
    2. Df.groups: atributo que exibe os grupos feitos acima em um dict, mostrando o índex do imóvel como no exemplo do primeiro item da barra da tijuca:

bairros = ['Barra da Tijuca', 'Copacabana', 'Ipanema', 'Leblon', 'Botafogo', 'Flamengo', 'Tijuca']

selecao = dados['Bairro'].isin(bairros)

dados = dados[selecao]

grupo\_bairro = dados.groupby('Bairro')

grupo\_bairro.groups

 

* + 1. Podemos fazer um for para entender melhor essa variável grupo\_bairro:

for bairro, dados in grupo\_bairro:

  print(bairro)



for bairro, dados in grupo\_bairro:

  print(dados)



* + - 1. Podemos ver que ela é divida entre os bairros que escolhemos e suas respectivas variáveis, como tipo, bairro, valor.... Considerando isso podemos fazer o seguinte código para tirar a média do valor de cada bairro:

for bairro, dados in grupo\_bairro:

  print(f'{bairro} -> {dados.Valor.mean()}')



* + - 1. De uma maneira muito mais simples e rápida, podemos fazer exatamente a mesma coisa apenas passando a média da variável valor do nosso grupo:

grupo\_bairro.Valor.mean()



* + - 1. Podemos ainda fazer com vários valores:

grupo\_bairro[['Valor', 'Condominio']].mean().round(2)



* 1. Utilizando nosso grupo, podemos fazer um df.describe() para retornar todas as estatísticas descritivas, tais como média, desvio padrão, quartis e valores mínimos e máximos:

grupo\_bairro['Valor'].describe().round(2)



* + 1. Fazer esse tipo de análise é super interessante pois conseguimos notar várias coisas que podem estar erradas, como no caso do bairro Botafogo, onde temos um desvio padrão muito alto. Ao ver o valor máximo, podemos notar qual é o maior valor que um ou mais imóveis estão disponíveis para locação, o que acaba sendo impossível. Não tem como um imóvel custar 45000000 para locação. Alguém pode ter misturado os dados de venda com locação.
    2. Df.aggregate([‘lista de estetística’]): Permite passarmos uma lista de testes estatísticos que desejamos ao invés de receber tudo aquilo:

grupo\_bairro['Valor'].aggregate(['min', 'max', 'sum'])



* + - 1. Podemos usar a forma simplificada df.agg().
    1. Documentação do pandas para mais testes estatísticos: <https://pandas.pydata.org/pandas-docs/stable/reference/index.html>.
    2. Df.rename(columns={‘name’: ‘other\_name’): Renomeia colunas pelos valores referentes às chaves:

grupo\_bairro['Valor'].aggregate(['min', 'max']).rename(columns={'min': 'Mínimo', 'max': 'Máximo'})



* 1. Caso tenhamos uma quantidade muito grande de dados fica impossível analisar todos, por isso podemos utilizar a matplolib, uma biblioteca de gráficos que nos ajuda na visualização:

%matplotlib inline

import matplotlib.pyplot as plt

plt.rc('figure', figsize = (20,10))

grupo\_bairro['Valor'].std().plot.bar(color='blue')



* + - 1. A primeira linha diz para ele mostrar tudo em linha.
      2. Na segunda é a importação da lib.
      3. A terceira é a qual configuramos o tamanho da imagem.
      4. E a quarta é a plotagem do gráfico em formato de barras a partir do desvio padrão da variável valor dos bairros do nosso grupo.
    1. Podemos determinar o fontsize das coisas, como no título:

fig.set\_title('Valor Médio do Aluguel por bairro', {'fontsize':22})



* 1. Extra) Criando faixas de valor:
     1. Para criar uma faixa de intervalos de valor podemos simplesmente criar uma lista com o intervalo que queremos e usar o pd.cut(df.variável, intervar), que ele cria automaticamente:

classes = [0, 2, 4, 6, 100]

quartos = pd.cut(dados.Quartos, classes)

quartos



* + - 1. Desse jeito fica um pouco confuso de visualizar, mas podemos colocar um pd.values\_count(df) para simplificar:

pd.value\_counts(quartos)



* + - 1. Os ‘(‘ significa que o valor seguinte não entra, ou seja, o primeiro ele conta de 1 a 2, o segundo de 3 a 4, o terceiro de 5 a 6 e o último de 7 a 100.
      2. Para deixar ainda mais claro podemos criar labels e passar como parâmetro para o values\_count():

labels = ['1 e 2 quartos', '3 e 4 quartos', '5 e 6 quartos', '7 quartos ou mais']

quartos = pd.cut(dados.Quartos, classes, labels= labels)

pd.value\_counts(quartos)



* + - 1. Para ter certeza que não tem imóveis com 0 quartos, podemos ainda passar o include\_lowest= True, dessa forma ele vai incluir o 0 que passamos na nossa lista classes:

quartos = pd.cut(dados.Quartos, classes, labels= labels, include\_lowest= True)

pd.value\_counts(quartos)



* + - 1. Note que a quantidade do primeiro aumentou... Mudei a label, por isso tá mostrando certo.
  1. O que aprendemos:
     1. Como criar agrupamentos com o groupby();
     2. Estatísticas descritivas com o describe() e o aggregate();
     3. Como renomear as colunas com o rename();
     4. Como fazer gráficos com o pacote Matplotlib;
     5. Como criar faixas de valor com o cut().

1. **Aula 9 – Removendo Outliers:**
   1. Df.boxplot([‘variavel’]): cria um gráfico de dados a partir da mediana.:

dados.boxplot(['Valor'])



* + 1. Não conseguimos ver nada pois temos um outlier muito alto.
    2. S.quantile(n): faz o quartil de uma series. N depende de qual quartil você quer utilizar:

valor = dados['Valor']

q1 = valor.quantile(.25)

q1



* + 1. O intervalo interquartílico (IIQ) é o q3 menos o q1:

q1 = valor.quantile(.25)

q3 = valor.quantile(.75)

iiq = q3 - q1

* + 1. Já os limites inferior e superior são, respectivamente, q1 - 1.5 \* iiq e q3 + 1.5 \* iiq:

limite\_inferior = q1 - 1.5 \* iiq

limite\_superior = q3 + 1.5 \* iiq

* + 1. Fazemos essas contas pois queremos eliminar quem está fora do limite inferior (li) e superior (ls), pois eles são os outliers do grupo.
    2. Após essas contas, criamos uma nova seleção de dados que pega tudo que for menor que o ls e maior que o li, os que estiverem fora disso são eliminados:

selecao = (valor >= limite\_inferior) & (valor <= limite\_superior)

dados\_new = dados[selecao]

dados\_new.boxplot(['Valor'])



* + 1. Podemos ver que ainda existem dados fora dessa seleção, mas aí o ideal é fazer com calma e analisar mais detalhadamente.
    2. Para fazer uma comparação podemos criar um histograma de antes e depois dos dados:

dados.hist(['Valor'])

dados\_new.hist(['Valor'])



* + - 1. Devido ao outlier, não conseguimos ver nada no gráfico 1, já no 2, após os out, conseguimos ver uma distribuição melhor.
  1. Podemos plotar um boxplot usando uma variável por outra, passando como argumento o by:

dados.boxplot(['Valor'], by = ['Tipo'])



* + 1. Podemos passer um índex para o df.groupby()[], criando uma series:

grupo\_tipo = dados.groupby('Tipo')['Valor']

* + - 1. Dessa forma temos um grupo separado por tipo, mas somente da variável valor.
    1. Ao fazer os quartis com grupos, eles se tornam series:

q1 = grupo\_tipo.quantile(.25)

q3 = grupo\_tipo.quantile(.75)

iiq = q3 - q1

limite\_inferior = q1 - 1.5 \* iiq

limite\_superior = q3 + 1.5 \* iiq

q1



* + - 1. Para acessar qualquer um deles, sistema básico de indexação.
    1. Para fazer uma seleção para cada elemento:

dados\_new = pd.DataFrame()

for tipo in grupo\_tipo.groups.keys():

  eh\_tipo = dados['Tipo'] == tipo

  eh\_dentro\_limite = (dados.Valor >= limite\_inferior[tipo]) & (dados.Valor <= limite\_superior[tipo])

  selecao = eh\_tipo & eh\_dentro\_limite

  dados\_selecao = dados[selecao]

  dados\_new = pd.concat([dados\_new, dados\_selecao])

dados\_new.boxplot(['Valor'], by = ['Tipo'])



* + - 1. Podemos notar que o resultado é bem diferente da imagem anterior.