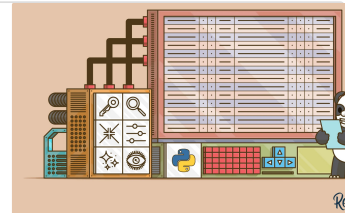


Explorando dados com Pandas

Using Pandas and Python to Explore Your Dataset - Real Python

In this step-by-step tutorial, you'll learn how to start exploring a dataset with Pandas and Python. You'll learn how to access specific rows and columns to answer questions about your data. You'll also see

<https://realpython.com/pandas-python-explore-dataset/>



Você tem um grande conjunto de dados cheio de insights interessantes, mas você não tem certeza por onde começar a explorá-lo? Seu chefe pediu para você gerar algumas estatísticas a partir dele, mas eles não são tão fáceis de extrair? Estes são precisamente os casos de uso onde **Pandas** e Python podem ajudá-lo!

Com essas ferramentas, você será capaz de cortar um grande conjunto de dados em partes gerenciáveis e obter insights dessas informações.

Neste tutorial, você aprenderá como:

- **Calcule** métricas sobre seus dados
- **Realizar** consultas básicas e agregações
- **Descubra** e manuseie dados incorretos, inconsistências e valores perdidos
- **Visualize** seus dados com plots

Explorar dados com Pandas

Você também aprenderá sobre as diferenças entre as principais estruturas de dados que Pandas e Python usam. Para acompanhar, você pode obter todo o código de exemplo neste tutorial no link abaixo:

realpython/materials

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or window. Reload to refresh your session. Reload to refresh your

<https://github.com/realpython/materials/tree/master/pandas-intro>



Configuração do ambiente

Há algumas coisas que você precisa começar com este tutorial. Primeiro é uma familiaridade com as estruturas de dados incorporadas do Python,

especialmente listas e dicionários. Para obter mais informações, confira Listas e Tuplas em Python e Dicionários em Python.

A segunda coisa que você vai precisar é de um ambiente Python funcionando. Você pode acompanhar em qualquer terminal que tenha Python 3 instalado. Se você quiser ver uma saída mais agradável, especialmente para o grande conjunto de dados da NBA com o qual você estará trabalhando, então você pode querer executar os exemplos em um notebook Jupyter.

A última coisa que você vai precisar é da biblioteca Pandas Python, que você pode instalar com pip:

```
python -m pip install pandas
```

Você também pode usar o gerenciador de pacotes Conda:

```
conda install pandas
```

Se você está usando a distribuição Anaconda, então você está pronto para ir! A Anaconda já vem com a biblioteca Pandas Python instalada.

Usando a Biblioteca Pandas Python

Agora que você instalou pandas, é hora de dar uma olhada em um conjunto de dados. Neste tutorial, você analisará os resultados da NBA fornecidos pelo FiveThirtyEight em um arquivo CSV de 17MB. Crie um script para baixar os dados: `download_nba_all_elo.py`

```
import requests

download_url = "https://raw.githubusercontent.com/fivethirtyeight/data/master/nba-elo/nbaallelo.csv"
target_csv_path = "nba_all_elo.csv"

response = requests.get(download_url)
response.raise_for_status() # Check that the request was successful
with open(target_csv_path, "wb") as f:
    f.write(response.content)
print("Download ready.")
```

Quando você executar o script, ele salvará o arquivo em seu diretório de trabalho atual. `nba_all_elo.csv`

NOTA

Você também pode usar seu navegador da Web para baixar o arquivo CSV.

No entanto, ter um script de download tem várias vantagens:

- Você pode dizer onde você conseguiu seus dados.
- Você pode repetir o download a qualquer hora! Isso é especialmente útil se os dados forem frequentemente atualizados.
- Você não precisa compartilhar o arquivo CSV de 17MB com seus colegas de trabalho. Normalmente, é o suficiente para compartilhar o script de download.

Agora você pode usar a biblioteca Pandas Python para dar uma olhada em seus dados:

```
>>> import pandas as pd
>>> nba = pd.read_csv("nba_all_elo.csv")
>>> type(nba)
<class 'pandas.core.frame.DataFrame'>
```

Aqui, você segue a convenção de importar Pandas em Python com o pseudônimo. Em seguida, você usa para ler em seu conjunto de dados e armazená-lo como um objeto DataFrame na variável `.pd.read_csv(nba)`

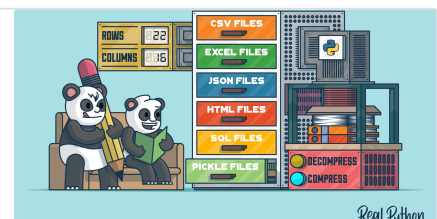
NOTA: Dados não estão no formato CSV

Seus dados não estão no formato CSV? Não se preocupe! A biblioteca Pandas Python fornece várias funções semelhantes. Para saber como trabalhar com esses formatos de arquivo, confira Lendo e Escrevendo Arquivos com Pandas:

Pandas: How to Read and Write Files - Real Python


In this tutorial, you'll learn about the Pandas IO tools API and how you can use it to read and write files. You'll use the Pandas `read_csv()` function to work with CSV files. You'll also

 <https://realpython.com/pandas-read-write-files/>



Input/output - pandas 1.2.0 documentation

Load pickled pandas object (or any object) from file.

 <https://pandas.pydata.org/pandas-docs/stable/reference/io.html>

Você pode ver a quantidade de dados que contém:nba

```
>>> len(nba)
126314
>>> nba.shape
(126314, 23)
```

Você usa a função incorporada python para determinar o número de linhas. Você também usa o atributo do para ver sua **dimensionalidade**. O resultado é uma tupla contendo o número de linhas e colunas. `len()` `.shape` `DataFrame`

Agora você sabe que há 126.314 linhas e 23 colunas em seu conjunto de dados. Mas como você pode ter certeza que o conjunto de dados realmente contém estatísticas de basquete? Você pode dar uma olhada nas cinco primeiras linhas com: `.head()`

```
>>> nba.head()
```

gameorder	game_id	lg_id	iscopy	year_id	date_game	seasongame	is_playoffs	team_id	fran_id	...	win_equiv	opp_id	opp_fran	opp_pts	<
0	1	194611010TRH	NBA	0	1947	11/1/1946	1	0	TRH Huskies	...	40.294830	NYK	Knicks	68	1
1	1	194611010TRH	NBA	1	1947	11/1/1946	1	0	NYK Knicks	...	41.705170	TRH	Huskies	66	1
2	2	194611020CHS	NBA	0	1947	11/2/1946	1	0	CHS Stags	...	42.012257	NYK	Knicks	47	1
3	2	194611020CHS	NBA	1	1947	11/2/1946	2	0	NYK Knicks	...	40.692783	CHS	Stags	63	1
4	3	194611020DTF	NBA	0	1947	11/2/1946	1	0	DTF Falcons	...	38.864048	WSC	Capitols	50	1

5 rows x 23 columns

Limitar número de colunas

A menos que sua tela seja bastante grande, sua saída provavelmente não exibirá todas as 23 colunas. Em algum lugar no meio, você verá uma coluna de elipses () indicando os dados faltantes. Se você está trabalhando em um terminal, então isso é provavelmente mais legível do que embrulhar longas filas. No entanto, os notebooks Jupyter permitirão que você role. Você pode configurar Pandas para exibir todas as 23 colunas como esta:

```
>>> pd.set_option("display.max.columns", None)
```

Casas Decimais

Embora seja prático ver todas as colunas, você provavelmente não precisará de seis casas decimais! Mude para dois:

```
>>> pd.set_option("display.precision", 2)
```

Para verificar se você alterou as opções com sucesso, você pode executar novamente ou você pode exibir as últimas cinco linhas com em

vez disso: `.head().tail()`

```
>>> nba.tail()
```

Agora, você deve ver todas as colunas, e seus dados devem mostrar dois lugares decimais:

	gameorder	game_id	lg_id	iscopy	year_id	date_game	seasongame	is_playoffs	team_id	fran_id	pts	elo_i	elo_n	win_equiv	op
126309	63155	201506110	CLE	NBA	0	2015	6/11/2015	100	1	CLE	Cavaliers	82	1723.41	1704.39	60.31
126310	63156	201506140	GSW	NBA	0	2015	6/14/2015	102	1	GSW	Warriors	104	1809.98	1813.63	68.01
126311	63156	201506140	GSW	NBA	1	2015	6/14/2015	101	1	CLE	Cavaliers	91	1704.39	1700.74	60.01
126312	63157	201506170	CLE	NBA	0	2015	6/16/2015	102	1	CLE	Cavaliers	97	1700.74	1692.09	59.29
126313	63157	201506170	CLE	NBA	1	2015	6/16/2015	103	1	GSW	Warriors	105	1813.63	1822.29	68.52

Conhecendo seus dados

Você importou um arquivo CSV com a biblioteca Pandas Python e deu uma primeira olhada no conteúdo do seu conjunto de dados. Até agora, você só viu o tamanho do seu conjunto de dados e suas primeiras e últimas linhas. Em seguida, **você aprenderá como examinar seus dados de forma mais sistemática.**

Informações sobre os dados

Você pode exibir todas as colunas e seus tipos de dados com: `.info()`

```
>>> nba.info()
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 126314 entries, 0 to 126313
Data columns (total 23 columns):
gameorder      126314 non-null int64
game_id        126314 non-null object
lg_id          126314 non-null object
_iscopy        126314 non-null int64
year_id        126314 non-null int64
date_game      126314 non-null object
seasongame     126314 non-null int64
is_playoffs    126314 non-null int64
team_id        126314 non-null object
fran_id        126314 non-null object
pts            126314 non-null int64
elo_i          126314 non-null float64
elo_n          126314 non-null float64
win_equiv      126314 non-null float64
opp_id         126314 non-null object
opp_fran       126314 non-null object
opp_pts        126314 non-null int64
opp_elo_i      126314 non-null float64
opp_elo_n      126314 non-null float64
game_location  126314 non-null object
game_result    126314 non-null object
forecast       126314 non-null float64
notes          5424 non-null object
dtypes: float64(6), int64(7), object(10)
memory usage: 22.2+ MB

```

Você verá uma lista de todas as colunas do seu conjunto de dados e o tipo de dados que cada coluna contém. Aqui, você pode ver os tipos de dados. Pandas usa a biblioteca [NumPy](#) para trabalhar com esses tipos.

Mostrando estatísticas básicas

Agora que você viu quais tipos de dados estão em seu conjunto de dados, é hora de obter uma visão geral dos valores que cada coluna contém.

Você pode fazer isso com: `.describe()`

```
nba.describe()
```

Esta função mostra algumas estatísticas básicas descritivas para todas as colunas numéricas:

	gameorder	iscopy	year_id	seasongame	is_playoffs	pts	elo_i	elo_n	win_equiv	opp_pts	opp_elo_i	opp_elo_n	forecast
count	126314.00	126314.0	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00
mean	31579.00	0.5	1988.20	43.53	0.06	102.73	1495.24	1495.24	41.71	102.73	1495.24	1495.24	0.50
std	18231.93	0.5	17.58	25.38	0.24	14.81	112.14	112.46	10.63	14.81	112.14	112.46	0.22
min	1.00	0.0	1947.00	1.00	0.00	0.00	1091.64	1085.77	10.15	0.00	1091.64	1085.77	0.02
25%	15790.00	0.0	1975.00	22.00	0.00	93.00	1417.24	1416.99	34.10	93.00	1417.24	1416.99	0.33
50%	31579.00	0.5	1990.00	43.00	0.00	103.00	1500.95	1500.95	42.11	103.00	1500.95	1500.95	0.50
75%	47368.00	1.0	2003.00	65.00	0.00	112.00	1576.06	1576.29	49.64	112.00	1576.06	1576.29	0.67
max	63157.00	1.0	2015.00	108.00	1.00	186.00	1853.10	1853.10	71.11	186.00	1853.10	1853.10	0.98

`.describe()` apenas analisa colunas numéricas por padrão, mas você pode fornecer outros tipos de dados se você usar o parâmetro: `include`

```
>>> import numpy as np
>>> nba.describe(include=np.object)
```

`.describe()` não tentará calcular uma média ou um desvio padrão para as colunas, uma vez que elas incluem principalmente strings de texto. No entanto, ele ainda exibirá algumas estatísticas descritivas: `object`

	game_id	lg_id	date_game	team_id	fran_id	opp_id	opp_fran	game_location	game_result	notes
count	126314	126314	126314	126314	126314	126314	126314	126314	126314	5424
unique	63157	2	12426	104	53	104	53	3	2	231
top	198801240POR	NBA	4/16/2014	BOS	Lakers	BOS	Lakers	H	W	at New York NY
freq	2	118016	30	5997	6024	5997	6024	63138	63157	440

Dê uma olhada nas colunas. Seu conjunto de dados contém 104 IDs de equipe diferentes, mas apenas 53 diferentes IDs de franquia. Além disso, o ID da equipe mais frequente é, mas o ID de franquia mais frequente. Como isso é possível? Você precisará explorar seu conjunto de dados um pouco mais para responder a esta pergunta. `team_idfran_idBOSLakers`

Explorando seu conjunto de dados

A **análise exploratória de dados** pode ajudá-lo a responder perguntas sobre seu conjunto de dados. Por exemplo, você pode examinar com que frequência valores específicos ocorrem em uma coluna:

```
>>> nba["team_id"].value_counts()
BOS    5997
NYK    5769
LAL    5078
...
SDS      11
>>> nba["fran_id"].value_counts()
Name: team_id, Length: 104, dtype: int64
Lakers    6024
Celtics    5997
Knicks    5769
...
```

```
Huskies          60
Name: fran_id, dtype: int64
```

Parece que um time chamado jogou 6024 jogos, mas apenas 5078 deles foram jogados pelo Los Angeles Lakers. Descubra quem é a outra equipe: "Lakers" "Lakers"

```
>>> nba.loc[nba["fran_id"] == "Lakers", "team_id"].value_counts()
LAL      5078
MNL       946
Name: team_id, dtype: int64
```

De fato, o Minneapolis Lakers jogou 946 jogos. Você pode até descobrir quando eles jogaram esses jogos: "MNL"

```
>>> nba.loc[nba["team_id"] == "MNL", "date_game"].min()
'1/1/1949'
>>> nba.loc[nba["team_id"] == "MNL", "date_game"].max()
'4/9/1959'
>>> nba.loc[nba["team_id"] == "MNL", "date_game"].agg(("min", "max"))
min      1/1/1949
max      4/9/1959
Name: date_game, dtype: object
```

Parece que o Minneapolis Lakers jogou entre os anos de 1949 e 1959. Isso explica por que você pode não reconhecer esta equipe!

Você também descobriu por que o time do Boston Celtics jogou mais jogos no conjunto de dados. Vamos analisar um pouco a história deles. Descubra quantos pontos o Boston Celtics marcou durante todas as partidas contidas neste conjunto de dados. Expanda o bloco de código abaixo para a solução: "BOS"

```
>>> nba.loc[nba["team_id"] == "BOS", "pts"].sum()
626484
```

O Boston Celtics marcou um total de 626.484 pontos.

Conhecendo as estruturas de dados da Pandas

Embora um forneça funções que podem parecer bastante intuitivas, os conceitos subjacentes são um pouco mais complicados de entender. Por esta razão, você vai deixar de lado a vasta NBA e construir alguns objetos menores pandas do zero. `DataFrame DataFrame`

Entendendo objetos de série

A estrutura de dados mais básica do Python é a lista, que também é um bom ponto de partida para conhecer `pandas. Objetos de série`. Crie um novo objeto com base em uma lista: `Series`

```
>>> revenues = pd.Series([5555, 7000, 1980])
>>> revenues
0    5555
1    7000
2     1980
dtype: int64
```

Você usou a lista para criar um objeto chamado `. Um objeto envolve dois componentes: [5555, 7000, 1980] Series revenues Series`

1. Uma sequência de **valores**
2. Uma sequência de **identificadores**, que é o índice

Você pode acessar esses componentes com `e`, respectivamente: `.values` `.index`

```
>>> revenues.values
array([5555, 7000, 1980])

>>> revenues.index
RangeIndex(start=0, stop=3, step=1)
```

Enquanto `pandas` se baseiam em `NumPy`, uma diferença significativa está em sua **indexação**. Assim como uma matriz `NumPy`, um `Pandas` também tem um índice inteiro que é implicitamente definido. Este índice implícito indica a posição do elemento no `. Series Series`

No entanto, um também pode ter um tipo arbitrário de índice. Você pode pensar neste índice explícito como rótulos para uma linha específica: `Series`

```
>>> city_revenues = pd.Series(
...     [4200, 8000, 6500],
...     index=["Amsterdam", "Toronto", "Tokyo"]
... )
>>> city_revenues
Amsterdam    4200
Toronto      8000
Tokyo        6500
dtype: int64
```

Aqui, o índice é uma lista de nomes de cidades representados por cordas. Você deve ter notado que os dicionários Python também usam

índices de string, e esta é uma analogia útil para ter em mente! Você pode usar os blocos de código acima para distinguir entre dois tipos de: `Series`

1. `receitas`: Isso se comporta como uma lista Python porque tem apenas um índice posicional. `Series`
2. `city_revenues`: Isso age como um dicionário Python porque possui um índice posicional e um índice de rótulo. `Series`

Veja como construir um com um índice de rótulos a partir de um dicionário Python: `Series`

```
>>> city_employee_count = pd.Series({"Amsterdam": 5, "Tokyo": 8})
>>> city_employee_count
Amsterdam    5
Tokyo        8
dtype: int64
```

As chaves do dicionário se tornam o índice, e os valores do dicionário são os valores. `Series`

Assim como dicionários, também suporte e a palavra-chave: `Series.keys() in`

```
>>> city_employee_count.keys()
Index(['Amsterdam', 'Tokyo'], dtype='object')

>>> "Tokyo" in city_employee_count
True

>>> "New York" in city_employee_count
False
```

Você pode usar esses métodos para responder perguntas sobre seu conjunto de dados rapidamente.

Entendendo objetos do DataFrame

Embora a seja uma estrutura de dados muito poderosa, ela tem suas limitações. Por exemplo, você só pode armazenar um atributo por tecla. Como você já viu com o conjunto de dados, que possui 23 colunas, a biblioteca Pandas Python tem mais a oferecer com seu `DataFrame`. Esta estrutura de dados é uma sequência de objetos que compartilham o mesmo índice. `Series nba Series`

Se você acompanhou os exemplos, então você já deve ter dois objetos com cidades como chaves: `Series Series`

1. `city_revenues`
2. `city_employee_count`

Você pode combinar esses objetos em um fornecendo um dicionário no construtor. As teclas do dicionário se tornarão os nomes das colunas e os valores devem conter os objetos: `DataFrame` `Series`

```
>>> city_data = pd.DataFrame({
...     "revenue": city_revenues,
...     "employee_count": city_employee_count
... })
>>> city_data
```

	revenue	employee_count
Amsterdam	4200	5.0
Tokyo	6500	8.0
Toronto	8000	NaN

Note como pandas substituiu o valor perdido para Toronto por

```
. employee_count NaN
```

O novo índice é a união dos dois índices: `DataFrame` `Series`

```
>>> city_data.index
Index(['Amsterdam', 'Tokyo', 'Toronto'], dtype='object')
```

Armazena seus valores em uma matriz NumPy: `Series` `DataFrame`

```
>>> city_data.values
array([[4.2e+03, 5.0e+00],
       [6.5e+03, 8.0e+00],
       [8.0e+03, nan]])
```

Você também pode se referir às 2 dimensões de um como eixos: `DataFrame`

```
>>> city_data.axes
[Index(['Amsterdam', 'Tokyo', 'Toronto'], dtype='object'),
 Index(['revenue', 'employee_count'], dtype='object')]
>>> city_data.axes[0]
Index(['Amsterdam', 'Tokyo', 'Toronto'], dtype='object')
>>> city_data.axes[1]
Index(['revenue', 'employee_count'], dtype='object')
```

O eixo marcado com 0 é o índice de **linha**, e o eixo marcado com 1 é o índice da **coluna**. Esta terminologia é importante saber porque você encontrará vários métodos que aceitam um parâmetro. `DataFrame` `axis`

A também é uma estrutura de dados semelhante a um dicionário, por isso também suporta a palavra-chave. No entanto, para estes não se relacionam com o índice, mas com as colunas: `DataFrame` `.keys()` `in DataFrame`

```
>>> city_data.keys()
Index(['revenue', 'employee_count'], dtype='object')
```

```
>>> "Amsterdam" in city_data
False

>>> "revenue" in city_data
True
```

Você pode ver esses conceitos em ação com o maior conjunto de dados da NBA. Contém uma coluna chamada, ou foi chamada? Para responder a esta pergunta, exiba o índice e os eixos do conjunto de dados e expanda o bloco de código abaixo para a solução:"points""pts"nba

Como você não especificou uma coluna de índice quando leu no arquivo CSV, pandas atribuiu um ao:RangeIndexDataFrame

```
>>> nba.index
RangeIndex(start=0, stop=126314, step=1)
```

nba, como todos os objetos, tem dois eixos:DataFrame

```
>>> nba.axes
[RangeIndex(start=0, stop=126314, step=1),
 Index(['gameorder', 'game_id', 'lg_id', '_iscopy', 'year_id', 'date_game',
       'seasongame', 'is_playoffs', 'team_id', 'fran_id', 'pts', 'elo_i',
       'elo_n', 'win_equiv', 'opp_id', 'opp_fran', 'opp_pts', 'opp_elo_i',
       'opp_elo_n', 'game_location', 'game_result', 'forecast', 'notes'],
       dtype='object')]
```

Você pode verificar a existência de uma coluna com: .keys()

```
>>> "points" in nba.keys()
False

>>> "pts" in nba.keys()
True
```

A coluna é chamada, não."pts""points"

Ao usar esses métodos para responder perguntas sobre seu conjunto de dados, não deixe de ter em mente se você está trabalhando com um ou um para que sua interpretação seja precisa.SeriesDataFrame

Acessando elementos da série

Na seção acima, você criou um Pandas baseado em uma lista Python e comparou as duas estruturas de dados. Você viu como um objeto é semelhante a listas e dicionários de várias maneiras. Uma outra semelhança é que você pode usar o **operador de indexação** () para também. `Series Series [] Series`

Você também aprenderá a usar dois métodos de acesso específicos do **Pandas**:

1. `.loc`
2. `.iloc`

Você verá que esses métodos de acesso a dados podem ser muito mais legíveis do que o operador de indexação.

Utilizando o Operador de Indexação

Lembre-se que um tem dois índices: `Series`

1. Um **índice posicional ou implícito**, que é sempre um `RangeIndex`
2. Um **rótulo ou índice explícito**, que pode conter quaisquer objetos hashable

Em seguida, revise o objeto: `city_revenues`

```
>>> city_revenues
Amsterdam    4200
Toronto      8000
Tokyo        6500
dtype: int64
```

Você pode acessar convenientemente os valores em um com os índices de rótulo e posicionais: `Series`

```
>>> city_revenues["Toronto"]
8000
>>> city_revenues[1]
8000
```

Você também pode usar índices e fatias negativas, assim como faria com uma lista:

```
>>> city_revenues[-1]
6500

>>> city_revenues[1:]
Toronto    8000
Tokyo      6500
dtype: int64

>>> city_revenues["Toronto":]
Toronto    8000
Tokyo      6500
dtype: int64
```

Usando e `.loc` `.iloc`

O operador de indexação é conveniente, mas há uma ressalva. E se os rótulos também forem números? Digamos que você tem que trabalhar com um objeto como este: `[] Series`

```
>>> colors = pd.Series(
...     ["red", "purple", "blue", "green", "yellow"],
...     index=[1, 2, 3, 5, 8]
... )
>>> colors
1      red
2    purple
3      blue
5     green
8     yellow
dtype: object
```

O que vai voltar? Para um índice posicional, é `.`. No entanto, se você passar pelo índice de rótulo, então está se referindo a

```
. colors[1] colors[1] "purple" colors[1] "red"
```

A boa notícia é que você não precisa descobrir! Em vez disso, para evitar confusão, a biblioteca Pandas Python fornece dois **métodos de acesso a dados**:

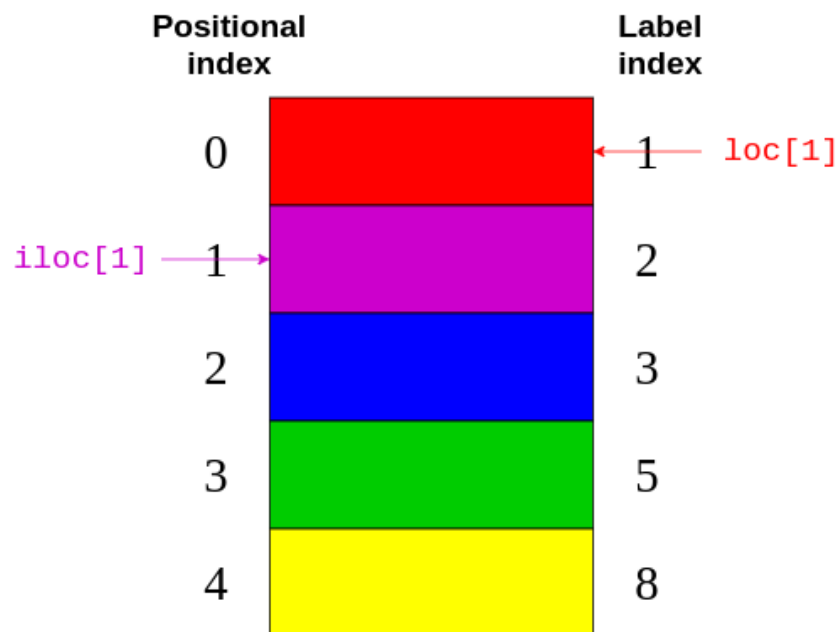
1. `.loc` refere-se ao índice de **rótulos**.
2. `.iloc` refere-se ao **índice posicional**.

Esses métodos de acesso a dados são muito mais legíveis:

```
>>> colors.loc[1]
'red'

>>> colors.iloc[1]
'purple'
```

A figura a seguir mostra quais elementos se referem: `.loc .iloc`



Novamente, aponta para o índice de rótulo no lado direito da imagem. Enquanto isso, aponta para o índice posicional no lado esquerdo da imagem. `.loc .iloc`

É mais fácil ter em mente a distinção entre `.loc` e `.iloc` do que descobrir o que o operador de indexação vai retornar. Mesmo que você esteja familiarizado com todas as peculiaridades do operador de indexação, pode ser perigoso assumir que todos que lêem seu código internalizaram essas regras também! `.loc .iloc`

`.loc` também suporta os recursos que você esperaria dos operadores de indexação, como o fatiamento. No entanto, esses métodos de acesso a dados têm uma diferença importante. Enquanto `.iloc` exclui o elemento de fechamento, `.loc` inclui-o. Dê uma olhada neste bloco de código:

```
>>> # Return the elements with the implicit index: 1, 2
>>> colors.iloc[1:3]
2    purple
3     blue
dtype: object
```

```
>>> # Return the elements with the explicit index between 3 and 8
>>> colors.loc[3:8]
3     blue
5    green
8    yellow
dtype: object
```

Você também pode passar um índice posicional negativo para `.iloc`

```
>>> colors.iloc[-2]
'green'
```

Acessando elementos do DataFrame

Uma vez que um `DataFrame` consiste em objetos, você pode usar as mesmas ferramentas para acessar seus elementos. Você usará o operador de indexação para as colunas e os métodos de acesso e nas linhas. `DataFrameSeriesDataFrame.loc.iloc`

Utilizando o Operador de Indexação

Se você pensa em um dicionário cujos valores são, então faz sentido que você possa acessar suas colunas com o operador de indexação: `DataFrame Series`

```
>>> city_data["revenue"]
Amsterdam    4200
Tokyo        6500
Toronto      8000
Name: revenue, dtype: int64

>>> type(city_data["revenue"])
pandas.core.series.Series
```

Here, you use the indexing operator to select the column labeled `"revenue"`

If the column name is a string, then you can use attribute-style accessing with dot notation as well:

```
>>> city_data.revenue
Amsterdam    4200
Tokyo        6500
Toronto      8000
Name: revenue, dtype: int64
```

`city_data["revenue"]` e retornar a mesma saída. `city_data.revenue`

Há uma situação em que acessar elementos com notação de pontos pode não funcionar ou pode levar a surpresas. É quando um nome de coluna coincide com um atributo ou nome do método: `DataFrame DataFrame`

```
>>> toys = pd.DataFrame([
...     {"name": "ball", "shape": "sphere"},
```



```

...     {"name": "Rubik's cube", "shape": "cube"}
... ])
>>> toys["shape"]
0    sphere
1     cube
Name: shape, dtype: object
>>> toys.shape
(2, 2)

```

A operação de indexação retorna os dados corretos, mas a operação no estilo atributo ainda retorna a forma do `.loc`. Você só deve usar o acesso ao estilo de atributo em sessões interativas ou para operações de leitura. Você não deve usá-lo para código de produção ou para manipular dados (como definir novas colunas). `toys["shape"]` `toys.shape` `DataFrame`

Usando e `.loc` `.iloc`

Semelhante a `toys`, um `DataFrame` também fornece e **métodos de acesso a dados**. Lembre-se, usa o rótulo e o índice posicional: `Series` `DataFrame` `.loc` `.iloc` `.loc` `.iloc`

```

>>> city_data.loc["Amsterdam"]
revenue      4200.0
employee_count    5.0
Name: Amsterdam, dtype: float64

>>> city_data.loc["Tokyo": "Toronto"]
      revenue employee_count
Tokyo    6500         8.0
Toronto   8000         NaN

>>> city_data.iloc[1]
revenue      6500.0
employee_count    8.0
Name: Tokyo, dtype: float64

```

Cada linha de código seleciona uma linha diferente de: `city_data`

1. `city_data.loc["Amsterdam"]` seleciona a linha com o índice de rótulos `.loc` `"Amsterdam"`
2. `city_data.loc["Tokyo": "Toronto"]` seleciona as linhas com índices de rótulos de `.loc`. Lembre-se, é inclusivo. `"Tokyo"` `"Toronto"` `.loc`
3. `city_data.iloc[1]` seleciona a linha com o índice posicional `.iloc` `1`, que é `"Tokyo"`

Tudo bem, você usou `.loc` e `.iloc` em pequenas estruturas de dados. Agora, é hora de praticar com algo maior! Use um método de acesso de dados para exibir a penúltima linha do conjunto de dados. Em seguida, expanda o bloco de código abaixo para ver uma solução: `.loc` `.iloc` `nba`

```
>>> nba.iloc[-2]
gameorder      63157
game_id        201506170CLE
lg_id          NBA
_iscopy         0
year_id        2015
date_game      6/16/2015
seasongame     102
is_playoffs    1
team_id        CLE
fran_id        Cavaliers
pts            97
elo_i          1700.74
elo_n          1692.09
win_equiv      59.2902
opp_id         GSW
opp_fran       Warriors
opp_pts        105
opp_elo_i      1813.63
opp_elo_n      1822.29
game_location  H
game_result    L
forecast       0.48145
notes          NaN
Name: 126312, dtype: object
```

Para a, os métodos de acesso aos dados e também aceitam um segundo parâmetro. Enquanto o primeiro parâmetro seleciona linhas com base nos índices, o segundo parâmetro seleciona as colunas. Você pode usar esses parâmetros juntos para selecionar um subconjunto de linhas e colunas do seu: `DataFrame.loc.iloc` `DataFrame`

```
>>> city_data.loc["Amsterdam": "Tokyo", "revenue"]
Amsterdam      4200
Tokyo          6500
Name: revenue, dtype: int64
```

Observe que você separa os parâmetros com uma vírgula (,). O primeiro parâmetro, diz para selecionar todas as linhas entre esses dois rótulos. O segundo parâmetro vem depois da vírgula e diz para selecionar a coluna. `, "Amsterdam" : "Tokyo," "revenue"`

É hora de ver a mesma construção em ação com o conjunto de dados maior.. Você só está interessado nos nomes das equipes e nas pontuações, então selecione esses elementos também. Expanda o bloco de código abaixo para ver uma solução: `nba55555559`

```
>>> nba.loc[5555:5559, ["fran_id", "opp_fran", "pts", "opp_pts"]]
```

	fran_id	opp_fran	pts	opp_pts
5555	Pistons	Warriors	83	56
5556	Celtics	Knicks	95	74
5557	Knicks	Celtics	74	95
5558	Kings	Sixers	81	86
5559	Sixers	Kings	86	81

Consultando seu conjunto de dados

Você viu como acessar subconjuntos de um enorme conjunto de dados com base em seus índices. Agora, você selecionará linhas com base nos valores nas colunas do seu conjunto de dados para **consultar** seus dados. Por exemplo, você pode criar um novo que contém apenas jogos jogados após 2010: `DataFrame`

```
>>> current_decade = nba[nba["year_id"] > 2010]
>>> current_decade.shape
(12658, 23)
```

Você ainda tem todas as 23 colunas, mas a sua nova consiste apenas em linhas onde o valor na coluna é maior do que `. DataFrame "year_id" 2010`

Você também pode selecionar as linhas onde um campo específico não é nulo:

```
>>> games_with_notes = nba[nba["notes"].notnull()]
>>> games_with_notes.shape
(5424, 23)
```

Isso pode ser útil se você quiser evitar qualquer valor faltando em uma coluna. Você também pode usar para alcançar o mesmo objetivo. `.notna()`

Você pode até mesmo acessar valores do tipo de dados como e executar métodos de sequência neles: `object str`

```
>>> ers = nba[nba["fran_id"].str.endswith("ers")]
>>> ers.shape
(27797, 23)
```

Você usa para filtrar seu conjunto de dados e encontrar todos os jogos onde o nome do time da casa termina com `.str.endswith() "ers"`

Você pode combinar vários critérios e consultar seu conjunto de dados também. Para isso, certifique-se de colocar cada um entre parênteses e usar os operadores lógicos e separá-los. `| &`

Faça uma busca por jogos em Baltimore onde ambas as equipes marcaram mais de 100 pontos. Para ver cada jogo apenas uma vez, você precisará excluir duplicatas:

```
>>> nba[
...     (nba["_iscopy"] == 0) &
...     (nba["pts"] > 100) &
...     (nba["opp_pts"] > 100) &
...     (nba["team_id"] == "BLB")
... ]
```

Aqui, você usa para incluir apenas as entradas que não são cópias. `nba["_iscopy"] == 0`

Sua saída deve conter cinco jogos agitados:

	gameorder	game_id	lg_id	_iscopy	year_id	date_game	seasongame	is_playoffs	team_id	fran_id	pts	elo_i	elo_n	win_equiv	opp_i
1726	864	194902260BLB	NBA	0	1949	2/26/1949	53	0	BLB	Baltimore	114	1421.94	1419.43	38.56	MN
4890	2446	195301100BLB	NBA	0	1953	1/10/1953	32	0	BLB	Baltimore	126	1328.67	1356.65	25.80	BO
4909	2455	195301140BLB	NBA	0	1953	1/14/1953	34	0	BLB	Baltimore	104	1349.83	1346.36	24.88	MN
5208	2605	195303110BLB	NBA	0	1953	3/11/1953	66	0	BLB	Baltimore	107	1284.52	1282.24	19.58	NY
5825	2913	195402220BLB	NBA	0	1954	2/22/1954	60	0	BLB	Baltimore	110	1303.75	1301.97	20.74	BO

Tente construir outra consulta com vários critérios. Na primavera de 1992, ambas as equipes de Los Angeles tiveram que jogar em casa em outra quadra. Consulte seu conjunto de dados para encontrar esses dois jogos. Expanda o bloco de código abaixo para ver uma solução: "LA"

Quando você souber como consultar seu conjunto de dados com vários critérios, você poderá responder a perguntas mais específicas sobre o seu conjunto de dados.

Agrupando e agregando seus dados

Você também pode querer aprender outras características do seu conjunto de dados, como a soma, média ou valor médio de um grupo de elementos. Felizmente, a biblioteca Pandas Python oferece **funções de agrupamento e agregação** para ajudá-lo a realizar essa tarefa.

A tem mais de vinte métodos diferentes para calcular estatísticas descritivas. Aqui estão alguns exemplos: `Series`

```
>>> city_revenues.sum()
18700
```

```
>>> city_revenues.max()
8000

>>> city_revenues.sum()
18700

>>> city_revenues.max()
8000
```

O primeiro método retorna o total enquanto o segundo retorna o valor máximo. Existem outros métodos que você pode usar,

como: `city_revenues.min().mean()`

Lembre-se, uma coluna de um é na verdade um objeto. Por essa razão, você pode usar essas mesmas funções nas colunas de: `DataFrame` `Series` `nba`

```
>>> points = nba["pts"]

>>> type(points)
<class 'pandas.core.series.Series'>

>>> points.sum()
12976235
```

A pode ter várias colunas, o que introduz novas possibilidades de agregações, como agrupamento: `DataFrame`

```
>>> nba.groupby("fran_id", sort=False)["pts"].sum()
fran_id
Huskies      3995
Knicks       582497
Stags        20398
Falcons       3797
Capitols     22387
...
```

Por padrão, pandas classifica as chaves do grupo durante a chamada. Este parâmetro pode levar a ganhos de desempenho. `.groupby()sort=False`

Você também pode agrupar por várias colunas:

```
>>> nba[
...     (nba["fran_id"] == "Spurs") &
...     (nba["year_id"] > 2010)
... ].groupby(["year_id", "game_result"])["game_id"].count()
year_id  game_result
2011     L           25
        W           63
2012     L           20
        W           60
2013     L           30
        W           73
2014     L           27
```

```

      W      78
2015   L      31
      W      58
Name: game_id, dtype: int64

```

Você pode praticar esses fundamentos com um exercício. Dê uma olhada na temporada 2014-15 do Golden State Warriors (). Quantas vitórias e derrotas eles marcaram durante a temporada regular e os playoffs? Expanda o bloco de código abaixo para a solução: `year_id: 2015`

Primeiro, você pode agrupar-se pelo campo, em seguida, pelo resultado: `"is_playoffs"`

```

>>> nba[
...     (nba["fran_id"] == "Warriors") &
...     (nba["year_id"] == 2015)
... ].groupby(["is_playoffs", "game_result"])["game_id"].count()
is_playoffs  game_result
0           L           15
           W           67
1           L           5
           W           16

```

`is_playoffs=0` mostra os resultados para a temporada regular, e mostra os resultados para os playoffs. `is_playoffs=1`

Colunas manipuladoras

Você precisará saber como **manipular as** colunas do seu conjunto de dados em diferentes fases do processo de análise de dados. Você pode adicionar e soltar colunas como parte da fase inicial de limpeza de dados, ou mais tarde com base nos insights de sua análise.

Crie uma cópia do seu original para trabalhar: `DataFrame`

```

>>> df = nba.copy()
>>> df.shape
(126314, 23)

```

Você pode definir novas colunas com base nas já existentes:

```

>>> df["difference"] = df.pts - df.opp_pts
>>> df.shape
(126314, 24)

```

Aqui, você usou as colunas para criar uma nova chamada. Esta nova coluna tem as mesmas funções das antigas: `"pts"` `"opp_pts"` `"difference"`

```

>>> df["difference"].max()
68

```

Aqui, você usou uma função de agregação para encontrar o maior valor da sua nova coluna. `.max()`

Você também pode renomear as colunas do seu conjunto de dados. Parece que e são muito verbosos, então vá em frente e renomeie-os

agora: `"game_result" "game_location"`

```
>>> renamed_df = df.rename(
...     columns={"game_result": "result", "game_location": "location"}
... )
>>> renamed_df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 126314 entries, 0 to 126313
Data columns (total 24 columns):
gameorder      126314 non-null int64
...
location       126314 non-null object
result         126314 non-null object
forecast       126314 non-null float64
notes          5424 non-null object
difference     126314 non-null int64
dtypes: float64(6), int64(8), object(10)
memory usage: 23.1+ MB
```

Note que há um novo objeto, `.`. Como vários outros métodos de manipulação de dados, retorna um novo por padrão. Se você quiser manipular o original diretamente, então também fornece um parâmetro que você pode definir `.renamed_df.rename() DataFrame DataFrame .rename() inplace True`

Seu conjunto de dados pode conter colunas que você não precisa. Por exemplo, as classificações da Elo podem ser um conceito fascinante para alguns, mas você não vai analisá-las neste tutorial. Você pode excluir as quatro colunas relacionadas:

```
>>> df.shape
(126314, 24)

>>> elo_columns = ["elo_i", "elo_n", "opp_elo_i", "opp_elo_n"]

>>> df.drop(elo_columns, inplace=True, axis=1)
>>> df.shape
(126314, 20)
```

Lembre-se, você adicionou a nova coluna em um exemplo anterior, elevando o número total de colunas para 24. Quando você remove as quatro colunas Elo, o número total de colunas cai para 20. "difference"

Especificando tipos de dados

Quando você cria um novo dado, seja chamando um construtor ou lendo um arquivo CSV, o Pandas atribui um **tipo de dados** a cada coluna com base

em seus valores. Embora faça um bom trabalho, não é perfeito. Se você escolher o tipo de dados certo para suas colunas antecipadamente, então você pode melhorar significativamente o desempenho do seu código. `DataFrame`

Dê outra olhada nas colunas do conjunto de dados: `nba`

```
>>> df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 126314 entries, 0 to 126313
Data columns (total 23 columns):
gameorder      126314 non-null int64
game_id        126314 non-null object
lg_id          126314 non-null object
_iscopy        126314 non-null int64
year_id        126314 non-null int64
date_game      126314 non-null object
seasongame     126314 non-null int64
is_playoffs    126314 non-null int64
team_id        126314 non-null object
fran_id        126314 non-null object
pts            126314 non-null int64
elo_i          126314 non-null float64
elo_n          126314 non-null float64
win_equiv      126314 non-null float64
opp_id         126314 non-null object
opp_fran       126314 non-null object
opp_pts        126314 non-null int64
opp_elo_i      126314 non-null float64
opp_elo_n      126314 non-null float64
game_location  126314 non-null object
game_result    126314 non-null object
forecast       126314 non-null float64
notes          5424 non-null object
dtypes: float64(6), int64(7), object(10)
memory usage: 22.2+ MB
```

Dez de suas colunas têm o tipo de dados. A maioria dessas colunas contém texto arbitrário, mas também há alguns candidatos à conversão de tipo de dados. Por exemplo, dê uma olhada na coluna: `objectobjectdate_game`

```
>>> df["date_game"] = pd.to_datetime(df["date_game"])
```

Aqui, você usa para especificar todas as datas do jogo como objetos. `.to_datetime()` `datetime`

Outras colunas contêm texto um pouco mais estruturado. A coluna pode ter apenas três valores diferentes: `game_location`


```
>>> df["game_location"].nunique()
3
>>> df["game_location"].value_counts()
A    63138
H    63138
N      38
Name: game_location, dtype: int64
```

Qual tipo de dados você usaria em um banco de dados relacional para tal coluna? Pandas fornece o tipo de dados para o mesmo propósito: `category`

```
>>> df["game_location"] = pd.Categorical(df["game_location"])

>>> df["game_location"].dtype
CategoricalDtype(categories=['A', 'H', 'N'], ordered=False)
```

dados categóricos têm algumas vantagens sobre texto não estruturado. Quando você especifica o tipo de dados, facilita a validação e salva uma tonelada de memória, pois o Pandas usará apenas os valores únicos internamente. Quanto maior a proporção de valores totais para valores únicos, mais economias de espaço você terá. `category`

Corra de novo. Você deve ver que alterar o tipo de dados para diminuiu o uso da memória. `df.info()`

Muitas vezes você encontrará conjuntos de dados com muitas colunas de texto. Uma habilidade essencial para os cientistas de dados terem é a capacidade de identificar quais colunas eles podem converter para um tipo de dados mais performático.

Tire um momento para praticar isso agora. Encontre outra coluna no conjunto de dados que tenha um tipo de dados genérico e converta-o em uma mais específica. Você pode expandir o bloco de código abaixo para ver uma solução em potencial: `nba`

```
>>> df["game_result"].nunique()
2

>>> df["game_result"].value_counts()
L    63157
W    63157
```

Para melhorar o desempenho, você pode convertê-lo em uma coluna: `category`

```
>>> df["game_result"] = pd.Categorical(df["game_result"])
```

À medida que você trabalha com conjuntos de dados mais massivos, a economia de memória se torna especialmente crucial. Certifique-se de manter o desempenho em mente à medida que você continua a explorar seus conjuntos de dados.

Dados de limpeza

Você pode se surpreender ao encontrar esta seção tão tarde no tutorial! Normalmente, você daria uma olhada crítica no seu conjunto de dados para corrigir quaisquer problemas antes de passar para uma análise mais sofisticada. No entanto, neste tutorial, você vai confiar nas técnicas que aprendeu nas seções anteriores para limpar seu conjunto de dados.

Valores perdidos

Você já se perguntou por que mostra quantos valores não nulos uma coluna contém? A razão é que esta é uma informação vital. **Valores nulos** muitas vezes indicam um problema no processo de coleta de dados. Eles podem fazer várias técnicas de análise, como diferentes tipos de aprendizado de máquina, difíceis ou até mesmo impossíveis. `.info()`

Quando você inspecionar o conjunto de dados, você verá que é bastante limpo. Apenas a coluna contém valores nulos para a maioria de suas linhas: `nbanba.info()notes`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 126314 entries, 0 to 126313
Data columns (total 23 columns):
gameorder      126314 non-null int64
game_id        126314 non-null object
lg_id          126314 non-null object
_iscopy        126314 non-null int64
year_id        126314 non-null int64
date_game      126314 non-null object
seasongame     126314 non-null int64
is_playoffs    126314 non-null int64
team_id        126314 non-null object
fran_id        126314 non-null object
pts            126314 non-null int64
elo_i          126314 non-null float64
elo_n          126314 non-null float64
win_equiv      126314 non-null float64
opp_id         126314 non-null object
opp_fran       126314 non-null object
opp_pts        126314 non-null int64
opp_elo_i      126314 non-null float64
opp_elo_n      126314 non-null float64
game_location  126314 non-null object
game_result    126314 non-null object
forecast       126314 non-null float64
notes          5424 non-null object
dtypes: float64(6), int64(7), object(10)
memory usage: 22.2+ MB
```

Esta saída mostra que a coluna tem apenas 5424 valores não nulos. Isso significa que mais de 120.000 linhas do seu conjunto de dados têm valores nulos nesta coluna. `notes`

Às vezes, a maneira mais fácil de lidar com registros que contêm valores perdidos é ignorá-los. Você pode remover todas as linhas com valores faltantes usando: `.dropna()`

```
>>> rows_without_missing_data = nba.dropna()

>>> rows_without_missing_data.shape
(5424, 23)
```

Claro, esse tipo de limpeza de dados não faz sentido para o seu conjunto de dados, porque não é um problema para um jogo não ter notas. Mas se o seu conjunto de dados contém um milhão de registros válidos e cem onde faltam dados relevantes, então deixar cair os registros incompletos pode ser uma solução razoável. `nba`

Você também pode soltar colunas problemáticas se elas não forem relevantes para sua análise. Para isso, use novamente e forneça o parâmetro: `.dropna(axis=1)`

```
>>> data_without_missing_columns = nba.dropna(axis=1)

>>> data_without_missing_columns.shape
(126314, 22)
```

Agora, o resultado contém todos os 126.314 jogos, mas não a coluna às vezes vazia. `DataFrame notes`

Se houver um valor padrão significativo para o seu caso de uso, então você também pode substituir os valores faltantes por isso:

```
>>> data_with_default_notes = nba.copy()
>>> data_with_default_notes["notes"].fillna(
...     value="no notes at all",
...     inplace=True
... )
>>> data_with_default_notes["notes"].describe()
count          126314
unique           232
top      no notes at all
freq          120890
Name: notes, dtype: object
```

Valores inválidos

Valores inválidos podem ser ainda mais perigosos do que valores perdidos. Muitas vezes, você pode realizar sua análise de dados como esperado, mas os resultados que você recebe são peculiares. Isso é especialmente importante se o seu conjunto de dados for enorme ou usado de entrada manual. Valores inválidos são muitas vezes mais desafiadores de detectar, mas você pode implementar algumas verificações de sanidade com consultas e agregações.

Uma coisa que você pode fazer é validar os intervalos de seus dados. Para isso, é muito útil. Lembre-se de que retorna a seguinte saída: `.describe()`

	gameorder	_iscopy	year_id	seasongame	is_playoffs	pts	elo_i	elo_n	win_equiv	opp_pts	opp_elo_i	opp_elo_n	forecast
count	126314.00	126314.0	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00
mean	31579.00	0.5	1988.20	43.53	0.06	102.73	1495.24	1495.24	41.71	102.73	1495.24	1495.24	0.50
std	18231.93	0.5	17.58	25.38	0.24	14.81	112.14	112.46	10.63	14.81	112.14	112.46	0.22
min	1.00	0.0	1947.00	1.00	0.00	0.00	1091.64	1085.77	10.15	0.00	1091.64	1085.77	0.02
25%	15790.00	0.0	1975.00	22.00	0.00	93.00	1417.24	1416.99	34.10	93.00	1417.24	1416.99	0.33
50%	31579.00	0.5	1990.00	43.00	0.00	103.00	1500.95	1500.95	42.11	103.00	1500.95	1500.95	0.50
75%	47368.00	1.0	2003.00	65.00	0.00	112.00	1576.06	1576.29	49.64	112.00	1576.06	1576.29	0.67
max	63157.00	1.0	2015.00	108.00	1.00	186.00	1853.10	1853.10	71.11	186.00	1853.10	1853.10	0.98

O varia entre 1947 e 2015. Isso soa plausível. `year_id`

O que acha de? Como pode ser o mínimo? Vamos dar uma olhada nesses jogos: `pts 0`

```
>>> nba[nba["pts"] == 0]
```

pts	team_id	fran_id	pts	elo_i	elo_n	win_equiv	opp_id	opp_fran	opp_pts	opp_elo_i	opp_elo_n	game_location	game_result	forecast	notes
0	DNR	Nuggets	0	1460.34	1457.45	40.41	VIR	Squires	2	1484.19	1487.08	A	L	0.33	at Richmond VA, forfeit to VIR
4															

Parece que o jogo foi perdido. Dependendo da sua análise, você pode querer removê-lo do conjunto de dados.

Valores Inconsistentes

Às vezes, um valor seria totalmente realista em si mesmo, mas não se encaixa com os valores nas outras colunas. Você pode definir alguns critérios de consulta que são mutuamente exclusivos e verificar se estes não ocorrem juntos.

No conjunto de dados da NBA, os valores dos campos devem ser consistentes uns com os outros. Você pode verificar isso usando o atributo: `ptsopp_ptsgame_result.empty`

```
>>> nba[(nba["pts"] > nba["opp_pts"]) & (nba["game_result"] != 'W')].empty
True
```

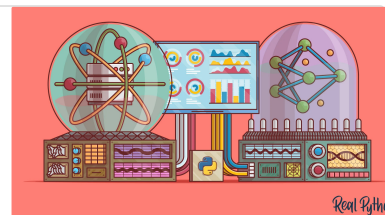
```
>>> nba[(nba["pts"] < nba["opp_pts"]) & (nba["game_result"] != 'L')].empty
True
```

Felizmente, ambas as consultas retornam um vazio `.DataFrame`

Esteja preparado para surpresas sempre que estiver trabalhando com conjuntos de dados brutos, especialmente se eles foram coletados de diferentes fontes ou através de um pipeline complexo. Você pode ver linhas onde um time marcou mais pontos do que seu adversário, mas ainda não ganhou – pelo menos, de acordo com o seu conjunto de dados! Para evitar situações como esta, certifique-se de adicionar mais técnicas de limpeza de dados ao seu arsenal Pandas e Python.

Pythonic Data Cleaning With Pandas and NumPy - Real Python

We'll cover the following: Note: I recommend using Jupyter Notebooks to follow along. Often, you'll find that not all the categories of data in a dataset are useful to you. For example, <https://realpython.com/python-data-cleaning-numpy-pandas/>



Combinando vários conjuntos de dados

Na seção anterior, você aprendeu a limpar um conjunto de dados bagunçado. Outro aspecto dos dados do mundo real é que muitas vezes vem em várias partes. Nesta seção, você aprenderá a pegar essas peças e combiná-las em um conjunto de dados pronto para análise.

Antes, você combinou dois objetos em um baseado em seus índices. Agora, você vai dar um passo adiante e usar para combinar com outro. Digamos que você conseguiu reunir alguns dados sobre mais duas

cidades: `SeriesDataFrame.concat()`city_dataDataFrame

```
>>> further_city_data = pd.DataFrame(
...     {"revenue": [7000, 3400], "employee_count": [2, 2]},
...     index=["New York", "Barcelona"]
... )
```

Esta segunda contém informações sobre as cidades e `.DataFrame` "New York" "Barcelona"

Você pode adicionar essas cidades ao uso: `city_data.concat()`

```
>>> all_city_data = pd.concat([city_data, further_city_data], sort=False)
>>> all_city_data
Amsterdam    4200    5.0
Tokyo        6500    8.0
Toronto      8000   NaN
New York     7000    2.0
Barcelona    3400    2.0
```

Por padrão, combina junto. Em outras palavras, ele anexa linhas. Você também pode usá-lo para anexar colunas fornecendo o parâmetro: `concat(axis=0)` `axis=1`

```
>>> city_countries = pd.DataFrame({
...     "country": ["Holland", "Japan", "Holland", "Canada", "Spain"],
...     "capital": [1, 1, 0, 0, 0]},
...     index=["Amsterdam", "Tokyo", "Rotterdam", "Toronto", "Barcelona"]
... )
>>> cities = pd.concat([all_city_data, city_countries], axis=1, sort=False)
>>> cities
```

	revenue	employee_count	country	capital
Amsterdam	4200.0	5.0	Holland	1.0
Tokyo	6500.0	8.0	Japan	1.0
Toronto	8000.0	NaN	Canada	0.0
New York	7000.0	2.0	NaN	NaN
Barcelona	3400.0	2.0	Spain	0.0
Rotterdam	NaN	NaN	Holland	0.0

Note como pandas adicionados para os valores faltantes. Se você quiser combinar apenas as cidades que aparecem em ambos os objetos, então você pode definir o parâmetro para: `NaNDataFramejoininner`

```
>>> pd.concat([all_city_data, city_countries], axis=1, join="inner")
```

	revenue	employee_count	country	capital
Amsterdam	4200	5.0	Holland	1
Tokyo	6500	8.0	Japan	1
Toronto	8000	NaN	Canada	0
Barcelona	3400	2.0	Spain	0

Embora seja mais simples combinar dados com base no índice, não é a única possibilidade. Você pode usar `.merge()` para implementar uma operação de join semelhante à do SQL:

```
>>> countries = pd.DataFrame({
...     "population_millions": [17, 127, 37],
...     "continent": ["Europe", "Asia", "North America"]
... }, index= ["Holland", "Japan", "Canada"])
>>> pd.merge(cities, countries, left_on="country", right_index=True)
```

Aqui, você passa o parâmetro para indicar em que coluna você deseja participar. O resultado é maior que contém não apenas dados da cidade, mas também a população e o continente dos respectivos países: `left_on="country".merge()` `DataFrame`

	revenue	employee_count	country	capital	population_millions	continent
Amsterdam	4200.0	5.0	Holland	1.0	17	Europe
Rotterdam	NaN	NaN	Holland	0.0	17	Europe
Tokyo	6500.0	8.0	Japan	1.0	127	Asia
Toronto	8000.0	NaN	Canada	0.0	37	North America

Note que o resultado contém apenas as cidades onde o país é conhecido e aparece no aderido. `DataFrame`

`.merge()` realiza uma junta interna por padrão. Se você quiser incluir todas as cidades no resultado, então você precisa fornecer o parâmetro: `how`

```
>>> pd.merge(
...     cities,
...     countries,
...     left_on="country",
...     right_index=True,
...     how="left"
... )
```

Com esta adesão, você verá todas as cidades, incluindo aquelas sem dados do país:left

	revenue	employee_count	country	capital	population_millions	continent
Amsterdam	4200.0	5.0	Holland	1.0	17.0	Europe
Tokyo	6500.0	8.0	Japan	1.0	127.0	Asia
Toronto	8000.0	NaN	Canada	0.0	37.0	North America
New York	7000.0	2.0	NaN	NaN	NaN	NaN
Barcelona	3400.0	2.0	Spain	0.0	NaN	NaN
Rotterdam	NaN	NaN	Holland	0.0	17.0	Europe

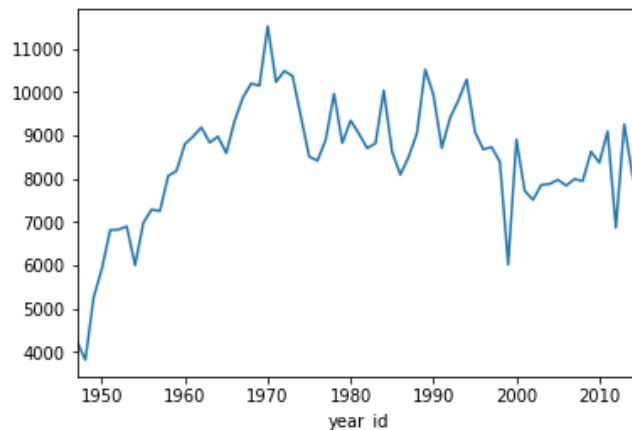
Visualizando seu DataFrame pandas

Visualização de dados é uma das coisas que funciona muito melhor em um notebook Jupyter do que em um terminal, então vá em frente e dispare um para cima. Se você precisa de ajuda para começar, então confira [Jupyter Notebook: An Introduction](#). Você também pode acessar o notebook Jupyter que contém os exemplos deste tutorial clicando no link abaixo:

```
%matplotlib inline
```

Ambos e objetos têm um método `.plot()`, que é um invólucro ao redor. Por padrão, ele cria um gráfico de linha. Visualize quantos pontos os Knicks marcaram ao longo das temporadas: `SeriesDataFramematplotlib.pyplot.plot()`

```
>>> nba[nba["fran_id"] == "Knicks"].groupby("year_id")["pts"].sum().plot()
```



```
>>> nba["fran_id"].value_counts().head(10).plot(kind="bar")
```

Os Lakers estão liderando os Celtics por uma vantagem mínima, e há mais seis equipes com uma contagem de jogos acima de 5000.

Agora tente um exercício mais complicado. Em 2013, o Miami Heat venceu o campeonato. Crie um enredo de torta mostrando a contagem de suas vitórias e perdas durante essa temporada. Em seguida, expanda o bloco de código para ver uma solução:

Primeiro, você define um critério para incluir apenas os jogos do Heat a partir de 2013. Então, você cria um enredo da mesma forma que você já viu acima:

```
>>> nba[
...     (nba["fran_id"] == "Heat") &
...     (nba["year_id"] == 2013)
... ]["game_result"].value_counts().plot(kind="pie")
```