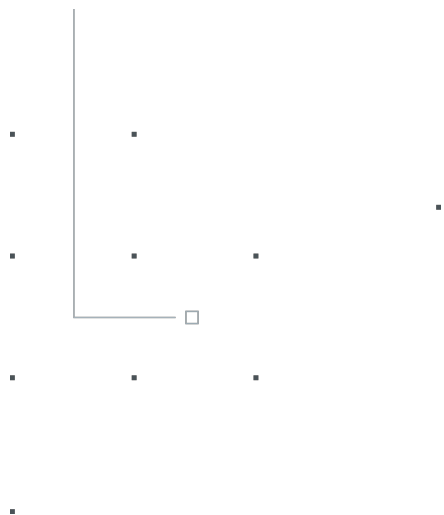


FIAP

NBA



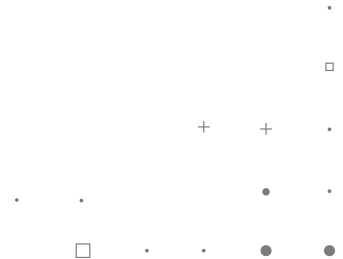
Pandas

Dheny R. Fernandes

1. Pandas

1. Estrutura de Dados
2. Funções Essenciais
 1. Eliminando entradas de um dos eixos
 2. Aplicação de Função e Mapeamento
3. Ordenação e Ranking
4. Sumarização e Estatística Descritiva
5. Carregamento e Armazenamento de Dados
6. Combinação de Dados

Pandas



Pandas é uma biblioteca que contém **estrutura de dados de alto nível e ferramentas de manipulação projetadas para tornar análise de dados rápida e fácil em Python.**

Pandas é construído sobre Numpy, o que torna simples a utilização de aplicações baseadas em Numpy. Embora o Pandas adote muitos estilos de codificação do NumPy, a maior diferença é que o Pandas foi projetado para trabalhar com dados tabulares ou heterogêneos. O NumPy, por outro lado, é mais adequado para trabalhar com dados homogêneos em forma de array.

Importação:

- `from pandas import Series, DataFrame`
- `import pandas as pd`

Para começar a entender Pandas é preciso compreender e ficar confortável com suas duas principais estrutura de dados: **Series** e **DataFrame**.

Ao mesmo tempo em que não são soluções universais, **elas** proveem uma base sólida e fácil de usar para a maioria das aplicações.

Série: uma série é um objeto 1D que contém um array de dados e é associado a um índice. Como não especificamos um índice para os dados, é criado um índice padrão que consiste nos inteiros de 0 a N - 1 (onde N é o comprimento dos dados).

```
from pandas import Series, DataFrame
import pandas as pd
obj = Series([4, 7, -5, 3])
obj
```

```
0    4
1    7
2   -5
3    3
dtype: int64
```

É possível recuperar os valores e o índice de uma série através de seus atributos:

```
print(obj.values)
print(obj.index) #obj.index.values
```

```
[ 4  7 -5  3]
RangeIndex(start=0, stop=4, step=1)
```

É possível criar uma série especificando o índice:

```
obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])  
obj2
```

```
d    4  
b    7  
a   -5  
c    3  
dtype: int64
```

As operações comuns em arrays que o Numpy oferece podem ser aplicadas nas series, preservando-se os índices:

```
import numpy as np  
print(obj2[obj2 > 0])  
print(obj2 * 2)  
print(np.exp(obj2))
```


É possível criar uma Serie a partir de um dicionário:

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}  
obj3 = Series(sdata)
```

Quando passa apenas um dicionário, o índice na série resultante terá as chaves dos dicionários na ordem escrita

```
states = ['Oregon', 'Texas', 'California', 'Ohio']  
obj4 = Series(sdata, index=states)  
obj4
```

```
Oregon      16000.0  
Texas       71000.0  
California   NaN  
Ohio        35000.0  
dtype: float64
```

Entretanto, isso gerou dados faltantes (NaN), visto que não tinha valor associado à key *California*. É possível usar algumas funções para detectar valores faltantes.

```
print(pd.isna(obj4)) #isnull  
print(pd.notna(obj4)) # notnull
```

Oregon	False
Texas	False
California	True
Ohio	False
dtype: bool	
Oregon	True
Texas	True
California	False
Ohio	True
dtype: bool	

Uma importante característica das Series é que ela alinha automaticamente dados de índices diferentes em operações aritméticas

```
print(obj3)
print(obj4)
obj3 + obj4
```

```
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
```

```
dtype: int64
```

```
Oregon      16000.0
Texas       71000.0
California   NaN
Ohio        35000.0
```

```
dtype: float64
```

```
California   NaN
Ohio         70000.0
Oregon       32000.0
Texas       142000.0
Utah         NaN
```

```
dtype: float64
```

Tanto a Serie quanto o índice possuem um atributo *name*

```
obj4.name = 'population'  
obj4.index.name = 'state'  
obj4
```

```
state  
Oregon      16000.0  
Texas       71000.0  
California   NaN  
Ohio        35000.0  
Name: population, dtype: float64
```

O índice de uma série pode ser alterado através de atribuição:

```
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']  
obj
```

```
Bob         4  
Steve       7  
Jeff       -5  
Ryan        3  
dtype: int64
```

Um DataFrame representa uma estrutura de dados tabular contendo uma coleção ordenada de colunas, sendo que cada uma pode ser de um tipo diferente.

Um DataFrame possui índice de coluna e linha.

Existem diversas maneiras de se construir um DataFrame. Uma das mais comuns é a partir de um dicionário de listas ou arrays de mesmo tamanho:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],  
        'year': [2000, 2001, 2002, 2001, 2002],  
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}  
df = DataFrame(data)  
df
```

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

É possível informar a sequencia das colunas no DataFrame e, se houver uma coluna sem dados, seus valores serão NaN. É possível definir o índice também.

```
df2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],  
                index=['one', 'two', 'three', 'four', 'five'])  
df2
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN

É possível retornar uma coluna de um DataFrame como um objeto Serie através da notação de dicionário ou atributo

```
print(df['state'])  
print(df.year)
```

Para recuperar uma linha, usamos *loc* e *iloc*. O primeiro é para índice baseado em string e o segundo para índice baseado em inteiro:

```
print(df2.loc['four']) #label  
print(df2.iloc[0]) #int
```


Visto que o DataFrame é bi-dimensional, podemos selecionar um subconjunto de linhas e colunas com uma notação parecida com a do Numpy, seja usando loc ou iloc. Veja:

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),  
                    index=["Ohio", "Colorado", "Utah", "New York"],  
                    columns=["one", "two", "three", "four"])  
data.loc['Colorado'] #seleciona a linha cujo índice é Colorado
```

```
data.loc[["Colorado", "New York"]] #seleciona as linhas Colorado e New York
```

```
data.loc["Colorado", ["two", "three"]] #seleciona a linha Colorado e as colunas two e three
```

O mesmo é válido quando usamos `iloc`:

```
data.iloc[2] #linha 2
```

```
data.iloc[[2, 1]] #linhas 2 e 1, nessa ordem
```

```
data.iloc[2, [3, 0, 1]] #linha 2, colunas 3, 0 e 1, nessa ordem
```

```
data.iloc[[1, 2], [3, 0, 1]] #linhas 1 e 2, colunas 3, 0 e 1, nessa ordem
```

É possível combinar slices:

```
# todas as linhas, as 3 primeiras colunas, desde que seja maior que 5  
data.iloc[:, :3][data.three > 5]
```

E usar arrays booleanos (apenas com loc):

```
data.loc[data.three >= 5]
```

Podemos modificar os valores de colunas através de atribuição:

```
df2['debt'] = np.arange(5)
```

Entretanto, a lista ou array usado para atribuição deve ser do mesmo tamanho da coluna. Se usar uma Serie, deve-se passar o índice das linhas. Se não houver, será inserido NaN

```
val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])  
df2['debt'] = val  
df2
```

Atribuir uma coluna que não existe a um DataFrame irá criar uma nova coluna:

```
df2['eastern'] = df2.state == 'Ohio'  
df2
```

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False

Para apagar uma coluna, use *del*:

```
del df2['eastern']  
df2
```

Para recuperar os valores de um DataFrame usamos a mesma estrutura vista em Series:

```
df.values
```

```
array([[1.5, 'Ohio', 2000],  
       [1.7, 'Ohio', 2001],  
       [3.6, 'Ohio', 2002],  
       [2.4, 'Nevada', 2001],  
       [2.9, 'Nevada', 2002]], dtype=object)
```

Os índices são responsáveis por manter os rótulos dos eixos.

```
obj = Series(range(3), index=['a', 'b', 'c'])  
obj.index.values
```

Índices são imutáveis e é possível verificar se existe um índice.

```
print('state' in df2.columns)  
print(0 in df.index)
```

Existem uma série de Métodos associados ao Índice:

Method	Description
append	Concatenate with additional Index objects, producing a new Index
diff	Compute set difference as an Index
intersection	Compute set intersection
union	Compute set union
isin	Compute boolean array indicating whether each value is contained in the passed collection
delete	Compute new Index with element at index <i>i</i> deleted
drop	Compute new index by deleting passed values
insert	Compute new Index by inserting element at index <i>i</i>
is_monotonic	Returns True if each element is greater than or equal to the previous element
is_unique	Returns True if the Index has no duplicate values
unique	Compute the array of unique values in the Index

Dropar (eliminar) uma ou mais entradas de um dos eixos é relativamente simples, visto que o Pandas fornece um método simples que nos auxilia nessa tarefa. Observe:

```
import numpy as np
obj = pd.Series(np.arange(5.), index=["a", "b", "c", "d", "e"])
obj
```

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

```
new_obj = obj.drop("c")
new_obj
```

Com DataFrame, a operação é a mesma:

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),  
                    index=["Ohio", "Colorado", "Utah", "New York"],  
                    columns=["one", "two", "three", "four"])  
data
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
data.drop(index=["Colorado", "Ohio"]) #linhas
```

```
data.drop(columns=["two"]) #colunas
```

É possível ainda dropar valores das colunas passando o parâmetro *axis=1* ou *axis="columns"*:

```
data.drop("two", axis=1)
```

```
data.drop(["two", "four"], axis="columns")
```

As funções universais do Numpy funcionam bem no pandas:

```
df = DataFrame(np.random.randn(4, 3), columns=list('bde'),  
               index=['Utah', 'Ohio', 'Texas', 'Oregon'])  
np.abs(df) #retorna valor absoluto
```

Outra frequente operação é aplicar uma função nos arrays de cada linha ou coluna. *Apply* faz isso:

```
f = lambda x: x.max() - x.min()  
print(df.apply(f))  
print(df.apply(f, axis=1))
```

A função passada ao *apply* pode retornar uma Serie também, não apenas um escalar:

```
def f2(x):  
    return Series([x.min(), x.max()], index=['min', 'max'])  
  
df.apply(f2)
```

É possível aplicar usar funções elemento-a-elemento. Para isso, usa-se *applymap*:

```
format2 = lambda x: '%.2f' % x  
df.applymap(format2)
```

Series e DataFrames podem ser ordenados através do método `sort_index()`:

```
obj = Series(range(4), index=['d', 'a', 'b', 'c'])
df2 = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
                 columns=['d', 'a', 'b', 'c'])

print(obj)
print(df2)
print(obj.sort_index())
print(df2.sort_index())
print(df2.sort_index(axis=1))
```

É possível ordenar uma Serie ou DataFrame pelos valores:

```
obj = Series([4, 7, -3, 2])
obj.sort_values() #igual para pandas
```

Rank cria um índice que pode ser ascendente ou descendente:

```
data = {'name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],  
        'nota': [8, 7, 7.5, 10, 5]}  
df4 = DataFrame(data)  
print(df4)  
df4['rank'] = df4['nota'].rank(ascending=0)  
df4
```

Pandas possui um vasto conjunto de métodos estatísticos e matemáticos. Vejamos:

```
df5 = DataFrame([[1.4, np.nan], [7.1, -4.5],  
                 [np.nan, np.nan], [0.75, -1.3]],  
                 index=['a', 'b', 'c', 'd'],  
                 columns=['one', 'two'])  
  
print(df5)  
print(df5.sum())  
print(df5.sum(axis=1))  
print(df5.count())
```


O método *describe* oferece um bom resumo dos dados:

```
df5.describe()
```

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

Em dados não numéricos, *describe* produz um resumo estatístico alternativo:

```
obj.describe()
```

```
count      16  
unique      3  
top         a  
freq        8  
dtype: object
```

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index values at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct_change	Compute percent changes

Unique retorna os valores únicos numa série:

```
obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])  
uniques = obj.unique()  
uniques
```

isin() verifica a relação de pertencimento entre dois conjuntos:

```
mask = obj.isin(['b', 'c'])  
mask
```

Dados faltantes são comuns em muitas aplicações de análises de dados. Pandas foi projetado para lidar com isso o menos dolorido possível:

```
string_data = Series(['laranja', 'uva', np.nan, 'abacate'])
print(string_data)
print(string_data.isnull())
string_data[0] = None
print(string_data.isnull())
```

filtrando valores faltantes:

```
data = DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],  
                  [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])  
print(data)  
cleaned = data.dropna()  
print(cleaned)  
data.dropna(how='all')
```

É possível preencher NaN:

```
print(data.fillna(0))  
print(data.fillna(data.mean()))
```

Resolver o exercício no notebook

Pandas provê alguns métodos de leitura de arquivos externos. A tabela abaixo mostra os métodos:

Format Type	Data Description	Reader	Writer
text	CSV	<code>read_csv</code>	<code>to_csv</code>
text	JSON	<code>read_json</code>	<code>to_json</code>
text	HTML	<code>read_html</code>	<code>to_html</code>
text	Local clipboard	<code>read_clipboard</code>	<code>to_clipboard</code>
binary	MS Excel	<code>read_excel</code>	<code>to_excel</code>
binary	HDF5 Format	<code>read_hdf</code>	<code>to_hdf</code>
binary	Feather Format	<code>read_feather</code>	<code>to_feather</code>
binary	Parquet Format	<code>read_parquet</code>	<code>to_parquet</code>
binary	Msgpack	<code>read_msgpack</code>	<code>to_msgpack</code>
binary	Stata	<code>read_stata</code>	<code>to_stata</code>
binary	SAS	<code>read_sas</code>	
binary	Python Pickle Format	<code>read_pickle</code>	<code>to_pickle</code>
SQL	SQL	<code>read_sql</code>	<code>to_sql</code>
SQL	Google Big Query	<code>read_gbq</code>	<code>to_gbq</code>

Uma importante característica desses métodos é a *Inferência de Tipo*. Com ela, não há necessidade de especificar qual coluna é numérica, string ou booleana.

Para continuarmos, é necessário ter feito o clone dos arquivos no github.

Manipulando arquivos externos:

```
poke = pd.read_csv('Pokemon.csv')  
poke.head()
```

#		Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	1	False
3	3	VenusaurMega Venusaur	Grass	Poison	625	80	100	123	122	120	80	1	False
4	4	Charmander	Fire	NaN	309	39	52	43	60	50	65	1	False

JSON, sigla para JavaScript Object Notation, se tornou um dos formatos padrões para enviar dados via HTTP. É um formato de dados muito mais flexível que os textos tabulares como o CSV.

Observe um exemplo:

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{ "name": "Scott", "age": 25, "pet": "Zuko"},
 {"name": "Katie", "age": 33, "pet": "Cisco"}]
}
"""

print(type(obj))
print(obj)
```

JSON pode trabalhar com vários tipos: objetos (dict), arrays, strings, numbers, booleanos e nulls.

Em Python, existem várias bibliotecas para se trabalhar com JSON. Vamos usar a *json*, built-in do Python.

```
import json
result = json.loads(obj)
result
```

Com isso, podemos facilmente criar um DataFrame:

```
siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])  
siblings
```

O Pandas oferece recurso para tornar o objeto json novamente:

```
asjson = json.dumps(result)  
print(type(asjson))  
asjson
```

Os dados contidos no Pandas podem ser combinados de algumas maneiras:

- *Merge*: conecta linhas em DataFrames baseado em uma ou mais chaves. Operações *join*.
- *Concat*: ‘cola’ objetos a partir de um eixo

Merge - vejamos o seguinte exemplo:

```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],  
                    'data1': range(7)})  
  
df2 = pd.DataFrame({'key': ['a', 'b', 'd'],  
                    'data2': range(3)})  
  
pd.merge(df1, df2) #default inner
```

Se o nome das colunas é diferente em cada objeto, é possível especificá-los separadamente:

```
df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],  
                    'data1': range(7)})  
  
df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],  
                    'data2': range(3)})  
  
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

Os valores 'c' e 'd' ficaram de fora do resultado, visto que foi usado um *inner join*. *Outer join* mescla os *left* e *right join*:

```
pd.merge(df3, df4, how='outer', left_on='lkey', right_on='rkey')
```

É possível realizar merge com múltiplas chaves on:

```
left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],  
                     'key2': ['one', 'two', 'one'],  
                     'lval': [1, 2, 3]})  
right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],  
                      'key2': ['one', 'one', 'one', 'two'],  
                      'rval': [4, 5, 6, 7]})  
pd.merge(left, right, on=['key1', 'key2'], how='outer')
```


Outro método de combinação de dados é a concatenação. Numpy oferece uma ideia de como ela funciona:

```
import numpy as np
arr = np.arange(12).reshape((3, 4))
print(arr)
np.concatenate([arr, arr], axis=1)
```

No pandas, vamos começar trabalhando com Series:

```
s1 = pd.Series([0, 1], index=['a', 'b'])
s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
s3 = pd.Series([5, 6], index=['f', 'g'])
pd.concat([s1, s2, s3])
```

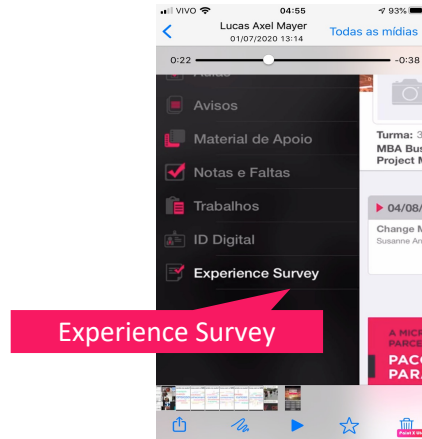
A mesma lógica se estende para DataFrames:

```
df1 = pd.DataFrame(np.arange(6).reshape(3, 2),  
                    index=['a', 'b', 'c'],  
                    columns=['one', 'two'])  
df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2),  
                    index=['a', 'c'],  
                    columns=['three', 'four'])  
  
print(df1)  
print(df2)  
pd.concat([df1, df2], axis=1)
```

Resolver o exercício no notebook

O que você achou da aula de hoje?


Entrar no aplicativo FIAPP, e no menu clicar em Experience Survey



Ou pelo link: <https://fiap.me/Pesquisa-MBA>

Obrigado!

profdheny.fernandes@fiap.com.br

 /dhenyfernandes

FIAP MBA⁺

Copyright © 2018 | Professor Dheny R. Fernandes

Todos os direitos reservados. Reprodução ou divulgação total ou parcial deste documento, é expressamente proibido sem consentimento formal, por escrito, do professor/autor.

FIAP