NOVA IMS INFORMATION MANAGEMENT SCHOOL

Programming for Data Science

Lecture 5

Flávio L. Pinheiro

fpinheiro@novaims.unl.pt | www.flaviolpp.com
www.linkedin.com/in/flaviolpp | X @flavio_lpp

Quizzes

https://www.socrative.com

Login as a student

Room Name: **PDS2025**

Student ID: **Student Number**

or

https://api.socrative.com/rc/5NpKRX

Student ID: **Student Number**

Please Join Now!


Welcome to
Pandas!

Data Structures

Variables
Lists
Dictionaries
Tuples
Sets

Series
DataFrames

PD Series

The first step is to load Pandas, we will use the alias pd

```
import pandas as pd
```

A **Series** is a one-dimensional labelled array-like object. It is capable of holding any data type, e.g. integers, floats, strings, Python objects, and so on. It can be seen as a data structure with two arrays: one functioning as the index, i.e. the labels, and the other one contains the actual data.

```
S = pd.Series([11, 28, 72, 3, 5, 8])
```

```
print(S)
```

Index Values

```
0    11
1    28
2    72
3     3
4     5
5     8
dtype: int64
```

PD DataFrames

Index

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	...
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	...
2	37	services	married	high.school	no	yes	no	telephone	may	mon	...
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	...
4	56	services	married	high.school	no	no	yes	telephone	may	mon	...

Columns

Data

5 rows × 21 columns

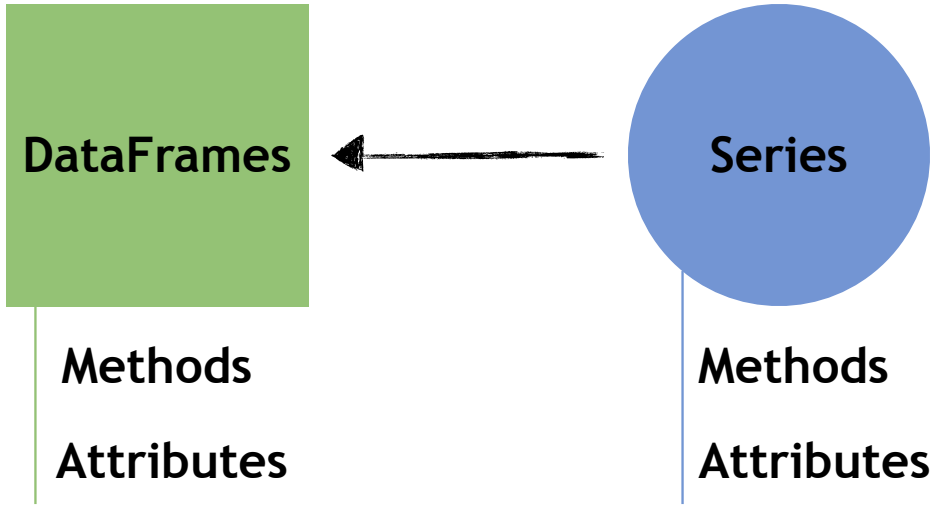
```
df.dtypes
```

```
age                int64
job               object
marital           object
education         object
default           object
housing           object
loan              object
```

dtype

PD Final Notes

Understand the Objects introduced by Pandas



a little bit more pandas

describe example

Pandas provides some handy auxiliar methods/functions that allows for a quick characterisation of the dataset we are working with. Let's look at a dataset from a bank telemarketing campaign (check in Moodle).

.describe() is used to view some basic statistical details like percentile, mean, std etc. of a data frame or a series of numeric values. When this method is applied to a series of string, it returns a different output.

```
1  import pandas as pd
```

```
1  df = pd.read_csv("bank-additional-full.csv",
2                   quotechar='"',
3                   header=0,
4                   delimiter=";")
```

Data Loading

```
1  df.describe()
```

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed
count	41188.00000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000
mean	40.02406	258.285010	2.567593	962.475454	0.172963	0.081886	93.575664	-40.502600	3.621291	5167.035911
std	10.42125	259.279249	2.770014	186.910907	0.494901	1.570960	0.578840	4.628198	1.734447	72.251528
min	17.00000	0.000000	1.000000	0.000000	0.000000	-3.400000	92.201000	-50.800000	0.634000	4963.600000
25%	32.00000	102.000000	1.000000	999.000000	0.000000	-1.800000	93.075000	-42.700000	1.344000	5099.100000
50%	38.00000	180.000000	2.000000	999.000000	0.000000	1.100000	93.749000	-41.800000	4.857000	5191.000000
75%	47.00000	319.000000	3.000000	999.000000	0.000000	1.400000	93.994000	-36.400000	4.961000	5228.100000
max	98.00000	4918.000000	56.000000	999.000000	7.000000	1.400000	94.767000	-26.900000	5.045000	5228.100000

describe

pandas.DataFrame.describe

`DataFrame.describe`(*self*, *percentiles=None*, *include=None*, *exclude=None*) [source]

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

Parameters:

percentiles : *list-like of numbers, optional*

The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include : *'all', list-like of dtypes or None (default), optional*

A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude : *list-like of dtypes or None (default), optional,*

A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

Returns:

Series or DataFrame

Summary statistics of the Series or Dataframe provided.

describe example

Pandas provides some handy auxiliar methods/functions that allows for a quick characterisation of the dataset we are working with.
Let's look at a dataset from a bank telemarketing campaign (check in Moodle).

with the include argument we can include in the describe output all non numeric variables.
However, the output will be different.

```
1 df.describe(include="all")
```

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	campaign	pdays	previous	pc
count	41188.00000	41188	41188	41188	41188	41188	41188	41188	41188	41188	...	41188.000000	41188.000000	41188.000000	
unique	NaN	12	4	8	3	3	3	2	10	5	...	NaN	NaN	NaN	
top	NaN	admin.	married	university.degree	no	yes	no	cellular	may	thu	...	NaN	NaN	NaN	no
freq	NaN	10422	24928	12168	32588	21576	33950	26144	13769	8623	...	NaN	NaN	NaN	
mean	40.02406	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	2.567593	962.475454	0.172963	
std	10.42125	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	2.770014	186.910907	0.494901	
min	17.00000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	1.000000	0.000000	0.000000	
25%	32.00000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	1.000000	999.000000	0.000000	
50%	38.00000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	2.000000	999.000000	0.000000	
75%	47.00000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	3.000000	999.000000	0.000000	
max	98.00000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	56.000000	999.000000	7.000000	

info

pandas.DataFrame.info

`DataFrame.info`(*self*, *verbose=None*, *buf=None*, *max_cols=None*, *memory_usage=None*, *null_counts=None*) [source]

Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

Parameters:

verbose : *bool, optional*
Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf : *writable buffer, defaults to sys.stdout*
Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols : *int, optional*
When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage : *bool, str, optional*
Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.
True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

null_counts : *bool, optional*
Whether to show the non-null counts. By default, this is shown only if the frame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

Returns:

None
This method prints a summary of a DataFrame and returns None.

info example

dtype object refers to strings

```
1 df.info(verbose=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 21 columns):
age               41188 non-null int64
job               41188 non-null object
marital           41188 non-null object
education         41188 non-null object
default           41188 non-null object
housing           41188 non-null object
loan              41188 non-null object
contact           41188 non-null object
month             41188 non-null object
day_of_week       41188 non-null object
duration          41188 non-null int64
campaign          41188 non-null int64
pdays             41188 non-null int64
previous          41188 non-null int64
poutcome          41188 non-null object
emp.var.rate      41188 non-null float64
cons.price.idx    41188 non-null float64
cons.conf.idx     41188 non-null float64
euribor3m         41188 non-null float64
nr.employed       41188 non-null float64
y                 41188 non-null object
dtypes: float64(5), int64(5), object(11)
memory usage: 6.6+ MB
```

Series.value_counts()

It takes a number of arguments:

- **normalize**: If True then the object returned will contain the relative frequencies of the unique values;
- **sort**: Sort by frequencies.
- **bins**: Rather than count values, group them into half-open bins, only works with numeric data;.
- **dropna**: Don't include counts of NaN.

Return a Series containing counts of unique values. The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

```
df['job'].value_counts()
```

```
admin.           10422
blue-collar       9254
technician        6743
services          3969
management        2924
retired           1720
entrepreneur      1456
self-employed     1421
housemaid         1060
unemployed        1014
student            875
unknown            330
Name: job, dtype: int64
```

```
df['age'].value_counts(normalize=True, bins=8)
```

```
(27.125, 37.25]     0.401889
(37.25, 47.375]     0.280349
(47.375, 57.5]      0.185515
(16.918, 27.125]    0.078057
(57.5, 67.625]      0.041177
(67.625, 77.75]     0.008376
(77.75, 87.875]     0.003812
(87.875, 98.0]      0.000825
Name: age, dtype: float64
```

```
df['age'].map(lambda x: int(x/10)*10).value_counts(normalize=True)
```

```
30    0.411236
40    0.255560
50    0.166602
20    0.135816
60    0.017578
70    0.007745
80    0.003399
10    0.001821
90    0.000243
Name: age, dtype: float64
```

DF.groupby()

Group DataFrame using a mapper or by a Series of columns. A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

It takes a number of arguments:

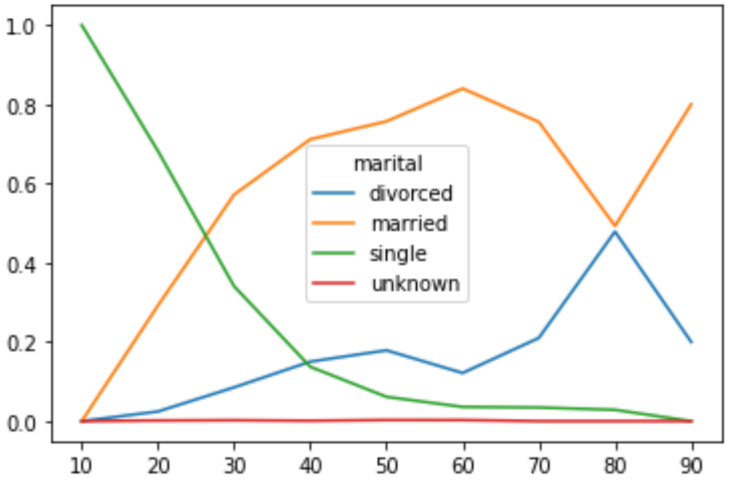
- **by**: Used to determine the groups for the groupby. A label or list of labels may be passed to group by the columns;
- **group_keys**: When calling apply, add group keys to index to identify pieces
- **dropna**: If True, and if group keys contain NA values, NA values together with row/column will be dropped.
- **as_index**: For aggregated output, return object with group labels as the index.

```
df['age'] = df['age'].map(lambda x: int(x/10)*10)
df.groupby('age')['marital'].value_counts(normalize=True).unstack().fillna(0)
```

marital	divorced	married	single	unknown
age				
10	0.000000	0.000000	1.000000	0.000000
20	0.024312	0.291741	0.682338	0.001609
30	0.085252	0.571850	0.340595	0.002303
40	0.150200	0.711571	0.137279	0.000950
50	0.178811	0.756922	0.061352	0.002915
60	0.121547	0.839779	0.035912	0.002762
70	0.210031	0.755486	0.034483	0.000000
80	0.478571	0.492857	0.028571	0.000000
90	0.200000	0.800000	0.000000	0.000000

marital situation by age group, in proportion

.plot() to visualize



Note that **no splitting occurs** until it's needed. Creating the GroupBy object only verifies that you've passed a valid mapping.

DF.groupby()

```
df['age'] = df['age'].map(lambda x: int(x/10)*10)
df.groupby('age')['marital'].value_counts(normalize=True).unstack().fillna(0)
```

Starting point

Age	Marital	Job
10	M	X
20	S	Y
10	D	Z
50	M	X
...	...	...

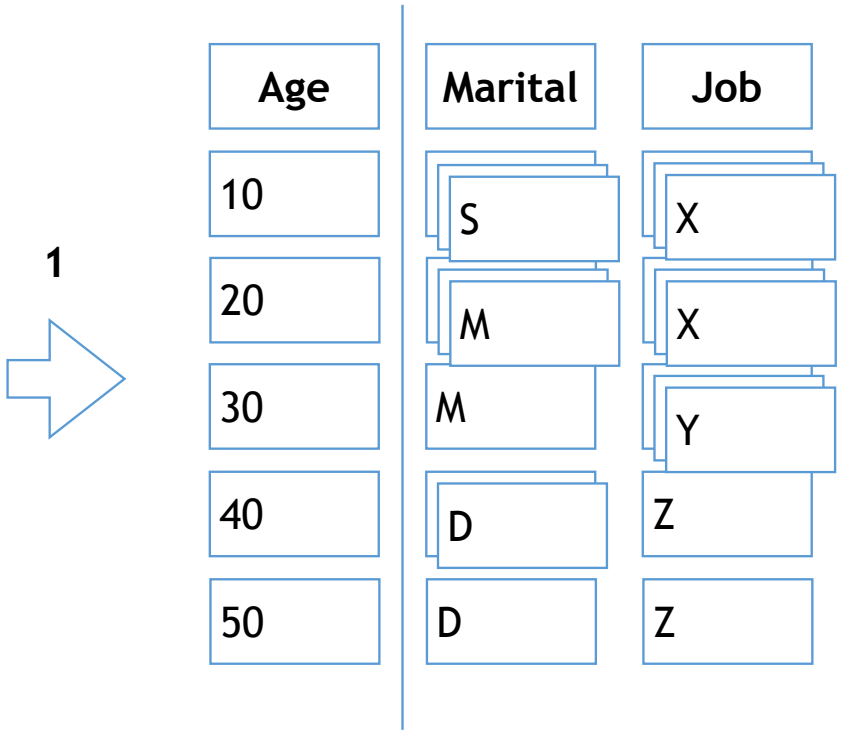
DF.groupby()

```
df['age'] = df['age'].map(lambda x: int(x/10)*10)
df.groupby('age')['marital'].value_counts(normalize=True).unstack().fillna(0)
```

1

Age	Marital	Job
10	M	X
20	S	Y
10	D	Z
50	M	X
...	...	...

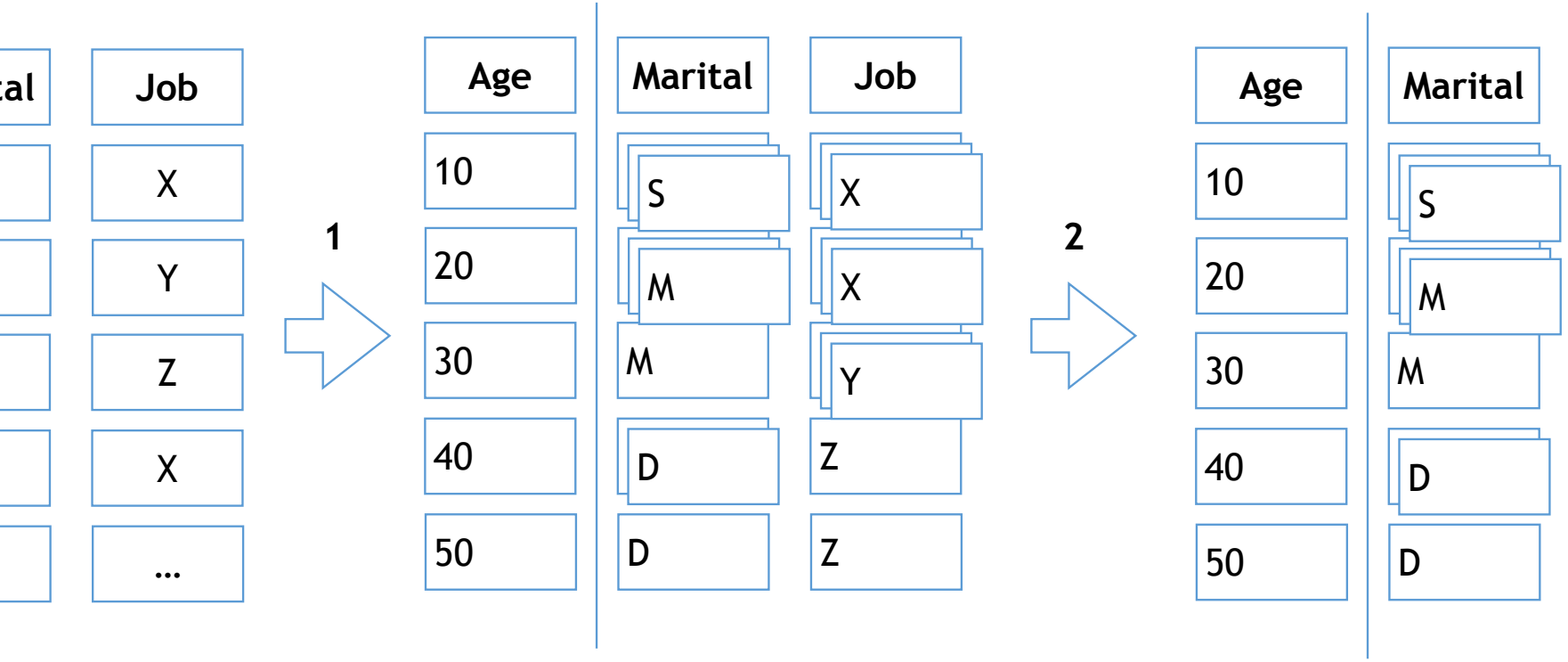
1



DF.groupby()

```
df['age'] = df['age'].map(lambda x: int(x/10)*10)
df.groupby('age')['marital'].value_counts(normalize=True).unstack().fillna(0)
```

1 2



DF.groupby()

```
df['age'] = df['age'].map(lambda x: int(x/10)*10)
df.groupby('age')['marital'].value_counts(normalize=True).unstack().fillna(0)
```

1 2 3

Age	Marital
10	S
20	M
30	M
40	D
50	D

3

Age	Marital	Marital
10	S	nan
	M	0.8
	D	0.2
20	S	nan
	M	nan
	D	1

DF.groupby()

```
df['age'] = df['age'].map(lambda x: int(x/10)*10)
df.groupby('age')['marital'].value_counts(normalize=True).unstack().fillna(0)
```

1 2 3 4

Age	Marital
10	S = 0.8
20	M = 0.5
30	M = 0.7
40	D = 0.1
50	D = 0.9

4

Age	S	M	D
10	nan	0.8	0.2
20	nan	nan	1.0
30	0.2	0.1	0.1
40	0.8	0.2	nan
50	0.3	0.3	0.4

DF.groupby()

don’t trust what I say, see with your own eyes

```
df['age'] = df['age'].map(lambda x: int(x/10)*10)
df.groupby('age')['marital'].value_counts(normalize=True).unstack().fillna(0)
```

1 2 3 4

1
```
pd.DataFrame(df.groupby('age').count())
```

2
```
pd.DataFrame(df.groupby('age')['marital'])
```

3
```
pd.DataFrame(df.groupby('age')['marital'].value_counts(normalize=True))
```

4
```
df.groupby('age')['marital'].value_counts(normalize=True).unstack()
```

Note1: We need to typecast the object that results from the group by to evaluate its contents
Note2: sometimes the output doesn’t make much sense, like in step 1, so it might be more practical to apply an aggregation function

pivot table

df

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

```
df.pivot(index='foo',
         columns='bar',
         values='baz')
```

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

Pivot tables are **used** to summarize, sort, reorganize, group, count, total, or average data stored in lists/tables of records. Pandas includes functions and methods that allow us to generate a Pivot Table from a dataframe.

The difference between pivot tables and GroupBy can sometimes cause confusion;
it helps me to think as essentially a *multidimensional* version of GroupBy aggregation. That is, you split- apply-combine, but both the split and the combine happen across not a one- dimensional index, but across a two-dimensional grid.

pivot table

df

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

```
df.pivot(index='foo',
         columns='bar',
         values='baz')
```

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

The **pivot()** dataframe method provides general purpose pivoting with various data types (strings, numerics, etc.), pandas also provides a **pivot_table()** function for pivoting with aggregation of numeric data.

pivot table example

It takes a number of arguments:

- **data**: a DataFrame object.
- **values**: a column or a list of columns to aggregate.
- **index**: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
- **columns**: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.
- **aggfunc**: function to use for aggregation, defaulting to numpy.mean.

```
1 pd.pivot_table(df, values="age", index="job", columns="contact",aggfunc="mean")
```

contact	cellular	telephone
job		
admin.	38.004631	38.582221
blue-collar	39.500393	39.623439
entrepreneur	41.548538	41.971714
housemaid	46.387500	44.147619
management	42.200841	42.664384
retired	63.826970	57.496933
self-employed	39.684211	40.397727
services	38.037213	37.772014
student	25.345753	27.700980
technician	38.006901	39.610161
unemployed	39.593548	39.954315
unknown	46.880952	44.197531

average age of clients by job and means of contact (cellular versus telephone

pivot table example

calling the function from the library
not the object dataframe

```
1 pd.pivot_table(df, values="age", index="job", columns="contact",aggfunc="mean")
```

contact	cellular	telephone
job		
admin.	38.004631	38.582221
blue-collar	39.500393	39.623439
entrepreneur	41.548538	41.971714
housemaid	46.387500	44.147619
management	42.200841	42.664384
retired	63.826970	57.496933
self-employed	39.684211	40.397727
services	38.037213	37.772014
student	25.345753	27.700980
technician	38.006901	39.610161
unemployed	39.593548	39.954315
unknown	46.880952	44.197531

average age of clients by job and means of contact (cellular versus telephone

It takes a number of arguments:

- **data**: a DataFrame object.
- **values**: a column or a list of columns to aggregate.
- **index**: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
- **columns**: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.
- **aggfunc**: function to use for aggregation, defaulting to numpy.mean.

pivot table example

We can also run multiple aggregations.

```
pd.pivot_table(df,
               values="age",
               index="job",
               columns="contact",
               aggfunc=["max","min","mean"])
```

	max		min		mean	
contact	cellular	telephone	cellular	telephone	cellular	telephone
job						
admin.	72	64	20	21	38.004631	38.582221
blue-collar	80	60	20	20	39.500393	39.623439
entrepreneur	69	62	23	20	41.548538	41.971714
housemaid	85	85	21	25	46.387500	44.147619
management	80	65	21	24	42.200841	42.664384
retired	98	86	23	32	63.826970	57.496933
self-employed	71	65	21	24	39.684211	40.397727
services	62	69	20	20	38.037213	37.772014
student	46	47	17	18	25.345753	27.700980
technician	70	66	20	20	38.006901	39.610161
unemployed	66	66	21	21	39.593548	39.954315
unknown	81	76	24	25	46.880952	44.197531

oldest client by job and means of contact (cellular versus telephone.

youngest client by job and means of contact (cellular versus telephone.

average age of clients by job and means of contact (cellular versus telephone.

pd.merge()

Merge DataFrame or named Series objects with a database-style join

It takes a number of arguments:

- **right**: The Object to merge with;
- **how**: Type of merge to be performed (left, right, outer, etc)
- **on**: Column or index level names to join on.
- **left_on**: Column or index level names to join on in the left DataFrame.
- **right_on**: Column or index level names to join on in the right DataFrame.

A DataFrame of the two merged objects

df1

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

df2

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

```
df3 = pd.merge(df1, df2)
df3
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

pd.merge()

Merge DataFrame or named Series objects with a database-style join

It takes a number of arguments:

- **right**: The Object to merge with;
- **how**: Type of merge to be performed (left, right, outer, etc)
- **on**: Column or index level names to join on.
- **left_on**: Column or index level names to join on in the left DataFrame.
- **right_on**: Column or index level names to join on in the right DataFrame.

A DataFrame of the two merged objects

Option	Behavior
`how="inner"`	Use only the key combinations observed in both tables
`how="left"`	Use all key combinations found in the left table
`how="right"`	Use all key combinations found in the right table
`how="outer"`	Use all key combinations observed in both tables together

df1

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

df2

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

```
df3 = pd.merge(df1, df2)
df3
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

DataFrame.query()

query the columns of a DataFrame with a boolean expression

It takes two main arguments:

- **The query string to evaluate**;
- **inplace**: True or False

Returns Null (inplace = True) or a DataFrame (inlace = False) filtered according to the query expression.

```
df = pd.read_csv("portugal_houses.csv")
df.head()
```

	id	title	price_int	category	started_at	location	region	latitude	longitude	area.gross	area.net	area.land
0	85721	Apartamento T2 em Benfica	299900	casas	2022-03-25 14:19:11+00:00	Lisboa	Benfica	38.750977	-9.205208	70.00 m²	63.00 m²	NaN
1	105998	Apartamento T4 em Canidelo	600000	casas	2022-02-08 12:53:29+00:00	Vila Nova de Gaia	Canidelo	41.123177	-8.642253	173.70 m²	162.50 m²	NaN
2	117727	Apartamento T3 em Laranjeiro e Feijó	245000	casas	2022-07-05 13:31:11+00:00	Almada	Laranjeiro e Feijó	38.657093	-9.155763	NaN	74.00 m²	NaN
3	105770	Moradia T4 em Albufeira e Olhos de Água	1295000	casas	2022-06-22 16:21:10+00:00	Albufeira	Albufeira e Olhos de Água	37.088085	-8.252891	383.00 m²	NaN	578.00 m²
4	22240	Moradia T5 em Samora Correia	400000	casas	2022-03-31 05:45:12+00:00	Benavente	Samora Correia	38.914267	-8.847097	5331.00 m2	200.00 m2	NaN

DataFrame.query()

query the columns of a DataFrame with a boolean expression

It takes two main arguments:

- **The query string to evaluate**;
- **inplace**: True or False

Returns Null (inplace = True) or a DataFrame (inlace = False) filtered according to the query expression.

```
df.query("price_int < 150000").head()
```

	id	title	price_int	category	started_at	location	region	latitude	longitude	area.gross	area.net	area.land
13	38518	Moradia T3 em São João Baptista e Santa Maria ...	125000	casas	2021-07-27 02:32:36+00:00	Tomar	São João Baptista e Santa Maria dos Olivais	39.604523	-8.413719	NaN	259.00 m²	4480.00 m²
26	122398	Apartamento T1 em Barreiro e Lavradio	99000	casas	2022-06-09 05:23:40+00:00	Barreiro	Barreiro e Lavradio	38.676457	-9.050631	54.00 m2	54.00 m2	54.00 m2
43	10097	Moradia T3 em Jarmelo São Miguel	29000	casas	2022-03-18 02:17:10+00:00	Guarda	Jarmelo São Miguel	40.622221	-7.142540	108.00 m2	108.00 m2	NaN
44	109975	Moradia T2 em São Vicente do Paul e Vale de Fi...	60000	casas	2021-07-01 18:21:23+00:00	Santarém	São Vicente do Paul e Vale de Figueira	39.350067	-8.621095	NaN	NaN	5394.40 m²
56	32527	Apartamento T2 em Cascais e Estoril	2950	casas	2022-06-15 19:09:10+00:00	Cascais	Cascais e Estoril	38.698246	-9.422469	166.90 m²	166.90 m²	NaN

DataFrame.query()

query the columns of a DataFrame with a boolean expression

It takes two main arguments:

- **The query string to evaluate**;
- **inplace**: True or False

Returns Null (inplace = True) or a DataFrame (inlace = False) filtered according to the query expression.

```
df.query('price_int < 150000 and location in ["Lisboa", "Almada", "Sintra"]').head()
```

	id	title	price_int	category	started_at	location	region	latitude	longitude	area.gross	area.net	area.land
242	118486	Apartamento T2 em Almada, Cova da Piedade, Pra...	132500	casas	2021-10-08 18:35:15+00:00	Almada	Almada, Cova da Piedade, Pragal e Cacilhas	38.682293	-9.159185	NaN	55.00 m²	60.00 m²
244	33152	Apartamento T2 em Estrela	1200	casas	2021-07-06 10:57:10+00:00	Lisboa	Estrela	38.706192	-9.167079	85.00 m²	80.00 m²	NaN
337	91365	Apartamento T1 em Almada, Cova da Piedade, Pra...	105000	casas	2022-07-01 16:59:49+00:00	Almada	Almada, Cova da Piedade, Pragal e Cacilhas	38.682293	-9.159185	NaN	45.00 m²	NaN
446	32600	Apartamento T3 em Algueirão-Mem Martins	980	casas	2022-04-29 15:45:11+00:00	Sintra	Algueirão-Mem Martins	38.792966	-9.349868	159.45 m²	128.30 m²	865.00 m²
518	32199	Apartamento T1 para arrendamento mobilado na M...	1700	casas	2022-06-07 08:59:11+00:00	Lisboa	Misericórdia	38.711887	-9.144074	NaN	NaN	NaN

DataFrame.query()

query the columns of a DataFrame with a boolean expression

It takes two main arguments:

- **The query string to evaluate**;
- **inplace**: True or False

Returns Null (inplace = True) or a DataFrame (inlace = False) filtered according to the query expression.

```
subset = ["Lisboa", "Almada", "Sintra"]
df.query('price_int < 150000 and location in @subset').head()
```

	id	title	price_int	category	started_at	location	region	latitude	longitude	area.gross	area.net	area.land
242	118486	Apartamento T2 em Almada, Cova da Piedade, Pra...	132500	casas	2021-10-08 18:35:15+00:00	Almada	Almada, Cova da Piedade, Pragal e Cacilhas	38.682293	-9.159185	NaN	55.00 m²	60.00 m²
244	33152	Apartamento T2 em Estrela	1200	casas	2021-07-06 10:57:10+00:00	Lisboa	Estrela	38.706192	-9.167079	85.00 m²	80.00 m²	NaN
337	91365	Apartamento T1 em Almada, Cova da Piedade, Pra...	105000	casas	2022-07-01 16:59:49+00:00	Almada	Almada, Cova da Piedade, Pragal e Cacilhas	38.682293	-9.159185	NaN	45.00 m²	NaN
446	32600	Apartamento T3 em Algueirão-Mem Martins	980	casas	2022-04-29 15:45:11+00:00	Sintra	Algueirão-Mem Martins	38.792966	-9.349868	159.45 m²	128.30 m²	865.00 m²
518	32199	Apartamento T1 para arrendamento mobilado na M...	1700	casas	2022-06-07 08:59:11+00:00	Lisboa	Misericórdia	38.711887	-9.144074	NaN	NaN	NaN

DataFrame.query()

query the columns of a DataFrame with a boolean expression

It takes two main arguments:

- **The query string to evaluate**;
- **inplace**: True or False

Returns Null (inplace = True) or a DataFrame (inlace = False) filtered according to the query expression.

Query is not more efficient than .loc[], but can drastically simplify complex filtering operations and result in a more readable code.

```
subset = ["Lisboa", "Almada", "Sintra"]
df.query('price_int < 150000 and location in @subset').head()
```

	id	title	price_int	category	started_at	location	region	latitude	longitude	area.gross	area.net	area.land
242	118486	Apartamento T2 em Almada, Cova da Piedade, Pra...	132500	casas	2021-10-08 18:35:15+00:00	Almada	Almada, Cova da Piedade, Pragal e Cacilhas	38.682293	-9.159185	NaN	55.00 m²	60.00 m²
244	33152	Apartamento T2 em Estrela	1200	casas	2021-07-06 10:57:10+00:00	Lisboa	Estrela	38.706192	-9.167079	85.00 m²	80.00 m²	NaN
337	91365	Apartamento T1 em Almada, Cova da Piedade, Pra...	105000	casas	2022-07-01 16:59:49+00:00	Almada	Almada, Cova da Piedade, Pragal e Cacilhas	38.682293	-9.159185	NaN	45.00 m²	NaN
446	32600	Apartamento T3 em Algueirão-Mem Martins	980	casas	2022-04-29 15:45:11+00:00	Sintra	Algueirão-Mem Martins	38.792966	-9.349868	159.45 m²	128.30 m²	865.00 m²
518	32199	Apartamento T1 para arrendamento mobilado na M...	1700	casas	2022-06-07 08:59:11+00:00	Lisboa	Misericórdia	38.711887	-9.144074	NaN	NaN	NaN

DataFrame.query()

query the columns of a DataFrame with a boolean expression

It takes two main arguments:

- **The query string to evaluate**;
- **inplace**: True or False

Returns Null (inplace = True) or a DataFrame (inlace = False) filtered according to the query expression.

```
100*df.query("`area.net` > `area.gross`").shape[0]/df.shape[0]
```

```
4.6316773088304
```

```
df = df.dropna().query('`area.gross` != "True" and `area.net` != "True"')
```

```
df['area.net'] = df['area.net'].map(lambda x: x.split()[0]).astype(float)
df['area.gross'] = df['area.gross'].map(lambda x: x.split()[0]).astype(float)
```

```
df.query("`area.net` > `area.gross`").head()
```

	id	title	price_int	category	started_at	location	region	latitude	longitude	area.gross	area.net	area.land
344	13043	Moradia T5 em Sé e São Lourenço	120000	casas	2021-12-08 18:07:10+00:00	Portalegre	Sé e São Lourenço	39.297873	-7.432020	247.00	631.0	292.00 m2
461	43582	Apartamento T2 em Agualva e Mira-Sintra	190000	casas	2022-07-05 08:52:11+00:00	Sintra	Agualva e Mira-Sintra	38.771621	-9.296370	73.25	86.0	213.25 m²
478	59018	Quintas e casas rústicas em Paredes	195000	casas	2021-07-27 16:07:37+00:00	Paredes	Paredes	41.220874	-8.334223	281.00	282.0	2371.00 m²
717	19997	Apartamento T1 em Funchal (Sé)	219000	casas	2021-10-01 13:27:34+00:00	Funchal	Funchal (Sé)	32.646985	-16.914026	70.00	88.0	88.00 m2

Dummy Variables

Often we need to convert a categorical variable into a dummy indicator matrix.
Suppose we have the following ages:

```
df = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],"data1": range(6)})
```

```
df
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

```
pd.get_dummies(df['key'],dtype=int)
```

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

In some cases, you may want to add a prefix to the columns in the indicator DataFrame, which can then be merged with the other data. pandas.get_dummies has a prefix argument for doing this

```
dummies = pd.get_dummies(df["key"], prefix="key", dtype=float)
df_with_dummy = df[["data1"]].join(dummies)
df_with_dummy
```

	data1	key_a	key_b	key_c
0	0	0.0	1.0	0.0
1	1	0.0	1.0	0.0
2	2	1.0	0.0	0.0

Dummy Variables

Often we need to convert a categorical variable into a dummy indicator matrix.
Suppose we have the following ages:

```
df = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],"data1": range(6)})
```

```
df
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

```
pd.get_dummies(df['key'],dtype=int)
```

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

In some cases, you may want to add a prefix to the columns in the indicator DataFrame, which can then be merged with the other data. pandas.get_dummies has a prefix argument for doing this

```
dummies = pd.get_dummies(df["key"], prefix="key", dtype=float)
df_with_dummy = df[["data1"]].join(dummies)
df_with_dummy
```

	data1	key_a	key_b	key_c
0	0	0.0	1.0	0.0
1	1	0.0	1.0	0.0
2	2	1.0	0.0	0.0

there are a lot more to pandas

Rank US states and territories by their 2010 population density

```
pop = pd.read_csv("https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-population.csv")
areas = pd.read_csv('https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-areas.csv')
abbrevs = pd.read_csv('https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-abbrevs.csv')
```

```
pop.head()
```

	state/region	ages	year	population
0	AL	under18	2012	1117489.0
1	AL	total	2012	4817528.0
2	AL	under18	2010	1130966.0
3	AL	total	2010	4785570.0
4	AL	under18	2011	1125763.0

	state	abbreviation
0	Alabama	AL
1	Alaska	AK
2	Arizona	AZ
3	Arkansas	AR
4	California	CA

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
4	California	163707

We have all the data that we need, but requires a bit of work around.

Rank US states and territories by their 2010 population density

pop

	state/region	ages	year	population
0	AL	under18	2012	1117489.0
1	AL	total	2012	4817528.0
2	AL	under18	2010	1130966.0
3	AL	total	2010	4785570.0
4	AL	under18	2011	1125763.0

abbrevs

	state	abbreviation
0	Alabama	AL
1	Alaska	AK
2	Arizona	AZ
3	Arkansas	AR
4	California	CA

areas

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
4	California	163707

first step merge pop with abbrevs to obtain the full state name (one-to-many join).

```
merged = pd.merge(pop, abbrevs, how='outer', left_on='state/region', right_on='abbreviation')
merged = merged.drop('abbreviation', 1) # drop duplicate info
merged.head()
```

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

Rank US states and territories by their 2010 population density

pop

	state/region	ages	year	population
0	AL	under18	2012	1117489.0
1	AL	total	2012	4817528.0
2	AL	under18	2010	1130966.0
3	AL	total	2010	4785570.0
4	AL	under18	2011	1125763.0

abbrevs

	state	abbreviation
0	Alabama	AL
1	Alaska	AK
2	Arizona	AZ
3	Arkansas	AR
4	California	CA

areas

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
4	California	163707

first step merge pop with abbrevs to obtain the full state name (one-to-many join).

```
merged = pd.merge(pop, abbrevs, how='outer', left_on='state/region', right_on='abbreviation')
merged = merged.drop('abbreviation', 1) # drop duplicate info
merged.head()
```

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

mayday mayday, we have problems

```
merged.isnull().any()
```

```
state/region    False
ages            False
year            False
population       True
state            True
dtype: bool
```

some states are not reported

```
merged[merged['population'].isnull()].head()
```

	state/region	ages	year	population	state
2448	PR	under18	1990	NaN	NaN
2449	PR	total	1990	NaN	NaN
2450	PR	total	1991	NaN	NaN
2451	PR	under18	1991	NaN	NaN
2452	PR	total	1993	NaN	NaN

Rank US states and territories by their 2010 population density

pop

	state/region	ages	year	population
0	AL	under18	2012	1117489.0
1	AL	total	2012	4817528.0
2	AL	under18	2010	1130966.0
3	AL	total	2010	4785570.0
4	AL	under18	2011	1125763.0

abbrevs

	state	abbreviation
0	Alabama	AL
1	Alaska	AK
2	Arizona	AZ
3	Arkansas	AR
4	California	CA

areas

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
4	California	163707

first step merge pop with abbrevs to obtain the full state name (one-to-many join).

```
merged = pd.merge(pop, abbrevs, how='outer', left_on='state/region', right_on='abbreviation')
merged = merged.drop('abbreviation', 1) # drop duplicate info
merged.head()
```

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

which states?

```
list(merged.loc[merged['state'].isnull(), 'state/region'].unique())
```

```
['PR', 'USA']
```

Puerto Rico and it seems there is an ID for the entire US states

can we fix this? Well we can correct the column “state” but we won’t be able to get the “population” column, so let it be.

Rank US states and territories by their 2010 population density

merged

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

abbrevs

areas

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
4	California	163707

second step we can merge the merged of pop and abbrevs with the areas.

```
final = pd.merge(merged, areas, on='state', how='left')
final.head()
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Rank US states and territories by their 2010 population density

merged

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

abbrevs

areas

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
4	California	163707

second step we can merge the merged of pop and abbrevs with the areas.

```
final = pd.merge(merged, areas, on='state', how='left')
final.head()
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

```
final.isnull().any()
```

```
state/region     False
ages             False
year             False
population        True
state             True
area (sq. mi)     True
dtype: bool
```

there are NaN records, let's drop them

```
final.dropna(inplace=True)
final.head()
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Rank US states and territories by their 2010 population density

third step the metric system is amazing

1 sq mi = 2.58999 sq km

```
final['area (sq. km)'] = final['area (sq. mi)']*2.5899
final.head()
```

	state/region	ages	year	population	state	area (sq. mi)	area (sq. km)
0	AL	under18	2012	1117489.0	Alabama	52423.0	135770.3277
1	AL	total	2012	4817528.0	Alabama	52423.0	135770.3277
2	AL	under18	2010	1130966.0	Alabama	52423.0	135770.3277
3	AL	total	2010	4785570.0	Alabama	52423.0	135770.3277
4	AL	under18	2011	1125763.0	Alabama	52423.0	135770.3277

Rank US states and territories by their 2010 population density

third step the metric system is amazing

1 sq mi = 2.58999 sq km

```
final['area (sq. km)'] = final['area (sq. mi)']*2.5899
final.head()
```

	state/region	ages	year	population	state	area (sq. mi)	area (sq. km)
0	AL	under18	2012	1117489.0	Alabama	52423.0	135770.3277
1	AL	total	2012	4817528.0	Alabama	52423.0	135770.3277
2	AL	under18	2010	1130966.0	Alabama	52423.0	135770.3277
3	AL	total	2010	4785570.0	Alabama	52423.0	135770.3277
4	AL	under18	2011	1125763.0	Alabama	52423.0	135770.3277

fourth step select the records of interest

```
data2010 = final.query("year == 2010 & ages == 'total'")
data2010.head()
```

	state/region	ages	year	population	state	area (sq. mi)	area (sq. km)
3	AL	total	2010	4785570.0	Alabama	52423.0	1.357703e+05
91	AK	total	2010	713868.0	Alaska	656425.0	1.700075e+06
101	AZ	total	2010	6408790.0	Arizona	114006.0	2.952641e+05
189	AR	total	2010	2922280.0	Arkansas	53182.0	1.377361e+05
197	CA	total	2010	37333601.0	California	163707.0	4.239848e+05

.query() it is a faster way of doing it, since it uses routines compiled in C

Rank US states and territories by their 2010 population density

fifth step estimate population density, sort values, and extract the top 5 with .head()

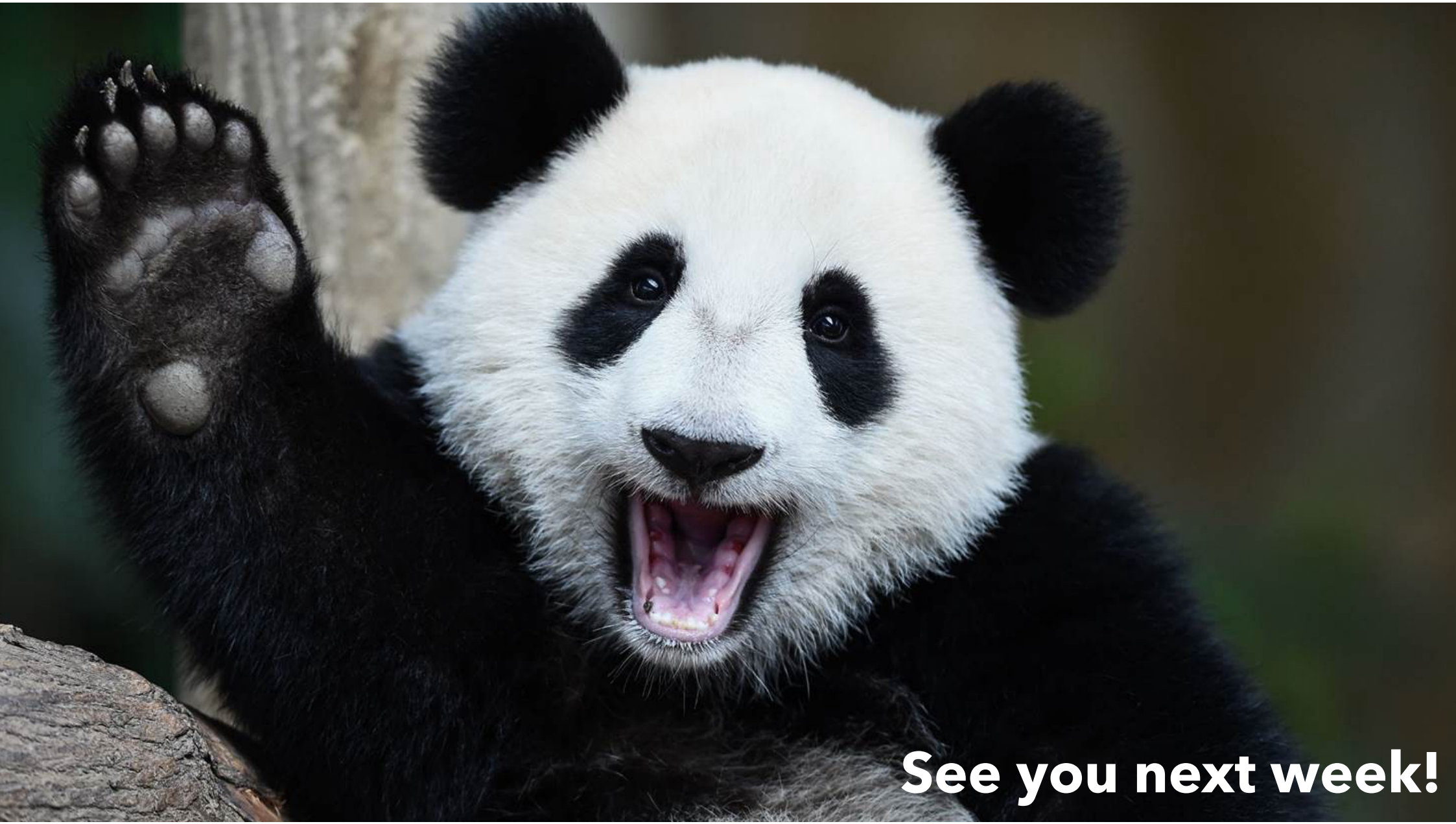
```
data2010.set_index('state', inplace=True)
density = data2010['population'] / data2010['area (sq. km)']
density.sort_values(ascending=False, inplace=True)
density.head()
```

```
state
District of Columbia    3436.000254
Puerto Rico              408.766805
New Jersey               389.688122
Rhode Island             263.075470
Connecticut              249.276285
dtype: float64
```

or bottom 5 with .tail()

```
density.tail()
```

```
state
South Dakota    4.086456
North Dakota    3.682600
Montana         2.600939
Wyoming         2.227144
Alaska          0.419904
dtype: float64
```


See you next week!