

 master ▾[pandas-videos / 21\\_more\\_pandas\\_tricks.ipynb](#)[Go to file](#)

...



justmarkham improve deprecation notes

Latest commit 2f7d40d on May 17  1 contributor

7139 lines (7139 sloc) | 198 KB

[Raw](#)[Blame](#)

▼

# Data School: 21 more pandas tricks ([video](#))

**See also:** "My top 25 pandas tricks" [video](#) and [notebook](#)

- Watch the [complete pandas video series](#)
- Connect on [Twitter](#), [Facebook](#), and [LinkedIn](#)
- Subscribe on [YouTube](#)
- Join the [email newsletter](#)

## Table of contents

1. [Check for equality](#)
2. [Check for equality \(alternative\)](#)
3. [Use NumPy without importing NumPy](#)
4. [Calculate memory usage](#)
5. [Count the number of words in a column](#)
6. [Convert one set of values to another](#)
7. [Convert continuous data into categorical data \(alternative\)](#)
8. [Create a cross-tabulation](#)
9. [Create a datetime column from multiple columns](#)
10. [Resample a datetime column](#)
11. [Read and write from compressed files](#)
12. [Fill missing values using interpolation](#)
13. [Check for duplicate merge keys](#)
14. [Transpose a wide DataFrame](#)
15. [Create an example DataFrame \(alternative\)](#)
16. [Identify rows that are missing from a DataFrame](#)
17. [Use query to avoid intermediate variables](#)
18. [Reshape a DataFrame from wide format to long format](#)
19. [Reverse row order \(alternative\)](#)
20. [Reverse column order \(alternative\)](#)
21. [Split a string into multiple columns \(alternative\)](#)

```
In [1]:  
import pandas as pd  
import numpy as np  
drinks = pd.read_csv('http://bit.ly/drinksbycountry')  
stocks = pd.read_csv('http://bit.ly/smallstocks', parse_dates=['Date'])  
titanic = pd.read_csv('http://bit.ly/kaggletrain')  
ufo = pd.read_csv('http://bit.ly/uforeports', parse_dates=['Time'])
```

## 1. Check for equality

Let's create an example DataFrame:

```
In [2]:  
df = pd.DataFrame({'a':[1, 2, np.nan], 'b':[1, 2, np.nan]})  
df
```

```
Out[2]:  
      a    b  
0  1.0  1.0  
1  2.0  2.0  
2  NaN  NaN
```

Do you ever have two DataFrame columns that look similar, and you want to know if they are actually identical?

This is not a reliable method for checking:

```
In [3]:  
df.a == df.b
```

```
Out[3]:  
0    True  
1    True  
2   False  
dtype: bool
```

You would think that would return 3 `True` values, but it actually returns `False` any time there is a missing value:

```
In [4]:  
np.nan == np.nan
```

```
Out[4]: False
```

Instead, you can check for equality using the `equals()` method:

```
In [5]: df.a.equals(df.b)
```

```
Out[5]: True
```

Similarly, this is how you would check if two DataFrames are identical:

```
In [6]: df_new = df.copy()
df_new.equals(df)
```

```
Out[6]: True
```

We made a copy of "df" and then used the DataFrame `equals()` method.

## 2. Check for equality (alternative)

Let's create another example DataFrame:

```
In [7]: df = pd.DataFrame({'c':[1, 2, 3], 'd':[1.0, 2.0, 3.0], 'e':[1.0, 2.0, 3.000005]})
```

```
Out[7]:   c    d        e
0  1  1.0  1.000000
1  2  2.0  2.000000
2  3  3.0  3.000005
```

It's important to note that the `equals()` method (shown in the first trick) requires identical data types in order to return `True`:

```
In [8]: df.c.equals(df.d)
```

```
Out[8]: False
```

This returned `False` because "c" is integer and "d" is float.

For more flexibility in how the equality checking is done, use the `assert_series_equal()` function:

```
In [9]: pd.testing.assert_series_equal(df.c, df.d, check_names=False, check_dtype=False)
```

The assertion passed (thus no error was raised) because we specified that data type can be ignored.

As well, you can check whether values are approximately equal, rather than identical:

```
In [10]: pd.testing.assert_series_equal(df.d, df.e, check_names=False, check_exact=False)
```

The assertion passed even though "d" and "e" have slightly different values.

For checking DataFrames, there's a similar function called `assert_frame_equal()`:

```
In [11]: df_new = df.copy()
pd.testing.assert_frame_equal(df, df_new)
```

### 3. Use NumPy without importing NumPy

Although pandas is mostly a superset of NumPy's functionality, there are occasions on which you still have to import NumPy. One example is if you want to create a DataFrame of random values:

```
In [12]: np.random.seed(0)
pd.DataFrame(np.random.rand(2, 4))
```

```
Out[12]:      0      1      2      3
0  0.548814  0.715189  0.602763  0.544883
1  0.423655  0.645894  0.437587  0.891773
```

However, it turns out that you can actually access all of NumPy's functionality from within pandas, simply by typing `pd.np.` before the NumPy function name:

```
In [13]: # NOTE: pd.np is deprecated in pandas 1.0
pd.np.random.seed(0)
pd.DataFrame(pd.np.random.rand(2, 4))
```

```
/var/folders/zn/8p5wr_855bjbd9s6fvn6wd7w0000gn/T/ipykernel_39811/2633834213.py:2: FutureWarning: The pandas.np module is deprecated and will be removed from pandas in a future version. Import numpy directly instead.  
    pd.np.random.seed(0)  
/var/folders/zn/8p5wr_855bjbd9s6fvn6wd7w0000gn/T/ipykernel_39811/2633834213.py:3: FutureWarning: The pandas.np module is deprecated and will be removed from pandas in a future version. Import numpy directly instead.  
    pd.DataFrame(pd.np.random.rand(2, 4))
```

Out[13]:

	0	1	2	3
0	0.548814	0.715189	0.602763	0.544883
1	0.423655	0.645894	0.437587	0.891773

To be clear, this would have worked even if we had not explicitly imported NumPy at the start of the notebook.

This could also be used to set a value as missing:

```
In [14]:  
# NOTE: pd.np is deprecated in pandas 1.0  
df.loc[0, 'e'] = pd.np.nan  
df
```

```
/var/folders/zn/8p5wr_855bjbd9s6fvn6wd7w0000gn/T/ipykernel_39811/4135636971.py:1: FutureWarning: The pandas.np module is deprecated and will be removed from pandas in a future version. Import numpy directly instead.  
    df.loc[0, 'e'] = pd.np.nan
```

Out[14]:

	c	d	e
0	1	1.0	NaN
1	2	2.0	2.000000
2	3	3.0	3.000005

That being said, I would still recommend following the convention of `import numpy as np` rather than using `pd.np` since that convention is so widespread.

## 4. Calculate memory usage

Here's a DataFrame of UFO sightings:

```
In [15]: ufo.head()
```

```
Out[15]:
```

	City	Colors Reported	Shape Reported	State	Time
0	Ithaca	NaN	TRIANGLE	NY	1930-06-01 22:00:00
1	Willingboro	NaN	OTHER	NJ	1930-06-30 20:00:00
2	Holyoke	NaN	OVAL	CO	1931-02-15 14:00:00
3	Abilene	NaN	DISK	KS	1931-06-01 13:00:00
4	New York Worlds Fair	NaN	LIGHT	NY	1933-04-18 19:00:00

You can calculate the memory used by the entire DataFrame:

```
In [16]: ufo.info(memory_usage='deep')
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18241 entries, 0 to 18240
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype  
 --- 
 0   City              18216 non-null   object  
 1   Colors Reported   2882 non-null   object  
 2   Shape Reported    15597 non-null   object  
 3   State             18241 non-null   object  
 4   Time              18241 non-null   datetime64[ns]
dtypes: datetime64[ns](1), object(4)
memory usage: 4.0 MB
```

You can also calculate memory used by each column (in bytes):

```
In [17]: ufo.memory_usage(deep=True)
```

```
Out[17]:
```

Index	128
City	1205618
Colors Reported	671313
Shape Reported	1065230
State	1076219
Time	145928
dtype:	int64

This information might help you to decide how to optimize your DataFrame storage.

## 5. Count the number of words in a column

Let's count the values in this column from the "ufo" DataFrame:

```
In [18]: ufo['Colors Reported'].value_counts()
```

```
Out[18]:
```

RED	780
GREEN	531
ORANGE	528
BLUE	450
YELLOW	169
RED GREEN	89
RED BLUE	78
RED ORANGE	44
GREEN BLUE	34
RED GREEN BLUE	33
ORANGE YELLOW	26
RED YELLOW	25
ORANGE GREEN	23
YELLOW GREEN	17
ORANGE BLUE	10
RED YELLOW GREEN	9
YELLOW BLUE	6
YELLOW GREEN BLUE	5
ORANGE GREEN BLUE	5
RED YELLOW GREEN BLUE	4
RED ORANGE YELLOW	4
RED YELLOW BLUE	3
RED ORANGE GREEN	3
RED ORANGE BLUE	3
RED ORANGE YELLOW BLUE	1
ORANGE YELLOW GREEN	1
ORANGE YELLOW BLUE	1

Name: Colors Reported, dtype: int64

Notice that many of the entries have multiple colors. What if all we cared about was the number of colors, and not the colors themselves?

We can count the colors by using a string method to count the number of spaces, and then add 1:

```
In [19]: (ufo['Colors Reported'].str.count(' ') + 1).value_counts()
```

```
Out[19]:
```

1.0	2458
2.0	252

```
2.0      352  
3.0       67  
4.0        5  
Name: Colors Reported, dtype: int64
```

## 6. Convert one set of values to another

Using the Titanic dataset as an example, I'm going to highlight three different ways that you can convert one set of values to another.

Let's start with the "Sex" column:

```
In [20]: titanic.Sex.head()
```

```
Out[20]: 0     male  
1   female  
2   female  
3   female  
4     male  
Name: Sex, dtype: object
```

There are two different values in this column. If you need to convert these values to 0 and 1, you can use the `map()` method and pass it a dictionary specifying how you want to map the values:

```
In [21]: titanic['Sex_num'] = titanic.Sex.map({'male':0, 'female':1})  
titanic.Sex_num.head()
```

```
Out[21]: 0    0  
1    1  
2    1  
3    1  
4    0  
Name: Sex_num, dtype: int64
```

As we specified, "male" has become 0 and "female" has become 1.

Next, let's look at the "Embarked" column:

```
In [22]: titanic.Embarked.head(10)
```

```
Out[22]: 0    S  
1    C  
2    S
```

```
3 S  
4 S  
5 Q  
6 S  
7 S  
8 S  
9 C  
Name: Embarked, dtype: object
```

There are three different values in this column: S, C, and Q. If you need to convert them to 0, 1, and 2, you could use the `map()` method, but the `factorize()` method is even easier:

```
In [23]: titanic['Embarked_num'] = titanic.Embarked.factorize()[0]  
titanic.Embarked_num.head(10)
```

```
Out[23]: 0    0  
1    1  
2    0  
3    0  
4    0  
5    2  
6    0  
7    0  
8    0  
9    1  
Name: Embarked_num, dtype: int64
```

`factorize()` returns a tuple in which the first element contains the new values, which is why I had to use `[0]` to extract the values.

You can see that "S" has become 0, "C" has become 1, and "Q" has become 2. It chose that mapping based on the order in which the values appear in the Series, and if you need to reference the mapping, it's stored in the second value in the tuple:

```
In [24]: titanic.Embarked.factorize()[1]
```

```
Out[24]: Index(['S', 'C', 'Q'], dtype='object')
```

Finally, let's look at the "SibSp" column:

```
In [25]: titanic.SibSp.head(10)
```

```
Out[25]: 0    1
```

```
1 1  
2 0  
3 1  
4 0  
5 0  
6 0  
7 3  
8 0  
9 1  
Name: SibSp, dtype: int64
```

Let's say that you needed to keep the zeros as-is and convert all other values to one. You can express this as a condition, `SibSp > 0`, which will return a boolean Series that you can convert to integers using the `astype()` method:

```
In [26]: titanic['SibSp_binary'] = (titanic.SibSp > 0).astype('int')  
titanic.SibSp_binary.head(10)
```

```
Out[26]: 0 1  
1 1  
2 0  
3 1  
4 0  
5 0  
6 0  
7 1  
8 0  
9 1  
Name: SibSp_binary, dtype: int64
```

Notice that the only value greater than 1 has been converted to a 1.

## 7. Convert continuous data into categorical data (alternative)

In the main tricks video, I used the `cut()` function to convert the "Age" column from continuous to categorical data:

```
In [27]: pd.cut(titanic.Age, bins=[0, 18, 25, 99], labels=['child', 'young adult', 'adult']).head(10)
```

```
Out[27]: 0  young adult  
1      adult  
2      adult
```

```
3      adult
4      adult
5      NaN
6      adult
7      child
8      adult
9      child
Name: Age, dtype: category
Categories (3, object): ['child' < 'young adult' < 'adult']
```

When using `cut()`, we had to choose the edges of each bin. But if you want pandas to choose the bin edges for you, you can use the `qcut()` function instead:

```
In [28]: pd.qcut(titanic.Age, q=3).head(10)
```

```
Out[28]: 0    (0.419, 23.0]
1    (34.0, 80.0]
2    (23.0, 34.0]
3    (34.0, 80.0]
4    (34.0, 80.0]
5        NaN
6    (34.0, 80.0]
7    (0.419, 23.0]
8    (23.0, 34.0]
9    (0.419, 23.0]

Name: Age, dtype: category
Categories (3, interval[float64, right]): [(0.419, 23.0] < (23.0, 34.0] < (34.0, 80.0)]
```

We told `qcut()` to create 3 bins, and it chose bin edges that would result in bins of approximately equal size:

```
In [29]: pd.qcut(titanic.Age, q=3).value_counts()
```

```
Out[29]: (0.419, 23.0]    246
(34.0, 80.0]    236
(23.0, 34.0]    232
Name: Age, dtype: int64
```

As you can see, the three bins are ages 0 to 23, 23 to 34, and 34 to 80, and they all contain roughly the same number of observations.

## 8. Create a cross-tabulation

Sometimes you just want to count the number of observations in each category. If you're interested in

Sometimes you just want to count the number of observations in each category... if you're interested in a single column, you would use the `value_counts()` method:

```
In [30]: titanic.Sex.value_counts()
```

```
Out[30]: male    577  
female   314  
Name: Sex, dtype: int64
```

But if you want to count the number of observations that appear in each combination of categories, you would use the `crosstab()` function:

```
In [31]: pd.crosstab(titanic.Sex, titanic.Pclass)
```

```
Out[31]: Pclass    1    2    3  
  
          Sex  
          _____  
female  94   76  144  
male   122  108  347
```

Just like a pivot table, you can include row and column totals by setting `margins=True`:

```
In [32]: pd.crosstab(titanic.Sex, titanic.Pclass, margins=True)
```

```
Out[32]: Pclass    1    2    3   All  
  
          Sex  
          _____  
female  94   76  144  314  
male   122  108  347  577  
All    216  184  491  891
```

In fact, you can actually create this same table using the `pivot_table()` method with 'count' as the aggregation function:

```
In [33]: titanic.pivot_table(index='Sex', columns='Pclass', values='Survived',  
                           aggfunc='count', margins=True)
```

```
Out[33]: Pclass    1    2    3   All  
  
          Sex  
          _____
```

```
In[33]: pclass  1   2   3   All
```

Sex	
female	94 76 144 314
male	122 108 347 577
All	216 184 491 891

## 9. Create a datetime column from multiple columns

Let's create an example DataFrame:

```
In [34]: df = pd.DataFrame([[12, 25, 2019, 'christmas'], [11, 28, 2019, 'thanksgiving']],
                      columns=['month', 'day', 'year', 'holiday'])
df
```

```
Out[34]:    month  day   year      holiday
0         12    25  2019    christmas
1         11    28  2019  thanksgiving
```

You can create a new datetime column simply by passing the relevant columns to `pd.to_datetime()`:

```
In [35]: df['date'] = pd.to_datetime(df[['month', 'day', 'year']])
df
```

```
Out[35]:    month  day   year      holiday      date
0         12    25  2019    christmas  2019-12-25
1         11    28  2019  thanksgiving  2019-11-28
```

The new date column has a datetime data type:

```
In [36]: df.dtypes
```

```
Out[36]: month          int64
          day           int64
```

```
year          int64
holiday       object
date   datetime64[ns]
dtype: object
```

Keep in mind that you must include month, day, and year columns at a minimum, but you can also include hour, minute, and second.

## 10. Resample a datetime column

Let's take a look at the stocks dataset:

```
In [37]: stocks
```

```
Out[37]:
```

	Date	Close	Volume	Symbol
0	2016-10-03	31.50	14070500	CSCO
1	2016-10-03	112.52	21701800	AAPL
2	2016-10-03	57.42	19189500	MSFT
3	2016-10-04	113.00	29736800	AAPL
4	2016-10-04	57.24	20085900	MSFT
5	2016-10-04	31.35	18460400	CSCO
6	2016-10-05	57.64	16726400	MSFT
7	2016-10-05	31.59	11808600	CSCO
8	2016-10-05	113.05	21453100	AAPL

What if you wanted to calculate the mean closing price by day across all stocks? Use the `resample()` method:

```
In [38]: stocks.resample('D', on='Date').Close.mean()
```

```
Out[38]:
```

Date	Close
2016-10-03	67.146667
2016-10-04	67.196667
2016-10-05	67.426667

Freq: D, Name: Close, dtype: float64

You can think of resampling as a `groupby()` for datetime data, and in fact the structure of the command looks very similar to a `groupby()`. "D" specifies that the resampling frequency should be daily, and the "on" parameter specifies the column on which we're resampling.

If the datetime column is the index, you can skip the `on` parameter. For example, let's give the ufo DataFrame a DatetimeIndex:

```
In [39]: ufo = ufo.set_index('Time')  
ufo.head()
```

```
Out[39]:
```

	City	Colors Reported	Shape Reported	State
Time				
1930-06-01 22:00:00	Ithaca	NaN	TRIANGLE	NY
1930-06-30 20:00:00	Willingboro	NaN	OTHER	NJ
1931-02-15 14:00:00	Holyoke	NaN	OVAL	CO
1931-06-01 13:00:00	Abilene	NaN	DISK	KS
1933-04-18 19:00:00	New York Worlds Fair	NaN	LIGHT	NY

Now we can use `resample()` and it will automatically resample based on the index:

```
In [40]: ufo.resample('Y').State.count().tail()
```

```
Out[40]:
```

Time	Count
1996-12-31	851
1997-12-31	1237
1998-12-31	1743
1999-12-31	2774
2000-12-31	2635

Freq: A-DEC, Name: State, dtype: int64

That's the count of the number of UFO sightings by year.

We can calculate the count by month by changing the resampling frequency from "Y" to "M":

```
In [41]: ufo.resample('M').State.count().tail()
```

```
Out[41]:
```

Time	Count
2000-08-31	250

```
2000-09-30    257  
2000-10-31    278  
2000-11-30    200  
2000-12-31    192  
Freq: M, Name: State, dtype: int64
```

The string that you pass to `resample()` is known as the offset alias, and pandas supports [many offset aliases](#) other than just "D", "M", and "Y".

## 11. Read and write from compressed files

When you want to save a DataFrame to a CSV file, you use the `to_csv()` method:

```
In [42]: ufo.to_csv('ufo.csv')
```

However, you can actually compress the CSV file as well:

```
In [43]: ufo.to_csv('ufo.csv.zip')  
ufo.to_csv('ufo.csv.gz')  
ufo.to_csv('ufo.csv.bz2')  
ufo.to_csv('ufo.csv.xz')
```

By using one of these file extensions, pandas infers the type of compression you want it to use.

You can use a shell command to see all of the files we've created:

```
In [44]: !ls -l ufo.*
```

```
-rw-r--r--  1 kevin  staff  748033 Apr 27 14:27 ufo.csv  
-rw-r--r--  1 kevin  staff  129189 Apr 27 14:27 ufo.csv.bz2  
-rw-r--r--  1 kevin  staff  198030 Apr 27 14:27 ufo.csv.gz  
-rw-r--r--  1 kevin  staff  149320 Apr 27 14:27 ufo.csv.xz  
-rw-r--r--  1 kevin  staff  200317 Apr 27 14:27 ufo.csv.zip
```

You can see that all of the compressed files are significantly smaller than the uncompressed CSV file.

Finally, you can actually read directly from a compressed file using `read_csv()`:

```
In [45]: ufo_new = pd.read_csv('ufo.csv.gz', index_col='Time', parse_dates=['Time'])  
ufo_new.head()
```

Out[45]:

Time	City	Colors Reported	Shape Reported	State
1930-06-01 22:00:00	Ithaca	NaN	TRIANGLE	NY
1930-06-30 20:00:00	Willingboro	NaN	OTHER	NJ
1931-02-15 14:00:00	Holyoke	NaN	OVAL	CO
1931-06-01 13:00:00	Abilene	NaN	DISK	KS
1933-04-18 19:00:00	New York Worlds Fair	NaN	LIGHT	NY

And we can confirm that the new ufo DataFrame is equivalent to the original ufo DataFrame:

In [46]:

```
ufo_new.equals(ufo)
```

Out[46]:

True

## 12. Fill missing values using interpolation

Let's create an example time series DataFrame with some missing values:

In [47]:

```
df = pd.DataFrame({'a':[100, 120, 130, np.nan, 140], 'b':[9, 9, np.nan, 7.5, 6.5]})  
df.index = pd.to_datetime(['2019-01', '2019-02', '2019-03', '2019-04', '2019-05'])  
df
```

Out[47]:

	a	b
2019-01-01	100.0	9.0
2019-02-01	120.0	9.0
2019-03-01	130.0	NaN
2019-04-01	NaN	7.5
2019-05-01	140.0	6.5

If appropriate, you can fill in the missing values using interpolation:

In [48]:

```
df.interpolate()
```

```
Out[48]:
```

	a	b
2019-01-01	100.0	9.00
2019-02-01	120.0	9.00
2019-03-01	130.0	8.25
2019-04-01	135.0	7.50
2019-05-01	140.0	6.50

This uses linear interpolation by default, though other methods are supported.

## 13. Check for duplicate merge keys

Let's create two example DataFrames:

```
In [49]:
```

```
left = pd.DataFrame({'color': ['green', 'yellow', 'red'], 'num':[1, 2, 3]})  
left
```

```
Out[49]:
```

	color	num
0	green	1
1	yellow	2
2	red	3

```
In [50]:
```

```
right = pd.DataFrame({'color': ['green', 'yellow', 'pink', 'green'], 'size':['S', 'M', 'L']},  
right
```

```
Out[50]:
```

	color	size
0	green	S
1	yellow	M
2	pink	L
3	green	XL

We want to merge these DataFrames.

What if we wanted to confirm that the merge keys ("color" in this case) are unique in the left dataset?

We would use "one-to-many" validation:

In [51]:

```
pd.merge(left, right, how='inner', validate='one_to_many')
```

Out[51]:

	color	num	size
0	green	1	S
1	green	1	XL
2	yellow	2	M

It did the merge, and validated that the values of "color" in the left dataset are unique.

What if we wanted to confirm that the merge keys are unique in the right dataset? We would use "many-to-one" validation:

In [52]:

```
# pd.merge(left, right, how='inner', validate='many_to_one')
```

This resulted in an error, because the values of "color" in the right dataset are not unique.

## 14. Transpose a wide DataFrame

Let's create an example DataFrame with 200 rows and 25 columns:

In [53]:

```
df = pd.DataFrame(np.random.rand(200, 25), columns=list('ABCDEFGHIJKLMNPQRSTUVWXYZ'))
```

If you wanted to get a sense of the data by examining the head, you wouldn't see all of the columns due to the default display options:

In [54]:

```
df.head()
```

Out[54]:

	A	B	C	D	E	F	G	H	I	J	...
0	0.963663	0.383442	0.791725	0.528895	0.568045	0.925597	0.071036	0.087129	0.020218	0.832620	...

```
1 0.568434 0.018790 0.617635 0.612096 0.616934 0.943748 0.681820 0.359508 0.437032 0.697631 ... 0.1
2 0.466311 0.244426 0.158970 0.110375 0.656330 0.138183 0.196582 0.368725 0.820993 0.097101 ... 0.0
3 0.692472 0.566601 0.265389 0.523248 0.093941 0.575946 0.929296 0.318569 0.667410 0.131798 ... 0.8
4 0.223082 0.952749 0.447125 0.846409 0.699479 0.297437 0.813798 0.396506 0.881103 0.581273 ... 0.0
```

5 rows × 25 columns

The easiest solution is just to transpose the head:

```
In [55]: df.head().T
```

```
Out[55]:      0      1      2      3      4
A 0.963663 0.568434 0.466311 0.692472 0.223082
B 0.383442 0.018790 0.244426 0.566601 0.952749
C 0.791725 0.617635 0.158970 0.265389 0.447125
D 0.528895 0.612096 0.110375 0.523248 0.846409
E 0.568045 0.616934 0.656330 0.093941 0.699479
F 0.925597 0.943748 0.138183 0.575946 0.297437
G 0.071036 0.681820 0.196582 0.929296 0.813798
H 0.087129 0.359508 0.368725 0.318569 0.396506
I 0.020218 0.437032 0.820993 0.667410 0.881103
J 0.832620 0.697631 0.097101 0.131798 0.581273
K 0.778157 0.060225 0.837945 0.716327 0.881735
L 0.870012 0.666767 0.096098 0.289406 0.692532
M 0.978618 0.670638 0.976459 0.183191 0.725254
N 0.799159 0.210383 0.468651 0.586513 0.501324
O 0.461479 0.128926 0.976761 0.020108 0.956084
P 0.780529 0.315428 0.604846 0.828940 0.643990
Q 0.118274 0.363711 0.739264 0.004695 0.423855
R 0.630021 0.570107 0.020188 0.677817 0.606302
```

```
R  0.059921  0.370197  0.059100  0.077017  0.000595
S  0.143353  0.438602  0.282807  0.270008  0.019193
T  0.944669  0.988374  0.120197  0.735194  0.301575
U  0.521848  0.102045  0.296140  0.962189  0.660174
V  0.414662  0.208877  0.118728  0.248753  0.290078
W  0.264556  0.161310  0.317983  0.576157  0.618015
X  0.774234  0.653108  0.414263  0.592042  0.428769
Y  0.456150  0.253292  0.064147  0.572252  0.135474
```

Since the columns have become the rows (and vice versa), we can now easily browse through the DataFrame's head.

Transposing is also helpful when using the `describe()` method on a wide DataFrame:

```
In [56]: df.describe().T
```

```
Out[56]:   count      mean       std      min     25%     50%     75%      max
A    200.0  0.492993  0.297821  0.000367  0.232334  0.473208  0.752013  0.995733
B    200.0  0.511242  0.296146  0.004655  0.244792  0.502961  0.778799  0.995713
C    200.0  0.493275  0.265398  0.003866  0.271378  0.507585  0.699023  0.993405
D    200.0  0.510597  0.279209  0.005206  0.273734  0.539189  0.736733  0.999949
E    200.0  0.488265  0.286736  0.003710  0.264644  0.469948  0.729416  0.992820
F    200.0  0.486180  0.300410  0.011355  0.248337  0.455500  0.751475  0.999931
G    200.0  0.487767  0.285290  0.018173  0.219824  0.496847  0.743175  0.980979
H    200.0  0.469514  0.288366  0.002703  0.217834  0.439801  0.712790  0.985155
I    200.0  0.475042  0.284267  0.001962  0.229787  0.455727  0.710170  0.996100
J    200.0  0.486357  0.292939  0.003860  0.242252  0.451637  0.734957  0.983426
K    200.0  0.526301  0.297170  0.005495  0.255519  0.559120  0.798100  0.994496
L    200.0  0.508508  0.284417  0.005939  0.275634  0.503145  0.741509  0.999278
M    200.0  0.531511  0.283377  0.000664  0.312775  0.548991  0.765784  0.997962
N    200.0  0.492445  0.283717  0.000546  0.253779  0.477410  0.725116  0.990345
```

```
O 200.0 0.496102 0.289496 0.009060 0.259039 0.468176 0.725585 0.995830
P 200.0 0.515763 0.302013 0.004475 0.247396 0.504664 0.806546 0.997354
Q 200.0 0.499350 0.296964 0.000903 0.249951 0.524557 0.747713 0.999964
R 200.0 0.506541 0.283706 0.008187 0.258631 0.480919 0.786813 0.997994
S 200.0 0.509491 0.292244 0.006238 0.276656 0.500628 0.745958 0.998355
T 200.0 0.489958 0.276607 0.005052 0.273893 0.469294 0.733976 0.998023
U 200.0 0.502008 0.295426 0.005510 0.252521 0.513378 0.749146 0.997858
V 200.0 0.533049 0.298239 0.002084 0.274829 0.553971 0.806714 0.997046
W 200.0 0.461300 0.294328 0.000074 0.187038 0.429369 0.709391 0.975735
X 200.0 0.470272 0.291905 0.000072 0.219438 0.453254 0.717279 0.995813
Y 200.0 0.468069 0.284750 0.001383 0.244620 0.436273 0.706354 0.998199
```

## 15. Create an example DataFrame (alternative)

These are the methods that I taught in the main tricks video for creating example DataFrames:

```
In [57]: pd.DataFrame({'col one':[100, 200], 'col two':[300, 400]})
```

```
Out[57]:   col one  col two
0        100      300
1        200      400
```

```
In [58]: pd.DataFrame(np.random.rand(4, 8), columns=list('abcdefgh'))
```

```
Out[58]:       a      b      c      d      e      f      g      h
0  0.278510  0.288027  0.846305  0.791284  0.578636  0.288589  0.318878  0.592218
1  0.739867  0.384098  0.509562  0.888033  0.649791  0.535550  0.071222  0.176015
2  0.200992  0.623148  0.108113  0.028995  0.360351  0.718859  0.693249  0.792670
3  0.696248  0.613286  0.486162  0.208498  0.568548  0.636625  0.123743  0.565147
```

If you want an even simpler method, you can use `makeDataFrame()` to create a 30x4 DataFrame filled with random values:

In [59]:

```
# NOTE: pd.util.testing is deprecated in pandas 1.0
pd.util.testing.makeDataFrame().head()
```

```
/Users/kevin/miniconda3/envs/pd14/lib/python3.9/site-packages/pandas/util/_init_.py:15: FutureWarning: pandas.util.testing is deprecated. Use the functions in the public API at pandas.testing instead.
```

```
import pandas.util.testing
```

Out[59]:

	A	B	C	D
<b>EmWXi3UyK3</b>	-0.250656	-0.040365	-0.829203	0.296782
<b>jgi4rj56bK</b>	0.919812	-1.527617	-0.670679	-0.270356
<b>a6Po2BAIfh</b>	2.445794	-0.057607	-0.315385	0.736752
<b>Ys4pDMq9Yd</b>	1.205698	-2.490090	0.442820	1.157280
<b>XfLAIHy1QI</b>	0.920549	-0.026303	-1.052564	-0.658804

`makeMissingDataframe()` is similar, except that some of the values are missing:

In [60]:

```
# NOTE: pd.util.testing is deprecated in pandas 1.0
pd.util.testing.makeMissingDataframe().head()
```

Out[60]:

	A	B	C	D
<b>8JAZ6EeRof</b>	0.000341	0.537035	0.656464	-0.941815
<b>GaUe9v7uYV</b>	-0.062205	-0.286835	0.065851	-0.702835
<b>No3fmW5dDH</b>	-0.317349	1.634406	-0.873631	-1.378837
<b>W5LsjKWdWf</b>	NaN	0.571227	-1.057754	2.925250
<b>8ijrQMvxS0</b>	-0.211620	-0.789954	-0.897237	0.571752

`makeTimeDataFrame()` is similar, except it creates a DatetimeIndex:

In [61]:

```
# NOTE: pd.util.testing is deprecated in pandas 1.0
pd.util.testing.makeTimeDataFrame().head()
```

Out[61]:

	A	B	C	D
<b>2000-01-03</b>	0.470549	-0.981102	1.358382	0.064058
<b>2000-01-04</b>	-0.235804	0.165183	0.060856	-1.063780
<b>2000-01-05</b>	0.678789	-0.016646	2.005891	-0.079188
<b>2000-01-06</b>	1.153618	0.140933	0.081658	-0.527496
<b>2000-01-07</b>	-0.554733	-0.725172	-0.079209	0.460215

Finally, `makeMixedDataFrame()` creates this exact 5x4 DataFrame:

In [62]:

```
# NOTE: pd.util.testing is deprecated in pandas 1.0
pd.util.testing.makeMixedDataFrame()
```

Out[62]:

	A	B	C	D
<b>0</b>	0.0	0.0	foo1	2009-01-01
<b>1</b>	1.0	1.0	foo2	2009-01-02
<b>2</b>	2.0	0.0	foo3	2009-01-05
<b>3</b>	3.0	1.0	foo4	2009-01-06
<b>4</b>	4.0	0.0	foo5	2009-01-07

It has 2 float columns, 1 object column, and 1 datetime column.

There are many other similar functions that you can use:

In [63]:

```
# NOTE: pd.util.testing is deprecated in pandas 1.0
[x for x in dir(pd.util.testing) if x.startswith('make')]
```

Out[63]:

```
['makeBoolIndex',
 'makeCategoricalIndex',
 'makeCustomDataframe',
 'makeCustomIndex',
 'makeDataFrame',
 'makeDateIndex',
 'makeFloatIndex',
 'makeFloatSeries',
 'makeIntIndex',
 'makeIntervalIndex']
```

```
'makeIntervalIndex',
'makeMissingDataframe',
'makeMixedDataFrame',
'makeMultiIndex',
'makeNumericIndex',
'makeObjectSeries',
'makePeriodFrame',
'makePeriodIndex',
'makePeriodSeries',
'makeRangeIndex',
'makeStringIndex',
'makeStringSeries',
'makeTimeDataFrame',
'makeTimeSeries',
'makeTimedeltaIndex',
'makeUIntIndex',
'makeUnicodeIndex']
```

However, keep in mind that most of these have no arguments and no docstring, and none of them are listed in the pandas documentation.

## 16. Identify rows that are missing from a DataFrame

Let's create a small example DataFrame:

In [64]:

```
# NOTE: pd.util.testing is deprecated in pandas 1.0
df1 = pd.util.testing.makeMixedDataFrame()
df1
```

Out[64]:

	A	B	C	D
0	0.0	0.0	foo1	2009-01-01
1	1.0	1.0	foo2	2009-01-02
2	2.0	0.0	foo3	2009-01-05
3	3.0	1.0	foo4	2009-01-06
4	4.0	0.0	foo5	2009-01-07

Then let's create a copy of that DataFrame in which rows 2 and 3 are missing:

In [65]:

```
df2 = df1.drop([2, 3], axis='rows')
df2
```

```
Out[65]:   A   B   C       D
```

0	0.0	0.0	foo1	2009-01-01
1	1.0	1.0	foo2	2009-01-02
4	4.0	0.0	foo5	2009-01-07

What if we needed to identify which rows are missing from the second DataFrame? The easiest way to do this would be to merge the two DataFrames using a left join and set `indicator=True`:

```
In [66]:
```

```
df3 = pd.merge(df1, df2, how='left', indicator=True)  
df3
```

```
Out[66]:
```

	A	B	C	D	_merge
0	0.0	0.0	foo1	2009-01-01	both
1	1.0	1.0	foo2	2009-01-02	both
2	2.0	0.0	foo3	2009-01-05	left_only
3	3.0	1.0	foo4	2009-01-06	left_only
4	4.0	0.0	foo5	2009-01-07	both

This adds a column to the DataFrame which shows the source of each row.

In order to locate the rows that were missing from "df2", we simply filter "df3" to show the rows that were only present in the left DataFrame:

```
In [67]:
```

```
df3[df3._merge == 'left_only']
```

```
Out[67]:
```

	A	B	C	D	_merge
2	2.0	0.0	foo3	2009-01-05	left_only
3	3.0	1.0	foo4	2009-01-06	left_only

Now we can see that rows 2 and 3 were the missing rows.

## 17. Use query to avoid intermediate variables

Let's take another look at the stocks DataFrame:

In [68]:

```
stocks
```

Out[68]:

	Date	Close	Volume	Symbol
0	2016-10-03	31.50	14070500	CSCO
1	2016-10-03	112.52	21701800	AAPL
2	2016-10-03	57.42	19189500	MSFT
3	2016-10-04	113.00	29736800	AAPL
4	2016-10-04	57.24	20085900	MSFT
5	2016-10-04	31.35	18460400	CSCO
6	2016-10-05	57.64	16726400	MSFT
7	2016-10-05	31.59	11808600	CSCO
8	2016-10-05	113.05	21453100	AAPL

If you wanted to filter the DataFrame to only show rows in which the Symbol is "AAPL", this is the usual approach:

In [69]:

```
stocks[stocks.Symbol == 'AAPL']
```

Out[69]:

	Date	Close	Volume	Symbol
1	2016-10-03	112.52	21701800	AAPL
3	2016-10-04	113.00	29736800	AAPL
8	2016-10-05	113.05	21453100	AAPL

However, this can also be done using the `query()` method:

In [70]:

```
stocks.query("Symbol == 'AAPL'")
```

Out[70]:

	Date	Close	Volume	Symbol
--	------	-------	--------	--------

```
1 2016-10-03 112.52 21701800 AAPL
3 2016-10-04 113.00 29736800 AAPL
8 2016-10-05 113.05 21453100 AAPL
```

There are three things worth noting about the `query()` method:

1. You don't have to repeat the name of the DataFrame within the query string.
2. The entire condition is expressed as a string, thus you lose any syntax highlighting.
3. Since there is a string within the condition, you have to use single quotes with the inner string and double quotes with the outer string.

Let's look at another example that shows the real usefulness of `query()`. First let's group by "Symbol" and then take the mean of all numeric columns:

```
In [71]: stocks.groupby('Symbol').mean()
```

```
Out[71]:      Close      Volume
```

Symbol	Close	Volume
<b>AAPL</b>	112.856667	2.429723e+07
<b>CSCO</b>	31.480000	1.477983e+07
<b>MSFT</b>	57.433333	1.866727e+07

What if I wanted to filter this DataFrame to only show rows in which "Close" is less than 100? The usual approach would be to create a temporary DataFrame and then filter that:

```
In [72]: temp = stocks.groupby('Symbol').mean()
temp[temp.Close < 100]
```

```
Out[72]:      Close      Volume
```

Symbol	Close	Volume
<b>CSCO</b>	31.480000	1.477983e+07
<b>MSFT</b>	57.433333	1.866727e+07

But `query()` works even better in this situation, since you can avoid creating an intermediate object:

```
In [73]: stocks.groupby('Symbol').mean().query('Close < 100')
```

```
Out[73]:      Close      Volume
```

Symbol	Close	Volume
CSCO	31.480000	1.477983e+07
MSFT	57.433333	1.866727e+07

In fact, `query()` is a great solution to our previous trick, because it would have allowed us to filter the merged DataFrame without creating the "df3" object:

```
In [74]: pd.merge(df1, df2, how='left', indicator=True).query("_merge == 'left_only'")
```

```
Out[74]:      A      B      C      D      _merge
```

	A	B	C	D	_merge
2	2.0	0.0	foo3	2009-01-05	left_only
3	3.0	1.0	foo4	2009-01-06	left_only

## 18. Reshape a DataFrame from wide format to long format

Let's create another example DataFrame:

```
In [75]: distances = pd.DataFrame([[12345, 100, 200, 300], [34567, 400, 500, 600], [67890, 700, 800, 900]], columns=['zip', 'factory', 'warehouse', 'retail'])  
distances
```

```
Out[75]:      zip  factory  warehouse  retail
```

	zip	factory	warehouse	retail
0	12345	100	200	300
1	34567	400	500	600
2	67890	700	800	900

Let's pretend that a manufacturing company has three locations: a factory, a warehouse, and a retail

store. They've created the DataFrame above, which shows the distance between every US zip code and that particular location.

Let's create one more DataFrame:

```
In [76]: users = pd.DataFrame([[1, '12345', 'factory'], [2, '34567', 'warehouse']],  
                           columns=['user_id', 'zip', 'location_type'])  
users
```

```
Out[76]:   user_id    zip  location_type  
0          1  12345        factory  
1          2  34567      warehouse
```

This is a DataFrame of users. It shows the user's zip code and the location they would like to visit. We want to add a fourth column to "users", which shows the distance between that user and the location they want to visit. This information is available in the "distances" DataFrame, but how do we get it into the "users" DataFrame?

We actually need to merge the DataFrames, but the problem is that the "distances" DataFrame doesn't have the right columns for merging. The solution is to reshape it using the `melt()` method:

```
In [77]: distances_long = distances.melt(id_vars='zip', var_name='location_type', value_name='distance')  
distances_long
```

```
Out[77]:    zip  location_type  distance  
0  12345        factory     100  
1  34567        factory     400  
2  67890        factory     700  
3  12345      warehouse     200  
4  34567      warehouse     500  
5  67890      warehouse     800  
6  12345       retail      300  
7  34567       retail      600  
8  67890       retail      900
```

We've reshaped the "distances" DataFrame from "wide format", meaning lots of columns, to "long format", meaning lots of rows. It contains the same data as before, but it's now structured such that it can easily be merged with the "users" DataFrame:

```
In [78]: pd.merge(users, distances_long)
```

```
Out[78]:
```

	user_id	zip	location_type	distance
0	1	12345	factory	100
1	2	34567	warehouse	500

If you're ever confused about "wide" versus "long" data, the easiest way to recognize a "wide format" DataFrame is that it doesn't tell you what you're looking at. For example, it doesn't tell me what these numbers represent, and it doesn't tell me what these column names represent. In contrast, the "long format" DataFrame tells you that the numbers represent distance and these names represent location types.

## 19. Reverse row order (alternative)

You might remember the drinks DataFrame from the main video:

```
In [79]: drinks.head()
```

```
Out[79]:
```

	country	beer_servings	spirit_servings	wine_servings	total_litres_of_pure_alcohol	continent
0	Afghanistan	0	0	0	0.0	Asia
1	Albania	89	132	54	4.9	Europe
2	Algeria	25	0	14	0.7	Africa
3	Andorra	245	138	312	12.4	Europe
4	Angola	217	57	45	5.9	Africa

This is the method that I taught in the main video for reversing row order, because it will always work:

```
In [80]:
```

```
In [80]: drinks.loc[::-1].head()
```

Out[80]:

	country	beer_servings	spirit_servings	wine_servings	total_litres_of_pure_alcohol	continent
192	Zimbabwe	64	18	4	4.7	Africa
191	Zambia	32	19	4	2.5	Africa
190	Yemen	6	0	0	0.1	Asia
189	Vietnam	111	2	1	2.0	Asia
188	Venezuela	333	100	3	7.7	South America

Alternatively, you can use Python's built-in `reversed()` function to reverse the index, and then use that to reindex the DataFrame:

```
In [81]: drinks.reindex(reversed(drinks.index)).head()
```

Out[81]:

	country	beer_servings	spirit_servings	wine_servings	total_litres_of_pure_alcohol	continent
192	Zimbabwe	64	18	4	4.7	Africa
191	Zambia	32	19	4	2.5	Africa
190	Yemen	6	0	0	0.1	Asia
189	Vietnam	111	2	1	2.0	Asia
188	Venezuela	333	100	3	7.7	South America

If you decide to use this alternative method, be aware that it will fail if the DataFrame has duplicate values in the index. To demonstrate this, let's give the stocks DataFrame a non-unique index:

```
In [82]: stocks = stocks.set_index('Date')
stocks
```

Out[82]:

	Close	Volume	Symbol
	Date		
2016-10-03	31.50	14070500	CSCO
2016-10-03	112.52	21701800	AAPL
2016-10-03	57.42	19189500	MSFT

```
2016-10-04 113.00 29736800 AAPL
2016-10-04 57.24 20085900 MSFT
2016-10-04 31.35 18460400 CSCO
2016-10-05 57.64 16726400 MSFT
2016-10-05 31.59 11808600 CSCO
2016-10-05 113.05 21453100 AAPL
```

Since the index above is not unique, this will result in an error:

```
In [83]: # stocks.reindex(reversed(stocks.index))
```

## 20. Reverse column order (alternative)

This is the method that I taught in the main video for reversing column order, because it will always work:

```
In [84]: drinks.loc[:, ::-1].head()
```

```
Out[84]: continent  total_litres_of_pure_alcohol  wine_servings  spirit_servings  beer_servings  country
0 Asia 0.0 0 0 0 Afghanistan
1 Europe 4.9 54 132 89 Albania
2 Africa 0.7 14 0 25 Algeria
3 Europe 12.4 312 138 245 Andorra
4 Africa 5.9 45 57 217 Angola
```

Alternatively, you can use Python's built-in `reversed()` function to reverse the columns attribute, and then pass that as a filter to the DataFrame:

```
In [85]: drinks[reversed(drinks.columns)].head()
```

```
Out[85]: continent  total_litres_of_pure_alcohol  wine_servings  spirit_servings  beer_servings  country
```

0	Asia	0.0	0	0	0	Afghanistan
1	Europe	4.9	54	132	89	Albania
2	Africa	0.7	14	0	25	Algeria
3	Europe	12.4	312	138	245	Andorra
4	Africa	5.9	45	57	217	Angola

If you decide to use this alternative method, be aware that it will fail if the DataFrame has duplicate column names. To demonstrate this, let's rename two of the columns in the stocks DataFrame:

```
In [86]: stocks = stocks.rename({'Symbol':'XYZ', 'Volume':'XYZ'}, axis='columns')
stocks
```

Out[86]:

	Close	XYZ	XYZ
	Date		
2016-10-03	31.50	14070500	CSCO
2016-10-03	112.52	21701800	AAPL
2016-10-03	57.42	19189500	MSFT
2016-10-04	113.00	29736800	AAPL
2016-10-04	57.24	20085900	MSFT
2016-10-04	31.35	18460400	CSCO
2016-10-05	57.64	16726400	MSFT
2016-10-05	31.59	11808600	CSCO
2016-10-05	113.05	21453100	AAPL

Since the column names are not unique, you will get multiple copies of those columns:

```
In [87]: stocks[reversed(stocks.columns)]
```

Out[87]:

	XYZ	XYZ	XYZ	XYZ	Close
	Date				
2016-10-03	14070500	CSCO	14070500	CSCO	31.50

<b>2016-10-03</b>	21701800	AAPL	21701800	AAPL	112.52
<b>2016-10-03</b>	19189500	MSFT	19189500	MSFT	57.42
<b>2016-10-04</b>	29736800	AAPL	29736800	AAPL	113.00
<b>2016-10-04</b>	20085900	MSFT	20085900	MSFT	57.24
<b>2016-10-04</b>	18460400	CSCO	18460400	CSCO	31.35
<b>2016-10-05</b>	16726400	MSFT	16726400	MSFT	57.64
<b>2016-10-05</b>	11808600	CSCO	11808600	CSCO	31.59
<b>2016-10-05</b>	21453100	AAPL	21453100	AAPL	113.05

## 21. Split a string into multiple columns (alternative)