20 Pandas Functions That Will Boost Your Data Analysis Process

Explained with examples.

Soner Yıldırım

Jul 15, 2020 · 10 min read

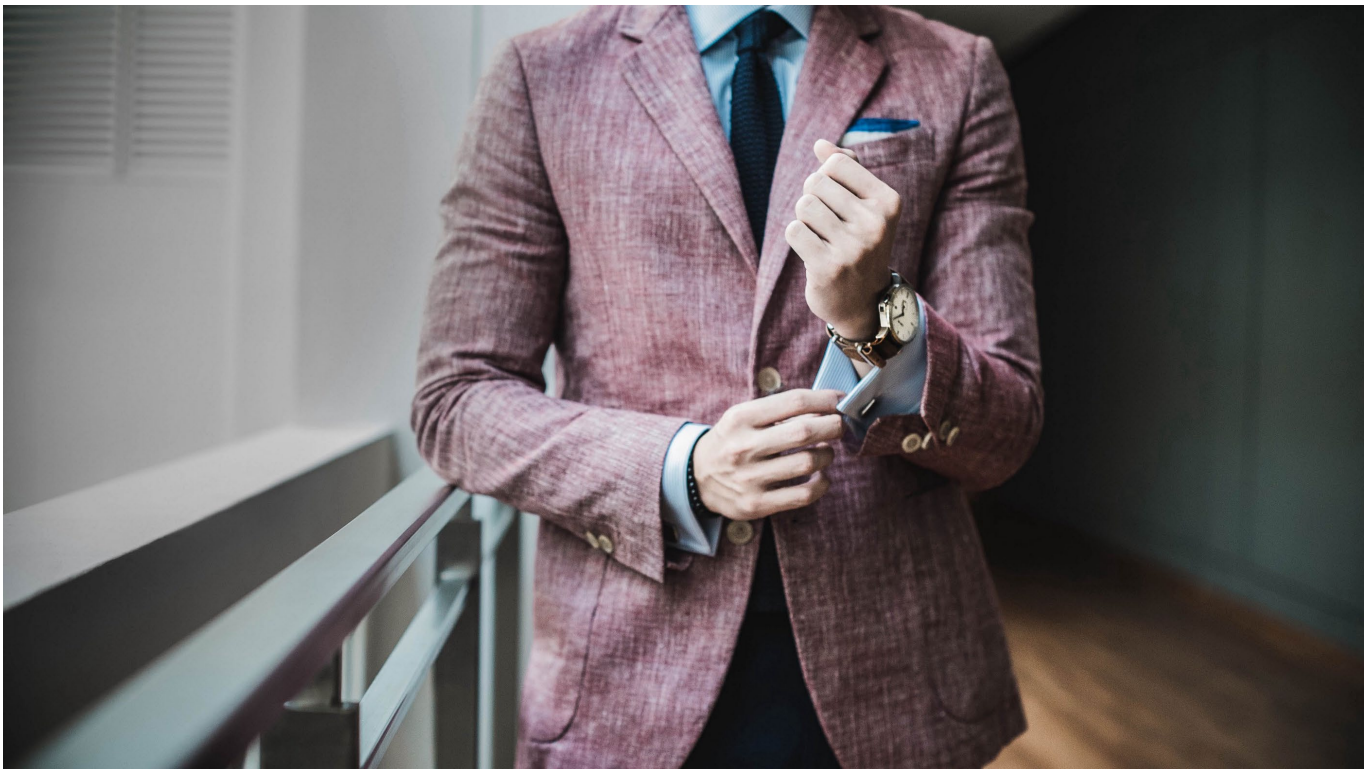

Photo by Heng Films on Unsplash

Pandas is a predominantly used python data analysis library. It provides many functions and methods to expedite the data analysis process. What makes pandas so common is its functionality, flexibility, and simple syntax.

In this post, I will explain 20 pandas functions with examples. Some of them are so common that I'm sure you have used before. Some of them might be new to you.

But, all of them will add value to your data analysis process.

Let's start.

```
import numpy as np
import pandas as pd
```

1. Query

We sometimes need to filter a dataframe based on a condition or apply a mask to get certain values. One easy way to filter a dataframe is **query** function. Let's first create a sample dataframe.

```
values_1 = np.random.randint(10, size=10)
values_2 = np.random.randint(10, size=10)
years = np.arange(2010,2020)
groups = ['A','A','B','A','B','B','C','A','C','C']

df = pd.DataFrame({'group':groups, 'year':years, 'value_1':values_1,
'value_2':values_2})

df
```

	group	year	value_1	value_2
0	A	2010	2	3
1	A	2011	3	3
2	B	2012	5	1
3	A	2013	9	4
4	B	2014	6	1
5	B	2015	7	9
6	C	2016	4	2
7	A	2017	8	5
8	C	2018	4	6
9	C	2019	0	9

It is very simple to use query function which only requires to write the condition for filtering.

```
df.query('value_1 < value_2')
```

	group	year	value_1	value_2
0	A	2010	2	3
5	B	2015	7	9
8	C	2018	4	6
9	C	2019	0	9

2. Insert

When we want to add a new column to a dataframe, it is added at the end by default. However, pandas offers the option to add the new column in any position

using **insert** function.

We need to specify the position by passing an index as first argument. This value must be an integer. Column indices start from zero just like row indices. The second argument is column name and the third argument is the object that includes values which can be **Series** or an **array-like** object.

```
#new column
new_col = np.random.randn(10)

#insert the new column at position 2
df.insert(2, 'new_col', new_col)

df
```

	group	year	new_col	value_1	value_2
0	A	2010	0.332581	2	3
1	A	2011	0.122312	3	3
2	B	2012	0.551692	5	1
3	A	2013	-1.515015	9	4
4	B	2014	0.078729	6	1
5	B	2015	-0.375737	7	9
6	C	2016	-1.159010	4	2
7	A	2017	-0.161005	8	5
8	C	2018	0.275472	4	6
9	C	2019	0.113718	0	9

3. Cumsum

The dataframe contains some yearly values of 3 different groups. We may only be interested in yearly values but there are some cases in which we also need a cumulative sum. Pandas provides an easy-to-use function to calculate cumulative sum which is **cumsum**.

If we only apply cumsum, groups (A, B, C) will be ignored. This kind of cumulative values may be useless in some cases because we are not able to distinguish between groups. Don't worry! There is a very simple and convenient solution for this issue. We can apply **groupby** and then **cumsum** function.

```
df['cumsum_2'] = df[['value_2','group']].groupby('group').cumsum()

df
```

	group	year	new_col	value_1	value_2	cumsum_2
0	A	2010	0.332581	2	3	3
1	A	2011	0.122312	3	3	6
2	B	2012	0.551692	5	1	1
3	A	2013	-1.515015	9	4	10
4	B	2014	0.078729	6	1	2
5	B	2015	-0.375737	7	9	11
6	C	2016	-1.159010	4	2	2
7	A	2017	-0.161005	8	5	15
8	C	2018	0.275472	4	6	8
9	C	2019	0.113718	0	9	17

4. Sample

Sample method allows you to select values randomly from a **Series** or **DataFrame**. It is useful when we want to select a random sample from a distribution.

```
sample1 = df.sample(n=3)
sample1
```

	group	year	new_col	value_1	value_2	cumsum_2
9	C	2019	0.113718	0	9	17
0	A	2010	0.332581	2	3	3
2	B	2012	0.551692	5	1	1

We specify the number of values with n parameter but we can also pass a ratio to **frac** parameter. For instance, 0.5 will return half of the rows.

```
sample2 = df.sample(frac=0.5)
sample2
```

	group	year	new_col	value_1	value_2	cumsum_2
0	A	2010	0.332581	2	3	3
2	B	2012	0.551692	5	1	1
5	B	2015	-0.375737	7	9	11
9	C	2019	0.113718	0	9	17
3	A	2013	-1.515015	9	4	10

To obtain reproducible samples, we can use **random_state** parameter. If an

integer value is passed to random_state, the same sample will be produced every time the code is run.

5. Where

"Where" is used to replace values in rows or columns based on a condition. The default replacement values is NaN but we can also specify the value to be put as a replacement.

```
df['new_col'].where(df['new_col'] > 0 , 0)
```

```
0    0.332581
1    0.122312
2    0.551692
3    0.000000
4    0.078729
5    0.000000
6    0.000000
7    0.000000
8    0.275472
9    0.113718
Name: new_col, dtype: float64
```

The way "where" works is that values that fit the condition are selected and the remaining values are replaced with the specified value. **where(df['new_col']>0, 0)** selects all the values in "new_col" that are greater than 0 and the remaining values are replaced with 0. Thus, where can also be considered as a mask operation.

One important point is that "**where**" for Pandas and NumPy are not exactly the same. We can achieve the same result but with slightly different syntax.
With **DataFrame.where**, the values that fit the condition are selected **as is** and

the other values are replaced with the specified value. **Np.where** requires to also specify the value for the ones that fit the condition. The following two lines return the same result:

```
df['new_col'].where(df['new_col'] > 0 , 0)

np.where(df['new_col'] > 0, df['new_col'], 0)
```

6. Isin

We use filtering or selecting methods a lot when working with dataframes. **Isin** method is kind of an advanced filtering. For example, we can filter values based on a list of selections.

```
years = ['2010','2014','2017']
df[df.year.isin(years)]
```

	group	year	new_col	value_1	value_2	cumsum_2
0	A	2010	0.332581	2	3	3
4	B	2014	0.078729	6	1	2
7	A	2017	-0.161005	8	5	15

7. Loc and iloc

Loc and iloc are used to select rows and columns.

- loc: select by labels

- iloc: select by positions

loc is used to select data by label. The labels of columns are the column names. We need to be careful about row labels. If we do not assign any specific indices, pandas created integer index by default. Thus, the row labels are integers starting from 0 and going up. The row positions that are used with **iloc** are also integers starting from 0.

Selecting first 3 rows and first 2 columns with iloc:

```
df.iloc[:3, :2]
```

	group	year
0	A	2010
1	A	2011
2	B	2012

Selecting first 3 rows and first 2 columns with loc:

```
df.loc[:2, ['group','year']]
```

	group	year
0	A	2010
1	A	2011
2	B	2012

Note: Upper boundaries of indices are included when loc is used whereas they are excluded with iloc.

Selecting rows “1”, “3”, “5” and columns “year” and “value_1”:

```
df.loc[[1,3,5], ['year','value_1']]
```

	year	value_1
1	2011	3
3	2013	9
5	2015	7

8. Pct_change

This function is used to calculate the percent change through the values in a series. Consider we have a series that contains [2,3,6]. If we apply pct_change to this series, the returned series will be [NaN, 0.5, 1.0]. There is 50% increase from the first element to the second and 100% from the second to the third one. Pct_change function is useful in comparing the percentage of change in a time series of elements.

```
df.value_1.pct_change()
```

```
0         NaN
1    0.500000
2    0.666667
3    0.800000
4   -0.333333
5    0.166667
6   -0.428571
7    1.000000
8   -0.500000
9   -1.000000
Name: value_1, dtype: float64
```

9. Rank

Rank function assigns rank to the values. Assume we have a series s that contains [1,7,5,3]. The ranks assigned to these values will be [1,4,3,2].

```
df['rank_1'] = df['value_1'].rank()
df
```

	group	year	new_col	value_1	value_2	cumsum_2	rank_1
0	A	2010	0.332581	2	3	3	2.0
1	A	2011	0.122312	3	3	6	3.0
2	B	2012	0.551692	5	1	1	6.0
3	A	2013	-1.515015	9	4	10	10.0
4	B	2014	0.078729	6	1	2	7.0
5	B	2015	-0.375737	7	9	11	8.0
6	C	2016	-1.159010	4	2	2	4.5
7	A	2017	-0.161005	8	5	15	9.0
8	C	2018	0.275472	4	6	8	4.5
9	C	2019	0.113718	0	9	17	1.0

10. Melt

Melt is used to convert wide dataframes to narrow ones. What I mean by wide is a dataframe with a high number of columns. Some dataframes are structured in a way that consecutive measurements or variables are represented as columns. In some cases, representing these columns as rows may fit better to our task. Consider the following dataframe:

```
df_wide
```

	city	day1	day2	day3	day4	day5
0	A	22	10	25	18	12
1	B	25	14	22	15	14
2	C	28	13	26	17	18

We have three different cities and measurements done on different days. We decide to represent these days as rows in a column. There will also be a column to show the measurements. We can easily accomplish this by using **melt** function:

```
df_wide.melt(id_vars=['city'])
```

	city	variable	value
0	A	day1	22
1	B	day1	25
2	C	day1	28
3	A	day2	10
4	B	day2	14
5	C	day2	13
6	A	day3	25
7	B	day3	22
8	C	day3	26
9	A	day4	18
10	B	day4	15
11	C	day4	17
12	A	day5	12
13	B	day5	14
14	C	day5	18

Variable and value column names are given by default. We can use **var_name** and **value_name** parameters of melt function to assign new column names.

11. Explode

Assume your data set includes multiple entries of a feature on a single observation (row) but you want to analyze them on separate rows.

	ID	measurement	day
0	a	4	1
1	b	6	1
2	c	[2, 3, 8]	1

We want to see the measurements of ‘c’ on day ‘1’ on separate rows which easily be done using **explode.**

```
df1.explode('measurement').reset_index(drop=True)
```

	ID	measurement	day
0	a	4	1
1	b	6	1
2	c	2	1
3	c	3	1
4	c	8	1

12. Nunique

Nunique counts the number of unique entries over columns or rows. It is very useful in categorical features especially in cases where we do not know the number of categories beforehand. Let’s take a look at our initial dataframe:

	group	year	new_col	value_1	value_2	cumsum_2	rank_1
0	A	2010	0.332581	2	3	3	2.0
1	A	2011	0.122312	3	3	6	3.0
2	B	2012	0.551692	5	1	1	6.0
3	A	2013	-1.515015	9	4	10	10.0
4	B	2014	0.078729	6	1	2	7.0
5	B	2015	-0.375737	7	9	11	8.0
6	C	2016	-1.159010	4	2	2	4.5
7	A	2017	-0.161005	8	5	15	9.0
8	C	2018	0.275472	4	6	8	4.5
9	C	2019	0.113718	0	9	17	1.0

```
df.year.nunique()
10


df.group.nunique()
3
```

We can directly apply nunique function to the dataframe and see the number of unique values in each column:

```
df.nunique()
```

```
group        3
year        10
new_col     10
value_1      9
value_2      7
cumsum_2     9
rank_1       9
dtype: int64
```

If **axis** parameter is set to 1, nunique returns the number of unique values in each row.

13. Lookup

It can be used to look up values in the DataFrame based on the values on other row, column pairs. This function is best explained via an example. Assume we have the following DataFrame:

	Day	Person	John	Alex	Oscar	Derek
0	1	Alex	4	4	4	6
1	2	John	7	6	2	2
2	3	Alex	8	9	3	1
3	4	Derek	9	2	5	8
4	5	Oscar	2	6	1	7
5	6	John	6	6	4	8
6	7	Derek	6	5	5	4
7	8	Oscar	5	5	9	5

For each day, we have measurements of 4 people and a column that includes the names of these 4 people. We want to create a new column that shows the measurement of the person in “Person” column. Thus, for the first row, the value in the new column will be 4 (the value in column “Alex”).

```
df['Person_point'] = df.lookup(df.index, df['Person'])
df
```

	Day	Person	John	Alex	Oscar	Derek	Person_point
0	1	Alex	4	4	4	6	4
1	2	John	7	6	2	2	7
2	3	Alex	8	9	3	1	9
3	4	Derek	9	2	5	8	8
4	5	Oscar	2	6	1	7	1
5	6	John	6	6	4	8	6
6	7	Derek	6	5	5	4	4
7	8	Oscar	5	5	9	5	9

14. Infer_objects

Pandas supports a wide range of data types, one of which is **object**. Object covers text or mixed (numeric and non-numeric) values. However, it is not preferred to use object data type if a different option is available. Certain operations is executed faster with more specific data types. For example, we prefer to have integer or float data type for numerical values.

infer_objects attempts to infer better data types for object columns. Consider the following dataframe:

	A	B	C	D
0	1	2.1	True	b
1	2	1.5	False	c
2	3	2	False	d
3	4	2.1	True	f

```
df2.dtypes
A    object
```

```
B    object
C    object
D    object
dtype: object
```

All of the data types are object. Let's see what the inferred data types are:

```
df2.infer_objects().dtypes


A      int64
B    float64
C       bool
D     object
dtype: object
```

It may seem trivial but will definitely be useful when there are lots of columns.

15. Memory_usage

Memory_usage() returns how much memory each column uses in bytes. It is useful especially when we work with large dataframes. Consider the following dataframe with 1 million rows.

```
df_large = pd.DataFrame({'A': np.random.randn(1000000),
                         'B': np.random.randint(100, size=1000000)})


df_large.shape
(1000000, 2)
```

And the memory usage for each column in bytes:

```
df_large.memory_usage()
Index         128
A         8000000
B         8000000
dtype: int64
```

Memory usage of entire dataframe in megabytes:

```
df_large.memory_usage().sum() / (1024**2) #converting to megabytes

15.2589111328125
```

16. Describe

Describe function calculates basic statistics for numeric columns which are count, mean, standard deviation, min and max values, median, first and third quartile. Thus, it provides a statistical summary of the dataframe.

```
df.describe()
```

	year	new_col	value_1	value_2	cumsum_2	rank_1
count	10.00000	10.000000	10.000000	10.000000	10.000000	10.000000
mean	2014.50000	-0.173626	4.800000	4.300000	7.500000	5.500000
std	3.02765	0.669093	2.780887	2.945807	5.681354	3.018462
min	2010.00000	-1.515015	0.000000	1.000000	1.000000	1.000000
25%	2012.25000	-0.322054	3.250000	2.250000	2.250000	3.375000
50%	2014.50000	0.096223	4.500000	3.500000	7.000000	5.250000
75%	2016.75000	0.237182	6.750000	5.750000	10.750000	7.750000
max	2019.00000	0.551692	9.000000	9.000000	17.000000	10.000000

17. Merge

Merge() combines DataFrames based on values in shared columns. Consider the following two dataframes.

		column_a	column_b	column_c
df1	0	1	a	True
	1	2	b	True
	2	3	c	False
	3	4	d	True
df2	0	1	a	False
	1	2	k	False
	2	9	l	False
	3	10	m	True

We can merge them based on shared values in a column. The parameter that sets the condition for merging is the "**on**" parameter.

```
df_merge = pd.merge(df1, df2, on='column_a')
df_merge
```

	column_a	column_b_x	column_c_x	column_b_y	column_c_y
0	1	a	True	a	False
1	2	b	True	k	False

df1 and df2 are merged based on the common values in column_a.
The **how** parameter of merge function allows to combine dataframes in different ways. The possible values for how are 'inner', 'outer', 'left', 'right'.

- inner: only rows with same values in the column specified by **on** parameter (default value of **how** parameter)
- outer: all the rows
- left: all rows from left DataFrame
- right: all rows from right DataFrame

18. Select_dtypes

Select_dtypes function returns a subset of the DataFrame's columns based on the condition set on data types. It allows to include or exclude certain data types using **include** and **exlude** parameters.

```
df.select_dtypes(include='int64')
```

	year	value_1	value_2	cumsum_2
0	2010	2	3	3
1	2011	3	3	6
2	2012	5	1	1
3	2013	9	4	10
4	2014	6	1	2
5	2015	7	9	11
6	2016	4	2	2
7	2017	8	5	15
8	2018	4	6	8
9	2019	0	9	17

```
df.select_dtypes(exclude='int64')
```

	group	new_col	rank_1
0	A	0.332581	2.0
1	A	0.122312	3.0
2	B	0.551692	6.0
3	A	-1.515015	10.0
4	B	0.078729	7.0
5	B	-0.375737	8.0
6	C	-1.159010	4.5
7	A	-0.161005	9.0
8	C	0.275472	4.5
9	C	0.113718	1.0

19. Replace

As the name suggests, it allows to replace values in a dataframe.

```
df.replace('A', 'A_1')
```

The first parameter is the value to replaced and the second one is the new value.

	group	year	new_col	value_1	value_2	cumsum_2	rank_1
0	A_1	2010	0.332581	2	3	3	2.0
1	A_1	2011	0.122312	3	3	6	3.0
2	B	2012	0.551692	5	1	1	6.0
3	A_1	2013	-1.515015	9	4	10	10.0
4	B	2014	0.078729	6	1	2	7.0
5	B	2015	-0.375737	7	9	11	8.0
6	C	2016	-1.159010	4	2	2	4.5
7	A_1	2017	-0.161005	8	5	15	9.0
8	C	2018	0.275472	4	6	8	4.5
9	C	2019	0.113718	0	9	17	1.0

We can also pass in a dictionary for multiple replacements at the same time.

```
df.replace({'A':'A_1', 'B':'B_1'})
```

	group	year	new_col	value_1	value_2	cumsum_2	rank_1
0	A_1	2010	0.332581	2	3	3	2.0
1	A_1	2011	0.122312	3	3	6	3.0
2	B_1	2012	0.551692	5	1	1	6.0
3	A_1	2013	-1.515015	9	4	10	10.0
4	B_1	2014	0.078729	6	1	2	7.0
5	B_1	2015	-0.375737	7	9	11	8.0
6	C	2016	-1.159010	4	2	2	4.5
7	A_1	2017	-0.161005	8	5	15	9.0
8	C	2018	0.275472	4	6	8	4.5
9	C	2019	0.113718	0	9	17	1.0

20. Applymap

Applymap function is used to apply a function to a dataframe elementwise. Please note that if a vectorized version of an operation is available, it should be preferred over applymap. For instance, if we want to multiple each element by a number, we don't need and should not use applymap function. A simple vectorized operation (e.g. df * 4) is much faster in that case.

However, there might be some cases where we do not have the option of vectorized operation. For instance, we can change the style of a dataframe using **Style** property of pandas dataframes. The following function changes the color of negative values as red.

```
def color_negative_values(val):
   color = 'red' if val < 0 else 'black'
   return 'color: %s' % color
```

We need to use applymap function to apply this function to a dataframe.

```
df3.style.applymap(color_negative_values)
```

	A	B
0	-1.200000	8.100000
1	-4.400000	3.100000
2	3.000000	2.200000
3	5.200000	-2.500000
4	-2.300000	5.200000