Introduction

I am not ashamed to admit. melt() and pivot() were the hardest of pandas to learn for me. It took me more than 3-4 months to wrap my head around. When I first saw them, I tried and tried but could not understand how or when they are used. So, I gave up, moved on, and met them again. Tried, failed, moved on, and met again. Repeated many times.

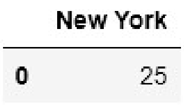
Setup

```
# Load necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Plotting pretty figures and avoid blurry images
%config InlineBackend.figure_format = 'retina'
# Larger scale for plots in notebooks
sns.set_context('talk')

# Ignore warnings
import warnings
warnings.filterwarnings('ignore')

# Enable multiple cell outputs
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = 'all'
```

Pandas melt()

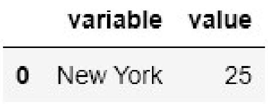
Let’s start with a very stupid example. I will create a 1x1 dataframe that holds a city name and a temperature for a single day. Then, I will call melt() on it to see what effect it has:

```
df = pd.DataFrame({'New York': [25]})
df
```



	New York
0	25

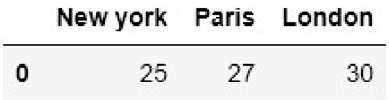
>>> df.melt()



	variable	value
0	New York	25

So, without any parameters melt() takes a column and turns it into a row with two new columns (excluding the index). Let's add two more cities as columns:

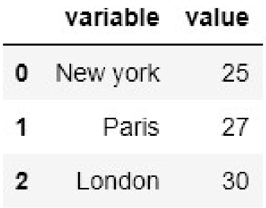
```
df = pd.DataFrame({'New york': [25], 'Paris': [27], 'London': [30]})
df
```



	New york	Paris	London
0	25	27	30

If you notice, this type of format for dataframes are not easy to work with and it is not clean. What would be ideal is to take the columns and turn them into rows with their temperature values on the right side:

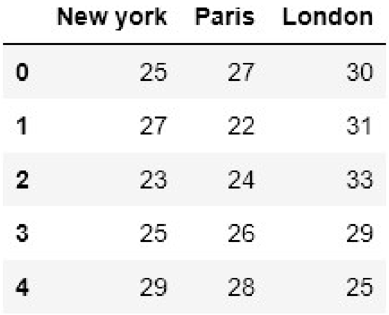
df.melt()



	variable	value
0	New york	25
1	Paris	27
2	London	30

Let's add more temperatures for the cities:

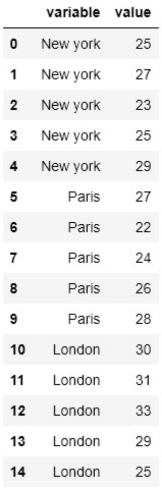
```
df_larger = pd.DataFrame({
    'New york': [25, 27, 23, 25, 29],
    'Paris': [27, 22, 24, 26, 28],
    'London': [30, 31, 33, 29, 25]
})
>>> df_larger
```



	New york	Paris	London
0	25	27	30
1	27	22	31
2	23	24	33
3	25	26	29
4	29	28	25

What do you think will happen if we call melt() on this version of the dataframe? Watch:

df_larger.melt()

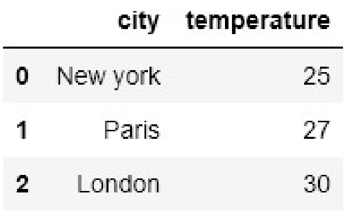


	variable	value
0	New york	25
1	New york	27
2	New york	23
3	New york	25
4	New york	29
5	Paris	27
6	Paris	22
7	Paris	24
8	Paris	26
9	Paris	28
10	London	30
11	London	31
12	London	33
13	London	29
14	London	25

Just like expected, it converts each column value into a row. For example, let’s take a key-value pair. New York’s temperatures are [25, 27, 23, 25, 29]. This means there are 5 key-value pairs and when we use melt(), pandas takes each of those pairs and displays them as a single row with two columns. After pandas is done with New York, it moves on to other columns.

When melt() displays each key-value pair in two columns, it gives the columns default names which are variable and value. It is possible to change them to something that makes more sense:

```
df.melt(var_name='city', value_name='temperature')
```



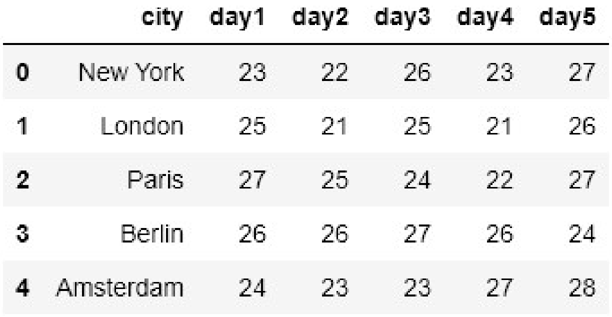
	city	temperature
0	New york	25
1	Paris	27
2	London	30

var_name and value_name can be used to change the labels of the melted dataframe's columns.

If we keep adding columns, melt() will always convert each value into a row with two columns that contain the previous column's name and its value.

Now, let’s get a little serious. Say we have this dataframe:

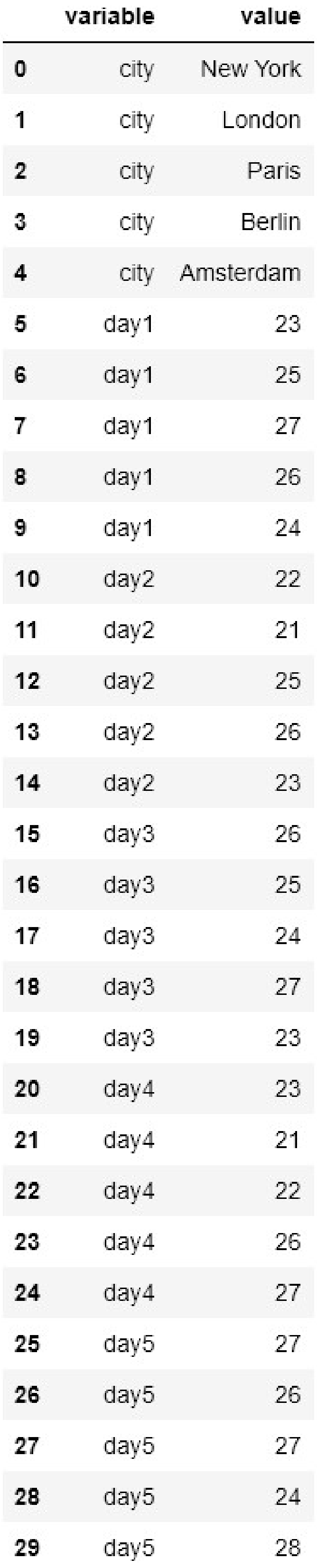
```
temperatures = pd.DataFrame({
    'city': ['New York', 'London', 'Paris', 'Berlin', 'Amsterdam'],
    'day1': [23, 25, 27, 26, 24],
    'day2': [22, 21, 25, 26, 23],
    'day3': [26, 25, 24, 27, 23],
    'day4': [23, 21, 22, 26, 27],
    'day5': [27, 26, 27, 24, 28]
})
temperatures
```



	city	day1	day2	day3	day4	day5
0	New York	23	22	26	23	27
1	London	25	21	25	21	26
2	Paris	27	25	24	22	27
3	Berlin	26	26	27	26	24
4	Amsterdam	24	23	23	27	28

This time, we already have the cities as a column. But still, this type of format for tables are not useful to work with. This dataset holds temperature information for 5 cities for 5 days. We can’t even perform simple computations like mean on this type of data. Let’s try melting the dataframe:

```
>>> temperatures.melt()
```

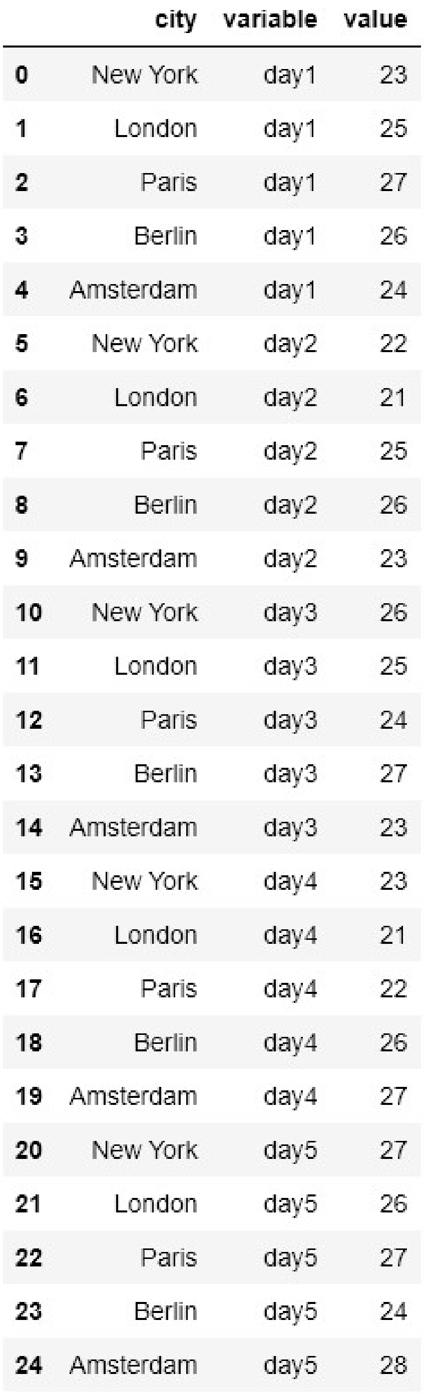


	variable	value
0	city	New York
1	city	London
2	city	Paris
3	city	Berlin
4	city	Amsterdam
5	day1	23
6	day1	25
7	day1	27
8	day1	26
9	day1	24
10	day2	22
11	day2	21
12	day2	25
13	day2	26
14	day2	23
15	day3	26
16	day3	25
17	day3	24
18	day3	27
19	day3	23
20	day4	23
21	day4	21
22	day4	22
23	day4	26
24	day4	27
25	day5	27
26	day5	26
27	day5	27
28	day5	24
29	day5	28

This is not what we want, melt() turned the city names into rows too. What would be ideal is if we kept the cities as columns and append the remaining columns as rows. melt() has a parameter called id_vars to do just that.

If we want to turn only some of the columns into rows, pass the columns to keep as a list (even if it is a single value) to id_vars . id_vars stands for identity variables.

temperatures.melt(id_vars=['city'])

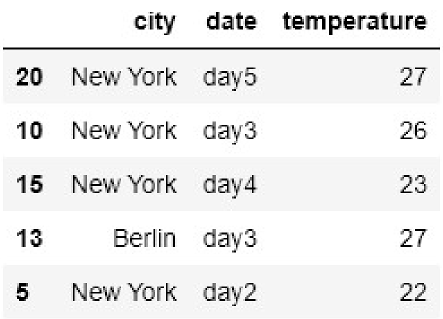


	city	variable	value
0	New York	day1	23
1	London	day1	25
2	Paris	day1	27
3	Berlin	day1	26
4	Amsterdam	day1	24
5	New York	day2	22
6	London	day2	21
7	Paris	day2	25
8	Berlin	day2	26
9	Amsterdam	day2	23
10	New York	day3	26
11	London	day3	25
12	Paris	day3	24
13	Berlin	day3	27
14	Amsterdam	day3	23
15	New York	day4	23
16	London	day4	21
17	Paris	day4	22
18	Berlin	day4	26
19	Amsterdam	day4	27
20	New York	day5	27
21	London	day5	26
22	Paris	day5	27
23	Berlin	day5	24
24	Amsterdam	day5	28

After using id_vars, the city column stayed as a column. But it has become longer. The reason is that for each city there were 5 days of observations. When we take those observations from columns and display them as rows, pandas automatically adds new rows to fit the new values.

Even though we have the table in better shape, the column names are not exactly what we want. Instead of changing them manually after melting the table, we can directly do it with melt():

```
temperatures.melt(id_vars=['city'],
                  var_name='date',
                  value_name='temperature').sample(5)
```



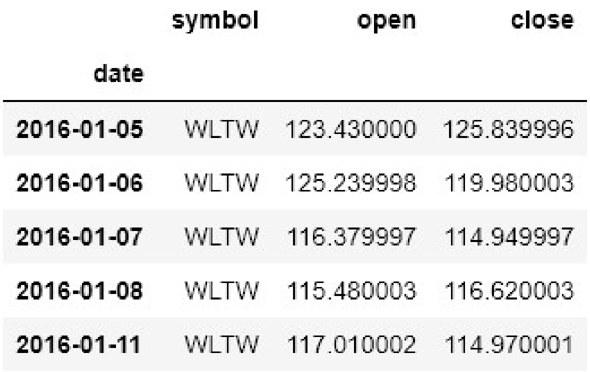
	city	date	temperature
20	New York	day5	27
10	New York	day3	26
15	New York	day4	23
13	Berlin	day3	27
5	New York	day2	22

The same dataframe with different column labels.

Pandas melt() on real-world data

Now, it is time we work on a real-world dataset to bring the point home. It contains stocks information for the year 2016 for more than 501 companies:

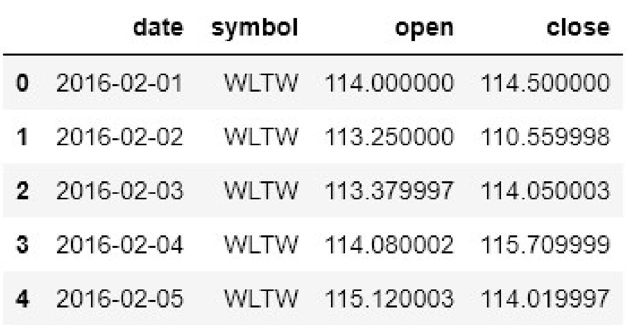
```
stocks = pd.read_csv('data/prices-split-adjusted.csv',
                     usecols=['date', 'symbol', 'open', 'close'],
                     parse_dates=['date'],
                     index_col='date')
stocks.head()
```



	symbol	open	close
date			
2016-01-05	WLTW	123.430000	125.839996
2016-01-06	WLTW	125.239998	119.980003
2016-01-07	WLTW	116.379997	114.949997
2016-01-08	WLTW	115.480003	116.620003
2016-01-11	WLTW	117.010002	114.970001

I will subset it for only one month because there are observations for each day:

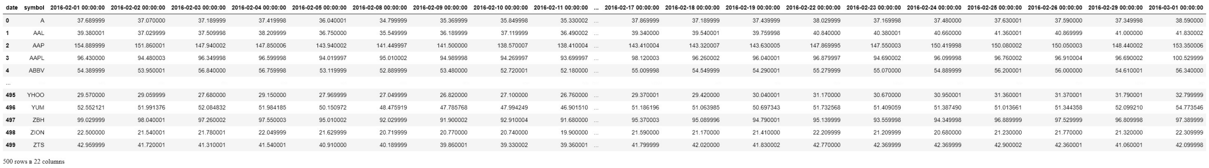
```
stocks_small = stocks.loc['2016-02-01':'2016-03-01'].reset_index()
stocks_small.head()
```



	date	symbol	open	close
0	2016-02-01	WLTW	114.000000	114.500000
1	2016-02-02	WLTW	113.250000	110.559998
2	2016-02-03	WLTW	113.379997	114.050003
3	2016-02-04	WLTW	114.080002	115.709999
4	2016-02-05	WLTW	115.120003	114.019997

Now, so that I can show you how to use melt() on a real world-data, I will perform an operation using pivot().

The data now is in this format:



date	symbol	2016-02-01 00:00:00	2016-02-02 00:00:00	2016-02-03 00:00:00	2016-02-04 00:00:00	2016-02-05 00:00:00	2016-02-08 00:00:00	2016-02-09 00:00:00	2016-02-10 00:00:00	2016-02-11 00:00:00	...
0	A	37.689999	37.070000	37.189999	37.419998	36.040001	34.799999	35.369999	35.849998	35.330002	...
1	AAL	39.380001	37.029999	37.509998	38.209999	36.750000	35.549999	36.189999	37.119999	36.490002	...
2	AAP	154.889999	151.860001	147.940002	147.850006	143.940002	141.449997	141.500000	138.570007	138.410004	...
3	AAPL	96.430000	94.480003	96.349998	96.599998	94.019997	95.010002	94.989998	94.269997	93.699997	...
4	ABBV	54.389999	53.950001	56.840000	56.759998	53.119999	52.889999	53.480000	52.720001	52.180000	...
...	...	...	...	...	...	...	...	...	...	...	...
495	YHOO	29.570000	29.059999	27.680000	29.150000	27.969999	27.049999	26.820000	27.100000	26.760000	...
496	YUM	52.552121	51.991376	52.084832	51.984185	50.150972	48.475919	47.785768	47.994249	46.901510	...
497	ZBH	99.029999	98.040001	97.260002	97.550003	95.010002	92.029999	91.900002	92.910004	91.680000	...
498	ZION	22.500000	21.540001	21.780001	22.049999	21.629999	20.719999	20.770000	20.740000	19.900000	...
499	ZTS	42.959999	41.720001	41.310001	41.540001	40.910000	40.189999	39.860001	39.330002	39.360001	...

date	symbol	2016-02-17 00:00:00	2016-02-18 00:00:00	2016-02-19 00:00:00	2016-02-22 00:00:00	2016-02-23 00:00:00	2016-02-24 00:00:00	2016-02-25 00:00:00	2016-02-26 00:00:00	2016-02-29 00:00:00	2016-03-01 00:00:00
0	A	37.869999	37.189999	37.439999	38.029999	37.169998	37.480000	37.630001	37.590000	37.349998	38.590000
1	AAL	39.340000	39.540001	39.759998	40.840000	40.380001	40.660000	41.360001	40.869999	41.000000	41.830002
2	AAP	143.410004	143.320007	143.630005	147.869995	147.550003	150.419998	150.080002	150.050003	148.440002	153.350006
3	AAPL	98.120003	96.260002	96.040001	96.879997	94.690002	96.099998	96.760002	96.910004	96.690002	100.529999
4	ABBV	55.009998	54.549999	54.290001	55.279999	55.070000	54.889999	56.200001	56.000000	54.610001	56.340000
...	...	...	...	...	...	...	...	...	...	...	...
495	YHOO	29.370001	29.420000	30.040001	31.170000	30.670000	30.950001	31.360001	31.370001	31.790001	32.799999
496	YUM	51.186196	51.063985	50.697343	51.732568	51.409059	51.387490	51.013661	51.344358	52.099210	54.773546
497	ZBH	95.370003	95.089996	94.790001	95.139999	93.559998	94.349998	96.889999	97.529999	96.809998	97.389999
498	ZION	21.590000	21.170000	21.410000	22.209999	21.209999	20.680000	21.230000	21.770000	21.320000	22.309999
499	ZTS	41.799999	42.020000	41.830002	42.770000	42.369999	42.369999	42.900002	42.360001	41.060001	42.099998

500 rows в 22 columns

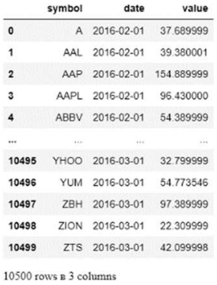
500 rows, 22 columns

It has got 500 rows for all companies and 22 columns for 22 days in February. Each cell contains the close prices of stocks on a given day. Why did I choose this format? Because often real-world data comes in this shape.

For example, when recording this dataset, let's say they wanted to add a new value for a new day. There are 500 companies and if you don't add the new day as a new column, you would have to write out 500 companies once again adding 500 more rows to the dataset. Adding new observations as new columns is very handy for people who are recording this data.

So, now we want to turn back this stocks_small dataset into its original format using melt() so that it is easier to work with. If you look at the data once again, we want to turn all the date columns into two columns which contain (date, close price) key-value pairs preserving the symbol column. Just like we did in the previous examples, if we pass symbol to id_vars, it stays a column and the other dates will become rows:

```
melted = reshaped.melt(id_vars=['symbol'])
melted
```



	symbol	date	value
0	A	2016-02-01	37.689999
1	AAL	2016-02-01	39.380001
2	AAP	2016-02-01	154.889999
3	AAPL	2016-02-01	96.430000
4	ABBV	2016-02-01	54.389999
...	...	...	...
10495	YHOO	2016-03-01	32.799999
10496	YUM	2016-03-01	54.773546
10497	ZBH	2016-03-01	97.389999
10498	ZION	2016-03-01	22.309999
10499	ZTS	2016-03-01	42.099998

10500 rows в 3 columns

```
print("Number of rows in melted table: " + str(melted.shape[0]))
```

Number of rows in melted table: 10500

Comparing it with the original data:

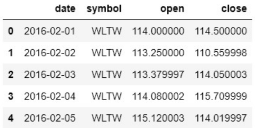
```
print("Number of rows in the original table: " +
str(stocks_small.shape[0]))
Number of rows in the original table: 10500
```

In a nutshell, melt() takes wide dataframes and makes them longer and thinner.

Pandas pivot()

pivot() is the complete opposite of melt(). Sometimes, there will be cases where you want to turn your clean, long formatted data into wide. Let's see how I turned that subset stocks into a wide format. Here is the data to remind you:

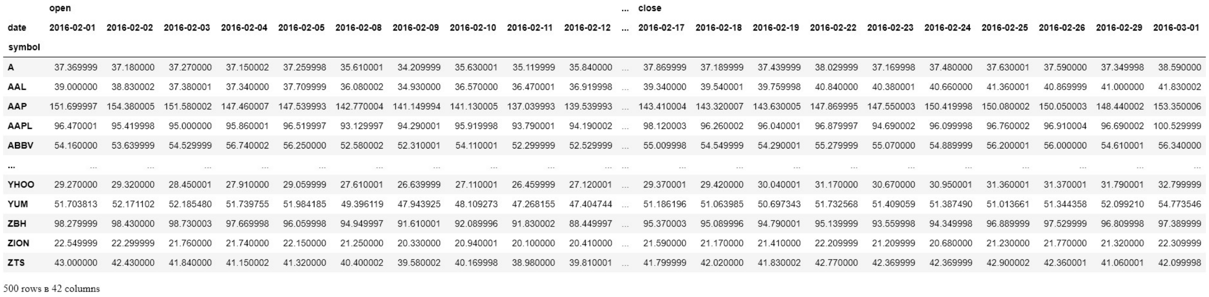
```
stocks_small.head()
```



	date	symbol	open	close
0	2016-02-01	WLTW	114.000000	114.500000
1	2016-02-02	WLTW	113.250000	110.559998
2	2016-02-03	WLTW	113.379997	114.050003
3	2016-02-04	WLTW	114.080002	115.709999
4	2016-02-05	WLTW	115.120003	114.019997

Now, I will pivot the table:

```
pivoted = stocks_small.pivot(index='symbol', columns='date')
>>> pivoted
```



	open										...	close									
date	2016-02-01	2016-02-02	2016-02-03	2016-02-04	2016-02-05	2016-02-08	2016-02-09	2016-02-10	2016-02-11	2016-02-12	...	2016-02-17	2016-02-18	2016-02-19	2016-02-22	2016-02-23	2016-02-24	2016-02-25	2016-02-26	2016-02-29	2016-03-01
symbol																					
A	37.369999	37.180000	37.270000	37.150002	37.259998	35.610001	34.209999	35.630001	35.119999	35.840000	...	37.869999	37.189999	37.439999	38.029999	37.169998	37.480000	37.630001	37.590000	37.349998	38.590000
AAL	39.000000	38.830002	37.380001	37.340000	37.709999	36.080002	34.930000	36.570000	36.470001	36.919998	...	39.340000	39.540001	39.759998	40.840000	40.380001	40.660000	41.360001	40.869999	41.000000	41.830002
AAP	151.699997	154.380005	151.580002	147.460007	147.539993	142.770004	141.149994	141.130005	137.039993	139.539993	...	143.410004	143.320007	143.630005	147.869995	147.550003	150.419998	150.080002	150.050003	148.440002	153.350006
AAPL	96.470001	95.419998	95.000000	95.860001	96.519997	93.129997	94.290001	95.919998	93.790001	94.190002	...	98.120003	96.260002	96.040001	96.879997	94.690002	96.099998	96.760002	96.910004	96.690002	100.529999
ABBV	54.160000	53.639999	54.529999	56.740002	56.250000	52.580002	52.310001	54.110001	52.299999	52.529999	...	55.009998	54.549999	54.290001	55.279999	55.070000	54.889999	56.200001	56.000000	54.610001	56.340000
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
YHOO	29.270000	29.320000	28.450001	27.910000	29.059999	27.610001	26.639999	27.110001	26.459999	27.120001	...	29.370001	29.420000	30.040001	31.170000	30.670000	30.950001	31.360001	31.370001	31.790001	32.799999
YUM	51.703813	52.171102	52.185480	51.739755	51.984185	49.396119	47.943925	48.109273	47.268155	47.404744	...	51.186196	51.063985	50.697343	51.732568	51.409059	51.387490	51.013661	51.344358	52.099210	54.773546
ZBH	98.279999	98.430000	98.730003	97.669998	96.059998	94.949997	91.610001	92.089996	91.830002	88.449997	...	95.370003	95.089996	94.790001	95.139999	93.559998	94.349998	96.889999	97.529999	96.809998	97.389999
ZION	22.549999	22.299999	21.760000	21.740000	22.150000	21.250000	20.330000	20.940001	20.100000	20.410000	...	21.590000	21.170000	21.410000	22.209999	21.209999	20.680000	21.230000	21.770000	21.320000	22.309999
ZTS	43.000000	42.430000	41.840000	41.150002	41.320000	40.400002	39.580002	40.169998	38.980000	39.810001	...	41.799999	42.020000	41.830002	42.770000	42.369999	42.369999	42.900002	42.360001	41.060001	42.099998

500 rows в 42 columns

500 rows, 42 columns

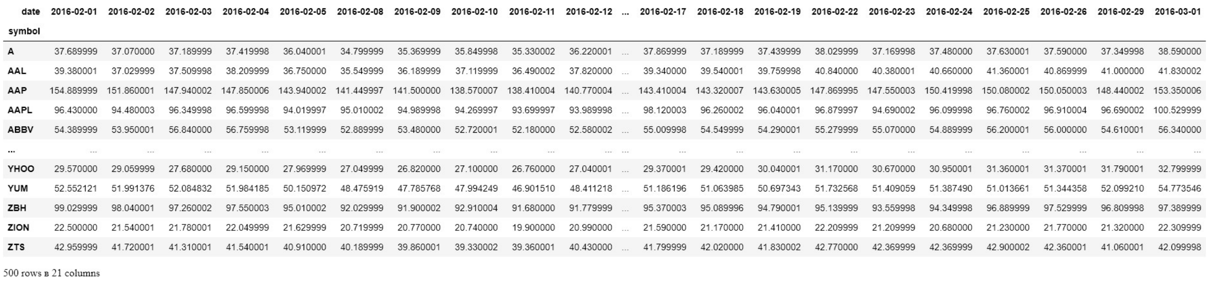
```
>>> pivoted.shape
(500, 42)
```

When you use pivot(), keep these in mind:

1. pandas will take the variable you pass for index parameter and displays its unique values as indexes.
2. pandas will take the variable you pass for columns and display its unique values as separate columns.

If you noticed, the above dataframe is not the one we used with melt(). That's because it also contains the open prices of stocks not just close. In pivot(), there is a parameter called values which if not specified tells pandasto include all of the remaining columns to the pivoted dataframe. Let's choose only the close prices this time:

```
pivoted2 = stocks_small.pivot(index='symbol', columns='date',
values='close')
>>> pivoted2
```



date	2016-02-01	2016-02-02	2016-02-03	2016-02-04	2016-02-05	2016-02-08	2016-02-09	2016-02-10	2016-02-11	2016-02-12	...	2016-02-17	2016-02-18	2016-02-19	2016-02-22	2016-02-23	2016-02-24	2016-02-25	2016-02-26	2016-02-29	2016-03-01
symbol																					
A	37.689999	37.070000	37.189999	37.419998	36.040001	34.799999	35.369999	35.849998	35.330002	36.220001	...	37.869999	37.189999	37.439999	38.029999	37.169998	37.480000	37.630001	37.590000	37.349998	38.590000
AAL	39.380001	37.029999	37.509998	38.209999	36.750000	35.549999	36.189999	37.119999	36.490002	37.820000	...	39.340000	39.540001	39.759998	40.840000	40.380001	40.660000	41.360001	40.869999	41.000000	41.830002
AAP	154.889999	151.860001	147.940002	147.850006	143.940002	141.449997	141.500000	138.570007	138.410004	140.770004	...	143.410004	143.320007	143.630005	147.869995	147.550003	150.419998	150.080002	150.050003	148.440002	153.350006
AAPL	96.430000	94.480003	96.349998	96.599998	94.019997	95.010002	94.989998	94.269997	93.699997	93.989998	...	98.120003	96.260002	96.040001	96.879997	94.690002	96.099998	96.760002	96.910004	96.690002	100.529999
ABBV	54.389999	53.950001	56.840000	56.759998	53.119999	52.889999	53.480000	52.720001	52.180000	52.580002	...	55.009998	54.549999	54.290001	55.279999	55.070000	54.889999	56.200001	56.000000	54.610001	56.340000
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
YHOO	29.570000	29.059999	27.680000	29.150000	27.969999	27.049999	26.820000	27.100000	26.760000	27.040001	...	29.370001	29.420000	30.040001	31.170000	30.670000	30.950001	31.360001	31.370001	31.790001	32.799999
YUM	52.552121	51.991376	52.084832	51.984185	50.150972	48.475919	47.785768	47.994249	46.901510	48.411218	...	51.186196	51.063985	50.697343	51.732568	51.409059	51.387490	51.013661	51.344358	52.099210	54.773546
ZBH	99.029999	98.040001	97.260002	97.550003	95.010002	92.029999	91.900002	92.910004	91.680000	91.779999	...	95.370003	95.089996	94.790001	95.139999	93.559998	94.349998	96.889999	97.529999	96.809998	97.389999
ZION	22.500000	21.540001	21.780001	22.049999	21.629999	20.719999	20.770000	20.740000	19.900000	20.990000	...	21.590000	21.170000	21.410000	22.209999	21.209999	20.680000	21.230000	21.770000	21.320000	22.309999
ZTS	42.959999	41.720001	41.310001	41.540001	40.910000	40.189999	39.860001	39.330002	39.360001	40.430000	...	41.799999	42.020000	41.830002	42.770000	42.369999	42.369999	42.900002	42.360001	41.060001	42.099998

500 rows в 21 columns

500 rows, 21 columns

```
>>> pivoted2.shape
(500, 21)
```

That’s pretty much it for pivot(). Surprisingly, it is one of the hardest functions in pandas and yet it only has 3 parameters, not even an extra parameter to fill missing values.

For pivot(), just remember that it takes two categorical variables and displays their unique values as index and columns. This resulting table will be a grid of those two variables. If the values parameter is not specified, all the remaining columns are given as cell values which will make the table even wider.