

```
In [3]: %%html
<h2>Chapter 7 - Data Cleaning and Preparation</h2>
Code examples are taken from <a href="https://github.com/wesm/pydata-book/blob/3rd-edition/ch07.ipynb">https://gith
```

Chapter 7 - Data Cleaning and Preparation

Code examples are taken from <https://github.com/wesm/pydata-book/blob/3rd-edition/ch07.ipynb>

```
In [6]: import pandas as pd
import numpy as np
```

```
In [8]: %%html
<h3>Handling Missing Data</h3>
```

Handling Missing Data

```
In [9]: float_data = pd.Series([1.2, -3.5, np.nan, 0])
float_data
```

```
Out[9]: 0    1.2
1   -3.5
2    NaN
3    0.0
dtype: float64
```

```
In [10]: float_data.isna()
```

```
Out[10]: 0    False
1    False
2     True
3    False
dtype: bool
```

```
In [11]: data = pd.Series([1, np.nan, 3.5, np.nan, 7])
data
```

```
Out[11]: 0    1.0
         1    NaN
         2    3.5
         3    NaN
         4    7.0
         dtype: float64
```

```
In [16]: data.dropna()
```

```
Out[16]: 0    1.0
         2    3.5
         4    7.0
         dtype: float64
```

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],
                               [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])
         data
         data.dropna()
```

```
Out[19]:
```

	0	1	2
0	1.0	6.5	3.0

```
In [20]: # It will drop only rows that are all NaN
         data.dropna(how="all")
```

```
Out[20]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

```
In [23]: data[4] = np.nan
         data
```

```
Out[23]:
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

```
In [24]: data.dropna(axis="columns", how="all")
```

```
Out[24]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
In [25]: %%html
<h3>Filling Missing Data</h3>
```

Filling Missing Data

```
In [26]: data
```

```
Out[26]:
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

```
In [27]: data.fillna(0.)
```

```
Out[27]:
```

	0	1	2	4
0	1.0	6.5	3.0	0.0
1	1.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	6.5	3.0	0.0

```
In [29]: data
```

```
Out[29]:
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

```
In [31]: data.fillna(method="ffill")
```

/tmp/ipykernel_11050/2321864255.py:1: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill() or obj.bfill() instead.
data.fillna(method="ffill")

```
Out[31]:
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	6.5	3.0	NaN
2	1.0	6.5	3.0	NaN
3	1.0	6.5	3.0	NaN

```
In [50]: %%html
<h3>Data Transformation</h3>
<h4>Removing the duplicates</h4>
```

Data Transformation

Removing the duplicates

```
In [38]: data = pd.DataFrame({"k1": ["one", "two"] * 3 + ["two"],
                             "k2": [1, 1, 2, 3, 3, 4, 4]})

print(data)
data.duplicated()
data.drop_duplicates()
# It removed the last row where both columns values of the row is same as 5th index row
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

Out[38]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

```
In [49]: # We can check for the subset of columns for the duplicates
data["v1"] = range(7)
print(data)
print("")
print("After one column in the subset attribute")
print("")
print(data.drop_duplicates(subset=["k1"]))
```

```
print("")
print("After two columns in the subset attribute")
print("")
print(data.drop_duplicates(subset=["k1","k2"]))
```

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5
6	two	4	6

After one column in the subset attribute

	k1	k2	v1
0	one	1	0
1	two	1	1

After two columns in the subset attribute

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5

```
In [51]: %%html
<h4>Transforming data using a function or mapping</h4>
```

Transforming data using a function or mapping

```
In [52]: data = pd.DataFrame({"food": ["bacon", "pulled pork", "bacon",
                                         "pastrami", "corned beef", "bacon",
                                         "pastrami", "honey ham", "nova lox"],
                              "ounces": [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

data
```

Out[52]:

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	12.0
3	pastrami	6.0
4	corned beef	7.5
5	bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

In [59]:

```
food_to_animal = {
    "bacon": "pig",
    "pulled pork": "pig",
    "pastrami": "cow",
    "corned beef": "cow",
    "honey ham": "pig",
    "nova lox": "salmon"
}

def get_animal(x):
    return food_to_animal[x]

data['animal'] = data['food'].map(get_animal)
data
```

```
Out[59]:
```

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	pastrami	6.0	cow
4	corned beef	7.5	cow
5	bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

```
In [60]: %%html
<h4>Renaming axis indexes</h4>
```

Renaming axis indexes

```
In [62]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
                             index=["Ohio", "Colorado", "New York"],
                             columns=["one", "two", "three", "four"])
data
```

```
Out[62]:
```

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

```
In [67]: def mapping(x):
          return x[:4].upper()

print(data.index.map(mapping))
```



```
data.index = data.index.map(mapping)
data
```

```
Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

```
Out[67]:
```

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

```
In [68]: %%html
<h4>Discretization & binning</h4>
```

Discretization & binning

```
In [ ]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
bins = [18, 25, 35, 60, 100]
# it converts the ages into bins: 18 to 25, 25 to 35, 35 to 60, 60 to 100
age_categories = pd.cut(ages, bins)
age_categories
```

```
In [72]: age_categories.categories
```

```
Out[72]: IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]], dtype='interval[int64, right]')
```

```
In [74]: pd.value_counts(age_categories)
```

```
/tmp/ipykernel_11050/3010498523.py:1: FutureWarning: pandas.value_counts is deprecated and will be removed in a future version. Use pd.Series(obj).value_counts() instead.
pd.value_counts(age_categories)
```

```
Out[74]: (18, 25]      5
(25, 35]      3
(35, 60]      3
(60, 100]      1
Name: count, dtype: int64
```

```
In [78]: # If we pass an integer instead of a list in bin attributes, it will divided the data specified integer no. of bins
```

```
data = np.random.uniform(size=21)
data
pd.cut(data, 4)
```

```
Out[78]: array([0.1079196 , 0.65967945, 0.4119345 , 0.13176666, 0.60006722,
                0.68983366, 0.01658643, 0.5010701 , 0.17507584, 0.3512456 ,
                0.39140835, 0.35190866, 0.32518611, 0.10655427, 0.04213229,
                0.25048114, 0.07727767, 0.8057591 , 0.20699889, 0.88114253,
                0.00606917])
```

```
In [87]: # Creating quartile: 4 => qcut creates the each bin with same number of data points where as cut wont do that
```

```
data = np.random.standard_normal(1000)
quartile = pd.qcut(data, 4, precision=2)
test_with_cut = pd.cut(data, 4, precision=2)
```

```
In [85]: pd.value_counts(quartile)
```

```
/tmp/ipykernel_11050/4228863495.py:1: FutureWarning: pandas.value_counts is deprecated and will be removed in a future version. Use pd.Series(obj).value_counts() instead.
pd.value_counts(quartile)
```

```
Out[85]: (-3.61, -0.72]      250
         (-0.72, -0.0028]    250
         (-0.0028, 0.66]     250
         (0.66, 3.02]        250
         Name: count, dtype: int64
```

```
In [88]: pd.value_counts(test_with_cut)
```

```
/tmp/ipykernel_11050/1120107730.py:1: FutureWarning: pandas.value_counts is deprecated and will be removed in a future version. Use pd.Series(obj).value_counts() instead.
pd.value_counts(test_with_cut)
```

```
Out[88]: (0.0065, 1.47]      439
         (-1.46, 0.0065]    427
         (-2.93, -1.46]      67
         (1.47, 2.94]        67
         Name: count, dtype: int64
```

```
In [89]: %%html
<h4>Detecting and filtering outliers</h4>
```

Detecting and filtering outliers

```
In [91]: data = pd.DataFrame(np.random.standard_normal((1000, 4)))
data.describe()
```

```
Out[91]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.032673	-0.028110	-0.024413	0.024666
std	1.007846	0.993278	1.024149	1.006475
min	-3.357931	-3.009217	-3.035152	-3.130328
25%	-0.626578	-0.701481	-0.704032	-0.700567
50%	0.055235	-0.035837	-0.033858	-0.011936
75%	0.676512	0.661261	0.676005	0.703756
max	3.042454	3.173949	2.972772	4.055141

```
In [94]: col = data[2]
col
```

```
Out[94]: 0    -0.981120
1     2.140322
2     0.830040
3     0.808377
4     0.487802
...
995   -1.622898
996   -1.962332
997   -0.868515
998    1.079087
999   -0.940871
Name: 2, Length: 1000, dtype: float64
```

```
In [93]: col[col.abs() > 3]
```

```
Out[93]: 598    -3.035152  
         Name: 2, dtype: float64
```

```
In [95]: %%html  
<h4>Permutation and random sampling</h4>
```

Permutation and random sampling

```
In [100]: df = pd.DataFrame(np.arange(5 * 7).reshape(5, 7))  
df
```

```
Out[100]:
```

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34

```
In [105]: sampler = np.random.permutation(5)  
sampler
```

```
Out[105]: array([2, 0, 1, 3, 4])
```

```
In [106]: df.iloc[sampler]
```

```
Out[106...
```

	0	1	2	3	4	5	6
2	14	15	16	17	18	19	20
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34

```
In [107... # We can do the same on columns
df[sampler]
```

```
Out[107...
```

	2	0	1	3	4
0	2	0	1	3	4
1	9	7	8	10	11
2	16	14	15	17	18
3	23	21	22	24	25
4	30	28	29	31	32

```
In [108... %%html
<h4>Computing Indicator/Dummy Variables</h4>
```

Computing Indicator/Dummy Variables

```
In [109... df = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
                        "data1": range(6)})
df
```

Out[109...

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

```
In [113... dummies = pd.get_dummies(df["key"])
```

```
In [117... df = df.drop(["key"], axis=1)  
df.join(dummies)
```

Out[117...

	data1	a	b	c
0	0	False	True	False
1	1	False	True	False
2	2	True	False	False
3	3	False	False	True
4	4	True	False	False
5	5	False	True	False

```
In [118... # to make example repeatable  
np.random.seed(12345)
```

```
In [120... %%html  
<h3>String manipulation</h3>  
<h4>String function in pandas</h4>
```

String manipulation

String function in pandas

```
In [121... data = {"Dave": "dave@google.com", "Steve": "steve@gmail.com",  
         "Rob": "rob@gmail.com", "Wes": np.nan}  
data = pd.Series(data)  
data
```

```
Out[121... Dave      dave@google.com  
Steve     steve@gmail.com  
Rob       rob@gmail.com  
Wes              NaN  
dtype: object
```

```
In [123... data.str.contains("gmail")
```


```
Out[123... Dave      False  
Steve      True  
Rob        True  
Wes        NaN  
dtype: object
```

```
In [124... %%html
```

```
Partial listing of series string methods  

```

Partial listing of series string methods

 No description has been provided for this image

```
In [125... data.str.count("gmail")
```

```
Out[125... Dave      0.0  
Steve     1.0  
Rob       1.0  
Wes       NaN  
dtype: float64
```

```
In [131... %%html
<h2>Chapter 8 - Data wrangling: Join, Combine, and Reshape</h2>
Code examples are taken from <a href="https://github.com/wesm/pydata-book/blob/3rd-edition/ch08.ipynb">https://gith
```

Chapter 8 - Data wrangling: Join, Combine, and Reshape

Code examples are taken from <https://github.com/wesm/pydata-book/blob/3rd-edition/ch08.ipynb>

```
In [167... %%html

<h3>Heirarchial Indexing</h3>

It's an important feature in pandas which enables the multiple index levels on an axis.
This is the way pandas can be used to represent higher dimensional data in tabular format
```

Heirarchial Indexing

It's an important feature in pandas which enables the multiple index levels on an axis. This is the way pandas can be used to represent higher dimensional data in tabular format

```
In [136... data = pd.Series(np.random.uniform(size=9),
                    index=[["a", "a", "a", "b", "b", "c", "c", "d", "d"],
                          [1, 2, 3, 1, 3, 1, 2, 2, 3]])

data
```

```
Out[136... a 1    0.929616
          2    0.316376
          3    0.183919
b 1    0.204560
   3    0.567725
c 1    0.595545
   2    0.964515
d 2    0.653177
   3    0.748907
dtype: float64
```

```
In [137... data.index
```


Out[143...

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	3	3	4	5
b	1	6	7	8
	2	9	10	11

In [147...

```
frame.index.names = ["key1", "key2"]
frame.columns.names = ["state", "colour"]
frame
```

Out[147...

		state	Ohio		Colorado
		colour	Green	Red	Green
key1	key2				
a	1	0	1	2	
	3	3	4	5	
b	1	6	7	8	
	2	9	10	11	

In [151...

```
frame["Ohio"]
```

Out[151...

		colour	Green	Red
key1	key2			
a	1	0	1	
	3	3	4	
b	1	6	7	
	2	9	10	


```
Out[165... 0  a      0
           b      7
           c      one
           d      0
           1  a      1
           b      6
           c      one
           d      1
           2  a      2
           b      5
           c      one
           d      2
           3  a      3
           b      4
           c      two
           d      0
           4  a      4
           b      3
           c      two
           d      1
           5  a      5
           b      2
           c      two
           d      2
           6  a      6
           b      1
           c      two
           d      3
dtype: object
```

```
In [160... frame = frame.set_index(["c","d"])
frame
```

Out[160...

	a	b
c d		
one	0 0 7	
	1 1 6	
	2 2 5	
two	0 3 4	
	1 4 3	
	2 5 2	
	3 6 1	

In [166... *# reset index to go back to previous indeing*
frame.reset_index()

Out[166...

	index	a	b	c	d
0	0	0	7	one	0
1	1	1	6	one	1
2	2	2	5	one	2
3	3	3	4	two	0
4	4	4	3	two	1
5	5	5	2	two	2
6	6	6	1	two	3

In [164...

```
Out[164... c    d
one  0    a    0
      1    b    7
      2    a    1
      3    b    6
      4    a    2
      5    b    5
two  0    a    3
      1    b    4
      2    a    4
      3    b    3
      4    a    5
      5    b    2
      6    a    6
      7    b    1
dtype: int64
```

```
In [168... %%html
<h3>Combining and Merging Datasets</h3>
```

Combining and Merging Datasets

```
In [173... df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "a", "b"],
                        "data1": pd.Series(range(7), dtype="Int64")})
df2 = pd.DataFrame({"key": ["a", "b", "d"],
                        "data2": pd.Series(range(3), dtype="Int64")})

print(df1)
df2
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

Out[173...

	key	data2
0	a	0
1	b	1
2	d	2

In [172... `pd.merge(df1,df2)`
*# It does the inner join. We haven't specified which column to match the inner join.
In this case key is the common and it automatically joined the dfs using key column.
Here, a & b are common and they are merged, but c from df1 and d from df2 is removed from the result as they are not common.
we can also perform the merge operation like this df1.merge(df2)*

Out[172...

	key	data1	data2
0	b	0	1
1	b	1	1
2	a	2	0
3	a	4	0
4	a	5	0
5	b	6	1

In [175... *# we can specify different types of join using how attribute: inner, left, right, outer => inner is by default*
`pd.merge(df1,df2, how="left")` *# in this case they merge all the items in df1 with common items in df2*

Out[175...

	key	data1	data2
0	b	0	1
1	b	1	1
2	a	2	0
3	c	3	<NA>
4	a	4	0
5	a	5	0
6	b	6	1

In [178... *# if there are multiple columns have the same in both dfs. We can specify which columns needs to be matched using o*
`pd.merge(df1,df2, on="key", how="outer")`

Out[178...

	key	data1	data2
0	a	2	0
1	a	4	0
2	a	5	0
3	b	0	1
4	b	1	1
5	b	6	1
6	c	3	<NA>
7	d	<NA>	2

In [185... *# In the below example we do not have common columns and in that case, we have to specify which of theses columns n*
`df3 = pd.DataFrame({"lkey": ["b", "b", "a", "c", "a", "a", "b"],
"data1": pd.Series(range(7), dtype="Int64")})
df4 = pd.DataFrame({"rkey": ["a", "b", "d"],
"data2": pd.Series(range(3), dtype="Int64")})
pd.merge(df3,df4,left_on="lkey",right_on="rkey")`


```
Out[185...
```

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	a	2	a	0
3	a	4	a	0
4	a	5	a	0
5	b	6	b	1

```
In [186... # we can merge with multiple keys
left = pd.DataFrame({"key1": ["foo", "foo", "bar"],
                      "key2": ["one", "two", "one"],
                      "lval": pd.Series([1, 2, 3], dtype='Int64')})
right = pd.DataFrame({"key1": ["foo", "foo", "bar", "bar"],
                      "key2": ["one", "one", "one", "two"],
                      "rval": pd.Series([4, 5, 6, 7], dtype='Int64')})
pd.merge(left, right, on=["key1", "key2"], how="outer")
```

```
Out[186...
```

	key1	key2	lval	rval
0	bar	one	3	6
1	bar	two	<NA>	7
2	foo	one	1	4
3	foo	one	1	5
4	foo	two	2	<NA>

```
In [189... # we use index as the merge key sing left_index=True and/or right_index=True

left1 = pd.DataFrame({"key": ["a", "b", "a", "a", "b", "c"],
                      "value": pd.Series(range(6), dtype="Int64")})
right1 = pd.DataFrame({"group_val": [3.5, 7]}, index=["a", "b"])
left1
```

```
right1
pd.merge(left1, right1, left_on="key", right_index=True)
```

```
Out[189...
   key  value  group_val
0    a      0          3.5
1    b      1          7.0
2    a      2          3.5
3    a      3          3.5
4    b      4          7.0
```

```
In [192... # There is a join method that simplify the merging by index
left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
                      index=["a", "c", "e"],
                      columns=["Ohio", "Nevada"]).astype("Int64")
right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
                       index=["b", "c", "d", "e"],
                       columns=["Missouri", "Alabama"]).astype("Int64")

left2.join(right2, how="outer")
```

```
Out[192...
   Ohio  Nevada  Missouri  Alabama
a      1       2      <NA>    <NA>
b  <NA>   <NA>       7         8
c      3       4       9        10
d  <NA>   <NA>      11        12
e      5       6      13        14
```


```
In [213... %%html

<h3>Concatening along an axis</h3>
Pandas.concat function argument

```

Concatening along an axis

Pandas.concat function argument

 No description has been provided for this image

```
In [197...] s1 = pd.Series([0, 1], index=["a", "b"], dtype="Int64")
            s2 = pd.Series([2, 3, 4], index=["c", "d", "e"], dtype="Int64")
            s3 = pd.Series([5, 6], index=["f", "g"], dtype="Int64")
            pd.concat([s1,s2,s3])
```

```
Out[197...] a    0
            b    1
            c    2
            d    3
            e    4
            f    5
            g    6
            dtype: Int64
```

```
In [198...] pd.concat([s1,s2,s3],axis=1)
```

```
Out[198...]    0    1    2
a      0  <NA> <NA>
b      1  <NA> <NA>
c  <NA>    2  <NA>
d  <NA>    3  <NA>
e  <NA>    4  <NA>
f  <NA>  <NA>    5
g  <NA>  <NA>    6
```

```
In [ ]: # we can do the join as welll
```

```
In [200... s4 = pd.concat([s1, s3])
pd.concat([s1,s4],axis=1, join="inner")
```

```
Out[200...    0  1
a  0  0
b  1  1
```

```
In [202... df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=["a", "b", "c"],
                      columns=["one", "two"])
df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=["a", "c"],
                      columns=["three", "four"])

pd.concat([df1, df2], axis="columns")
```

```
Out[202...    one  two  three  four
a     0    1     5.0   6.0
b     2    3     NaN   NaN
c     4    5     7.0   8.0
```

```
In [207... test_df = pd.concat([df1, df2], axis="columns", keys=["level1", "level2"])
test_df
```

```
Out[207...      level1      level2
      one  two  three  four
a     0    1     5.0   6.0
b     2    3     NaN   NaN
c     4    5     7.0   8.0
```

```
In [206... test_df["level1"]
```

```
Out[206...
```

	one	two
a	0	1
b	2	3
c	4	5

```
In [208... pd.concat([df1, df2],keys=["level1", "level2"])
```

```
Out[208...
```

		one	two	three	four
level1	a	0.0	1.0	NaN	NaN
	b	2.0	3.0	NaN	NaN
	c	4.0	5.0	NaN	NaN
level2	a	NaN	NaN	5.0	6.0
	c	NaN	NaN	7.0	8.0

```
In [212... pd.concat([df1, df2],keys=["level1", "level2"], join="inner")
```

```
Out[212...
```

level1	a
	b
	c
level2	a
	c

```
In [214... %%html
<h3>Combining data with overlap</h3>
```

Combining data with overlap

```
In [215... a = pd.Series([np.nan, 2.5, 0.0, 3.5, 4.5, np.nan],
               index=["f", "e", "d", "c", "b", "a"])
b = pd.Series([0., np.nan, 2., np.nan, np.nan, 5.],
               index=["a", "b", "c", "d", "e", "f"])

np.where(pd.isna(a), b, a)
```

```
Out[215... array([0. , 2.5, 0. , 3.5, 4.5, 5. ])
```

```
In [217... np.where(pd.isna(b), a, b)
```

```
Out[217... array([0. , 2.5, 2. , 3.5, 4.5, 5. ])
```

```
In [218... a.combine_first(b)
```

```
Out[218... a    0.0
b    4.5
c    3.5
d    0.0
e    2.5
f    5.0
dtype: float64
```

```
In [220... %%html

<h3>Reshaping & pivoting</h3>
<h4>Reshaping with heirarchial indexing</h4>
```

Reshaping & pivoting

Reshaping with heirarchial indexing

```
In [222... # stack rotates or pivots from the columns in the data to the rows.
# unstack pivots from the rows into the columns.

data = pd.DataFrame(np.arange(6).reshape((2, 3)),
                    index=pd.Index(["Ohio", "Colorado"], name="state"),
                    columns=pd.Index(["one", "two", "three"],
```

```
data                                     name="number"))
```

Out[222... **number one two three**

state			
Ohio	0	1	2
Colorado	3	4	5

```
In [228... result = data.stack()
result
```

Out[228... state number
Ohio one 0
two 1
three 2
Colorado one 3
two 4
three 5
dtype: int64

```
In [227... result.unstack()
```

Out[227... **number one two three**

state			
Ohio	0	1	2
Colorado	3	4	5

```
In [229... # We can specify the levels as during the unstack & stack
df = pd.DataFrame({"left": result, "right": result + 5},
                  columns=pd.Index(["left", "right"], name="side"))
df
```

Out[229...

		side	left	right
		state	number	
Ohio	one		0	5
	two		1	6
	three		2	7
Colorado	one		3	8
	two		4	9
	three		5	10

In [230... `df.index`

Out[230... MultiIndex([('Ohio', 'one'),
('Ohio', 'two'),
('Ohio', 'three'),
('Colorado', 'one'),
('Colorado', 'two'),
('Colorado', 'three')],
names=['state', 'number'])

In [234... *# from rows to column*
`df.unstack(level="state")`

Out[234...

		side		left		right	
		state	Ohio	Colorado	Ohio	Colorado	
		number					
one	0		3	5	8		
	1		4	6	9		
	2		5	7	10		

In [235... `df.unstack(level="number")`


```
Out[235...
```

	side	left			right		
	number	one	two	three	one	two	three
	state						
	Ohio	0	1	2	5	6	7
	Colorado	3	4	5	8	9	10

```
In [237... #From column to rows
df
```

```
Out[237...
```

	side	left	right
state	number		
Ohio	one	0	5
	two	1	6
	three	2	7
Colorado	one	3	8
	two	4	9
	three	5	10

```
In [238... df.stack(level="side")
```

```
Out[238... state    number  side
Ohio      one     left    0
           two     right   5
           three  left    1
           three  right   6
           three  left    2
           three  right   7
Colorado  one     left    3
           two     right   8
           two     left    4
           three  right   9
           three  left    5
           three  right  10

dtype: int64
```

```
In [239... %%html

<h3>Pivoting long to wide format</h3>
```

Pivoting long to wide format

```
In [247... # Pivoting from long to wide format

data = pd.read_csv("examples/macrodta.csv")
data = data.loc[:, ["year", "quarter", "realgdp", "infl", "unemp"]]
data.head()
```

```
Out[247...   year  quarter  realgdp  infl  unemp
0  1959         1  2710.349   0.00    5.8
1  1959         2  2778.801   2.34    5.1
2  1959         3  2775.488   2.74    5.3
3  1959         4  2785.204   0.27    5.6
4  1960         1  2847.699   2.31    5.2
```

```
In [248... periods = pd.PeriodIndex(year=data.pop("year"),
                           quarter=data.pop("quarter"),
                           name="date")

periods
data.index = periods.to_timestamp("D")
data.head()
```

/tmp/ipykernel_11050/539240372.py:1: FutureWarning: Constructing PeriodIndex from fields is deprecated. Use PeriodIndex.from_fields instead.

```
periods = pd.PeriodIndex(year=data.pop("year"),
```

Out[248... **realgdp** **infl** **unemp**

date			
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

```
In [249... data = data.reindex(columns=["realgdp", "infl", "unemp"])
data.columns.name = "item"
data.head()
```

Out[249... **item** **realgdp** **infl** **unemp**

date			
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

```
In [253... long_data = (data.stack()
                .reset_index()
                .rename(columns={0: "value"}))

long_data
# Normally this would be the way the data is stored in the database.
# We can transform or pivot this result in such a way it can be much easier to process
```

```
Out[253...
```

	date	item	value
0	1959-01-01	realgdp	2710.349
1	1959-01-01	infl	0.000
2	1959-01-01	unemp	5.800
3	1959-04-01	realgdp	2778.801
4	1959-04-01	infl	2.340
...
604	2009-04-01	infl	3.370
605	2009-04-01	unemp	9.200
606	2009-07-01	realgdp	12990.341
607	2009-07-01	infl	3.560
608	2009-07-01	unemp	9.600

609 rows × 3 columns

```
In [252... long_data.pivot(index="date", columns="item", values="value")
```

Out[252...

item	infl	realgdp	unemp
date			
1959-01-01	0.00	2710.349	5.8
1959-04-01	2.34	2778.801	5.1
1959-07-01	2.74	2775.488	5.3
1959-10-01	0.27	2785.204	5.6
1960-01-01	2.31	2847.699	5.2
...
2008-07-01	-3.16	13324.600	6.0
2008-10-01	-8.79	13141.920	6.9
2009-01-01	0.94	12925.410	8.1
2009-04-01	3.37	12901.504	9.2
2009-07-01	3.56	12990.341	9.6

203 rows × 3 columns

In [254...

```
long_data["value2"] = np.random.standard_normal(len(long_data))
long_data[:10]
```

Out[254...

	date	item	value	value2
0	1959-01-01	realgdp	2710.349	1.248804
1	1959-01-01	infl	0.000	0.774191
2	1959-01-01	unemp	5.800	-0.319657
3	1959-04-01	realgdp	2778.801	-0.624964
4	1959-04-01	infl	2.340	1.078814
5	1959-04-01	unemp	5.100	0.544647
6	1959-07-01	realgdp	2775.488	0.855588
7	1959-07-01	infl	2.740	1.343268
8	1959-07-01	unemp	5.300	-0.267175
9	1959-10-01	realgdp	2785.204	1.793095

In [257...

```
# if we omit the last argument(values), then we get df with hierarchical columns
pivoted = long_data.pivot(index="date", columns="item")
pivoted.head()
```

Out[257...

	value			value2		
item	infl	realgdp	unemp	infl	realgdp	unemp
date						
1959-01-01	0.00	2710.349	5.8	0.774191	1.248804	-0.319657
1959-04-01	2.34	2778.801	5.1	1.078814	-0.624964	0.544647
1959-07-01	2.74	2775.488	5.3	1.343268	0.855588	-0.267175
1959-10-01	0.27	2785.204	5.6	-0.652929	1.793095	-1.886837
1960-01-01	2.31	2847.699	5.2	0.644448	1.059626	-0.007799

In [258...

```
pivoted["value"]
```

Out[258...

item	infl	realgdp	unemp
------	------	---------	-------

date

1959-01-01	0.00	2710.349	5.8
1959-04-01	2.34	2778.801	5.1
1959-07-01	2.74	2775.488	5.3
1959-10-01	0.27	2785.204	5.6
1960-01-01	2.31	2847.699	5.2
...
2008-07-01	-3.16	13324.600	6.0
2008-10-01	-8.79	13141.920	6.9
2009-01-01	0.94	12925.410	8.1
2009-04-01	3.37	12901.504	9.2
2009-07-01	3.56	12990.341	9.6

203 rows × 3 columns

In [259... pivoted["value"]["infl"]

Out[259...

```
date
1959-01-01    0.00
1959-04-01    2.34
1959-07-01    2.74
1959-10-01    0.27
1960-01-01    2.31
...
2008-07-01   -3.16
2008-10-01   -8.79
2009-01-01    0.94
2009-04-01    3.37
2009-07-01    3.56
Name: infl, Length: 203, dtype: float64
```

```
In [260... # Pivoting from wide to long format
```

```
In [261]: df = pd.DataFrame({"key": ["foo", "bar", "baz"],
                             "A": [1, 2, 3],
                             "B": [4, 5, 6],
                             "C": [7, 8, 9]})

df
```

Out[261...	key	A	B	C	
	0	foo	1	4	7
	1	bar	2	5	8
	2	baz	3	6	9

```
In [262... melted = pd.melt(df, id_vars="key")
melted
```

	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6
6	foo	C	7
7	bar	C	8
8	baz	C	9

```
In [263... reshaped = melted.pivot(index="key", columns="variable",
                             values="value")
reshaped
```


Out[263...

variable	A	B	C
key			
bar	2	5	8
baz	3	6	9
foo	1	4	7

In []: