Chapter 5: Getting started with pandas

p. 123 - 165

"While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by con- trast, is best suited for working with homogeneous numerical array data."

```
Remember:

 import pandas as pd
         &
 from pandas import Series, DataFrame
```

5.1 Introduction to pandas Data Structures

```
The two data structures common with pandas are called "Series", & "Data Structures"
```

In [1]:
```python
import pandas as pd

from pandas import Series, DataFrame

import numpy as np
```

Series

```
A one-dimensional array-like object containing a sequence of values, and an associa
ted array
of data labels, called its index.
Series is a fixed length, ordered dict.
```

In [2]:
```
obj = pd.Series([4, 7, -5, 3])
obj
```

Out[2]:
```
0    4
1    7
2   -5
3    3
dtype: int64
```

In [3]:
```
obj.values
```

Out[3]:
```
array([ 4,  7, -5,  3], dtype=int64)
```

In [4]:
```
obj.index # like range(4)
```

Out[4]:
```
RangeIndex(start=0, stop=4, step=1)
```

In [3]:
```
# Can also set an index yourself in a Series, and it doesn't have to be intege
rs! Or ordered!
# Create a series with an index identifying each data point with a label
obj2 = pd.Series([4, 7, -5, 2], index = ['d', 'b', 'a', 'c'])
obj2
```

Out[3]:
```
d    4
b    7
a   -5
c    2
dtype: int64
```

In [6]:
```
obj2.index
```

Out[6]:
```
Index(['d', 'b', 'a', 'c'], dtype='object')
```

In [10]:
```
# You can use labels in the index when selecting single values or a set of val
ues
obj2['a']

# and re-write the values using an index, like a key
obj2['d'] = 6
obj2[['c', 'a', 'd']]
```

Out[10]:
```
c    2
a   -5
d    6
dtype: int64
```

Wtf? pg 125

For # 11, what is this, and why is it in this order??
ANSWER: NO REASON, or, its by the index??

In [4]:
```
obj2[obj2 > 0] ive me the 2nd and one wards
# give me all the values that are greated than 0
```

Out[4]:
```
d    4
b    7
c    2
dtype: int64
```

In [13]:
```
obj2 * 2
```

Out[13]:
```
d    12
b    14
a   -10
c     4
dtype: int64
```

In [2]:
```
import numpy as np
np.exp(obj2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-c573edae633a> in <module>
      1 import numpy as np
----> 2 np.exp(obj2)

NameError: name 'obj2' is not defined
```

In [17]:
```
# Can think of a Series also like a fixed-length, ordered dict.
# 'b' in obj2
'e' in obj2
```

Out[17]:
```
False
```

In [6]:
```
# You can create a Series from a dict like this:
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000} # this
 is the dict
obj3 = pd.Series(sdata)  #like dis, u 'pass it through' and convert it to a Se
ries

obj3
```

Out[6]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

In [7]:
```
# When you do this, the Series will have the dicts key in a particular order.
# You can reorder the "keys"/indexes like this:

states = ['California', 'Ohio', 'Oregon', 'Texas'] # make a list with the righ
t order u want

obj4 = pd.Series(sdata, index = states) # then apply!
obj4

#Notice that since Utah from sdata was not includ in the list u made for index
es, tis not included in the final output either
```

Out[7]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

In [8]:
```
# Can us isnull OR notnull to detect missing values! - very common tool!

# isnull and notnull functions in panda can be used to detect missing data
pd.isnull(obj4)

# This says there IS missing data for California, but not the others
```

Out[8]:
```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

In [ ]:
```
pd.notnull(obj4) # Does the opposite
```

In [23]:
```
# Series package also has a similar method
obj4.isnull()
```

Out[23]:
```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

A useful Series feature for many applications is that it automatically aligns by in
dex label
in arithmetic operations:

In [9]:
```
obj3
obj4
obj3 + obj4

# See how the addition was automatically aligned by the index?  e.g. Ohio from
obj3 + Ohio from obj4
# SIMILAR TO SQL - this is like a FULL outer join!
```

Out[9]:
```
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

In [25]:
```
# You can name the Series itself, and/or the index and the
obj4.name = 'population'    # naming just the Series
obj4.index.name = 'state'  # naming hust the Index

obj4
```

Out[25]:
```
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

In [28]:
```
# A series' index can be altered in-place by assignment
obj
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
obj  # before, the index was just 0 - 3
```

Out[28]:
```
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

Data Frame

```
They have both a row and column index.
There are many ways to construct a DataFrame. One common way is from a dict of equa
l-length
lists or Numpy arrays:
```

In [11]:
```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
frame

##THERE IS A JUPYTER NOTEBOOK EXTENSION THAT ALLOWS YOU TO SORT THE COLUMNS!
```

Out[11]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

In [12]:
```
# can use the head or tail to select the first or last five, e.g.
frame.head()
```

Out[12]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

In [5]:
```
# You can *specify* the order of the columns in a data frame:
pd.DataFrame(data, columns = ['year', 'state', 'pop'])
```

Out[5]:

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

In [6]:
```
# if you pass a column that isn’t in the dict, it will appear with missing val
ues in the result:
```

In [13]:
```
frame2 = pd.DataFrame(data, columns = ['year', 'state', 'pop', 'debt'],
                     index = ['one', 'two', 'three', 'four', 'five', 'six'])
frame2
```

Out[13]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN
six	2003	Nevada	3.2	NaN

In [15]:
```
frame2.columns

# A column in a dataframe can be retrived, like with a Series or other dict-li
ke notations

frame2['state']  # a dict like notation to retrive


# or if its an int then you can do

frame2.state  # OR an attribute-like notation to retrieve! Only works if the n
ame of col is Python compliant, e.g. no spaces
              # you cannoy create a new column using this attribute-like metho
d
```

Out[15]:
```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

In [16]:
```
# Rows can also be retrived in a Dataframe
frame2.loc['three']

# LOC = RETRIEVE ROWS, as a series
```

Out[16]:
```
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

In [10]:
```
# Columns can be modify by assignment:

frame2['debt'] = 16.5
frame2

# See how it added this debt value to ALL rows? Thats because u only set 1 val
ue, and it applies to all
```

Out[10]:

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5
six	2003	Nevada	3.2	16.5

In [17]:
```
# U can also do this

frame2['debt'] = np.arange(6.)  # set debt values to be a range from 0 to 6
frame2
```

Out[17]:

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0
six	2003	Nevada	3.2	5.0

In [18]:
```
# If you set some value, e.g. like above with a range, it will match the lengt
h of the column/row
val = pd.Series([-1.2, -1.5, -1.7], index = ['two', 'four', 'five'])
frame2['debt'] = val  # U NEED THIS DICT LIKE NOTATION TO CREATE, cant us the
 attribute like one.
frame2
```

Out[18]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

In [13]:
```
# Assigning a colunm that doesnt exist creates new col.
# The keyword del will delete a col. as with a dict. Ex"
frame2['eastern'] = frame2.state == "Ohio"

#here, we first add a new column that has boolean value,
#Tells is to add mew col "eastern" and for it to be true for ever Ohio item.
```

In [16]:
```
# Now we use del method to remove this column like this:

del frame2['eastern']

frame2.columns  #And we execute this to check the name of all columns in the d
ata frame
```

Out[16]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

In [20]:
```
# Another common form of data is a nested dict of dicts... !
 #Here it is the dict of the population and its values NESTED within the dict
 of the STATE

pop = {'Nevada': {2001: 2.4, 2002: 2.9},
      'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

        # Outer dict keys here are the names of the columns, (the states)
        # Inner dict keys are the indexes, the year and other integers

# Can be passed to a Data Frame
frame3 = pd.DataFrame(pop)
frame3
```

Out[20]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

In [20]:
```
# Can transpse the DF (swap rows and columns) with syntax similar to Numpy arr
ay
frame3.T
```

Out[20]:

	2001	2002	2000
Nevada	2.4	2.9	NaN
Ohio	1.7	3.6	1.5

In [21]:
```
# The keys in the inner dicts (the year&pop.) are combined and sorted to form
 the index in the results
pd.DataFrame(pop, index = [2001, 2002, 2003])


# AKA we replaced the previous dicts that were converted to indexes(cuz that's
that DFs in Python use)
# To make these new ones...think of "pop" as REPLACE
```

Out[21]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

In [22]:
```
# Dicts of Series are treated in pretty much the same way as in a DF
pdata = {'Ohio': frame3['Ohio'][:-1], # take away the last year?
         'Nevada': frame3['Nevada'][:2]} # give the last values only??
pd.DataFrame(pdata)

## ???????? Why is mine in a different order? And the Values are diff then in
 the book?
## p. 133...
```

Out[22]:

	Ohio	Nevada
2001	1.7	2.4
2002	3.6	2.9

In [23]:
```
# Index an Columns of a DF will also be displayed if theyve been set
frame3.index.name = 'year'; frame3.columns.name = 'state'
frame3
```

Out[23]:

state	Nevada	Ohio
year		
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

In [24]:
```
# the 'values' attribute returns the data contained in the DF as a 2d array
frame3.values
```

Out[24]:
```
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])
```

In [25]:
```
frame2.values
```

Out[25]:
```
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

See Table 5-1 for data inputs for DataFrame

Index Objects

These hold axis labels and other metadata (like names).
The labels in a Series or DF get converted to an Index.
Index objects are immutatble, cannot be modified by the user.

In [26]:
```
obj = pd.Series(range(3), index = ['a', 'b', 'c'])
index = obj.index
index

# Here we are saying, set obj to be a pandas Series, with 3 values, which had
 the index of the following ^
# Then we re-name it to 'index'
```

Out[26]:
```
Index(['a', 'b', 'c'], dtype='object')
```

In [27]:
```
# Give me the 2nd and onwards index value (# not at cuz its 1st/ 0)
index[1:]
```

Out[27]:
```
Index(['b', 'c'], dtype='object')
```

Index objects are immutable and cannot be modified by the user!

In [28]:
```
index[1] = 'd'  # This will not work, cuz immutable!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-39d16ef18702> in <module>
----> 1 index[1] = 'd'  # This will not work, cuz immutable!

~\anaconda3\lib\site-packages\pandas\core\indexes\base.py in __setitem__(sel
f, key, value)
   3908
   3909     def __setitem__(self, key, value):
-> 3910         raise TypeError("Index does not support mutable operations")
   3911
   3912     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

Immutability makes it safer to share Index objects among data structures.

In [30]:
```
import numpy as np

labels = pd.Index(np.arange(3))
labels

# Make labels an Index, with 3 values
```

Out[30]:
```
Int64Index([0, 1, 2], dtype='int64')
```

In [33]:
```
obj2 = pd.Series([1.5, -2.5, 0], index = labels)
obj2

#This says, create obj2 as a Series, with the following values, and set the in
dex values to the prev. defined "labels"
```

Out[33]:
```
0    1.5
1   -2.5
2    0.0
dtype: float64
```

In [34]:
```
obj2.index is labels
```

Out[34]:
```
True
```

In [37]:
```
# Index behave like arrays, and also like a fixed-size set
frame3.columns
```

Out[37]:
```
Index(['Nevada', 'Ohio'], dtype='object', name='state')
```

In [39]:
```
#'Ohio' in frame3.columns
2003 in frame3.columns
```

Out[39]:
```
False
```

In [40]:
```
# a pandas Index can containt duplicate labels (unlike Python sets)
dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
dup_labels

# with pandas Index, it will recognize the duplicates and its own, unique inde
x...but y tf would u want that
```

Out[40]:
```
Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

See Table 5.2 pg 136 for table on Index methods & properites

5.2 Essential Functionatlity

Reindexing

You can re/index, but this creates a new object with the data conformed to a new index... see example below.

In [4]:
```
# Reindexing example
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index = ['d', 'b', 'a', 'c'])
obj
```

Out[4]:
```
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

In [5]:
```
# Reindex will rearrange the data according to the new index, and intrduce missing values for those not yet defined
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
obj2
```

Out[5]:
```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

Indexes are really important when you are working with Time Series

In that case, the indexes will most likely be a date!

In [7]:
```
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
obj2
```

Out[7]:
```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

In [22]:
```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index = [0, 2, 4])
obj3
```

Out[22]:
```
0      blue
2    purple
4    yellow
dtype: object
```

In [23]:
```
obj3.reindex(range(6), method = 'ffill') #Forward Filling. Gets the values fro
m the index before!
```

Out[23]:
```
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

In [6]:
```
# Reindex can calter the (row) index, columns, or both
# When passed only a sequence, it reindexes the rows in the results

frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                    index = ['a', 'c', 'd'],
                    columns = ['Ohio', 'Texas', 'California'])   # how you ren
ame columns
frame
```

Out[6]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

In [10]:
```
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
frame2
```

Out[10]:

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

In [11]:
```
states = ['Texas', 'Utah', 'California']
frame.reindex(columns = states)
```

Out[11]:

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

See Table 5-3 for Reindex function arguments, pg 138

Dropping Entries from an Axis

The 'drop' method will return a new object with the indicate value or values delete
d from the axis.

In [24]:
```
obj = pd.Series(np.arange(5.), index = ['a', 'b', 'c', 'd', 'e'])
obj
```

Out[24]:
```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [14]:
```
new_obj = obj.drop('c')
new_obj
```

Out[14]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

In [15]:
```
obj.drop(['d', 'c'])
```

Out[15]:
```
a    0.0
b    1.0
e    4.0
dtype: float64
```

In [25]:
```
# Index values in DF can be deleted from either axis...
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index = ['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns = ['one', 'two', 'three', 'four'])
data
```

Out[25]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [26]:
```
# Calling 'drop' w. a sequence of labels will drop values from the row labels
 axis0
data.drop(['Colorado', 'Ohio'])
```

Out[26]:

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

In [27]:
```
# You can drop values from the col. by passing axis=1, or axis ='columns'
data.drop('two', axis = 1)

#axis=1 or axis='column' both work for dropping the column, otherwise drop rem
oves rows!

# Note that this only displays a COPY of the DF, not make a new one!
```

Out[27]:

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

In [19]:
```
data
```

Out[19]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [20]:
```
data.drop(['two', 'four'], axis = 'columns')
```

Out[20]:

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

In [21]:
```
obj.drop('c', inplace = True)  # This does it without printint it out/ returni
ng a new object
obj
```

Out[21]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

Indexing, Selection, and Filtering

```
Filtering data- very common!
Similar to NumPy, but the only difference is, the END POINT IS INCLUDED! For NumPy,
it is not!
```

In [29]:
```
# Can use a Serie's index values instead of only integers. e.g.:
import numpy as np
obj = pd.Series(np.arange(4.), index = ['a', 'b', 'c', 'd'])
obj
```

Out[29]:
```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

In [4]:
```
obj['b']

#is the same as obj[1]
```

Out[4]:
```
1.0
```

In [5]:
```
obj[2:4]
```

Out[5]:
```
c    2.0
d    3.0
dtype: float64
```

In [6]:
```
obj[['b', 'a', 'd']]
```

Out[6]:
```
b    1.0
a    0.0
d    3.0
dtype: float64
```

In [7]:
```
obj[[1, 3]]
```

Out[7]:
```
b    1.0
d    3.0
dtype: float64
```

In [8]:
```
obj[obj < 2]
```

Out[8]:
```
a    0.0
b    1.0
dtype: float64
```

In [9]:
```
# Slicing labels is a bit dif than with typical Python slicing. Here, the endp
oint is inclusive
obj['b':'c']
```

Out[9]:
```
b    1.0
c    2.0
dtype: float64
```

In [10]:
```
# U can reset the values
obj['b':'c'] = 5
obj
```

Out[10]:
```
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

In [30]:
```
# Indexing into a Dataframe is for retreiving 1 or more columnes either with a
single vlue or sequence

data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                   index = ['Ohio', 'Colorado', 'Utah', 'New York'],
                   columns = ['one', 'two', 'three', 'four'])
data
```

Out[30]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [31]:
```
data['two']
```

Out[31]:
```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int32
```

In [15]:
```
data[['three', 'one']] # use double brackets for selecting multiple!
```

Out[15]:

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

Some special cases with indexing:

In [16]:
```
data[:2]
```

Out[16]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

In [17]:
```
data[data['three'] > 5] # only show the values that are greate than 5, for the
THREE column...
```

Out[17]:

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [18]:
```
# Can also index with booleanz
data < 5
```

Out[18]:

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

In [20]:
```
data[data < 5] = 0  # change all values less than 5 to 0
data
```

Out[20]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Selection with loc and iloc - label-indexing on the rows for DataFrame

- they allow you to select a subset of the rows and columnes from a DataFrame with NumPy like notation, using either axis labels (loc) or intergers (iloc)
- both of these work with slicing

Again:

- with .loc you are selecting using the string names
- with .iloc, you are selecting with integers/more similar to NumPy
- YOU CANNOT COMBINE SYNTAX!

In [32]:
```
# Example: select a single row and multiple columns by label:
data.loc['Colorado', ['two', 'three']]
```

Out[32]:
```
two      5
three    6
Name: Colorado, dtype: int32
```

In [22]:
```
data.iloc[2, [3, 0, 1]] # from the 3rd row, give me the 4th, 1st and 2nd value
s
```

Out[22]:
```
four    11
one      8
two      9
Name: Utah, dtype: int32
```

In [23]:
```
data.iloc[2] # give me the 3rd row, aka Utah
```

Out[23]:
```
one       8
two       9
three    10
four     11
Name: Utah, dtype: int32
```

In [24]:
```
data.iloc[[1, 2], [3, 0, 1]]
```

Out[24]:

	four	one	two
Colorado	7	0	5
Utah	11	8	9

In [25]:
```
data.loc[:'Utah', 'two'] # Give me everything up until Utach, from column 2
```

Out[25]:
```
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int32
```

In [26]:
```
data.iloc[:, :3][data.three > 5]
```

Out[26]:

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

Integer Indexes

In [27]:
```
ser = pd.Series(np.arange(3.))
ser


# ser[-1] DOES NOT WORK,
```

Out[27]:
```
0    0.0
1    1.0
2    2.0
dtype: float64
```

In [28]:
```
# The last in the above would work, if u have an non-integer index...whut
ser2 = pd.Series(np.arange(3.), index = ['a', 'b', 'c'])
ser2[-1]
```

Out[28]:
```
2.0
```

In [34]:
```
ser2
```

Out[34]:
```
a    0.0
b    1.0
c    2.0
dtype: float64
```

In [31]:
```
# If you have an axis index containing integers, data selection will always be label-oriented
# use loc for labels, and iloc for integers
ser[:1]

# Above it standard Python, reads as, give me everything up until the 2nd row (EXCLUDING the 2nd row)
```

Out[31]:
```
0    0.0
dtype: float64
```

In [32]:
```
ser.loc[:1]

# loc is different than standard Python, reads as: give me everything up til the 2nd, INCLUDING the 2nd row
```

Out[32]:
```
0    0.0
1    1.0
dtype: float64
```

In [33]:
```
ser.iloc[:1] ## ??????? pg 146

# But iloc is more like standard python...whut, why?
```

Out[33]:
```
0    0.0
dtype: float64
```

Arithmetic and Data Alignment

If the indexes don't match, then the result will just be NaN

In [39]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index = ['a', 'c', 'd', 'e'])
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
              index = ['a', 'c', 'e', 'f', 'g'])
```

In [42]:
```
 s1 + s2  #notice that where there is an identical index in both, it shows NaN

    # aka is dose "Internal data alignment". and is performed on both rows and columns
```

Out[42]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

In [44]:
```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns = list('bcd'),
                   index = ['Ohio', 'Texas', 'Colorado'])
df2 = pd.DataFrame(np.arange(12.). reshape((4, 3)), columns = list('bde'),
                   index = ['Utah', 'Ohio', 'Texas', 'Oregon'])

df1
```

Out[44]:

	b	c	d
Ohio	0.0	1.0	2.0
Texas	3.0	4.0	5.0
Colorado	6.0	7.0	8.0

In [45]:
```
df2
```

Out[45]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [46]:
```
# Adding these together returns a DataFrame whose index and columns are the un
ions of the ones in each DataFrame:
df1 + df2
```

Out[46]:

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

In [47]:
```
# If you add DataFrame objects with no column or row labels common, the result
will contain nulls/NaN

df1 = pd.DataFrame({'A': [1, 2]})
df2 = pd.DataFrame({'B': [3, 4]})

df1
```

Out[47]:

	A
0	1
1	2

In [48]:
```
df2
```

Out[48]:

	B
0	3
1	4

In [49]:
```
df1 - df2
```

Out[49]:

	A	B
0	NaN	NaN
1	NaN	NaN

Arithmetic methods and fill values

Sometimes you'll need to use a fill value, like 0, when working with differently indexed objects.

In [4]:
```
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
                   columns = list ('abcd'))

df1
```

Out[4]:

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

In [33]:
```
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
                   columns = list('abcde'))
df2
```

Out[33]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

In [8]:
```
df2.loc[1, 'b'] = np.nan  # Change the item in the second row, col b
df2
```

Out[8]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	NaN	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

In [9]:
```
# Adding the two frames above results in NA values in the locations where they
dont overlap
df1 + df2
```

Out[9]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	NaN	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

In [11]:
```
# You can use the add method on df1, tp pass df2 an argument to fill_value
df1.add(df2, fill_value = 0) # e.g. place a 0 in place of all the places where
it doesnt match, and add them togetha
```

Out[11]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	5.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

```
See Table 5.5 on pg 149 for table of arithmetic methods, they begin with an "r", fo
r Series and Dataframes
```

In [12]:
```
# These two are the same!

# 1 / df1

df1.rdiv(1)
```

Out[12]:

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

In [13]:
```
# You can also use a fill value when reindexing a Series or a Dataframe!
# Useful when you are reindexing, and the index AKA COLs dont match!

df1.reindex(columns = df2.columns, fill_value = 0)
```

Out[13]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	0
1	4.0	5.0	6.0	7.0	0
2	8.0	9.0	10.0	11.0	0

Operations between DataFrame and Series

In [14]:
```
arr = np.arange(12.).reshape((3, 4))
arr
```

Out[14]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

In [15]:
```
arr[0]
```

Out[15]:
```
array([0., 1., 2., 3.])
```

In [16]:
```
arr - arr[0]


#See Below for more explanation!
# What this is, is it sub. arr[0] from EACH row in arr...lolz bad example cuz
 confusing a.f.
```

Out[16]:
```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

The above is: When we subtract arr[0] from arr, the subtraction is performed once for each row. This is referred to as broadcasting and is explained in more detail as it relates to gen- eral NumPy arrays in Appendix A. Operations between a DataFrame and a Series are similar:

In [34]:
```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                    columns = list('bde'),
                    index = ['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[34]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [35]:
```
series = frame.iloc[0] # what did this do again??  # Normally iloc should just
be a index selector? # OH!
# the above line, we are saying that series is EQUAL to the first row in frame
```

In [23]:
```
series
```

Out[23]:
```
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

In [36]:
```
frame - series  # this is doable, because its automatically broadcasted, when
 u do it between
# a DataFrame and Series, with matched indexes!
# Just so long as the Series indexes match the DFs columns!
```

Out[36]:

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

In [26]:
```
# If the index value is not in either the DF's col, or the Series'index, the o
bjects will be
# reindexed to form the union

series2 = pd.Series(range(3), index = ['b', 'e', 'f'])
series2
```

Out[26]:
```
b    0
e    1
f    2
dtype: int64
```

In [27]:
```
frame + series2
```

Out[27]:

	b	d	e	f
Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

In [28]:
```
# If you want to broadcast over the columns & matching the rows, u have to use
the arithmetic methods
series3 = frame['d']
frame
```

Out[28]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [29]:
```
series3
```

Out[29]:
```
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64
```

In [30]:
```
frame.sub(series3, axis = 'index') # sub from all columns in frame, with serie
s3
```

Out[30]:

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

```
The axis number that you pass is the axis to match on. In this case we mean to matc
h
on the DataFrame’s row index (axis='index' or axis=0) and broadcast across.
```

Function Application and Mapping

```
NumPy ufuncs also work with pandas objects
```

In [37]:
```
frame = pd.DataFrame(np.random.randn(4, 3), columns = list('bde'),
                    index = ['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[37]:

	b	d	e
Utah	-0.949673	-0.444189	-1.370146
Ohio	-0.906499	-0.255191	-0.763454
Texas	0.626443	-0.475932	1.661539
Oregon	-0.062912	0.453811	0.568546

In [33]:
```
np.abs(frame) # Take the absolute value
```

Out[33]:

	b	d	e
Utah	0.882065	0.777160	0.511517
Ohio	0.933985	2.280215	0.372572
Texas	0.565829	0.073267	0.369304
Oregon	0.851754	0.185753	0.019222

In [35]:
```
# Can use the 'apply' method to DataFrames to use functions
f = lambda x: x.max() - x.min()
frame.apply(f)


# The f function calculates the diff between the max and min of a series, once
on each column in frame.
```

Out[35]:
```
b    1.816050
d    2.465968
e    0.492295
dtype: float64
```

In [36]:
```
# If you do 'axis = 'columns' to 'apply', the function will be invoked once pe
r row instead
frame.apply(f, axis = 'columns')
```

Out[36]:
```
Utah      1.659224
Ohio      2.652788
Texas     0.492563
Oregon    1.037507
dtype: float64
```

In [ ]:
```
# For common array stats (like sum and mean), are DataFrame methods, so no nee
d to use apply.
```

In [37]:
```
# Functions passed to apply can also return on a Series with multiple value
def f(x):
    return pd.Series([x.min(), x.max()], index = ['min', 'max'])  # give the m
in and max value of the frame
frame.apply(f)
```

Out[37]:

	b	d	e
min	-0.882065	-0.185753	-0.511517
max	0.933985	2.280215	-0.019222

In [ ]:
```
# Can also use df.applymap ...???
```

Sorting and Ranking

Use 'sort_index' method, which reutrns a new, sorted objects

In [2]:
```
obj = pd.Series(range(4), index = ['d', 'a', 'b', 'c'])
obj
```

Out[2]:
```
d    0
a    1
b    2
c    3
dtype: int64
```

In [3]:
```
obj.sort_index()

# Can also sort by values,
obj.sort_values()
```

Out[3]:
```
a    1
b    2
c    3
d    0
dtype: int64
```

In [39]:
```
# With a DataFrame, you can sort by index on either axis
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                    index = ['three', 'one'],
                    columns = ['d', 'a', 'b', 'c'])
frame
```

Out[39]:

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

In [5]:
```
frame.sort_index()  # this sorts it by the index name... aka the rows
```

Out[5]:

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

In [6]:
```
frame.sort_index(axis = 1) # this sorts it COLUMN wise
```

Out[6]:

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

In [7]:
```
# You can also sort data in a descending order
frame.sort_index(axis = 1, ascending = False)
```

Out[7]:

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

In [8]:
```
# Can also sort Series by its values with this method
obj = pd.Series([4, 7, -3, 2])
obj.sort_values()
```

Out[8]:
```
2   -3
3    2
0    4
1    7
dtype: int64
```

In [10]:
```
# By default, any missing values in a Series are sorted to the end
obj = pd.Series([4, np.nan, 7, np.nan, -3,2])
obj.sort_values()
```

Out[10]:
```
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

In [11]:
```
# For DataFrames, can sortthe data in one or more cols as the sort keys... see below
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame
```

Out[11]:

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

In [12]:
```
frame.sort_values(by = 'b')
```

Out[12]:

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

In [13]:
```
# To sort by multiple cols also works
frame.sort_values(by = ['a', 'b'])
```

Out[13]:

	b	a
2	-3	0
0	4	0
3	2	1
1	7	1

WTF is does it mean to rank? and why do it??? p.155

```
"Ranking" assigns ranks from one through the number of valid data points in an arra
y.
 By default, 'rank' breaks ties by assigning each group the mean rank...
```

Seems like it is a stats thing: https://en.wikipedia.org/wiki/Rank_correlation (https://en.wikipedia.org/wiki/Rank_correlation)

In [14]:
```
obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
obj.rank()
```

Out[14]:
```
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

In [15]:
```
# Ranks can also be assigned accorind to the order in which they're observed i
n the data
obj.rank(method = 'first')
```

Out[15]:
```
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

Axis Indexes with Duplicate Labels

In [2]:
```
# Ex. of a small Series with Duplicate Indicies
obj = pd.Series(range(5), index =['a', 'a', 'b', 'b', 'c'])
obj
```

Out[2]:
```
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

In [3]:
```
# is_unique can tell you whether its labels are unique or not
obj.index.is_unique
```

Out[3]:
```
False
```

In [5]:
```
# Indexing a label with multiple entries returns a Series, while single entrie
s return a scalar value
obj['a']
# vs
obj['c']
```

Out[5]:
```
4
```

In [6]:
```
# the same logic as above applies to indexing rows in a Data Frame
df = pd.DataFrame(np.random.randn(4, 3), index = ['a', 'a', 'b', 'b'])
df
```

Out[6]:

	0	1	2
a	0.636808	-1.346675	-0.341887
a	-0.741066	-0.573600	1.147929
b	-0.168931	-1.395656	0.913830
b	0.942742	-0.034146	-0.490275

In [7]:
```
df.loc['b']  # .loc is a way of indexing. This says "give me all the values in index b in the dataframe df"
```

Out[7]:

	0	1	2
b	-0.168931	-1.395656	0.913830
b	0.942742	-0.034146	-0.490275

5.3 Summarising and Computing Descriptive Statistics

Also called 'reductions' or 'summary statistics', methods that extract a single value.

In [43]:
```
# First consider this df as part of the example
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
                  [np.nan, np.nan], [0.75, -1.3]],
                 index = ['a', 'b', 'c', 'd'],
                 columns = ['one', 'two'])
df
```

Out[43]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

In [9]:
```
# Using 'sum' returns a Series containing column sums
df.sum()
```

Out[9]:
```
one    9.25
two   -5.80
dtype: float64
```

In [10]:
```
# If you want to some ACROSS the columns, aka ROWS, using axis='columns', axis = 1
df.sum(axis = 'columns')
```

Out[10]:
```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

In [12]:
```
# NA calues are excluded in the sum, unless the entire slice is NA
#But this can defaulted with the skipna option - which returns NA as the total
result if the row/column has just 1 NA value

df.mean(axis = 'columns', skipna=False)
```

Out[12]:
```
a       NaN
b     1.300
c       NaN
d    -0.275
dtype: float64
```

In [14]:
```
# idxmin/idxmax, return indirect stats like the index value where the min and m
ax calues are in the df
df.idxmax()
```

Out[14]:
```
one     b
two     d
dtype: object
```

In [44]:
```
df.describe?  # to see more info on what you can do there/ use with this
```

In [15]:
```
# Other methods are *accumulations*
df.cumsum()  # adds each row by row
```

Out[15]:

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

In [16]:
```
# Another type of methos is neither a reduction or accumulations. 'describe' i
s 1 example, produces multiple summary stats in 1 shot
df.describe()
```

Out[16]:

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

In [18]:
```
# On non-numeric data, describe produces alternative summary stats
obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
obj

obj.describe()
```

Out[18]:
```
count     16
unique     3
top        a
freq       8
dtype: object
```

See page 160, Table 5-8 for all the possible Descriptive & Summary stats

Correlation and Covariance

```
These are computed from pairs of arguments.
Here, will use examples of DFs with stock prices and volumes obtained
from Yahoo! Finance, using the add-on pandas-datareader package.
```

In [2]:
```
# Put this into Anaconda conda install pandas-datareader
```

In [45]:
```
# Download le data
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
           for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}

price = pd.DataFrame({ticker: data['Adj Close']
                     for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
                      for ticker, data in all_data.items()})
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-45-60975ad68ee5> in <module>
      1 # Download le data
----> 2 import pandas_datareader.data as web
      3 all_data = {ticker: web.get_data_yahoo(ticker)
      4            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}
      5

ModuleNotFoundError: No module named 'pandas_datareader'
```

In [5]:
```
# Compute percentage changes of the prices, a time series operation,
# More on time series operations in Ch 11

returns = price.pct_change()
returns.tail()

# .tails() returns the last 'n' rows, where n = 5 by default
```

Out[5]:

Date	AAPL	IBM	MSFT	GOOG
2020-12-09	-0.020904	0.008591	-0.019490	-0.018927
2020-12-10	0.011989	-0.014433	-0.006043	-0.004932
2020-12-11	-0.006735	-0.005522	0.013015	0.003628
2020-12-14	-0.005147	-0.005955	0.004408	-0.012184
2020-12-15	0.041961	0.017405	-0.004597	0.003339

In [7]:
```
# corr method computes the correlation of the overlapping, non NAN values, cov
computes the covariance
returns['MSFT'].corr(returns['IBM'])  # The correlation between MSFT and IBM
```

Out[7]: 0.5568744474769819

In [8]:
```
returns['MSFT'].cov(returns['IBM'])
```

Out[8]: 0.00015871418423275921

In [9]:
```
# MSFT for eg. is a valid Python attribute, so u can also select these columns
using other syntax!
returns.MSFT.corr(returns.IBM)
```

Out[9]: 0.5568744474769819

NOTE THAT, all of the above are SERIES!

DataFrame corr and cov methods return a full corr/cov matrix as a DF, respectively.

In [11]:
```
returns.corr()   # remember that the data was already loaded previously!
```

Out[11]:

	AAPL	IBM	MSFT	GOOG
AAPL	1.000000	0.471858	0.715408	0.655802
IBM	0.471858	1.000000	0.556874	0.519063
MSFT	0.715408	0.556874	1.000000	0.779649
GOOG	0.655802	0.519063	0.779649	1.000000

In [12]:
```
returns.cov()  # this and above output a correlation matrix
```

Out[12]:

	AAPL	IBM	MSFT	GOOG
AAPL	0.000361	0.000146	0.000238	0.000207
IBM	0.000146	0.000265	0.000159	0.000141
MSFT	0.000238	0.000159	0.000306	0.000227
GOOG	0.000207	0.000141	0.000227	0.000277

In [14]:
```
# With corr/cov in DFs, can cimpute pair-wise correlations between a DF's cols
and rows
# with another Series or DF. Passing a series returns a Series with correlatio
n values computer for each column

returns.corrwith(returns.IBM)  # compare all the data Series (AAPL, MSFT, etc)
with IBM!


''' This could be useful for Qs like:
Which of these companies have a higher correlation to the IBM stocks?
'''
```

Out[14]:
```
AAPL    0.471858
IBM     1.000000
MSFT    0.556874
GOOG    0.519063
dtype: float64
```

In [15]:
```
# Passing a DataFrame computes the correlations of matching col names.
# Here it is done to compute correlations of percent changes with volume

returns.corrwith(volume)

# Passing axis = 'columns' does things row by row instead. In all cases, the d
ata points are aligned before the correlation is comptuer
```

Out[15]:
```
AAPL   -0.096660
IBM    -0.077429
MSFT   -0.098960
GOOG   -0.165158
dtype: float64
```

Unique Values, Value Counts, and Membership

Another class of related methods extracts info about values contained in a 1D Serie
s. Consider the following example:

In [19]:
```
obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
obj

# Unique is one example, which gives and array of the unique values in a Serie
s
uniques = obj.unique()
uniques

# these are not in a special order by default, but u can with (unique.sort())
```

Out[19]:
```
array(['c', 'a', 'd', 'b'], dtype=object)
```

In [20]:
```
# There is also 'value_counts' computes a Series containing value frequencies
obj.value_counts()
```

Out[20]:
```
a    3
c    3
b    2
d    1
dtype: int64
```

In [22]:
```
# The Series above is sorted by value in descending order, as a convenience.
# Value counts is also available at a top/level pandas method
pd.value_counts(obj.values, sort = True)
```

Out[22]:
```
a    3
c    3
b    2
d    1
dtype: int64
```

In [24]:
```
# 'isin' can be used as well,
mask = obj.isin(['b', 'c'])
mask
```

Out[24]:
```
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

In [25]:
```
obj[mask]  # All the cases of TRUE
```

Out[25]:
```
0    c
5    b
6    b
7    c
8    c
dtype: object
```

In [28]:
```
# Related, is Index.get_indexer, which gives you an index array from an array
 of possible
# non-disctincs values into another array of distinct values:
to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
unique_vals = pd.Series(['c', 'b', 'a'])

pd.Index(unique_vals).get_indexer(to_match)
```

Out[28]:
```
array([0, 2, 1, 1, 0, 2], dtype=int64)
```

In [29]:
```
# E.g. if u want to compute a histogram on multiple related columnes in a DF
data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
                     'Qu2': [2, 3, 1, 2, 3],
                     'Qu3': [1, 5, 2, 4, 4]})
data
```

Out[29]:

	Qu1	Qu2	Qu3
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

WTF is this??? pg 164

In [31]:
```
result = data.apply(pd.value_counts).fillna(0)
result
```

Out[31]:

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	0.0	2.0	1.0
3	2.0	2.0	0.0
4	2.0	0.0	2.0
5	0.0	0.0	1.0

In [ ]:

Title