Ch. 7 Data Cleaning and preparation
pg. 191 - 251

In [1]:
```
import pandas as pd
import numpy as np
```

In general:

https://realpython.com/python-map-function/
A helpful link for this entire chapter, that explains what Mapping, Filtering and Reducing are.

7.1 Handling Missing Data

The standard in pandas is to use NaN for floating pt values.

In [7]:
```
string_data = pd.Series(['aardvark', 'artichoke',  np.nan, 'avocado'])
string_data
```

Out[7]:
```
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object
```

In [8]:
```
string_data.isnull()
```

Out[8]:
```
0    False
1    False
2     True
3    False
dtype: bool
```

```
pandas use of NA/not available is adopted from R.
There is also the built-in Python 'None' value which is also treated like an NA obj
ect array.
```

In [10]:
```
string_data[0] = None  # We are replacing the FIRST value in string_data with
 an NaN equivalent.
string_data.isnull() # We ask how many values are NaN now.
```

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

See Tabel 7-1 on pg 192 for a table for some other NA methods. E.g. - fillna, fill the missing data with some value.

Filtering out missing data

Can do this by using pandas.isnull and boolean indexing, OR by using 'dropna'

In [11]:
```
# FOR A Series:

from numpy import nan as NA

data = pd.Series([1, NA, 3.5, NA, 7])  #Create the series
data
```

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

In [12]:
```
# Now drop all NA values from the series
data.dropna()
```

Out[12]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

In [13]:
```
# You can do the above .dropna() OR you can do:
data[data.notnull()]
```

Out[13]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

FOR DATAFRAMES: You may want to drop rows or columns. Using .dropna by defult drops any row containing a missing value.

In [15]:
```
data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
                     [NA, NA, NA], [NA, 6.5, 3.]])
data
```

Out[15]:

	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 [17]:
```
cleaned = data.dropna() # deleting all ROWS with NA values
cleaned
```

Out[17]:

	0	1	2
0	1.0	6.5	3.0

However, using how='all' will only drop rows where ALL its calues are NA

In [18]:
```
data.dropna(how='all') # So only row 2 was dropped since it had ALL NaN value
s.
```

Out[18]:

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

In [19]:
```
# To drop columns the same way as above, use axis = 1
data[4] = NA  # This says, add a column '4', and make all values NA
data
```

Out[19]:

	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 [21]:
```
# Next, drop columnes there ALL values are NaN
data.dropna(axis = 1, how = 'all')

# MUST put in the axis =1, otherwise only rows with all NAs will be dropped, n
ot columns.
```

Out[21]:

	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

```
It is also common to filter rows of a DataFrame from time series data. Suppose you
 want to keep only rows containing a certain no. of observation. You can indicate t
his by use the *thresh* argument.
```

In [26]:
```
df = pd.DataFrame(np.random.rand(7, 3)) # Create the dataframe.
df.iloc[:4, 1] = NA  # iloc is an index selector, saying replace up the first
 4 rows, in column '1' with NaN
df.iloc[:2, 2] = NA # replace all values up to the 3rd row of the column '2' w
ith NA
df
```

Out[26]:

	0	1	2
0	0.325750	NaN	NaN
1	0.547567	NaN	NaN
2	0.125879	NaN	0.413911
3	0.397435	NaN	0.525204
4	0.374344	0.207910	0.446015
5	0.834835	0.098177	0.026528
6	0.261710	0.499677	0.554304

In [28]:
```
df.dropna() # Get rid of anything row/column with a NA value.
```

Out[28]:

	0	1	2
4	0.374344	0.207910	0.446015
5	0.834835	0.098177	0.026528
6	0.261710	0.499677	0.554304

In [29]:
```
# Instead of the above, can do the following if tou want keep certain rows?
df.dropna(thresh = 2)

#The above reads, drop all NA values that are in col '2'
# Notice that in turn this elimnated two rows as well, cuz otherwise the dataf
rame would be imbalanced.
```

Out[29]:

	0	1	2
2	0.125879	NaN	0.413911
3	0.397435	NaN	0.525204
4	0.374344	0.207910	0.446015
5	0.834835	0.098177	0.026528
6	0.261710	0.499677	0.554304

Filtering *in* missing data

You may want to fill in the 'holes' in the data. The most common method is fillna, with a constant ro replace the missing values with that given constant.

In [30]:
```
df
```

Out[30]:

	0	1	2
0	0.325750	NaN	NaN
1	0.547567	NaN	NaN
2	0.125879	NaN	0.413911
3	0.397435	NaN	0.525204
4	0.374344	0.207910	0.446015
5	0.834835	0.098177	0.026528
6	0.261710	0.499677	0.554304

In [31]:
```
df.fillna(0) # Fill in all the NAs in df with a 0
```

Out[31]:

	0	1	2
0	0.325750	0.000000	0.000000
1	0.547567	0.000000	0.000000
2	0.125879	0.000000	0.413911
3	0.397435	0.000000	0.525204
4	0.374344	0.207910	0.446015
5	0.834835	0.098177	0.026528
6	0.261710	0.499677	0.554304

```
You can also call fillna with a dict, if you want to have a different fill value fo
r each column!
```

In [32]:
```
df # What it looks like before
```

Out[32]:

	0	1	2
0	0.325750	NaN	NaN
1	0.547567	NaN	NaN
2	0.125879	NaN	0.413911
3	0.397435	NaN	0.525204
4	0.374344	0.207910	0.446015
5	0.834835	0.098177	0.026528
6	0.261710	0.499677	0.554304

In [33]:
```
# What it looks like after
df.fillna({1: 0.5, 2:0}) # For col '1', replace all NA with 0.5, and for col
 '2' replace all NAs with 0
```

Out[33]:

	0	1	2
0	0.325750	0.500000	0.000000
1	0.547567	0.500000	0.000000
2	0.125879	0.500000	0.413911
3	0.397435	0.500000	0.525204
4	0.374344	0.207910	0.446015
5	0.834835	0.098177	0.026528
6	0.261710	0.499677	0.554304

[!] WTF does this mean?? pg. 195

```
fillna returns a _new object_, but you can modify the existing object in place...
 (??)
```

In [39]:
```
_ = df.fillna(0, inplace = True) # It's the same output if False?
df
```

Out[39]:

	0	1	2
0	0.325750	0.000000	0.000000
1	0.547567	0.000000	0.000000
2	0.125879	0.000000	0.413911
3	0.397435	0.000000	0.525204
4	0.374344	0.207910	0.446015
5	0.834835	0.098177	0.026528
6	0.261710	0.499677	0.554304

In [40]:
```
# You can use the same redindexing methods with fillna:
df = pd.DataFrame(np.random.randn(6, 3)) #creating a new data from, 6x3 with r
andom no.
df
```

Out[40]:

	0	1	2
0	-0.639978	-0.799902	-0.070594
1	0.725652	0.243492	0.511909
2	-0.386548	0.867206	0.272159
3	-0.475659	-1.148953	0.139650
4	0.110501	0.692779	0.253224
5	1.260373	2.251253	-1.638483

In [41]:
```
df.iloc[2:, 1] = NA   #Replace row 2 onwards, in col '1', with NA
df.iloc[4:, 2] = NA   #Replace rows 4 onwards in col '2' with NA
df
```

Out[41]:

	0	1	2
0	-0.639978	-0.799902	-0.070594
1	0.725652	0.243492	0.511909
2	-0.386548	NaN	0.272159
3	-0.475659	NaN	0.139650
4	0.110501	NaN	NaN
5	1.260373	NaN	NaN

In [42]:
```
# can also do
df.fillna(method = 'ffill')
```

Out[42]:

	0	1	2
0	-0.639978	-0.799902	-0.070594
1	0.725652	0.243492	0.511909
2	-0.386548	0.243492	0.272159
3	-0.475659	0.243492	0.139650
4	0.110501	0.243492	0.139650
5	1.260373	0.243492	0.139650

> **ffill() function is used to fill the missing value in the dataframe. 'ffill' stands for 'forward fill' and will propagate last valid observation forward**

AKA repeats the NA with the previous data value

In [43]:
```
# Only apply the Forward Fill to 2 values, max
df.fillna(method = 'ffill', limit = 2)
```

Out[43]:

	0	1	2
0	-0.639978	-0.799902	-0.070594
1	0.725652	0.243492	0.511909
2	-0.386548	0.243492	0.272159
3	-0.475659	0.243492	0.139650
4	0.110501	NaN	0.139650
5	1.260373	NaN	0.139650

In [44]:
```
# You can also pass the mean or median value with ffill to a Series!
data = pd.Series([1., NA, 3.5, NA, 7])
data # what the series looks like before:
```

Out[44]:
```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

In [45]:
```
#What it looks like after, when we replace all NAs with the median value
data.fillna(data.mean())
```

Out[45]:
```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

See Table 7.2 on pg 197 for more fillna arugments

7.2 Data Transformation

Removing Duplicates

In [46]:
```
# Example of a df with duplicates:
data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
                     'k2': [1, 1, 2, 3, 3, 4, 4]})
data
```

Out[46]:

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

In [47]:
```
# The df method 'duplicated' returns a boolean Series indication if there are any duplicates in the NAME of the row, based on the PREVIOUS one... i.e. so if there are 2 in a row
data.duplicated()
```

Out[47]:
```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

In [48]:
```
# drop_duplicates returns a df where the 'duplicated' array is False:
data.drop_duplicates()


#aka. this method will drop all cases where method 'duplicated' is True.
```

Out[48]:

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

You can also indicate if you only want to drop duplicates from a specific column!

In [49]:
```
# First, start by adding another col to the df, called v1
data['v1'] = range(7)
data
```

Out[49]:

	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

In [50]:
```
# Next, drop any duplicates that are found in k1 only
data.drop_duplicates(['k1'])


# Remember that when deletings values like NAs or duplicates, this will drop t
he ENTIRE row it is found in, so that the data is neat/matches an even matrix/
dataframe form.
```

Out[50]:

	k1	k2	v1
0	one	1	0
1	two	1	1

Transforming data using a function or mapping

You may want to transform values in an array, Series, or column in DF.

In [6]:
```
# First, create some data to work with

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[6]:

	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 [4]:
```
# If you want to add a column that tells the type of animal of each good.
# First, we can write down the _mapping_ or legend for these meatz

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

Mapping!

> **Python's map() is a built-in function that allows you to process and transform all the items in an iterable without using an explicit for loop, a technique commonly known as mapping. map() is useful when you need to apply a transformation function to each item in an iterable and transform them into a new iterable**

https://realpython.com/python-map-function/

```
In other words, the map is like adding a "label" to X variables/categories in that
 label to allow for transformations to ba applied to just that category. See exampl
e below.
```

In [7]:
```
# The 'map' method on a Series accpets a function or dict-like object containi
ng a map...
# BUT before that, the capitalisations needs to match EXACTLY.
# So, we need to transform the data so capitalisations match.

lowercased = data['food'].str.lower()
lowercased
```

Out[7]:
```
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object
```

In [55]:
```
# Now, we are going to APPLY this new lower case key to the data + the map...

data['animal'] = lowercased.map(meat_to_animal) # Create the key/column 'anima
l', which will be the lowercased MAP values of meat_to_animal
data


# See page 199 for a better, color-coated version of the table. There you will
see that animal col looks different since it is a _map_
```

Out[55]:

	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

[!] WTF does this mean?? pg. 300

```
fillna returns a _new object_, but you can modify the existing object in place...
 (??)
```

[https://realpython.com/python-lambda/ (https://realpython.com/python-lambda/)](https://realpython.com/python-lambda/)

In [56]:
```
# Instead of the above, a function could have also been passed instead

data['food'].map(lambda x: meat_to_animal[x.lower()])

#the above is saying...???? BUT WHY x. lower() ?! pg 200
# Answer: for the 'food' col in data df, add the map to x, of the 'meat_to_ani
mal',
```

Out[56]:
```
0        pig
1        pig
2        pig
3        cow
4        cow
5        pig
6        cow
7        pig
8     salmon
Name: food, dtype: object
```

Replacing Values

In [8]:
```
# First, create example data/Series
data = pd.Series([1., -999., 2., -999., -1000., 3.])
data


#In this example -999 could stand for missing data.
```

Out[8]:
```
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

In [9]:
```
# Now, we want to replace -999 values with NA/NaN
data.replace(-999, np.nan)
```

Out[9]:
```
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

In [10]:
```
# IF you want to replace MULTIPLE values at the same time, pass a list!
data.replace([-999, -1000], np.nan)
```

Out[10]:
```
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

In [11]:
```
# to place MULTIPLE values with THEIR OWN unique value:
data.replace([-999, -1000], [np.nan, 0])  # Where np.nan is for -999, and 0 fo
r -1000
```

Out[11]:
```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

In [12]:
```
# The above can ALSO be passed as a dict
data.replace({-999: np.nan, -1000: 0})
```

Out[12]:
```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

Renaming Axis Indexes

Axis labels can also be transformed with a function or mapping. This will produce a new, differently labeled object. You can also modify axes in/place and without creating a new data structure.

In [2]:
```
data = pd.DataFrame(np.arange(12).reshape((3, 4)),
                    index = ['Ohio', 'Colorado', 'New York'],
                    columns = ['one', 'two', 'three', 'four'])
data
```

Out[2]:

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

In [3]:
```
# Axis indexes also have a map method:
transform = lambda x: x[:4].upper()

# First, we are creating the function, which states:
''' The function called transform does this:
takes the input value(x), up to the first 4 observations of the
upper half of the input value (x)'''
```

Out[3]:
```
' The function called transform does this:\ntakes the input value(x), up to t
he first 4 observations of the \nupper half of the input value (x)'
```

In [4]:
```
#Now, apply the lambda function 'transform'
data.index = data.index.map(transform)
```

In [5]:
```
# Now look at the data
data

# Here, the 'transform' function was successful, and replaced the previous axi
s names with shorter ones.
```

Out[5]:

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

In [6]:
```
# Using the method 'rename' will transform the data, but NOT modify the origin
al data
data.rename(index = str.title, columns = str.upper)

# The above just changes the title of the index to uppercase
```

Out[6]:

	ONE	TWO	THREE	FOUR
Ohio	0	1	2	3
Colo	4	5	6	7
New	8	9	10	11

In [7]:
```
# Rename can also be like with a dict-like object, and new axis labels
data.rename(index = {'OHIO': 'INDIANA'},
            columns = {'three': 'peekaboo'})

# The above renames the index Ohio to Indian, and col three to peekaboo
```

Out[7]:

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

```
Rename is used when you DO NOT want to copy the Dataframe manuall and assign values
to is index and cols.
If you want to modify the data with rename officially, you can pass 'inplace = Tru
e'.
```

In [8]:
```
data.rename(index = {'OHIO': 'INDIANA'}, inplace = True)
data
```

Out[8]:

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

Discretization and Binning

Continuous data is usually 'discretized' aka, separated into bins. See example below with groups of people in a study and you want to group them into discrete age buckets:

In [12]:
```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
bins = [18, 25, 35, 60, 100]


# Now divide the ages data indo bins, using pd.cut

cats = pd.cut(ages, bins) # This says put the data along with its bin value.
cats


# What is returned is a special pandas 'Categorical' objects, and can be treat
ed like and array of strings.
```

Out[12]:
```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100],
(35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

See official doc on pd.cut aka, using bins in pandas: [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html)

An example project with pd.cut / bins:

[https://realpython.com/fast-flexible-pandas/ (https://realpython.com/fast-flexible-pandas/)](https://realpython.com/fast-flexible-pandas/)

In [13]:
```
cats.codes  # which bin each data from 'ages' belongs to...the first, second, etc.
```

Out[13]:
```
array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

In [14]:
```
cats.categories # all the different bins that are observed for the cats/ages data
```

Out[14]:
```
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]],
              closed='right',
              dtype='interval[int64]')
```

In [17]:
```
pd.value_counts(cats)  # the bin counts for the results of pd.cuts, like a bin tally
```

Out[17]:
```
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

> **Consistent with mathematical notation for intervals, a parenthesis means that the side is open, while the square bracket means it is closed (inclusive). You can change which side is closed by passing right=False: pg. 203**

In [19]:
```
pd.cut(ages, [18, 26, 36, 61, 100], right = False) # see above for this to make sense, now it is reverse which bin is open, which is closed.
```

Out[19]:
```
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36, 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

You can also pass your OWN bin names, by passing a list or array to the 'labels' option!

In [20]:
```
group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
pd.cut(ages, bins, labels = group_names)
```

Out[20]:
```
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, MiddleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

> **If you pass an integer number of bins to cut instead of explicit bin edges, it will compute equal-length bins based on the minimum and maximum values in the data. Consider the case of some uniformly distributed data chopped into fourths: (see below) pg. 204**

In [22]:
```
data = np.random.rand(20)
data
```

Out[22]:
```
array([0.79823906, 0.75341494, 0.09454416, 0.39304138, 0.10175479,
       0.95731211, 0.57518799, 0.38454512, 0.5360092 , 0.98710699,
       0.43339816, 0.70082235, 0.36679895, 0.40754292, 0.35509414,
       0.64113761, 0.75310718, 0.74036004, 0.81134476, 0.53548824])
```

In [24]:
```
pd.cut(data, 4, precision = 2) # precision limits the decimal place to 2
```

Out[24]:
```
[(0.76, 0.99], (0.54, 0.76], (0.094, 0.32], (0.32, 0.54], (0.094, 0.32], ...,
(0.54, 0.76], (0.54, 0.76], (0.54, 0.76], (0.76, 0.99], (0.32, 0.54]]
Length: 20
Categories (4, interval[float64]): [(0.094, 0.32] < (0.32, 0.54] < (0.54, 0.7
6] < (0.76, 0.99]]
```

[?] Does this mean that you shuold NOT use interger number of bins?

Using qcut

There is also th function 'qcut' which will bing the data based on sample quantiles. You may want to use this instead of 'cut' because 'cut' may not give you an even number of data points in each bin.

> **Since qcut uses sample quantiles instead, by definition you will obtain roughly equal-size bins: (see below)**

In [25]:
```
data = np.random.randn(1000) # Normally distributed
cats = pd.qcut(data, 4) # Cut into 4 quartiles, approx. evenly.

cats
```

Out[25]:
```
[(0.0267, 0.695], (-0.629, 0.0267], (-0.629, 0.0267], (-3.046, -0.629], (0.69
5, 2.716], ..., (-0.629, 0.0267], (-3.046, -0.629], (-0.629, 0.0267], (-0.62
9, 0.0267], (-3.046, -0.629]]
Length: 1000
Categories (4, interval[float64]): [(-3.046, -0.629] < (-0.629, 0.0267] < (0.
0267, 0.695] < (0.695, 2.716]]
```

In [26]:
```
pd.value_counts(cats) # Shows you how many data points are in each 'bin'
```

Out[26]:
```
(0.695, 2.716]      250
(0.0267, 0.695]     250
(-0.629, 0.0267]    250
(-3.046, -0.629]    250
dtype: int64
```

In [29]:
```
# You can also pass your OWN quantiles with qcut, using no. between 0 and 1, i
nclusive.
dog = pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
dog
```

Out[29]:
```
[(0.0267, 1.242], (-1.3, 0.0267], (-1.3, 0.0267], (-1.3, 0.0267], (0.0267, 1.
242], ..., (-1.3, 0.0267], (-3.046, -1.3], (-1.3, 0.0267], (-1.3, 0.0267], (-
1.3, 0.0267]]
Length: 1000
Categories (4, interval[float64]): [(-3.046, -1.3] < (-1.3, 0.0267] < (0.026
7, 1.242] < (1.242, 2.716]]
```

In [30]:
```
pd.value_counts(dog) # this is how it looks...
```

Out[30]:
```
(0.0267, 1.242]    400
(-1.3, 0.0267]     400
(1.242, 2.716]     100
(-3.046, -1.3]     100
dtype: int64
```

[?] When and why would you ever want to set your own quantiles? I dont get the example either... (see above chunks).

Detecting and Filtering Outliers

Maily just applying array operations. Below will be an example with normally distributed data.

In [31]:
```
data = pd.DataFrame(np.random.randn(1000, 4))
data.describe()
```

Out[31]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.045703	-0.006905	-0.035234	-0.031098
std	1.003481	0.999982	0.975964	1.001750
min	-3.004345	-3.727977	-3.728292	-3.228248
25%	-0.615337	-0.684660	-0.667265	-0.687612
50%	0.058913	0.044233	-0.006272	-0.035844
75%	0.672576	0.668767	0.597902	0.659980
max	3.164499	2.834683	2.631226	2.802174

In [32]:
```
#Suppose you wanted to find values in one of the columns exceeding 3 in absolu
te value:
col = data[2] # first isolate/ select only the 3rd column of the data

col[np.abs(col) >3]
```

Out[32]:
```
599   -3.728292
Name: 2, dtype: float64
```

In [33]:
```
# To select the rows that have a value of 3 or -3 you can use the 'any' metho
d...
data[(np.abs(data) >3).any(1)]

# Notes, SELECT A ROW, THE WHOLE DAMN ROW that has a +-3 value
```

Out[33]:

	0	1	2	3
143	-3.004345	-0.899828	0.240154	0.192567
400	3.164499	0.813197	-0.078097	-0.080511
426	3.002462	-1.296210	-0.842690	1.191065
503	-1.410977	-3.727977	0.805915	0.865970
599	-1.009860	-0.396570	-3.728292	-1.332789
626	2.153357	-1.127845	0.182386	-3.228248

In [37]:

```
# 'Code to cap values outside the interval -3 to 3'
data[np.abs(data) > 3] = np.sign(data) * 3  # we do this to get just a clean -
3 or 3
data.describe()

## ??? But what is the point of doing this
```

Out[37]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.045541	-0.006177	-0.034506	-0.030870
std	1.002962	0.997532	0.973474	1.001047
min	-3.000000	-3.000000	-3.000000	-3.000000
25%	-0.615337	-0.684660	-0.667265	-0.687612
50%	0.058913	0.044233	-0.006272	-0.035844
75%	0.672576	0.668767	0.597902	0.659980
max	3.000000	2.834683	2.631226	2.802174

In [38]:

```
# np.sign(data) produced 1 and -1 values on whether the values in the data are
pos or negative
np.sign(data).head()
```

Out[38]:

	0	1	2	3
0	-1.0	1.0	-1.0	-1.0
1	-1.0	1.0	1.0	-1.0
2	1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0
4	-1.0	1.0	-1.0	-1.0

Permutation and Random Sampling

Permuting (randomly reordering) can be down with np.random.permutation. Calling permutation with the length of the axis you want to permute creates an array of intergers indicating the new reordering: (see below).

Also for offic. doc.
[https://numpy.org/doc/stable/reference/random/generated/numpy.random.permutation.html (https://numpy.org/doc/stable/reference/random/generated/numpy.random.permutation.html)](https://numpy.org/doc/stable/reference/random/generated/numpy.random.permutation.html)

In [39]:
```
df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))
df

#Creating the data and taking a preliminary look at the data
```

Out[39]:

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19

In [40]:
```
sampler = np.random.permutation(5)  # Randomly rearrange an array of 5 items
sampler  # This outputs says re-arrange items 0, 1, 2, 3, 4 like so:
```

Out[40]: array([2, 0, 3, 4, 1])

In [41]:
```
# The array can then be used in _iloc_ based indexing or the equivalent _take_
function
df
```

Out[41]:

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19

In [42]:
```
df.take(sampler) # This randomly permutated the INDEXES aka the ENTIRE ROWS of
the data
```

Out[42]:

	0	1	2	3
2	8	9	10	11
0	0	1	2	3
3	12	13	14	15
4	16	17	18	19
1	4	5	6	7

[?] So, does that mean sampler MUST match the no. of rows/indexes in the dataframe to work?

In [43]:
```
# To select a random subset w.o. replacement, you can use _sample_ method on a Series or DF
df.sample(n = 3)  # Give me 3 random rows/indexes
```

Out[43]:

	0	1	2	3
1	4	5	6	7
0	0	1	2	3
2	8	9	10	11

In [45]:
```
# To generate a sample WITH replacement, pass replace = True to sample.
choices = pd.Series([5, 7, -1, 6, 4])
draws = choices.sample(n = 10, replace = True)
draws
```

Out[45]:
```
0    5
4    4
3    6
2   -1
3    6
1    7
2   -1
2   -1
0    5
0    5
dtype: int64
```

Computing Indicator/Dummy Variables

Pandas has a get_dummies function, and converts categorical variables into dummy or indicator variables.

In [47]:
```
df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                   'data1': range(6)})

df
```

Out[47]:

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

In [48]:
```
pd.get_dummies(df['key']) # Because there are 3 diff values, a, b, c each have their own key...
# Where col a means give a 0 to all values that are NOT a, and col b says give value 0 to all those thare NOT b, etc.
```

Out[48]:

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

In [50]:
```
# To make it easier to read, can add a prefix to name the different keys better.
dummies = pd.get_dummies(df['key'], prefix = 'key')
df_with_dummy = df[['data1']].join(dummies)

df_with_dummy
```

Out[50]:

	data1	key_a	key_b	key_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0
5	5	0	1	0

If a row in a DF belongs to multipl categories, it is more complcated. See example below looking at the MovieLens 1M dataset, which is investigated in more detail in Ch. 14

In [2]:
```
mnames = ['movie_id', 'title', 'genres']

movies = pd.read_table('C:\\Users\\Kitty\\Desktop\\learnpy\\movies.dat', sep =
'::',
                        header = None, names = mnames)  # Had to put r in fron
t, or \\

movies[:10] # give me the first 10 rows of data
```

```
<ipython-input-2-0c96f39157f3>:3: ParserWarning: Falling back to the 'python'
engine because the 'c' engine does not support regex separators (separators >
1 char and different from '\s+' are interpreted as regex); you can avoid this
warning by specifying engine='python'.
  movies = pd.read_table('C:\\Users\\Kitty\\Desktop\\learnpy\\movies.dat', se
p = '::',
```

Out[2]:

	movie_id	title	genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy
5	6	Heat (1995)	Action|Crime|Thriller
6	7	Sabrina (1995)	Comedy|Romance
7	8	Tom and Huck (1995)	Adventure|Children's
8	9	Sudden Death (1995)	Action
9	10	GoldenEye (1995)	Action|Adventure|Thriller

In [16]:
```
# To add indicator variables, need to wrangle more.
#First, extract the list of unique genres in the datset

all_genres = []

for x in movies.genres:
    all_genres.extend(x.split('|'))

# Above- creating a for loop to go through all genres and split them up by the
```

In [17]:
```
genres = pd.unique(all_genres) #print all the unique cases of genres, to see h
ow many different ones there are

genres
```

Out[17]:
```
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
       'Western'], dtype=object)
```

One way to construct the indicator DF is to start with a DF of all zeros (see code below)

[?] Is there another way to do this?

In [19]:
```
zero_matrix = np.zeros((len(movies), len(genres))) # make a 0 matrix that has
 same length/dimensions as the following variables in the DF
dummies = pd.DataFrame(zero_matrix, columns = genres)

dummies
```

Out[19]:

	Animation	Children's	Comedy	Adventure	Fantasy	Romance	Drama	Action	Crime	Th
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
...	...	...	...	...	...	...	...	...	...	
3878	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3879	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3880	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3881	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3882	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

3883 rows × 18 columns

Now interact through each movie and set in each row of dummies to 1... by using dummies.columnes to calculate the column indices for each genre.

In [24]:
```
gen = movies.genres[0] # the genre of the movies in the first row
gen

# This is kind of like setting the stage/starting pt for creating the dummie v
ariables
```

Out[24]: `"Animation|Children's|Comedy"`

In [25]:
```
gen.split('|') #split the gen variable
```

Out[25]: `['Animation', "Children's", 'Comedy']`

[?] WTF does this do, pg 210. pls explain

In [4]:
```
dummies.columns.get_indexer(gen.split('|'))

# WTF HAPPENED BRUH
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-ae17c8654c4a> in <module>
----> 1 dummies.columns.get_indexer(gen.split('|'))
      2 
      3 # WTF HAPPENED BRUH

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

[?] WTF happened here pg 209. pls explain

In [3]:
```
# Next, can use .iloc to set values based on these indicies that were jsut cre
ated
for i, gen in enumerate(movies.genres):
    indicies = dummies.columns.get_indexer(gen.split('|'))
    dummies.iloc[i, indicies] = 1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-572db78995b7> in <module>
      1 # Next, can use .iloc to set values based on these indicies that were
jsut created
      2 for i, gen in enumerate(movies.genres):
----> 3     indicies = dummies.columns.get_indexer(gen.split('|'))
      4     dummies.iloc[i, indicies] = 1

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

In [31]:
```
# Now combine with movies...
movies_windic = movies.join(dummies.add_prefix('Genre_'))
movies_windic.iloc[0]
```

Out[31]:
```
movie_id                                       1
title                           Toy Story (1995)
genres               Animation|Children's|Comedy
Genre_Animation                                1
Genre_Children's                               1
Genre_Comedy                                   1
Genre_Adventure                                0
Genre_Fantasy                                  0
Genre_Romance                                  0
Genre_Drama                                    0
Genre_Action                                   0
Genre_Crime                                    0
Genre_Thriller                                 0
Genre_Horror                                   0
Genre_Sci-Fi                                   0
Genre_Documentary                              0
Genre_War                                      0
Genre_Musical                                  0
Genre_Mystery                                  0
Genre_Film-Noir                                0
Genre_Western                                  0
Name: 0, dtype: object
```

Another useful method for stats applications is to combine get_dummies with a discretization function like 'cut':

In [32]:
```
np.random.seed(12345)
values = np.random.rand(10)
values
```

Out[32]:
```
array([0.92961609, 0.31637555, 0.18391881, 0.20456028, 0.56772503,
       0.5955447 , 0.96451452, 0.6531771 , 0.74890664, 0.65356987])
```

In [34]:
```
bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
pd.get_dummies(pd.cut(values, bins))

#Set the random seed with numpy.random.seed to make the example 'deterministi
c'
#later, the book will explore pd.get_dummies
```

Out[34]:

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]
0	0	0	0	0	1
1	0	1	0	0	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	0
5	0	0	1	0	0
6	0	0	0	0	1
7	0	0	0	1	0
8	0	0	0	1	0
9	0	0	0	1	0

7.3 String Manipulation

String Object Methods

In [5]:
```
# E.g., a common-separate string can be broken into pieces with 'split'
val = 'a,b,  guido'
val.split(',')
```

Out[5]: ['a', 'b', ' guido']

In [6]:
```
# 'split' is often combined with 'strip' to strim white space, including line
 breaks
pieces = [x.strip() for x in val.split(',')]
pieces
```

Out[6]: ['a', 'b', 'guido']

In [7]:
```
# These substrings could be added together with 2 colon delimiter using additi
on
# WHUT

first, second, third = pieces
first + '::' + second + '::' + third

# ... Interesting...but why?  -See below, this is not practical or very 'Pytho
n' like.
```

Out[7]: 'a::b::guido'

In [8]:
```
# However, the above is not practical...
# Faster to pass a list or tuple to the 'join' method on the string '::'
'::'.join(pieces)
```

Out[8]: 'a::b::guido'

In [9]:
```
# Other methods deal with locating substrings
'guido' in val
```

Out[9]: True

In [41]:
```
val.index(',')
```

Out[41]: 1

In [10]:
```
val.find(':') # Returns -1 if the string isn't found
```

Out[10]: -1

In [11]:
```
val.index(':') # Where as this just returns an error, compared to "val find"
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-11-2c016e7367ac> in <module>
----> 1 val.index(':')

ValueError: substring not found
```

In [12]:
```
# 'Count' returns the no. of occurences of a particular substring
val.count('')
```

Out[12]: 12

In [13]:
```
# 'Replace' will subsisitute a value with the one u tell it to.
# Commonly used to delete patterns too, by just passing an empty string

val.replace(',', '::') # For every ',' replace it with '::'
```

Out[13]: 'a::b:: guido'

In [14]:
```
val.replace(',', '') # For every ',' DELETE by replacing it with nothing
```

Out[14]:
```
'ab  guido'
```

See table 7.3 on pg 213 for a more Python built-in string methods

Regular Expressions

A way to search or match string patterns in a text. A single expression is called a 'regex', and is a string formed according to the the regular expression language.

Python has built in 're', and its functions fall into 3 categories: pattern matching, subsitution, and splitting.

In [15]:
```
# Ex. We want to split a string with a variable number of whitespace character
s,
# in regext, whitespace characters a '\s+'
# Whitespace includes: tabs, spaces, and newlines.

import re
```

In [16]:
```
text = "foo         bar\t baz   \tqux"
text
```

Out[16]:
```
'foo         bar\t baz   \tqux'
```

In [18]:
```
re.split('\s+', text)  # Split up 'text', by the whitespace
```

Out[18]:
```
['foo', 'bar', 'baz', 'qux']
```

In [19]:
```
# When re.split is called, the regex is first *compiled*, and then split is ca
lled.
# You can compile a regex urself, to make a reusable regex object.

regex = re.compile('\s+')
regex.split(text)

# So basically, make a function of the compiling first (?)
## ?? What is the point? Faster? Asnwer: YES
```

Out[19]:
```
['foo', 'bar', 'baz', 'qux']
```

In [20]:
```
# If you want to get a list of all patterns matching the regex, use findall.

regex.findall(text)  # Show me a list of all the whitespace in 'text'
```

Out[20]:
```
['         ', '\t ', '   \t']
```

'match' and 'search' are similar to 'findall'

'findall' returns ALL matches in a string.
'search' returns only the FIRST match.
'match' ONLY matches at the beginning of the string.

See below for an example with a block of text and a regex capable of identifying most email addresses.

In [21]:
```
# JUST AN EXAMPLE GIVEN IN THE BOOK, pg 214
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}' #give me only email addres
ses, anything WITH an @ symbol

# re.IGNORECASE makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

In [22]:
```
regex.findall(text)

#So this is saying, use the previously define 'regex' that are an email addres
s, and not jsut a name
```

Out[22]:
```
['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

In [23]:
```
# 'search' returns a special match object for the 1st email only.... huh
m = regex.search(text)
m

# And tells us where it is... >_>  ????
```

Out[23]:
```
<re.Match object; span=(5, 20), match='dave@google.com'>
```

In [24]:
```
text[m.start():m.end()] # of the value of the search result (aka 'm') pls give
me the full beginning and end of that string
```

Out[24]:
```
'dave@google.com'
```

In [25]:
```
#regex.match returns None, as it onyl will match if the pattern occurs at the
 START of the string
print(regex.match(text))
```

```
None
```

In [26]:
```
# Similarly, 'sub' will return a new string with occurrences of the pattern re
place by the string
print(regex.sub('REDACTED', text))

# AKA use sub to replace whatever meets the previously defined 'regex' with th
e indicated value/in this case REDACTED
```

```
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

> **Suppose you wanted to find email addresses and simultaneously segment each address into its three components: username, domain name, and domain suffix. To do this, put parentheses around the parts of the pattern to segment:**
> **pg. 215**

In [27]:
```
# Step 1 - COPIED FROM BOOK
pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

# This is the defining first what we want in the example text above:
# To segment the emails in 3 component.
```

In [28]:
```
# Step 2
regex = re.compile(pattern, flags = re.IGNORECASE)

# We are compiling it now/defining the regex, and saying that upper or lower c
ase dont matter
```

In [29]:
```
# Step 3 -
# A 'match' object produced by this regex returns a tuple of the pattern compo
nents with method 'groups'
m = regex.match('wesm@bright.net') # instead of our text list, we are applying
above steps to this new email address.
m.groups()

# We we can put it all together, to get what we originally wanted in the examp
le
```

Out[29]:
```
('wesm', 'bright', 'net')
```

In [30]:
```
# 'findall' returns of list of tuples when the pattern has groups.
regex.findall(text)  # now apply the above to our previous "text" with all oth
er emails
```

Out[30]:
```
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

In [31]:
```
# 'sub' also has access to groups in each match using special symbols
# like \1 (first matches group) or \2 (second matched group, etc)

print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))

# the above is saying, using the regex, add these labels to each of the matche
d groups.
# E.g., call the 1st group 'Username', etc.
```

```
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

See table 7.3 pg. 216 for table of more regex methods

Vectorized String Functions in pandas

(e.g., for when having to clean data, and some strings have missing data)

In [33]:
```
data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
        'Rob': 'rob@gmail.com', 'Wes': np.nan}
```

In [34]:
```
data = pd.Series(data)
data
```

Out[34]:
```
Dave     dave@google.com
Steve    steve@gmail.com
Rob        rob@gmail.com
Wes                  NaN
dtype: object
```

In [35]:
```
data.isnull() # Check if/where is there missing data
```

Out[35]:
```
Dave     False
Steve    False
Rob      False
Wes       True
dtype: bool
```

You can apply string and regex methods can be applied (by passing a lambda or other function) to each value using 'data.map' but it will fail on the NA / null values.

To deal with this, Series has array-oriented methods for string operations that <u>skip NA values</u>! These are accessed via Serie's 'str' attribute.
E.g., we can check whether each email address has 'gmail' in it with 'str.contains' (see below)

In [36]:
```
data.str.contains('gmail')
```

Out[36]:
```
Dave     False
Steve     True
Rob       True
Wes        NaN
dtype: object
```

In [37]:
```
# Regex can be use too, along with any other 're' optionsl ike IGNORECASE
pattern
```

Out[37]:
```
'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'
```

In [38]:
```
data.str.findall(pattern, flags = re.IGNORECASE)
```

Out[38]:
```
Dave     [(dave, google, com)]
Steve    [(steve, gmail, com)]
Rob        [(rob, gmail, com)]
Wes                        NaN
dtype: object
```

In [50]:
```
# Are a few ways to do *vectorised element retrival*
# Either with 'str.get' or index into the 'str' attribute
matches = data.str.match(pattern, flags = re.IGNORECASE)
matches
```

Out[50]:
```
Dave     True
Steve    True
Rob      True
Wes       NaN
dtype: object
```

In [49]:
```
# To access elements in the embedded lists, can pass an index to either of the
se functions
matches.str.get(1)

## ** I GOT AN ERROR? but y
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-49-43eaae90677b> in <module>
      1 # To access elements in the embedded lists, can pass an index to eith
er of these functions
----> 2 matches.str.get(1)
      3
      4 ## ** I GOT AN ERROR? but y

~\anaconda3\lib\site-packages\pandas\core\generic.py in __getattr__(self, nam
e)
   5268             or name in self._accessors
   5269         ):
-> 5270             return object.__getattribute__(self, name)
   5271         else:
   5272             if self._info_axis._can_hold_identifiers_and_holds_name(n
ame):

~\anaconda3\lib\site-packages\pandas\core\accessor.py in __get__(self, obj, c
ls)
    185             # we're accessing the attribute of the class, i.e., Datas
et.geo
    186             return self._accessor
--> 187         accessor_obj = self._accessor(obj)
    188         # Replace the property with the accessor object. Inspired by:
    189         # http://www.pydanny.com/cached-property.html

~\anaconda3\lib\site-packages\pandas\core\strings.py in __init__(self, data)
   2039
   2040     def __init__(self, data):
-> 2041         self._inferred_dtype = self._validate(data)
   2042         self._is_categorical = is_categorical_dtype(data)
   2043         self._is_string = data.dtype.name == "string"

~\anaconda3\lib\site-packages\pandas\core\strings.py in _validate(data)
   2096
   2097         if inferred_dtype not in allowed_types:
-> 2098             raise AttributeError("Can only use .str accessor with str
ing values!")
   2099         return inferred_dtype
   2100

AttributeError: Can only use .str accessor with string values!
```

In [45]:
```
str(matches)
```

Out[45]:
```
'Dave     True\nSteve    True\nRob      True\nWes       NaN\ndtype: object'
```

In [ ]:
```
matches.str[0]
```

In [47]:
```
data.str[:5]
```

Out[47]:
```
Dave     dave@
Steve    steve
Rob      rob@g
Wes        NaN
dtype: object
```

See table 7.5 for more pandas string methods

7.4 Conclusion

EFFECTIVE DATA PREP CAN SIGNIFICANTLY IMPROVE PRODUCTIVITY, ALLOWING FOR MORE TIME TO BE SPENT ON ANALYSING THE DATA AND LESS TIME GETTING IT READ FOR ANALYSIS.