Data Cleaning and Preparation

roughly 80% of the total time spent on data analysis is used for data manipulation and cleaning
making sure it is in the correct form in order to be analysed. This is usually completed in general purpose programming languages such as python, R, Perl or java. fortunately python and
pandas has a number of built in functions for data manipulation

Handling missing Data

pandas is designed to make missing data less painful. all descriptive stats on pandas objects skips missing data by defualt.

missing data in for numberic data and a pandas object represented by floating-point value NaN to represent missing data. We call it a sentinel value.
when analysing data it is important to consider where the missing data is an what affect or bias's it may make on your final analysis.

In [2]:
```
import pandas as pd
import numpy as np
string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
string_data
```

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

In [3]:
```
string_data.isnull() #if null returns true
```

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

In [4]:
```
string_data[0] = None
string_data
```

Out[4]:
```
0         None
1    artichoke
2          NaN
3      avocado
dtype: object
```

In [5]:
```
string_data.isnull()
```

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

NA handling methods

- dropna -- Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate
- fillna -- Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'
- isnull -- Return boolean values indicating which values are missing/NA
- notnull -- Negation of isnull
  ## Filtering Out Missing data

In [6]:
```
from numpy import nan as NA
data = pd.Series([1, NA, 3.5, NA, 7])
data
```

Out[6]:
```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

In [7]:
```
data.dropna() #gets rid of the NaN values
```

Out[7]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

In [8]:
```
data[data.notnull()] # is the equivalent of above
```

Out[8]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

with DataFrames its a bit more complicated. you may want to drop rows or columns that are NA or only those containing NAs. dropna by default drops any row containing a missing value.

In [9]:
```
data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA], [NA, NA, NA], [NA, 6
cleaned = data.dropna()
data
```

Out[9]:

	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 [10]:
```
cleaned
```

Out[10]:

	0	1	2
0	1.0	6.5	3.0

In [11]:
```
cleaned = data.dropna(how='all') # will only dro the rows that are all
cleaned
```

Out[11]:

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

In [12]:
```
#should you which to drop the columns in the same way you can pass the
cleaned = data.dropna(axis=1)
cleaned
```

Out[12]:

0
1
2
3

Filling In Missing Data

an alternative to filtering out data and possibly losing rows or columns you can use fillna()

In [13]:
```
df = pd.DataFrame(np.random.randn(7,3))
df.iloc[:4, 1]=NA
df
```

Out[13]:

	0	1	2
0	-0.183526	NaN	1.080390
1	0.330354	NaN	0.306555
2	1.583294	NaN	0.786937
3	-0.875215	NaN	-2.076086
4	-0.726774	-0.884924	0.404506
5	-0.490088	0.120185	-0.870638
6	0.011211	0.595754	-0.087095

In [14]:
```
df.dropna()
```

Out[14]:

	0	1	2
4	-0.726774	-0.884924	0.404506
5	-0.490088	0.120185	-0.870638
6	0.011211	0.595754	-0.087095

In [15]:
```
df.fillna(0)
```

Out[15]:

	0	1	2
0	-0.183526	0.000000	1.080390
1	0.330354	0.000000	0.306555
2	1.583294	0.000000	0.786937
3	-0.875215	0.000000	-2.076086
4	-0.726774	-0.884924	0.404506
5	-0.490088	0.120185	-0.870638
6	0.011211	0.595754	-0.087095

if you pass a dict to filna it will fill each of of the key value's present in the dataset will its corresponding value

In [16]:
```
df = pd.DataFrame(np.random.randn(6, 3))
df.iloc[2:, 1] = NA
df.iloc[4:, 2] = NA
df
```

Out[16]:

	0	1	2
0	-0.904534	0.535601	0.706661
1	-1.671020	0.836299	-0.299906
2	-1.097650	NaN	-1.784176
3	-1.301735	NaN	1.090550
4	0.826451	NaN	NaN
5	-0.385091	NaN	NaN

In [17]:
```
df.fillna(method='ffill') #extends the value to the rest of the rows i
```

Out[17]:

	0	1	2
0	-0.904534	0.535601	0.706661
1	-1.671020	0.836299	-0.299906
2	-1.097650	0.836299	-1.784176
3	-1.301735	0.836299	1.090550
4	0.826451	0.836299	1.090550
5	-0.385091	0.836299	1.090550

In [18]:
```
df.fillna(method='ffill', limit=2) ##limits the replication to two rov
```

Out[18]:

	0	1	2
0	-0.904534	0.535601	0.706661
1	-1.671020	0.836299	-0.299906
2	-1.097650	0.836299	-1.784176
3	-1.301735	0.836299	1.090550
4	0.826451	NaN	1.090550
5	-0.385091	NaN	1.090550

references for fillna()

- value -- Scalar value or dict-llike object to use to fill missing values
- method -- interpolation; by default 'ffill' if function called with no arguments
- axis -- Axis to fill on; default axis=0
- inplace -- Modify the calling object without producing a copy
- limit -- For foward and backward filling, maximum number of consecutive periods to fill

Data Transformation

Removing Duplicates

duplicates may be found in a dataframe for a number of reasons. lets create some to find out how to get rid of them

In [19]:
```
data = pd.DataFrame({'k1':['one', 'two'] * 3 + ['two'], 'k2':[1,1,2,3,
data
```

Out[19]:

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

In [20]:
```
data.duplicated() #returns boolean Series wether the data is duplicate
```

Out[20]:
```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

In [21]:
```
data.drop_duplicates() #drops the duplicated rows
```

Out[21]:

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

Tansforming Data Using a Function or Mapping

In [26]:
```
data = pd.DataFrame({'food':['bacon', 'pulled pork', 'bacon', 'corcodi
data
```

Out[26]:

	food	ounces
0	bacon	4
1	pulled pork	3
2	bacon	12
3	corcodile	18

should you have a dict that maps on value to another. e.g. food source to meat

In [27]:
```
meat_to_animal = {'bacon':'pig', 'pulled pork':'pig'}
```

In [28]:
```
data['animal'] = data['food'].map(meat_to_animal)
data
```

Out[28]:

	food	ounces	animal
0	bacon	4	pig
1	pulled pork	3	pig
2	bacon	12	pig
3	corcodile	18	NaN

Replacing Values

In [30]:
```
data = pd.Series([-999, 1, 2, 3, 456, 5, -999])
data
```

Out[30]:
```
0   -999
1      1
2      2
3      3
4    456
5      5
6   -999
dtype: int64
```

In [33]:
```
data.replace(-999, np.nan)
```

Out[33]:
```
0      NaN
1      1.0
2      2.0
3      3.0
4    456.0
5      5.0
6      NaN
dtype: float64
```

In [35]:
```
data.replace([np.nan, 3], [-999, 'replaced'])
```

Out[35]:
```
0        -999
1           1
2           2
3    replaced
4         456
5           5
6        -999
dtype: object
```

Renaming Axis Indexes

In [40]:
```
data = pd.DataFrame(np.arange(12).reshape((3,4)))
data
```

Out[40]:

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

In [41]:
```
data.rename(index={1:'replacement'}, columns={3:'replacement'})
```

Out[41]:

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

Discretization and Binning

continuous data is often seperated into bin for analysis.

In [46]:
```
ages = np.arange(30)
ages
```

Out[46]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 1
5, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
```

In [53]:
```
bins = [5, 15, 25,30]
cats = pd.cut(ages, bins)
cats
```

Out[53]:
```
[NaN, NaN, NaN, NaN, NaN, ..., (15, 25], (25, 30], (25, 30], (25, 30
], (25, 30]]
Length: 30
Categories (3, interval[int64]): [(5, 15] < (15, 25] < (25, 30]]
```

In [54]:
```
cats.codes
```

Out[54]:
```
array([-1, -1, -1, -1, -1, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2], dtype=in
t8)
```

In [55]:
```
pd.value_counts(cats)
```

Out[55]:
```
(15, 25]    10
(5, 15]     10
(25, 30]     4
dtype: int64
```

As with mathematics the parenthesis indicated an open side and the square bracket a closed limit in the range.
you can also pass your own catagory labels.

In [58]:
```
group_names = ['youth', 'young adult', 'middle ages']
pd.cut(ages, bins, labels=group_names)
```

Out[58]:
```
[NaN, NaN, NaN, NaN, NaN, ..., young adult, middle ages, middle ages
, middle ages, middle ages]
Length: 30
Categories (3, object): [youth < young adult < middle ages]
```

pd.cut can place the relevant data into the number of bins you specify then can be evenly distributed.

In [60]:
```
data = np.random.rand(20) #equally idstributed random
pd.cut(data, 4, precision=2) #cuts into 4 bins precision=... limits to
```

Out[60]:
```
[(0.68, 0.9], (0.68, 0.9], (0.026, 0.25], (0.46, 0.68], (0.68, 0.9],
..., (0.25, 0.46], (0.026, 0.25], (0.68, 0.9], (0.026, 0.25], (0.026
, 0.25]]
Length: 20
Categories (4, interval[float64]): [(0.026, 0.25] < (0.25, 0.46] < (
0.46, 0.68] < (0.68, 0.9]]
```

Cutting into Quatiles

In [63]:
```
data = np.random.randn(1000) # Normally distributed
cats = pd.qcut(data, 4)
cats
```

Out[63]:
```
[(-3.3689999999999998, -0.713], (-0.0426, 0.664], (-0.0426, 0.664],
(-0.713, -0.0426], (-0.0426, 0.664], ..., (-0.0426, 0.664], (-0.0426
, 0.664], (-0.713, -0.0426], (0.664, 3.361], (-3.3689999999999998, -
0.713]]
Length: 1000
Categories (4, interval[float64]): [(-3.3689999999999998, -0.713] <
(-0.713, -0.0426] < (-0.0426, 0.664] < (0.664, 3.361]]
```

In [64]:
```
pd.value_counts(cats)
```

Out[64]:
```
(0.664, 3.361]                   250
(-0.0426, 0.664]                 250
(-0.713, -0.0426]                250
(-3.3689999999999998, -0.713]    250
dtype: int64
```

You can pass your own quantiles, numbers between 0 and 1.

In [68]:
```
cats = pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
```

In [70]:
```
pd.value_counts(cats)
```

Out[70]:
```
(-0.0426, 1.211]                 400
(-1.286, -0.0426]                400
(1.211, 3.361]                   100
(-3.3689999999999998, -1.286]    100
dtype: int64
```

Detecting and Filtering Outliers

this process is simply a case of applying array operations.

In [71]:
```
data = pd.DataFrame(np.random.randn(1000, 4))
data.describe()
```

Out[71]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.004946	-0.009557	-0.002804	0.024518
std	0.996536	0.990844	0.961222	0.986038
min	-3.581137	-3.518803	-2.755059	-3.049681
25%	-0.629803	-0.668221	-0.664707	-0.657763
50%	-0.000486	0.005532	-0.007814	0.008222
75%	0.636974	0.647030	0.656196	0.748108
max	3.319428	3.092605	2.711915	3.459830

suppose you wanted to find values in one columns exceeding 3 in absolute value

In [73]:
```
col = data[2]
col
```

Out[73]:
```
0      1.195596
1     -0.207076
2      0.490025
3     -0.025798
4      0.848825
         ...
995    0.176935
996    0.216788
997    0.439255
998    0.843606
999    1.119824
Name: 2, Length: 1000, dtype: float64
```

In [79]:
```
col[np.abs(col) > 2.7]
```

Out[79]:
```
100    2.711915
960   -2.755059
Name: 2, dtype: float64
```

To select all rows having a value exceedin 2.7 or -2.7 you can use the any method on the boolean Data frame

In [83]:
```
data[(np.abs(data) > 2.7).any(1)] # selects all rows containing values
```

Out[83]:

	0	1	2	3
100	1.737397	-0.014729	2.711915	0.836709
194	-3.087513	-1.588594	0.878167	-1.399535
196	-2.994581	-0.194604	0.563069	0.319071
215	-0.357609	1.335791	1.424017	-3.049681
351	-0.408232	3.081563	0.563135	-0.445249
432	3.319428	-0.005683	-1.069489	2.770649
554	-2.784007	1.317804	-0.161120	0.992402
567	0.729090	2.701250	-0.843015	-0.815244
644	-0.418380	-0.448061	1.186963	3.459830
700	-2.735510	0.945955	1.107537	1.050384
727	3.167322	0.533816	2.546234	2.499271
752	-0.456142	3.092605	-0.377769	-0.242404
809	-3.581137	-0.069892	-1.919781	-0.611178
851	0.479112	-3.194128	0.068630	1.580397
960	-0.158464	-1.005514	-2.755059	-0.388204
974	-3.206862	1.300587	-1.657217	0.208138
977	-0.788273	-3.518803	0.740244	0.456317

In [87]:
```
data[np.abs(data) > 3] = np.sign(data) * 2.7
data.describe()
```

Out[87]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.004558	-0.009018	-0.002804	0.024108
std	0.992163	0.987993	0.961222	0.984387
min	-3.000000	-3.000000	-2.755059	-3.000000
25%	-0.629803	-0.668221	-0.664707	-0.657763
50%	-0.000486	0.005532	-0.007814	0.008222
75%	0.636974	0.647030	0.656196	0.748108
max	3.000000	3.000000	2.711915	3.000000

np.sign(data) produces 1 and -1 values based on wether the data is positive or negative. therefore capping the max and min values at 2.7 and -2.7.

In [89]:
```
np.sign(data).head() #
```

Out[89]:

	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

Computing Indicator/Dummy Variables

Another type of transformation for statistical or ML. Turns catagorical data into a "dummy" or "indicator" matrix which could be used to turn parameters on and of in a regression equation for example

In [91]:
```
df = pd.DataFrame({'key':['b', 'b', 'a', 'c', 'a', 'b'], 'data1':range
df
```

Out[91]:

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

In [94]:
```
dummies = pd.get_dummies(df['key'], prefix='key')
df_with_dummy = df[['data1']].join(dummies)
dummies
```

Out[94]:

	key_a	key_b	key_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 [95]:
```
df_with_dummy
```

Out[95]:

	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

String Manipulation

pythons built-in methods for manipulating strings and text processing is particularly useful for data manipulations. Pandas builds apon it by enabling manipulations for whole array's of objects

In [97]:
```
val = 'a, b, guido'
val.split(',')
```

Out[97]:
```
['a', ' b', ' guido']
```

In [98]:
```
val
```
Out[98]: 'a, b, guido'

In [106]:
```
pieces = [x.strip() for x in val.split(',')] # pieces is now an iterab
pieces
```
Out[106]: ['a', 'b', 'guido']

In [102]:
```
first, second, third = pieces
second
```
Out[102]: 'b'

In [103]:
```
first + '::' + second + '::' + third
```
Out[103]: 'a::b::guido'

In [104]:
```
'::'.join(pieces)
```
Out[104]: 'a::b::guido'

In [108]:
```
'guido' in val
```
Out[108]: True

In [115]:
```
val.index(',')
```
Out[115]: 1

In [116]:
```
val.count(',')
```
Out[116]: 2

In [122]:
```
val.replace(',', '::')
```
Out[122]: 'a:: b:: guido'

Built-in string manipulations

count -- Return the number of non-overlapping occurences of substring in the string endswith -- Returns True if string ends with suffix. startswith -- Returns True if string starts with prefix join -- Use string as delimiter for concatenating a sequence of other strings index -- Return position of first character in substring if found in the strin; raises ValueError if not found
find -- Return position of first character of first occurence of substring in the string; like index, but returns -1 if not found. rfind -- Return position of first character of last occurence of substring in the string; -1 if not found. replace -- Replace occurrences of string with another string strip -- Trim whitespace, including newlines; equivalent to x.strip() (and rstrip, lstrip, respectively) rstrip,
lstrip
split -- Break string into list of substrings using passed delimiter
lower -- Convert a alphabet characters to lowercase
upper -- Convert alphabet characters to uppercase
casefold -- Conver haracters to lowercase, and convert any region-specific variable character combinations to a common comparable form.
ljust, rjust -- Left justify or right justify, respectively; pad opposite side of string with spaces(or some other fill character) to return a string with a minimum width.

Vectorised String Functions in Pandas

In [132]:
```
data = {'Dave': 'dave@google.com', 'Steve':'steve@google.com', 'Rob':'
data = pd.Series(data)
data
```

Out[132]:
```
Dave      dave@google.com
Steve    steve@google.com
Rob        rob@gmail.com
Wes                   NaN
dtype: object
```

In [133]:
```
data.isnull()
```

Out[133]:
```
Dave     False
Steve    False
Rob      False
Wes       True
dtype: bool
```

In [134]:
```
data.str.contains('gmail')
```

Out[134]:
```
Dave      False
Steve     False
Rob        True
Wes         NaN
dtype: object
```

In [135]:
```
data.str[:5]
```

Out[135]:
```
Dave      dave@
Steve     steve
Rob       rob@g
Wes         NaN
dtype: object
```

Vectorized String Methods

cat -- Concateate strings element-wise with optional delimiter
contains -- returns boolean array if each string contains pattern/regex
count -- Count occurences of pattern extract -- Use a regular exprssion with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
endswith -- equivalent to x.endswith(pattern) for each element
startswith -- Equivalent to x.startsith(pattern)
findall - Compute a list of all occurences of pattern/regex for each string
get -- index into each element (retrieve ith element)
isalnum -- Equivalent to built-in str.alnum
isalpha -- Equivalent to built-in str.isalpha
isdecimal -- Equivalent to built-in str.isdecimal
isdigit, islower, isnumeric, isupper -- Equivalent to built-in str.***
join -- join strings in each element of the Series with passed separator
len -- compute the lnegth of each string
match -- User re.match with the passed regular expression on each element, returning True or Flase whether it matches
extract -- Extract captured group element by indexing each string
pad -- add whitespace to left, right or both sides of the string
center -- Equivalent to pad(side='both')
repeat -- Duplicate values e.g. str.repeat(3) is equal to x * 3 for each string
replace -- Replace occurances of pattern/regex with some oher string
slice -- Slice each string in Series
split -- Split strings on delimiter or regular expression
strip -- Trim whitespace from both sides, including newlines
rstrip -- trim whitespace on right side
lstrip -- trim whitespace on left side

In [ ]: