Module 5

Use Case 1

LESSON 1
Data Preparation: **Cleaning** and Wrangling

José Manuel García Nieto – University of Málaga

- The context of data analytics

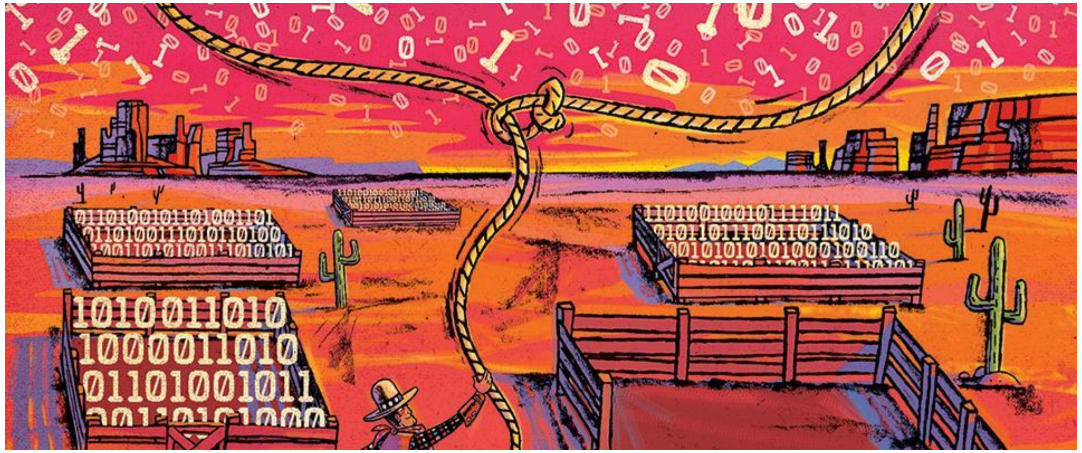
  - One of the most common mistakes of data analytic projects is thinking that they start with “analysis”

  - In their natural form, the “Raw Data” usually have registry errors that make an exact analysis impossible

  - All the data records have to be pre-processed:

    - **In other words, the data must be cleaned, unified, consolidated and normalized, so that it can be used and extract valuable information.**

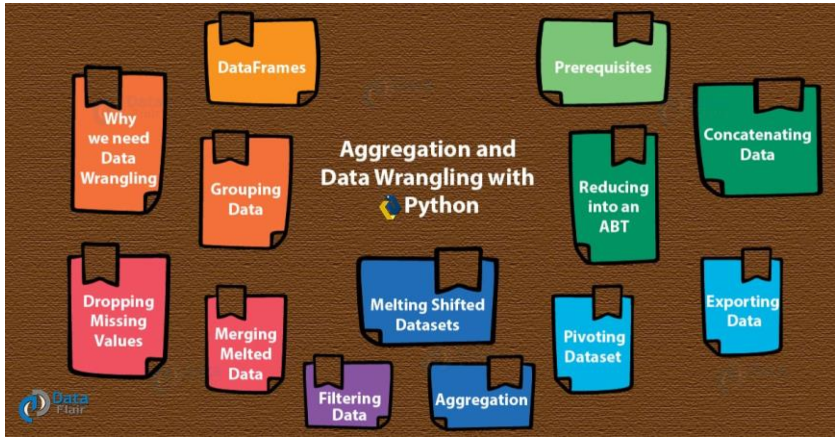
- The context of data analytics

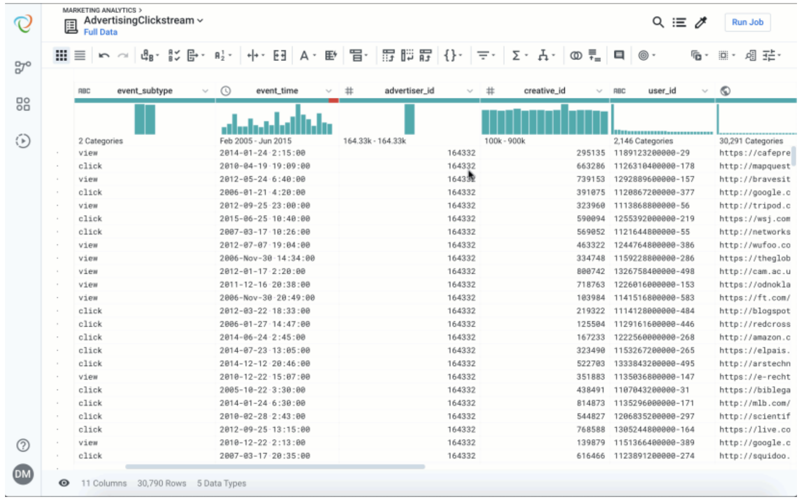


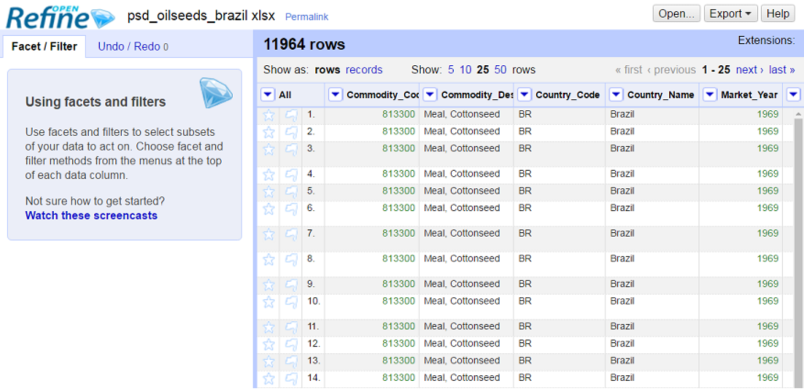
  - Any data project needs a previous step to be successful, which comprises two main tasks:
    - Data Cleaning
    - Data Wrangling

  - In fact, they are usually the most time consuming for data analysts
    - According to a survey conducted in 2017, a data analyst can spend, on average, 80% of their time on Data Wrangling.

- Tools and methods for data preparation
  - Commercial: Trifacta https://www.trifacta.com/
  - Open source: Open Refine https://openrefine.org/
  - Do it by yourself!







OPEN Refine

- Data Cleaning and Preparation
  - Handling Missing Data
    - Filtering Out Missing Data
    - Filling In Missing Data
  - Data Transformation
    - Removing Duplicates
    - Transforming Data Using a Function or Mapping
    - Replacing Values
    - Renaming Axis Indexes
    - Discretization and Binning
    - Detecting and Filtering Outliers
    - Permutation and Random Sampling
    - Derived Variables
  - String Manipulation
    - String Object Methods
    - Regular Expressions

- Data Cleaning and Preparation
  - Handling Missing Data
    - Filtering Out Missing Data
    - Filling In Missing Data
  - Data Transformation
    - Removing Duplicates
    - Transforming Data Using a Function or Mapping
    - Replacing Values
    - Renaming Axis Indexes
    - Discretization and Binning
    - Detecting and Filtering Outliers
    - Permutation and Random Sampling
    - Derived Variables
  - String Manipulation
    - String Object Methods
    - Regular Expressions

- Handling Missing Data
  - For numeric data, pandas uses the floating-point value **NaN (Not a Number)** to represent missing data

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado’])

In [11]: string_data
Out[11]:
0 aardvark
1 artichoke
2 NaN
3 avocado
dtype: object
```

```
In [12]: string_data.isnull()

Out[12]:
0 False
1 False
2 True
3 False
dtype: bool
```

- Handling Missing Data
  - The built-in Python **None** value is also treated as NA in object arrays

```
In [13]: string_data[0] = None

In [14]: string_data.isnull()
Out[14]:
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

- Handling Missing Data: Filtering Out Missing Data
  - **dropna** on a Series, it returns the Series with only the non-null data and index values

```
In [15]: from numpy import nan as NA

In [16]: data = pd.Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()

Out[17]:
0 1.0
2 3.5
4 7.0
dtype: float64
```

This is equivalent to:

```
In [18]: data[data.notnull()]

Out[18]:
0 1.0
2 3.5
4 7.0
dtype: float64
```

- Handling Missing Data: Filtering Out Missing Data
  - **dropna** on Dataframe, it drops rows or columns that are all NA or only those containing any NAs. dropna by default drops any row containing a missing value:

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

In [20]: cleaned = data.dropna()

In [21]: data

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

In [22]: cleaned

Out[22]:
     0     1     2
0    1.0   6.5   3.0
```

  - Passing how='all' will only drop rows that are all NA:

```
In [23]: data.dropna(how='all')
Out[23]:
     0     1     2
0    1.0   6.5   3.0
1    1.0   NaN   NaN
3    NaN   6.5   3.0
```

  - To drop columns, pass axis=1:

```
In [24]: data.dropna(axis=1, how='all')
```

- Handling Missing Data: Filtering Out Missing Data
  - Filter out DataFrame rows tends to concern time series data
  - Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the thresh argument:

```
In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [31]: df.dropna()
Out[31]:
          0         1         2
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [32]: df.dropna(thresh=2)
Out[32]:
          0         1         2
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

- Handling Missing Data: Filling In Missing Data
  - Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the "holes" in any number of ways
  - **fillna** method with a constant replaces missing values with that value
  - **fillna** with a dict, can use a different fill value for each column
  - **fillna** returns a new object, but it is possible to modify the existing object in-place

```
In [33]: df.fillna(0)
Out[33]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [34]: df.fillna({1: 0.5, 2: 0})
Out[34]:
          0         1         2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
3  1.246435  0.500000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [35]: _ = df.fillna(0, inplace=True)
```

- Handling Missing Data: Filling In Missing Data
  - A set of interpolation methods can be also used with **fillna:**
    - **ffill:** to fill from front values
    - **bfill:** to backward fill the missing values
  - pass the mean or median value of a Series

```
In [40]: df
Out[40]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772       NaN  1.343810
3 -0.713544       NaN -2.370232
4 -1.860761       NaN       NaN
5 -1.265934       NaN       NaN
```

```
In [41]: df.fillna(method='ffill')
Out[41]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761  0.124121 -2.370232
5 -1.265934  0.124121 -2.370232
```

```
In [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761       NaN -2.370232
5 -1.265934       NaN -2.370232
```

```
In [43]: data = pd.Series([1., NA, 3.5, NA, 7])

In [44]: data.fillna(data.mean())
Out[44]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

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

- Data Cleaning and Preparation
  - Handling Missing Data
    - Filtering Out Missing Data
    - Filling In Missing Data
  - Data Transformation
    - Removing Duplicates
    - Transforming Data Using a Function or Mapping
    - Replacing Values
    - Renaming Axis Indexes
    - Discretization and Binning
    - Detecting and Filtering Outliers
    - Permutation and Random Sampling
    - Derived Variables
  - String Manipulation
    - String Object Methods
    - Regular Expressions

- Data Transformation: Removing Duplicates
  - DataFrame method **duplicated** returns a boolean Series indicating whether each row is a duplicate
  - **drop_duplicates** solves this issue
  - Specify any subset of columns to detect duplicates

```
In [46]: 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]: data.duplicated()
Out[47]:
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

```
In [48]: data.drop_duplicates()
Out[48]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

```
In [49]: data['v1'] = range(7)
```

```
In [51]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[51]:
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
6  two   4   6
```

Keep='last' will return the last duplicate
Instead of the first one

- Data Transformation: Transforming Data Using a Function or Mapping
  - For many datasets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame
    - Consider the following data collected about various kinds of meat

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

```
In [53]: data
Out[53]:
          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
```

We want to add a column indicating the type of animal that each food came from. Let's write down a mapping of each distinct meat type to the kind of animal:

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

```
In [55]: lowercased = data['food'].str.lower()
```

```
In [56]: lowercased
Out[56]:
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
```

Then we map

```
In [57]: data['animal'] = lowercased.map(meat_to_animal)

In [58]: data
Out[58]:
          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
```

Alternatively, we may use a lambda function to do the same

```
In [59]: data['food'].map(lambda x: meat_to_animal[x.lower()])
```

Data Transformation: Replacing Values

- **replace** provides a simpler and more flexible way to substitute values

  Let's consider this Series

```
In [61]: data
Out[61]:
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

  - We can use replace, producing a new Series (unless you pass inplace=True)

```
In [62]: data.replace(-999, np.nan)
```

  - **replace** multiple values at once, you instead pass a list and then the substitute value

```
In [63]: data.replace([-999, -1000], np.nan)
```

  - To use a different replacement for each value, pass a list of substitutes

```
In [64]: data.replace([-999, -1000], [np.nan, 0])
```

  - The argument passed can also be a dict

```
In [65]: data.replace({-999: np.nan, -1000: 0})
```

```
Out[65]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

Data Transformation: Renaming Axis Indexes

- Axis labels can be similarly transformed by a function or mapping of some form to produce new differently labelled objects

```
In [66]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
   ....:                     index=['Ohio', 'Colorado', 'New York'],
   ....:                     columns=['one', 'two', 'three', 'four'])
```

- Axis indexes have a map method

```
In [67]: transform = lambda x: x[:4].upper()
```

```
In [69]: data.index = data.index.map(transform)

In [70]: data
Out[70]:
      one  two  three  four
OHIO    0    1      2     3
COLO    4    5      6     7
NEW     8    9     10    11
```

- Data Transformation: Renaming Axis Indexes
  - **rename** method: to create a transformed version of a dataset without modifying the original

```
In [71]: data.rename(index=str.title, columns=str.upper)
Out[71]:
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11
```

  - **rename** can be used in conjunction with a dict-like object providing new values for a subset of the axis labels

```
In [72]: data.rename(index={'OHIO': 'INDIANA'},
   ....:             columns={'three': 'peekaboo'})
Out[72]:
         one  two  peekaboo  four
INDIANA    0    1         2     3
COLO       4    5         6     7
NEW        8    9        10    11
```

To modify a dataset in-place, pass **inplace=True**

```
In [73]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
```

- Data Transformation: Discretization and Binning
  - Continuous data is often discretized or otherwise separated into “bins” for analysis

```
In [75]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

  - Let’s divide these into bins of 18 to 25, 26 to 35, 36 to 60, and finally 61 and older: use **cut** function in pandas

```
In [76]: bins = [18, 25, 35, 60, 100]

In [77]: cats = pd.cut(ages, bins)

In [78]: cats
Out[78]:
[(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]]
```

special Categorical object

Notación	Intervalo
$[a,b]$	$a \le x \le b$
$[a,b)$	$a \le x < b$
$(a,b]$	$a < x \le b$
(a,b)	$a < x < b$

Notación	Intervalo
$[a,b]$	$a \le x \le b$
$[a,b)$	$a \le x < b$
$(a,b]$	$a < x \le b$
(a,b)	$a < x < b$

- Data Transformation: Discretization and Binning
  - **pandas.cut**:
    - like an array of strings indicating the bin name
    - internally it contains a **categories** array specifying the distinct category names along with a labelling for the ages data in the **codes** attribute

```
In [79]: cats.codes
Out[79]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [80]: cats.categories
Out[80]:
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]
              closed='right',
              dtype='interval[int64]')

In [81]: pd.value_counts(cats)
Out[81]:
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

Histograming

a parenthesis means that the side is *open*, while the square bracket means it is *closed* (inclusive);

It can be changed by passing right=False

```
In [82]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
```

- Data Transformation: <u>Discretization and Binning</u>
  - **pandas.cut**:
    - It allows to pass own bin names by passing a list or array to the labels option

```
In [83]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

In [84]: pd.cut(ages, bins, labels=group_names)
Out[84]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, Mid
dleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

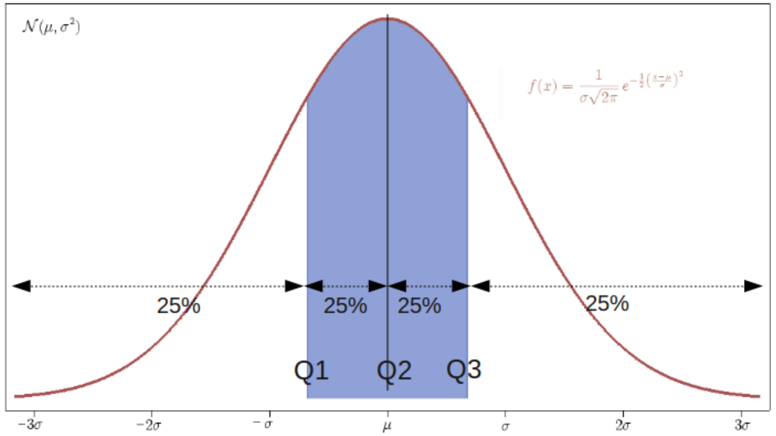
    - Compute equal-length bins based on the minimum and maximum values in the data

```
In [85]: data = np.random.rand(20)

In [86]: pd.cut(data, 4, precision=2)
Out[86]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55], ..., (0.34
, 0.55], (0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0.12, 0.34]]
Length: 20
Categories (4, interval[float64]): [(0.12, 0.34] < (0.34, 0.55] < (0.55, 0.76] <
(0.76, 0.97]]
```

precision=2 option limits the decimal precision to two digits

- Data Transformation: Discretization and Binning
  - **pandas.qcut**: bins the data based on sample **quantiles**
    - by definition roughly equal-size bins are obtained

```
In [87]: data = np.random.randn(1000)  # Normally distributed

In [88]: cats = pd.qcut(data, 4)  # Cut into quartiles

In [89]: cats
Out[89]:
[(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928], (-0.0265, 0.62]
, ..., (-0.68, -0.0265], (-0.68, -0.0265], (-2.95, -0.68], (0.62, 3.928], (-0.68,
 -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.68] < (-0.68, -0.0265] < (-0.0265,
 0.62] <
                                    (0.62, 3.928]]
```




```
In [90]: pd.value_counts(cats)
Out[90]:
(0.62, 3.928]       250
(-0.0265, 0.62]     250
(-0.68, -0.0265]    250
(-2.95, -0.68]      250
dtype: int64
```

- Similar to cut you can pass your own quantiles (numbers between 0 and 1, inclusive)

```
In [91]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
```

- Data Transformation: Detecting and Filtering Outliers
  - It's a matter of array indexing
  - Having this DataFrame:

```
In [92]: data = pd.DataFrame(np.random.randn(1000, 4))

In [93]: data.describe()
Out[93]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.049091     0.026112    -0.002544    -0.051827
std       0.996947     1.007458     0.995232     0.998311
min      -3.645860    -3.184377    -3.745356    -3.428254
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
75%       0.756646     0.695298     0.699046     0.623331
max       2.653656     3.525865     2.735527     3.366626
```

    - Let's find values in one of the columns exceeding 3 in absolute value:

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

In [95]: col[np.abs(col) > 3]
Out[95]:
41     -3.399312
136    -3.745356
Name: 2, dtype: float64
```

    - cap values outside the interval –3 to 3:

```
In [97]: data[np.abs(data) > 3] = np.sign(data) * 3

In [98]: data.describe()
Out[98]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.050286     0.025567    -0.001399    -0.051765
std       0.992920     1.004214     0.991414     0.995761
min      -3.000000    -3.000000    -3.000000    -3.000000
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
75%       0.756646     0.695298     0.699046     0.623331
max       2.653656     3.000000     2.735527     3.000000
```

np.sign(data) produces 1 and –1 values based on whether the values in data are positive or negative

- Data Transformation: Detecting and Filtering Outliers
  - Remove all the random numbers that lie in the lowest **quantile** and the highest quantile

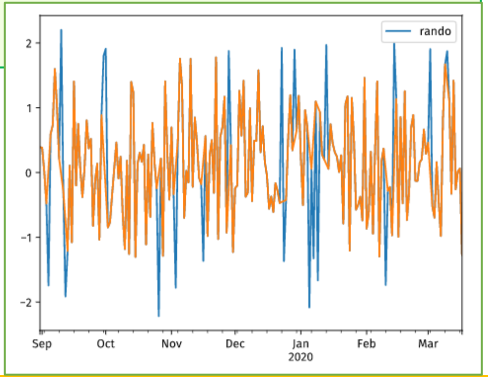
```
size=200

x = pd.Series(np.random.normal(size=size)) # 200 values
x = x[x.between(x.quantile(.15), x.quantile(.85))] # without outliers

print(x) # Now only 140 values
```

```
0       0.691716
1       0.519569
3      -0.145321
4      -0.490216
5      -0.357589
          ...
191    -0.254481
192     0.808355
194    -0.755343
196    -0.132525
199    -0.494869
Length: 140, dtype: float64
```

  - With two axes

```
import datetime

todays_date = datetime.datetime.now().date()
dates = pd.date_range(todays_date-datetime.timedelta(10), periods=size, freq='D')

rando_nums = np.random.normal(size=size)
columns = ['rando']

df = pd.DataFrame(rando_nums, index=dates, columns=columns)
df              # 200 random numbers indexed by days in a week
df.plot().get_figure()
```

```
y = df['rando']
removed_outliers = y.between(y.quantile(.05), y.quantile(.95))

print(str(y[removed_outliers].size) + "/" + str(size) + " data points remain.")

y[removed_outliers].plot().get_figure()
```



  - **np.percentile**

- Data Transformation: Permutation and Random Sampling
  - **permutation** with the length of the axis you want to permute produces an array of integers indicating the new ordering

```
In [100]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))

In [101]: sampler = np.random.permutation(5)

In [102]: sampler
Out[102]: array([3, 1, 4, 2, 0])
```

```
In [103]: df
Out[103]:
    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 [104]: df.take(sampler)
Out[104]:
    0   1   2   3
3  12  13  14  15
1   4   5   6   7
4  16  17  18  19
2   8   9  10  11
0   0   1   2   3
```

  - To select a random subset without replacement, you can use the **sample** method on Series and DataFrame

```
In [105]: df.sample(n=3)
Out[105]:
    0   1   2   3
3  12  13  14  15
4  16  17  18  19
2   8   9  10  11
```

- Data Transformation: Derived Variables
  - Create one or more variables based on other variable(s)
    - Ex) From the "Date" variable derive the "Year", "Month", "Day", "Weekday", etc.

Date
2009-8-21
2013-8-27
2019-2-11

Year	Month	Day	Weekday
2009	8	21	Friday
2013	8	27	Tuesday
2019	2	11	Monday

- Ex) From the "X_coordinate" and "Y_coordinate" derive the "Distance" variable:
  - Distance = $\sqrt{\text{X\_coordinate}^2 + \text{Y\_coordinate}^2}$

X_coordinate	Y_coordinate
3	4
5	12
9	12

Distance
5
13
15

- Data Transformation: Derived Variables
  - Dealing with categorical variables: **Dummy variables**
    - A dummy variable is a derived variable that can have only 0 or 1 as value.
    - A categorical variable generates "$Category\ count - 1$" dummy variables.
    - Ex) "Gender" variable with two category values: "male" and "female".

      → Only one dummy variable is required: "Gender_male".

Gender
female
male

↔

Gender_male
0
1

pandas.get_dummies

```
pandas.get_dummies(data, prefix=None, prefix_sep='_', dummy_na=False, columns=None, sparse=False, drop_first=False, dtype=None)
```

https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html

- Data Transformation: Derived Variables
  - Dealing with categorical variables: **Dummy variables**
    - Ex) “Gender” variable with two category values: “male” and “female”.

```
# Load the data
df = pd.read_table('http://data.princeton.edu/wws509/datasets/salary.dat',delim_whitespace=True)
# Take a look
df.head()
```

	sx	rk	yr	dg	yd	sl
0	male	full	25	doctorate	35	36350
1	male	full	13	doctorate	22	35350
2	male	full	10	doctorate	23	28200
3	female	full	7	doctorate	27	26775
4	male	full	19	masters	30	33696

```
dummy = pd.get_dummies(df['sx'])  #turns my type column into a dummy variable
dummy.head()
```

	female	male
0	0	1
1	0	1
2	0	1
3	1	0
4	0	1

- Data Transformation: Derived Variables
  - Dealing with categorical variables: **One hot encoding**
    - A Similar to dummy variables.
    - In this case, there are as many dummy variables as the count of unique category values.
      - Ex) One hot encoding for a variable that can have integer values 0~9.
        - Integer variables often represent category or class rather than the numeric value!

X0	X1	X2	X3	X4	X5	X6	X7	X8	X9
0	0	0	0	0	1	0	0	0	0

5

sklearn.preprocessing.OneHotEncoder

class `sklearn.preprocessing.OneHotEncoder`(**, categories='auto', drop=None, sparse=True, dtype=<class 'numpy.float64'>, handle_unknown='error'*) [source]

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

- Data Transformation: Derived Variables
  - Turning numerical variables into categorical:
    - From a numerical variable we can derive a categorical one that contains intervals as values.
    - These intervals can be quantiles or custom made.
    - From the categorical variable, the corresponding dummy variables can be generated as usual.
      - Ex) A numerical “Age” variable can be converted into “Age_10”, “Age_20”, etc.

Age
8
17
26
63

⟷

Age_10	Age_20	Age_30	Age_40	Age_50	Age_60	Age_70	Age_80
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	1	0	0

- Data Cleaning and Preparation
  - Handling Missing Data
    - Filtering Out Missing Data
    - Filling In Missing Data
  - Data Transformation
    - Removing Duplicates
    - Transforming Data Using a Function or Mapping
    - Replacing Values
    - Renaming Axis Indexes
    - Discretization and Binning
    - Detecting and Filtering Outliers
    - Permutation and Random Sampling
    - Derived Variables
  - String Manipulation
    - String Object Methods
    - Regular Expressions

- String Manipulation: String Object Methods
  - In many **string munging** and scripting applications, built-in string methods are sufficient. Example
    - a comma-separated string can be broken into pieces with split

```
In [134]: val = 'a,b,  guido'

In [135]: val.split(',')
Out[135]: ['a', 'b', '  guido']
```

munging

noun [U] • COMPUTING • specialized

UK /ˈmʌŋ.ɪŋ/ US /ˈmʌŋ.ɪŋ/

Cambridge Dictionary

the process of changing data into another format (= arrangement) so that it can be used or processed:

- *data **munging***
- *A lot of munging has to take place to get the data from one schema into another.*

    - split is often combined with strip to trim whitespace (including line breaks)

```
In [136]: pieces = [x.strip() for x in val.split(',')]

In [137]: pieces
Out[137]: ['a', 'b', 'guido']
```

    - These substrings could be concatenated together with a two-colon delimiter by passing a list or tuple to the **join** method on the string '::'

```
In [140]: '::'.join(pieces)
Out[140]: 'a::b::guido'
```

- String Manipulation: String Object Methods
  - Other methods are concerned with locating substrings
    - **index** and **find** methods to detect substrings

```
In [134]: val = 'a,b,  guido'
```

```
In [141]: 'guido' in val
Out[141]: True

In [142]: val.index(',')
Out[142]: 1

In [143]: val.find(':')
Out[143]: -1
```

index raises an exception if the string isn't found (versus returning –1)

val.index(':')

    - **count** returns the number of occurrences of a particular substring

```
In [145]: val.count(',')
Out[145]: 2
```

    - **replace** will substitute occurrences of one pattern for another.

```
In [146]: val.replace(',', '::')
Out[146]: 'a::b::  guido'

In [147]: val.replace(',', '')
Out[147]: 'ab  guido'
```

It is commonly used to delete patterns, too, by passing an empty string

- String Manipulation: String Object Methods
  - Python built-in string methods

Argument	Description
`count`	Return the number of non-overlapping occurrences 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 string; raises `ValueError` if not found.
`find`	Return position of first character of *first* occurrence of substring in the string; like `index`, but returns –1 if not found.
`rfind`	Return position of first character of *last* occurrence of substring in the string; returns –1 if not found.
`replace`	Replace occurrences of string with another string.
`strip`, `rstrip`, `lstrip`	Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip`, `lstrip`, respectively) for each element.
`split`	Break string into list of substrings using passed delimiter.
`lower`	Convert alphabet characters to lowercase.
`upper`	Convert alphabet characters to uppercase.
`casefold`	Convert characters 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.

- String Manipulation: Regular Expressions
  - A single expression, commonly called a regex, is a string formed according to the regular expression language

```
In [148]: import re
```

  - Python's built-in **re** module is responsible for applying regular expressions to strings

```
In [149]: text = "foo    bar\t baz  \tqux"

In [150]: re.split('\s+', text)
Out[150]: ['foo', 'bar', 'baz', 'qux']
```

  - **re** functions fall into three categories
    - Pattern matching
    - Substitution
    - Splitting

Describing one or more whitespace characters is \s+

```
In [151]: regex = re.compile('\s+')

In [152]: regex.split(text)
Out[152]: ['foo', 'bar', 'baz', 'qux']
```

Compiling by yourself

To avoid unwanted escaping with \ in a regular expression, use *raw* string literals like **r'C:\x'** instead of the equivalent 'C:\\x'.

```
In [153]: regex.findall(text)
Out[153]: ['    ', '\t ', '  \t']
```

- String Manipulation: Regular Expressions
  - **findall** returns all matches in a string
  - **search** returns only the first match
  - **match** only matches at the beginning of the string

```python
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}'

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

```
In [155]: regex.findall(text)
Out[155]:
['dave@google.com',
 'steve@gmail.com',
 'rob@gmail.com',
 'ryan@yahoo.com']
```

```
In [156]: m = regex.search(text)

In [157]: m
Out[157]: <_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>

In [158]: text[m.start():m.end()]
Out[158]: 'dave@google.com'
```

```
In [159]: print(regex.match(text))
None
```

None, because it only will match if the pattern occurs at the start of the string

- String Manipulation: Regular Expressions

```
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}'

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

  - **sub** will return a new string with occurrences of the pattern replaced by the a new string

```
In [160]: print(regex.sub('REDACTED', text))
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

  - To segment components (username, domain name, and domain suffix) put parentheses around the parts of the pattern

```
In [161]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [162]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

  - A match object produced by this modified regex returns a tuple of the pattern components with its **groups** method

```
In [163]: m = regex.match('wesm@bright.net')

In [164]: m.groups()
Out[164]: ('wesm', 'bright', 'net')
```

- String Manipulation: Regular Expressions
  - **findall** returns a list of tuples when the pattern has groups

```
In [165]: regex.findall(text)
Out[165]:

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

  - **sub** also has access to groups in each match using special symbols like \1 and \2
    - The symbol \1 corresponds to the first matched group, \2 corresponds to the second, and so forth

```
In [166]: print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
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
```