CO7093/CO3093 - Big Data & Predictive Analytics
Lecture Notes on Sampling distributions & Summarizing data

School of Informatics
University of Leicester

1 Sampling distributions

Sampling consists in drawing a smaller set of individuals in a given population in order to reach conclusions to reach conclusions on that population parameters. This is known as *inferential statistics*.

For a given *sample size*, all possible samples represent all possible combinations of values from a population. For example, for a population of size 10, there are 45 samples without replacement of size 2. The probability distribution of a sample statistic for all possible samples of a given size is known as the *sampling distribution*. This way, we may be interested in the distribution of the sample mean.

Sampling distributions express the variability in the sample statistic computed for each sample. For example, in the sampling distribution of the mean for a numerical variable, some of the sample means might be smaller than others, some might be larger, and some might be similar to each other. Calculating the sample statistic for all possible samples would be one way but would be impractical for realistic examples. What then?

2 Central Limit Theorem

Consider the expenses for a certain number of groups of customers. Their expenses may have different distributions but bow much *total* do they spend in the shop?

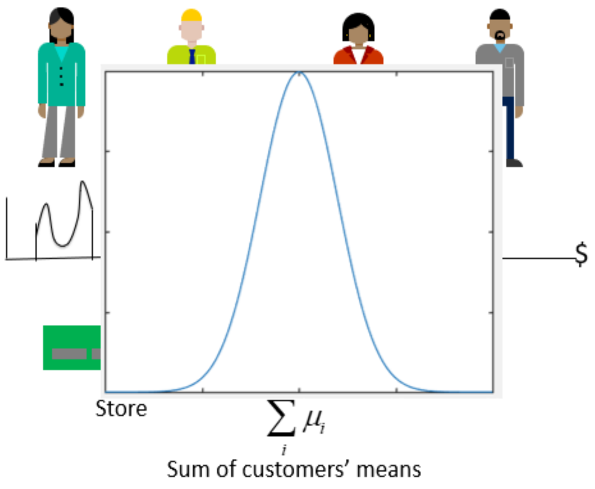
We have independent customers' samples of various sizes and for each sample i, we can calculate the average expense μ_i. We can then address the question by considering the sum of the samples' means

$$S = \sum_i \mu_i = \mu_1 + \mu_2 + \dots$$

The idea is that $S = \sum_i \mu_i$

- can be a pretty good estimator of how much the customers spend on average at the shop.
- can have a normal distribution.

When this is the case is given by the Central Limit Theorem (CLT).

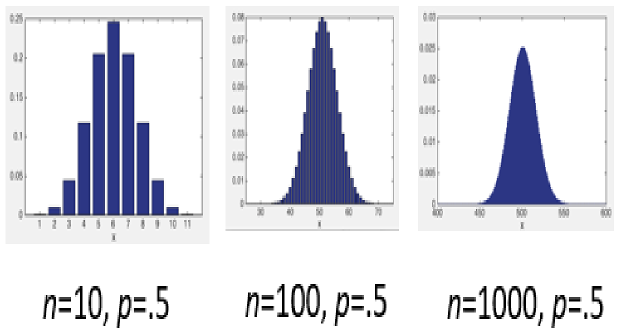


CLT(Central Limit Theorem): The sampling distribution of the mean can be approximated by the normal distribution when the sample size of all possible samples gets *large enough*.

Not only the CLT (Central Limit Theorem) enables us to make use of our knowledge about the normal distribution, but it also allows us to determine the shape of a sampling distribution. In the CLT, 'large enough' is usually understood as a sample size of greater or equal to 30.

2.1 The CLT and the Normal distribution

As we have seen in previous lectures, the Binomial distribution comes from the sum of independent random variables. Precisely, it comes from the sum of n Bernouilli trials. However, the Binomial distribution is discrete.



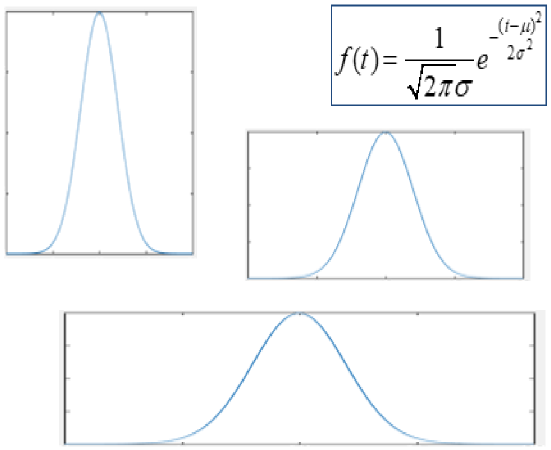
It turns out that when n is large enough, the binomial distribution becomes the normal distribution.

2.2 Sample Z-scores

The sample z-score of a sample X is the number of sample standard deviations above or below the sample mean. For a sample X (of mean $\bar{X}$ and standard deviation S), its corresponding Z score (of mean 0 and standard deviation 1) can be found by the following formula:

$$Z = \frac{X - \bar{X}}{S}$$

The mean ($\bar{X}$) and standard deviation (S) represent respectively the center and the 'width' of the bell-shaped curve. A higher value for S indicates a 'fatter' bell-shaped curve meaning a high spread of the data around the mean.



$$f(t) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(t-\mu)^2}{2\sigma^2}}$$

We will revisit this formula when we will be calculating confidence intervals in the next set of lectures.

3 Summarizing data

Data is the fuel for predictive analytics. However, data is often partial and incomplete. It provides us with a sample for a specified population, hopefully a useful sample enabling us to analyse and build up a useful model fit for a well-specified purpose. Summarizing data enables to get a touch and feel of the data at hand. From now on we will be using publically available datasets to illustrate concepts and give examples.

Datasets can be provided in different formats (.csv, .txt, .xls, etc.). In all cases, we will need to be able to

- read it and store it in an appropriate data structure;
- get some basic information, such as dimensions, column names, and statistics summary;
- perform some basic data cleaning such as removing blanks, NAs, renaming some columns, imputing missing values, etc;
- Generating simple plots like scatter plots, bar charts, histograms, box plots, etc.

3.1 Data frames

A data frame is a common data structure in Python. It is quite similar to the a table in a spreadsheet or a SQL table. It has index labels (analogous to rows) and column labels (analogous to columns). Data frame is common within the `pandas` package and represents a 2D object with columns of different or same types. Standard operations, such as aggregation, filtering, pivoting can be applied to data frames by using methods within the pandas package. These methods are summarised in a cheat sheet that can be found at the end of these notes.

To read the data, we use a *data frame*, a Python data structure comparable to a spreadsheet or a SQL table. Pandas provides methods `read_excel` and `read_csv` enabling us the read in an `.xls/.xlsx` file or a `.csv` file as follows:

```
>>> import pandas as pd
>>> data = pd.read_csv('Advertising.csv')
>>> data.head()
```

The output looks as follows:

```
      TV  Radio  Newspaper  Sales
0  230.1   37.8       69.2   22.1
1   44.5   39.3       45.1   10.4
2   17.2   45.9       69.3    9.3
3  151.5   41.3       58.5   18.5
4  180.8   10.8       58.4   12.9

[5 rows x 4 columns]
```

3.2 Basics – summary, dimensions, and structure

After reading in the dataset, we can perform certain tasks that should help us start understanding the data:

- check whether the data is read in correctly.
- determine its shape and size.
- get the column names and summary statistics of numerical variables.
- plot some variables of interest.

To create a summary statistics of the numerical variables, we can proceed as follows:

```
>>> data.describe()

               TV       Radio   Newspaper       Sales
count  200.000000  200.000000  200.000000  200.000000
mean   147.042500   23.264000   30.554000   14.022500
std     85.854236   14.846809   21.778621    5.217457
min      0.700000    0.000000    0.300000    1.600000
25%     74.375000    9.975000   12.750000   10.375000
50%    149.750000   22.900000   25.750000   12.900000
75%    218.825000   36.525000   45.100000   17.400000
max    296.400000   49.600000  114.000000   27.000000

[8 rows x 4 columns]
```

We can see that for each numerical variable (`TV, Radio, Newspaper, Sales`), we have calculated the sample size (count), mean, standard deviation, min, max, and the percentiles (25%, 50%, 75%).

At this phase, the analyst would like to know **what are the variables; their types; what to slice, group by, or aggregate; whether there is an interesting information to explore further**. Note that Pandas provides us with a certain number of methods to carry out some of those preliminary explorations:

- `columns.values` gives us all the variables in the data
- `dtypes` gives the type of variable that can be continuous or discrete.
- `groupby` enables to group together certain variables mainly those that are categorical e.g., gender since there are only two values: male and female.

Python functions commonly used to achieve these simple tasks are given the cheat sheet at the end of these notes.

3.3 Handling missing values

Missing values are usually due to unavailability during data collection or incompatibilities during data extraction from various databases. They need be checked and handled properly. If they are left untreated they can lead to the behavior between the variables not being analyzed correctly or to incorrect interpretation and incorrect inference from data.

`Nan` is the default keyword for a missing value in Python. `None` is also considered as a missing value by the `isnull` and `notnull` functions.

Missing values can be treated by the following methods:

- deleting rows or columns with missing values

```
#drop all rows with
#all columns having missing values
data.dropna(axis=0,how='all')
#drop a row even if one column has
#missing value
data.dropna(axis=0,how='any')
```

- filling the missing values in the entire dataset with some number or character variable

```
#replace all by zero
data.fillna(0)
#replace all by miss
data.fillna('miss')
#replace missing values in column 'age'
#by the average value in 'age'.
data['age'].fillna(data['age'].mean())
```

- performing a *forward* (`ffill`) or *backward* (`backfill`) *fill* meaning respectively replace the missing values with the nearest preceding non-missing value or replace the missing value with the nearest succeeding non-missing value.

```
data['age'].fillna(method='ffill')
data['age'].fillna(method='backfill')
```

3.4 Basic plotting

Python provides us with a wealth of graphs to visualise the data. Plots can help us to match our expectations of what clean data or 'good' behaviour between two variables might look like. For example, plotting data may lead us to observe a Gaussian distribution of a variable.

To visualise a dataset, we can start from basic plots such as a **scatter plot, pie chart, histogram, box-plot**, etc. To illustrate some of these different types of plots, let us consider a Bank dataset and draw some of them from within Python. This dataset is from the marketing department of a bank and provides data on the customers subscribed to a term deposit, given the information on how the bank has reached out to customers in order to sell the term deposit as well as specific information about the customer. The following code snippet loads the data into the Python interactive environment:

```
>>> import pandas as pd
>>> df = pd.read_csv('bank.csv')
```

3.4.1 Scatter plots

If we suspect that there is a correlation between the employment variation rate `emp.var.rate` and the consumer price index `cons.price.idx`, we can draw a scatter plot to see if that is the case. The following code snippet

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(df['emp.var.rate'], df['cons.price.idx'], 'bo')
```

produced the following figure From Figure 1, if there is a correlation between our two variables of interest, then that correlation looks quite weak.

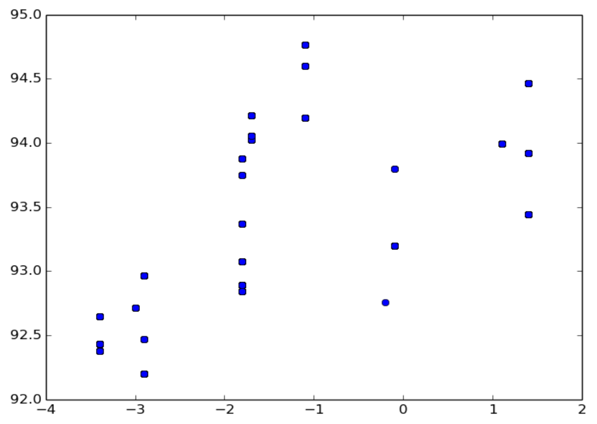


Figure 1: Scatter plot between emp.var.rate and cons.price.idx for the Bank dataset

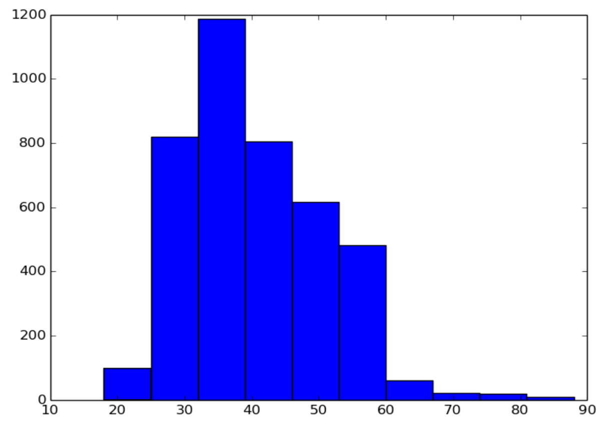


Figure 2: Histogram of the Customers' ages variable

3.4.2 Histograms

Histograms enable us to understand the distribution of a numerical variable by showing the most frequent ranges into which the variable falls. We can also check if the variable of interest is skewed or normally distributed. To plot the histogram for the variable `age` representing the ages of the bank's customers, we can write the following code snippet

```
>>> import matplotlib.pyplot as plt
>>> plt.hist(df['age'])
```

to produce this figure: Note that while plotting a histogram, one can specify the number of bins or provide a list of bins as a parameter. For example, if a variable has values ranging from 1 to 200, we can specify the number of bins to be 10 for example.

3.4.3 Boxplots

Boxplots can help us understand the distribution of a numerical variable. In a boxplot, we can identify the *quartiles*. If we sort say 100 numbers in increasing order, the first quartile will be at the 25*th* position, the second at the 50*th* position, etc. The median will be the middle position. To draw a Boxplot for the consumer price index variable from our Bank dataset, we can write the following code

```
>>> import matplotlib.pyplot as plt
>>> plt.boxplot(df['cons.price.idx'])
```

to produce this figure:

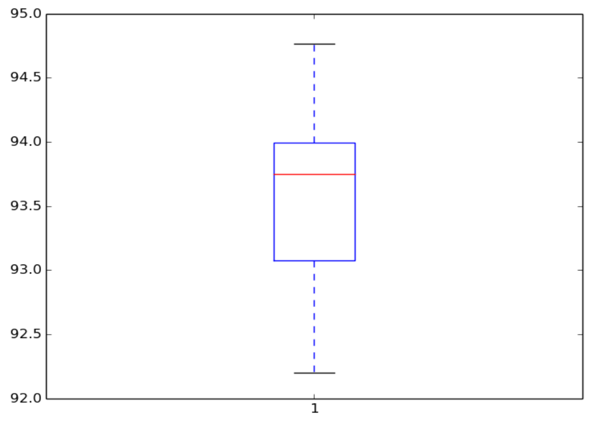


Figure 3: Box plot for the consumer price index variable

The blue box is crucial for an interpretation of the distribution of the variable. The upper horizontal line of the box indicates the third quartile while the lower one indicates the first quartile. The red line inside the box specifies the median value. The difference in the first and third quartile values is known a the IQR (Inter Quartile Range). The IQR can be used to spot outliers. For example, any value located $1.5 \times IQR$ below the first quartile or above the third quartile can be considered as an outlier.

Various other plots can be drawn depending on the problem but for the purpose of exploratory analysis, these four basic plots can help us to check our assumptions about the data and discard some of our hypotheses.

3.4.4 Pie and Bar charts

Pie charts enable us to visualise proportions for a numerical variable. Although, it can be useful to spot which slice is larger, a pie chart can be replaced by a *bar chart*. A bar chart is essentially used to compare variables with categorical data. For example, if we consider our Bank dataset and draw a bar chart of the customers who bought the term deposit and those who did not, we can use the following code

```
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> df = pd.read_csv('bank.csv')
>>> df['y']=(df['y']=='yes').astype(int) #converts yes and no to 1 and 0
>>> pd.crosstab(df.education, df.y).plot(kind='bar')
```

to produce the following figure: It is clear from Figure 4 that the frequency of purchase of the term deposit depends

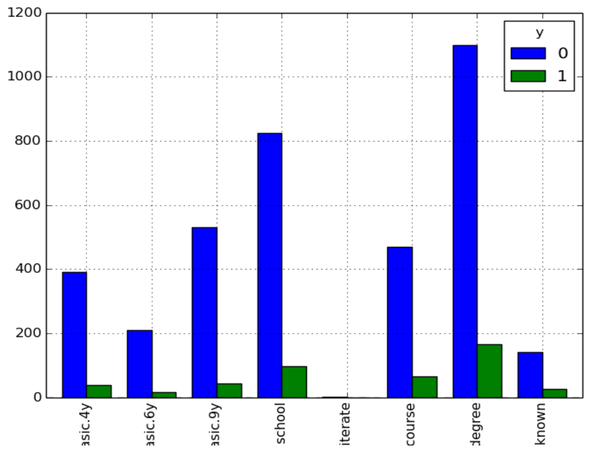


Figure 4: Bar Chart for education level and purchase

highly on the level of education and that the education level of the customer is an important predictor of the outcome variable, which is buy or not the term deposit.

4 Data model fitting

After data pre-processing and visualisation, we may want to fit a theoretical model to the data by using a

- polynomial function
- exponential function
- or other models such as Gaussian, which we will not consider in these notes.

The general idea is to consider a model or a function f, which predicts $\hat{y} = f(x)$ for each data point (x, y) so that the *residual* or error $|y - f(x)|$ is as small as possible. Instead of solving the mathematical problem, we focus on how to use the Python **scipy.optimize.curve_fit** function to fit a curve for a given data.

4.1 Polynomial fit

To fit a polynomial, we consider a function of the form:

$$f(x) = a_0 + a_1x^1 + a_2x^2 + \ldots + a_nx^n$$

wherein n is known as the degree of the polynomial f. If the degree of the polynomial is 1, then the curve is simply a line. Naturally, we should have a closer look at the data in order to decide which degree of the polynomial can fit well. However, there are ways of measuring how good is the curve fit by measuring the mismatch between predicted and observed data.

In the following code snippet, we have fitted the `total number of Corona virus confirmed cases (up to 22 October 2020)` in France with a polynomial curve:

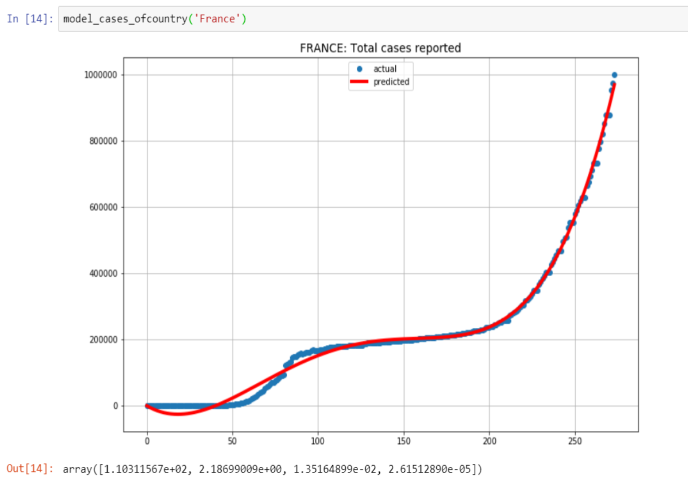
```
## Reading the data from John Hopkins
base = 'https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_covid_19_data/csse_covid_19_time_series'
confirmed = base+'/time_series_covid19_confirmed_global.csv'
deaths = base+'/time_series_covid19_deaths_global.csv'
recovers = base+'/time_series_covid19_recovered_global.csv'
#
recov = pd.read_csv(recovers)
death = pd.read_csv(deaths)
confi = pd.read_csv(confirmed)
```

```
## ... code truncated

## Data fitting
def model1(x, p1, p2, p3, p4):
    y = p1*np.power(x, 1)+p2*np.power(x,2)+p3*np.power(x,3)+p4*np.power(x,4)
    return y

# Fit the data wwith the defined model
def model_cases_ofcountry(name):
    df = get_total_confirmed_ofcountry(name)
    df = df.reset_index(drop = True)
    pars1, cov1 = curve_fit(f=model1, xdata=df.index, ydata=df, p0=[0, 0, 0, 0], bounds=(-np.inf, np.inf))
    # Standard deviations of the parameters;
# square roots of the diagonal of the cov matrix
    stdevs = np.sqrt(np.diag(cov1))
    pred1 = model1(df.index, *pars1)
    plt.figure(figsize=(12, 8))
    plt.title(name.upper()+': Total cases reported', fontsize=14)
    #plt.label('Total cases')
    g1, = plt.plot(df.index, df, 'o', lw=3, label = 'actual')
    g2, = plt.plot(df.index, pred1, color='red', lw=4, label = 'predicted:poly')
    plt.legend(handles=[g1, g2], loc='upper center')
    plt.grid()
    plt.show()
    return stdevs
```

This has produced the following figure: Note the standard deviations obtained for the coefficients of higher degree



```
In [14]: model_cases_ofcountry('France')
```

```
Out[14]: array([1.10311567e+02, 2.18699009e+00, 1.35164899e-02, 2.61512890e-05])
```

Figure 5: Total Covid-19 confirmed cases in France

terms are smaller and also the way the predicted data is closer of the observed data as time increases.

4.2 Exponential fit

To fit an exponential, we can consider a function of the form:

$$f(x) = a \exp bx$$

Similarly, we can define our model f and use the Python function `curve_fit` to estimate the parameters a and b as well as the standard deviation of the residuals as we did for the case of polynomial fitting.

4.3 Gaussian fit

To fit a Gaussian, we generally consider a function of the form:

$$f(x) = a \exp - \frac{(x-b)^2}{2c^2}$$

In general, `scikit-learn` offers kernels to build up models with a Gaussian process by using `GaussianProcessRegressor`.

Basic plotting functions and the use of pandas can also be found in the cheat sheets at the end of these notes.

Python For Data Science *Cheat Sheet*

Matplotlib

Learn Python **Interactively** at www.DataCamp.com

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.

matplotlib

1 Prepare The Data

Also see Lists & NumPy

1D Data

```
>>> import numpy as np
>>> x = np.linspace(0, 10, 100)
>>> y = np.cos(x)
>>> z = np.sin(x)
```

2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))
>>> data2 = 3 * np.random.random((10, 10))
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]
>>> U = -1 - X**2 + Y
>>> V = 1 + X - Y**2
>>> from matplotlib.cbook import get_sample_data
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

2 Create Plot

```
>>> import matplotlib.pyplot as plt
```

Figure

```
>>> fig = plt.figure()
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

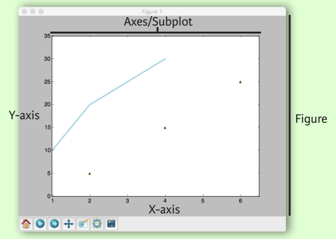
Axes

All plotting is done with respect to an `Axes`. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()
>>> ax1 = fig.add_subplot(221) # row-col-num
>>> ax3 = fig.add_subplot(212)
>>> fig3, axes = plt.subplots(nrows=2,ncols=2)
>>> fig4, axes2 = plt.subplots(ncols=3)
```

Plot Anatomy & Workflow

Plot Anatomy



Workflow

The basic steps to creating plots with matplotlib are:

1 Prepare data   2 Create plot   3 Plot   4 Customize plot   5 Save plot   6 Show plot

```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4]                                   Step 1
>>> y = [10,20,25,30]
>>> fig = plt.figure()                          Step 2
>>> ax = fig.add_subplot(111)                  Step 3
>>> ax.plot(x, y, color='lightblue', linewidth=3)   Step 3, 4
>>> ax.scatter([2,4,6],
               [5,15,25],
               color='darkgreen',
               marker='^')
>>> ax.set_xlim(1, 6.5)
>>> plt.savefig('foo.png')
>>> plt.show()                          Step 6
```

4 Customize Plot

Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)
>>> ax.plot(x, y, alpha = 0.4)
>>> ax.plot(x, y, c='k')
>>> fig.colorbar(im, orientation='horizontal')
>>> im = ax.imshow(img,
                   cmap='seismic')
```

Markers

```
>>> fig, ax = plt.subplots()
>>> ax.scatter(x,y,marker=".")
>>> ax.plot(x,y,marker="o")
```

Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)
>>> plt.plot(x,y,ls='solid')
>>> plt.plot(x,y,ls='--')
>>> plt.plot(x,y,'--',x**2,y**2,'-.')
>>> plt.setp(lines,color='r',linewidth=4.0)
```

Text & Annotations

```
>>> ax.text(1,
            -2.1,
            'Example Graph',
            style='italic')
>>> ax.annotate("Sine",
                xy=(8, 0),
                xycoords='data',
                xytext=(10.5, 0),
                textcoords='data',
                arrowprops=dict(arrowstyle="->",
                             connectionstyle="arc3"),)
```

Mathtext

```
>>> plt.title(r'$sigma_i=15$', fontsize=20)
```

Limits, Legends & Layouts

Limits & Autoscaling

`>>> ax.margins(x=0.0,y=0.1)`	Add padding to a plot
`>>> ax.axis('equal')`	Set the aspect ratio of the plot to 1
`>>> ax.set(xlim=[0,10.5],ylim=[-1.5,1.5])`	Set limits for x-and y-axis
`>>> ax.set_xlim(0,10.5)`	Set limits for x-axis

Legends

```
>>> ax.set(title='An Example Axes',
           ylabel='Y-Axis',
           xlabel='X-Axis')
```
Set a title and x-and y-axis labels

```
>>> ax.legend(loc='best')
```
No overlapping plot elements

Ticks

```
>>> ax.xaxis.set(ticks=range(1,5),
                 ticklabels=[3,100,-12,"foo"])
```
Manually set x-ticks

```
>>> ax.tick_params(axis='y',
                   direction='inout',
                   length=10)
```
Make y-ticks longer and go in and out

Subplot Spacing

```
>>> fig3.subplots_adjust(wspace=0.5,
                         hspace=0.3,
                         left=0.125,
                         right=0.9,
                         top=0.9,
                         bottom=0.1)
```
Adjust the spacing between subplots

```
>>> fig.tight_layout()
```
Fit subplot(s) in to the figure area

Axis Spines

`>>> ax1.spines['top'].set_visible(False)`	Make the top axis line for a plot invisible
`>>> ax1.spines['bottom'].set_position(('outward',10))`	Move the bottom axis line outward

3 Plotting Routines

1D Data

`>>> lines = ax.plot(x,y)`	Draw points with lines or markers connecting them
`>>> ax.scatter(x,y)`	Draw unconnected points, scaled or colored
`>>> axes[0,0].bar([1,2,3],[3,4,5])`	Plot vertical rectangles (constant width)
`>>> axes[1,0].barh([0.5,1,2.5],[0,1,2])`	Plot horiontal rectangles (constant height)
`>>> axes[1,1].axhline(0.45)`	Draw a horizontal line across axes
`>>> axes[0,1].axvline(0.65)`	Draw a vertical line across axes
`>>> ax.fill(x,y,color='blue')`	Draw filled polygons
`>>> ax.fill_between(x,y,color='yellow')`	Fill between y-values and 0

Vector Fields

`>>> axes[0,1].arrow(0,0,0.5,0.5)`	Add an arrow to the axes
`>>> axes[1,1].quiver(y,z)`	Plot a 2D field of arrows
`>>> axes[0,1].streamplot(X,Y,U,V)`	Plot 2D vector fields

Data Distributions

`>>> ax1.hist(y)`	Plot a histogram
`>>> ax3.boxplot(y)`	Make a box and whisker plot
`>>> ax3.violinplot(z)`	Make a violin plot

2D Data or Images

```
>>> fig, ax = plt.subplots()
>>> im = ax.imshow(img,
                   cmap='gist_earth',
                   interpolation='nearest',
                   vmin=-2,
                   vmax=2)
```
Colormapped or RGB arrays

`>>> axes2[0].pcolor(data2)`	Pseudocolor plot of 2D array
`>>> axes2[0].pcolormesh(data)`	Pseudocolor plot of 2D array
`>>> CS = plt.contour(Y,X,U)`	Plot contours
`>>> axes2[2].contourf(data1)`	Plot filled contours
`>>> axes2[2]= ax.clabel(CS)`	Label a contour plot

5 Save Plot

Save figures

```
>>> plt.savefig('foo.png')
```

Save transparent figures

```
>>> plt.savefig('foo.png', transparent=True)
```

6 Show Plot

```
>>> plt.show()
```

Close & Clear

`>>> plt.cla()`	Clear an axis
`>>> plt.clf()`	Clear the entire figure
`>>> plt.close()`	Close a window

Python For Data Science *Cheat Sheet*

Pandas Basics

Learn Python for Data Science Interactively at www.DataCamp.com

Pandas

The **Pandas** library is built on NumPy and provides easy-to-use **data structures** and **data analysis** tools for the Python programming language.

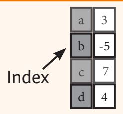
Use the following import convention:

```
>>> import pandas as pd
```

Pandas Data Structures

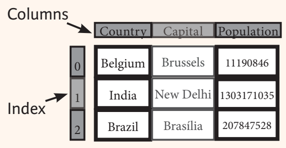
Series

A **one-dimensional** labeled array capable of holding any data type



```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame



A **two-dimensional** labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasília'],
            'Population': [11190846, 1303171035, 207847528]}

>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Also see NumPy Arrays

Getting

```
>>> s['b']
 -5
```

Get one element

```
>>> df[1:]
   Country    Capital   Population
 1   India  New Delhi   1303171035
 2  Brazil   Brasília    207847528
```

Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[[0],[0]]
 'Belgium'
>>> df.iat([0],[0])
 'Belgium'
```

Select single value by row & column

By Label

```
>>> df.loc[[0], ['Country']]
 'Belgium'
>>> df.at([0], ['Country'])
 'Belgium'
```

Select single value by row & column labels

By Label/Position

```
>>> df.ix[2]
 Country      Brazil
 Capital    Brasília
 Population  207847528
```

Select single row of subset of rows

```
>>> df.ix[:,'Capital']
 0      Brussels
 1     New Delhi
 2      Brasília
```

Select a single column of subset of columns

```
>>> df.ix[1,'Capital']
 'New Delhi'
```

Select rows and columns

Boolean Indexing

`>>> s[~(s > 1)]`	Series `s` where value is not >1
`>>> s[(s < -1) | (s > 2)]`	`s` where value is <-1 or >2
`>>> df[df['Population']>1200000000]`	Use filter to adjust DataFrame

Setting

`>>> s['a'] = 6`	Set index `a` of Series `s` to 6

Dropping

`>>> s.drop(['a', 'c'])`	Drop values from rows (axis=0)
`>>> df.drop('Country', axis=1)`	Drop values from columns(axis=1)

Sort & Rank

`>>> df.sort_index()`	Sort by labels along an axis
`>>> df.sort_values(by='Country')`	Sort by the values along an axis
`>>> df.rank()`	Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

`>>> df.shape`	(rows,columns)
`>>> df.index`	Describe index
`>>> df.columns`	Describe DataFrame columns
`>>> df.info()`	Info on DataFrame
`>>> df.count()`	Number of non-NA values

Summary

`>>> df.sum()`	Sum of values
`>>> df.cumsum()`	Cummulative sum of values
`>>> df.min()/df.max()`	Minimum/maximum values
`>>> df.idxmin()/df.idxmax()`	Minimum/Maximum index value
`>>> df.describe()`	Summary statistics
`>>> df.mean()`	Mean of values
`>>> df.median()`	Median of values

Applying Functions

`>>> f = lambda x: x*2`	
`>>> df.apply(f)`	Apply function
`>>> df.applymap(f)`	Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
 a     10.0
 b      NaN
 c      5.0
 d      7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
 a    10.0
 b    -5.0
 c     5.0
 d     7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
```

Read multiple sheets from the same file

```
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Read and Write to SQL Query or Database Table

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:')
>>> pd.read_sql("SELECT * FROM my_table;", engine)
>>> pd.read_sql_table('my_table', engine)
>>> pd.read_sql_query("SELECT * FROM my_table;", engine)
```

`read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()`

```
>>> pd.to_sql('myDf', engine)
```