Introduction to Python

Python is a widely used general-purpose, high-level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foudation.

What is An Interpreter?

Before we start on how to start your first coding, we will need to have an **interpreter**. An **interpreter** is computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.

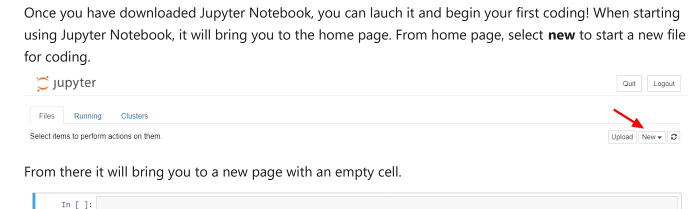
One of the few best interpreter is ***Jupyter Notebook***. Jupyter Notebook is an open source software open-standards, and services for interactive computing across dozens of programming languages. Link for downloading: **https://jupyter.org**

Other alternative is by downloading ***Anaconda Navigator***. In Anaconda Navigator, there are variety of interpreter to choose from.
Link for downloading: **https://www.anaconda.com/products/individual#windows**

Anaconda navigator is available for Windows, Mac and Linux. Tutorial for installing are as below:
Windows: **https://docs.anaconda.com/anaconda/install/windows/**
MacOS: **https://docs.anaconda.com/anaconda/install/mac-os/**
Linux: **https://docs.anaconda.com/anaconda/install/linux/**

Getting Started

Once you have downloaded Jupyter Notebook, you can lauch it and begin your first coding! When starting using Jupyter Notebook, it will bring you to the home page. From home page, select **new** to start a new file for coding.



From there it will bring you to a new page with an empty cell.

Notice the color of the cell is now in blue? There are two type of color for the cell indicating different mode.
Blue: **Command mode**
Green: **Edit mode**

To view every command in different mode, press the **"H"** key.

Command Mode (press `Esc` to enable) — Edit Shortcuts

- `F` : find and replace
- `Ctrl-Shift-F` : open the command palette
- `Ctrl-Shift-P` : open the command palette
- `Enter` : enter edit mode
- `P` : open the command palette
- `Shift-Enter` : run cell, select below
- `Ctrl-Enter` : run selected cells
- `Shift-J` : extend selected cells below
- `Ctrl-A` : select all cells
- `A` : insert cell above
- `B` : insert cell below
- `X` : cut selected cells
- `C` : copy selected cells
- `Shift-V` : paste cells above

Edit Mode (press `Enter` to enable)

- `Tab` : code completion or indent
- `Shift-Tab` : tooltip
- `Ctrl-]` : indent
- `Ctrl-[` : dedent
- `Ctrl-A` : select all
- `Ctrl-Right` : go one word right
- `Ctrl-Backspace` : delete word before
- `Ctrl-Delete` : delete word after
- `Ctrl-Y` : redo
- `Alt-U` : redo selection

Once you are familiar with the control and keys of Jupyter Notebook, then we can proceed to writing your first code.😎

Basic Commands in Python

When using Python, we can use `print` command for getting an output. For example, we can print a simple command: **"Hello World!"**.

```
print("Hello World!")
```

** Please note that every code must have open and close bracket '"()"' in order for it to run.

If the code is **not complete**, the system will display a **syntax error**.

We can also include a comment before the code by adding `#` to provide information or making notes.

```
#This is a comment
#It can be as long as you like
print("Hello World!")
```

Variable and Data type

- Variable are containers for storing data values.
- Data type is the type of data that are able to be stored.

Few basic variable that we can see commonly when getting in touch with programming language is `int`, `float`, `str` which are *integer, float* and *string* respectively.
Here is a list of data type that is built-in default in Python:

- Text Type : `str`
- Numeric Types : `int`, `float`, `complex`
- Sequence Types: `list`, `tuple`, `range`
- Mapping Type : `dict`
- Set Types : `set`, `frozenset`
- Boolean Type : `bool`
- Binary Types : `bytes`, `bytearray`, `memoryview`

Knowing what type of data you are working is **extremely** important since different data type require different notations. Using a **wrong** notations would result in misinformation or syntax error.

```
#This is an example of different data type gives
#different meaning and why different notations would
#affect it
print("2+2")

#Now by removing the "" symbol , the data type
#changes from text type to numeric type
print(2+2)
```

We can also determine the type of data by using `type`.

```
x= 5
y= John
print(type(x))
print(type(y))
```

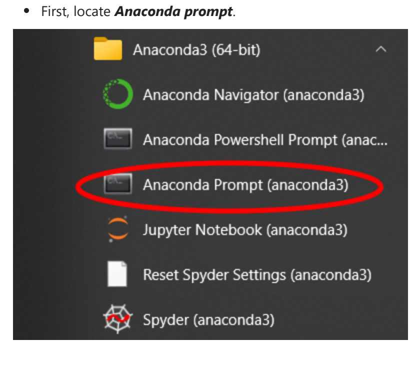
That is all for the basic commands for Python. For more basic commands there is always website that provide a better tutorial for example w3schools, code cademy and many more.

Python PIP

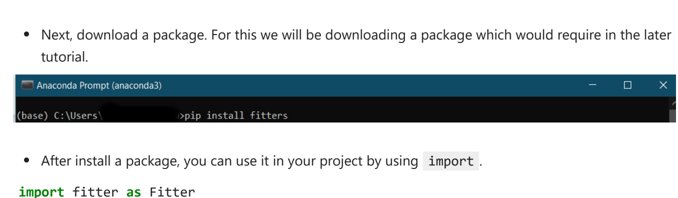
- PIP: A package manager for Python packages. It is build in defualt for Python version 3.4 or later.
- Package: Contains all the files you need for a module.
- Modules: Python code libraries you can include in your project.

To install a package:

- First, locate ***Anaconda prompt***.



- Next, download a package. For this we will be downloading a package which would require in the later tutorial.



- After install a package, you can use it in your project by using `import`.

```
import fitter as Fitter
f= fitter.Fitter()
```

For more packages, you can visit **https://pypi.org/****

Project Introduction

Now that you know the basic in Python and how to import a package, it is time to start your first project. The project that we are going to discuss now is relating to offshore structures.

What is Offshore structures?

Offshore structure is a platform that is used to extract oil and gas. It consists of fixed, complaint and floatable type platform.

Realiability

When we are discussing about offshore structure, we need to account for its structural integrity as the platforms ages. Reliability of a system is an analytical problem that involves both statistical and engineering aspects. It must be given critical attention throughout the life of a system including its development, design, production, quality control, shipping, installation, operation and maintenance. [1][2]

Probability of Failure

Probability of failure is the probability of a component failing. In basic probability, we know that probability is the numerical descriptions of how likely an event is to occur, or how likely it is that a proposition is true. The probability of an event is a number between 0 and 1, where, rougly speaking, 0 indicates impossibility of the event and 1 indicates certainty.[3]

The same goes here, the probability of failure is the performance below a given standard (from 0 to 1, where 1 is an absolute failure). The mathematical expression for probability of failure given as below:

Probability of failure (POF) can be expressed as;

$$POF = P(S \le L) = 1 - P(L \le S) = 1 - R \quad (1)$$

$$POF = 1 - \int_{-\infty}^{\infty} f_S(s)\left[\int_{-\infty}^{s} f_L(l)dl\right]ds \quad (2)$$

$$POF = \int_{-\infty}^{\infty}[1 - F_L(s)]\, f_S(s)ds \quad (3)$$

where $f_S(s)$ and $f_L(l)$ are the probability density function (PDF) of the strength and load, respectively. While $F_L(s)$ is probability distribution function (PDF^*) of L in a unit of strength.

Alternate for the probability of failure (POF) can also be expressed as;

$$POF = 1 - R = 1 - (S \ge L) \quad (4)$$

$$POF = 1 - \int_{-\infty}^{\infty} f_L(l)\left[\int_{l}^{\infty} f_S(s)d_s\right] dl \quad (5)$$

$$POF = \int_{-\infty}^{\infty} f_L(l)\, F_S(l)dl \quad (6)$$

where $F_S(l)$ is the probability distribution function (PDF^*) of S in the unit of load.

In this project, we will be focusing on using Python as a tool to find the probability of failure for offshore structure.

Analyzing the Data

Before we proceed to calculate the probability of failure for offshore structure, we will need to analyze the given data.

Knowing the Data

In this project we will be handling the raw data which consists of Base shear(BS), Reserve Strength Ratio for 100 years (RSR 100).

- Base shear(BS): An estimate of the maximum expected lateral force on the base of the offshore structure due to seismic activity.
- Ultimate Base shear: The maximum allowable stress.
- Reserve Strength Ratio: The ratio of Ultimate base shear to base shear.

The raw data are given in an ***Excel*** file. Using Python, we are able to read the raw data using `pandas`, which is a package that enable us to read file like excel, csv and many more.

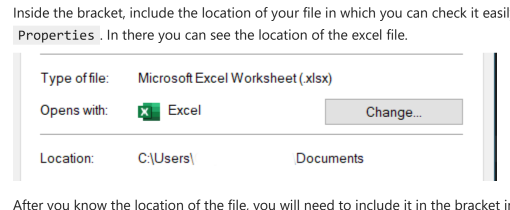
```
import pandas as pd
```

** We will name the package as **pd** therefore we don't have to type pandas everytime we want to use this package.

Next is to read the excel file and also giving it a name.

```
import pandas as pd
main_data= pd.ExcelFile()
```

Inside the bracket, include the location of your file in which you can check it easily by going to `Properties`. In there you can see the location of the excel file.



After you know the location of the file, you will need to include it in the bracket in order for pandas to read the file.

```
import pandas as pd
main_data= pd.ExcelFile(r'C:\Users\Documents\YOUR DATA NAME GOES HERE.xlsx')
```

** Notice the litte **r'** in the beginning of the bracket? That is a raw string. For more information on what is a raw string, there is a website that explains this [here](https://www.journaldev.com/23598/python-raw-string).

Since the excel file consists of large amount of data and several sheets, it is wise that we split it into a small part without having to deal with every data. The data that we are handling consists of 11 sheets and we are interested to view one of the sheet, therefore we will use `pandas` to read it.

```
import pandas as pd
main_data= pd.ExcelFile(r'C:\Users\Documents\YOUR DATA NAME GOES HERE.xlsx')
df = pd.read_excel(main_data, 'THE NAME OF THE SHEET GOES HERE', names= ['RSR 100',
'BS'])
```

We can change the index of the data to start from 1 instead of 0. Here we will need to use another handy package which is `numpy`.

```
import numpy as np
df.index= np.arange(1, len(df['BS'])+1)
```

After arranging the data, we still need to obtain the ultimate which is not given. From the definition above, we know that:

$$Reserve\ Strength\ Ratio\ (RSR) = \frac{Ultimate\ Base\ Shear}{Base\ Shear}$$

```
df['Ultimate']= df['RSR 100']* df['BS']
df
```

By using `df`, we can show the data that we are working with just now. You can also include `df.head()` to show only the top few data or `df.tail()` to show only the bottom few of the data.

The overall code should look like this:

```
import pandas as pd
import numpy as np
main_data= pd.ExcelFile(r'C:\Users\Documents\YOUR DATA NAME GOES HERE.xlsx')
df = pd.read_excel(main_data, 'THE NAME OF THE SHEET GOES HERE', names= ['RSR 100',
'BS'])
df.index= np.arange(1, len(df['BS'])+1)
df['Ultimate']= df['RSR 100']* df['BS']
df
```

	RSR 100	BS	Rank	Ultimate
1	0.099895	0.0	1	0.000000e+00
2	0.199762	0.0	2	0.000000e+00
3	0.299662	0.0	3	0.000000e+00
4	0.399523	0.0	4	0.000000e+00
5	0.499441	0.0	5	0.000000e+00
...	...	...	...	...
877	2.140170	5741750.0	877	1.228832e+07
878	2.140400	5742380.0	878	1.229099e+07
879	2.140580	5742840.0	879	1.229301e+07
880	2.193970	5886070.0	880	1.291386e+07
881	5.867950	15742700.0	881	9.237738e+07

The data that we will perform here is only limited to 64 deg, for other data set it would be the same by changing the file names only.

Goodness of Fit Test

In this section, we will discuss about how to perform goodness of fit test on the raw data. The purpose of this is to find out which distributions fits the data best.

Anderson- Darling Test

The Anderson-Darling test is a statistical test of whether a given sample of data is drawn from a given probability distribution. It is a modification of the Kolmogorov-Smirnov(K-S) test and gives more weight to the tails than does the K-S test.

The Anderson-Darling test statistics is defined as:

$$A^2 = -n - S$$

where,

$$S = \sum_{i=1}^{n} \frac{2i-1}{n}\left[\ln(F(Y_i)) + \ln(1 - F(Y_{n+1-i}))\right].$$

- ***F*** is the cumulative distribution function of the specified distribution.
- ***n*** is the sample size

Now we perform the goodness of fit test on the data that we have extracted from the excel file.

For this test, we will now use two additional package which are fitter and reliability. Both of these packages are useful when you are handling data needs to perform statistical test.

* Remember that you need to **install** the package via **anaconda prompt** in order to use it.

Continuing from the previous code,

```
import fitter

#Here we will sort the data before performing the test
#Since we will be handling BS and Ultimate so we will
#name data1= BS and data2= Ultimate
rank = df.index
data1 = np.sort(df['BS'].to_numpy())
data2 = np.sort(df['Ultimate'].to_numpy())
```

After we have sort our data, we will continue to use the package mentioned above. Fitter will help us to fit the data for some distributions while Reliability will help us to fit the data for 3-parameter distributions.

Below are the distributions that we will be performing test statistics:

Distribution
Normal
3-Parameter Lognormal
2-Parameter Exponential
3-Parameter Weibull
Smallest Extreme Value
Largest Extreme Value
3-Parameter Gamma
Logistic
3-Parameter Loglogistic

Since we will be handling 5 distributions at once when using reliability, it would be very messy if we were to type the code for one distribution then run it then repeat the same process over and over again. Therefore we will be using the `if`, `elif` and also `for` function.

- `if` : a conditional argument.
- `elif` : a conditional argument when the condition in `if` is not meet/ satisfy.
- `for` : a loop argument.

Here is how we use it in the code, again, continuing from the previous code,

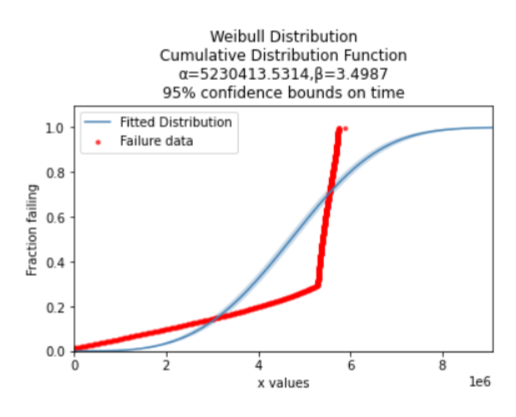
```
from reliability.Fitters import Fit_Weibull_3P, Fit_Exponential_2P,
Fit_Lognormal_3P, Fit_Loglogistic_3P, Fit_Gamma_3P
from reliability.Probability_plotting import plot_points
```

Now, we will put the distributions in like a "box" so we can use the `for` loop function.

```
distributions = ['Lognormal_3P','Exponential_2P', 'Weibull_3P', 'Loglogistic_3P',
'Gamma_3P']
for dist in distributions:
        if dist=='Lognormal_3P':
        lognorm_fit = Fit_Lognormal_3P(failures= data1, show_probability_plot=
False, print_results= True)
        lognorm_fit.distribution.CDF(label='Fitted Distribution', color='Steelblue')
        plot_points(failures=data1, func='CDF', label='Failure data', color='red',
alpha= 0.7)
        plt.legend()
        plt.show()

    elif dist=='Weibull_3P':
        weibull_fit = Fit_Weibull_3P(failures= data1, show_probability_plot= False,
print_results= True)
        weibull_fit.distribution.CDF(label='Fitted Distribution', color='Steelblue')
        plot_points(failures=data1, func='CDF', label='Failure data', color='red',
alpha= 0.7)
        plt.legend()
        plt.show()

    elif dist=='Loglogistic_3P':
        loglog_fit= Fit_Loglogistic_3P(failures= data1, show_probability_plot=
False, print_results= True)
        loglog_fit.distribution.CDF(label='Fitted Distribution', color='Steelblue')
        plot_points(failures=data1, func='CDF', label='Failure data', color='red',
alpha= 0.7)
        plt.legend()
        plt.show()

    elif dist=='Gamma_3P':
        gamma_fit = Fit_Gamma_3P(failures= data1, show_probability_plot= False,
print_results= True)
        gamma_fit.distribution.CDF(label='Fitted Distribution', color='Steelblue')
        plot_points(failures=data1, func='CDF', label='Failure data', color='red',
alpha= 0.7)
        plt.legend()
        plt.show()
```

All the API reference are available on the Reliability website.

After you have run the code, the graph will apear and all the result will be available and also the Anderson-Darling test value.



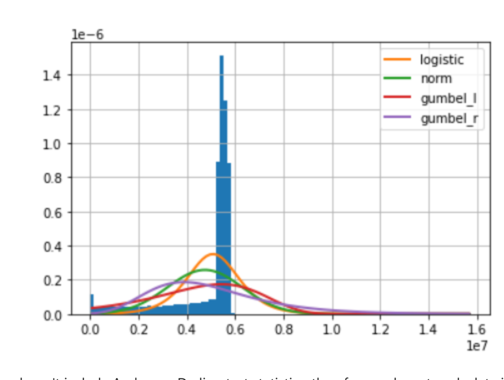

```
Goodness of fit    Value
 Log-likelihood -13858.3
           AICc  27722.7
            BIC    27737
             AD  136.788
```

After we have done the 3-parameter distributions, we will use **Fitter** to continue for the remaining distributions.

Continuing,

```
from fitter import Fitter
f1 = Fitter(data1, distributions =['norm', 'logistic', 'gumbel_l', 'gumbel_r'])
f1.fit()
f1.summary()
```

	sumsquare_error	aic	bic	kl_div
logistic	3.66925e-12	3972.94	-29158.2	inf
norm	4.14177e-12	4310	-29051.5	inf
gumbel_l	4.31517e-12	6215.27	-29015.3	inf
gumbel_r	4.59954e-12	3486.94	-28959.1	inf



Fitter doesn't include Anderson-Darling test statistics, therefore we have to calculate it manually. From the definition above, we know that:

```
s= (((2*rank-1))*(log_cdf+log_sf[::-1]))
```

where,

- cdf= Cumulative density function
- sf= survival function (1-CDF)

We can also obtain the parameter of the fitted distributions.

```
f.fitted_param[distributions]
```

We can place the parameters obtain above into an array so that it is easier for us to handle that piece of information that we obtained.

```
para= np.array(f.fitted_param[distributions])
```

After obtaining the parameters, we can fit it to the distributions that we desire.

```
cdf = norm.cdf(data, loc=para[0], scale=para[1])
log_cdf= norm.logcdf(data, loc=para[0], scale=para[1])
log_sf= norm.logsf(data, loc=para[0], scale=para[1])
```

After getting all the components done, we can include it back to the formula of the Anderson-Darling test. We can also print out the result.

```
s= (((2*rank-1))*(log_cdf+log_sf[::-1]))
print('\ns= ', sum(s)/len(data))
ad=-len(data)-sum(s)/len(data)
print("\n\033[1mAnderson-Darling test\033[0m= ",ad)

#This is a format for bold text in python coding.
print("\033[1m <YOUR TEXT GOES HERE>\033[0m=")
```

After you have done this, we can continue it for other distributions, which we can use `for` loop again.

The overall code will look like this:

```
from scipy.stats import norm, logistic, gumbel_l, gumbel_r
dist = ['norm', 'logistic', 'gumbel_l', 'gumbel_r']
for distributions in dist:
    para = np.array(f.fitted_param[distributions])
    print('\nFor\033[1m', distributions, '\033[0mdistribution, the parameters are=
', para[0], 'and', para[1])
    if distributions =='norm':
        cdf = norm.cdf(data1, loc=para[0], scale=para[1])
        log_cdf= norm.logcdf(data1, loc=para[0], scale=para[1])
        log_sf= norm.logsf(data1, loc=para[0], scale=para[1])
        s= (((2*rank-1))*(log_cdf+log_sf[::-1]))
        print('\ns= ', sum(s)/len(data1))
        ad=-len(data1)-sum(s)/len(data1)
        print("\n\033[1mAnderson-Darling test\033[0m= ",ad)

    elif distributions =='logistic':
        cdf = logistic.cdf(data1, loc=para[0], scale=para[1])
        log_cdf= logistic.logcdf(data1, loc=para[0], scale=para[1])
        log_sf= logistic.logsf(data1, loc=para[0], scale=para[1])
        s= (((2*rank-1))*(log_cdf+log_sf[::-1]))
        print('\ns= ', sum(s)/len(data1))
        ad=-len(data1)-sum(s)/len(data1)
        print("\n\033[1mAnderson-Darling test\033[0m=",ad)

    elif distributions =='gumbel_l':
        cdf = gumbel_l.cdf(data1, loc=para[0], scale=para[1])
        log_cdf= gumbel_l.logcdf(data1, loc=para[0], scale=para[1])
        log_sf= gumbel_l.logsf(data1, loc=para[0], scale=para[1])
        s= (((2*rank-1))*(log_cdf+log_sf[::-1]))
        print('\ns= ', sum(s)/len(data1))
        ad=-len(data1)-sum(s)/len(data1)
        print("\n\033[1mAnderson-Darling test\033[0m=",ad)

    elif distributions =='gumbel_r':
        cdf = gumbel_r.cdf(data1, loc=para[0], scale=para[1])
        log_cdf= gumbel_r.logcdf(data1, loc=para[0], scale=para[1])
        log_sf= gumbel_r.logsf(data1,loc=para[0], scale=para[1])
        s= (((2*rank-1))*(log_cdf+log_sf[::-1]))
        print('\ns= ', sum(s)/len(data1))
        ad=-len(data1)-sum(s)/len(data1)
        print("\n\033[1mAnderson-Darling test\033[0m=",ad)
```

Then we can repeat the whole code for data2 which is ultimate. You can go creative by including it in a loop and maybe go through fewer steps!

Construct a table

After you have done with the coding, it is time to place it in a table for better reference.

With the help of `pandas`, we can create a table and highlight the important values that we want to show.

```
new_data1={'Distributions':['Normal', 'Lognormal_3P', 'Exponential','Weibull_3P',
'Smallest Extreme Value','Largest Extreme Value','Gamma_3P', 'Logistic',
'Loglogistic_3P'],
          'Anderson-Darling test':['126.61618047143998','164.824',
'234.8252416965936', np.nan, '144.17320597035314', '139.51795347162397',np.nan,
'111.8059051356015', ' 136.788']}
df1= pd.DataFrame(new_data1)
df1.index=np.arange(1, len(df1)+1)
df1['Anderson-Darling test']=df1['Anderson-Darling test'].astype(float,
errors='raise')
#This is the code for using a hightlight in pandas
df1.style.highlight_min(color ='yellow', axis= 0).set_caption('Goodness of fit test
for BS')
```

Goodness of fit test for BS

	Distributions	Anderson-Darling test
1	Normal	126.616180
2	Lognormal_3P	164.824000
3	Exponential	234.825242
4	Weibull_3P	nan
5	Smallest Extreme Value	144.173206
6	Largest Extreme Value	139.517953
7	Gamma_3P	nan
8	Logistic	111.805905
9	Loglogistic_3P	136.788000

From the above test statistics, we know that **Logistic** distributions is the best fit for this data. The **NaN** in the table is null value. This is due to the fact that the value calculated by reliability is beyond it's calculation limit.(There are explanation on the website)

After obtaining the result in Python, we can compared it to Minitab to test the accuracy of the estimation.

Goodness of Fit Test

Distribution	AD	P
Normal	127.909	<0.005
3-Parameter Lognormal	127.413	*
2-Parameter Exponential	224.763	<0.010
3-Parameter Weibull	122.220	<0.005
Smallest Extreme Value	139.601	<0.010
Largest Extreme Value	139.751	<0.010
3-Parameter Gamma	146.659	*
Logistic	113.877	<0.005
3-Parameter Loglogistic	114.623	*

The value obtained here for 64 deg is similiar but for other data set would have some differences.

Now that we know what distribution is best fit for the data, we can proceed to plot graph and estimate the area under the overlapping distributions.

Plotting a bell curve graph

After getting the test statistics value, we can use a suitable distribution to plot the graph. To plot graph in Python, we will make use of two package that is useful, `matplotlib` and `seaborn`. Either one is good for graph plotting and their API reference can be found here.Matplotlib, seaborn

Before we start to plot the graph, we will need to gather the data to plot the graph. For graph plotting, we mention that the probability of failure can be expressed as the overlapping of PDF for two distributions which are PDF for BS and PDF for Ultimate.

Now we will calculate the PDF for the data. You can start in a new notebook or continue from the previous code. If continuing from the previous code, package that have been imported don't need to be import the second time.

For this code, we will be using a new notebook.

```
import pandas as pd
import numpy as np
import matplotlib.py as plt
import seaborn as sns
import scipy.stats as ss
```

After importing the required packages for graph plotting, we will calculate the PDF for each data set. Since we are using a new notebook here, we will need to do calculation for ultimate again.

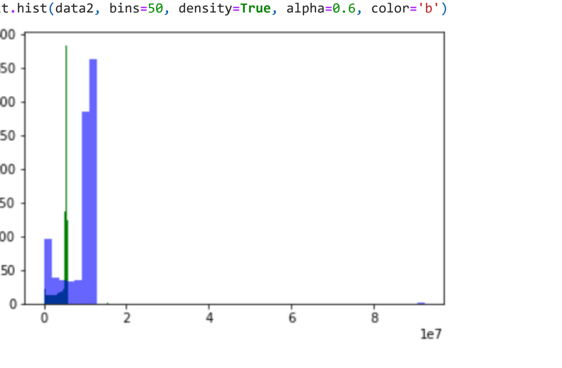
```
main_data= pd.ExcelFile(r'C:\Users\Ong Kang Chiew\Documents\All data.xlsx')
df= pd.read_excel(main_data, '64 deg' , names= ['RSR 100', 'BS'])
df.index=np.arange(1,len(df['BS'])+1)
df['Rank']=df.index
df['Ultimate']= df['RSR 100']* df['BS']
df

data1= np.sort(df['BS'].to_numpy())
data2=np.sort(df['Ultimate'].to_numpy())
```

The parameter for the distributions we can use `.fit()` to obtained it.

```
from scipy.stats import logistic
m1, std1= logistic.fit(data1)
m2, std2= logistic.fit(data2)
```

We can use `matplotlib` to plot the histogram of the data if required.

```
plt.hist(data1, bins=20, density= True, alpha=1, color='g')
plt.hist(data2, bins=50, density=True, alpha=0.6, color='b')
```



Next we will define the x-axis and y-axis of the graph.

```
x1= data1
x2= data2
y1= logistic.pdf(x1, m1, std1)
y2= logistic.pdf(x2, m2, std2)
```

Next we will plot the graph.

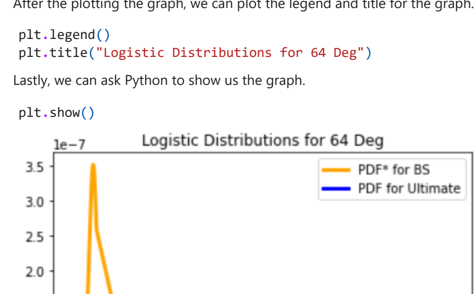
```
plt.plot(x1,y1, 'orange', lw='3', label='PDF* for BS')
plt.plot(x2,y2, 'b', lw='3', label='PDF for Ultimate')
```

After the plotting the graph, we can plot the legend and title for the graph.

```
plt.legend()
plt.title("Logistic Distributions for 64 Deg")
```

Lastly, we can ask Python to show us the graph.

```
plt.show()
```



Wait a minute, if we look at the graph , it is not a smooth bell curve. Don't worry, we can solve this by using `np.linspace`.

`numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)` [source]

> Return evenly spaced numbers over a specified interval.
>
> Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].
>
> The endpoint of the interval can optionally be excluded.

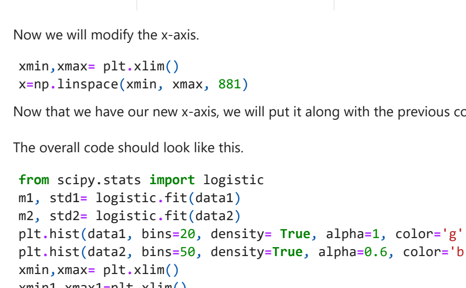
Now we will modify the x-axis.

```
xmin,xmax= plt.xlim()
x=np.linspace(xmin, xmax, 881)
```

Now that we have our new x-axis, we will put it along with the previous code.

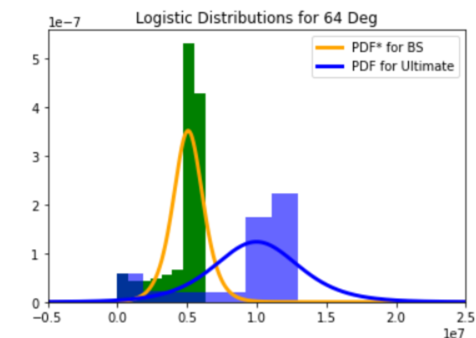
The overall code should look like this.

```
from scipy.stats import logistic
m1, std1= logistic.fit(data1)
m2, std2= logistic.fit(data2)
plt.hist(data1, bins=20, density= True, alpha=1, color='g')
plt.hist(data2, bins=50, density=True, alpha=0.6, color='b')
xmin,xmax= plt.xlim()
xmin1,xmax1=plt.xlim()
x=np.linspace(xmin, xmax, 881)
y1=logistic.pdf(x, m1, std1)
y2=logistic.pdf(x,m2,std2)
plt.plot(x,y1, 'orange', lw='3', label='PDF* for BS')
plt.plot(x,y2, 'b', lw='3', label='PDF for Ultimate')
plt.legend()
plt.title("Logistic Distributions for 64 Deg")
plt.show()
```



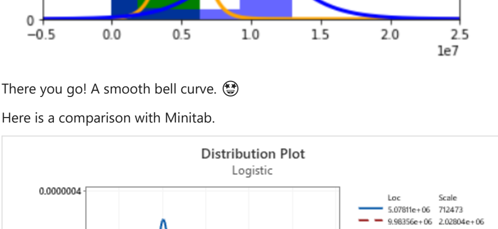
We can't really see the graph closely, so we can change the `x.lim()` of the graph to suit it for the data.

```
from scipy.stats import logistic
m1, std1= logistic.fit(data1)
m2, std2= logistic.fit(data2)
plt.hist(data1, bins=20, density= True, alpha=1, color='g')
plt.hist(data2, bins=50, density=True, alpha=0.6, color='b')
# Here we can change the x limit.
xmin,xmax= plt.xlim([-0.5e+07,0.25e+08 ])
xmin1,xmax1=plt.xlim([-0.5e+07,0.25e+08 ])
x=np.linspace(xmin, xmax, 881)
y1=logistic.pdf(x, m1, std1)
y2=logistic.pdf(x,m2,std2)
plt.plot(x,y1, 'orange', lw='3', label='PDF* for BS')
plt.plot(x,y2, 'b', lw='3', label='PDF for Ultimate')
plt.legend()
plt.title("Logistic Distributions for 64 Deg")
plt.show()
```



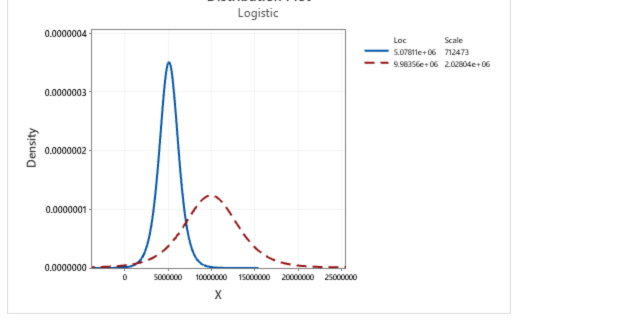
There you go! A smooth bell curve. 😍

Here is a comparison with Minitab.



Area Calculation (POF Calculation)

Finally we are in the last section of this Python tutorial. We will be calculating the area of the two overlapping distributions.



For this we will use the `def` class to define a function to help us find the intersecting points of the two distributions.

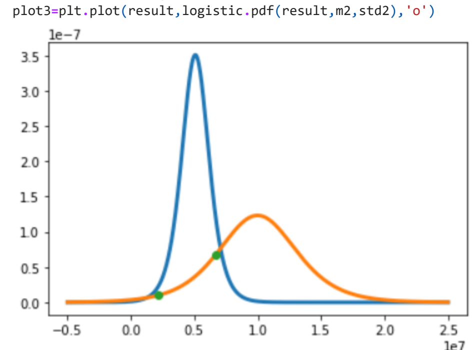
```
def solve(m1,m2,std1,std2):
    a = 1./(2.*std1**2) - 1./(2.*std2**2)
    b = m2/(std2**2) - m1/(std1**2)
    c = m1**2 /(2*std1**2) - m2**2 / (2*std2**2) - np.log(std2/std1)
    return np.roots([a,b,c])
```

Next, we will input our mean and standard deviations for the result to work. Mean and standard deviations are also the parameter for the distributions that we are using here.

```
result = solve(m1,m2,std1,std2)
print(result)
```

We can also show the intersection point on the graph.

```
x = np.linspace(xmin,xmax,881)
plot1=plt.plot(x,logistic.pdf(x,m1,std1), lw='3')
plot2=plt.plot(x,logistic.pdf(x,m2,std2), lw='3')
plot3=plt.plot(result,logistic.pdf(result,m2,std2),'o')
```



For the calculation, we can use two method.

Method 1

Using the cdf of the second distribution (orange), then add with 1-CDF for first distributions (blue).

```
overlap= logistic.cdf(result.max(), m2, std2)+ (1-logistic.cdf(result.max(), m1, std1))
print(overlap)
```

Method 2

Using the pdf of both distributions between the intersection point and sum up the values.

```
possible_num= np.linspace(result.min(), result.max(), 8)
overlap_probability = sum(logistic.pdf(possible_num, m2, std2))+ sum(logistic.pdf(possible_num, m1,std1))
print(overlap_probability)
```