# APS360 Tutorial 0 - Python and Libraries

## Summary:

Welcome to the first Tutorial of APS360! This is a warm up to get you used to the programming environment used in the course, and also to help you review and renew your knowledge of Python and relevant Python libraries. The labs must be done **individually** and even though this tutorial is not for marks, it is recommended that you get into the practise of working alone.

By the end of this lab, you should be able to:

1. Setup and use Google Colab.
2. Write basic, object-oriented Python code.
3. Be able to perform matrix operations using `NumPy`.
4. Be able to plot using `matplotlib`.
5. Be able to load, process, and visualize data.
6. Be able to perform basic machine learning operations using `Scikit-learn`.

You will need to use NumPy, matplotlib and SciPy. Documentations for there libraries are provided:

- https://docs.scipy.org/doc/numpy/reference/
- https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot
- https://scikit-learn.org/stable/supervised_learning.html

You can also reference Python API documentations freely.

## How to submit PDF for Labs?

- Creating PDF directly from Colab would result in segmented plots and code. If this happens you will lose mark for those components.
- To avoid this you need to first create an HTML version of your notebook and then create the PDF based on the HTML file.

For that:

1. Download your notebook: File -> Download .ipynb
2. Click on the Files icon on the far left
3. Select & upload your .ipynb file you just downloaded, and then obtain its path (right click) (you might need to hit the Refresh button before your file shows up)
4. Execute in a Colab cell (**There should not be any SPACEs in your file name**):

```
%%shell
jupyter nbconvert --to html "/PATH/TO/YOUR/NOTEBOOKFILE.ipynb"
```

5. Your notebook.html will appear in the files, so you can download it.
6. Open the HTML file with your browser (e.g., Chrome) and print the HTML file as an PDF.
7. Submit this PDF file on Quercus for grading.

Ref: https://stackoverflow.com/a/64487858

## Colab Link

Please make sure to include a link to your colab file here

1. Click on the share button on the top-right side of the screen.
2. On the new window, In the Get Link section,click on "Change to anyone with the link".
3. Copy the generated link and include it in your submissions.

Colab Link: N/A, for future labs you will be required to provide a link to your Colab file.

## LaTeX for Project Deliverables:

We suggest to take advantage of overleaf `link sharing` feature to work on your project deliverables.

1. Get the template form Quercus.
2. Upload the template on Overleaf.
3. Share the link with your team members.
4. If you are not familiar with LaTeX: https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes

- Cheat latex with Rich Text
- Look for youtube videos! example: https://www.youtube.com/watch?v=y8y_KIs9JLs

# Part 0

# Environmental Setup

Please refer to Colab instructions https://colab.research.google.com/drive/1YKHHLSlG-B9Ez2-zf-YFxXTVgfC_Aqtt

If you want to use Jupyter Notebook locally, please refer to
https://www.cs.toronto.edu/~lczhang/aps360_20191/files/install.pdf

# Part 1

# Problem: Predicting Titanic Survivors

## Problem Background

The sinking of the RMS Titanic is one of the most infamous shipwrecks in history. On April 15, 1912, during her maiden voyage, the Titanic sank after colliding with an iceberg, killing 1502 out of 2224 passengers and crew. This tragedy shocked the international community and led to better safety regulations for ships.

One of the reasons that the shipwreck led to such loss of life was that there were not enough lifeboats for the passengers and crew. Although there was some element of luck involved in surviving the sinking, some groups of people were more likely to survive than others, such as women, children, and the upper-class.

In 1997, the story of the Titanic was brought to the big screen and it allowed us to relive the story of two passengers, Jack and Rose, members of different social classes, who fall in love on the ill-fated passenger liner.

How many of you have seen the movie Titanic?

Most of you probably were not born when the movie came out, so in case you have not seen it yet, I'm not going to spoil it. In this lab we are going to work with a Titanic dataset to determine what sort of people were likely to survive the disaster. In particular we would like to predict if Jack and Rose would have survived. Afterwards, those of you who have not seen the movie can go and watch it to see if our predictions match the Hollywood story.

## Define the Problem

Our objective is to predict whether or not Jack and Rose would have survived the Titanic tragedy, based on what we know about them from the movie Titanic directed by James Cameron.

From the movie we can assume the following about Jack and Rose:

- Jack: 3rd class, no siblings, male, 25 years old, no cabin, fare = 7, embarked from Southhampton
- Rose: 1st class, no siblings, has spouse, 22 years old, cabin, fare = 50, embarked from Southhampton

To achieve this objective we are provided with historical data obtained after the Titanic tragedy. The historical data is provided as a CSV file containing information on 891 passengers as summarized below:

```
PASSENGER INFORMATION:
survival        Survival
                (0 = No; 1 = Yes)
pclass          Passenger Class
                (1 = 1st; 2 = 2nd; 3 = 3rd)
name            Name
sex             Sex
age             Age
sibsp           Number of Siblings/Spouses Aboard
parch           Number of Parents/Children Aboard
ticket          Ticket Number
fare            Passenger Fare
cabin           Cabin
embarked        Port of Embarkation
                (C = Cherbourg; Q = Queenstown; S = Southampton)
```

### Open Data using Spreadsheet Software

To solve this problem we will start off by investigating the data. We will also be looking at a couple modules namely, numpy, matplotlib and scipy to achieve our objective of predicting passenger survival outcome. To start off let's open the file `train.csv` which you can find on Quercus under Lab 0, or using the following link:

https://drive.google.com/open?id=1aOqkEx5mXBJ5u63NgJ_abRgzVs9awQh9

## Define Test Cases

To validate our predictions we will create test cases with male a female survivors and non-survivors.

### Test Case 1: Sample Male and Female Survivor

| Passenger | PassengerID | Pclass | Sex | Age | SibSp | Parch | Fare | Embarked_Val |
|---|---|---|---|---|---|---|---|---|
| Mrs. Laina Heikkinen | 3 | 3 | 1 | 26 | 0 | 0 | 7.925 | 2 |
| Master. Michel Navratil | 194 | 2 | 0 | 3 | 1 | 1 | 26 | 2 |

### Test Case 2: Sample Male and Female Non-Survivor

| Passenger | PassengerID | Pclass | Sex | Age | SibSp | Parch | Fare | Embarked_Val |
|---|---|---|---|---|---|---|---|---|
| Miss. Augusta Planke | 39 | 3 | 1 | 18 | 2 | 0 | 18 | 2 |
| Mr. Owen Harris Braund | 1 | 3 | 0 | 22 | 1 | 0 | 7.25 | 2 |

Once we are confident with the accuracy on accurately predicting survival on our test cases, we can then confidently predict what would happen to Jack and Rose.

### Test Case 3: Jack and Rose from the movie Titanic

| Passenger | Pclass | Sex | Age | SibSp | Parch | Fare | Embarked |
|---|---|---|---|---|---|---|---|
| Jack | 3 | male | 25 | 0 | 0 | 7 | S |
| Rose | 1 | female | 22 | 1 | 0 | 50 | S |

## Loading and Accessing Titanic Dataset

First let's review comma separated value (CSV) files. CSV files are simple text-based files well-suited for organizing spreadsheet data similar. In the CSV format all values are separated by a comma or some unique character. Using the Python csv module we can load our dataset:

In [1]:
```
# # load train.csv to Google Colab
# from google.colab import files
# uploaded = files.upload()
```

In [2]:
```
import csv
with open('./train.csv','r') as csvfile:
    data_reader = csv.reader(csvfile)

    data_orig = []
    for row in data_reader:
        data_orig.append(row)
```

(Optional) If you run into issues loading the file. You can also try the following code.

In [3]:
```
# # link to your Google Drive
# from google.colab import drive
# drive.mount('/content/drive')
```

In [4]:
```
# you will need to load the train.csv file to a folder on your Google Drive

#file_dir = '/content/drive/My Drive/_______/' #csv file location
# file_dir = "/content/drive/MyDrive/Colab Notebooks/APS360/Tut0/"
import csv
with open('./train.csv','r') as csvfile:
    data_reader = csv.reader(csvfile)

    data_orig = []
    for row in data_reader:
        data_orig.append(row)
```

Next we're going to look through our dataset to make sure it was loaded correctly

In [5]:
```
# display the full dataset
print(data_orig)
```

In [6]:
```
# display the first row (column titles)
print(data_orig[0])
```

```
['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp', 'Parch', 'Ticket', 'Fare', 'Cab
in', 'Embarked']
```

In [7]:
```
# display first two samples
print(data_orig[0:2])
```

```
[['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp', 'Parch', 'Ticket', 'Fare', 'Ca
bin', 'Embarked'], ['1', '0', '3', 'Braund, Mr. Owen Harris', 'male', '22', '1', '0', 'A/5 21171',
'7.25', '', 'S']]
```

In [8]:
```
# how would you display the last five samples?
```

In [9]:
```
# how would you display the first five odd samples?
```

In [10]:
```
# can you find the Master. Michel Navratil, Passenger ID 194?
```

## Numpy Module

Numpy provides support for working with multi-dimensional data such as our CSV file. In particular, it has a number of methods for efficient computation of linear algebra equations, provides capability for finding, extracting and/or changing information in multi-dimensional data, and allows for slicing of matrices simultaneously by column and row indices (i.e. using numpy in Python gives functionality similar to MATLAB).

We'll highlight some of these traits as we proceed with our data analysis.

To start we'll load our numpy module and convert our nested list into a numpy array

In [11]:
```
import numpy as np
data_numpy = np.array(data_orig)
```

data_numpy now holds all of the Titanic data.

In [12]:
```
# display numpy dataset
print(data_numpy)
```

```
[['PassengerId' 'Survived' 'Pclass' ... 'Fare' 'Cabin' 'Embarked']
 ['1' '0' '3' ... '7.25' '' 'S']
 ['2' '1' '1' ... '71.2833' 'C85' 'C']
 ...
 ['889' '0' '3' ... '23.45' '' 'S']
 ['890' '1' '1' ... '30' 'C148' 'C']
 ['891' '0' '3' ... '7.75' '' 'Q']]
```

Notice how numpy 2-dimensional data is printed across multiple rows rather than a continuous row as we've seen previously with nested lists.

Since there are a large number of samples, we cannot display all of them at the same time. Instead we can verify the structure of the data by displaying some of the samples at a time. Before we can do that we first need to understand how the numpy comma slicing notation works.

### Numpy Slicing

Slicing in Numpy is done differently from what we've seen so far. To highlight the differences we will compare numpy indexing and slicing of 1-dimensional and 2-dimensional data and compare it to what we did for lists.

#### 1-dimensional data: Indexing and Slicing

To index a list we use square brackets [], and to slice a list we would use a colon operator:

- list_variable[index]
- list_variable[start:end:step]

The same can be done for numpy:

- numpy_variable[index]
- numpy_variable[start:end:step]

In [13]:
```
list_variable = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

#list indexing
print(list_variable[2])

#list slicing
print(list_variable[1:8:2])
```

```
3
[2, 4, 6, 8]
```

In [14]:
```
numpy_variable = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

#numpy array indexing
print(numpy_variable[2])
#numpy array slicing
print(numpy_variable[1:8:2])
```

```
3
[2 4 6 8]
```

#### 2-dimensional data: Indexing and Slicing

To index a 2-dimensional list (nested list) we attach a second set of square brackets [][], however we are not able to slice a nested list by row and column simultaneously.

- list_variable[index1][index2]
- list_variable[start:end:step][start:end:step] -> does something completely different

To index a 2-dimensional numpy array we use the comma notation, which unlike with nested lists, allows us slice a numpy array simultaneously by column and row.

- numpy_variable[index1,index2]
- numpy_variable[start:end:step, start:end:step]

In [15]:
```
list_variable = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]

# nested list indexing
print(list_variable[2][0])

# nested list slicing
print(list_variable[0:2][0]) # creates a list of the first two rows, then gets the first element
```

```
9
[1, 2, 3, 4]
```

In [16]:
```
numpy_variable = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

# 2-d numpy array indexing
print(numpy_variable[2, 0])

# 2-d numpy array slicing
# get the first two entries in the first column
print(numpy_variable[0:2, 0])
```

```
9
[1 5]
```

In [17]:
```
numpy_variable
```

Out[17]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Now we can go about verifying the format of the data by displaying some of the samples at a time.

In [18]:
```
# select first row
print(data_numpy[0,:])
print(data_numpy[0])
```

```
['PassengerId' 'Survived' 'Pclass' 'Name' 'Sex' 'Age' 'SibSp' 'Parch'
 'Ticket' 'Fare' 'Cabin' 'Embarked']
['PassengerId' 'Survived' 'Pclass' 'Name' 'Sex' 'Age' 'SibSp' 'Parch'
 'Ticket' 'Fare' 'Cabin' 'Embarked']
```

In [19]:
```
# select first column
print(data_numpy[:,0])
```

In [20]:
```
# select first five columns and rows
print(data_numpy[:5,:5])
```

```
[['PassengerId' 'Survived' 'Pclass' 'Name' 'Sex']
 ['1' '0' '3' 'Braund, Mr. Owen Harris' 'male']
 ['2' '1' '1' 'Cumings, Mrs. John Bradley (Florence Briggs Thayer)'
  'female']
 ['3' '1' '3' 'Heikkinen, Miss. Laina' 'female']
 ['4' '1' '1' 'Futrelle, Mrs. Jacques Heath (Lily May Peel)' 'female']]
```

For our purposes it is not necessary to transpose the matrix as numpy allows for slicing columns and rows simultaneously.

If we did need to apply a transform, it can be done relatively easily using the numpy transpose methods as shown:

In [21]:
```
print(data_numpy.transpose())
```

```
[['PassengerId' '1' '2' ... '889' '890' '891']
 ['Survived' '0' '1' ... '0' '1' '0']
 ['Pclass' '3' '1' ... '3' '1' '3']
 ...
 ['Fare' '7.25' '71.2833' ... '23.45' '30' '7.75']
 ['Cabin' '' 'C85' ... '' 'C148' '']
 ['Embarked' 'S' 'C' ... 'S' 'C' 'Q']]
```

## Back to Predicting Passenger Survival

Let's get back to our original goal which is to predict whether Rose and Jack survive the Titanic disaster.

In order to do that we will need to examine the dataset in more detail. For example, perhaps there is a correlation between survival likelihood and particular passenger information, such as gender, class, age, etc.

As a first step, we can try to determine how many males and females survived and how many did not. How can we do that?

- one approach is to iterate through the passengers and create 4 accumulator variables to keep the counts:
  1. males_survived
  2. males_not_survived
  3. females_survived
  4. females_not_survived

Hmmm... What if we wanted to compare the survival across classes. There are three classes and two states of survival, hence we would need 6 accumulator variables to capture all the combinations. Certainly there has to be a better way to do this than writing code for each situation.

- Another option might be to keep track of indices of a particular feature, along with the indices of survival states. Once we have the indicies, we can find the indicies common to both lists and get our counts that way.

Turns out this can be done relatively easily with numpy if we know which method to use.

### Obtaining Indices using Numpy

Finding indices of specific values or range of values can be done using the np.where() method.

numpy.where(condition[, x, y])

- Return elements, either from x or y, depending on condition.
- If only condition is given, return indices where condition is True.

Since we're only interested in obtaining indices, we'll only provide a one argument to the method which will return two arrays (of the same size) corresponding to the row and column indices where the condition is true. For example:

In [22]:
```
numpy_data = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print('numpy_data:\n', numpy_data, '\n')

# obtain indicies with values > 7
indices = np.where(numpy_data > 7)

# display row and column indices
print('all indices:', indices, '\n')

# display row indices
print('row indices:', indices[0], '\n')

# display column indices
print('col indices:', indices[1], '\n')
```

```
numpy_data:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

all indices: (array([1, 2, 2, 2, 2], dtype=int64), array([3, 0, 1, 2, 3], dtype=int64))

row indices: [1 2 2 2 2]

col indices: [3 0 1 2 3]
```

In [23]:
```
numpy_data[numpy_data>7]
```

Out[23]:
```
array([ 8,  9, 10, 11, 12])
```

Note that the entries that are greater than 7 are: (1,3), (2,0), (2,1), (2,2), (2,2), and (2,3).

Let's apply this now to find indices in our data.

In [24]:
```
# obtain index of column with title 'Sex'
sex_index = np.where(data_numpy[0,:] == 'Sex')

# print indices
print(sex_index)
print(sex_index[0])

# print value at index found
print(data_numpy[0,sex_index[0]])
```

```
(array([4], dtype=int64),)
[4]
['Sex']
```

In [25]:
```
data_numpy[:,sex_index]
```

Out[25]:

In [26]:
```
# get indices of all male passengers
indices_male = np.where(data_numpy[1:,sex_index[0][0]] == 'male')
print(indices_male[0])

# display the number of male passengers
print(len(indices_male[0]))

# compute percentage of male passengers
percent = 100*len(indices_male[0])/len(data_numpy[:,sex_index])
print(round(percent,2), '% male passengers')
```

In [27]:
```
# what is the percentage of female passengers?
```

Let us now use the numpy module to calculate the percentage of males and females that survived. To start we need to find the indices of the survivors, then we can move on to find the indices of the male and female passengers.

Hmmm, which column was "Survived" again?

In [28]:
```
# obtain index of column with title 'Survived'
survived_index = np.where(data_numpy[0,:] == 'Survived')
print(survived_index[0])
```

```
[1]
```

To make it easier to search by field name, we can create a dictionary for easy indexing.

In [29]:
```
# Loop through field names and populate a dictionary with indices
fields = {}

# cycle through the first row (i.e. fields)
for i in range(len(data_numpy[0,:])):
    fields[data_numpy[0, i]] = i

print(fields)
```

```
{'PassengerId': 0, 'Survived': 1, 'Pclass': 2, 'Name': 3, 'Sex': 4, 'Age': 5, 'SibSp': 6, 'Parch':
7, 'Ticket': 8, 'Fare': 9, 'Cabin': 10, 'Embarked': 11}
```

In [30]:
```
fields['Age']
```

Out[30]:
```
5
```

Now we can use the dictionary to quickly obtain the index of the filed we are intersted in searching.

Let's find the percentage of male passengers that survived

In [31]:
```
# get indices for male passengers
field1 = 'Sex'
field1_val = 'male'
male_indices = np.where(data_numpy[0:,fields[field1]] == field1_val)
male_indices = list(male_indices[0])

# get indices for surviving passengers
field2 = 'Survived'
survived_indices = np.where(data_numpy[0:,fields[field2]] == '1')
survived_indices = list(survived_indices[0])
```

Now that we have a list of indices of passengers that survived, and a separate list of indices for the ones that are males, how can we use that information to find the number of male survivors?

There are a couple ways we could do this. One option is to convert our lists of indices into sets and take advantage of the set intersection method/operator (i.e. &).

In [32]:
```
# compute percentage that survived
percent = 100*len(set(male_indices) & set(survived_indices))/len(male_indices)
print(round(percent,2), '% of', field1_val, 'passengers survived')
```

```
18.89 % of male passengers survived
```

We could do the same thing to find the percentage of female passengers that survived or even to find the percentage of first class, female passengeres who survived. But doing this would seem like a lot of code for each combination of characteristics. Why not write a function that generalizes?

We are given some number characteristics (A, B, C) (e.g., A could be male, B could be first class, etc) and we want to find out the percentage of passengers with all of those characteristics that survived. The little algorithm we could write is:

- find ind_A (the indices with characteristic A), ind_B, and ind_C and intersect them to form ind_characteristics
- find ind_survived (the indices of all passengers who survived)
- the length of ind_survived intersected with ind_characteristics divided by the length of ind_survived gives the proportion of surivors

Since we want to be able to do this with any number of characteristics, lets put them in a list.

In [33]:
```
def get_survival(characteristics):
    """Return the percentage of passengers with the (field, value) entries in
       characteristics that survived.
       characteristics is a list of the form [(field, value), (field, value), ...]
    """
    indices = set()
    for i in range(len(characteristics)):
        # get search category
        field = characteristics[i][0]

        # get value to search for
        val = characteristics[i][1]

        # find the matching indices
        new_indices = set(list(np.where(data_numpy[0:,fields[field]] == val)[0]))

        # intersect
        if len(indices) == 0:
            indices = new_indices
        else:
            indices &= new_indices

    # find the indices of the survivors
    indices_survived = set(list(np.where(data_numpy[0:,fields["Survived"]] == "1")[0]))

    return 100*len(indices_survived & indices)/len(indices)

percent = get_survival([("Sex", "male")])
print(round(percent,2), '% of male passengers survived')
```

```
18.89 % of male passengers survived
```

Now we can easily do the same thing for other combinations.

In [34]:
```
# find the percentage of female passengers that survived
```

How about we combine gender and class to see how many first class males survived compared to other classes. How could we do that?

In [35]:
```
# find the percentage of  male class 1 passengers that survived
```

In [36]:
```
# find the percentage of  female class 1 passengers that survived
```

Let's now use the passenger gender and class to improve our prediction.

### Test Case 1: Sample Male and Female Survivor

| Passenger | PassengerID | Pclass | Sex | Age | SibSp | Parch | Fare | Embarked_Val |
|---|---|---|---|---|---|---|---|---|
| Mrs. Laina Heikkinen | 3 | 3 | 1 | 26 | 0 | 0 | 7.925 | 2 |
| Master. Michel Navratil | 194 | 2 | 0 | 3 | 1 | 1 | 26 | 2 |

### Test Case 2: Sample Male and Female Non-Survivor

| Passenger | PassengerID | Pclass | Sex | Age | SibSp | Parch | Fare | Embarked_Val |
|---|---|---|---|---|---|---|---|---|
| Miss. Augusta Planke | 39 | 3 | 1 | 18 | 2 | 0 | 18 | 2 |
| Mr. Owen Harris Braund | 1 | 3 | 0 | 22 | 1 | 0 | 7.25 | 2 |

In [37]:
```
# Estimate survival likelihood of the test case passengers and comment on the accuracy of the finding
```

# Part 2

# Problem: Visualizing Titanic Dataset

This part will focus on visualizing the data to find patterns that may allow us to make a prediction on who would survive the Titanic tragedy.

Python has many modules available dealing with visualization. One of the most popular to use is the **matplotlib module** which replicates the plotting capability of MATLAB (matplotlib is short for "MATLAB plotting library). In what is to follow, we will discuss how to import and use this module.

To start we can use the plot() method, which takes an optional format string argument that specifies the color and style of the plotted line. For example, plot(x_values, y_values, 'r--') uses 'r' to specify a red color, and '--' to specify a dashed line. You can find more information on formatting options at the following link.

In [38]:
```
import matplotlib.pyplot as plt
```

The program imports the pyplot module from the matplotlib package, renaming matplotlib.pyplot to plt using the **as** keyword.

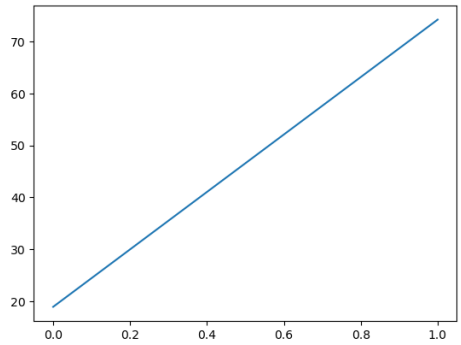
The plt.plot() function plots data onto the graph. plot() accepts various arguments. If provided just one list, as in plt.plot(val), plot() uses 0, 1, ... for x values, as in (0, val[0]), (1, val[1]), etc.

plt.plot() on its own will not display anyting. One needs to call the plt.show() function to displays the graph.

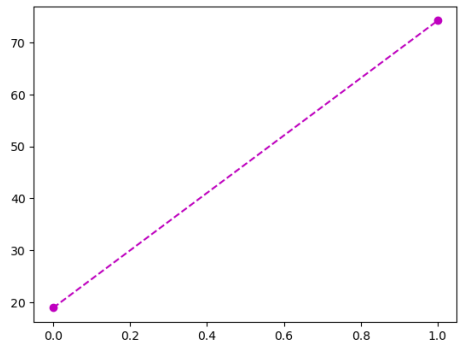
To start let's plot the survival percentages based on gender:

In [39]:
```
# plot survival by gender
male_survived = 18.9
female_survived = 74.2
survived_percent = [male_survived, female_survived]

# plot survival percentages
```

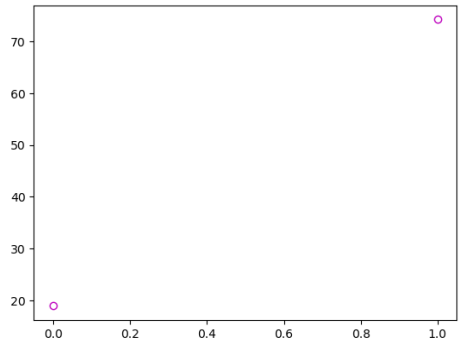
```
plt.plot(survived_percent)
plt.show()
```



In [40]:
```
# plot survival with dotted connecting lines
plt.plot(survived_percent, 'm--o')
plt.show()
```

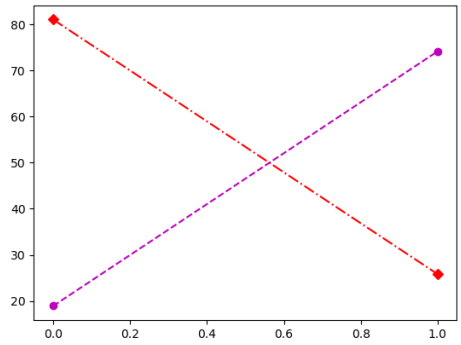


In [41]:
```
# plot survival without connecting lines
plt.plot(survived_percent, 'mo', markerfacecolor = 'None')
plt.show()
```



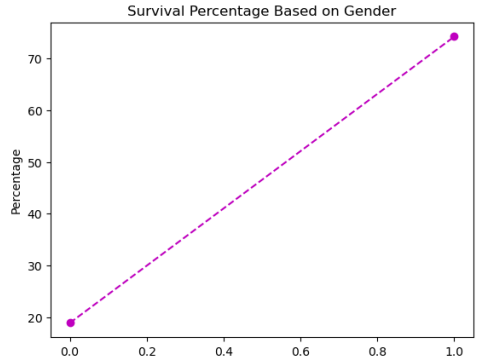
Calling plot multiple times draws multiple lines.

In [42]:
```
# we can plot percentages of those that survived with overlapping percentages
# of those that did not survive
survived_percent = [male_survived, female_survived]
n_survived_percent = [100 - male_survived, 100 - female_survived]

plt.plot(survived_percent, 'm--o')
plt.plot(n_survived_percent, 'r-.D')

plt.show()
```

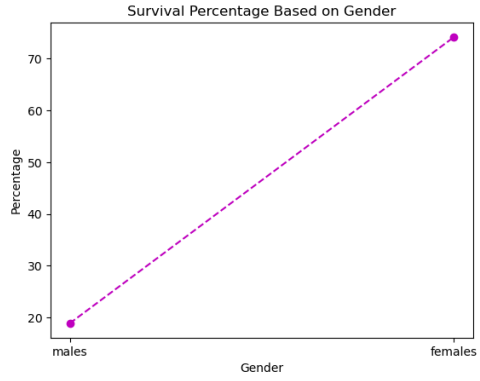


## Text and Annotations

In [43]:
```
# add title and axis labels
plt.plot(survived_percent, 'm--o')
plt.title('Survival Percentage Based on Gender')
plt.ylabel('Percentage')
plt.show()
```




In [44]:
```
# add tick labels
plt.plot(survived_percent, 'm--o')
plt.xticks([0, 1],['males', 'females'])
plt.title('Survival Percentage Based on Gender')
plt.xlabel('Gender')
plt.ylabel('Percentage')
plt.show()
```

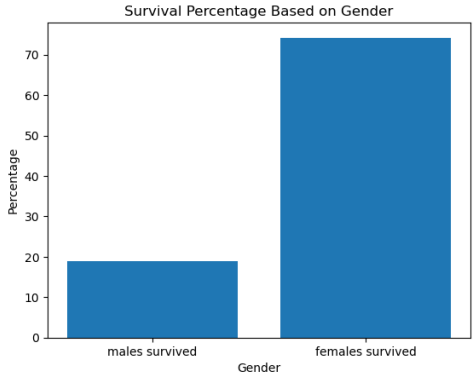



Hmmmm, this information would be best represented as a bar graph. Turns out we can do that as well using matlibplot.

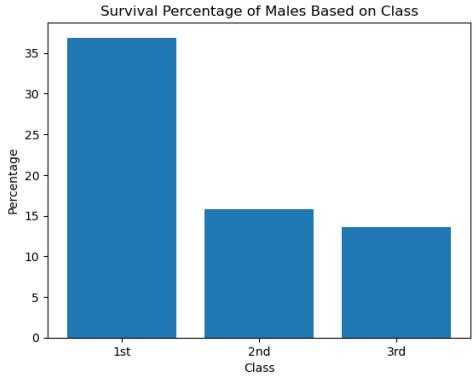
### Bar Graphs - Averaged Data

We can visualize the survival rates using bar graphs as shown:

In [45]:
```
# plot bar graph of survival by gender
pos = [0, 1]
plt.bar(pos, survived_percent, align = 'center')
plt.xticks(pos,['males survived', 'females survived'])
plt.title('Survival Percentage Based on Gender')
plt.xlabel('Gender')
plt.ylabel('Percentage')
plt.show()
```




In [46]:
```
# plot bar graph of survival by class
male_class_survived = [36.88, 15.74, 13.54]


# x axis position of bars graph
pos = range(len(male_class_survived))
# generate bar graph
plt.bar(pos, male_class_survived, align = 'center')
# provide labels for each bar based on provided positions
plt.xticks(pos,['1st', '2nd', '3rd'])
plt.title('Survival Percentage of Males Based on Class')
plt.xlabel('Class')
plt.ylabel('Percentage')
plt.show()
```




**(optional) We can also use subplots to plot everything together. In your spare time see if you can plot the percentage of male and female survivors using subplots.**

In [47]:
```
# (optional) use subplots to show male and female survivors by class
```

What if we wanted to plot an entire column? For example, we could print the ages of all the passengers.

### Numpy: Converting Column of str to int

When we loaded our data all the values were converted into strings because unlike lists numpy arrays can be of only one type (i.e. string or float, not both). This is somewhat problematic as we cannot plot strings, we need numerical values. We need to fix our dataset before we can plot it. How might we do that?

If we want to plot the age of passengers we need to apply the following steps:

1. Select the age column
2. find all the missing ages and replace them with a numerical value (i.e. np.nan)
3. convert the numpy array into a float

Selecting the age column is something we've done, but what about finding and overwriting data? One option could be to find the indicies that are blank and overwrite them. There is an easier way to do this.

Turns out we can also select numpy data by value using conditionals. For examples:

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

x[x >= 5]
```

Will select only the data that is greater than or equal to 5 in a numpy array x.

We can take this further by assigning that data a particular value.

```
x[x >= 5] = 100
```

Will select only the data greater than or equal to 5 and change the value to 100.

In [48]:
```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
print(x[x >= 5])
x[x >= 5] = 100
print(x)
```

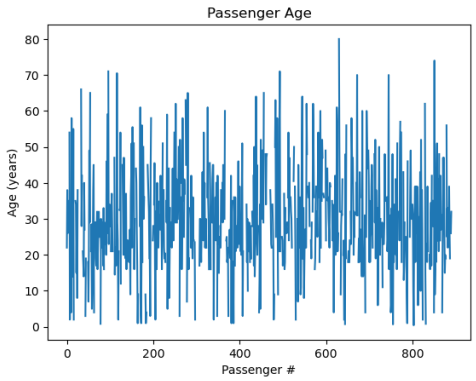
```
[5 6 7 8 9]
[  1   2   3   4 100 100 100 100 100]
```

Now we can use the same technique to update our age data to numerical values.

In [49]:
```
# select the age of passengers
index = fields['Age']
age = data_numpy[1:, index]

# find the ones that are empty and make them nan
age[age == ''] = np.nan

# convert all the strings into floats
age = age.astype(float)

# verify conversion to float
print(age)
```

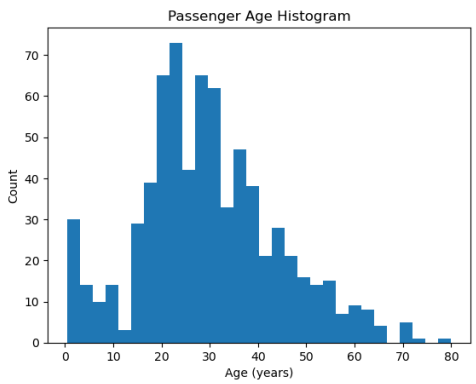
By making the missing data of type nan (not a number), plot will ignore those values when plotting age values.

In [50]:
```
# plot age
plt.plot(age)
plt.title('Passenger Age')
plt.xlabel('Passenger #')
plt.ylabel('Age (years)')
plt.show()
```




Since we don't care about the sequence of the age of passengers, it may be more informative to see how many passengers we have within each age group, i.e. plot a histogram of our data.

In [51]:
```
plt.hist(age, bins=30)
plt.title('Passenger Age Histogram')
plt.xlabel('Age (years)')
plt.ylabel('Count')
plt.show()
```



Hmmmm, seems like we obtain a histogram for nan values. How can we fix this? Let's search on Google to see if anyone else has had this problem. A quick search reveals the following stackoverflow discussion (link).

```
x = x[numpy.logical_not(numpy.isnan(x))]
```

or

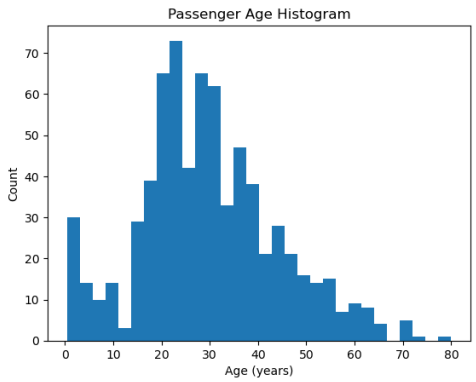
```
x = x[~numpy.isnan(x)]
```

This is actually the same as:

```
x = x[numpy.isnan(x) == 0]
```

Let's use this to select all age values excluding the nan type.

In [52]:
```
# plot a histogram of passenger ages excluding nan values
# to plot the histogram we need to exclude the missing ages, i.e. nan values
# the np.isnan() method returns True for values that are of type nan
plt.hist(age[np.isnan(age) == 0], bins=30)
plt.title('Passenger Age Histogram')
plt.xlabel('Age (years)')
plt.ylabel('Count')
plt.show()
```




**(Optional) As another exploration activity, it might be useful to see how survival changes with age. To do that it might be helpful to plot the survivor and non-survivor age histograms overtop of each other. Hmmm, how might we do that?** (Hint)

In [53]:
```
# (Optional) plot a histogram of passenger ages excluding nan values and overlap survivors and non-su
```

Will select only the data that is greater than or equal to 5 in a numpy array x.

We can take this further by assigning that data a particular value. For example:

```
x[x >= 5] = 100
```

Will select only the data greater than or equal to 5 and change the value to 100.

In [54]:
```
# any value >= 5 will be replaced with 100
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
x[x >= 5] = 100
print(x)
```

```
[  1   2   3   4 100 100 100 100 100 100 100 100]
```

Now let's use that to fix our data to only hold numerical values.

Let's put everything together into a class data structre and test. **Make sure that the trian.csv file is loaded.**

In [55]:
```
# Helper Titanic_Data Class

# Correct values in Survival, Gender, Embarked, and Age columns
import csv
import numpy as np

class Titanic_Data:
    """Titanic data set"""

    def __init__(self, Filename):
        """load Titanic data set"""
        with open(Filename,'r') as csvfile:
            data_reader = csv.reader(csvfile)
            data_orig = []
            for row in data_reader:
                data_orig.append(row)

        # Loop through field names and populate a dictionary with indices
        fields = {}
        for i in range(len(data_orig[0])):
            fields[data_orig[0][i]] = i

        # exclude the first row when preparing the numpy data structure
        self.data = np.array(data_orig[1:])
        self.fields = fields

    def get_survival(self, characteristics):
        """Return the percentage of passengers with the (field, value) entries in
           characteristics that survived.
           characteristics is a list of the form [(field, value), (field, value), ...]
        """
        indices = set()
        for i in range(len(characteristics)):
            # obtain search category
            field = characteristics[i][0]

            # obtain value to search for
            val = characteristics[i][1]

            # find and intersect the matching indices
            new_indices = set(list(np.where(self.data[0:,self.fields[field]] == val)[0]))

            # intersect
            if len(indices) == 0:
                indices = new_indices
            else:
                indices &= new_indices

        # find the indices of the survivors
        indices_survived = set(list(np.where(self.data[0:,self.fields["Survived"]] == "1")[0]))

        return 100*len(indices_survived & indices)/len(indices)

    def clean_data(self):
        """Converts all data into numerical values
        (missing data is converted into nan)"""
        self.clean('Sex', ['male', 'female'])
        self.clean('Embarked', ['C', 'Q', 'S'])
        self.clean('Age')
        self.clean('Pclass')
        self.clean('SibSp')
        self.clean('Parch')
        self.clean('Fare')

    def clean(self, col_header, values = []):
        """Converts column data into numerical values
        (missing data is convereted into nan)"""
        # select the column
        column = self.data[:,self.fields[col_header]]
        # find the ones that are empty and make them nan
        column[column == ''] = np.nan
        # encode the the strings as numbers
        for i in range(len(values)):
            column[column == values[i]] = i
        # overwrite
        self.data[:,self.fields[col_header]] = column

    def keep_columns(self, L):
        """Select Features """
        feature_data = self.data[:,L]
        feature_data = feature_data.astype(float)
        return feature_data

# call function to prepare data structure
titanic = Titanic_Data('train.csv')
print(titanic.data[0,:])

# cleaned data
titanic.clean_data()
print(titanic.data[0,:])

# remove unnecessary columns and convert array to float
feature_data = titanic.keep_columns([1, 2, 4, 5, 6, 7, 9, 11])
print(feature_data[0,:])
```

```
['1' '0' '3' 'Braund, Mr. Owen Harris' 'male' '22' '1' '0' 'A/5 21171'
 '7.25' '' 'S']
['1' '0' '3' 'Braund, Mr. Owen Harris' '0' '22' '1' '0' 'A/5 21171' '7.25'
 '' '2']
[ 0.    3.    0.   22.    1.    0.    7.25  2.  ]
```

Including age with the class and gender may benefit our predictions on the test cases. It would take some work to implement this change. It may be a good idea to pause and reconsider our approach continuing any further.

In the next part we will discuss how to take the information we have gained about our dataset to apply commonly used machine learning algorithms to predict passenger survival outcome.

# Part 3

# Problem: Predicting Titanic Survivors

After visualizing the data in the previous part, we can see that the odds of surviving vary depending on passenger information such as: age, sex, class. Is there some way we can use all our passenger information to predict survival?

In this part we will take a look at how to use the readily available machine learning algorithms to make predictions on survival.

The machine learning community has grown substantially over the years and there are many modules available to implement the different algorithms. To implement the algorithms, all we need to do is obtain a dataset and arrange it to follow the machine learning conventions which can be described by the following algorithm plan:

## End-to-End Machine Learning Project:

- Look at the big picture
- Get the data
- Visualize the data to gain insights
- Preprocess (clean) data to ensure all values are numerical
- Divide dataset into a training and testing set
- Select machine learning model and train it
- Perform prediction on the testing data
- Evaluate prediction performance of machine learning algorithm (use new data or holdout set to be sure of performance)

Note that we've already completed steps 1 - 3 and we've already done a bit of work on step 4 when we ploted age of passengers.

### Replace strings and missing values with numbers

As part of the data structure we require that all the samples are numerical. We need to prepare data by removing strings and filling in missing values.

As we saw previously, we can select numpy data by value using conditionals. For examples:

```
x[numpy.isnan(x) == 0]
```

Will select only the data that is not of type nan. Similarly,

```
x[x >= 5]
```

### Divide data into a training and testing dataset

In [56]:
```
feature_data = titanic.keep_columns([1, 2, 4, 5, 6, 7, 9, 11])

# replace all nan values with -1 to prevent an error
feature_data[np.isnan(feature_data)==1] = -1

# segment data into training and testing datasets (features)
halfway = len(feature_data)//2
```

```
training_data = feature_data[0:halfway,1:]
testing_data = feature_data[halfway:,1:]

# segment data into training and testing labels (targets)
training_labels = feature_data[0:halfway,0]
testing_labels = feature_data[halfway:,0]
```

In [57]:
```
# verify the training data and labels are correct
print(training_data)
print(training_labels)
```

### Import a Machine Learning Algorithm

There are many machine learning algorithms developed for making predictions (classification) similar to the one we are trying to do in this design problem. To use these algorithms we first need to import them from the scikit-learn machine learning modules for Python. More information on the different algorithms can be found at the following link.

There are many machine algorithms to choose from such as:

1. Decision Trees
2. Random Forests
3. Support Vector Machines
4. Naïve Bayes

For now let us focus on the decision trees classifier. We import the classifier and setup some default parameters.

In [58]:
```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(max_depth=5)
```

### Train machine learning algorithm

Next we train our algorithm on the training set. This updates the model parameters to make predictions specific to our data.

In [59]:
```
model.fit(training_data, training_labels)
```

Out[59]:
```
DecisionTreeClassifier(max_depth=5)
```

### Perform prediction on the test cases.

Now that model is trained, we can use it to predict the outcome on our test cases:

### Test Case 1: Sample Male and Female Survivor

| Passenger | PassengerID | Pclass | Sex | Age | SibSp | Parch | Fare | Embarked_Val |
|---|---|---|---|---|---|---|---|---|
| Mrs. Laina Heikkinen | 3 | 3 | 1 | 26 | 0 | 0 | 7.925 | 2 |
| Master. Michel Navratil | 194 | 2 | 0 | 3 | 1 | 1 | 26 | 2 |

### Test Case 2: Sample Male and Female Non-Survivor

| Passenger | PassengerID | Pclass | Sex | Age | SibSp | Parch | Fare | Embarked_Val |
|---|---|---|---|---|---|---|---|---|
| Miss. Augusta Planke | 39 | 3 | 1 | 18 | 2 | 0 | 18 | 2 |
| Mr. Owen Harris Braund | 1 | 3 | 0 | 22 | 1 | 0 | 7.25 | 2 |

We first need to select our testcase samples. We can do that by obtaining the samples from the original feature data by providing their order in the list (i.e. PassengerID).

In [60]:
```
# select test case 1 and 2 passengers
passenger_ids = np.array([3, 194, 39, 1])
print(passenger_ids-1)
testcase_data = feature_data[passenger_ids-1, 1:]

# display details on selected passengers
print(testcase_data)

# predict survival outcome
testcase_predict = model.predict(testcase_data)
print('Predicted Survival Outcome: ', testcase_predict)
```

```
[  2 193  38   0]
[[ 3.     1.    26.     0.     0.     7.925  2.   ]
 [ 2.     0.     3.     1.     1.    26.     2.   ]
 [ 3.     1.    18.     2.     0.    18.     2.   ]
 [ 3.     0.    22.     1.     0.     7.25   2.   ]]
Predicted Survival Outcome:  [1. 1. 0. 0.]
```

On the above sample we see that our algorithm was able to successfully predict the survival outcome of the passengers in the test cases.

Wait a second... it seems like all our test case samples are also in the training data. The algorithm was trained on the same data, which means they're going to do better on this data than on completely new data (i.e. testing_data).

Let's compare the prediction on the training_data and testing_data to get a proper validation of prediction performance.

In [61]:
```
# predict the first 10 training samples and compare to the actual survival outcomes
print('Training Sample Labels:     ', training_labels[0:10])
sample_predict = model.predict(training_data[0:10,:])
print('Predicted Survival Outcome: ', sample_predict)
```

```
Training Sample Labels:      [0. 1. 1. 1. 0. 0. 0. 0. 1. 1.]
Predicted Survival Outcome:  [0. 1. 1. 1. 0. 0. 0. 0. 1. 1.]
```

Seems like the prediction worked well on the training samples. How about the testing samples.

In [62]:
```
# test on the first 10 testing samples and compare to the actual survival outcomes
print('Testing Sample Labels:      ', testing_labels[0:10])
sample_predict = model.predict(testing_data[0:10,:])
print('Predicted Survival Outcome: ', sample_predict)
```

```
Testing Sample Labels:       [1. 1. 1. 1. 1. 0. 0. 0. 1. 0.]
Predicted Survival Outcome:  [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Already we see that some of the predictions were not correct.

### Evaluate performance by computing the percentage of correctly predicted survival outcomes.

To get a true evaluation of performance we need to test on a larger set of data. Let's predict the outcome on all of the testing data and compute a percentage of how many survival outcomes were correctly predicted.

In [63]:
```
# obtain survival predictions on all testing data
testing_predicted = model.predict(testing_data)

# obtain a percentage score of performance on all testing data
score = 100*(1-sum(abs(testing_predicted-testing_labels))/len(testing_predicted))
print('Testing data performance', score, '% correctly predicted')
```

```
Testing data performance 77.57847533632287 % correctly predicted
```

How does that compare with the samples obtained from the training data?

In [64]:
```
# obtain survival predictions on all training data
training_predicted = model.predict(training_data)

# obtain a percentage score of performance on all training data
score = 100*(1-sum(abs(training_predicted-training_labels))/len(training_predicted))
print('Training data performance', score, '% correctly predicted')
```

```
Training data performance 85.39325842696628 % correctly predicted
```

The prediction achieved better performance, 85.4% on the training data than the testing data 77.6%. This makes sense because the machine learning algorithms are trying to model the training data not the testing data.

Let's see if we can get better performance by adjusting the training parameters (i.e. max_depth) and applying other machine learning algorithms.

## Compare Prediction Performance Across Machine Learning Algorithms

As a final step we will adjust our training parameters and machine learning algorithms to see if we can do any better on the survival prediction performance.

Test **Decision Tree** Learning Algorithm

In [65]:
```
# Decision Tree
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(max_depth=15)

# Fit the model to our training data
model.fit(training_data, training_labels)

# Make predictions
testing_predicted = model.predict(testing_data)
score = 100*(1-sum(abs(testing_predicted-testing_labels))/len(testing_predicted))
print("DT Test:", score)
```

```
DT Test: 75.56053811659193
```

There you go, you've just implemented a high level prediction in only a couple lines. Since each dataset is different, we may find that there are other algorithms that are better suited for this prediction. Let's examine some of the other popular machine learning algorithms.

Test **Random Forest** Learning Algorithm

In [66]:
```
# Random Forest
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=250)

# Fit the model to our training data
model.fit(training_data, training_labels)

# Make predictions
testing_predicted = model.predict(testing_data)
score = 100*(1-sum(abs(testing_predicted-testing_labels))/len(testing_predicted))
print("RF Test:", score)
```

```
RF Test: 79.14798206278026
```

Test **Support Vector Machine** Learning Algorithm

In [67]:
```
# Support Vector Machines
from sklearn import svm
model = svm.SVC(gamma=2, C=1)

# Fit the model to our training data
model.fit(training_data, training_labels)

# Make predictions
testing_predicted = model.predict(testing_data)
score = 100*(1-sum(abs(testing_predicted-testing_labels))/len(testing_predicted))
print("SVM Test:", score)
```

```
SVM Test: 61.43497757847533
```

Test **Naïve Bayes** Machine Learning Algorithm

In [68]:
```
# Naive Bayes
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()

# Fit the model to our training data
model.fit(training_data, training_labels)

# Make predictions
testing_predicted = model.predict(testing_data)
score = 100*(1-sum(abs(testing_predicted-testing_labels))/len(testing_predicted))
print("NB Test:", score)
```

```
NB Test: 79.82062780269058
```

These are just a few of the available machine learning algorithms at our disposal. Designing these algorithms would have taken hours of work, fortunately, the Python open source community is strong and many people out there are willing to contribute. Hence, all we have to do is change one or two lines in our code to evaluate.

# Perform Final Testing

From the above algorithms tested, it seems like the Naive Bayes performed the best.

Now that we have selected our machine learning model for predicting survival, we can finally answer the question of whether or not Jack and Rose would have survived (test case 3). To make this prediction we need to provide information on the passengers in the expected form:

| Passenger | Pclass | Sex | Age | SibSp | Parch | Fare | Embarked |
|---|---|---|---|---|---|---|---|
| Jack | 3 | 0 | 25 | 0 | 0 | 7 | 2 |
| Rose | 1 | 1 | 22 | 1 | 0 | 50 | 2 |

Given the provided information about Jack and Rose, would they have survived the tragedy?

In [69]:
```
# Decision Tree
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(max_depth=15)

# Fit the model to our training data
model.fit(training_data, training_labels)

# Create test data (features of Jack and Rose)
test_jack = np.array([3, 0, 25, 0, 0, 7, 2])
test_rose = np.array([1, 1, 22, 1, 0, 50, 2])
```

```
testing_predicted = model.predict(test_jack.reshape(1,-1))
print("Jack Survival:", testing_predicted)

testing_predicted = model.predict(test_rose.reshape(1,-1))
print("Rose Survival:", testing_predicted)
```

```
Jack Survival: [0.]
Rose Survival: [1.]
```

## Take home questions

(1) In performing our testing we have made some fundamental mistakes that results in overfitting to our 4 test cases. Spend some time looking through how we setup our training and testing data, in particular take a look at the test cases. Indicate what the problem is and suggest some ways to correct it.

In [70]:
```
# correct the above test process and evaluate the performance of the algorithms
```

(2) Exploration! Compare different machine learning algorithms and data preprocessing to see if you can achieve a better prediction accuracy. How can you be sure that you are not overfitting to the data?

In [71]:
```
# write your code here
```