Jupyter 101

A friendly introduction into using the Jupyter Notebook environment.
Learn to process and visualise data!

[Task 1] Preface

Data science is a very broad, enormous topic that isn't (in my opinion) easy to approach and much harder to master. It has a huge variety of uses across all types of industries! To name a few real-world examples of where data science can be found day-to-day:
- **Recommendations of content on Netflix and YouTube based upon your previous viewing history**
- **Fraud detection in Banking**
- **Intrusion Detection for Cyber Security**
- **Weather Forecasting / Prediction**
- **Metrics of a Business' sales performance**
- **Route planning in Google Maps.**

It is also goes hand-in-hand with Machine Learning / Artificial Intelligence.
Despite the following room using Python, I have tried to explain any code as much as possible. The code itself is fairly simple, so if you're not familiar with Python, you can still follow along just fine!
Hopefully the following room provides a friendly introduction into the "Jupyter Notebook" environment which is an extremely useful application in the world of data science.

Moreover, I hope the tasks below are a practical and engaging approach into the very common "Pandas" library, which is where the data is crunched. Finally, the "Matplotlib" framework - which is my favourite part, where we can visualise all of the data we have handled!
This is my first room release, so please excuse me if I have overlooked anything. If there is an interest about the topics from this Room, I'll create further content where we will go much more in depth and apply knowledge to solve practical problems (assuming my University studies permits, alongside being generally super-hectic) But in the meanwhile, I hope you enjoy!
For any and all feedback, questions, problems or future ideas you'd like to be covered - please get in touch:
Discord: @CMNatic#9812

I frequent the TryHackMe Discord often, but regardlessly, please feel more than free to ping me in that Discord!
So long and thanks for all the fish!
~CMNatic
Shout out to MuirlandOracle for his valuable feedback during testing!

#1
Lets' ago!

No answer needed

[Task 2] What is Jupyter?

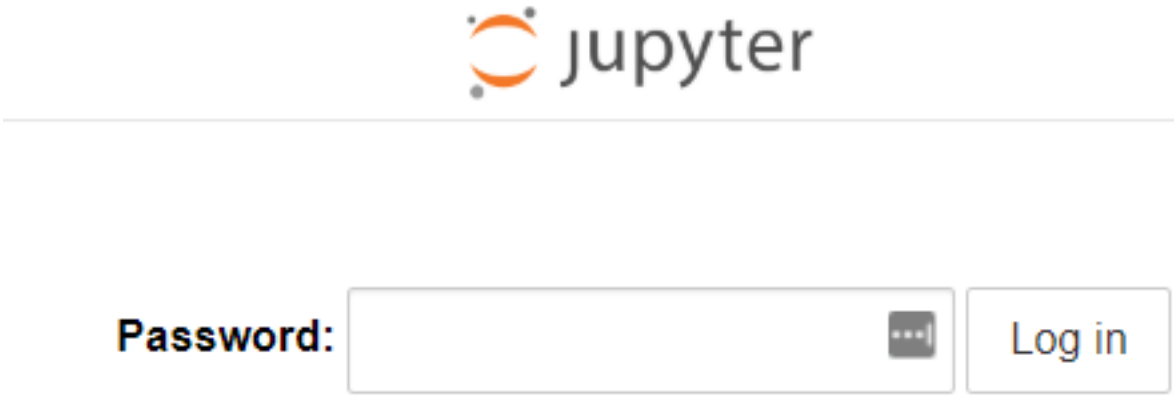
Jupyter is a web-based platform often used for data analytics / plotting, machine learning, where code is stored in "Notebooks" whom can imported/exported and shared in many formats such as LaTex, HTML, PDF and many more!
I have created a pretty useful introductory post on my blog, annotating how to interact with Jupyter. Feel free to figure your way around the interface yourself! The interface is fairly intuitive itself, but if you're new to it, I'd very strongly recommend going through it to get familiar. It's only a minute or two read.
Read the support material here!
The Jupyter Notebook environment isn't the easiest thing to install and deploy in the world. Normally you have to provide configuration files, Python environments and all sorts of fun things. I've done the leg-work here and made a cloud-friendly deploy of Jupyter that launches on boot.
Jupyter is great because you are able to import/export any Notebooks that you create or any other Notebooks that people have made.
Due to the nature of Jupyter, it's pretty hard to break anything. But if you wish to restore the provided Notebooks - you can simply redeploy the instance!

#1
Read the supporting material.

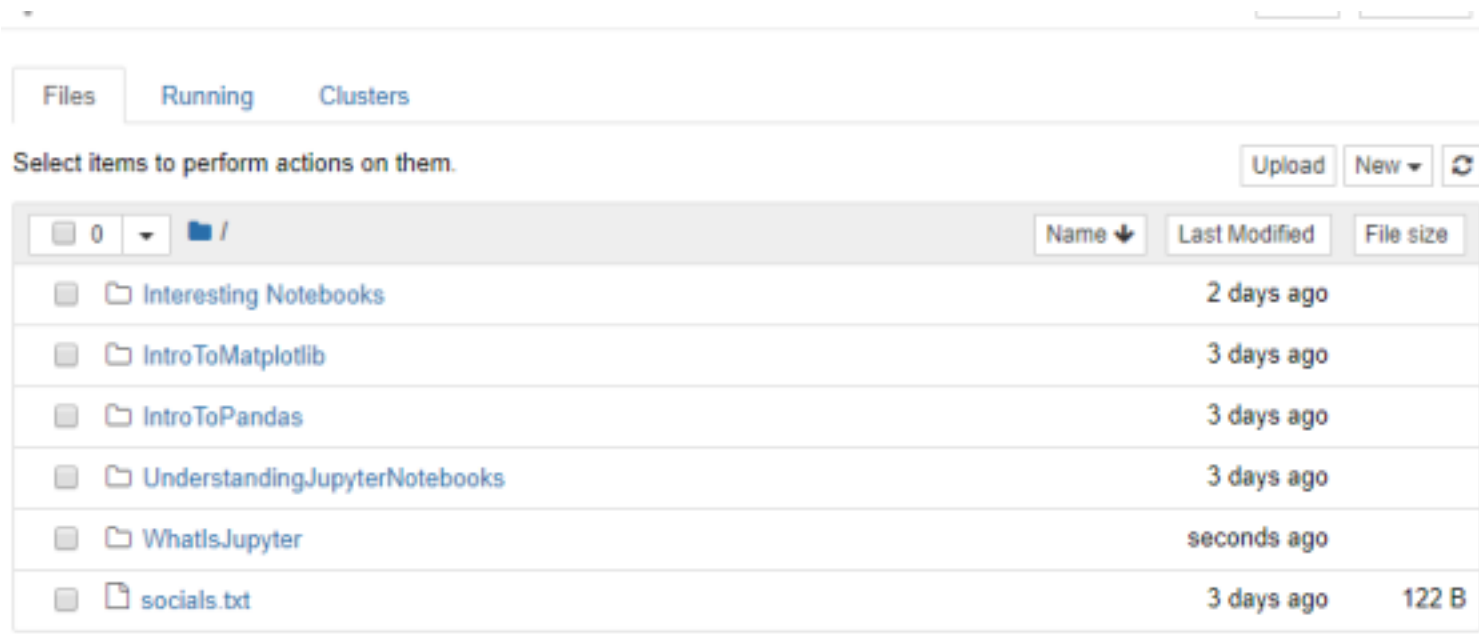
No answer needed

[Task 3] Deploying Instance & Logging In

Firstly, deploy the instance by pressing the "Deploy" icon on the top right of this box.
Please wait 5 minutes for the Instance to fully boot before trying to access it. If you cannot access it after 5 minutes, please "Terminate" and re"Deploy"
When you launch the instance and navigate to it, you will be presented with a login screen like so:



Jupyter uses authentication protocols and tokens for user management, which are a pain to manage. I couldn't find a workable solution to completely remove this, so I have done my best to work around it.
Use the password: tryhackme
If you have logged in successfully, you will be presented with the following:



You're ready to go!

Excerpt from the supporting material:

"Jupyter directly interacts with the Operating Systems filesystem (assuming the user it is running as has the necessary permissions!). This means you can create and upload files and traverse folders - as if you would on the host itself. Similarly, anything you delete on the host - gets deleted on Jupyter...you get the picture..."

We'll be exploring this later...

#1
I have logged in!

No answer needed

[Task 4] Let's Learn More About Jupyter

Enter the "WhatIsJupyter" directory and launch "WhatIsJupyter.ipynb" and have a read through.
There's no questions, but I detail some use cases of Jupyter and why someone might use it over an IDLE such as PyCharm!

#1
Launch Jupyter

No answer needed

WhatIsJupyter.ipynb

Let's go into further detail about Jupyter itself, why you'd use it and why it's so great!

I'll be fighting Jupyter's corner here (and rightly so), explaining why it's such a powerful application and a really great environment to add to your toolbelt!

Comparing it to the traditional IDLE:

Jupyter is in by no means a fully-fledged replacement of the traditional IDLEs like PyCharm (Yes...It's still worth renewing your license for it). It can be considered merely as an "addon", but a powerful one at it.

Firstly: Performance

IDLE's can be fairly intensive on your systems resources when running large and/or complex scripts. If you've got a powerful computer - that's manageable! But if you're out-and-about, at a hackathon or any scenario where you have a lower-spec system, it's an inconvenience to say the least.
Jupyter can be installed on both your local system and a remote Server - such as this instance! Servers by nature have much larger hardware capabilities, where it's fairly safe to say they won't have an IDLE like PyCharm at hand for you to use.
A Jupyter Server install rectifies the problem with developers needing large computing power, in a situation where they do not have the facilities for such.
Finally, when using an IDLE, your workflow consists of modifying a bit of code, running the entire script, finding the output and acting accordingly. Because of Jupyters use of "Notebooks", it allows sections / Cells of code to be repetitively executed, where the output is shown directly below the Cell, all without needing to re-run the entire program! What a time saver. Your workflow now becomes:
1. Program the Cell
2. Execute the Cell
3. Visualise the data instantly
4. Modify the code if you need too
5. Execute the exact same Cell you modified
6. Review the new visualisation
7. Repeat
This workflow does not interact with any of the other Cells i.e. you're not modifying code and re-running entire Python scripts like you would in an IDLE.

Secondly: Documentation

You can't use languages like Markdown in an IDLE like PyCharm (sorry to be picking on you guys, I love you really...Just as long as you keep on providing me my educational license ta!)
This entire notebook here is built using Markdown, imagine trying to comment all of this in a python script? Well for one, you can't style it with Headers or bold or insert pictures!
Jupyter Notebooks are incredible for documenting (assuming the programmer documents grumbles).

Thirdly: Sharing

IDLE's like PyCharm facilities a great thing here: Git / Version Control. You can use git within these applications to push/pull/merge/branch to your hearts content. Jupyter Notebook doesn't have this facility unfortunately. Git is a very large narrative in a developers life.
However, where Jupyter prevails is its ability to "Import" and "Export" its Notebooks, and this extensive capability is extremely attractive to data scientists, or programmers who need to run little snippets.
Any Notebooks you make, you can export into various formats like PDF's, HTML, LaTeX for example. Where all the data is already visualised! You can handle a dataset, plot it in a graph, export it to a PDF and whack it on Mr. Executives desk!

Summary:

Jupyter is a great environment for testing snippets and prototypes of Python code ultimately visualising this data.
Jupyter does not however, replace the extensive capabilities of IDLE's, where applications are created.
Jupyter is incredible for exporting pretty graphs and metrics into friendly PDF formats for non-programmers such as executives. It is additionally fantastic in being used alongside material in a Workshop or demonstration - everything is already rendered and is itself easily accessible.

Jupyter can...

• Be used within the browser!
• Directly interact with the Operating System, such as installing system packages, python packages and traversing/-modifying the filesystem!
• Perform little snippets of code, multiple times without affecting other Cells.
• Support up to 40 languages using "Kernels" or virtual environments which are easily interchangeable.
• Visualise plots
• Be imported into other environments, making it perfect to use as material within workshops, or sharing with Students!

• Facilitate extensive documentation
• Display a wide variety of media such as graphs, images, videos, equations (and I think someone has even made a game?!)
• Be tricky to install, especially on a remote server - as you have to deal with IP address bindings, token managements, and all sorts of gubbins.

Jupyter can't...

• Replace the existing IDLE. These applications facilitate whole software development, where you can debug your code, use syntax highlighting, version control, use plugins like Console integration to name a few.
• Display things like GUI's

Finally

I have included a few varied Notebooks from the A gallery of interesting Jupyter Notebooks](https://github.com/-jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks) just to show how versatile Jupyter is, the accessibility and how you too can share your own Notebooks! You can find these in the "Interesting Notebooks" directory

[Task 5] Understanding how Jupyter Notebooks Run

Navigate into the "UnderstandingJupyterNotebooks" directory and launch "UnderstandingHowJupyterExecutes.ipynb" and read the annotations I have made to understand how to answer this.

UnderstandingHowJupyterExecutes.ipynb on next page

#1
What do "Cells" act like?

interpreter

#2
What would be the In[#] value of the first Cell when it is ran for the first time? (Where # would be the numerical value)

1

#3
What keyboard shortcut can you press to execute a cell?

shift + enter

#4
If you was to execute the first Cell again, what would the value of In[#] now become? (Where # would be the numerical value)

2

UnderstandingHowJupyterExecutes.ipynb

Cells act just like an interpreter!

This means you can run code like arthmetic and well anything else you can do on an interpreter.

```
In [1]: print('Hello World!')

        Hello World!
```

Now lets use multiple cells

Normally, once a Notebook is executed, it will step through to each Cell unless it runs into a problem. Each "step" the Notebook has done is indicated by the "In [#]:" next to the Cell, where # would be the value of the order it was executed.
So naturally, if there was no issue, the notebook will proceed to the next cell, where its "In [#] value would now become 2

```
In [2]: print('See how the In[2] value is 2?')

        See how the In[2] value is 2?
```

However, a beautiful feature of Jupyter is that you don't have to run the entire Notebook just to execute the code within a Cell. You can just execute that individual Cell! You can do this by either of the following:
• Shift + Enter
• Pressing the >| sign next to the Cell
However, as Cells are treated "individually", it means that if you have say variables in cells above that you have not yet executed, you will have a problem.
It should also be noted that if you execute a cell again, the "In [#]" value will increment, as it is technically another step through the notebook (albeit that you are simply executing the same cell again)

For example:

Cell 2 that has the In[2] value, if it is executed 5 more times before Cell 3 is, then Cell 2's will become In[7]. If Cell 3 is then executed, Cell 3 will increment by one, so In[8], and will incremenent appropriately as it is executed. Restarting the kernel (Refresh-looking icon on the Toolbar) will reset these In[] values.These values aren't super important, but it's good to know why those values are what they are when trying to debug your code.

Example of using Cells Together

Let's store the value "TryHackMe is a great platform!" into a string and then output the length of the string.

```
In [3]: s = "TryHackMe is a great platform!"
```

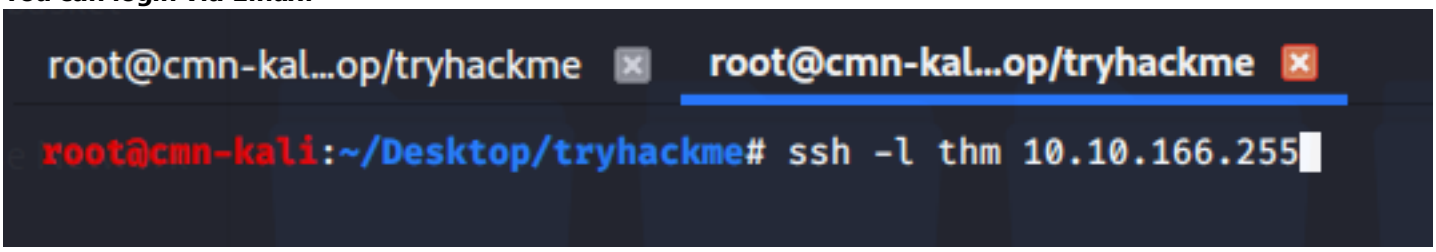
```
In [4]: len(s)
Out[4]: 30
```

The cell containing the string "s" must be executed for the cell with `len(s)` to work. If the cell storing "TryHackMe is a great platform" into a string is not executed, then `len(s)` will not work. This is essential to know.

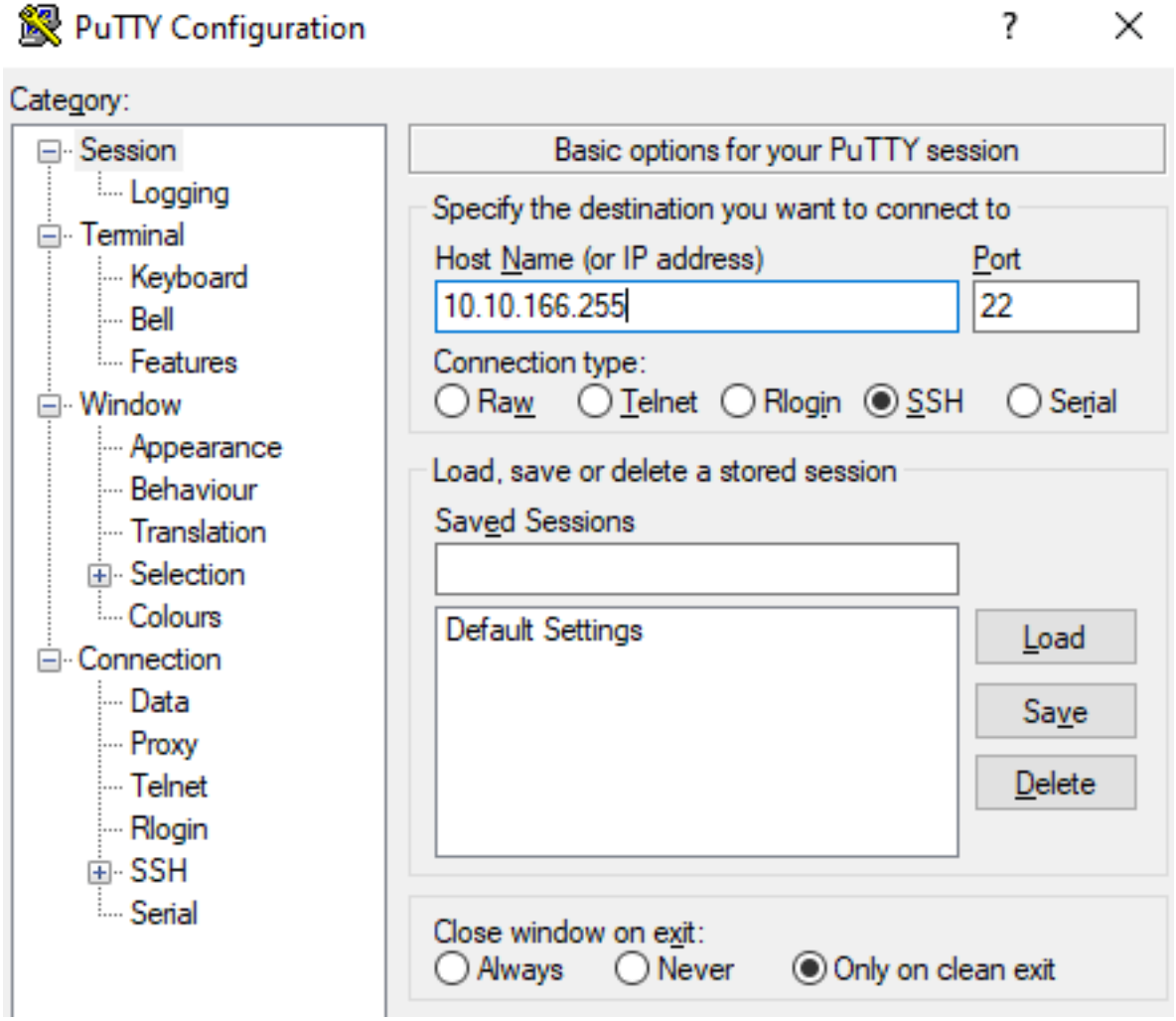
[Task 6] Interacting With the Filesystem!

As previously mentioned throughout the walkthrough, Jupyter directly interacts with the Operating System's filesystem. For example, making files, folders and/or Notebooks.
These files are reflected on the Operating System that Jupyter is running on. The directory that you are shown after logging into Jupyter is for all intents and purposes, the "root" directory of Jupyter.
However, just because it is the "root" directory, it does not mean it is the Operating Systems /root/ directory. It is simply the directory of wherever Jupyter was launched upon / or told to set to during configuration.
In this case, I have told Jupyter to launch within the "thm" users home directory, where you will later log into and see for yourself.
Lets log into the Instance, replacing any IP address in the pictures below with that of the Instance you have deployed.
Username: thm
Password: tryhackme
Port: 22

You can login via Linux:



Or via a Windows application such as Putty



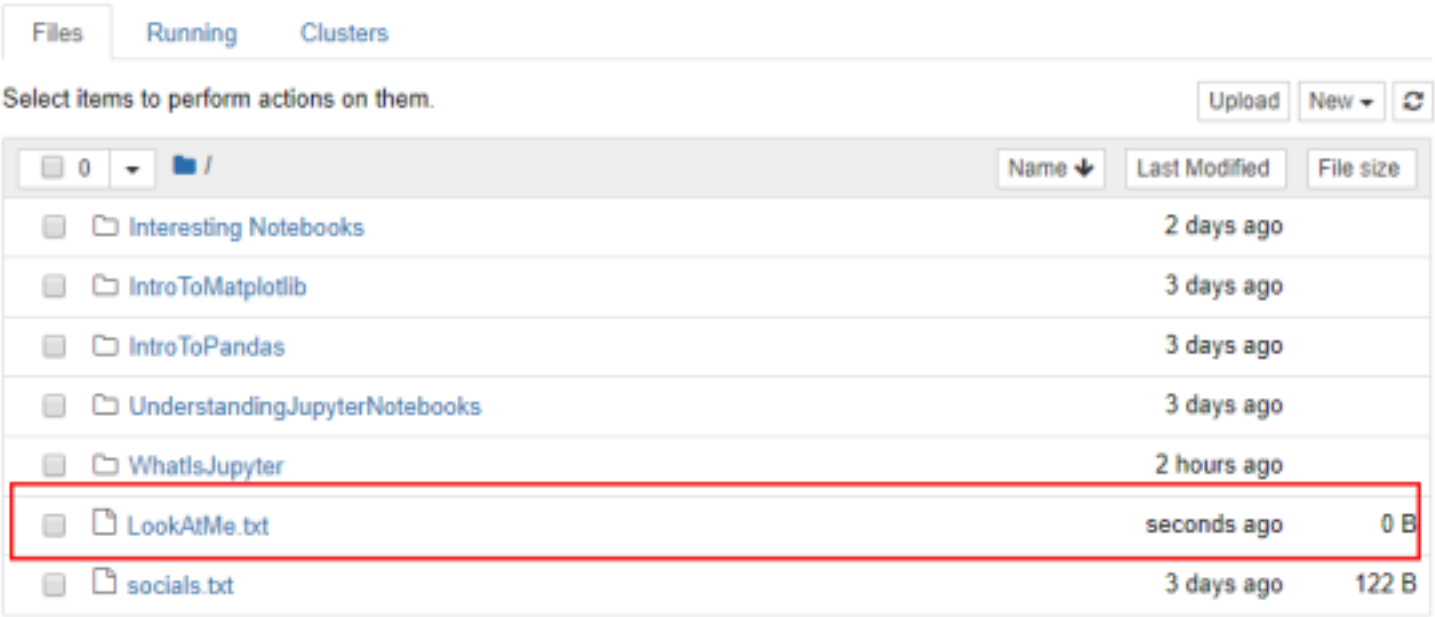
Now, after successful login, lets see what's going on, using ls.

```
thm@thm-jupyter101-vm:~$ ls
'Interesting Notebooks'   IntroToPandas   UnderstandingJupyterNotebooks
 IntroToMatplotlib        socials.txt     WhatIsJupyter
thm@thm-jupyter101-vm:~$ touch LookAtMe.txt
```

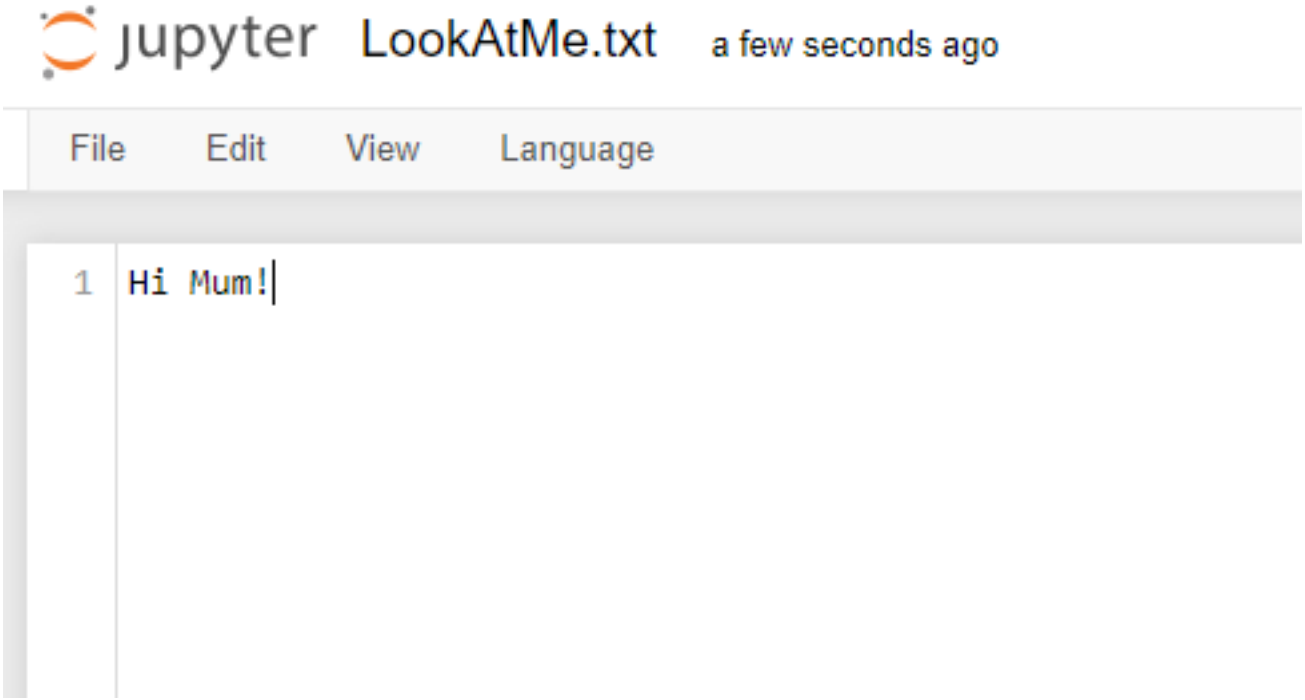
We then create a new file using touch in this case, "LookAtMe.Txt"

```
thm@thm-jupyter101-vm:~$ ls
'Interesting Notebooks'   IntroToPandas   socials.txt                  WhatIsJupyter
 IntroToMatplotlib        LookAtMe.txt    UnderstandingJupyterNotebooks
thm@thm-jupyter101-vm:~$
```

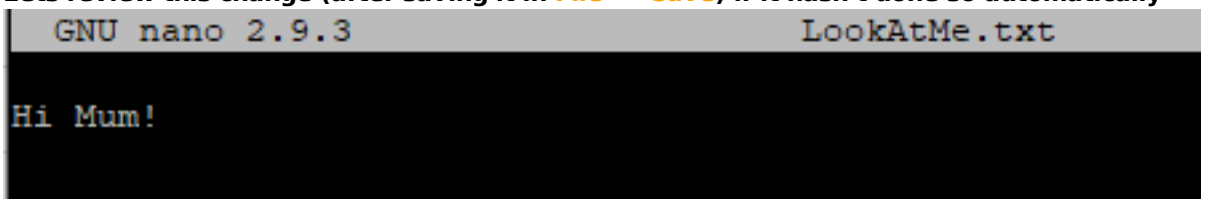
Returning back to the Jupyter instance in your web browser, we can now see the file we have just created is visible within Jupyter.!



When opening the file, I've put written in some text using Jupyter:



Lets review this change (after saving it in File -> Save) if it hasn't done so automatically



Ta-dah! And there we have it.

Jupyter directly uses Linux' User permissions. So you will only be able to read/write/modify the files that the you (the user the Jupyter Server is running as) within the directory (that has either been specified in configuration, or from where it is launched)

#1
Follow the instructions above.

No answer needed

[Task 7] Handling Data With Pandas

Pandas is a fantastic library for data wrangling. It allows us to read data from a wide variety of formats such as CSV files, JSONs, Databases and more!
Datasets are often very large. Whilst the contents of them might be useful to us, these contents may not be in the exact format we need! Nor may we need every single value - only a select few. Let's get started...
Navigate to the "IntroToPandas" directory on the Jupyter instance and run "IntroToPandas.ipynb" please read the annotations.

IntroToPandas.ipynb on next page

#1
What are the two main types of data within Pandas?

series and dataframes

#2
What is the name of the Pandas function that reads a CSV file?

read_csv

#3
Name the Pandas function you would use if you only wanted to display the first few rows

head

#4
Name the Pandas function you would use if you only wanted to display the last few rows

tail

#5
What Pandas function will give you a numerical count of the amount of columns and rows the dataset contains?

shape

IntroToPandas.ipynb

Pandas!
Whilst they're cute, we're not talking about those Pandas. We're talking about Pydata's Pandas
I'm assuming you're familiar with the basics of Python or programming syntax as a general.

Pandas is:
• A fantastic framework built specifically for data analysis, number crunching and manipulation!
• An essential tool in Data Science
• Easy to pickup, hard to master. Your results will ultimately depend on how useful your dataset is. Especially considering there's many ways of going about it.
I mean, if the cheatsheet doesn't scare you...Fret not though!

The two main components of Pandas
• Series
• Dataframes
Series can be thought of a singular colum in a spreadsheet. It is one-dimension, so can still hold anything like integers, strings, etc.
Dataframes are more complex, and a lot more common. These can be considered as a fully-capable spreadsheet, with multiple columns and rows.

First, we need a dataset!
There's loads and I mean loads out there for you to experiment with, but for the moment I've made some simple, lightweight and fairly generic datasets that we can use.
You can import data through various means, for example:
• Database
• CSV Files
• API's

In [1]:
```
# Lets import pandas! Normally this is not a standard Python library, so you'd have to install via pip (or Anaconda
import pandas as pd


# Store the integers into "series"
series = pd.Series([10, 15, 30, 35, 70, 75])

print(series)
```

```
0    10
1    15
2    30
3    35
4    70
5    75
dtype: int64
```

We're already using Pandas!
We can see Pandas has outputted our integers into a single column, displaying their index numbers (remember that Python starts at 0)

But that's very basic and not very helpful, so let's start getting to the nitty-gritty
Dataframes

In [2]:
```
# Lets visualise the data of a few purchases from an e-commerce site

orders = {
    'T-Shirts': [5],
    'Mugs': [2],
    'Keyrings': [10]
}

# Insert the values from "orders" into Pandas "Dataframe"

orders = pd.DataFrame(orders)

# And display!
orders
```

Out[2]:

	T-Shirts	Mugs	Keyrings
0	5	2	10

Okay, so it's printed out a table! Great! We have the three columns - the categories, but there's still the index number (0) Let's add more data and see what happens
Pandas requires the array values to be the same.

In [3]:
```
orders = {
    'T-Shirts': [5, 12, 8, 1],
    'Mugs': [2, 3, 22, 7],
    'Keyrings': [10, 20, 55, 30]
}

# Insert the values from "orders" into Pandas "DataFrame"
# Each Category in the array will correspond to a column in Pandas.

orders = pd.DataFrame(orders)

# And display!
orders
```

Out[3]:

	T-Shirts	Mugs	Keyrings
0	5	2	10
1	12	3	20
2	8	22	55
3	1	7	30

Sweet! We got more data, it's a bit jumbled up though, and we still see the index numbers which tell us nothing!

In [4]:
```
# We can use Pandas to add an index - replacing the numerical "0 - 3" for each row, we can replace it with somethin
# For example, Months!

orders = {
    'T-Shirts': [5, 12, 8, 1],
    'Mugs': [2, 3, 22, 7],
    'Keyrings': [10, 20, 55, 30]
}


orders = pd.DataFrame(orders, index=['January', 'February', 'March', 'April'])

orders
```

Out[4]:

	T-Shirts	Mugs	Keyrings
January	5	2	10
February	12	3	20
March	8	22	55
April	1	7	30

Ah! Now that makes more sense, we can see that March was their busiest Month for purchases!
But we are only showing 4 Months here, what if we had a dataset that had all 12 Months? We can use Pandas to single out specific indexes.
In other cases, we could use this to locate orders by a specific customer, for one of many examples.

In [5]:
```
orders.loc['April']
```

Out[5]:
```
T-Shirts     1
Mugs         7
Keyrings    30
Name: April, dtype: int64
```

Lets start reading from CSV Files

I've made a few little spreadsheets with random values that we can have a play with. Again, because these are multi-dimensional, we are still using dataframes
CSV & misc files contains delimiters, these are identifiers that are used to seperate text values. Without these, all the text will be merged and read into one row for example.
My dataset uses the ',' delimiter, so lets specify that when loading the CSV

In [6]:
```
# load the csv file into the dataframe
# Pandas has a literal function for reading CSV files! How about that
# Then we specify our datasets delimiter

csvfile = pd.read_csv('simple_orders.csv', delimiter = ',')

csvfile
```

Out[6]:

	Month	T-Shirts	Mugs	Keyrings
0	January	22	43	22
1	Feburary	10	246	231
2	March	14	14	34
3	April	2	10	12
4	May	4	2	4
5	June	87	63	7
6	July	43	432	2
7	August	14	123	4
8	September	16	67	8
9	October	29	100	3
10	November	3	12	1
11	December	5	23	7

We have literally just outputted an entire spreadsheet in essentially two lines of code (ignoring the imports at the start of the notebook)
Still notice the index numbers though? I certainly didn't put them in the spreadsheet, so that's Pandas doing that - as we've seen before.

In [7]:
```
csvfile = pd.read_csv('simple_orders.csv', delimiter = ',', index_col=0)

csvfile
```

Out[7]:

Month	T-Shirts	Mugs	Keyrings
January	22	43	22
Feburary	10	246	231
March	14	14	34
April	2	10	12
May	4	2	4
June	87	63	7
July	43	432	2
August	14	123	4
September	16	67	8
October	29	100	3
November	3	12	1
December	5	23	7

By specifying what column is the index, we have removed the automatic assumption that Pandas makes which results in the addition of the index numbers as an additional column. Sweet!
You don't need to specify the column number that is the index, you can instead just specify the value of it, like so.

In [8]:
```
csvfile = pd.read_csv('simple_orders.csv', delimiter = ',', index_col="Month")

csvfile
```

Out[8]:

Month	T-Shirts	Mugs	Keyrings
January	22	43	22
Feburary	10	246	231
March	14	14	34
April	2	10	12
May	4	2	4
June	87	63	7
July	43	432	2
August	14	123	4
September	16	67	8
October	29	100	3
November	3	12	1
December	5	23	7

We get the exact same result, just by specifying the column name instead of its index number!

But! This is just a small dataset...

Very oftenly, datasets such as CSV files are not small in the slightest. Our current method outputs the entire contents, imagine trying to display 4000 rows and 200 columns? Not very effective...
We can instead, just tell Pandas to display a certain number of rows.
In our example we have 12 rows, lets tell Pandas to only display 3.
Just because only 3 are being rendered, the entire 12 rows are still loaded

In [9]:
```
# You can replace "3" with whatever number you like. Well..upto 12 anyways.

csvfile.head(3)
```

Out[9]:

Month	T-Shirts	Mugs	Keyrings
January	22	43	22
Feburary	10	246	231
March	14	14	34

In [10]:
```
# You can also just display a certain number of the bottom rowss if you wish! This is fairly common when handling d
# So that we can get a good picture of makeup of the dataset without trying to render it all.

# Again, you can replce "3" with whatever number you like.

csvfile.tail(3)
```

Out[10]:

Month	T-Shirts	Mugs	Keyrings
October	29	100	3
November	3	12	1
December	5	23	7

We've managed to tell Pandas to only display "January, Feburary and March" and then later to only display "October, November, December" using `.head(#)` `.tail(#)` Neat! These sort of things are super super common when handling data, so we've nailed a core principle.

Before we start the pretty stuff:

There's a final little snippet that is again, another super useful and important function in data handling. the `shape` This is often used when cleaning up datasets (they are often very wishy-washy to what we need, so `.shape` is a great way to display what exactly our filtering has done.

```
In [11]: # Simply, rather then rendering any data, lets just figure out a numerical value for what's in there!

         csvfile.shape

Out[11]: (12, 3)
```

Panda's `.shape` processes the specified file, and counts the contents in the following format: rows, columns
So in our case, we have 12 rows and 3 columns
Finally, what if we wanted to just count the amount of values in a column or category, such as mugs?

```
In [12]: csvfile[['Mugs']].count()

Out[12]: Mugs    12
         dtype: int64
```

Good job! Answer the questions in the room, and then we'll see how we can plot this data into graphs!

[Task 8] Visualising Data With Matplotlib

Navigate to the "IntroToMatplotlib" directory on the Jupyter Instance and run "IntroToMatplotlib.ipynb" please read the annotations.

IntroToMatplotlib.ipynb on next page

#1
How do you display a plot?

plot()

#2
How would you label the "x" axis on a plot? Note: do not add the brackets () for this answer

xlabel

#3
How would you label the "y" axis on a plot? Note: do not add the brackets () for this answer

ylabel

#4
How would you add a "Title" to a plot? Note: do not add the brackets () for this answer

title

#5
What word would you use to change the color of the plot?

color

#6
How would you label the "z" axis on a plot? Note: do not add the brackets () for this answer

zlabel

IntroToMatplotlib.ipynb

Matplotlib!

Like I said, data handling is fun and all (not in the slightest when the datasets are large and iffy). But I for one, certainly enjoy a Bar Chart or Plot Line or two.

And that's what we'll be doing. We'll be extending upon the previous dataset that I made for task 2 so this could be interesting, as there's quite some numerical ranges...

In [1]:
```
# Lets import matplotlib
# However, you never really have matplotlib without pandas, so we'll do both

import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import style
from numpy import genfromtxt
```

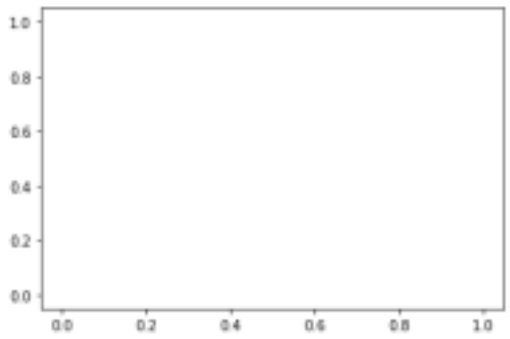
In [2]:
```
# This is only because we are using Jupyter Notebook, this ensures that any plotting and figures are displayed in t
# Otherwise it gets very iffy.
# You ONLY use this snippet if you using Matplotlib in a Jupyter Notebook

%matplotlib inline
```

Lets begin

Like I said, we're going to be using the dataset from the last task, but we'll use a bigger - more populated one to make pretty graphs.
This ones a little bit tricky in comparison, as we're combining both Pandas and a new library, which isn't quite so straight-forward as Pandas.
We don't need even need data to make our first plot!

In [3]:
```
plt.plot()
```

Out[3]:
```
[]
```



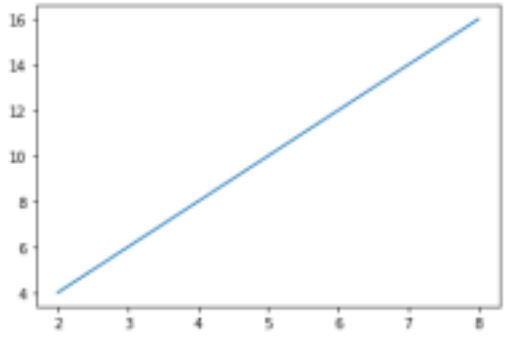
Ta-dah!

But what use is that? Sure we have a plot, but there's no actual data, no labels - what does it tell us?
We can populate the graph with data from a dataset or an array for example! using `plt.plot`
If you do not provide a style type, e.g. bar chart or scatter graph, matplotlib will default to a line graph.
plt plot interprets the format like so:
1. Data along the graph
2. Data up the graph
X -> Y
Or how I was taught in secondary (high) school:

Along the corridor, up the stairs! shudders

In [4]:
```
# We'll start by making a very simple plot using an array before we get into the nitty gritty.
# Remembering it goes X -> Y

plt.plot([2,4,6,8],[4,8,12,16])
```

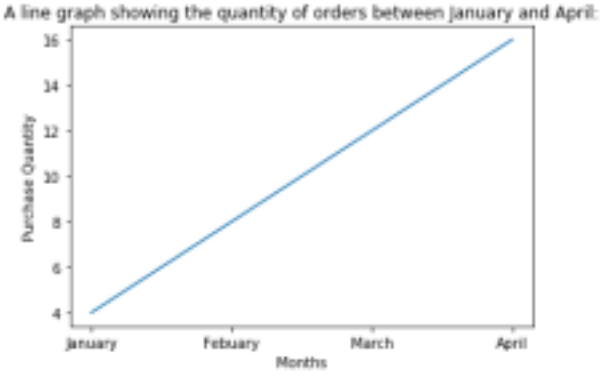
Out[4]:
```
[<matplotlib.lines.Line2D at 0x7f83fcc699e8>]
```



Now we have a bit of data! What it means however - only the programmer knows! That's not very helpful...Perhaps we should add labels? Let's use `plt.xlabel` and the respective `plt.ylabel`
Of course, if you were using a 3D plot (which is very possible, but beyond this introduction, you'd have to label the Z dimension. Take a guess at how you'd do that)

In [5]:
```
# Lets provide data to plot - again, remembering it goes X -> Y
plt.plot(['January','Febuary','March','April'],[4,8,12,16])

# Set X,Y Labels
plt.xlabel('Months')
plt.ylabel('Purchase Quantity')

# We can also add a title to describe the graph using plt.title!
plt.title('A line graph showing the quantity of orders between January and April:')
```

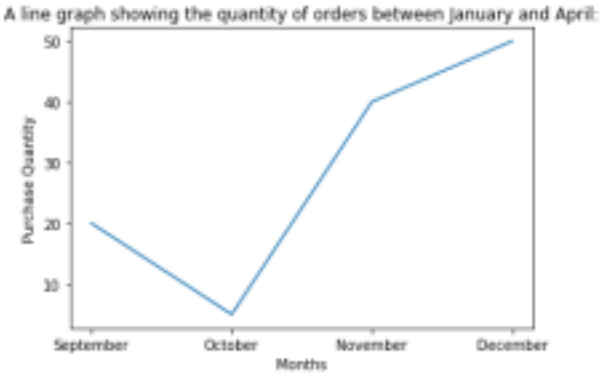
Out[5]:
```
Text(0.5, 1.0, 'A line graph showing the quantity of orders between January and April:')
```



The e-commerce site in this case done rather well considering! Now we can deduce that as the Months have passed, they have recieved more orders.
Let's plot another one, assuming that they didn't have quite a similar pattern in the later Months of the year.

In [6]:
```
# Lets provide data to plot - again, remembering it goes X -> Y
plt.plot(['September','October','November','December'],[20,5,40,50])

# Set X,Y Labels
plt.xlabel('Months')
plt.ylabel('Purchase Quantity')

# We can also add a title to describe the graph using plt.title!
plt.title('A line graph showing the quantity of orders between January and April:')
```

Out[6]:
```
Text(0.5, 1.0, 'A line graph showing the quantity of orders between January and April:')
```



Woah! They obviously didn't do too well in October - maybe a PR disaster? Or simply their products were "out of fashion"?
Line graphs are a great way to visualise simple data, such as that of quantity of purchases over Months.

Adding our own little flair

there's so so many combinations and configurations you can make of your plot, Matplotlib is super detailed and customisable. It's incredible really. See a list of a 'few' examples
For example, lets change the plot line to green! We specify this paramater at the `plt.plot` function

In [7]:
```
# Lets provide data to plot - again, remembering it goes X -> Y
plt.plot(['September','October','November','December'],[20,5,40,50], color='green')

# Set X,Y Labels
plt.xlabel('Months')
plt.ylabel('Purchase Quantity')

# We can also add a title to describe the graph using plt.title!
plt.title('A line graph showing the quantity of orders between January and April (but now in green):')

plt.plot()
```

Out[7]:
```
[]
```

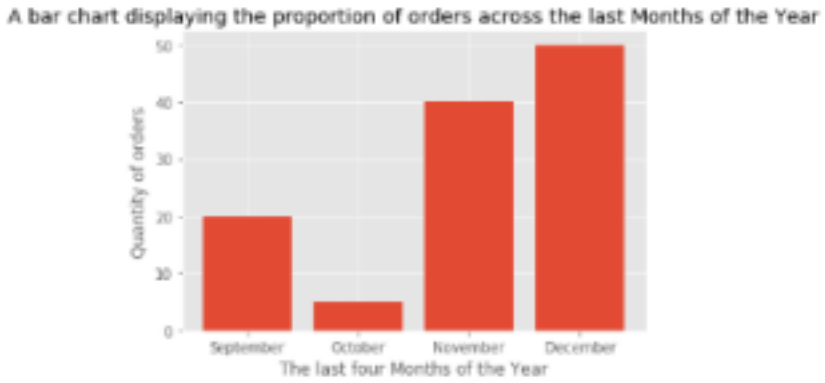
But that's not all...

Rather then using a line graph, why not use a bar chart? This too is a great way of visualising data and can be greatly extended for different use cases, as we'll explore below.
For example, we might want to use bar charts to display values from categories. We'll use the same data as above in this example, and will further extend it later.

In [8]:

```
# Here is where we have to specify the type of plot we want to use with matplotlib
# As otherwise, matplotlib (plt) will use a line graph.
plt.style.use('ggplot')

# Lets specify the Months, and their order values
months = ['September', 'October', 'November', 'December']
orders = [20, 5, 40, 50]

# Lets create a new variable that enumerates every value in 'months'
# Aka, lets get every 'Month' that we have specified in 'months'

# Then lets store that into months_orders, which we will use for plotting
# This allows the Months to be displayed under the bars, essentially replacing the "x label" if we wanted too...be
months_orders = [i for i, _ in enumerate(months)]

# Plot the bar graph, using the values from months_orders and the quantity from the 'orders' variable
# months_pos = every 'Month' under the bar chart
# orders will be quantity we specified earlier

# Remember! it's X -> Y or Along the corridor, up the stairs!

plt.bar(months_orders, orders)
plt.xlabel("The last four Months of the Year")
plt.ylabel("Quantity of orders")
plt.title("A bar chart displaying the proportion of orders across the last Months of the Year")

# plt.xticks counts every value in "months_orders" to know how many Months there are to display
# plt.xticks then uses "months" to categorise the values from "months_orders"
plt.xticks(months_orders, months)

# Lets display the plot!
plt.plot()
```

Out[8]: []



Here we can see (in my opinion) a much preferable way of visualising the quantity of orders over September - December.
The code snippet is a little wordy and I understand is a bit more advanced then just plotting a line graph.
Essentially, what happens is:
1. We specify the style of plot for matplotlib to use, other wise it'd default to line plots
1. We provide the values for "Months" and "Quantities" in two arrays: `months` and `orders`
1. We then later use `months_orders` to count every value in those two arrays, so that the bar chart knows what it needs to plot!
1. We use `plt.bar` to plot the data using `months_orders` (this value will be 4 as it counted by `months_orders`) Say if there was another Month in "months" such as August, `months_orders` will now have a value of 5!
1. And then again, we provide the data stored in the `months` variable to `plt.bar`, so that any values displayed will have their respective Months displayed at the bottom!

2. Here we specify our title, x and y labels

1. Finally, we use our good friend plt.show() to output the bar chart
I hope I've explained that fairly well.

We can again customise our bar plot graph

Lets change the colors! Say we wanted blue...
I have removed the comments of this snippet, as it is exact same as the snippet as above. However, the only comment I've added is where we change the colour!

In [9]:

```
plt.style.use('ggplot')

months = ['September', 'October', 'November', 'December']
orders = [20, 5, 40, 50]

months_orders = [i for i, _ in enumerate(months)]

# Here we specify the color "blue"
# Much like in our line graphs, we specify this in the plt.plot
# Or in this instance, plt.bar as we're using a bar chart!
plt.bar(months_orders, orders, color='blue')
plt.xlabel('The last four Months of the Year')
plt.ylabel('Quantity of orders')
plt.title('A bar chart displaying the proportion of orders across the last Months of the Year')

plt.xticks(months_orders, months)


plt.plot()
```

Out[9]: []

And that's all folks!

I've showed you two types of plotting in Matplotlib, hopefully you've found this a nice introductory. I actively encourage you to experiment with both Pandas and Matplotlib yourself! Feel free to launch this instance again and create new Notebooks, or adapt those already provided.

For example:

- Adding more Months in the line chart
- Adding more Months in the bar chart
- Using a different use case of data, rather then order quantities from a website, why not users visiting a website?
- How about customising the plots even further? e.g. changing background colours, changing them to vertical bar graphs?
- Changing the data itself!

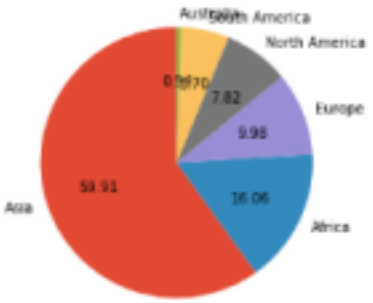
Below, I have modified an example of a [Pie Chart] from Matplotlib's Example library (https://-pythontic.com/visualization/charts/piechart)

Just to show you how versatile Matplotlib is.

We haven't even begun reading from datasets yet

In [10]:
```
# The slice names of a population distribution pie chart
pieLabels              = 'Asia', 'Africa', 'Europe', 'North America', 'South America', 'Australia'

# Population data
populationShare     = [59.69, 16, 9.94, 7.79, 5.68, 0.54]

figureObject, axesObject = plt.subplots()

# Draw the pie chart
axesObject.pie(populationShare,
        labels=pieLabels,
        autopct='%1.2f',
        startangle=90)

# Aspect ratio - equal means pie is a circle
axesObject.axis('equal')

plt.plot()
```

Out[10]: []



Finally:

If there is any questions, problems, feedback or features you'd like to see in the further rooms I hope to make for these topics, please get in touch!

I've made this room in the hopes that it provides a nice introductory into:

• What is Jupyter and why is it useful?

• A hopefully light introduction as to how it can be used. We've barely scraped the surface here, and hopefully - as I get time, I'd like to go more in-depth into these libraries and the Data Science topic as a whole.

I wouldn't consider myself an expert in the slightest, however I've spent the last 4-6 Months studying the topic & Machine Learning for my University Dissertation. Content will be released when and if I get the time. I'm super hectic, so I can't make any promises.

I'm in the TryHackMe Discord fairly frequently. I'd say my main method of contact is via Discord:

Discord: @CMNatic#9812