PUBS Day 2 – Sea Ice Extent Practical

1. Research Rationale

We know that as a result of climate change, the Arctic is experiencing warmer water temperatures which are contributing to increased levels of ice melting. The aim of this practical is to investigate a simple, real-life science question: "How is Arctic sea ice extent changing through time?". To do this we need to carry out the following tasks:

1. Calculate the extent of sea ice present by analysing sea ice concentration data from satellite imagery.
2. Repeat this same analysis for every successive January from the start of our dataset to the end of our dataset.
3. Use these extent values to identify if there is a trend in sea ice extent, determine the strength of that relationship, and create a plot of sea ice extent through time.

2. Data

The data we will be using is this workbook originates from the National Snow and Ice Data Centre who collate cryospheric (frozen) data from around the world. Specifically, the data we will be using represents sea ice concentration from their Sea Ice Index repository. This looks at the percentage of cover in a grid cell averaged over the entire month. We will be using the month of January to investigate winter ice extent and basing our analysis on data spanning from 1979 right through to 2021. For more information and to see the limitations of using this data see https://nsidc.org/sites/nsidc.org/files/G02135-V3.0_0.pdf#page=10.

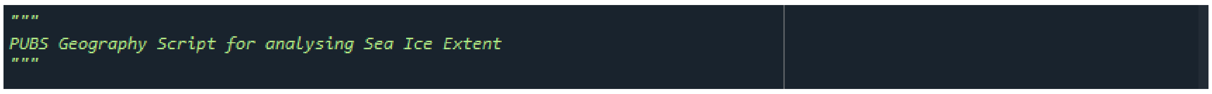
As a result, you will see in your data folder for day 2 that there are numerous '.tiff' files that we will be processing in order to answer our question of how sea ice extent is changing through time.

3. Let's code!

3.1. Project Setup and Initial Data Loading

First let's open up Spyder as we have done previously, using Search → Python (PUBS_Env). Let's first save this untiled file, navigate to your PUBS2021/Day_2 directory, saving this in the scripts section of the Arctic_Sea_ice_Entent folder as 'Sea_Ice_Analysis.py'.

It can be useful to write some information at the top of your script letting you know what it is used for. For example, at the top of my script I have written…



```
"""
PUBS Geography Script for analysing Sea Ice Extent
"""
```

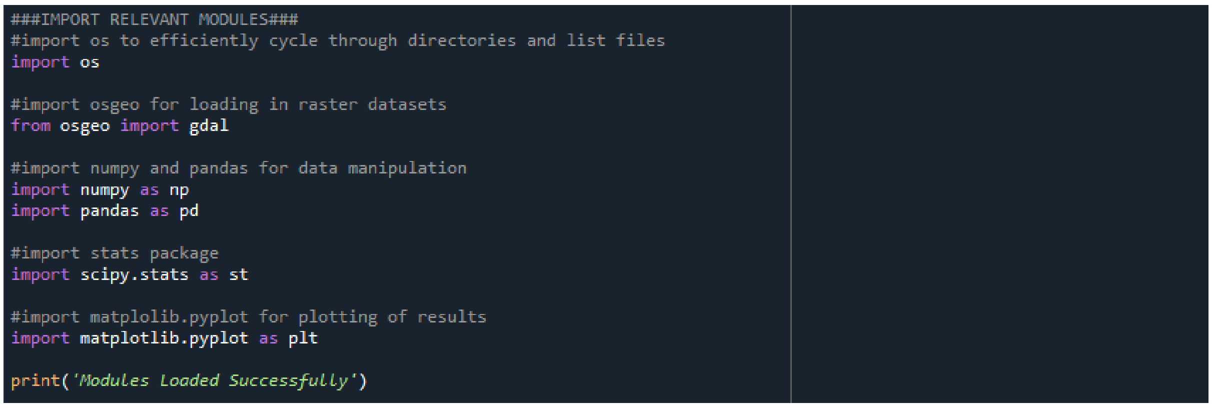
The use of triple quotation marks “”” above and below text ‘comments’ that text out, so python will not try to read it and execute commands. This means it is very helpful for writing longer pieces of text which describe what is going on in your script, to either remind you or to inform a new user. The other way to ‘comment out’ text is to use the # key. This will stop any text being executed to the right hand side of the # key This is commonly used to help explain to the user what the script is trying to do, either by providing headers, or in some instances, explanation line by line.

3.1.1. Import Modules

Let’s now import the modules we will need which contain the functions required to perform the analysis we want to do. Remember from yesterday we use the function ‘import’ followed by the module name to import the module.

Package	Description
import os	os is used to navigate your file structure (windows explorer)
from osgeo import gdal	gdal is the geospatial module we will be using
import numpy as np	numpy allows us to perform mathematical calculations
import pandas as pd	pandas is great for creating dataframes (a.ka. python tables)
import scipy.stats as st	stats is a well-used statistical package
import matplotlib.pyplot as plt	matplotlib is a plotting package to make professional plots

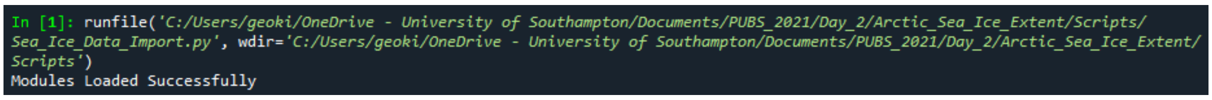
You may want to include a similar explanation as below within your .py file, by using the # key above where you enter the data. For example, the first section it is useful to complete is importing relevant modules.



```
###IMPORT RELEVANT MODULES###
#import os to efficiently cycle through directories and list files
import os

#import osgeo for loading in raster datasets
from osgeo import gdal

#import numpy and pandas for data manipulation
import numpy as np
import pandas as pd

#import stats package
import scipy.stats as st

#import matplolib.pyplot for plotting of results
import matplotlib.pyplot as plt

print('Modules Loaded Successfully')
```

Finally, to make sure we know that the modules have all been loaded, let's include a print statement to tell us. Type print('Modules Imported Successfully') at the end of your module import code.

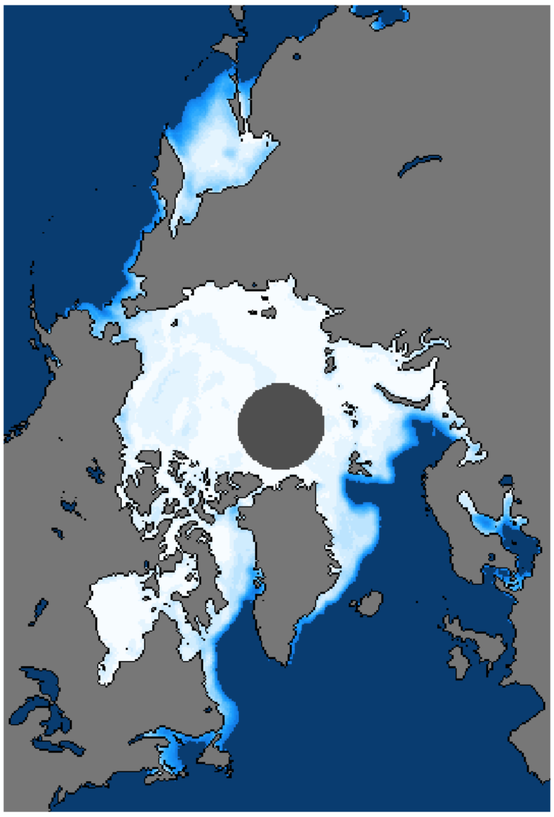
That's our initial project setup. We have a .py file with our modules ready to be loaded so we can start to do some data analysis. Let's check that everything is working properly at this stage. Use the run command (green play arrow) or press f5. You should see a similar output to below in your console, but with your own filepath.



```
In [1]: runfile('C:/Users/geoki/OneDrive - University of Southampton/Documents/PUBS_2021/Day_2/Arctic_Sea_Ice_Extent/Scripts/
Sea_Ice_Data_Import.py', wdir='C:/Users/geoki/OneDrive - University of Southampton/Documents/PUBS_2021/Day_2/Arctic_Sea_Ice_Extent/
Scripts')
Modules Loaded Successfully
```

3.1.2. Import Data

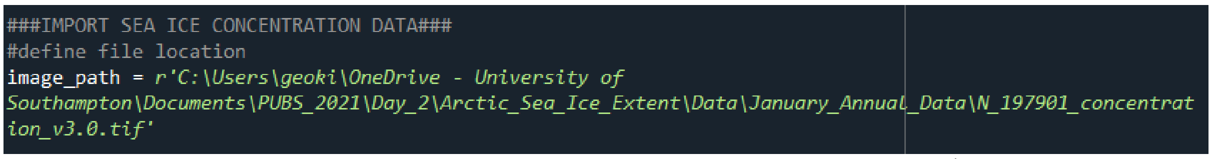
Let's load some data in!!



What we have currently in our data folder is a series of '.tiff' files showing sea ice concentration across the Arctic. Normally, you would be used to opening up such files in ArcGIS or similar GIS software, and if you were to do this you would see the data looks like this (see right). It is a raster showing the sea ice concentration in a series of 25*25 km grid squares. When you first download data, opening it up in a format you recognise can be really useful to get a feel for what the data looks like, or find any nuances within the data you have to be aware of, which in this case we have!! As you can see, we have a colour scheme for the sea ice concentrations; land is coloured a mid-grey and a circle in the middle is coloured in a dark grey. This centre circle represents an area that is not monitored by satellites and as such is given a set value (which we can reasonably assume has significant ice coverage). This initial data is from 1979, and as we progress through time this circle of no data reduces in size.

So, now we know we are working with raster data we can load this in to python. We do that using the GDAL module we imported at the start of our script. But first let's create a variable which is assigned to the filepath of this particular dataset. Remember, you must find the filepath for your unique file structure, you can't simply copy and paste the code below as it won't work!! Type the following, **replacing the <filepath> with your own data folder filepath**:

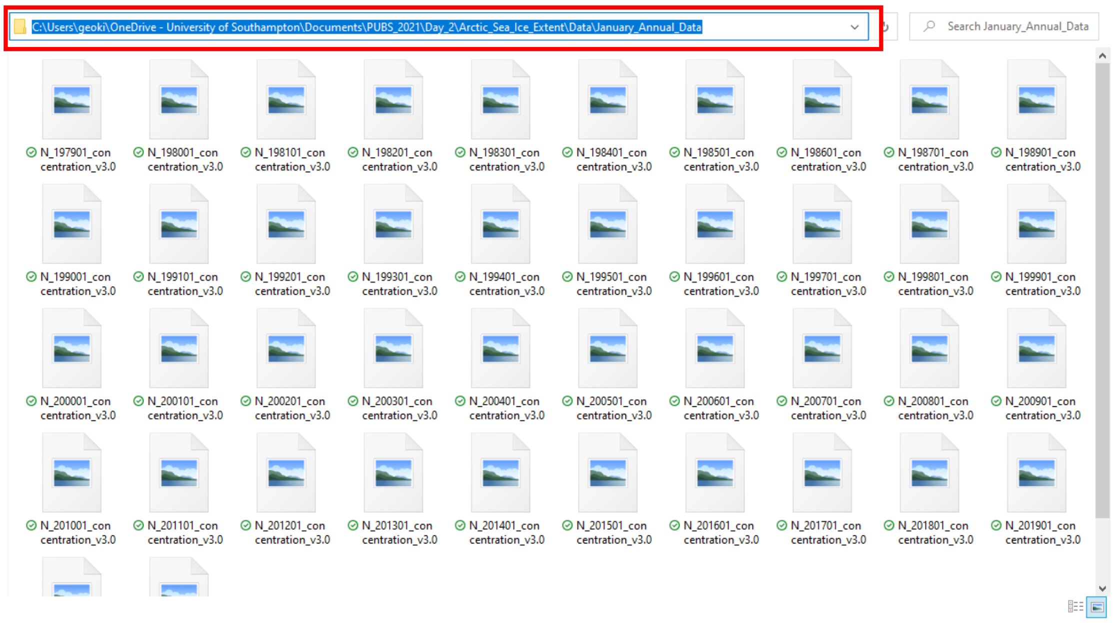
image_path = r'<filepath>\N_197901_concentration_v3.0.tif'



```
###IMPORT SEA ICE CONCENTRATION DATA###
#define file location
image_path = r'C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_2021\Day_2\Arctic_Sea_Ice_Extent\Data\January_Annual_Data\N_197901_concentration_v3.0.tif'
```

The r in the above statement tells python to read the following backslashes (\) as forward slashes (/) when searching for the file.

To copy your filepath to the .py requires two quick steps (unless you wish to type the whole thing out…). First navigate to your data folder in windows explorer, click in the **file bar** and copy this across, replacing the <filepath> in the above code. Then either type or copy across the filename of the first dataset in your folder which should be for 1979.



C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_2021\Day_2\Arctic_Sea_Ice_Extent\Data\January_Annual_Data

Search January_Annual_Data

```
N_197901_concentration_v3.0  N_198001_concentration_v3.0  N_198101_concentration_v3.0  N_198201_concentration_v3.0  N_198301_concentration_v3.0  N_198401_concentration_v3.0  N_198501_concentration_v3.0  N_198601_concentration_v3.0  N_198701_concentration_v3.0  N_198901_concentration_v3.0
N_199001_concentration_v3.0  N_199101_concentration_v3.0  N_199201_concentration_v3.0  N_199301_concentration_v3.0  N_199401_concentration_v3.0  N_199501_concentration_v3.0  N_199601_concentration_v3.0  N_199701_concentration_v3.0  N_199801_concentration_v3.0  N_199901_concentration_v3.0
N_200001_concentration_v3.0  N_200101_concentration_v3.0  N_200201_concentration_v3.0  N_200301_concentration_v3.0  N_200401_concentration_v3.0  N_200501_concentration_v3.0  N_200601_concentration_v3.0  N_200701_concentration_v3.0  N_200801_concentration_v3.0  N_200901_concentration_v3.0
N_201001_concentration_v3.0  N_201101_concentration_v3.0  N_201201_concentration_v3.0  N_201301_concentration_v3.0  N_201401_concentration_v3.0  N_201501_concentration_v3.0  N_201601_concentration_v3.0  N_201701_concentration_v3.0  N_201801_concentration_v3.0  N_201901_concentration_v3.0
```

Now let's read the data in. We will use the command gdal.Open() here to read in our variable image whose location/filepath we just defined. We will also assign this data that we are reading in to a new variable too, let's call it ras_data (shorthand for raster dataset). In order to not accidently edit the original dataset, we will also use the term gdal.GA_ReadOnly, which tells gdal not to modify the original data.

ras_data = gdal.Open(image_path, gdal.GA_ReadOnly)

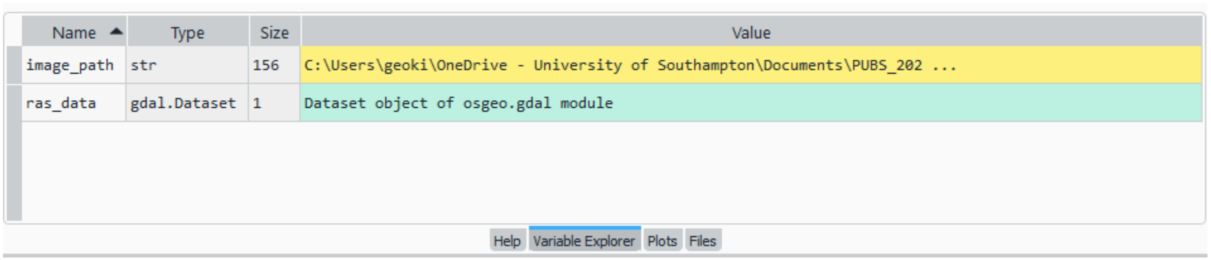
New variable of our raster data

Use the command gdal.Open()

Specify our image

Make sure it is only in read only mode

Let’s run our code again and see what happens. We can see that although we have added new bits of code, not much has changed as it still only says Modules Loaded Successfully… However, if we look in our variable explorer tab we will now see 2 new variables have been created. This is a good sign as it means our script is currently doing as it should.

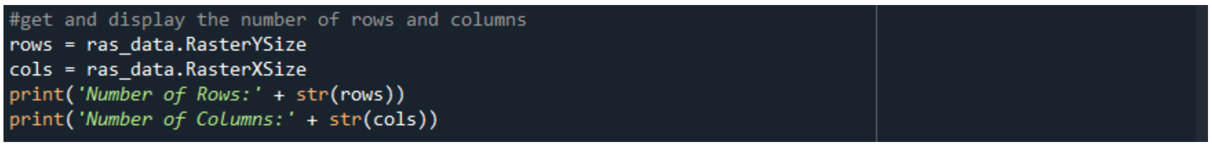


Name	Type	Size	Value
image_path	str	156	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
ras_data	gdal.Dataset	1	Dataset object of osgeo.gdal module

3.1.3. Understanding the Data

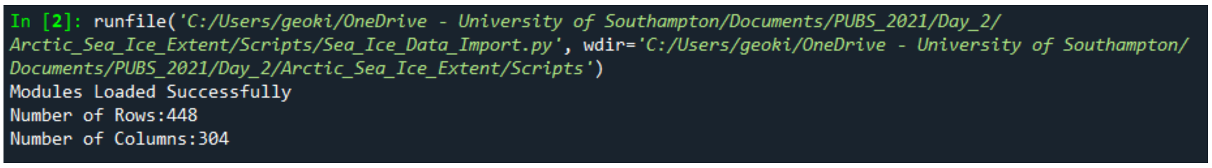
Now let’s find out some information about our raster data, like how many rows and columns it has. To do this we can make use of in built GDAL functionality. Type the following:

rows = ras_data.RasterYSize
cols = ras_data.RasterXSize
print('Number of Rows:' + str(rows))
print('Number of Columns:' + str(cols))



```
#get and display the number of rows and columns
rows = ras_data.RasterYSize
cols = ras_data.RasterXSize
print('Number of Rows:' + str(rows))
print('Number of Columns:' + str(cols))
```

What this is doing is saying, using our ras_data dataset, tell me the size in a certain direction, and assign it to the variable rows or cols. Therefore, rows is the size (number of rows) in the Y direction of our data and so on. Let’s run the script again, we should see the number of rows and columns printed as an output.



```
In [2]: runfile('C:/Users/geoki/OneDrive - University of Southampton/Documents/PUBS_2021/Day_2/
Arctic_Sea_Ice_Extent/Scripts/Sea_Ice_Data_Import.py', wdir='C:/Users/geoki/OneDrive - University of Southampton/
Documents/PUBS_2021/Day_2/Arctic_Sea_Ice_Extent/Scripts')
Modules Loaded Successfully
Number of Rows:448
Number of Columns:304
```

Tip: If you get these numbers back to front, you may have incorrectly assigned your rows and columns!

3.1.4. Creating an Array

The raster data, which is currently in its GDAL format, needs to be converted into a format that we can manipulate within python with greater ease. For this we are going to create an ‘array’. An array is similar to a raster whereby a grid or numbers is used to represent values at different points, so in our case we have a grid of sea ice concentration. You may be wondering why we need an array if we already have our raster, effectively they look the same right? Not to python, so we need to use an array to alter raster data.

The first step to create an array from a raster, is to identify the layer of the raster we want to use[1]. In this instance, our raster only has one layer, but we still need to specify this for us to turn it in to an array. To do this we will create a variable called ras_data_layer which will hold this first raster band:

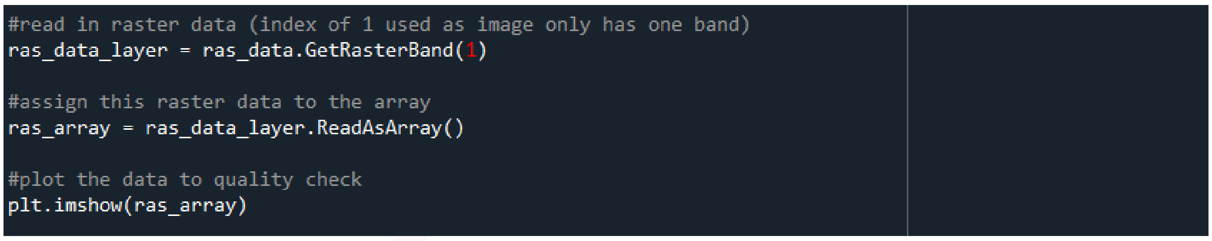
ras_data_layer = ras_data.GetRasterBand(1)

This is saying from our raster data, get the first raster band (denoted by the one in this instance) and assign it to the variable ras_data_layer. We will then read this data as an array and assign it to the variable ras_array:

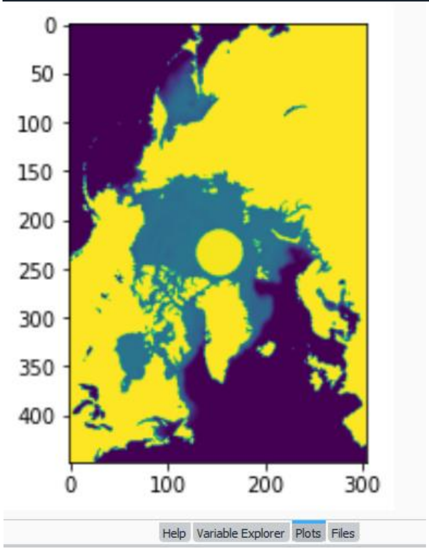
ras_array = ras_data_layer.ReadAsArray()

In these two steps we go from our original raster dataset, select the band we want to use, and then read it in as an array. The final step for this initial processing is to check the data looks like we think it should, so we are going to plot it. Type:

plt.imshow(ras_array)



```
#read in raster data (index of 1 used as image only has one band)
ras_data_layer = ras_data.GetRasterBand(1)

#assign this raster data to the array
ras_array = ras_data_layer.ReadAsArray()

#plot the data to quality check
plt.imshow(ras_array)
```



This is using matplotlib, to plot the array as an image, using the cell value to define the colour. Now when we run our script, we should get a plot appear (next tab along from variable explorer) showing us our map. Depending on your default colours your map may look different, but from this we can clearly see land, sea, ice, and the area of no data.

This is the first step in to reading in raster data before going on to further analysis. All this in a couple of lines of code too!

[1] Raster data can contain multiple layers of data, e.g. colour image rasters contain 3 layers, or bands, made up of Red Green and Blue (RGB).

3.2. Calculating Extents and Automating the Process

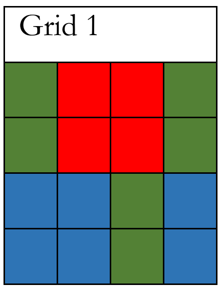
3.2.1. Manipulating Our Array To Calculate Extent

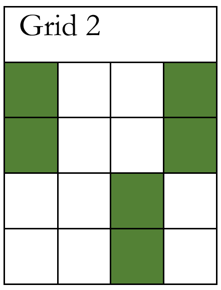
Our first task is to work out the extent of sea ice in the image. We will do this by calculating the area within the raster containing sea ice, but before we can do this, we need to exclude any areas which do not contain a significant level of sea ice, *e.g.* areas covered by land, or areas with a low percentage % of sea ice concentration. We will classify these areas as 0 so that they are discounted from the analysis. We will also need to 'add back in' the large area of no data over the north pole.

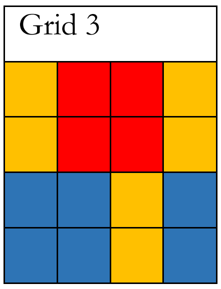
Firstly, we want to only choose cells that have a good proportion of sea ice to calculate the extent, for this practical we will use the 15% value based off of the NDIDC guidance, so that any cells containing less than 15% sea ice are omitted. For our data 0-100% is represented by 0-1000, so we will first set values less than 150 (15%) to 0.

ras_array[ras_array < 150] = 0

What this line of code is doing is creating a mask and applying a value to it. The grids below are there to help us visualise what this statement is doing. Grid 1 is the data we are starting with, a series of coloured in squares, and Grid 2 is this same data with a condition applied to it (only green squares selected). We then are going to edit the squares in the first grid, where the condition is met (so only changing green squares), to a new colour of orange. This results in us having a new grid (Grid 3) where all the original colours are the same apart from those that were green which have now changed to orange. In the above format, this would look like **grid[grid == green] = orange.**







So for our sea ice dataset, the input array, mask, and subsequent array will look similar to grids 4, 5, and 6 below, but on a bigger scale at 448 rows and 304 columns! We go through the same process, selecting those values that are < 150, and replacing them with the value of 0. This updates the original array, so we do not need to assign it to a new variable.

Grid 4			
151	202	200	812
78	154	450	810
335	53	320	690
178	235	34	14

Grid 5			
78			
	53		
		34	14

Grid 6			
151	202	200	812
0	154	450	810
335	0	320	690
178	235	0	0

In our sea ice data, land masses are identified as value 2540 and land boundaries as 2530, and as such we need to set these to 0.

```
ras_array[ras_array == 2540] = 0
```

```
ras_array[ras_array == 2530] = 0
```

To account for the area of land not mapped, which is given the code 2510, we will assume these to be complete sea ice in this scenario, we will assign this a value of 1000 (100%).

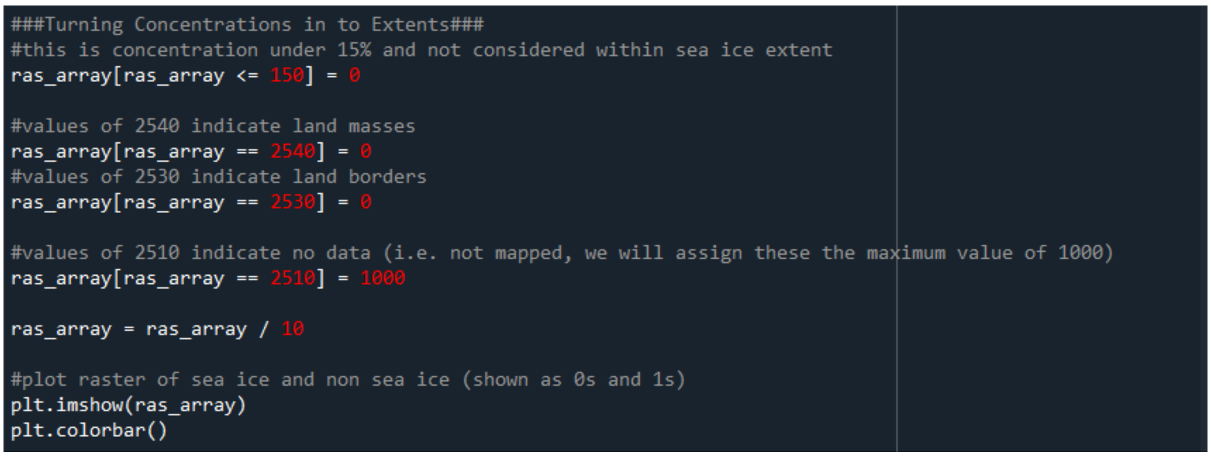
```
ras_array[ras_array == 2510] = 1000
```

Finally, we will rescale our data which goes from 0 to 1000 to 0 to 100 by dividing our array by 10 using the following code. This simply takes every value in our array, and divides it by 10.

```
ras_array = ras_array / 10
```

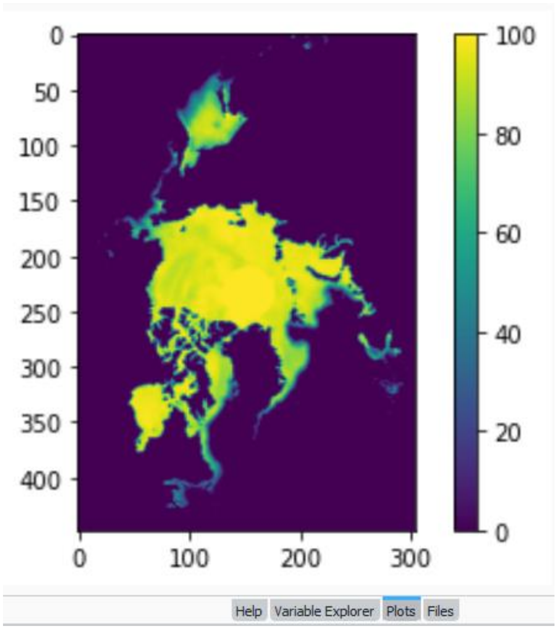
We can then view this data in the same way we did previously using the imshow() function. The plt.colorbar() function displays a scale bar alongside our plot for reference.

```
plt.imshow(ras_array)
plt.colorbar()
```



```
###Turning Concentrations in to Extents###
#this is concentration under 15% and not considered within sea ice extent
ras_array[ras_array <= 150] = 0

#values of 2540 indicate land masses
ras_array[ras_array == 2540] = 0
#values of 2530 indicate land borders
ras_array[ras_array == 2530] = 0

#values of 2510 indicate no data (i.e. not mapped, we will assign these the maximum value of 1000)
ras_array[ras_array == 2510] = 1000

ras_array = ras_array / 10

#plot raster of sea ice and non sea ice (shown as 0s and 1s)
plt.imshow(ras_array)
plt.colorbar()
```

We can see from the outputted plot of the array that we have successfully isolated areas which are sea ice within the arctic and can display the concentration. This means that we can now begin to calculate the extent of area covered by sea ice for January 1979.



It's important to think what may have affected our sea ice extent, for example the threshold we use to determine where sea ice is present. What would the extent look like if we used all values over 0, or over 250 (25%)? Have a play with some separate values to see the differences.

Now the final step is to calculate the extent of sea ice that is present. To do this we want to times the fraction of sea ice coverage per cell, by the cell size. For example, if we had 50% coverage of a 625 km^2 cell (25*25 km), we would want the output to be 312.5 km^2. Therefore, we need to rescale our data from 0-100% for display, to 0-1 for calculations.

ras_array = ras_array / 100

Note: this is not a scientifically correct method, but for the purposes of this exercise gets aims to add a complexity to improve your python understanding.

Following on from this we need to define our cell size, which we know is 25 * 25 km, before we can then multiple the two together to get an extent of sea ice in km^2 for each cell of our array.

cell_size = 25*25

ras_array = ras_array * cell_size

We can then sum all of the cell values together to provide us with an overall figure for sea ice extent for January of 1979. We can then print this to an output message to say the extent of sea ice in this year was…

total_area = np.nansum(ras_array)

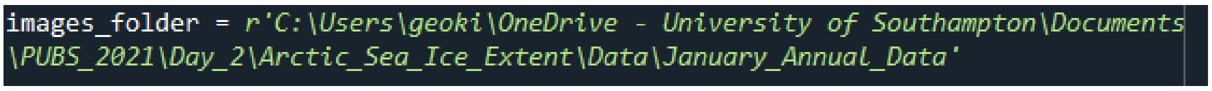
print('Sea Ice Extent Equal To: ' + str(total_area))

3.2.2. Looping Through All Our Datasets

So far, we have taken a raster dataset, imported it in to python, and performed some analysis on it to isolate grid cells with sea ice extent over 15%. However, we probably could have done this in something such as a GIS package pretty easily and quickly. The power of python comes through looping over multiple datasets to perform the same calculations. This is what we shall do next, loop over all of these and produce outputs of extent for all of them.

The first step in being able to do this, is to create a list of all the files that we want to analyse. This will allow us to go through this list, analyse each dataset, and produce an output much quicker than doing each dataset individually. The more sets of data you have to analyse, the more useful a for loop becomes.

To create a list of files to go through, we will create a for loop similar, but slightly different, to the loops you made yesterday. In the script you are writing, lets head up to where you define your image_path variable, and change the name of this to image_folder. Next remove the filename at the end of the path and the backslash before it, so that you have a folder location rather than a file location defined.



```
images_folder = r'C:\Users\geoki\OneDrive - University of Southampton\Documents
\PUBS_2021\Day_2\Arctic_Sea_Ice_Extent\Data\January_Annual_Data'
```

Below this we can define a new, empty, list variable, which we will call filepaths. The purpose of this being that we will choose filepaths to append (add) to this variable for us to loop over later.

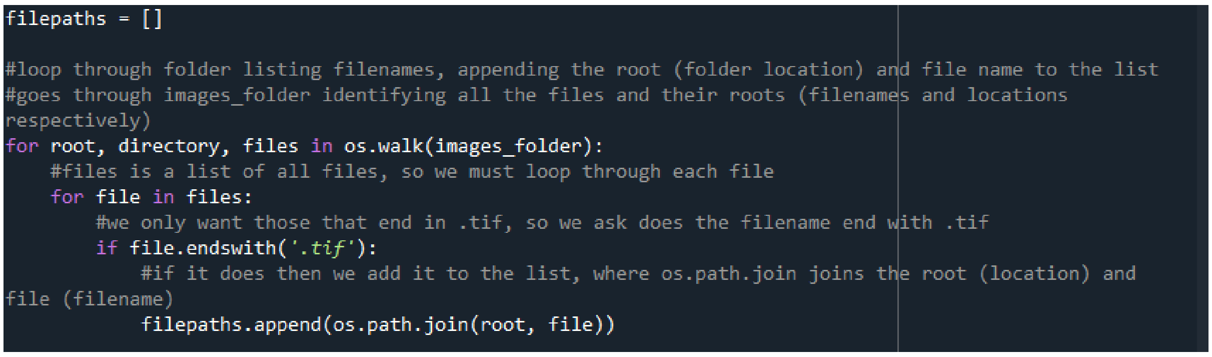
filepaths = []

Next we will loop through this folder, listing all the files that are within it. We use the function os.walk() which will return a list of the root path (where the folder we are searching is), directories (any folders within the folder we are looking at), and files (all the files within the folder we are searching. We then add in another for loop, which will go through each of these listed files, effectively going over each file within our folder individually.

We only want to use .tif files, and although in the data we have given you this is the case, in other cases you may have different files within your folder which you don't want to analyse, perhaps a word document containing your methods. To only use .tif files, we will use an 'if' statement where we ask if the file we are looping over ends in .tif, and perform an action on it if it does. In this case, we will append this file to the filepaths list we made earlier. What we append is not just the filename, but the whole filepath of this data, so for example instead of raster1.tif being appended, we append c:\data\raster1.tif.

To perform this whole loop, type the following. Remember, a for loop uses indentation to display different levels of the for loop.

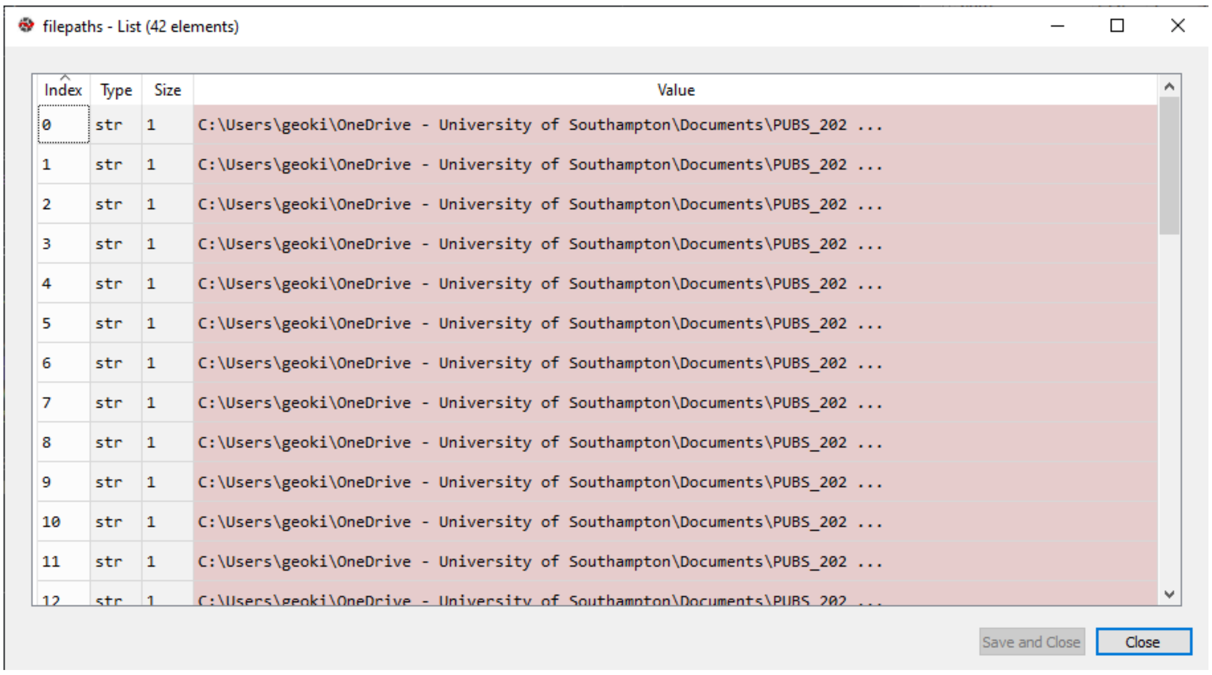
```
filepaths = []

for root, directory, files in os.walk(images_folder):
    for file in files:
        if file.endswith('.tif'):
            filepaths.append(os.path.join(root, file))
```



```
filepaths = []

#loop through folder listing filenames, appending the root (folder location) and file name to the list
#goes through images_folder identifying all the files and their roots (filenames and locations
respectively)
for root, directory, files in os.walk(images_folder):
    #files is a list of all files, so we must loop through each file
    for file in files:
        #we only want those that end in .tif, so we ask does the filename end with .tif
        if file.endswith('.tif'):
            #if it does then we add it to the list, where os.path.join joins the root (location) and
file (filename)
            filepaths.append(os.path.join(root, file))
```

Highlight and run the code you have just written, to only run the selected code use F9 instead of F5. The output of this is that our filepaths list should look like the below image, whereby if we open our filepaths variable in the variable explorer, we can see all of our files listed.

This means we can now use the filepaths variable to create a loop that goes over each file, performing the analysis we have previously used.


filepaths - List (42 elements)

Index	Type	Size	Value
0	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
1	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
2	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
3	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
4	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
5	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
6	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
7	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
8	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
9	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
10	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
11	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...
12	str	1	C:\Users\geoki\OneDrive - University of Southampton\Documents\PUBS_202 ...

First, let's highlight all of the code we have written from ras_data = gdal.Open() all the way to the end of our script where we print the sea ice extent statement. Press the tab button, and the whole block of code should indent. In the line above the ras_data variable being first defined, let's start our for loop by typing:

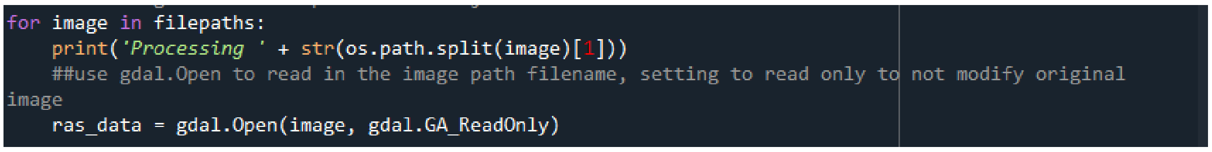
for image in filepaths:

This will initiate the for loop, whereby each loop will change the variable 'image' to correspond with one of the filepaths in our list we created previously. So we know which image is being processed, lets type a print statement on the next line. To make sure the indenting is lined up, press enter after the 'filepaths:' and this will automatically indent the code for you. Now lets type:

print('Processing ' + str(os.path.split(image)[1]))

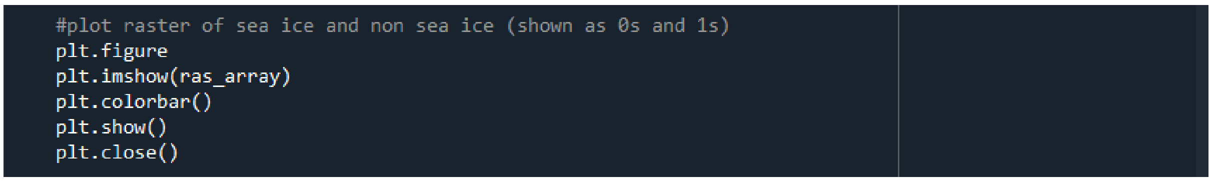
Although this is the for loop now set up, we need to make two small adjustments to the code we created previously. First, we can remove our 'image_path' variable, so delete this line of code which defines this. Next, in the first gdal.Open() statement, remove the '_path' after image so that we are left with:

ras_data = gdal.Open(image, gdal.GA_ReadOnly)


```
for image in filepaths:
    print('Processing ' + str(os.path.split(image)[1]))
    ##use gdal.Open to read in the image path filename, setting to read only to not modify original
image
    ras_data = gdal.Open(image, gdal.GA_ReadOnly)
```

The final adjustment we need to make is to alter the way we plot our maps. We need to make sure we close our plot each time we create it, otherwise they won't plot successively. Add the plt.figure, plt.show(), and plt.close() statements around our original plt.imshow(ras_array) and plt.colorbar() statements.

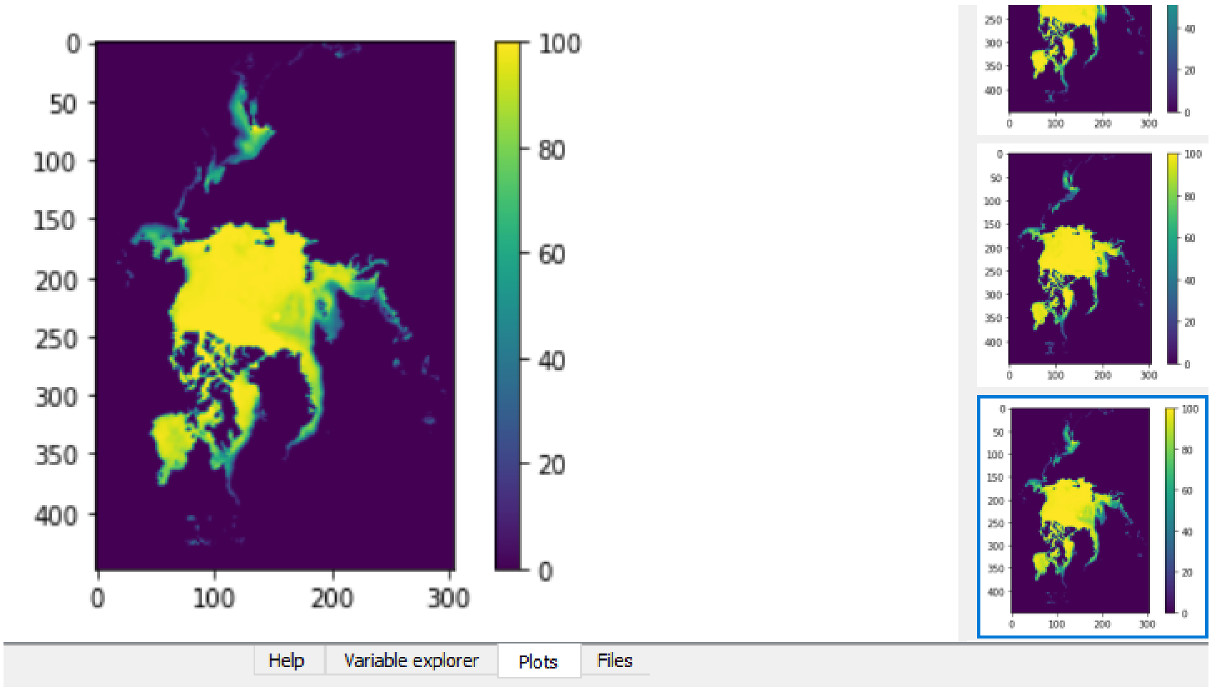
plt.figure
plt.imshow(ras_array)
plt.colorbar()
plt.show()
plt.close()



```
#plot raster of sea ice and non sea ice (shown as 0s and 1s)
plt.figure
plt.imshow(ras_array)
plt.colorbar()
plt.show()
plt.close()
```

We can now press run (F5) to see what happens! You should see in your console that we get pieces of information showing which image is being processed and also the sea ice extent for that year. Hopefully you can spot the trend that appears to be happening.

If you open up the plots pane, you can see all of the plots that have been created for each year. Scroll up to the first plot and use the down arrow on your keyboard to scroll through and see how sea ice coverage is changing through time.



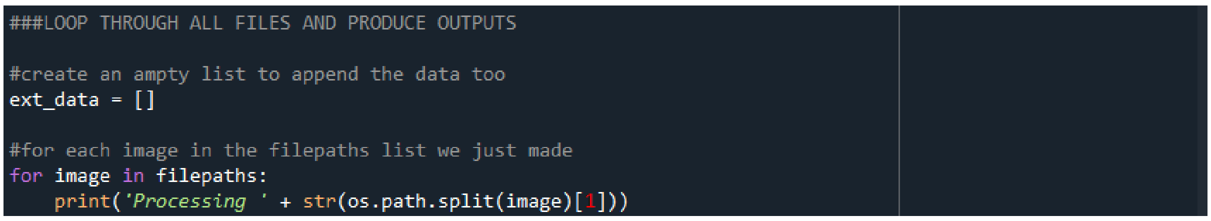
Excellent work, we have now got to a point where we have automatically worked through a series of raster datasets, processed them as arrays, and outputted some maps and stats to reveal how sea ice extent is changing.

3.3. Creating a Database of Results and Running Some Statistics

So we have managed to go through the data, and output some extent areas and create some cool maps. But what happens if we want to analyse the trend in our data? Is there a strong relationship between time and extent? Let's do some stats to find out!

Once again, we are going to build on what we learnt in day 1 by creating a database (a python table) of image year and calculated sea ice extent. Let's create an empty list for the moment, much like we did when appending filepaths for the processing loop. Let's call this variable ext_data (extent data) and place this just before your for loop that goes through all the images.

Ext_data = []



```
###LOOP THROUGH ALL FILES AND PRODUCE OUTPUTS

#create an ampty list to append the data too
ext_data = []

#for each image in the filepaths list we just made
for image in filepaths:
    print('Processing ' + str(os.path.split(image)[1]))
```

In order to add the year and extent to this, we must create a variable that pairs these two together and appends them to this empty list for every image that is processed.

First, we need to isolate the year of each image. As we know, our extent files have the year in them, so we can use this to create our year variable. So let's isolate the filename and then extract the year from this. We can use os.path.split() to isolate the filename from the path. We follow this with the [1] as os.path.split() returns a list of two elements, the path, and the filename. Remember, pythons indexing starts at 0, so the filepath is [0] and the filename is [1]. To do all this, type the following after our print statement that tells us our sea ice extent:

filename = os.path.split(image)[1]

Once we have the filename, to get the year we want to choose the characters from 3rd to 6th place, based on N_197901_concentration_v3.0.tif where we want to extract the 1979. This requires us to index **from index 2** (place 3) **up to but not including python index 6** (place 7).

Letters	N	_	1	9	7	9	0	1	_	C	O	N	C	E	N	T	R	A	T	I	O	N	…
Place	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	…
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	…

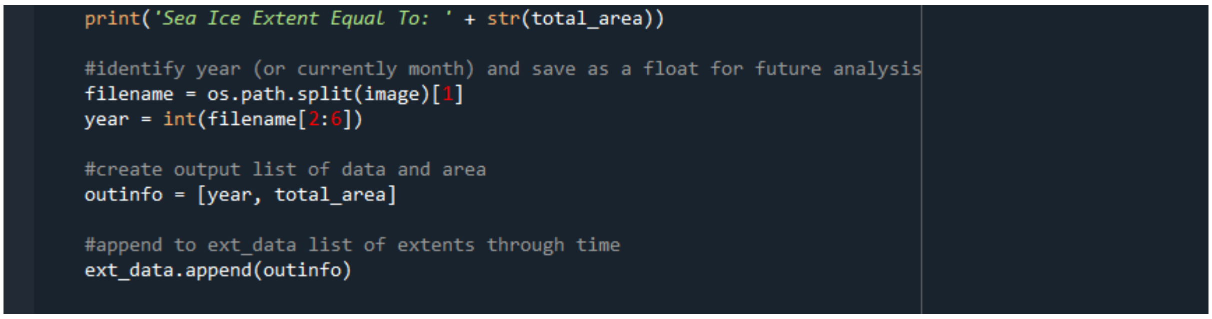
The final adjustment is to make sure this value is an integer (number) and not a string (text) as we want to perform statistical analysis on it, and we can't do that with text variables. To do this, type the following:

year = int(filename[2:6])

Now we will create a list of two values, year and total area, before appending this to our ext_data variable we created earlier.

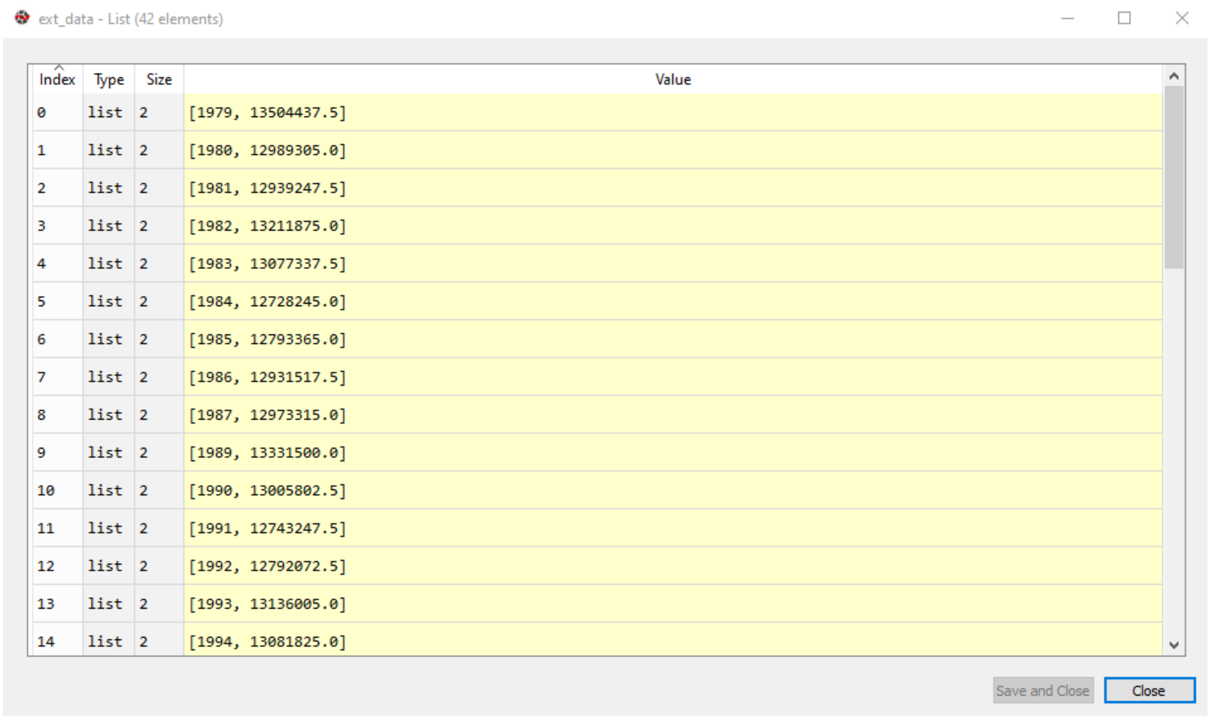
outinfo = [year, total_area]
ext_data.append(outinfo)

When we run our script now we will now end up with our ext_data variable being populated with these pairs of year and extent data, which we can then use to inform our statistical analysis.



```
    print('Sea Ice Extent Equal To: ' + str(total_area))

    #identify year (or currently month) and save as a float for future analysis
    filename = os.path.split(image)[1]
    year = int(filename[2:6])

    #create output list of data and area
    outinfo = [year, total_area]

    #append to ext_data list of extents through time
    ext_data.append(outinfo)
```

Now let's create a dataframe which stores all of these results. Currently, if we look at our ext_data we can see that it's not in a format that we can do anything with…



ext_data - List (42 elements)

Index	Type	Size	Value
0	list	2	[1979, 13504437.5]
1	list	2	[1980, 12989305.0]
2	list	2	[1981, 12939247.5]
3	list	2	[1982, 13211875.0]
4	list	2	[1983, 13077337.5]
5	list	2	[1984, 12728245.0]
6	list	2	[1985, 12793365.0]
7	list	2	[1986, 12931517.5]
8	list	2	[1987, 12973315.0]
9	list	2	[1989, 13331500.0]
10	list	2	[1990, 13005802.5]
11	list	2	[1991, 12743247.5]
12	list	2	[1992, 12792072.5]
13	list	2	[1993, 13136005.0]
14	list	2	[1994, 13081825.0]

So let's put this in to a format that we can do something with. On a new line at the end of your script (make sure you are outside the for loop *i.e.* your cursor is at the left of the screen), let's turn this data in to a dataframe using pandas. Call this new dataframe ice_ext, and specify the columns as 'Year' and 'Extent'. The pandas.DataFrame function takes an input, and turns it in to a table. In this instance, it will take the first value in each of our pairs and put them in a column, and the same with the second value. We have defined these columns as 'Year' and 'Extent' as we know this is the order we combined them in.

ice_ext = pd.DataFrame(ext_data, columns=['Year', 'Extent'])

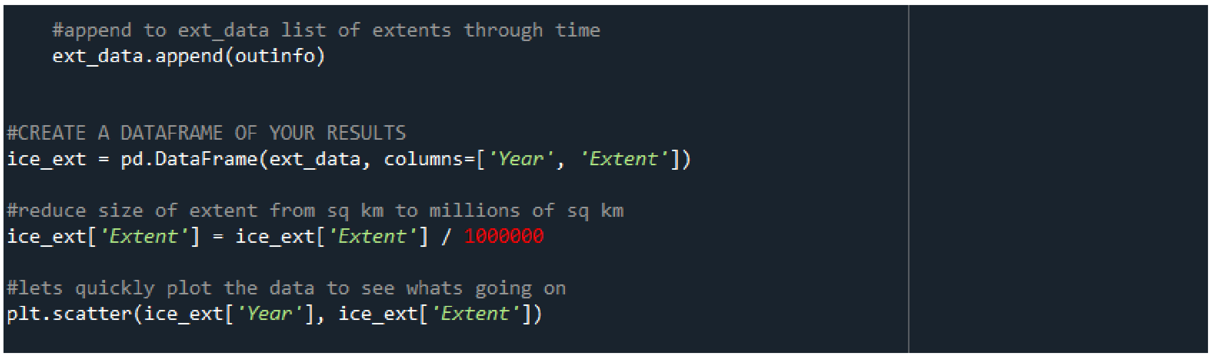
pd is used instead of pandas here because when we imported the pandas module we specified that we would implement it using the letters pd. This is a shortcut so each time we use it we don't have to type our pandas, the same goes for shorting matplotlib.pyplot to plt later on.

In order to make the numbers a bit easier to handle, we will also reduce the 'Extent' variable from sq km to millions of sq km, by dividing every cell in the extent column by 1,000,000.

ice_ext['Extent'] = ice_ext['Extent'] / 1000000

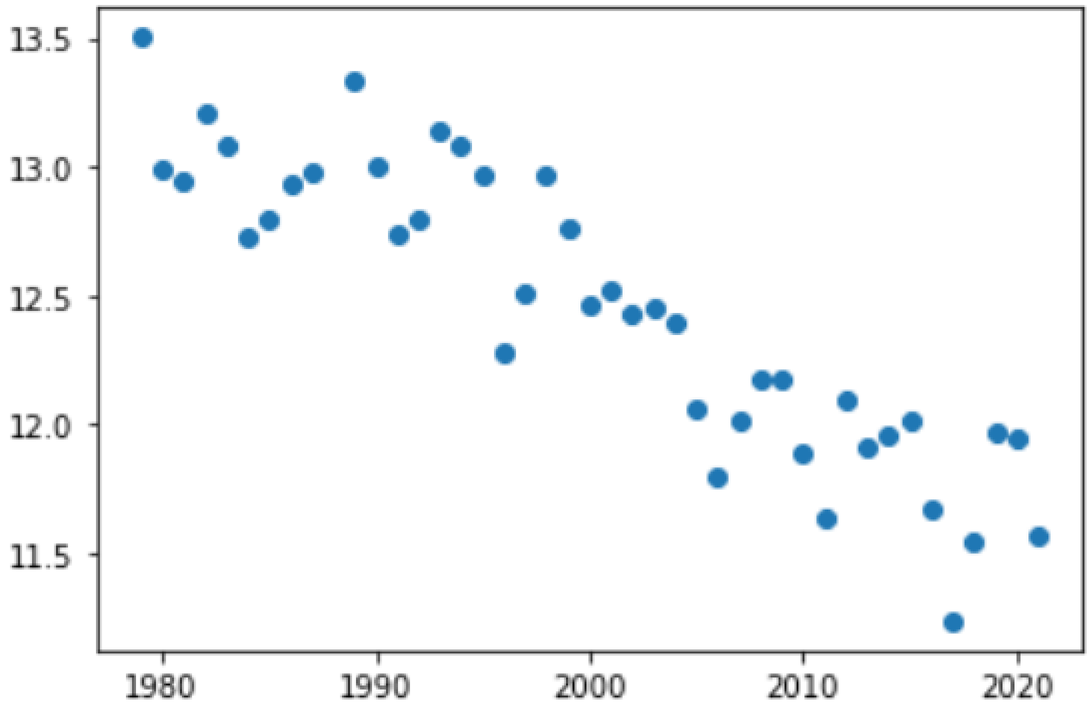
Let's quickly plot this data, just to have a look at what is happening before we go on to analyse the data and make more professional looking plots.

plt.scatter(ice_ext['Year'], ice_ext['Extent'])



```
    #append to ext_data list of extents through time
    ext_data.append(outinfo)


#CREATE A DATAFRAME OF YOUR RESULTS
ice_ext = pd.DataFrame(ext_data, columns=['Year', 'Extent'])

#reduce size of extent from sq km to millions of sq km
ice_ext['Extent'] = ice_ext['Extent'] / 1000000

#lets quickly plot the data to see whats going on
plt.scatter(ice_ext['Year'], ice_ext['Extent'])
```

Hopefully you should see a plot similar to below, where we can clearly see that ice extent is decreasing as time moves on. But how strong is this relationship?



In order for us to analyse this data we will make use of the scipy module which allows us to do scientific computing. We will perform a linear regression analysis of our year vs extent data to see how strong a relationship we have. The st.linregress() function returns multiple pieces information. We can use these pieces of information to inform how well our two variables are correlated.

Value	Meaning
Slope	The slope (m) value of the regression line.
Intercept	The intercept (c) value of the regression line.
R value	The correlation coefficient, strength of the regression, values closer to 1 or -1 indicate a stronger relationship
P value	How significant changes in one variable are in predicting another. If below 0.05 then the two variables are significantly related.
Standard Error	Average distance from samples to the regression line.

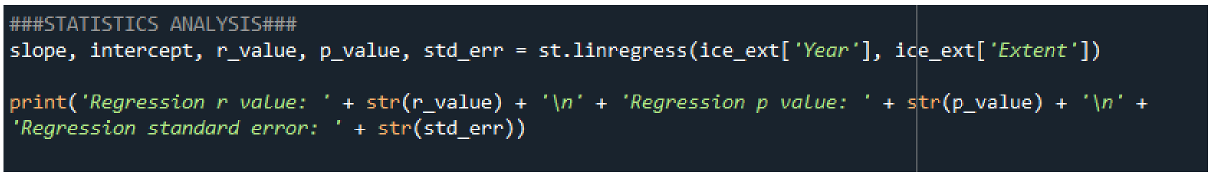
Let's use the linregress feature by typing the following:

```
slope, intercept, r_value, p_value, std_err = st.linregress(ice_ext['Year'], ice_ext['Extent'])
```

As per the above, all of the above values are produced by the linregress function, so we need to assign them to variables. As a result, we get 5 new variables each telling us about the relationship between our two variables.

Let's construct some print statements to tell us about our data. The '\n' variable here is telling python to introduce a line break so that the next part of the statement is on a new line.

print('Regression r value: ' + str(r_value) + '\n' + 'Regression p value: ' + str(p_value) + '\n' +'Regression standard error: ' + str(std_err))

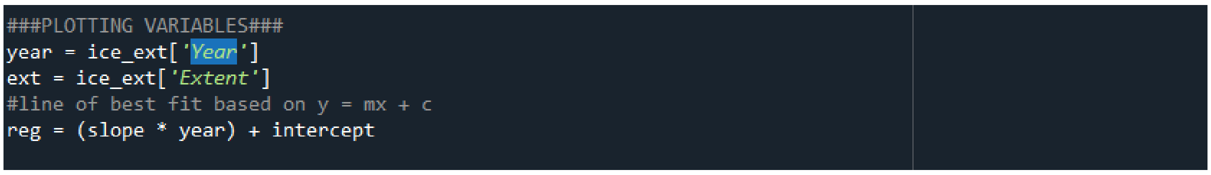


```
###STATISTICS ANALYSIS###
slope, intercept, r_value, p_value, std_err = st.linregress(ice_ext['Year'], ice_ext['Extent'])

print('Regression r value: ' + str(r_value) + '\n' + 'Regression p value: ' + str(p_value) + '\n' +
'Regression standard error: ' + str(std_err))
```

We can see form our output in the console that we have a strong negative correlation (-0.90), a significant p-value (2.98e-16), and a low standard error of 0.0030. So we can definitely say that sea ice extent is decreasing through time with a high degree of confidence.

The final part before we create a nice plot is to create our variables for plotting. We will create a variable for the time, the extent, and also a regression line (line of best fit). We can use the formulae y = mx + c for us to work out for each year where the line of best fit would be. We know m and c, these are the slope and intercept we calculated earlier in the linear regression, and we are applying this for each x value (year) that we have.

year = ice_ext['Year']
ext = ice_ext['Extent']
reg = (slope * year) + intercept



```
###PLOTTING VARIABLES###
year = ice_ext['Year']
ext = ice_ext['Extent']
#line of best fit based on y = mx + c
reg = (slope * year) + intercept
```

Before we move on to the next stage of the course, lets clean up our script quickly by getting rid of the plot statement we just wrote to quickly view the data. So remove the line 'plt.scatter(ice_ext['Year'], ice_ext['Extent'])'.

3.4. Plotting Graphs

For plotting, we will use the module matplotlib, which is the most common but not only plotting package in python. It is well documented (https://matplotlib.org/) and easy to find how to add to or adjust your plots. First we defining two elements of our plots, the figure and axis variables. We do this by typing the following:

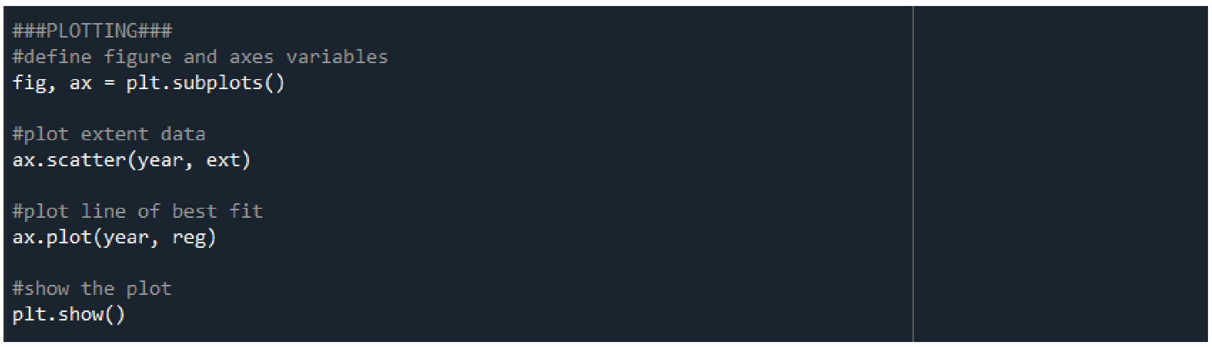
fig, ax = plt.subplots()

The subplots command returns an overall figure, and a set of axis. If you want to have multiple sets of axes on one plot, you can use subplots to specify this. Don't worry about this for now, but if you want to take this further you can google plt.subplots() to find out more.

When adjusting elements that are plotted on the axes, or editing the axis themselves, we use the ax variable. So let's plot our variables using this. The first type of plot will be a scatter plot for our time and extent variables. Then we will plot our line of best fit using the normal plot() function which will plot a line, perfect for creating a line of best fit. Then to see the plot we will use the function show(). This will display it in our plot area as previously.

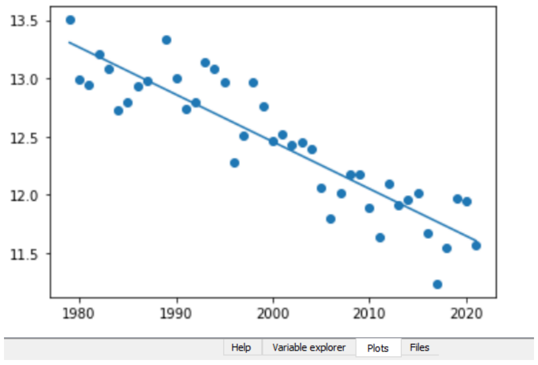
Tip: Instead of running the whole script each time, just highlight your plotting lines of code and press F9!

fig, ax = plt.subplots()
ax.scatter(year, ext)
ax.plot(year, reg)
plt.show()



```
###PLOTTING###
#define figure and axes variables
fig, ax = plt.subplots()

#plot extent data
ax.scatter(year, ext)

#plot line of best fit
ax.plot(year, reg)

#show the plot
plt.show()
```

This is the simplest of plots, where we have had no control over how the output will look. However, one of the benefits of python is creating plots of publishable quality. So let's explore some of the things we can do to make these plots look a bit more professional.



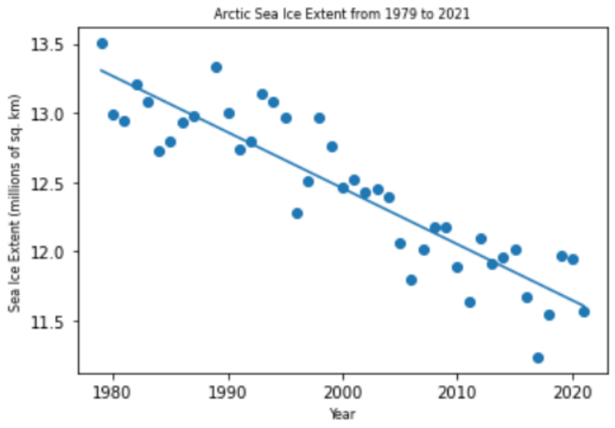
We currently have no idea what these axes represent, so let's add some titles to them. To start this process, many of our elements may use a fontsize argument to determine how large the font is and to be consistent, and not have to change all the numbers each time we want to change them, we will define the fontsize as a variable. Let's start with a fontsize of 8. Below your fig, ax line type the following.

fs = 8

Now let's include some titles. We will call our x axis 'Year' and our y axis 'Sea Ice Extent (millions of sq.km)'. At the same time, we will add a title, the title is referenced by the figure, so instead of using ax.function we will use fig.function. It is important you type the following before the plt.show() function otherwise they won't be included. Plt.show() should always be the last function you call. Type the following after the lines where you plot your data.

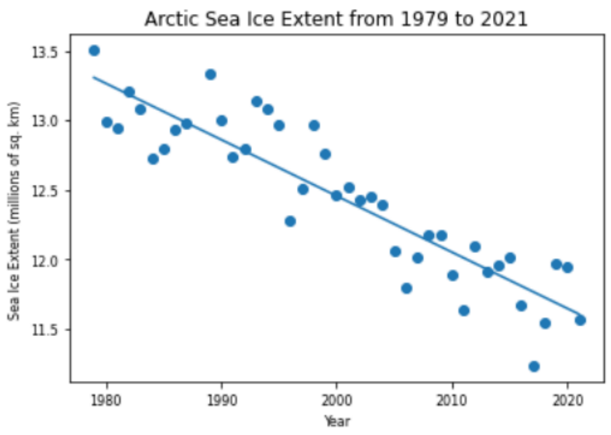
ax.set_xlabel('Year', fontsize = fs)
ax.set_ylabel('Sea Ice Extent (millions of sq. km)', fontsize = fs)

plt.title('Arctic Sea Ice Extent from 1979 to 2021', fontsize = fs)



This is better, but everything doesn't look quite in proportion. So let's make the title 1.5 times as big as the other labels. We will also adjust the axes tick mark labels (the numbers along the sides) to be smaller too. To make the fontsize of the title 1.5 times as big, where we had previously typed fontsize = fs, replace the fs with (fs * 1.5). That way if we change the size of the axis labels, the title font will always be proportionally 1.5 times as big. We can also add the lines below after setting our axis labels to adjust the tick fontsize.

plt.xticks(fontsize= fs)
plt.yticks(fontsize= fs)



Now the only thing to adjust is the plot contents. Currently they look a bit oversized compared to the rest of the graph, you could almost say they look like they were made in excel… so let's fix that. Alter the ax.scatter() command so it reads as follows:

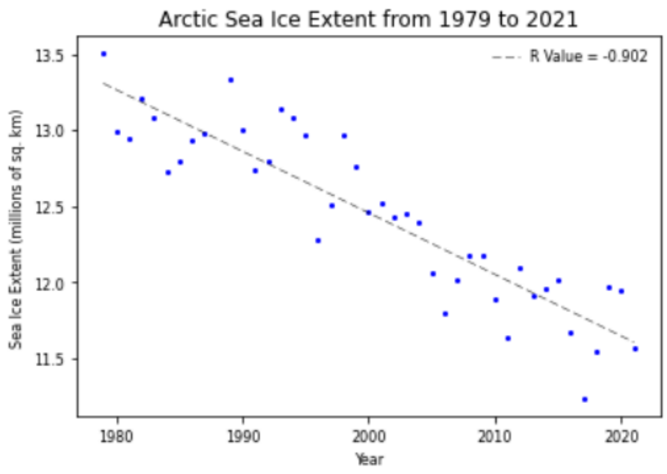
ax.scatter(year, ext, s = 4, color = 'blue')

This is setting the size of dots to 4 point, and the colour to stay as blue. For the regression line we can be slightly more creative. First we can decide the linewidth and the style of that line. In this instance we will create a dashed line ('- -') with a width of 0.5. We will set the colour to black, and give it a label. The label will be based off the r value of our linear regression stats. So add on to your previous ax.plot() function by typing:

ax.plot(year, reg, linewidth = 0.5, linestyle = '--', color = 'black', dashes = [10, 5], label = 'R Value= ' + str(r_value) [0:6])

The label indicates to matplotlib to add this to the legend which we will plot below. The use of str() turns the r_value in to a string to plot as text, with the [0:6] indicating we only want the first 6 characters which with the negative sign equals 3 decimal places rather than the full 10 or so from the whole r_value. To add the label to the plot, type the following where frameon simply refers to not having a box around the legend:

```
plt.legend(frameon = False, fontsize = fs)
```



The final element we may want to change, is to remove the right and top borders. Each border can be adjusted individually, so we must 'turn off' the right and top borders. For this we type the following:

```
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
```

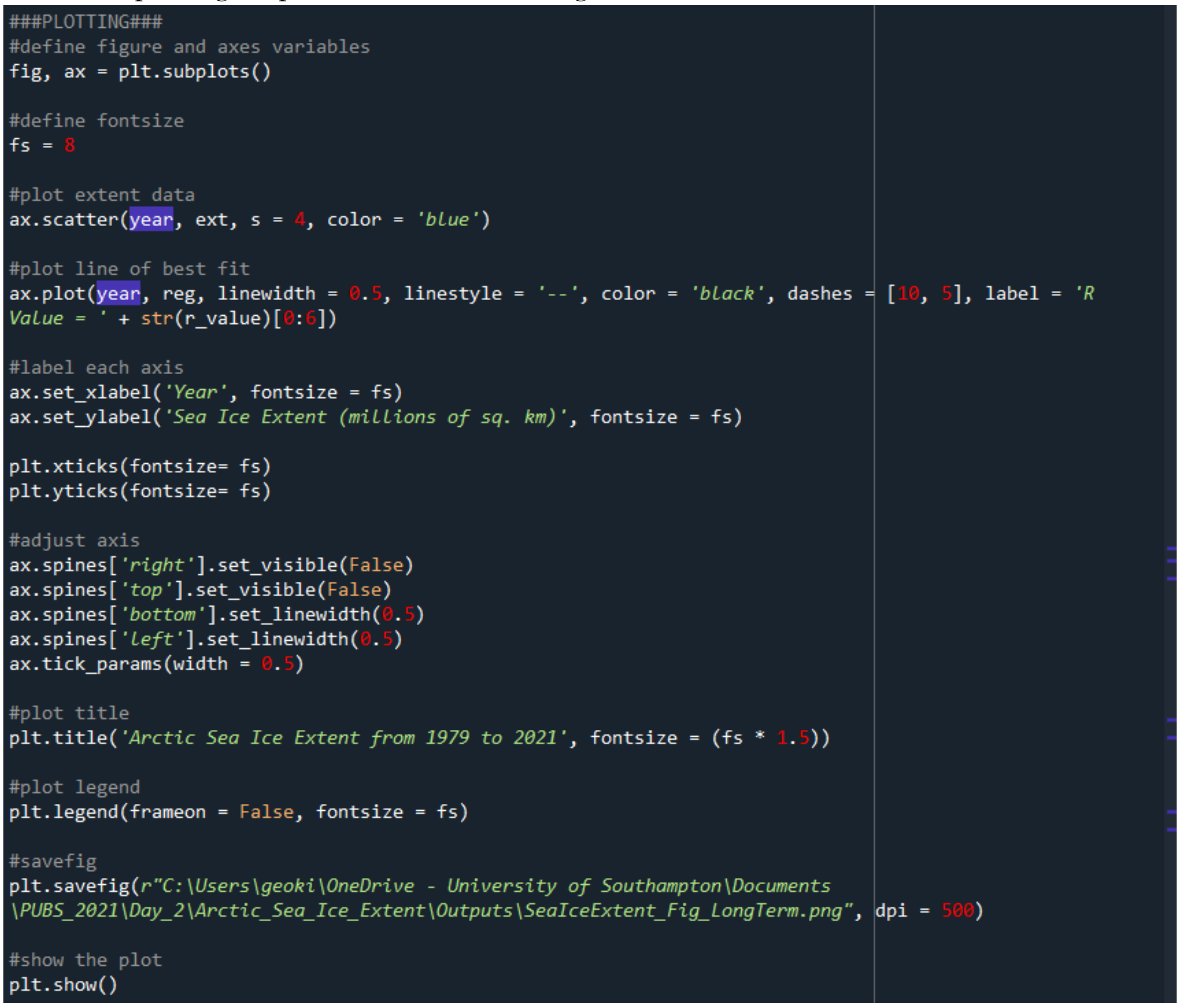
On top of this, the remaining borders can seem quite prominent, to reduce this we will make them and the tick marks thinner. Use the below code to do this.

```
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['left'].set_linewidth(0.5)

ax.tick_params(width = 0.5)
```

The final element, is to save the plot for use in reports as high quality figures. To do this, above the plt.show() command we will use the below plt.savefig(). The dpi specifies the quality of the output. Anything over 300 dpi should produce suitable quality figures.

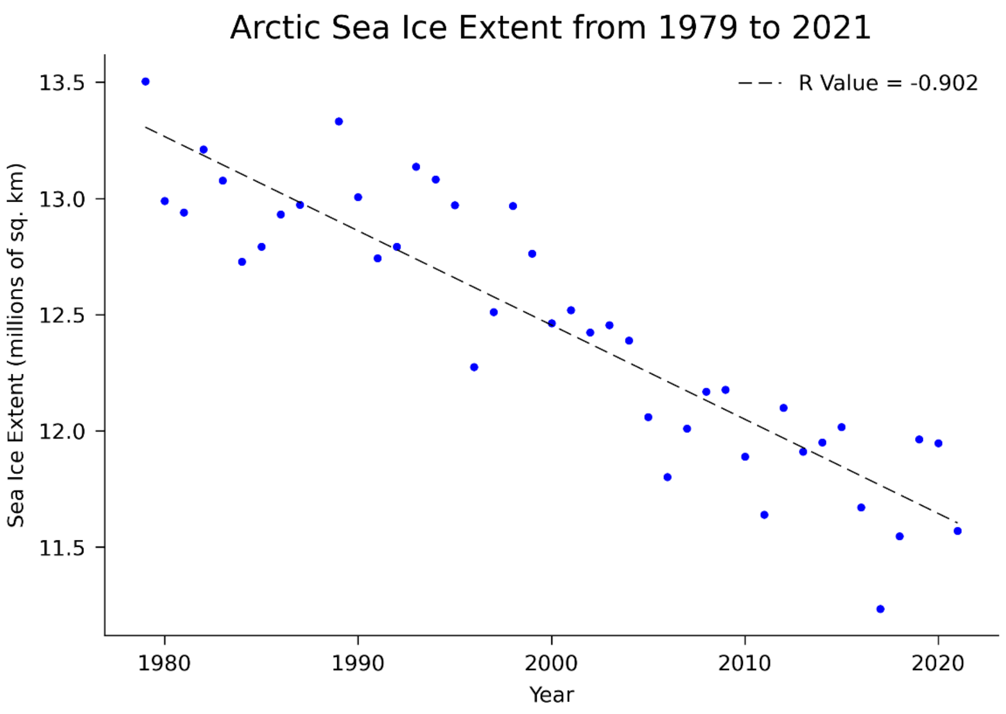
```
plt.savefig(r"YourOutputDirectory\SeaIceExtent_Fig_LongTerm.png", dpi = 500)
```

Your final plotting script should look something like below…



```
###PLOTTING###
#define figure and axes variables
fig, ax = plt.subplots()

#define fontsize
fs = 8

#plot extent data
ax.scatter(year, ext, s = 4, color = 'blue')

#plot line of best fit
ax.plot(year, reg, linewidth = 0.5, linestyle = '--', color = 'black', dashes = [10, 5], label = 'R
Value = ' + str(r_value)[0:6])

#label each axis
ax.set_xlabel('Year', fontsize = fs)
ax.set_ylabel('Sea Ice Extent (millions of sq. km)', fontsize = fs)

plt.xticks(fontsize= fs)
plt.yticks(fontsize= fs)

#adjust axis
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['left'].set_linewidth(0.5)
ax.tick_params(width = 0.5)

#plot title
plt.title('Arctic Sea Ice Extent from 1979 to 2021', fontsize = (fs * 1.5))

#plot legend
plt.legend(frameon = False, fontsize = fs)

#savefig
plt.savefig(r"C:\Users\geoki\OneDrive - University of Southampton\Documents
\PUBS_2021\Day_2\Arctic_Sea_Ice_Extent\Outputs\SeaIceExtent_Fig_LongTerm.png", dpi = 500)

#show the plot
plt.show()
```

That may seem like a lot of effort to go through to produce a single figure, however you can adapt the input data to make consistent style plots or produce multiple plots for various datasets. For example, at the top of the whole script you could change the folder path for Antarctic data, run the identical analysis and have a comparison plot. Alternatively, what would the graph look like if we changed that original 15% sea ice concentration threshold? Now instead of having to create the plot again by hand, we can change one number and the name of our output figure to see how this would change. If all has gone well, your final plot should look similar to the one below.


Arctic Sea Ice Extent from 1979 to 2021
R Value = -0.902
Sea Ice Extent (millions of sq. km)
Year
13.5
13.0
12.5
12.0
11.5
1980
1990
2000
2010
2020