technoteach

technocamps

Llywodraeth Cymru
Welsh Government

Prifysgol Abertawe
Swansea University

Cardiff University
Prifysgol Caerdydd

Prifysgol Bangor University

Cardiff Metropolitan University | Prifysgol Metropolitan Caerdydd

University of South Wales
Prifysgol De Cymru

Cyngor Cyllido Addysg Uwch Cymru
Higher Education Funding Council for Wales
hefcw

Prifysgol Cymru Y Drindod Dewi Sant
University of Wales Trinity Saint David

Prifysgol Aberystwyth University

Prifysgol Glyndŵr Wrecsam | Wrexham Glyndŵr University

institute of CODING in wales technocamps

Python IDEs

All software development could be performed within simple text editors and compiled/executed from the terminal.

However, using an Integrated Development Environments (IDE) offers language specific tools such as: code completion, syntax highlighting, code analysis, code refactoring, project organisation and management, debugging tools, plugins… and many more!

There are many (IDEs) available for Python, find the best for you!

JupyterLab / Google Colab

Today we will be using Google Colab as our development platform. This uses the Project Jupyter Notebook interface for Python.

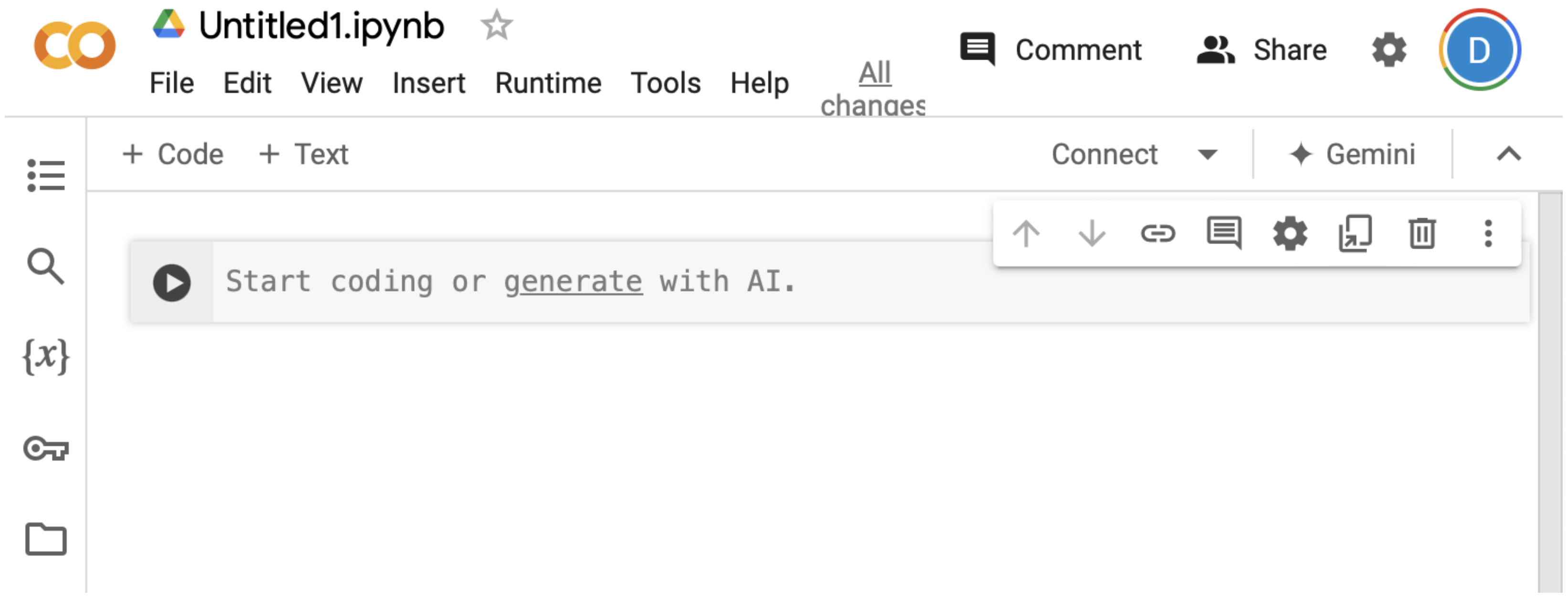
Essentially, JupyterLab allows you to write Python code in small executable chunks, instead of one large executable program.

This is a great tool for learning or demonstrating Python, as you can observe your data as it is altered with each execution.

Colab

Log into Google Colab with your **personal** email.

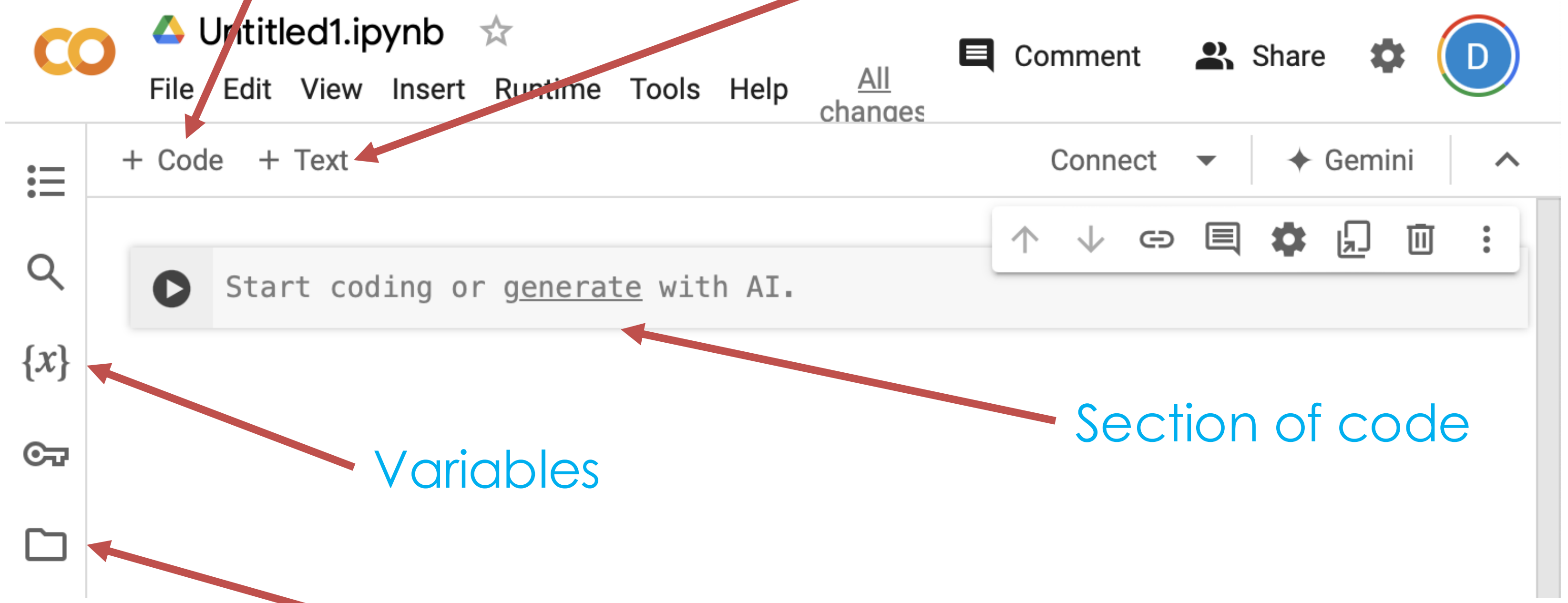
Go to **File** > **New Notebook in Drive**.



Colab

Add section of code

Add section of text



```
Start coding or generate with AI.
```

Section of code

Variables

Additional Files

Python – Pandas

Pandas

Pandas is a powerful data manipulation and analysis library for Python.

It provides data structures and functions that are incredibly efficient for working with complex data (tables, spreadsheets etc.)

Pandas can easily load data from various sources, clean and process it, perform complex operations like filtering or grouping, manipulate the data in many ways, and visualize the results.

It's like having a Swiss Army knife for data analysis tasks in Python.

Pandas Data Structures

The primary data structures within pandas are Series and DataFrames.

A **Series** is a one-dimensional array-like object containing a sequence of values – essentially the same as a list within Python (with some minor differences). It can hold any data-type and has an associated index.

A **DataFrame** is a two-dimensional, size-mutable, and potentially heterogenous tabular structure with labelled axes (rows and columns). This can be thought of as a dictionary of Series objects.

Pandas HTML Table

As well as importing files via an API, the pandas library offers us a way to scrape webpages for HTML tables.

We will use this method to learn the basics of navigating a pandas data structure.

Go to this link to begin:

tc1.me/EltonJohn

Pandas HTML Table

We will need to copy that Wikipedia link into our code. Decide which table you would like to scrape (you are more than welcome to find your own page and table!):

```
import pandas as pd

url =
"https://en.wikipedia.org/wiki/Elton_John_albums_disc
ography"

tables = pd.read_html(url)

print(f"Number of tables found: {len(tables)}")
```

Pandas HTML Table

We can find the index of our desired table by printing the top of each table using pandas **head()** function:

```
for i, table in enumerate(tables):
        print(f"\nTable: {i}")
        print(table.head(0))
```

Pandas HTML Table

Fill in the value of your chosen table to store it as a DataFrame and print it to the screen:

```
chosenTable = tables[0]


print("Selected Table:")
print(chosenTable)
```

Pandas DataFrame - Columns

To access a column of a DataFrame, use the following format specifying the column label:

```
print(chosenTable[“Title”])
```

If the column has sub-titles, then it is (*should be*) a multi-indexed DataFrame, we can specify this with the format:

```
print(chosenTable[(“Peak chart positions”,
                                                  “UK [16]” )])
```

Pandas DataFrame - Rows / Entries

To access a row of a DataFrame, use the following format specifying the index of the row:

```
print(chosenTable.loc[6])
```

To access a particular entry, we specify both the index and label:

```
print(chosenTable.loc[6, “Title”])
```

Pandas DataFrame - Reindexing

To reindex our DataFrame (i.e. choosing the desired indexes, or adding new empty indexes) we use the **reindex** function:

```
favouriteAlbums = chosenTable.reindex(
                                      [3, 4, 6, 7, 500])
```

We can reset the indexing with the **reset_index** function:

```
favouriteAlbums = favouriteAlbums.reset_index(
                                            drop = True)
print(favouriteAlbums)
```

Pandas DataFrame - Remove

To remove a row or column from a DataFrame we use the drop function (this can take a list parameter to remove multiple values):

```
favouriteAlbums  = favouriteAlbums.drop(
columns = [“Album Details”, “Peak chart positions”]))

favouriteAlbums = favouriteAlbums.drop(
                                      index = [4]))

print(favouriteAlbums)
```

Pandas DataFrame - Add Column

To add a column from a DataFrame we use the following syntax.

Note that the new column **must** match the length of the DataFrame.

```
favouriteAlbums[“Personal Rating”] = [10, 8, 9, 9]
```

Pandas DataFrame - Add Row

To add a row to the DataFrame we create a new DataFrame with identical columns containing the row(s) to add, then concatenate the two DataFrames.

Note that this becomes more complicated with multi-index DataFrames, so has been omitted from this example.

```
new_row = pd.DataFrame(
                 {'A': [5], 'B': [9], 'C': [13]})

df = pd.concat([df, new_row], ignore_index=True)
```

Pandas DataFrame - Add Row

To add a row to the multi-index DataFrame we use the same syntax, but must pass all labels as tuples to show that they are multi-index:

```
new_row = pd.DataFrame({
('Title’, 'Title’): ['Greatest Hits’],
('Certifications', 'Certifications'): ['N/A’],
('Personal Rating', ''): [10]})

favouriteAlbums = pd.concat([favouriteAlbums, new_row], ignore_index = True, axis = 0)
print(favouriteAlbums)
```

Pandas DataFrame - Changes

To change the value of a cell in the DataFrame we use the **at** function, with a similar syntax to checking the value:

```
favouriteAlbums.at[3, ‘Personal Rating’] = 10
```

Import Data

Today we are going to use an even more complex data set, all Olympic athletes and the events they competed in since 1896!

This will allow us to demonstrate the benefits of using Pandas.

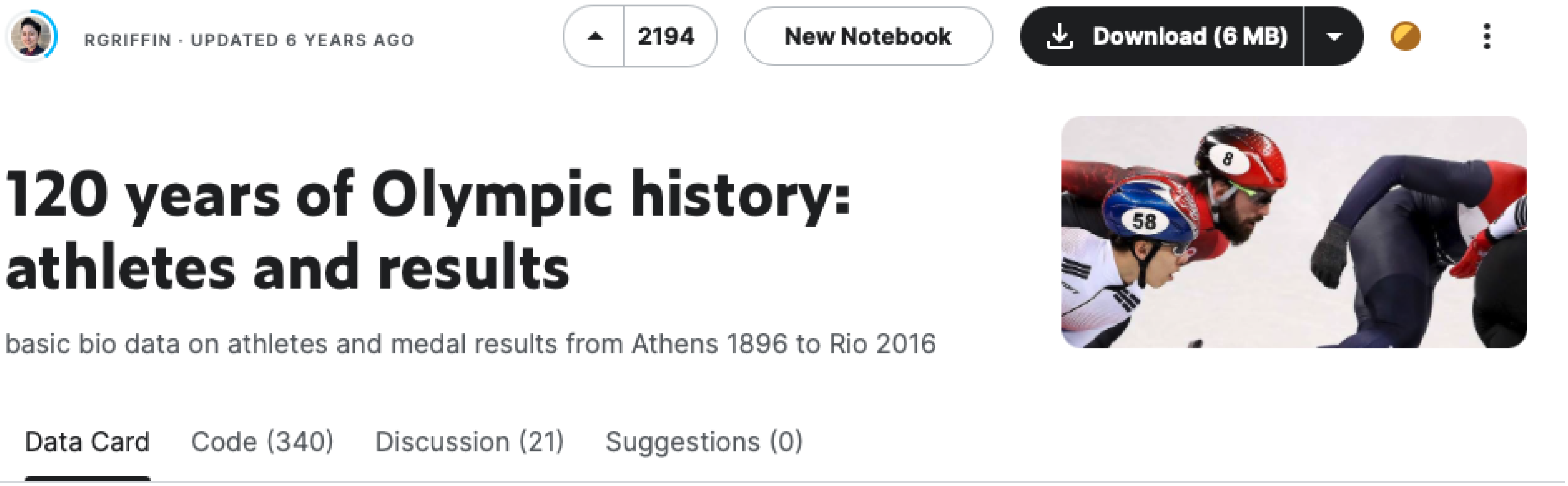
Like yesterday, we will add the dataset via an API call. Can you remember how we did that?

Go to this link to begin:

tc1.me/OlympicData

Download Metadata

Again, this webpage does not offer us a direct download page. So, we have to manually find the link:



RGRIFFIN · UPDATED 6 YEARS AGO

2194

New Notebook

Download (6 MB)

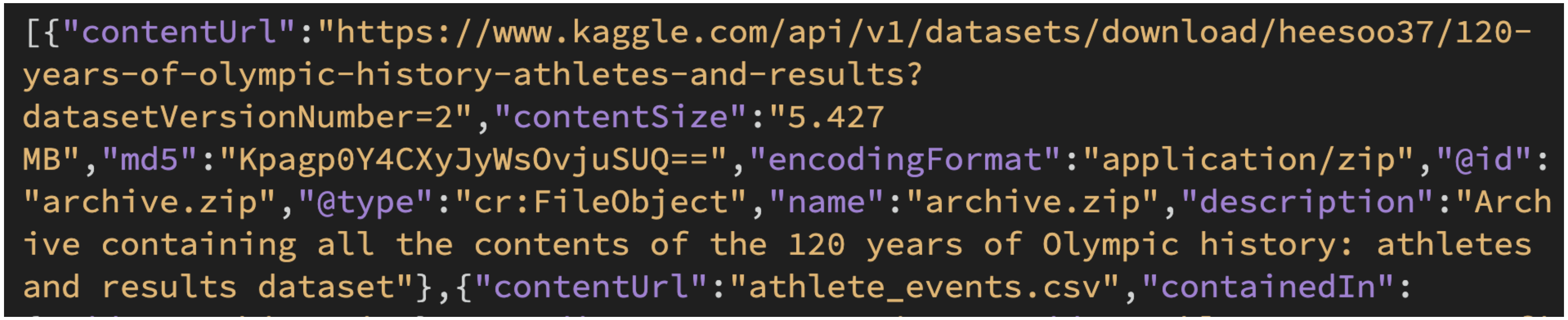
120 years of Olympic history: athletes and results

basic bio data on athletes and medal results from Athens 1896 to Rio 2016

Data Card · Code (340) · Discussion (21) · Suggestions (0)

Metadata File

By opening the metadata file, we can find the **contentUrl**:



```
[{"contentUrl":"https://www.kaggle.com/api/v1/datasets/download/heesoo37/120-
years-of-olympic-history-athletes-and-results?
datasetVersionNumber=2","contentSize":"5.427
MB","md5":"Kpagp0Y4CXyJyWsOvjuSUQ==","encodingFormat":"application/zip","@id":
"archive.zip","@type":"cr:FileObject","name":"archive.zip","description":"Arch
ive containing all the contents of the 120 years of Olympic history: athletes
and results dataset"},{"contentUrl":"athlete_events.csv","containedIn":
```

This also, conveniently, tells us that the content is a **zip** file.

This means that in order to retrieve the **csv**, we will have to extract the **zip** archive.

Import Athlete Data

Firstly, we will request the file from the server via an API call from within our Python code, this requires an import:

```
import requests


response = requests.get(
‘https://www.kaggle.com/api/v1/datasets/download/hees
oo37/120-years-of-olympic-history-athletes-and-
results?datasetVersionNumber=2’)
```

Extract Athlete Data

As the response from the server is a **zip** file, we will have to extract the contents for Python to use the data within:

```
import zipfile as zip, io
```

We can import multiple libraries on one line by comma separating.

```
archive = zip.ZipFile(io.BytesIO(response.content),
'r')

athleteFile = archive.open('athlete_events.csv')
```

Load in File

Now we will begin to learn opening a file with pandas:

```
import pandas as pd

athleteData = pd.read_csv(athleteFile)

print(athleteData)
```

Check Column Names

We can see that this is a very large dataset, over 270,000 rows and 15 columns!

This is too large for pandas to automatically display, however, we can use pandas methods to find out more about the data!

Try printing the column labels:

```
print(athleteData.columns)
```

Load in Partial File

Now that we know the column labels, we can choose to only open the data we desire:

```
athleteData = pd.read_csv(athleteFile,
                  usecols = [“ID”, “Sex”, “Age”,
                  “Height”, “Weight”, “NOC”, “Year”,
                  “Season”, “Sport”, “Medal”])
print(athleteData)
```

Load in File

```
0             1   M  24.0   180.0  ...  1992  Summer     Basketball   NaN
1             2   M  23.0   170.0  ...  2012  Summer           Judo   NaN
2             3   M  24.0     NaN  ...  1920  Summer       Football   NaN
3             4   M  34.0     NaN  ...  1900  Summer    Tug-Of-War   Gold
4             5   F  21.0   185.0  ...  1988  Winter  Speed Skating   NaN
...         ...  ..   ...     ...  ...   ...     ...            ...   ...
271111   135569   M  29.0   179.0  ...  1976  Winter           Luge   NaN
271112   135570   M  27.0   176.0  ...  2014  Winter    Ski Jumping   NaN
271113   135570   M  27.0   176.0  ...  2014  Winter    Ski Jumping   NaN
271114   135571   M  30.0   185.0  ...  1998  Winter      Bobsleigh   NaN
271115   135571   M  34.0   185.0  ...  2002  Winter      Bobsleigh   NaN
```

**This is a pandas DataFrame.
It is a unique data structure to hold a table of information, with headers and row numbers.**

Load in File

```
0             1   M  24.0   180.0  ...  1992  Summer      Basketball   NaN
1             2   M  23.0   170.0  ...  2012  Summer            Judo   NaN
2             3   M  24.0     NaN  ...  1920  Summer        Football   NaN
3             4   M  34.0     NaN  ...  1900  Summer     Tug-Of-War   Gold
4             5   F  21.0   185.0  ...  1988  Winter  Speed Skating   NaN
...         ...  ..   ...     ...  ...   ...     ...             ...   ...
271111  135569   M  29.0   179.0  ...  1976  Winter            Luge   NaN
271112  135570   M  27.0   176.0  ...  2014  Winter     Ski Jumping   NaN
271113  135570   M  27.0   176.0  ...  2014  Winter     Ski Jumping   NaN
271114  135571   M  30.0   185.0  ...  1998  Winter       Bobsleigh   NaN
271115  135571   M  34.0   185.0  ...  2002  Winter       Bobsleigh   NaN
```

Same athlete, different events.

This is a pandas DataFrame.
It is a unique data structure to hold a table of information, with headers and row numbers.

Pandas Basics

For the first task we will begin to get familiar with pandas' data structures and the basic functions available!

To **sample the data**, we can simply call the sample function:

```
print(athleteData.sample(5))
```

Pandas Basics

To **filter the data** for a specific condition, we can use this format using a boolean expression:

```
goldAthletes = athleteData[
                    athleteData["Medal"] == "Gold"]


print(goldAthletes)
```

Pandas Basics

We can also **filter** for multiple conditions, using Pythons **boolean** operators (and - **&** / or - **|**):

```
goldBobsleighers = athleteData[
            (athleteData["Medal"] == "Gold") &
            (athleteData["Sport"] == "Bobsleigh")]

print(goldBobsleighers)
```

Pandas Basics

We may often come across missing or empty entries in our datasets. For instance, in this dataset all Olympic athletes are included, however, not all athletes are medal winners.

We can filter out our empty entries with:

```
medalWinners = athleteData[
                  athleteData[“Medal”].notna()]

print(medalWinners)
```

Pandas Basics

To get the **mean average** and **standard deviation** results of a particular column, we can use the following functions (note that we will also round this value to get an exact age):

```
meanAge = athleteData[“Age”].mean().round()

stdDevAge = athleteData[“Age”].std().round()

print(f“Mean Average: {meanAge}\nStandard Deviation:
{stdDevAge}”)
```

Pandas Basics

Now that we know the average age, we can filter by those older and younger who won a medal:

```
medalBelowMean = athleteData[
                  (athleteData[“Medal”].notna()) &
                  (athleteData[“Age”] < meanAge)]
medalAboveMean = athleteData[
                  (athleteData[“Medal”].notna()) &
                  (athleteData[“Age”] >= meanAge)]
print(f“Below: {len(medalBelowMean)}\n
            Above: {len(medalAboveMean)}”)
```

Pandas Basics

We can get the oldest and youngest athletes by sorting by age, then looking at the top and bottom of the DataFrame:

```
athleteAgeSort = athleteData[athleteData[“Age”]
                   .notna()].sort_values(“Age”,
                          ascending = False)


print(athleteAgeSort.head())
print(athleteAgeSort.tail())
```

Pandas Basics

As well as sorting, we can rank the data. Let's rank the height of Gold medal winning Lugers:

```
lugers = athleteData[
            (athleteData["Height"].notna()) &
            (athleteData["Sport"] == "Luge")]

lugers["rankHeight"] = lugers["Height"].rank()
lugers = lugers.sort_values("rankHeight",
                            ascending = False)
print(lugers)
```

Pandas Basics

Using the rank we can easily get the midpoint of the data to determine the number of gold medal winners above and below:

```
belowMid = len(lugers[
            (lugers[“rankHeight”] < len(lugers)//2)
            & (lugers[“Medal”] == “Gold”)]))
aboveMid = len(lugers[
            (lugers[“rankHeight”] > len(lugers)//2)
            & (lugers[“Medal”] == “Gold”)]))
print(f“Gold Medal below Mid Rank: {belowMid}\n
      Gold Medal above Mid Rank: {aboveMid}” )
```

Plot Scatter Graph

To plot a basic graph using pandas is very simple:

```
import matplotlib.pyplot as plt
...
plt.figure()
fig1 = plt.scatter(athleteData.get(“Year”),
                          athleteData.get(“Age”))
...
plt.show()
```

Better Scatter Graph

We can now use our previous knowledge to both filter and label:

```
fig2 = plt.scatter(
                  athleteData[athleteData[“Medal”] ==
                  “Gold”].get(“Year”),
                  athleteData[athleteData[“Medal”] ==
                  “Gold”].get(“Age”), color = “gold”)
plot.xlabel(“Year”)
plot.ylabel(“Age”)
plot.title(“Gold Medal Winners Age
            across all Olympic Years”)
```

Grouping Data

A very similar method to our filtering is grouping, which groups our data into the discrete entries within a column.

We will begin by filtering our data by country, and adding variables to store the number of medals:

```
athletesCountry = athleteData[
                        athleteData["NOC"] == "GBR"]

medals = ["Gold", "Silver", "Bronze"]
medalsCountry = [0, 0, 0]
```

Grouping Data

Now we can plot both variables:

```
medalsCountry[0] = len(athletesCountry.groupby
                  ("Medal").get_group("Gold"))

medalsCountry[1] = len(athletesCountry.groupby
                  ("Medal").get_group("Silver"))

medalsCountry[2] = len(athletesCountry.groupby
                  ("Medal").get_group("Bronze"))
```

Grouping Data

We will get the length of the grouped results as a method of counting the number of each medal:

```
plt.figure()
fig3 = plt.bar(medals, medalsCountry)

plt.title(f“Medal wins for GBR”)
plt.xlabel(“Medals”)
plt.ylabel(“Number of Medals”)
plt.show()
```

Plotting Multiple Data Groups

Now we'll plot the medals for a select number of years:

```
athletesCountry = athleteData.groupby("NOC")
                  .get_group("GBR")

goldMedals   = len(athletesCountry.groupby
                  ("Medal").get_group("Gold"))
silverMedals = len(athletesCountry.groupby
                  ("Medal").get_group("Silver"))
bronzeMedals = len(athletesCountry.groupby
                  ("Medal").get_group("Bronze"))
```

Plotting Multiple Data Groups

```
medalsPerYear = {"Gold": [], "Silver": [],
                 "Bronze": []}
years = [2008, 2012, 2016]

for year in years:
      medalsPerYear["Gold"].append(len(goldsCountry
                 .groupby("Year").get_group(year)))
      medalsPerYear["Silver"].append(len(goldsCountry
                 .groupby("Year").get_group(year)))
      medalsPerYear["Bronze"].append(len(goldsCountry
                 .groupby("Year").get_group(year)))
```

Plotting Multiple Data Groups

This is a more complex plot and so requires some configuration! Don't worry about the specifics, these come with practice:

```
x = np.arange(len(years))

width = 0.25

barID = 0

fig, ax = plt.subplots(layout = “constrained”)
```

Plotting Multiple Data Groups

```
for attribute, measurement in medalsPerYear.items():
      if attribute == “Gold”: barColor = “gold”
      if attribute == “Silver”: barColor = “silver”
      if attribute == “Bronze”: barColor = “peru”

      offset = width * barID
      bar = ax.bar(x + offset, measurement, width,
            label = attribute, color = barColor)
      ax.bar_label(bar, padding = 3)
      barID += 1
```

Plotting Multiple Data Groups

```
ax.set_ylabel("Number of Medals")
ax.set_title(f"Medal Wins for {countryCode}")

ax.legend(loc = "upper left")

ax.set_xticks(x + width, years)
ax.set_ylim(0, 250)
```

Plotting Multiple Data Groups

To instead plot the medals won by a country across all Olympic years, we will need to account for the fact that some years a country may <u>not</u> have won a particular medal (Gold, Silver, Bronze) or any medal at all!

This is because when grouping the data, Python won't create groups for cases with no entries, meaning when we search for that group an error will be thrown.

This will require two new keywords:

`try:` and `except _____:`

Plotting Multiple Data Groups

The previous for loop must be slightly altered to use all years that the country competed:

```
for year in athletesCountry.groupby(“Year”).groups:
```

Plotting Multiple Data Groups

Within the for loop this format of try and catch will be required for each of the medals:

```
try:
    medalsPerYear[“Gold”].append(len(goldsCountry
                     .groupby(“Year”).get_group(year)))

except KeyError:
    medalsPerYear[“Gold”].append(0)
    print(f“No Gold for {year}”)
```

Lab Tasks