Introduction to Python Workshop

Hosted by the BCB-GSO

August 16, 2021

Written by Kelby Kies

Tuteja Lab

Purpose

The purpose of this workshop is not to turn you into a computer scientist or an expert Python programmer. The purpose of this workshop is to introduce you to basic syntax/commands of Python and to give you examples of how Python can be used to visualize data.

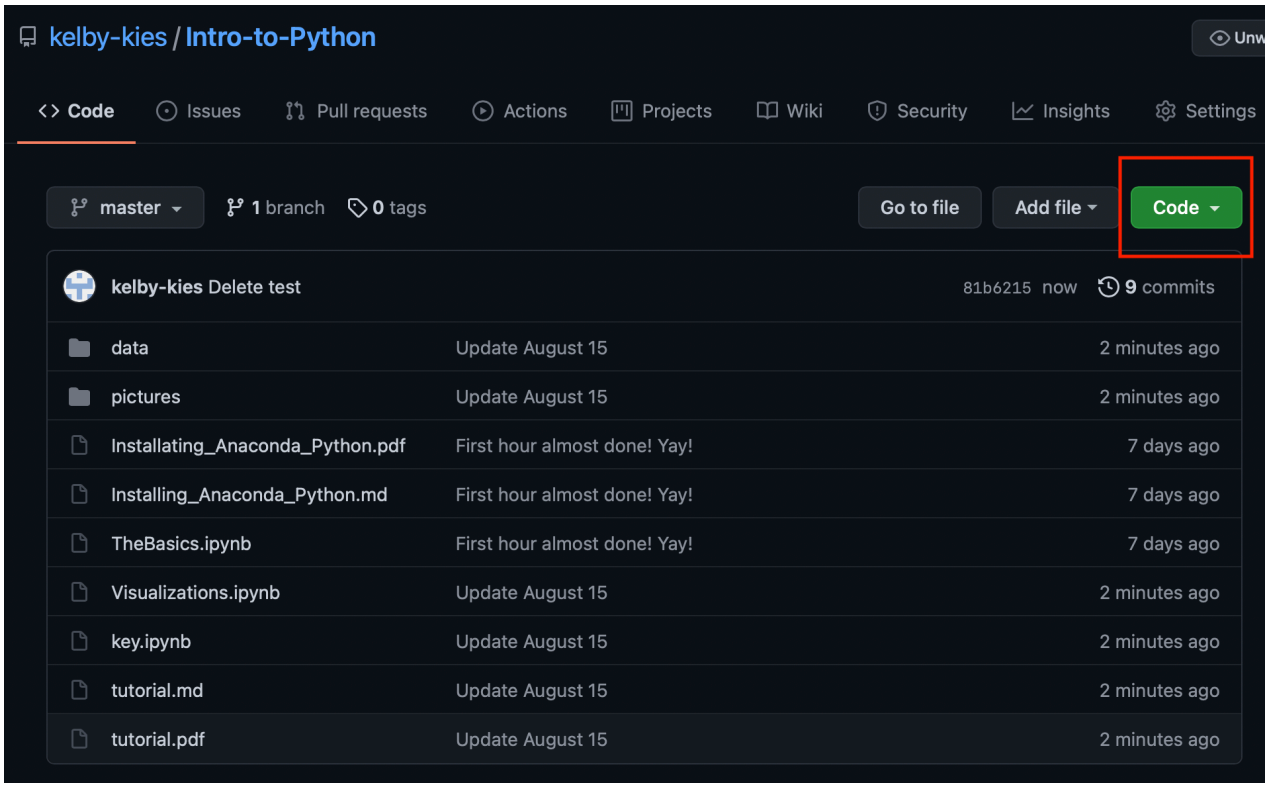
Outline

- Getting started: 15 minutes
- Python is Awesome & here's why!: 5-10 minutes
- The Basics... (first Hour)
- Visualizations (Take a 5-10 minute break; 2nd hour)

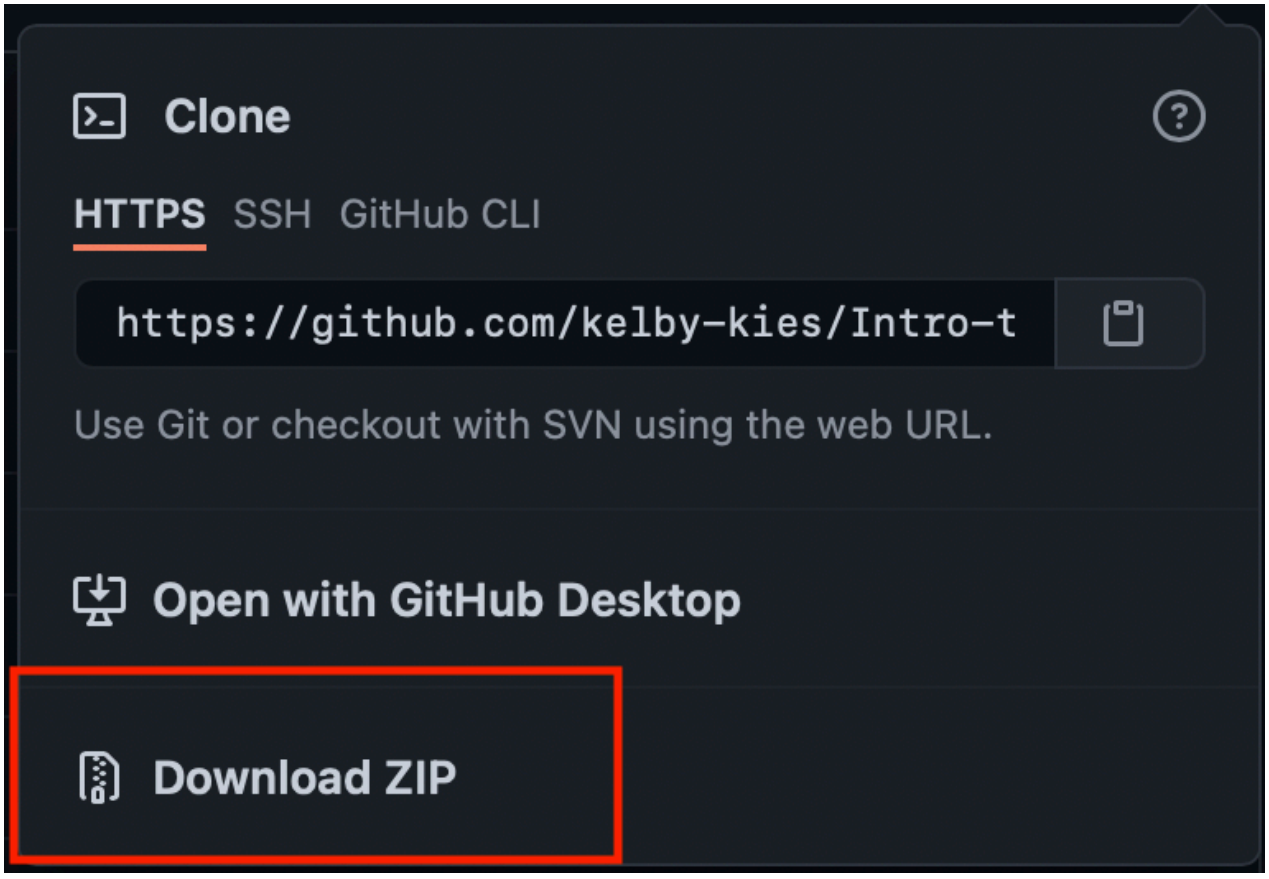
Getting Started

Downloading the GitHub repository

1. Go to https://github.com/kelby-kies/Intro-to-Python
2. Click on the green Code button that has a little download symbol.



3. Click 'Download Zip'. This should download the entire repository (folder) to your downloads folder. I recommend moving this folder to the Desktop or somewhere easy to find.



Installation

Please install Anaconda using the *Installing _ Anaconda_Python.pdf* tutorial in this repository if you have not done so already.

Python is Awesome & here's why!

What is Python?

"Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms." *-Python.org*

Pros & Cons

Pros	Cons
efficient high level data structures	speed limitations
support object oriented programming	issues with threading
simple syntax	Not native to mobile environment
versatile	high memory consumption
easy to use	
fast to develop	
Has a ton of packaages (Pandas, NumPY, BioPython etc.)	

Where to get help on Python

- https://www.python.org
- Your BCB-GSO Tutorial! Or any other tutorial out there
- Google! (This goes for anything in grad school. Google is your friend.)

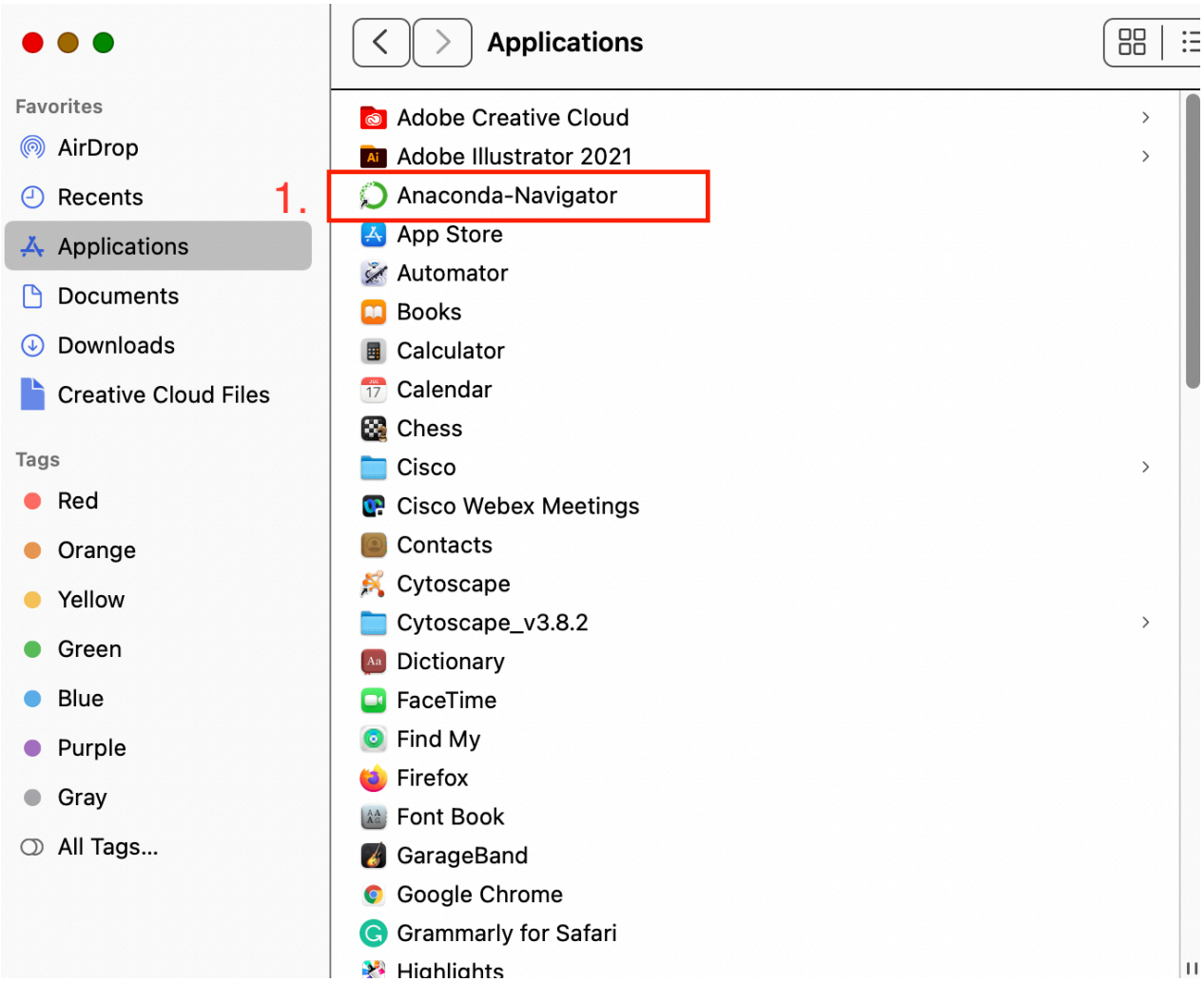
What could I use Python for?

- writing bioinformatic tools:
  - Biopython, PyMOL, PyCogent, Galaxy, pygr and so many more!
- visualizing data
- statistically modeling data
- developing websites and software
- task automation
- data analysis

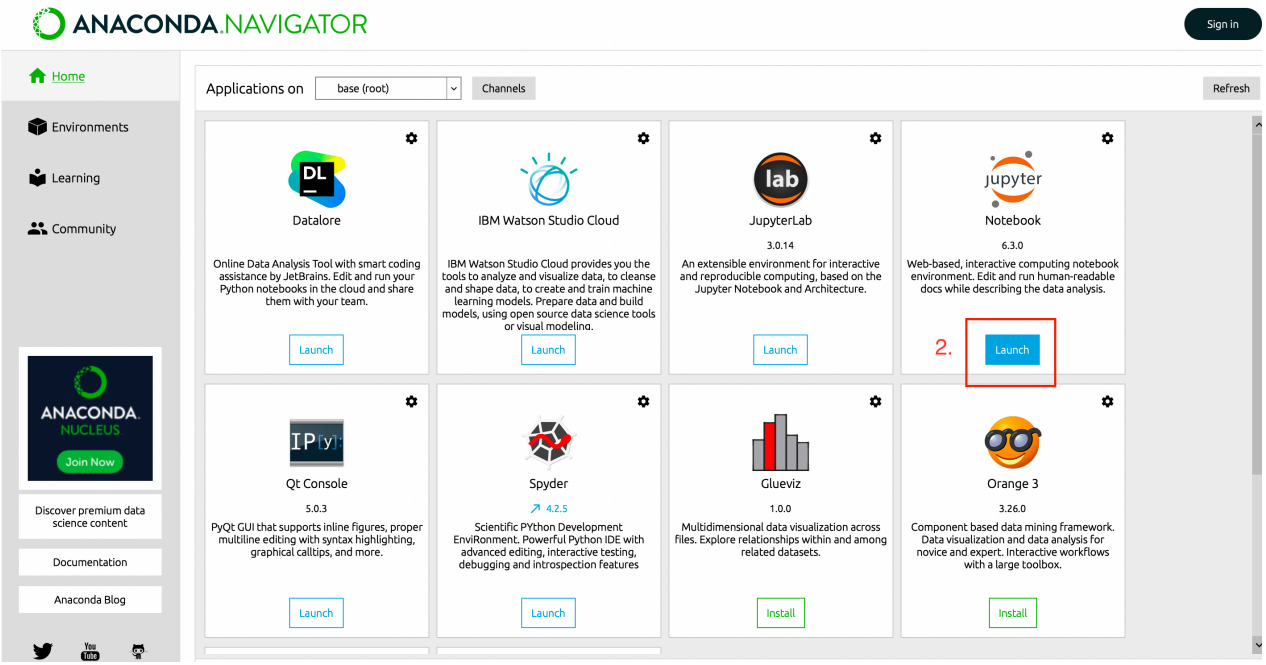
The Basics....

How to run Python in a Jupyter Notebook

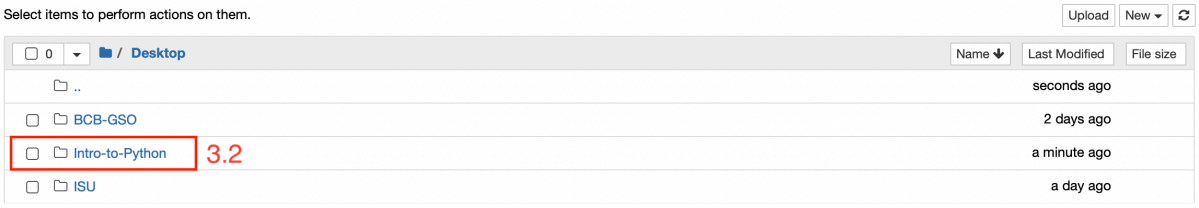
1. Double click on the 'Anaconda Navigator' that we just downloaded



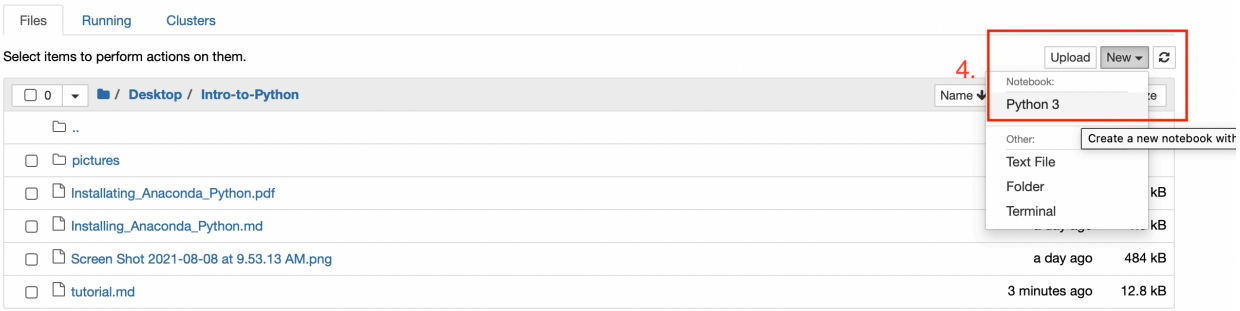
2. Launch a new Jupyter Notebook



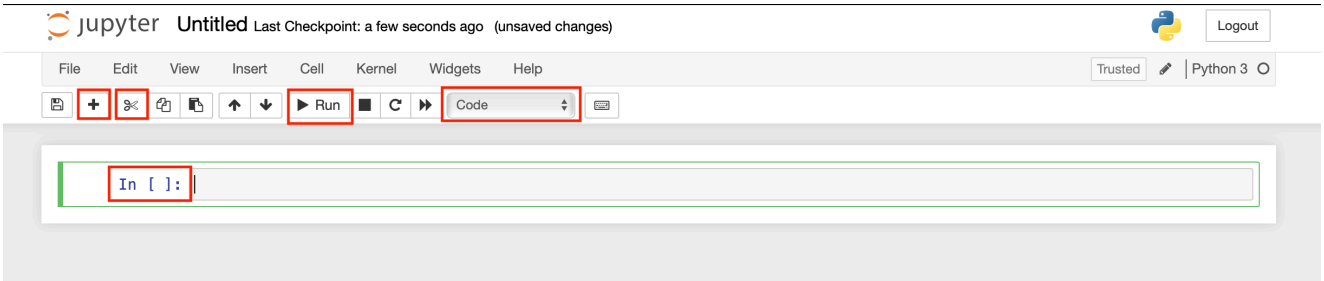
3. Navigate to the *Intro-to-Python* directory that we just downloaded from GitHub



4. Open a new Jupyter Notebook file.



Description of a Jupyter Notebook



- **+** sign: Adds a new line to your notebook.
- 'scissors' symbol: If you want to delete a line then you can click on this button. This will delete the current highlighted line.
- **Code** dropdown menu: You can indicate what type of material this new line will contain using this drop down menu. Options include: Code, Markdown, Raw NBConvert, Heading. For today's purposes we will just be using the Code option.
- **Run**: To execute/run your code there is a run button at the top
- **In[]:**: The lines where we can type code are indicated by 'In[ ]:' which stands for input. We are inputting the code in this line and when we click 'Run' the notebook will take your input code and generate an output.

Let's try a quick example. Copy & Paste the line below into a 'In[]:' line.

```
# Let's write our first program!
print("Hello World")
```

Click the 'Run' button.

We can see that Hello World is printed out on a new line. Notice that the line: `# Let's write our first program` did not print. This is because this line is commented out using #. Comments are used for good documentation and good documentation is important when coding in in ANY language!

Arithmetic

We can use python as a basic calculator. Let's run some basic calculations below.

Add numbers with a `+` sign

```
4 + 10
2 + 2
1.5 + 3.95
```

Subtract numbers with a `-` sign

```
20 - 40
100 - 12.5
5 - 3
```

Multiply numbers with a `*`

```
2 * 3 * 10
1046 * 2345
```

Divide numbers with `/`

```
100/16
12567/3
```

Floor Division (integer division): Does normal division, but then rounds down to the next nearest integer.

```
5.0 / 2
```

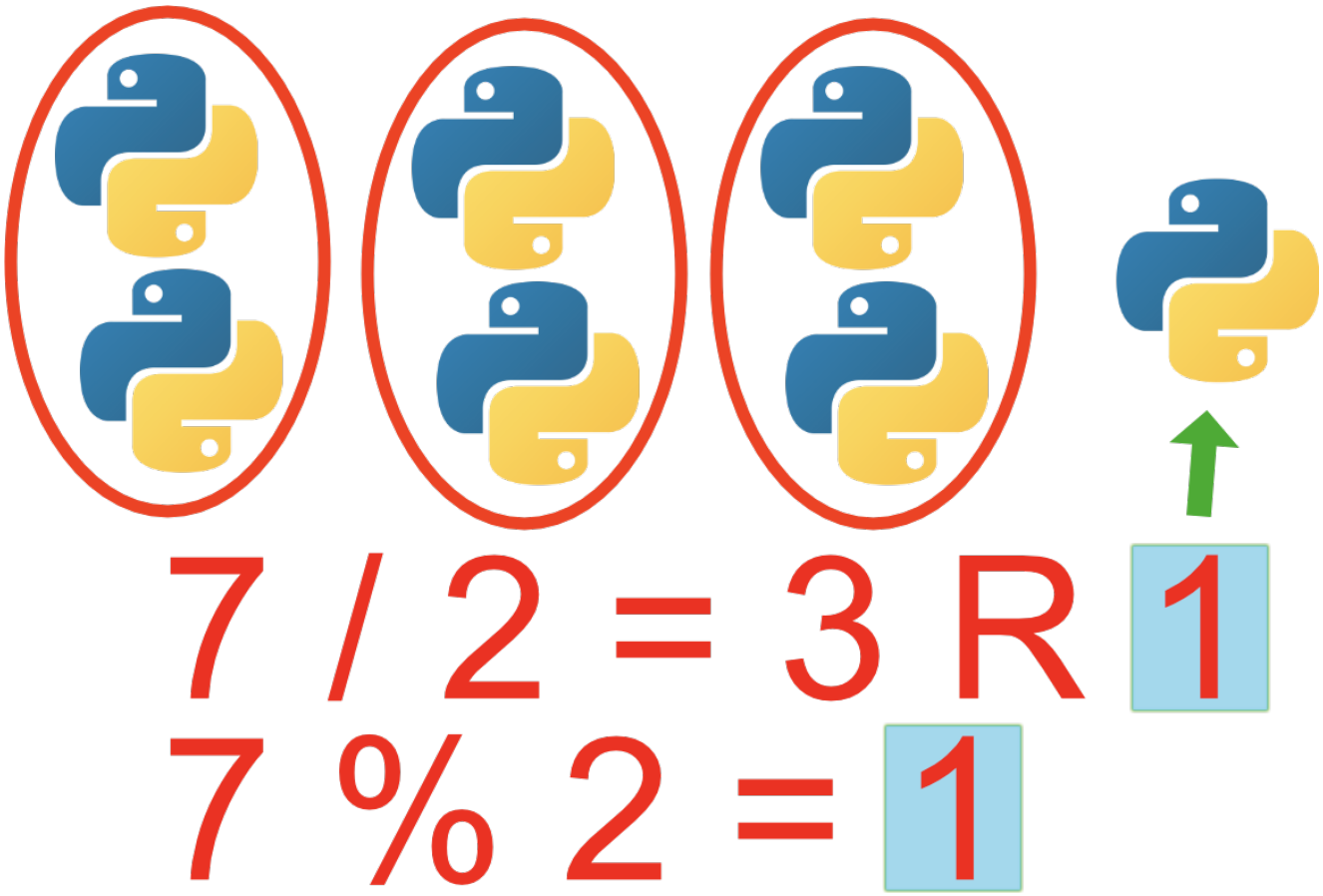
vs.

```
5.0 // 2
```

The `**` operator in Python is used to raise the number on the left to the power of the exponent of the right.

```
4 ** 3

12 ** 2
```

The `modulus operator (%)` returns the remainder left over after dividing the left hand number by the right hand number.



Try this on your own!

```
12 % 5
```

Here you should get an output of `2` because 5 will go into 12 at most 2 times and there is a remainder of 2 left over.

The **Order of Operations** are the same as normal math in Python. Think PEMDAS!

```
# First the sum within the parentheses will be exectued
# This quantity is then multiplied
# The resulting quantity is then subtracted from 100
100 - (4+3) * 2
```

Variables

A variable is a way we can store a value in memory to be used later on.

Here I am storing the value 2 in memory with the name `x`.

```
x = 2
# When we print x the variable x, we should get 2.
print(x)
```

We can later change the value of the variable or use it in a function as well.

```
x = (x * 2)/12 + 10
print(x)
```

What is the value of x now?

It is generally good practice to make your variable name descriptive. For example, below is a list of countries. It would be hard to know what kind of values the variable holds if we called it `x`. Let's give it a more descriptive name like `countries`.

```
countries = ['Japan', 'Mexico', 'Russia', 'Belgium', 'Venezuela']
```

Exercise #1

1. Assign `x` to a numerical value.
2. Add 100 to x.
3. Divde by 4.5.
4. Subtract 10.
5. Multiply by 2.
6. Print the new `x`

Bonus: Can you do #2-5 on the same line to get the same value of x? (Hint: Utilize Python's order of operations!)

Strings

A commonly used datatype in Python is called a *String*. A string is just a collection of characters that are enclosed in quotes.

Strings in python are either surrounded in single (') or double ("") quotes.

```
'hello'
"Hello"
```

Strings can be assigned to a variable as well.

```
blue_sky = "The sky is blue."
```

We can print a string to the screen or 'make it literal' by using `print()`.

```
print("Python is Awesome!")
```

We can the length of a string using `len()`.

```
print(len(school))
```

Indexing a String:

Because a string is just a collection of characters we are able to *index* a string to get a specific character.

```
# Assign a string to variable
school = "Iowa State University"
# print full string
print(school)
```

Great! Now let's print the W in Iowa!

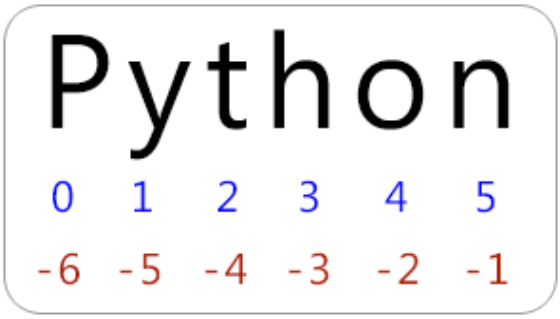
To print out the 'W', we need to take our variable that we assigned a string value (in this case `school`) and *index* it with a specific number.

To *index* means you are telling the Python interpreter to go to that position in the string and pull out the character in that position.

```
print(school[2])
```

Yes! Awesome you printed the W, but WAIT! Did you notice that the W is the 3rd letter in Iowa, but we passed in the index 2?

This is because Python is '0-based', which means that when looking for the character we want to pull out we start counting at 0 rather than 1! This is VERY important because other languages can be different and can lead to problems!



If we want to get the last character of a string we can use `-1`

```
print(school[-1])
```

We can *subset* a chunk of a string using `str[start:end]`. This will index the given `string` at the `start` position and return each character up until the `end-1` position.

```
print(school[0:4])
```

Lists

We can group objects(ints, doubles, strings, etc) together into a `list` A list is defined by using `[ ]` and separating each object with a `,`

```
list_1 = [1,2,3,4]
print(list_1)
list_2 = ['Hello', 'How', 'are', 'you', 'today?']
print(list_2)
```

You can index a list the same way we index strings by passing in a position number in `[]`. This will select a specific object from your list. (A string is just a list of characters.)

```
list_1[2]
list_1[-1]
```

We can also index a string within a list of strings. Lets say I want to get the `?` in `'today?'`.

```
list_2[4][-1]
```

We can *add* and *remove* objects from a list. To *add* a single object to a list use `append`.

```
list_1.append(15)
print(list_1)
list_2.append('Great!')
print(list_2)
```

Note: `append` will only work for adding a single object. If you try to add multiple objects then we will get an error.

One cool thing about `lists` is that they can contain mixed datatypes.

```
list_1.append('List 1 can contain Strings')
print(list_1)
# While list_2 can contain numbers
list_2.append(4)
print(list_2)
```

If we wanted to add more than 1 object, we can use `extend`.

```
list_1.extend(['I', 5, 'love', 7, 'Python'])
print(list_1)
list_2.extend([4.5, 'Happy Monday!'])
print(list_2)
```

We can even add 2 pre-defined lists together and assign to a new list name!

```
new_list = list_1.extend(list_2)
print(new_list)
```

To *remove* objects from a list:

- Remove all items: `clear()`

```
list_1.clear()
print(list_1)
```

- Remove an item by index and get its value: `pop()`

```
print(list_2)
#Let's remove the 3rd element in list_2
print(list_2.pop(2))
print(list_2)
```

  If we don't pass in an index into `pop()` then the last element is deleted.

- Remove an item by value: `remove()`

```
#Let's remove Happy Monday from the new_list
new_list.remove('Happy Monday!')
```

Dictionaries

A dictionary is a collection of `key:values` pairs. One key can have multiple values. Dictionaries in Python are similar to dictionaries in real life. For a real dictionary we can look up the definition (value) of a specific word (key). This is similar in Python; we can look up the values attached to the desired key using the command: `dictionary[key]`

We can define a dictionary by using `dict()` or by using `{ }`. To add a new value to a dictionary: `dictionary[key] = new_value`.

```
# defining the gene_ontologies dictionary with dict()
gene_ontologies = dict()
# Let's add new values to gene_ontologies.
gene_ontologies = dict()
# Let's add new values to gene_ontologies.
gene_ontologies['mesenchymal to epithelial transition'] = ['Stat1','Bmp4', 'Sall1']

# Notice when we add only single values, it will overwrite what is there.
gene_ontologies['angiogenesis'] = 'Nrarp'
gene_ontologies['angiogenesis'] = 'Adam15'

print(gene_ontologies)

# Let's define a new dictionary by using {}
movies_2021 = {
    "comedy":"The Hitman's Bodyguard"
    "Disney":"Cruella"
    "Thriller": "Old"
    }
print(movies_2021)
```

Dictionaries are useful because we can look up a value by using the key.

```
# Let's see the genes that correspond to 'Mesenchymal to Epithelial Transition'.
print(gene_ontologies['mesenchymal to epithelial transition'])
```

We can also see if a specific key exists within a dictionary.

```
"Drama" in movies_2021
"angiogenesis" in gene_ontologies
```

Exercise #2:

1. Make a list of your 3 favorite foods and assign to a variable called `food`. Print. your list using the variable name.
2. Now add your 1 of your neighbor's favorite foods to your list. For those of you that are virtual feel free to type a food in the chat or add my favorite food `'chocolate'.` Print your updated list.
3. Access the `chocolate` element and print the middle `o` character.
4. Create a dictionary with the key's food, hobby, subject and assign each key your favorite. Call this dictionary `favorites`.
5. Add a new favorite to your dictionary. Give it a unique `key` name and assign it a value.

Functions

What is a function?

-a function is a chunk of code that takes input values and performs a 'function' on it to get an output. Some

functions are simple, but some can be complicated.

Basic Structure of a function

To define a function Python uses `def` .

```
def function_name(input, values)
{
    # This function takes 2 input parameters (input, values)
    # And generates an output value.

    output_value = input * values

    # This output value is returned or printed.
    print(output_value)
}
```

To call a function you must use the give name with `()` :

```
function_name(2,4)
```

Built in Python functions

Python has many built in functions.

Example 1: `print(value)` . Here the variable named `value` is the input value into the `print()` and the print function will take that value and print it to the standard out.

Example 2: `abs(-14)` . Here the `-14` is the input value into the `abs()` . `abs()` will return the absolute value of whatever object is passed into it.

Example 3: `len('string')` . The length function is used to find the length of a give string or list.

Importing new functions using Packages

One can import a package with `import <package_name>` to load all of the functions of that package.

Popular packages used by Biologists:

- **NumPy**: "NumPy is the fundamental package for scientific computing in Python. It is a Python package that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more." https://numpy.org/doc/stable/user/absolute_beginners.html
- **Pandas**: "Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language." https://pandas.pydata.org/docs/getting_started/index.html#getting-started

- **Biopython**: "The goal of Biopython is to make it as easy as possible to use Python for bioinformatics by creating high-quality, reusable modules and classes. Biopython features include parsers for various Bioinformatics file formats (BLAST, Clustalw, FASTA, Genbank,...), access to online services (NCBI, Expasy,...), interfaces to common and not-so-common programs (Clustalw, DSSP, MSMS...), a standard sequence class, various clustering modules, a KD tree data structure etc. and even documentation." http://biopython.org/DIST/docs/tutorial/Tutorial.html
    - Fun Fact: Dr. Iddo Friedberg here at ISU helped create BioPython!

Visualizations

Scientists in all areas typically look at their results in the form of a graph. Visualizing data is one of the most utlized features in Python!

When to use the right visualization?

It all depends on the type of data and the question you are trying to answer!

Python Packages for Visualization:

- **Seaborn**: dataset oriented package for making statistical representations in Python; uses Pandas data structures;
- **Matplotlib**: uses numPy; comes with a wide variety of plots like line, bar, scatter, historgram, etc.; John Hunter 2022
- **plotly**: interactive, open-source, high-level, declarative and browser-based visualization package for Python. scientifc charts, 3D graphs, statistical charts, financial charts;
- **ggplot**: grammar of graphics implemented in Python; mapping of data to asthetic attributes and geometric objects.

For today's workshop, we will be use **Seaborn** and **Matplotlib**.

1. Let's `install` the necessary packages.

    ```
    # Install Required Packages
    !pip install pandas
    !pip install matplotlib
    !pip install seaborn
    ```

2. Now these packages are installed, but now we need to `import` them into our Jupyter Notebook.

    ```
    import pandas as pd
    import matplotlib.pyplot as plt
    import seaborn as sns
    ```

    Note: When we import a package we can give it an *alias* such as `pd`. This is just a shortcut way to use this package's functions later on.

3. Now we should import the data that we will be working with into a `pandas` dataframe.
    - Pandas DataFrames: is a tabular data structure comprised of rows and columns, akin to a spreadsheet, database table, or R's data.frame object.

```
# Here is a general way to read in data (.csv file) into a Pandas          da
taframe
df = pd.read_csv('data.csv')
```

Once imported, the dataframe will look similar to this:

	year	team	wins	losses
0	2010	Bears	11	5
1	2011	Bears	8	8
2	2012	Bears	10	6
3	2011	Packers	15	1
4	2012	Packers	11	5
5	2010	Lions	6	10
6	2011	Lions	10	6
7	2012	Lions	4	12

You can learn more on how to use the Pandas package here: https://pandas.pydata.org

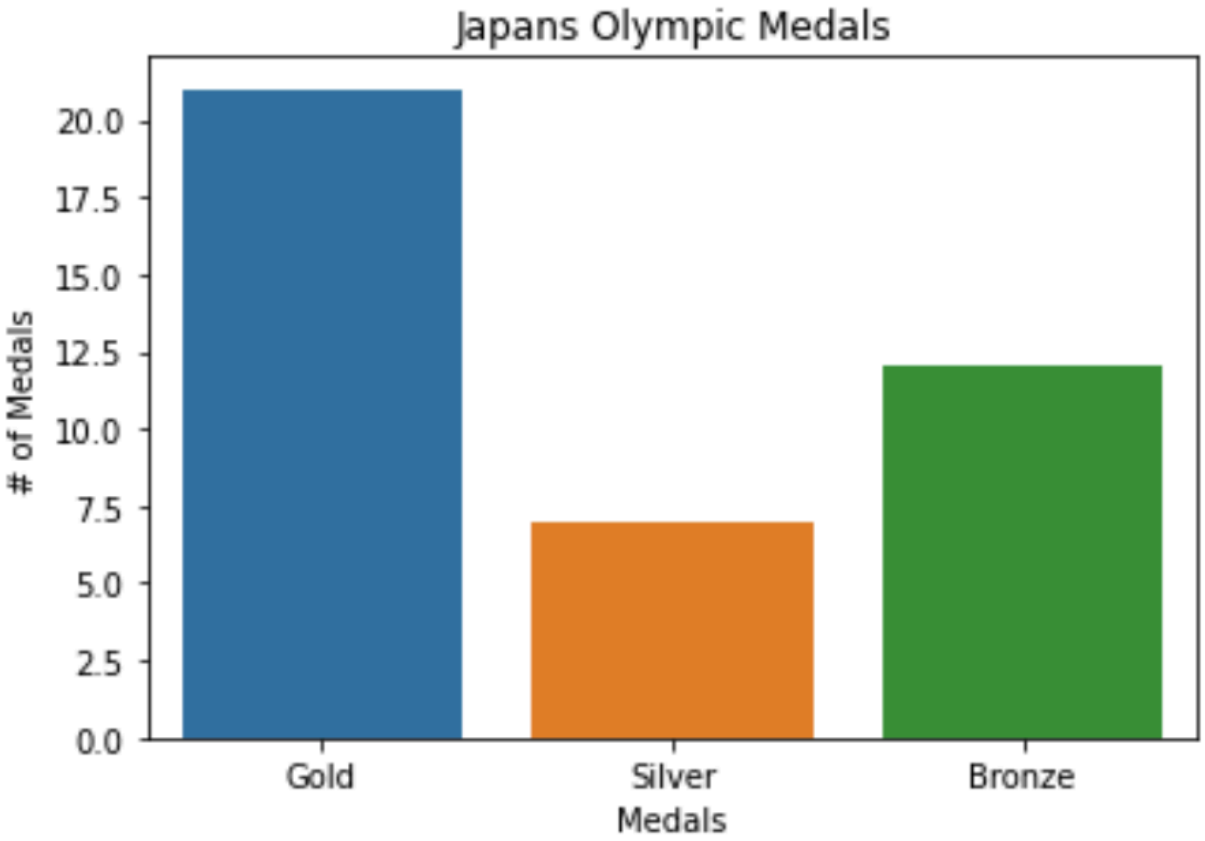
Now, Let's look at some different types of graphs:

To demonstrate the use of **bar graphs**, we will use data that contains the details of over 11,000 athletes, with 47 disciplines, along with 743 Teams taking part in the 2021(2020) Tokyo Olympics. This dataset contains the details of the Athletes, Coaches, Teams participating as well as the Entries by gender. It contains their names, countries represented, discipline, gender of competitors, name of the coaches. Source: Tokyo Olympics 2020 Website

```
# Import medals.csv: this file contains a break down of
# how many Gold, Silver & Bronze Olympic medals each country
# won.
medals = pd.read_csv('data/medals.csv', index_col=0)
medals

# Import EntriesGender.csv: this file has a breakdown of the # of men/women athletes f
or each sport.
gender = pd.read_csv('data/EntriesGender.csv')
gender
```

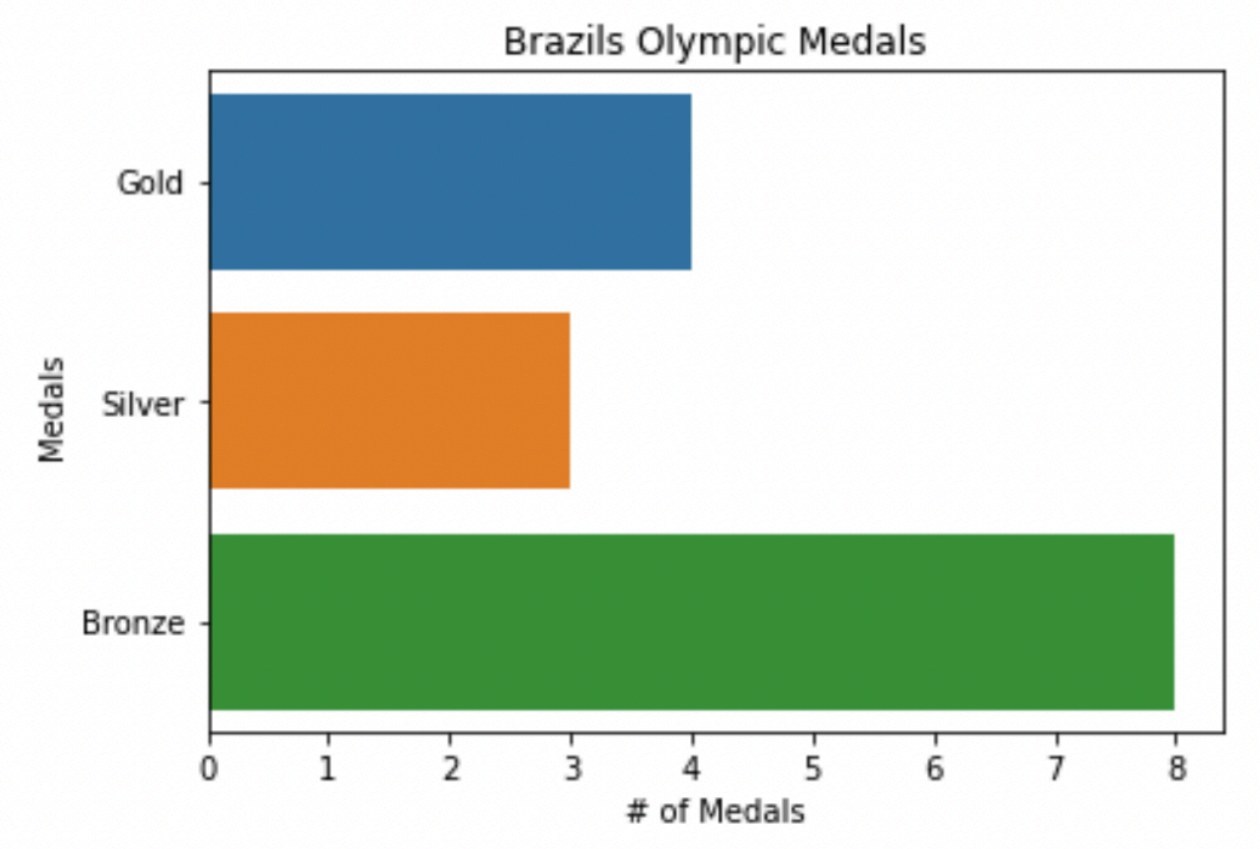
- **bar chart** (vertical) : used to compare metric values across diff. subgroups of data. more groups is better for a bar chart.

```
"""
Description of basic bar chart:
    sns.barplot(xAxis,yAxis)
    plt.title('Plot Title')
    plt.xlabel('X-Axis subtitle')
    plt.ylabel('Y-Axis subtitle')
    plt.show()
"""
# Let's choose a country and see how many medals they won
# In this example, let's choose Japan
medals_won = ['Gold', 'Silver', 'Bronze']
num_of_medals = [21,7,12]

# Create Bar plot using Seaborn
sns.barplot(x=medals_won, y=num_of_medals)
plt.title('Japans Olympic Medals')
plt.xlabel('Medals')
plt.ylabel('# of Medals')
plt.show()
```



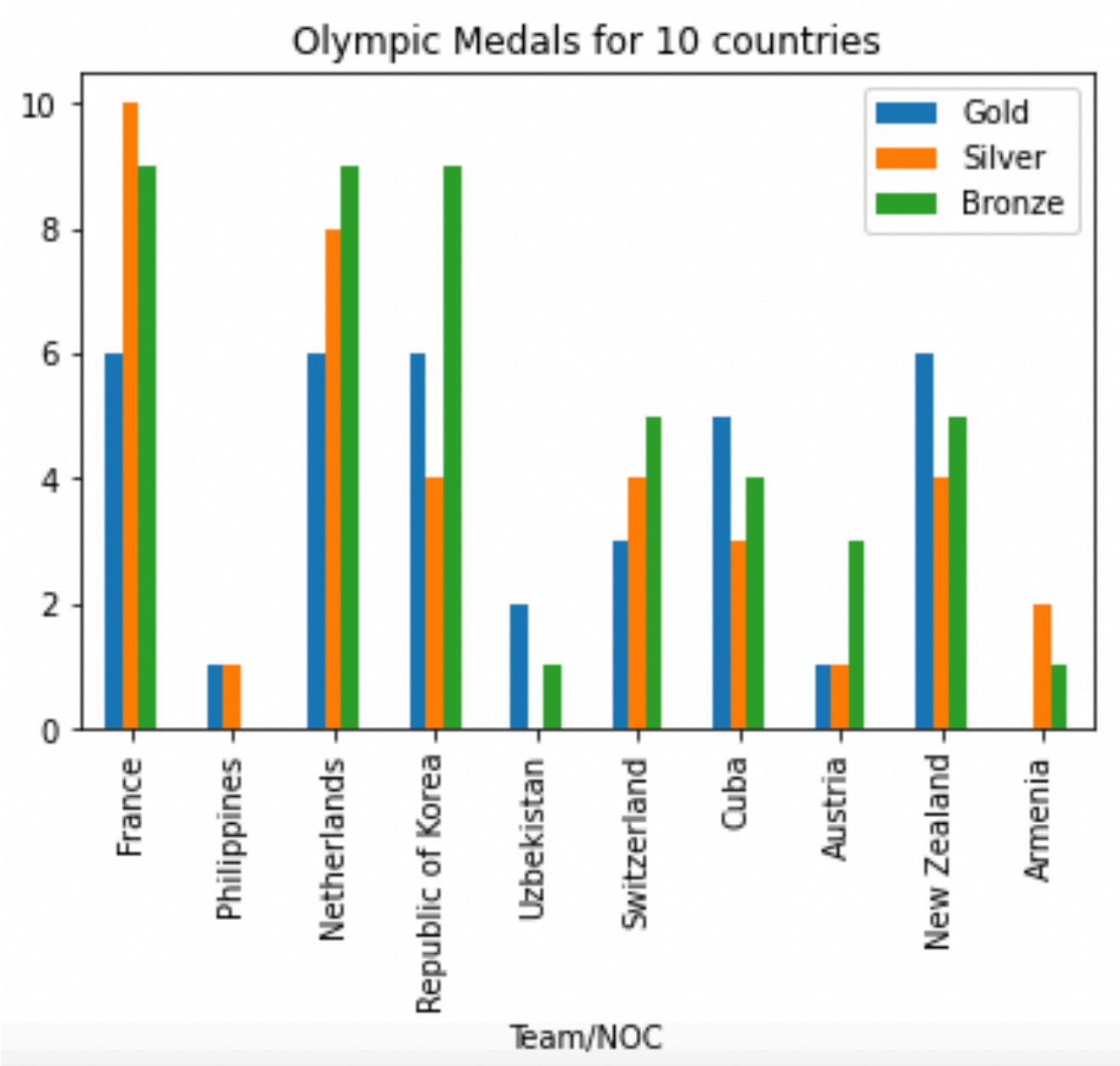
- **column chart** (horizontal): used to compare a single category of data between individual sub-items

```
"""
Description of basic column chart:
    plt.barh(y_axis,x_axis)
    plt.title('title name')
    plt.ylabel('y axis name')
    plt.xlabel('x axis name')
    plt.show()
"""
# Let's choose a country and see how many medals they won
# In this example, let's choose Brazil
medals_won = ['Gold', 'Silver', 'Bronze']
num_of_medals = [4,3,8]

# Create column chart using Seaborn
# Here we use the orient parameter to distinguish that the bars will be horizontal
.
sns.barplot(x=num_of_medals, y=medals_won, orient = "h")
plt.title('Brazils Olympic Medals')
plt.xlabel('# of Medals')
plt.ylabel('Medals')
plt.show()
```



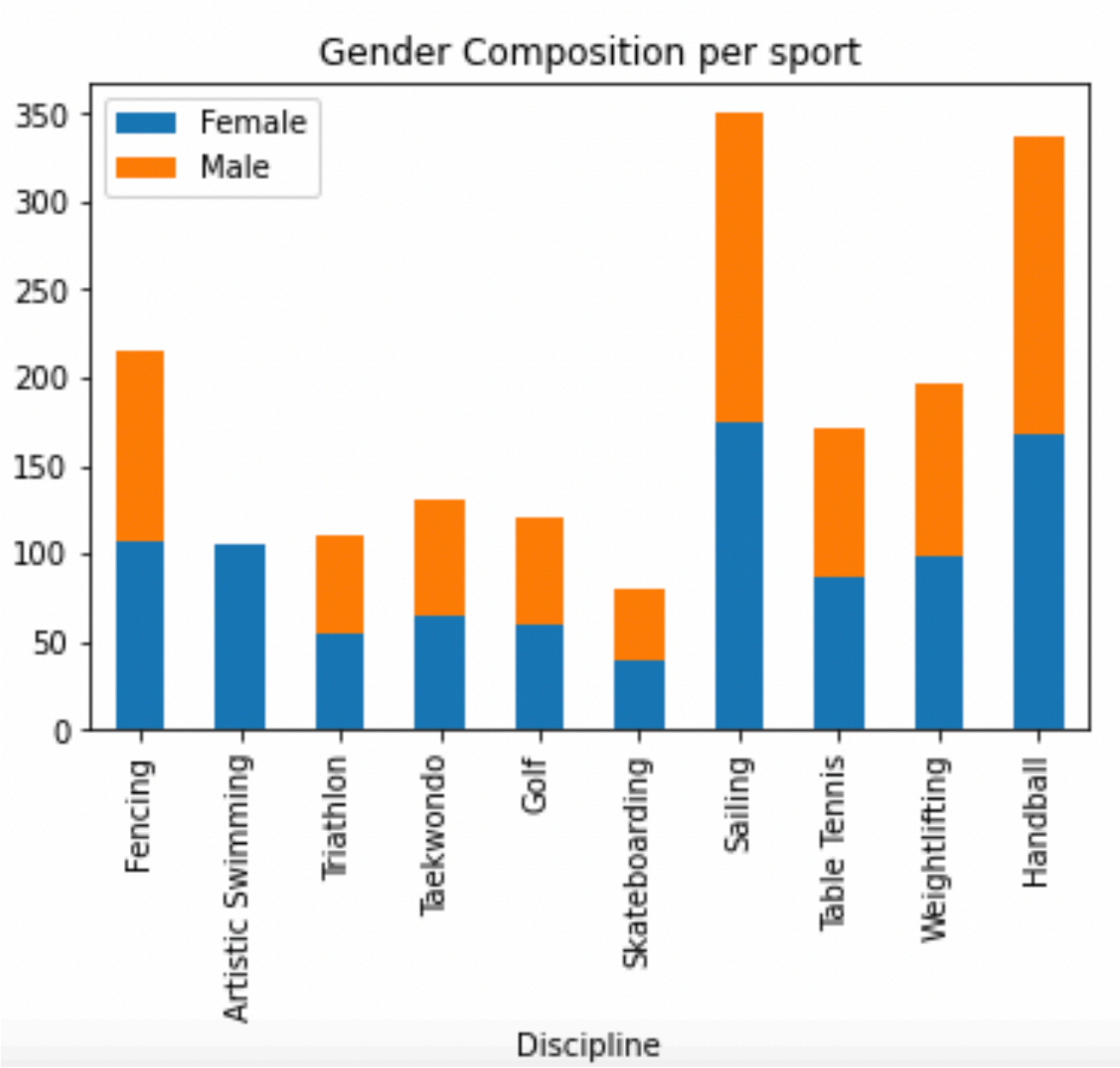
- **grouped bar chart**: used to compare values in certain groups and sub-groups

```
# Now let's compare 10 random countries with a 'Grouped Bar Chart'
# Used the built in sample() function to randomly sample
# 10 countries from our medals data.
ten_countries = medals.sample(n=10)

# Delete extra information
del ten_countries['Total']
del ten_countries['Rank by Total']

# Plot Grouped Bar plot using Matplotlib
ten_countries.plot(x='Team/NOC',
           kind='bar',
           stacked=False,
           title='Olympic Medals for 10 countries')
```



- **stacked bar chart**: compare the total sizes across the available groups and composition of different subgroups.

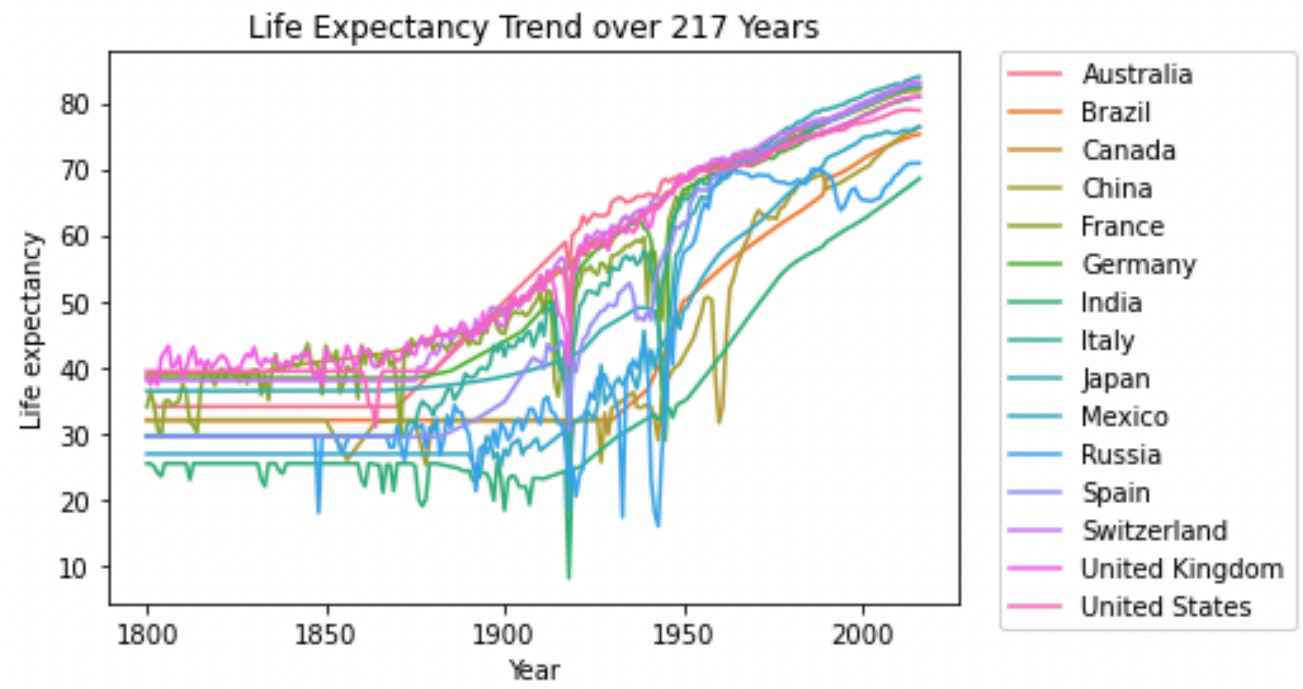
```
# Stacked Bar Plot
# Import EntriesGender.csv: this file has a breakdown of the # of men/women athl
gender = pd.read_csv('data/EntriesGender.csv')
gender

# Select the columns we care about
gender = gender[['Discipline', 'Female', 'Male']]
ten_sports = gender.sample(n=10)

# Create Stacked Bar chart using Matplotlib
# The stacked parameter needs to be set to True to create a stacked bar graph
ten_sports.plot(x='Discipline', kind='bar', stacked=True,
title='Gender Composition per sport')
```



To demonstrate the use of **line chart**, **histogram** & **box plot**, we will use a dataset that contains the life expectancy for 15 different countries over 217 years. https://www.kaggle.com/brendan45774/countries-life-expectancy

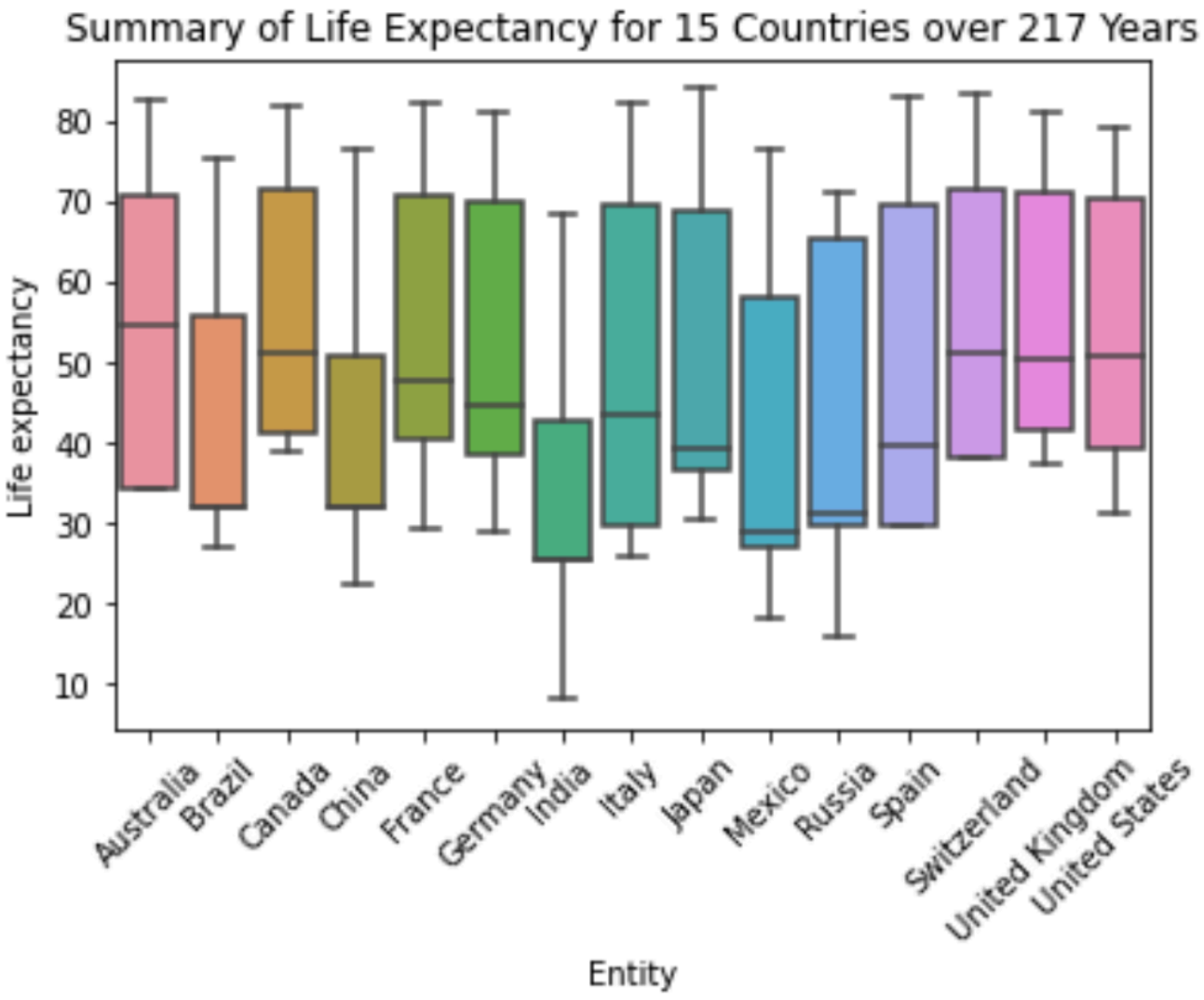
- **line chart**: used to represent continuous data; trend across time.

```
# Read in Life Expectancy data
life = pd.read_csv('data/Life_expectancy.csv')

# Here's what our data looks like
print(life)

# Create Line Plot using Seaborn
sns.lineplot(data=life, x='Year', y='Life expectancy',      hue='Entity').set(titl
e='Life Expectancy Trend over 217 Years')
# Put the legend out of the figure
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```



- **box plot**: used to show shape of distribution and the summary of the data.

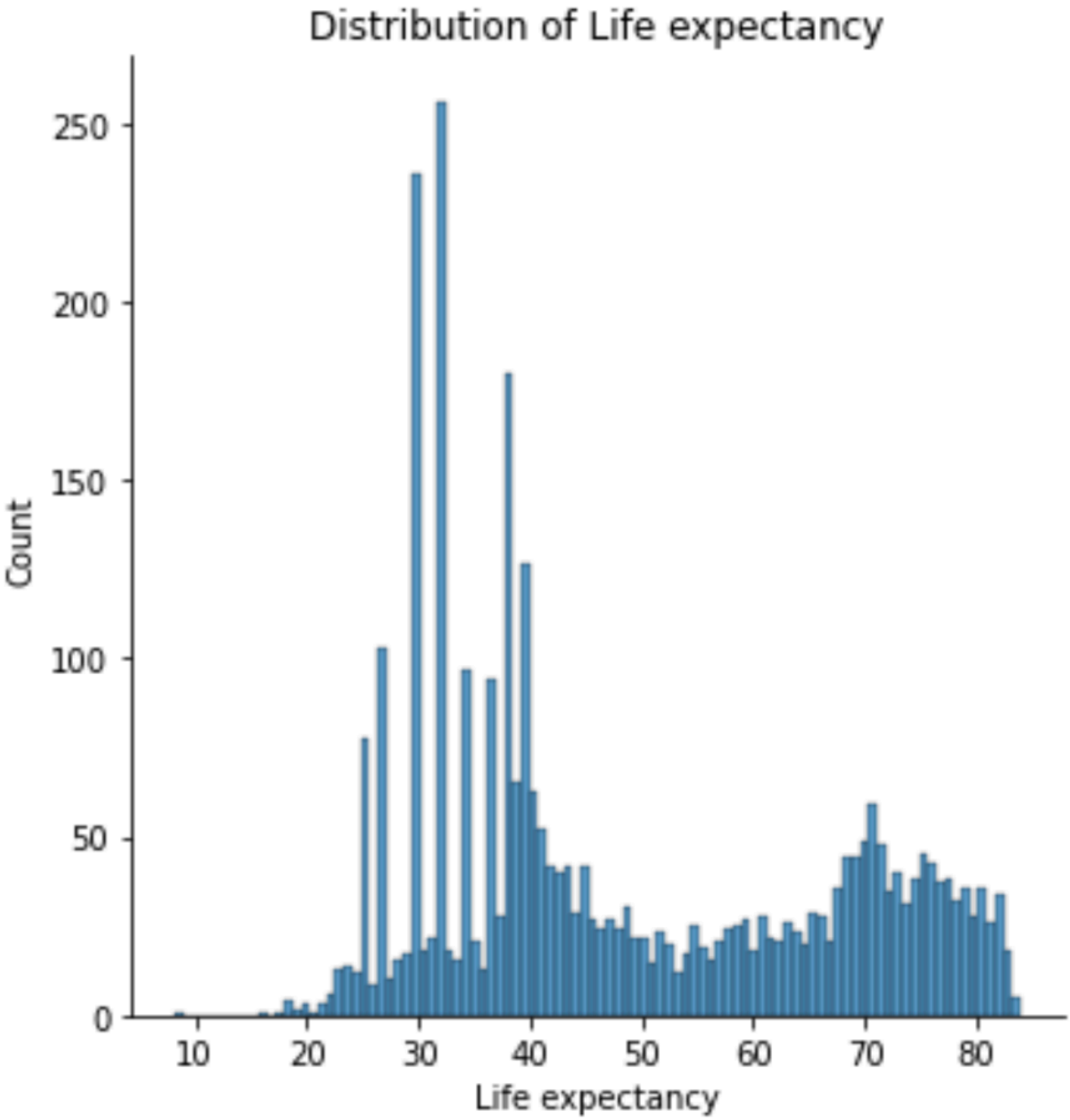
```
# Let's show the summary of life expectancy for each country using a Seaborn Box P
lot
ax = sns.boxplot(x="Entity", y="Life expectancy", data=life)
# Set the title of the plot
ax.set_title('Summary of Life Expectancy for 15 Countries over 217 Years')
# Rotate the labels on the x-axis for better readability
ax.set_xticklabels(ax.get_xticklabels(),rotation=45)
```

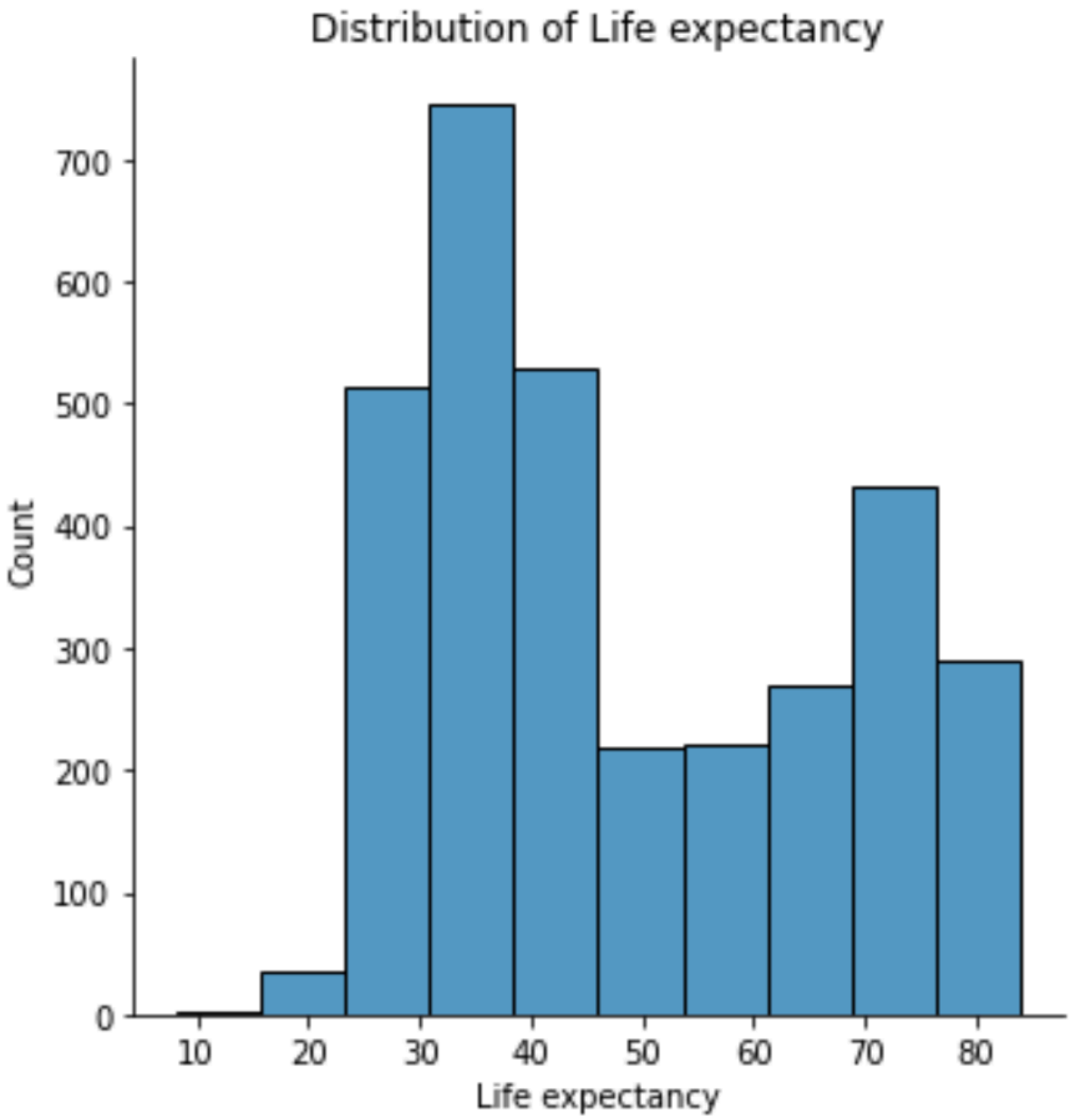


- **histogram**: observe distribution of a single variale with a few data points.

```
# Histogram
# Let's look at the distribution of life expectancy over 217 years

# Create Histogram with seaborn using either histplot() or displot()
# Change the color using the color parameter
sns.histplot(life['Life expectancy'], color='red', bins=20).set(title='Distributio
n of Life expectancy')
```


```
# We can change the bin size using the bins parameter
# It is a good idea to try different bin sizes to get an overall feel for the data d
sns.displot(life['Life expectancy'], bins=100).set(title='Distribution of Life expec
sns.displot(life['Life expectancy'], bins=10).set(title='Distribution of Life expect
```

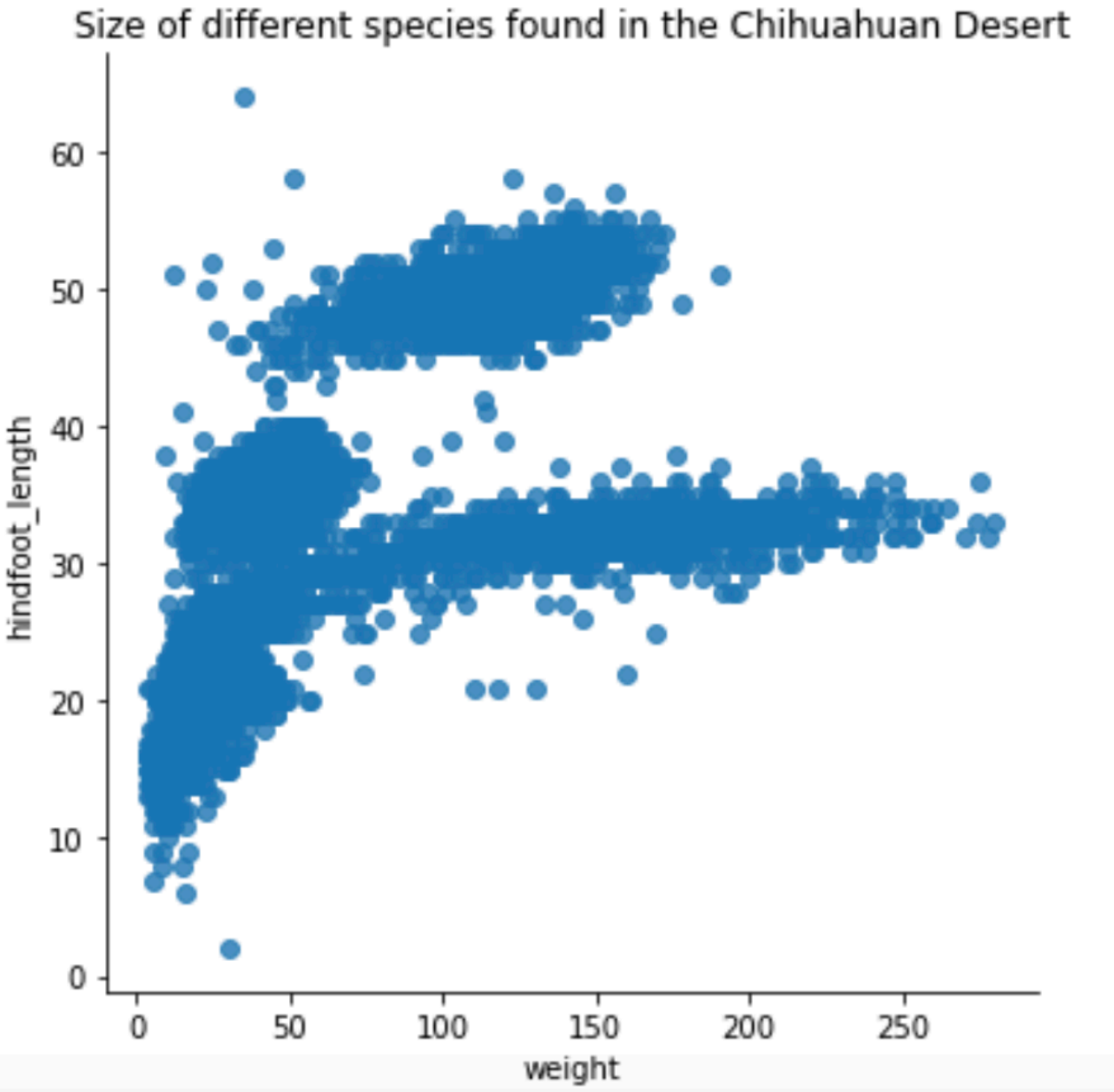

Distribution of Life expectancy
Count
0
50
100
150
200
250
10
20
30
40
50
60
70
80
Life expectancy



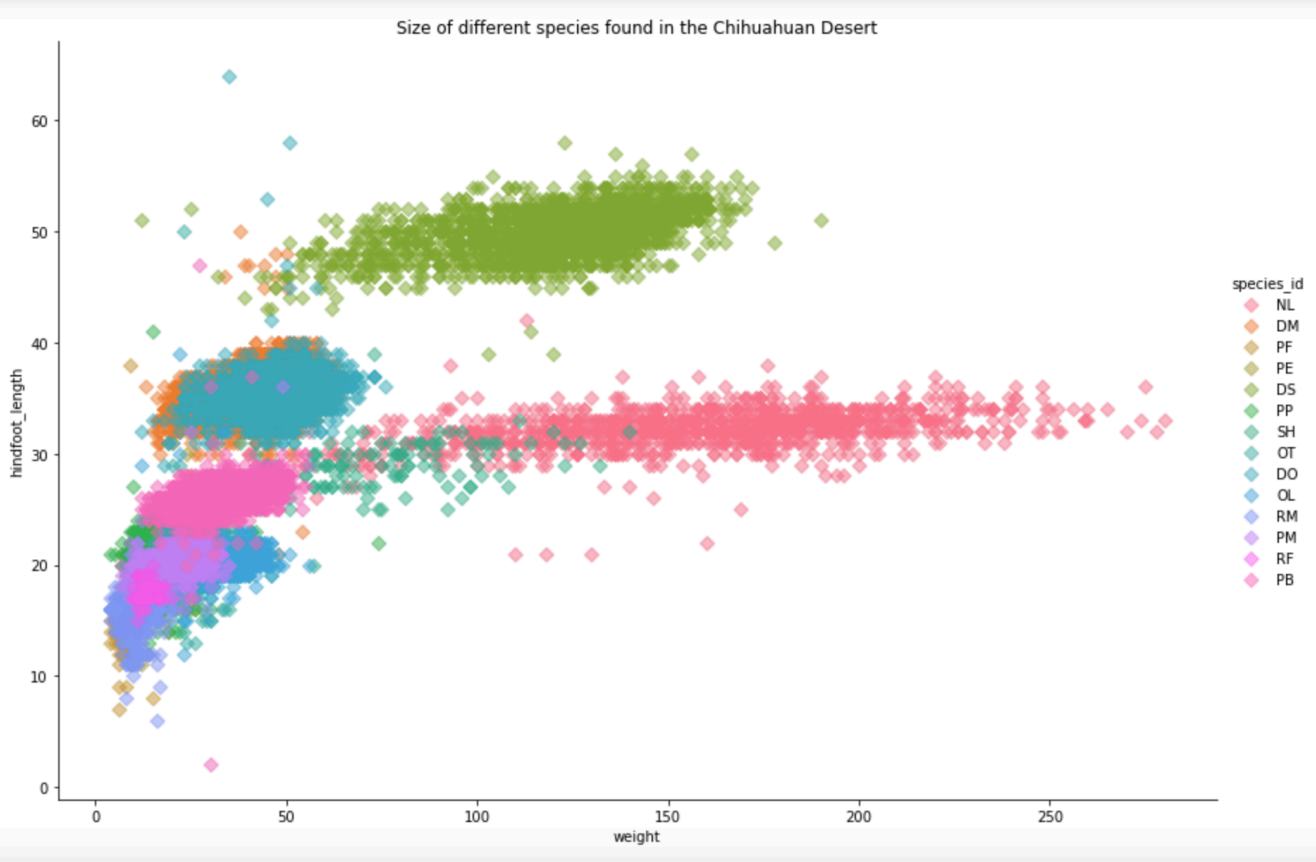
For our last visualization we will use a dataset retrieved from the Portal Teaching database. This data is a subset from the ecological study by Ernst et al. (2009): Long-term monitoring and experimental manipulation of a Chihuahuan Desert ecosystem near Portal, Arizona, USA. This version has only the complete observations, without missing entries for any of the columns. https://figshare.com/collections/Long-term_monitoring_and_experimental_manipulation_of_a_Chihuahuan_Desert_ecosystem_near_Portal_Arizona_USA/3301073

- **scatter plot**: identify relations between 2 variables. can be used when the dependent(y) variable can have multiple values for the independent(x) variable.

```
# Import surveys_complete.csv
surveys = pd.read_csv('data/surveys_complete.csv', index_col=0)

# Graph Scatter plot using Seaborn
sns.lmplot(x="weight", y="hindfoot_length", data=surveys, fit_reg=False)
plt.title('Size of different species found in the Chihuahuan Desert')
plt.show()
```



```
    # Here we can distinguish between the different species using the hue parameter
    sns.lmplot(x="weight", y="hindfoot_length",
       data=surveys, fit_reg=False, height=8,
       aspect=1.5, scatter_kws={'alpha':0.5,"s": 50},
       hue='species_id', markers='D')
    plt.title('Size of different species found in the Chihuahuan Desert')
    plt.show()
```



Exercise #3

1. Using the Chihuahuan Desert ecosystem make a line plot comparing the hindfoot length of M/F over time.
2. Make a Box plot summary for each plot using the column `plot_id`.

Citations:

1. Based on Paul Villanueva's 2018 Python tutorial https://nbviewer.jupyter.org/github/pommevilla/p3.bootcamp.python.2018/blob/master/lessons/P3Bootcamp2018-00.ipynb
2. Used BCB546 class material https://eeob-biodata.github.io/BCB546X-python/
3. https://docs.python.org/3.6/tutorial/
4. https://www.analyticsvidhya.com/blog/2021/02/an-intuitive-guide-to-visualization-in-python/