IntroPython

March 22, 2024

1 About Python, R and RStudio Briefly

`Python` is a popular programming language commonly used for statistical analysis, data visualization, and machine learning. The many ways to interact with `Python` are called Integrated Development Environments (IDE's). IDE choice depends on the context, the nature of the project, and/or personal preference, among other factors. Here are a few IDE's for interacting with `Python`.

- IDLE (Python's Integrated Development and Learning)
- PyCharm
- Jupyter Notebook
- Visual Studio Code
- Google Colab
- RStudio

Additionally, `Python` has broad applications beyond data analysis and can be used for web development, automation, and more. Ultimately, the choice between `R` and `Python` will depend on your specific needs and preferences. If you decide to use `RStudio` as your IDE of choice, RStudio, R, and Python all need to be installed on your computer. Otherwise the any code in a `Python` *chunk* will work in any `Python` IDE. Both `R` and `RStudio` are cross-platform, so that everyone's versions look and operate the same regardless of their operating system!

As getting `Python` to run in `RStudio` is not always seemless, we will be working in a Jupyter Notebook today. We encourage you to follow the instructions from Data Carpentries to install Anaconda and `plotnine`, which you can find here: https://datacarpentry.org/python-ecology-lesson/index.html#installing-python-using-anaconda, or follow the instructions below. You will need to have these installed before attending the workshop.

If you did not complete that or do not have a computer available, you can also borrow a laptop from the MSU library (or maybe from a good friend) for the workshop. If you borrow a laptop from the MSU library, you will need to use a cloud solution to open the Juypter Notebook. There are several options out there including: - Anaconda Cloud - Google Colab - Many others found by a quick internet search.

1.1 Prepare for the Workshop Using Anaconda Navigator

Please take these steps if you haven't already done so:

1. Download and install `Anaconda Navigator` (https://www.anaconda.com/anaconda-navigator)
2. Download the ZIP file provided for the workshop

3. ***Unzip the file and save the contents locally***
4. Install `plotnine` using `Anaconda Navigator` and the package navigator.
    a. Open `Anaconda Navigator`,
    b. On the left, click on "Environments",
    c. Change the drop down menu from "Installed" to "All",
    d. In the "Search Packages" box, type `plotnine`,
    e. Check the box in front of `plotnine`,
    f. Click "Apply" at the bottom of the window,
    g. Follow the dialogs to install the package.
5. Go back "Home" and launch "Jupyter Lab".
6. Open the "IntroPython.ipynb" file from the zipped folder.
7. You should see exactly what is here in this PDF or Juypter Notebook.

1.2 Working in JupyterLab

The document we provided for you is an iPython Notebook (.ipymd) document. It allows you to work in a reproducible fashion, with both code (placed in what are called code chunks) and descriptions of results in the same file. The boxes and grey sections indicate a chunk.

Jupyter Notebooks have three different kinds of chunks, Markdown (like this chunk), Code (for executable code), and Raw (for visualization of code that will not be executed). The Notebook will either be in edit mode or in compiled mode.

For instance if I want to show you what a code chunk will look like from a command prompt I would use a Raw chunk. »> 2 + 2 4 »> If I want to include a code chunk that we can excute directly from this iPython Notebook, then I will use a code chunk. Here is that same calculation (see below).

To execute this code chunk you can press SHIFT + ENTER or SHIFT + RETURN, or click the "Play" button at the top of the window.

[1]:
```
2 + 2
```

[1]:
```
4
```

Type all of your code in these code chunks and other documentation and interpretation of results in Markdown chunks (like we are doing here). You should start today with saving this .ipynb file into a folder that also contains any data and other figures you might want to read into this document. When you do that, JupyterLab will know where to look to read in the data so you do not need to know the path for its physical location on your computer. Once we are done with our local work, we can save the notebook and export as a PDF (File > Save and Export Notebook as > PDF).

A few last things, the default chunk type is "Code", and you can preview your notebook at any time by clicking on "Notebook" in the upper right corner.

1.3 Calculator

For each of the following commands, confirm that the response is the correct answer.

[2]:
```
1 + 2
```

[2]:
```
3
```

[3]:
```
16 * 9
```

[3]:
```
144
```

[4]:
```
20 / 5
```

[4]:
```
4.0
```

[5]:
```
18.5 - 7.21
```

[5]:
```
11.29
```

[6]:
```
2 ^ 2
```

[6]:
```
0
```

Notice how `2^2`≠ (is not equal to) `4`! In `Python` the exponent operator is not the expected *caret* `^` symbol, but rather exponentiation is defined by *double star* `**`:

[7]:
```
2 ** 2
```

[7]:
```
4
```

Also notice algebraic operations are not space-sensitive, this is helpful when trying to format, and read code in `Python`.

We had all of those calculations in separate code chunks. If we want to have multiple commands in one chunk and display the results for all of the commands then we need to run the following commands:

[8]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

[9]:
```
1 + 2
16 * 9
20 / 5
18.5 - 7.21
2 ** 2
```

[9]: 3

[9]: 144

[9]: 4.0

[9]: 11.29

[9]: 4

1.4 Packages

Like `R`, `Python` has a wide range of libraries and packages (often referred as `Modules`) that make it a powerful tool for data analysis. You can import modules into a Python environment or chunk using the `import` function. You can also specify particular components to import if you do not want or need to import the whole module.

In the code above, from the `IPython.core.interactiveshell` module, we just imported `InteractiveShell` Then we were able to set the `InteractiveShell.ast_node_interactivity` parameter to "all".

Another package that we will need for data science is `pandas`. You might get tired of writing `pandas` out all of the time. Fortunately, `Python` lets you provide packages with aliases!

[10]:
```
import pandas as pd
```

This will import the pandas package and give it an alias `pd`, which you can use to access its functions and objects. Aliases are useful because they are fewer characters to type as you have to use either the package name or alias anytime you call a function. ***So choose your alias wisely!***

Now that we know how to import and load `Python` modules, let us work with some examples of commonly used modules, such as the `math` module. The math module is a built-in module in `Python` that provides various mathematical functions and constants such as the square root, trigonometric, logarithmic, and exponential functions. The general syntax for `Python` when calling a function from a specific module is `module.attribute`.

Try the following code:

[11]:
```
import math
math.sqrt(2)

math.cos(math.pi)

math.factorial(4)

math.cos(math.pi/4) ** 2 + math.sin(math.pi/4) ** 2
```

[11]: 1.4142135623730951

[11]: -1.0

```
[11]: 24
```

```
[11]: 1.0000000000000002
```

2 Creating Objects

These operations, however, are not very interesting. To do more useful things in `Python`, we need to assign values to an object. To create an object, we provide the object's name, followed by an equal sign (`=`), and finally the value of the object. This would look something like this: x = 6
Remarks:

- If you are familiar with `R` language you might think of using the assignment arrow `<-` instead of `=`, however, in `Python` this is not a proper syntax.
  - `Python` is case sensitive, so if you name your variable `cat` but then try to run the code `Cat + 2`, you will get an error saying that `Cat` does not exist.
  - You also want your object's name to be explanatory, but not too long. Think `current_temperature` verses `current_temp`. Do you really want to type out temperature every time?
  - Finally, you should not begin any object's name with a number. You can end a name with a number (e.g. clean_data2), but does that give you much information about what is in the contents of `clean_data2` relative to `clean_data`?
  - The name cannot contain any punctuation symbols, except for `.` and `_` (`.` is not recommended)
  - You should not name your object the same as any common functions you may use (`mean`, `stdev`, etc.) although Pythons module.attribute function access should prevent most overwritting issues.
- Using a consistent coding style makes your code clearer to read for your future self and your collaborators.

2.1 Clean Code

Yes, writing code may be completely new to you, but there is a difference between code that looks nice and code that does not. Generally, object names should be nouns and function names should be verbs. It is also important that your code looks presentable, so that a friend/college/professor can read it and understand what you are doing. For these reasons, there are style guides for writing code in `Python`. The two main style guides are Google's (link) and PEP 8, the official Python style guide, maintained by the Python Software Foundation: (link)

2.2 Working with Objects

When you assign a value to an object (like we did previously) `Python` does not output anything by default. If you want to see the output you can use the command `print(value)`, then `Python` will output the value of the object you created.

[12]:
```
x = 6
```

[13]:
```
w = 9
print(w)
```

```
9
```

Once the object has been created, you can use it! Run the following lines of code:

[14]:
```
2.2 * x

4 + x
```

[14]:
```
13.200000000000001
```

[14]:
```
10
```

We can also overwrite an object's value, so that it has a new value. In the code below create a new object `y` and then we give `x` a new value of 2.

[15]:
```
y = x + 6

x = 2
```

Exercise 1: What is the current value of `y`? 12 or 8?

[16]:
```
# Exercise 1 code here!
```

3 Working with Different Data Types

A vector in `Python`, formally a `list` or a `NumPy` array is the basic data type in `Python`. A vector is a series of values, which can be either numbers or characters, but every entry of the vector must be the same data type. `Python` can tell that you are building a vector when you use the squared brackets separating each element with a comma `[a,b,...,c]`, `list()` or `numpy.array([])` functions, which concatenates a series of entries together.

Install the `Numpy` module in `Python` command window using the syntax `!pip install` (this may take a few moments):

[17]:
```
!pip install numpy
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: numpy in c:\programdata\anaconda3\lib\site-
packages (1.26.4)
```

Remarks: *When you include `!` at the beginning of a command in a `Python` script, `Notebook` will treat it as a shell command rather than a Python command.

`+ A shell command is a command that is executed in a command-line interface (CLI) or shell, which is a text-based user interface for interacting with an operating system or software application.`

```
+ In a shell environment, you can type commands and execute them by pressing
the Enter key.
+ The shell interprets the command and executes it, displaying the output
in the console or terminal window.
```

[18]:
```
import numpy as np

temps_list = [50, 55, 60, 65]
temps_list

temps_numpy = np.array([50, 55, 60, 65])
print(temps_numpy)
```

[18]:
```
[50, 55, 60, 65]
```

```
[50 55 60 65]
```

To make a vector of characters, you are required to use quotation marks `" "` to indicate to `Python` that the value you are using is not an object you already created in `Python`.

[19]:
```
animals_list = ["cat", "dog", "bird", "fish"]
animals_list

animals_numpy = np.array(["cat", "dog", "bird", "fish"])
print(animals_numpy)
```

[19]:
```
['cat', 'dog', 'bird', 'fish']
```

```
['cat' 'dog' 'bird' 'fish']
```

Important features of a vector is the type of data they store. Run the following lines of code and decide what type of data the vectors contain.

[20]:
```
type(temps_list)

type(temps_numpy)
```

[20]:
```
list
```

[20]:
```
numpy.ndarray
```

Remarks:

- `numpy.ndarray` is a `Python` class that represents an n-dimensional array or a multi-dimensional array. An array is a data structure that stores a collection of values of the same type.
- `list` is a `Python` class type that represents a list data structure in Python. A list is an ordered collection of objects or values, enclosed in square brackets and separated by commas. A list can contain objects of different data types, such as integers, floats, strings, or even other lists.

Exercise 2: Create a vector, named `dec`, that contains decimal valued numbers. Then check what data type that vector contains?

[21]:
```
# Exercise 2 code goes here
```

Another possible data type are dictionaries. In `Python`, a dictionary is an unordered collection of key-value pairs. It is represented by a pair of curly braces `{}`, with each key-value pair separated by a colon `:`. *Also known as associative arrays, maps, or hash tables in other programming languages.*

[22]:
```
translate = {'Hello':'Hola', 'How are you?':'Como estas?', 'Fibula':'Perone'}
translate['Fibula']
```

[22]:
```
'Perone'
```

So in `Python`, vectors (lists, numpy arrays) and dictionaries are data structures that allow you to store collections of values, but they have some key differences:

1.- Vectors are ordered while Dictionaries are not.

2.- Vectors are indexed by integer positions while Dictionaries are indexed by keys.

3.- All elements in a vector are of the same type, while in Dictionaries this need not be the case.

4.- You can "mutate" (change) the values contained in a vector or dictionary, BUT you cannot change the keys in a dictionary once they are assigned, i.e., they are "immutable".

[23]:
```
# Creating a vector
vec = [1, 2, 3, 4]

# Accessing values in a vector by index
print(vec[0])  # Output: 1


# Modifying a value in a vector
vec[0] = 5
print(vec[0])

# Creating a dictionary
translate = {'Hello':'Hola', 'How are you?':'Como estas?', 'Fibula':'Perone'}

# Accessing values in a dictionary by key
print(translate['How are you?'])  # Output: 3

# Adding a new key-value pair to a dictionary
translate['I like cheese'] = 'Me gusta el queso'
print(translate['I like cheese'])

# Modifying a value in a dictionary
translate['Hello'] = 'Adios'
print(translate['Hello'])
```

```
print(translate)
```

```
1
5
Como estas?
Me gusta el queso
Adios
{'Hello': 'Adios', 'How are you?': 'Como estas?', 'Fibula': 'Perone', 'I like
cheese': 'Me gusta el queso'}
```

Another possible data type is a logical (Boolean) value. This type of data takes on values of `True` and `False`(*WARNING* `Python` is case sensitive so `TRUE`, `FALSE` won't work). But, we said that vectors could only be numbers or characters. If `TRUE` and `FALSE` don't have quotations around them, then they aren't characters. So, then they must be numbers. What numbers do you think they are?

[24]:
```
logic = [True, False, False, True]

type(logic)
```

[24]:
```
list
```

Notice that the `type()` command is telling us what we already know, it's a list! But is we are interested in knowing the type of each element in a list one can perform a for loop, but if we know that all elements of a list are of the same type then we can just look at the type of any element in that list:

[25]:
```
type(logic[0])
```

[25]:
```
bool
```

[26]:
```
diff_types = [False, "Space", 2]
for i in diff_types:
  print(type(i))
```

```
<class 'bool'>
<class 'str'>
<class 'int'>
```

Another instance of a data type in `Python` is called a *tuple* which is an ordered, immutable collection of elements. Like lists, tuples can contain elements of different data types, including numbers, strings, and other objects. However, unlike lists, *tuples cannot be modified once they are created*. This means that you cannot add, remove, or modify elements in a tuple after it has been defined. *Tuples are defined using parentheses*, and elements are separated by commas:

[27]:
```
tuple_ex = (1, 'apple', True)
```

What do you think the output of the following code will be?

[28]:
```
#tuple_ex[0] = 4
```

Exercise 3: What happens when we try to mix different data types into one vector? Speculate what will happen when we run each of the following lines of code:

[29]:
```
num_char = [1, 2, 3, "a"]

# for i in num_char:
#   print(type(i))

num_logic = [1, 2, 3, False]

#type(num_logic)
# for i in num_logic:
#   print(type(i))

char_logic = ["a", "b", "c", True]

guess = [1, 2, 3, "4"]
```

In each of these vectors, the two types of data were *coerced* into a single data type. This happens in a hierarchy, where some data types get preference over others. Can we draw a diagram of the hierarchy?

Remarks:

In `Python` an `R` list is equivalent to a dictionary. In `R` a `Python` list is equivalent to vector.

What about a list of lists in 'Python'?

[30]:
```
OS = ["Mac", "Windows", "Linux"]
fav_nums = [12, 13, 17]
logic = [True, False, False]

layered_list = [OS, fav_nums, logic]
layered_list[1]

layered_list[0][1]
```

[30]:
```
[12, 13, 17]
```

[30]:
```
'Windows'
```

So to access the elements of list within a list we use the syntax `list[list_i][element]`.

Remarks:

- We have been outputting values with `print()` and by just running the name of the variable, why can we do this?
  - When running code in a Jupyter notebook or in an interactive Python shell (such as `RStudio`), you can simply type the name of a variable or expression to see its value printed. This is because the notebook automatically displays the output of any expression that is executed without explicitly requiring the use of `print()`.

– When running a Python script from the command line or in an IDE, you must use print() to display output. If you don't use `print()`, the output will not be displayed.

3.1 Named Lists

To create a named list in Python, you can use a dictionary:

[31]:
```
my_named_list = {'title': 'statistics', 'numbers': list(range(1, 11)),
'data': True}
print(my_named_list)
```

```
{'title': 'statistics', 'numbers': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 'data':
True}
```

4 Importing Data

Recall the module pandas mentioned earlier? pandas is a popular Python library used for data manipulation and analysis. It provides data structures for efficiently storing and manipulating large, structured data sets and includes functions for data cleaning, merging, filtering, and appending.

If you haven't already, import the pandas module as pd, and if you are working in a folder that contained all of the files we sent you (extracted from the zip file) then you should be able to run the following code:

[32]:
```
import pandas as pd
BlackfootFish = pd.read_csv('BlackfootFish.csv')
```

The path provided can also be simplified to just include the file name - if the .Rmd and data are saved in the same folder. If you have any trouble reading in the data set, here is code that allows you to read the data from our github repository:

[33]:
```
#BlackfootFish = pd.read_csv("https://github.com/saramannheimer/data-science-r\
#-workshops/raw/master/Introduction%20to%20R/AY%202020-2021/Student%20Version/
#\BlackfootFish.csv")
```

5 Structure of Data

The data we will use is organized into data tables. When you imported the BlackfootFish data into Python it was saved as an object. You are able to inspect the structure of the BlackfootFish object using functions built in to Python (no packages necessary).

Run the following code. What is output from each of the following commands?

[34]:
```
print(type(BlackfootFish)) # Class of the object

print(BlackfootFish.shape) # Number of rows and columns

print(BlackfootFish.columns) # Column names
```

```
print(BlackfootFish.dtypes) # Data types of each column and the first few rows␣
  ↪of data
```

```
<class 'pandas.core.frame.DataFrame'>
(18352, 7)
Index(['trip', 'mark', 'length', 'weight', 'year', 'section', 'species'],
dtype='object')
trip         int64
mark         int64
length     float64
weight     float64
year         int64
section     object
species     object
dtype: object
```

[35]:
```
BlackfootFish.describe()

type(BlackfootFish)
```

[35]:

	trip	mark	length	weight	year
count	18352.000000	18352.000000	18352.000000	16556.000000	18352.000000
mean	1.500763	0.092851	262.330025	246.220363	1996.626744
std	0.500013	0.290231	99.902006	272.285241	5.922414
min	1.000000	0.000000	16.000000	0.000000	1989.000000
25%	1.000000	0.000000	186.000000	65.000000	1991.000000
50%	2.000000	0.000000	250.000000	150.000000	1996.000000
75%	2.000000	0.000000	330.000000	330.000000	2002.000000
max	2.000000	1.000000	986.000000	4677.000000	2006.000000

[35]:
```
pandas.core.frame.DataFrame
```

When we inspect dataframes, or other objects in `Python`, there are some general functions that are useful to check the content/structure of the data. Here are some (assuming you are importing data using `pandas`):

- size:
  - `datasetname.shape` : rows and columns
  - If your data set is a pandas dataframe `datasetname.shape[0]`: number of rows,
  - If your data set is a pandas dataframe `datasetname.shape[1]`: number of columns.
  - If your data set is a pandas dataframe and variable is a column in the dataframe: `len(datasetname['variable'])`: number of columns.
- content:
  - `datasetname.head()`: first 5 rows.
  - `datasetname.tail()`: last 5 rows.
- names:
  - `colnames(datasetname)`: column names.
  - `datasetname.index`: row names.
- summary of content:

- `datasetname.info()`: structure of object and information about the columns.
- `datasetname.describe()`: summary statistics for each column.

6 Dataframes

What is a dataframe? A dataframe is a two-dimensional table-like data structure, similar to a spreadsheet. It is a primary data structure provided by the pandas library. You can create a dataframe in several ways, such as loading data from a CSV, Excel file, SQL database, or by constructing it from scratch using `Python` lists or dictionaries. A dataframe is made up of columns, where each column is a series object. Each column can have a different data type, but all the values in a column must have the same data type.

7 Extracting Data

In `Python`, you can access a specific column of a dataframe using square bracket notation `[]` with the column name inside the brackets. What do you think the following code will output?

[36]:
```
df = pd.read_csv('BlackfootFish.csv')

# Access the 'weight' column
weight_col = df['weight']

print(weight_col)
```

```
0        175.0
1        190.0
2        245.0
3        275.0
4        300.0
         …
18347     35.0
18348     60.0
18349     10.0
18350    215.0
18351     10.0
Name: weight, Length: 18352, dtype: float64
```

[37]:
```
years = BlackfootFish['year']
# extracts year from the dataset and saves it into a new variable named years
years.head

years_str = str(years)
print(years_str)

## How would you determine how long the vector is?
```

[37]:
```
<bound method NDFrame.head of 0        1989
1        1989
2        1989
3        1989
4        1989
         …
18347    1991
18348    1991
18349    1991
18350    1991
18351    1991
Name: year, Length: 18352, dtype: int64>
```

```
0        1989
1        1989
2        1989
3        1989
4        1989
         …
18347    1991
18348    1991
18349    1991
18350    1991
18351    1991
Name: year, Length: 18352, dtype: int64
```

Another method for accessing data in the dataset is using the `pandas` function `iloc[]`. You can (roughly) view the dataset as a matrix of entries, with variable names for each of the columns. I could instead use `iloc` function to perform the same task as above, using the following code,

[38]:
```
# Extract the values in the fifth column and store them in a variable called␣
↪"years"
years = BlackfootFish.iloc[:, 4].values
```

7.1 Practice:

The following is a preview of the dataframe `df`:

[39]:
```
df = pd.DataFrame({
  'x': ["H", "N", "T", "W", "V"],
  'y': ["May", "Oct", "Mar", "Aug", "Feb"],
  'z': [2010, 2015, 2018, 2017, 2019]
})

df
```

[39]:

	x	y	z
0	H	May	2010
1	N	Oct	2015

```
2  T  Mar  2018
3  W  Aug  2017
4  V  Feb  2019
```

Exercise 4: What would be output if you entered: `df.iloc[2, :]`?

[40]:
```
# Code for Exercise 4 goes here
```

Exercise 5: What would you input to get an output of `2015`? Can you think of two ways to do it?

[41]:
```
# Code for Exercise 5 goes here
```

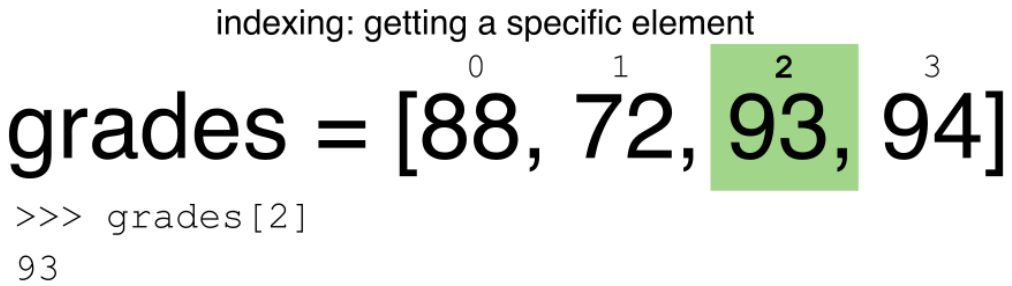
Remarks:

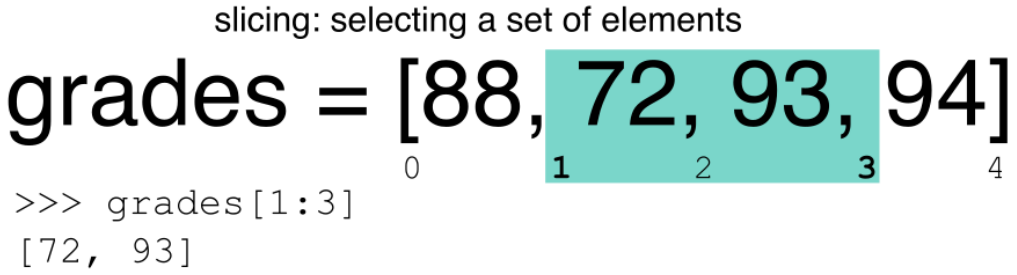
- Notice the difference between `loc` and `iloc`. Both are indexing methods in `pandas` that allow for selection of data from a data frame, how do they differ?
  - `iloc` selects data based on the integer index of the rows and columns, with syntax `df.iloc[row_index, column_index]`.
  - `loc` selects data base on the row and column labels, with syntax `df.loc[row_label, colum_label]`.
  - *WARNING* `iloc` is exclusive of the end index, `loc` is inclusive.
  - `iloc` does not support Boolean indexing, `loc` does.

7.2 Accessing Data

When we have a data frame or a list of objects, we can extract specific items by specifying their position in the data frame or list. We saw how to do this with a single item, row, or column that we want to extract (`df.iloc[1, 5]`, `df.iloc[3,]` or `df.iloc[, 5]`). We can specify two items, rows, or columns by either using the fact that the two things we want are adjacent, or we can combine what we want into a list that we can then use for extracting. We will continue to use the indices to specify what we want to extract.

Python uses zero-based indexing, which means you have to shift your start and stop indices, they might not be exactly as you would intuitively expect. The images below came from datacarpentry.org (https://datacarpentry.org/python-ecology-lesson/03-index-slice-subset.html).





For example, `[0]` or `0:1` could identify the first object in a list, and `[0, 1]` could identify the first two objects in a list. However they are adjacent so we could use `a:b+1` to indicate that we want everything from position `a` to position `b` in the list. So, `[1,3]` would give the same items as `1:3`.

7.3 Quick Check

Run the following code to see what you get. Play around with the first and last numbers to see what happens.

[42]:
```
list([1,2,3,4,5])
list(range(1,6))
list(range(6,1,-1))
list(range(123,131))
list(range(3,-1,-1))

list([1,2,3,4,5])[1:3]
list(range(123,131))[3:6]
list(range(6,1,-1))[2:4]
```

[42]: `[1, 2, 3, 4, 5]`

[42]: `[1, 2, 3, 4, 5]`

[42]: `[6, 5, 4, 3, 2]`

[42]: `[123, 124, 125, 126, 127, 128, 129, 130]`

[42]: `[3, 2, 1, 0]`

[42]: `[2, 3]`

[42]: `[126, 127, 128]`

[42]: `[4, 3]`

When you want adjacent items, using the `a:b` notation is nicer than having to type out all of the values.

What about non-adjacent items?

For non-adjacent items, you just need to fall back on listing everything out. Say you want the first, third, and 9th item in a list. Then you can use `[1, 3, 9]`.

Now that we've talked about how to create lists of the indices we want, how do we use these lists to extract items?

Using our data frame that we created above, to extract rows 1 and 2 we can use the notation `[1,2]` (or `1:3` since they are adjacent).

[43]:
```
df.iloc[[1,2],]
df.iloc[1:3, ]
```

[43]:
```
   x    y     z
1  N  Oct  2015
2  T  Mar  2018
```

[43]:
```
   x    y     z
1  N  Oct  2015
2  T  Mar  2018
```

If we want to extract rows 1 and 3, we can use this list of indices: `[1, 3]`

[44]:
```
df.iloc[[1,3],]
```

[44]:
```
   x    y     z
1  N  Oct  2015
3  W  Aug  2017
```

NOTE: Instead of using indices you can also do the same technique with names if the rows or columns are named.

Exercise 6: How would you pull off only columns `x` and `y`? What about pulling off only columns `x` and `z`?

[45]:
```
# Code for Exercise 6 goes here
```

Exercise 7: How would you modify the script below, to get an output of `22 24`?

[46]:
```
# Code for Exercise 7 goes here
```

Exercise 8: What would be output if you entered: `s[2, ]`?

[47]:
```
# Code for Exercise 8 goes here
```

8 Changing Data Type

In `Python`, when building or importing a dataframe, character columns are usually represented as strings (i.e., text) by default. **Unlike in R, there is no factor data type in Python**. In `Python` you can use the `Pandas` library, which provides functions like `read_csv()` and `read_table()` to import data from CSV. `Pandas` does not convert string columns to any other data type, but it does infer the data type of each column.

If you want to explicitly set the data type of a column or keep a string column as a string, you can use the dtype parameter in `read_csv()` to specify the data type for each column:

[48]:
```
import pandas as np
df = pd.read_csv('BlackfootFish.csv', dtype = {'column':str})
```

Remarks:

- If you are familiar with `R` note that there is no equivalent `stringAsFactors` argument in `read_csv()` from `pandas`.

If you want to determine the unique elements in a column of a dataframe in `Python`, you can use the `unique()` method of `pandas`:

[49]:
```
BlackfootFish['species'].unique()

BlackfootFish['species'].unique
```

[49]:
```
array(['RBT', 'WCT', 'Bull', 'Brown'], dtype=object)
```

[49]:
```
<bound method Series.unique of 0          RBT
1          RBT
2          RBT
3          RBT
4          RBT
         …
18347    Brown
18348    Brown
18349    Brown
18350    Brown
18351    Brown
Name: species, Length: 18352, dtype: object>
```

What are the difference in outputs? Using `unique()` the output is an array of the unique values in the `species` column of BlackfootFish. If we leave out the parenthesis, i.e., `unique` the output will be the details of the dataframe, including data type, and its length, so no unique values. The key difference is that `unique()` is a **method** from the `pandas` library, that is, some sort of function. On other hand `unique` is an **attribute** (if it exists) of the object.

Exercise 9: Year was saved as an integer data type (1989 - 2006), but we may want to consider it to be a categorical variable (a factor). Write the `Python` code to create a new variable called `yearF` and then check the unique categories for this new variable.

[50]:
```
# Code for Exercise 9 goes here
```

Exercise 10: Now, verify that `yearF` is viewed as a categorical variable, with the same levels as `year`. (hint: you have already used functions that would do this for you)

[51]:
```
# Code for Exercise 10 goes here
```

9 Packages

As we mentioned previously, `Python` has many packages (or modules), which people around the world work on to provide and maintain new software and new capabilities for `Python`. We have already shown some examples like `panda`, `numpy`, and `math` and how to load them, we give a formal description now. You will slowly accumulate a number of packages that you use often for a variety of purposes. In order to use the elements (data, functions) of the packages, you have to first install the package (only once on a given computer) and then load the package (every time).

You can install any `Python` library as illustrated with the following example:

[52]:
```
#Install statsmodels
#!pip install scipy
#!pip install numpy
from scipy import stats as stats
import numpy as np

# generate random numbers from a normal distribution
mu, sigma = 0, 1
samples = stats.norm.rvs(loc=mu, scale=sigma, size=1000)

# compute summary statistics
mean = np.mean(samples)
std = np.std(samples)

print(f'The mean is {mean} and the standard deviation is {std}')
```

```
The mean is 0.0010456403980880751 and the standard deviation is
0.9901703180265482
```

Here we have installed the package/module `scipy.stats` which provides a wide range of probability distributions. The module includes functions for working with both continuous and discrete distributions. The general syntax is `stats.distribution`, in the above example `stats.norm.rvs(loc, scale, size)` generates a sample of size = n of random numbers from a normal distribution where `loc` (location) specifies the mean of the distribution, and `scale` specifies the standard deviation.

Moreover, notice that we made use of the `numpy` module, as it contains functions such as `mean` and `std` that can be applied to Numpy arrays. Finally we print out the output.

Additional Comments: For printing the output we used a `Python f-string` which is used to print a sentence along with a calculated value of a variable, with syntax `print(f'The result is {variable}')`.

10 Finding Help

`Python` has a great collection of packages and documentation, making it a popular choice for data science and scientific computing. Some of the most commonly used modules for statistical analysis, machine learning, data visualization, and much more are `NumPy`, `Pandas`, `SciPy`, `scikit-learn`, `Matplotlib`, among others. If you need a function to complete a task (say find the variance), but are not quite sure how it's spelled, what arguments it takes, or what package it lives in, don't fret!

`Python` has a built-in `help()` function used to get information about `Python` functions, modules, and classes.

Another great resource for finding help is *Stack Overflow* Python's community, which is a good place to start as a beginner to learn and use `Python`.

[53]:
```
#import scipy
#help(scipy)
```

11 Functions

In `Python` there are both functions that are built in (require no package to be loaded), as well as functions that are housed within specific packages. You have already used a few built in functions to inspect the structure of the BlackfootFish data (`print`, `type`, `describe`).

As you may know, a function transforms an input (potentially multiple) into an output. You have to provide `Python` with the inputs (arguments) required for the function to generate an output. The argument(s) inside a function happen after the `(` symbol. One way to identify that an object is a function when it is immediately followed by a `(` and the corresponding closing `)` comes after the arguments are complete. The output of a function does not have to be numerical and it typically is not a single number, it can be a set of things or a dataset.

Suppose we wanted to create a vector of 10 zeros. To do this, we would use the `np.array` function:

[54]:
```
import numpy as np
# repeating 0 ten times
np.array([0] * 10 )
[0] * 10

# switching order of arguments
np.array(10 * [0])
10 * [0]
```

[54]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

[54]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

[54]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

[54]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Now let's look over some other functions that are often used:

[55]:
```
# takes a numerical input, but there are NA's in our data
np.mean(BlackfootFish['weight'])

# notice Python ignores NAs automatically.
np.mean(BlackfootFish['weight'].dropna())

# gives an error because the input is not the correct data type
```

```
# np.median(BlackfootFish['species'])


np.corrcoef(BlackfootFish['length'], BlackfootFish['weight'])

# Does corrcoef have an option to remove NA's? No, np.corrcoef does not have an␣
 ↪option
# to remove NA's, you must deal with missing values before calling the function.
```

[55]:
```
246.22036337279536
```

[55]:
```
246.22036337279536
```

[55]:
```
array([[ 1., nan],
       [nan, nan]])
```

One way to remove the missing values and then compute the correlation coefficient is:

[56]:
```
valid_mask = ~np.isnan(BlackfootFish['length']) & ~np.
 ↪isnan(BlackfootFish['weight'])
length_valid = BlackfootFish['length'][valid_mask]
weight_valid = BlackfootFish['weight'][valid_mask]

corr_coeff = np.corrcoef(length_valid, weight_valid)[0,1]

print(f"The correlation coefficient is: {corr_coeff}")
```

```
The correlation coefficient is: 0.8856327037417443
```

Let's break it down line by line:

- Line 1: `np.isnan()` finds the data points that have missing values in the data. `~` is a logical operator that inverts/negates, so `~np.isnan()` will return `True` for all data points with no NA's. Now this will create a "mask" which is a logical (boolean) array. The symbol `&` will combine the two masks into one.
- Line 2 and 3: We use the index of `valid_mask` to create `length` and `weight` variables with no missing values.
- Line 4: `np.corrcoef()` returns a correlation matrix where the (0,0) entry is the correlation coefficient between the first array with itself. The entry (1,1) is the correlation coefficient between the second array with itself. The entries (0,1)=(1,0) are the correlation coefficient between both arrays, hence the `[0,1]`.
- Line 5: We then use an `f-string` to print the output in a nice way.

As seen in the functions above, some functions have *optional* arguments. If they are not specified by the user, then they take on their default value (`True` for `dropna`). These options control the behavior of the functions, such as whether it includes/excludes NA values.

12 Cleaning Data

In many instances, you will deal with data that are not "clean". Based on the output we received from the `np.mean()` function, we know that there are NA's in the BlackfootFish data, possibly across a variety of variables. Before we used `dropna` as an option to remove NA's *within* a function. We can use the same command on the entire data frame before applying any functions to it. Based on the output below, how many rows in the BlackfootFish data have an NA present?

[57]:
```
BlackfootFish.shape # gives the dimensions of the dataset in (row, column)␣
↪format

BlackfootFish.dropna().shape
# .dropna() is method in Pandas which removes missing values (NaN).
```

[57]:
```
(18352, 7)
```

[57]:
```
(16556, 7)
```

Remark: The computer is using an algorithm to return a dataset with no NA values anywhere in it. This algorithm goes through every row of the dataset and (roughly) has the following steps,

- Inspect each row or column to check for missing values (an alternative to `dropna` is the `isnull` method).
- If a row or column contains NaN's, it is dropped (removed) from the data frame.
- Once it has stepped through every row, the function outputs the "cleaned" dataframe

If we wish to remove all of the NA's from the dataset, we can use the `dropna` command from above. We can save the new "clean" dataset under a new name (creating a new object) or under the same name as before (replacing the old object with the new object).

[58]:
```
BlackfootFish_clean = BlackfootFish.dropna()
# Creates a new dataframe, where the NA's have all been removed
```

13 Data Visualization

There are many different genres of data graphics, with many different variations on each genre. Here are some commonly encountered kinds:

- **scatterplots**: showing relationships between two quantitative variables
- **distributions**: showing distributions of a single quantitative variable
- **bar charts**: displaying frequencies or densities of a single categorical variable

NOTE:

For plots, we do not want to display the output from all lines of code, just the final plot, so we'll change the parameter for how the output is displayed from a code chunk.

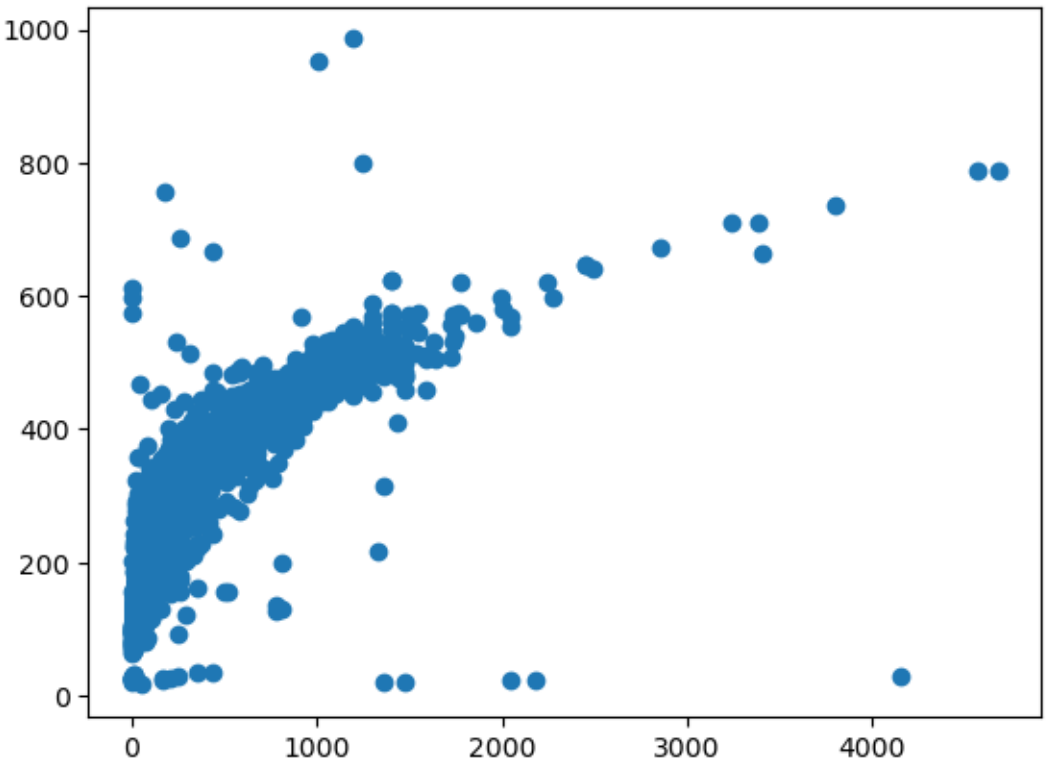
[59]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "last"
```

13.1 Scatterplots

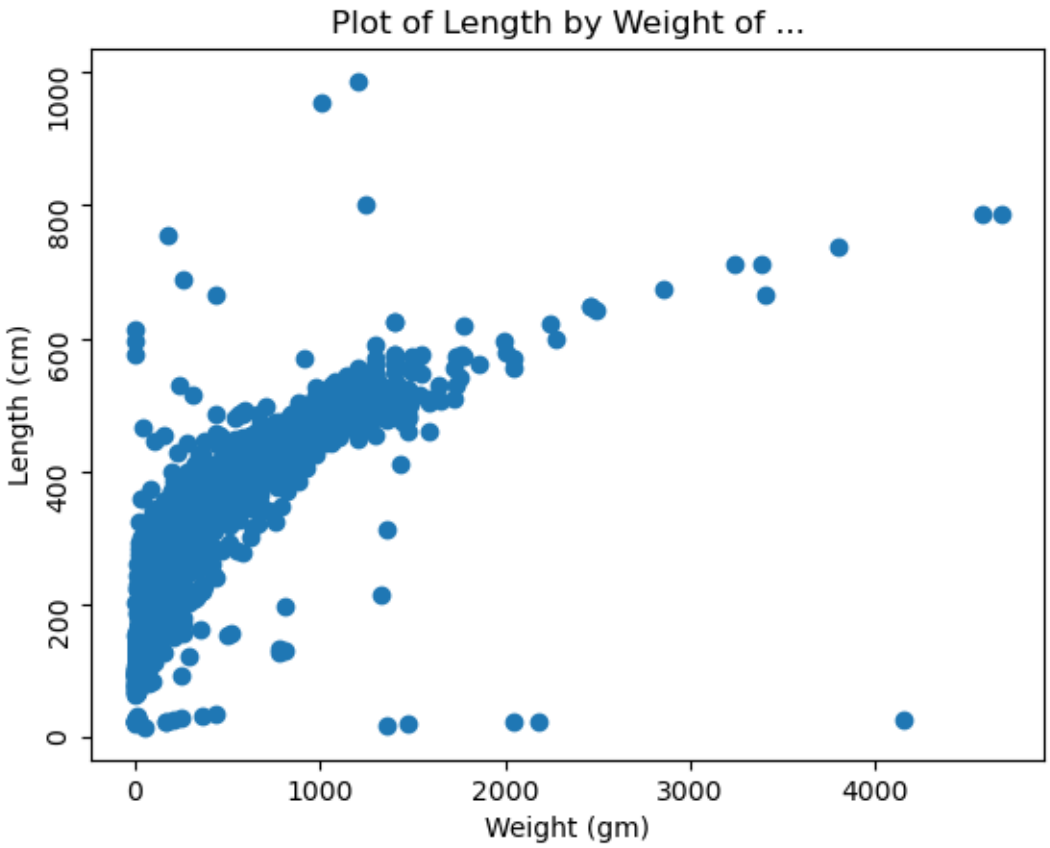
The main purpose of the scatterplot is to show the relationship between two variables across several or many cases. Most often, there is a Cartesian coordinate system in which the x-axis represents one variable and the y-axis the second variable.

[60]:
```
#!pip install matplotlib
import matplotlib.pyplot as plt

plt.scatter(BlackfootFish_clean['weight'], BlackfootFish_clean['length'])
plt.show()
```



[61]:
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots() # creates a new figure and an axes object to plot data␣
 ↪on
ax.scatter(BlackfootFish_clean['weight'], BlackfootFish_clean['length'])
ax.set_xlabel('Weight (gm)') # adding in axis labels
ax.set_ylabel('Length (cm)')
ax.tick_params("y", labelrotation=90)
ax.set_title('Plot of Length by Weight of ...') # adds a title                ␣
 ↪
plt.show()
```



Let's breakdown the following line `ax.tick_params("y", labelrotation=90)`:

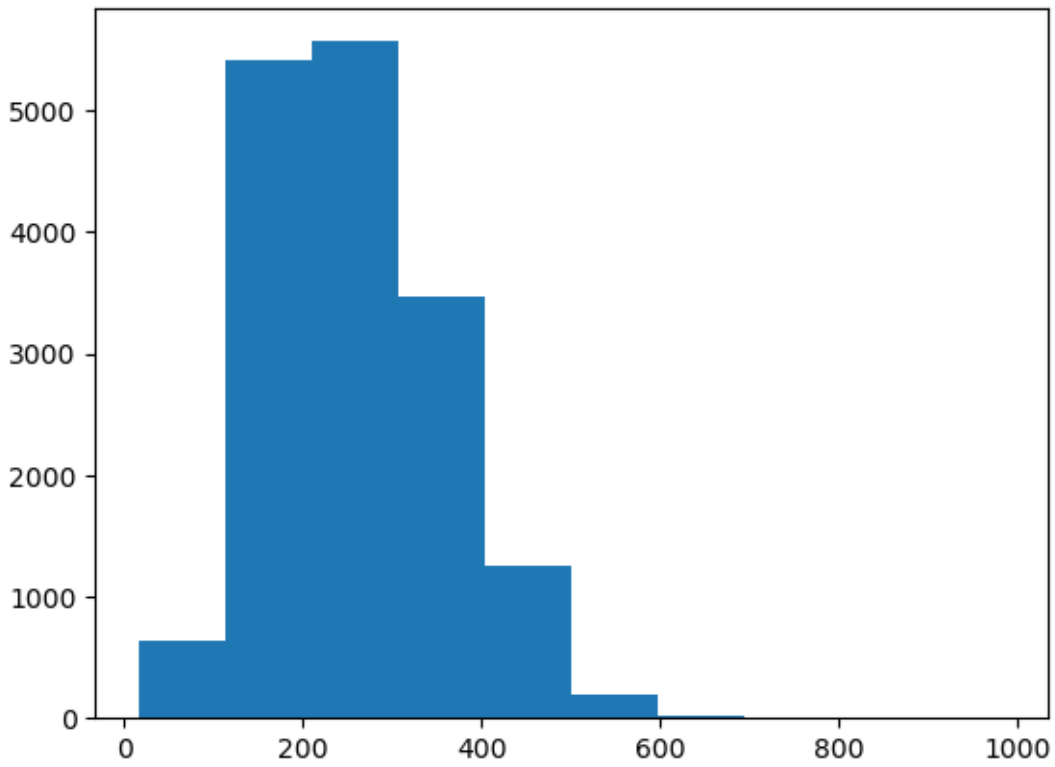
- `ax` is for `Axes` which is an object in `Matplotlib`, representing, you guessed it, the plot's axes.
- `ax.tick_params()` is a method of `Axes` class, which set the parameters for the "ticks" for either (or both) axes.
- `"y"` indicates we want to set a paramter for the "y" axis only.
- `labelrotation = 90` sets the rotation for the tick labels.

13.2 Distribution

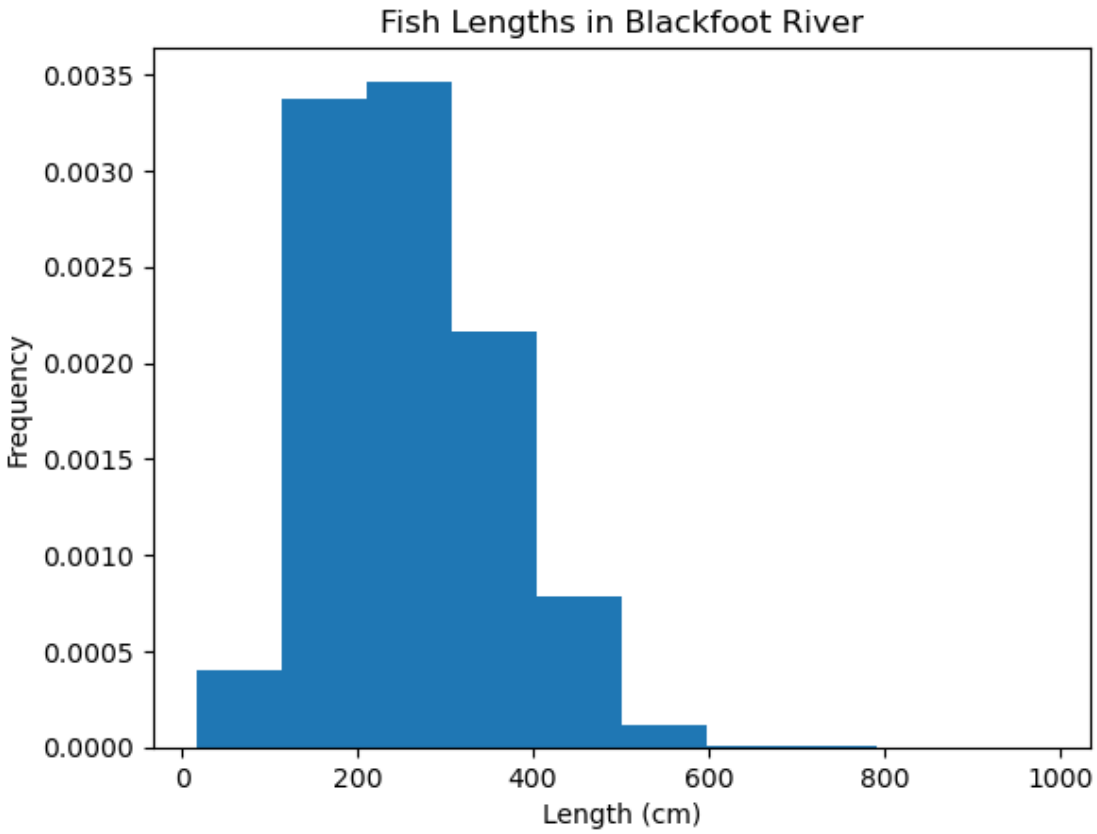
A histogram shows how many observations fall into a given range of values of a variable and can be used to visualize the distribution of a single quantitative variable.

By default, Matplotlib will use the current figure and axes to plot any new data, unless you explicitly create a new figure and axes object.

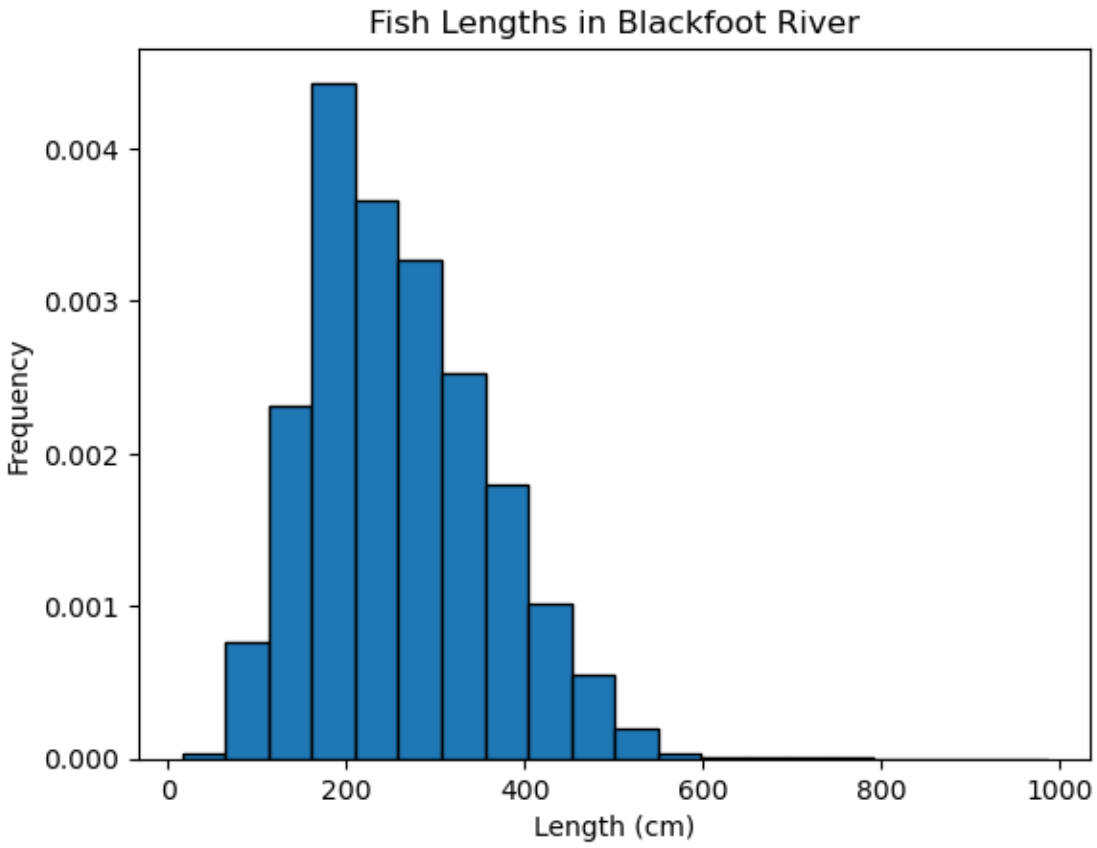
[62]:
```
fig, ax_hist = plt.subplots()
ax_hist.hist(BlackfootFish_clean['length']);
plt.show()
```



[63]:
```
fig, ax_hist = plt.subplots()
# converts to a density plot (area adds to 1)
ax_hist.hist(BlackfootFish_clean['length'], density=True);
# adds x-axis label
ax_hist.set_xlabel('Length (cm)')
ax_hist.set_ylabel('Frequency')
# adds title to plot
ax_hist.set_title('Fish Lengths in Blackfoot River')
plt.show()
```



[64]:
```
fig, ax_hist = plt.subplots()
# converts to a density plot (area adds to 1)
ax_hist.hist(BlackfootFish_clean['length'], density=True, edgecolor = 'black',
bins = 20);
# adds x-axis label
ax_hist.set_xlabel('Length (cm)')
ax_hist.set_ylabel('Frequency')
# adds title to plot
ax_hist.set_title('Fish Lengths in Blackfoot River')
plt.show()
```

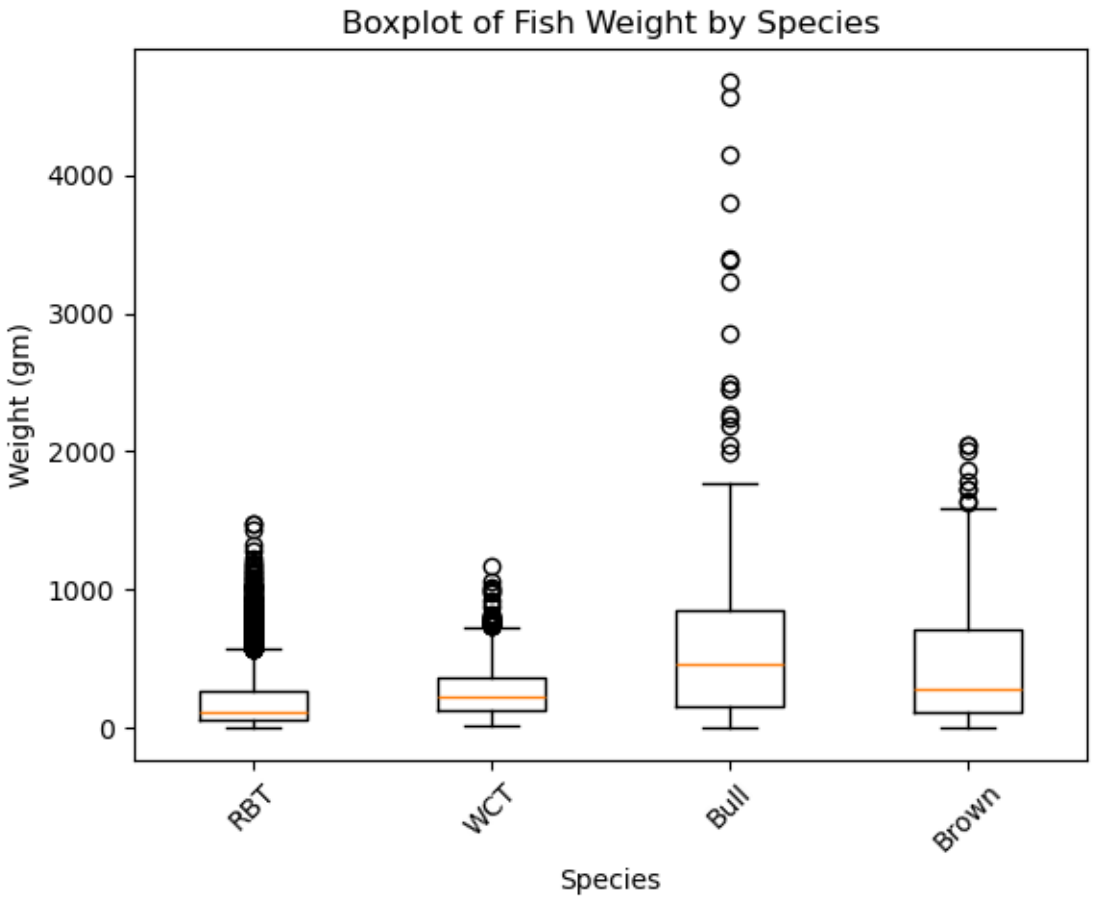


13.3 Side-by-Side Boxplots

The familiar boxplot is a simple display to use when the objective is to compare the distribution of a quantitative variable across different levels of a categorical variable.

[65]:
```
# Create a list of weight data for each species
weight_data = [BlackfootFish_clean[BlackfootFish_clean['species'] ==␣
 ↪s]['weight']
for s in BlackfootFish_clean['species'].unique()];

fig, ax_box = plt.subplots()
ax_box.boxplot(weight_data);
ax_box.set_xticklabels(BlackfootFish_clean['species'].unique(), rotation=45);
ax_box.set_xlabel('Species')
ax_box.set_ylabel('Weight (gm)')
ax_box.set_title('Boxplot of Fish Weight by Species')
plt.show()
```



Let us explore the line `[BlackfootFish_clean[BlackfootFish_clean['species'] == s]['weight'] for s in BlackfootFish_clean['species'].unique()];`: The code creates a list of weight data for each unique species in the following way:

- `BlackfootFish_clean['species'] == s` selects the rows where the `species` column matches a particular (s) unique species name.
- `BlackfootFish_clean['species'].unique()` generates such s values.
- `BlackfootFish_clean[...]['weight]` selects only the `weight` column. In particular the weight data for each unique species s.

13.4 Bar Charts

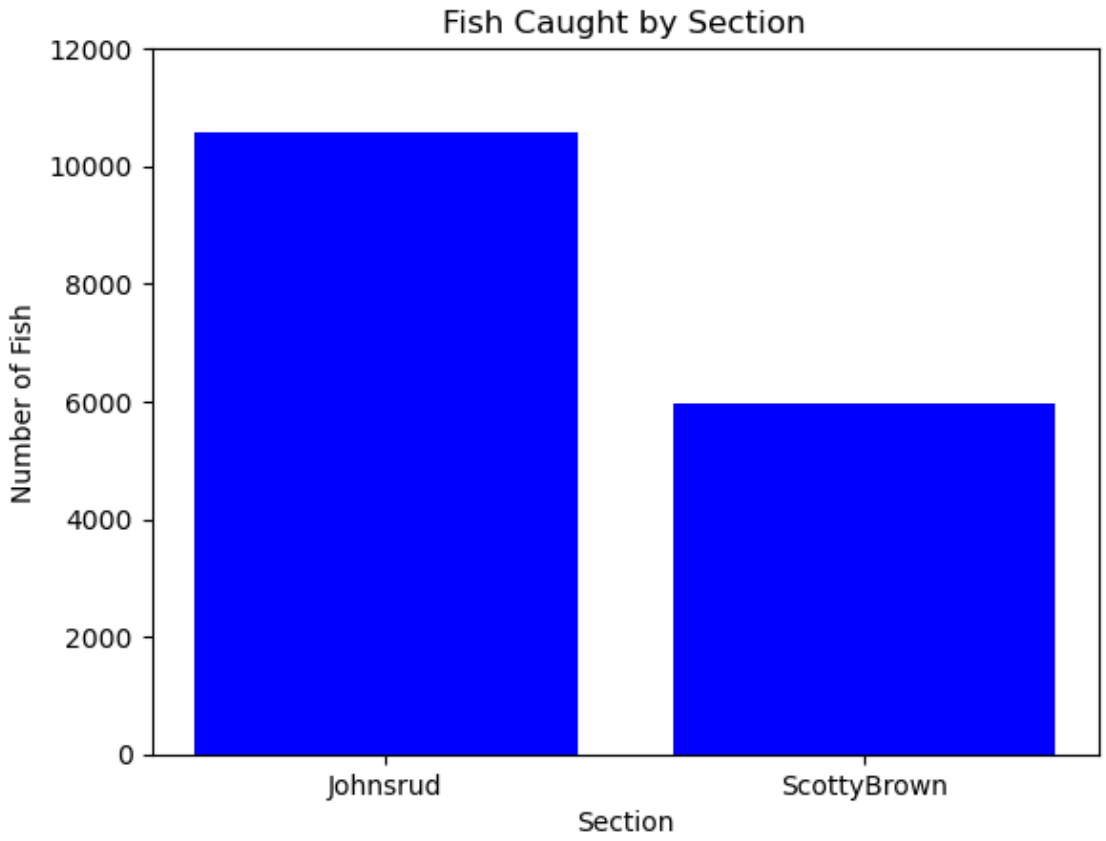
Bar charts are an effective way to compare the frequencies of levels of a categorical variable.

[66]:
```
section = BlackfootFish_clean['section'].value_counts()

fig, ax_bar = plt.subplots()

ax_bar.bar(section.index, section.values, color='blue')
ax_bar.set_xlabel('Section')
```

```
ax_bar.set_ylabel('Number of Fish')
ax_bar.set_title('Fish Caught by Section')
ax_bar.set_ylim([0, 12000])
plt.show()
```



13.5 Practice

Exercise 11: Using statistics or graphics, which year in our dataset had the most fish caught?

[67]:
```
# Code for Exercise 11 goes here
```

Exercise 12: Make a boxplot of the fish weights over the different years in the dataset.

[68]:
```
# Code for Exercise 12 goes here
```

14 Compiling

Hopefully at this point you have fixed all the code errors in the file we initially provided as you learned about the `Python` code and functions. A final (and often intermediate step) is to compile your iPython Notebook file into an HTML or PDF document. The compiling process will verify that the code you wrote works in the order that it is present in the document and is your reproducible

result to share with colleagues and collaborators. The .ipynb is your archive of the code used to produce those results from the data.

The HTML format is the least nice for printing but also has the fewest dependencies on other software to compile. Compiling to PDF yields a printable document.

You can quickly get an HTML view by **SAVING** first and then either pressing the "fast forward" button or by going to Kernel > Restart Kernel and Run All Cells…

To get a PDF document, go to File > Save and Export Notebook As… > PDF.

15 Exiting Jupyter

Finally, when you are done with your work and attempt to exit out of `Jupyter`, make sure to **SAVE** your notebook. Work is not automaically saved!

16 Terminology Used in Workshop

- *Command*: A command is what `Python` executes. In an `Python` script file (script.py), commands are automatically implied, as this type of file does not accept text, only in comments.
- *Comment*: Helpful text added into a script environment. Comments can be used to describe functions, processes, a train of thought, so that when you return to your code, tomorrow or next year, you are able to understand the purpose of each line of code! Comments are preceded by `#` within a code chunk.
- *Object*: A variable created in `Python`, to be used elsewhere in the code. Objects can be a variety of things, such as scalars (`x = 3`), vectors (`x = c(1, 2, 3, 4, 5)`), matrices, and dataframes, to name a few.
- *Assignment*: The assignment `=` is used to assign values on the right to the objects on the left (`x = 1`).
- *Type*: Most `Python` objects have a type attribute, a character vector giving the names of the types from which the object inherits. Examples of types are numeric, boolean, sequence types, DataFrame, tuple, list.
- *Vector*: A vector is a list of entries, all sharing the same class. A vector has only one dimension, so data extraction uses only a single entry in brackets (e.g. `x[3]`). You can create vectors of characters (`["a", "b", "c"]`), vectors of numbers (`[1, 2, 3]`), to name a few.
- *Matrix*: Similar to what you may have seen in a mathematics class, a matrix in `Python` is a two-dimensional array of numbers, symbols, or expressions arranged in rows and columns.
- *List*: A generic vector, which contains other objects. A list can contain a variety of different classes of objects, e.g., characters, vectors, data.frames, matrices, or outputs from a model! A dataframe is a special type of list where the components are vectors and they all have the same length.
- *Dataframe*: In Python, a dataframe is a two-dimensional table-like data structure that is commonly used for data analysis and manipulation, where you are able to extract elements using bracket (`[]`) notation.

- *Argument*: Input(s) into a function, so that an output is created. Most functions take named arguments (e.g., `data = BlackfootFish`) and the order of the arguments is assumed to follow the order found in the function's help file. When using a named argument in a function, the name comes first, followed by an `=` sign, then the input.
- *Logical Value*: `True` and `False` value(s) that can be used to turn off/on options in functions and plots, and also to manipulate data.

17 Workshop Materials & Recordings Available:

- email Sara Mannheimer (sara.mannheimer@montana.edu)
- through the MSU Library YouTube channel: https://www.youtube.com/watch?v=W6E3hpcoUkQ&feature=
- other related information: http://www.montana.edu/datascience/

18 How to Learn More About `Python`

This material is intended to provide you with an introduction to using `Python` for scientific analyses of data. The best way for you to continue to learn more about `Python` is to use it in your research! This may sound daunting, but writing `Python` scripts is the best way to become familiar with the syntax. This will help you progress through more advanced operations, such as cleaning your data, using statistical methods, or creating graphics.

The best place to start is playing around with the code from today's workshop. Change parts of the code and see what happens! Better yet, use the code from the workshop to investigate your own data!

19 Montana State University Data Science Workshops Team

These materials were adapted from materials generated by the Data Carpentries (https://datacarpentry.org/) and were originally developed at MSU by Dr. Allison Theobold. The workshop series is co-organized by the Montana State University Library, Department of Mathematical Sciences, and Statistical Consulting and Research Services (SCRS, https://www.montana.edu/statisticalconsulting/). SCRS is supported by Montana INBRE (National Institutes of Health, Institute of General Medical Sciences Grant Number P20GM103474). The workshops for 2021-2022 are supported by Faculty Excellence Grants from MSU's Center for Faculty Excellence.

Research related to the development of these workshops appeared in:

- Allison S. Theobold, Stacey A. Hancock & Sara Mannheimer (2021) Designing Data Science Workshops for Data-Intensive Environmental Science Research, *Journal of Statistics and Data Science Education*, 29:sup1, S83-S94, DOI: 10.1080/10691898.2020.1854636

The workshops for 2023-2024 involve modifications of materials and are being taught by:

19.1 Greta Linse

- Greta Linse is the Interim Director of Statistical Consulting and Research Services (https://www.montana.edu/statisticalconsulting/) and the Project Manager for the Human Ecology Learning and Problem Solving (HELPS) Lab (https://helpslab.montana.edu). Greta has been teaching, documenting and working with statistical software including Python, R and RStudio for over 15 years.

19.2 Sara Mannheimer

- Sara Mannheimer is an Associate Professor and Data Librarian at Montana State University, where she helps shape practices and theories for curation, publication, and preservation of data. Her research examines the social, ethical, and technical issues of a data-driven world. She is the project lead for the MSU Dataset Search, and she is working on a book about data curation to support responsible qualitative data reuse and big social research.

19.3 Devin Goodwin

- Devin Goodwin graduated with a Masters in Mathematics at Montana State University and while a GRA with Statistical Consulting and Research Services (SRCS) developed these materials thanks in part to a CLS grant. Devin has always focused in pure theoretical mathematics up until 2021, when he decided to dedicate his efforts to applied mathematics. Along with the already existing `R` Workshops developed by the team members mentioned above, Devin adapted this series to `Python`.

The materials have also been modified and improved by:

- Dr. Mark Greenwood
- Harley Clifton
- Eliot Liucci
- Dr. Allison Theobold