Programming Review

Foundations of Machine Learning

```
! git clone https://www.github.com/DS3001/programming
```

Introduction/Review of Programming

- This lecture reviews some Python programming essentials. Not exciting, but necessary for many students.
- There are many programming paradigms, but in data science, we typically use the *procedural programming* approach: We write a *script* file, containing a sequence of instructions, which we pass to an interpreter, which carries them out in real time as we iterate.
  - Imagine writing a "script" for a play: You name the characters (objects), you give them lines and tell them what to do (expressions, functions), and the director makes it all happen (control structures, interpreter)
- By contrast, for example, in *object oriented programming*, you design an abstraction of an object that a user will interact with through a variety of methods: For example, one object in Excel is a spreadsheet, and the programming revolves around how the user interacts with spreadsheets.

Git and GitHub

- Remember, Git is version control software that allows teams of people to work on a project all at the same time, track their changes, and merge their work back together. GitHub is a web-based service that hosts directories that use Git, called **repositories** or "repos."
- Over the course, we'll gradually introduce more features of Git and GitHub, and by the end, it will just be the ecosystem we're all working in together.
- For now, you'll just import this lecture by going to Google Colab and `File -> Open Notebook -> GitHub -> programming.ipynb`
- Remember, files and work are not persistent on Colab, so make sure to download any files you edit and want to keep before you exit Colab (you can download and upload a notebook for homework, for example)

Getting Started: Objects

- We assign *values* to *objects* in Python using the = operator. For example, the value 5 is assigned to the object x by typing:

```
x = 5
```

- To have Python exhibit the value of an object in output, you use the `print()` function:

```
print(x)
```

```
5
```

- You can manipulate the value of x numerically the way you'd expect (mostly):

```
print(2*x) # Multiplication
print(x**2) # Exponentiation
```

```
10
25
```

Objects: Naming Conventions

- Try to give objects names that suggest clearly what they are: x and *temp* can get reused by accident and cause problems when you overwrite their values later on, or you might forget what they actually are when reading the code later

- *Camel case*: looks like `firstRegression`
- *Snake case*: looks like `first_regression`
- Avoid putting the _ at the beginning of a variable name in Python: Sometimes, it can interpret this symbol in ways that influence how the code runs

Commenting Code

- The most important thing to do when writing code, is *comment* it
- In Python, when you write a sharp-hash-pound, #, it tells the interpreter to stop reading and executing, and skip the rest of that line
- In a month, when you return to code from the beginning of class for a project, you will wish you commented your code so you can remember what various lines do
- You don't have to comment every line, but important or hard-to-read parts should be documented for other people or your future self
- You can also "comment out" code you don't want to be run without deleting it

```
x = 5  # 5 is assigned to x
# x = 7  # This line is "commented out"
print(x) # x is still 5
```

```
5
```

Objects: Types

- All objects in Python have *types*: What kind of a thing is this?
- *Integer* (`int`): Taking whole number values, like . . . , -2, -1, 0, 1, 2, . . .
- *Floating-Point* (`float`): Essentially, real numbers. You can signal to Python that something is a float by initializing its value as 0.0 rather than 0.
- *Boolean* (`bool`): Takes the values `True` [1] or `False` [0].
- *Strings* (`str`): Strings of "text" (including letters, numbers, symbols)

```
y = "Machine Learning @ 8:00 a.m."
```

You can use double " " or single ' ' quotes, but they have to match. The symbol `\` has to be handled with care, since this is the *escape character* that Pyton interprets as a formatting command, like a newline `\n` or a backslash `\\`. This matters a lot when dealing with text data. - *Complex Numbers* (`complex`): Values involving $\sqrt{-1}$. These are used in data science mainly in time series analysis, and won't come up in this class.

```
string = 'This is a string. \nThis is a new line. \nThis is a single backslash: \\.'
print(string)
```

```
This is a string.
This is a new line.
This is a single backslash: \.
```

Boolean Arithmetic

- When you pass a Boolean into a calculation, it *coerces* `True` to 1 and `False` to zero, then evaluates the resulting expression
- We will end up using this kind of type coercion all the time, since we will often create Boolean variables to capture complicated qualitative states (e.g. "Is this defendant facing felony charges and do they have a prior conviction?")

```
# Some Boolean Arithmetic
x = True
y = False
z = True
```

```
w = False
print(x*y) # x and y?
print(x+y-x*y) # x or y?
print(x+z-x*z) # x or z?
print(y+w-y*w) # y or w?
print(2*x)
```

```
0
1
1
0
2
```

Objects: Ordered Collection Types

- We often collect objects together into a collection, but the best way to represent that collection depends on the natural relationships the objects have with one another
- A *list* is an ordered collection of objects which can be changed after it is defined. Python lists are initialized using brackets, `x=[...]`.
- A *tuple* is an ordered collection of objects, but cannot be edited or changed after it is initialized. Python tuples are initialized using parentheses, `x=(...)`.
- Every string object is an ordered collection of letters/symbols, like a list.

```
x = (3,5,7)
print(x)
# x[2] = 8 # Throws an error
# print(x)
x = [3,5,7]
print(x)
x[2] = 8
print(x)
```

```
(3, 5, 7)
[3, 5, 7]
[3, 5, 8]
```

Objects: Unordered Collection Types

- A *set* is an unordered object, like the , but supports useful set operators. Python sets are initialized using braces, `x={...}`. The objects in a set can be of any kind (tuple, list, other sets, objects of any class type).
- A *dictionary* is an unordered collection of key-value pairs, like, `x = { key1: value1, key2: value2, ... }`
  - `x = { 'Name': 'University of Virginia', 'City': 'Charlottesville', 'State': 'VA' }`
- Like a set, a dictionary can hold multiple objects of different types
- Sets and dictionaries can be particularly useful if you want to return multiple outputs for a function: Put everything you want to return into the set or dictionary, and then return that object.

```
x = { 'University of Virginia', 'Virginia Tech', 'George Mason University', 'College of William and Mar
print(x) # Notice the order
print(type(x),'\n')
x = { 'Name': 'University of Virginia', 'City': 'Charlottesville', 'State': 'VA' }
print(x)
print(x['City'])
print(type(x))
```

```
{'College of William and Mary', 'University of Virginia', 'George Mason University', 'Virginia Tech'}
<class 'set'>

{'Name': 'University of Virginia', 'City': 'Charlottesville', 'State': 'VA'}
Charlottesville
<class 'dict'>
```

Objects: Collection Types

- The built-in Python collection types are useful, but will probably not behave exactly as you expect
- We'll quickly add in NumPy, a package that does numerical vector and matrix calculations the way you would expect
- Our data wrangling package, Pandas, builds its approach to modeling data on NumPy

```
x = (3,2,7)
print(x+x) # This is concatenation, not vector addition
print(2*x) # This is concatenation, not scalar multiplication
```

```
(3, 2, 7, 3, 2, 7)
(3, 2, 7, 3, 2, 7)
```

Indexing: Order and Counting

- Every `string` in Python is an ordered structure, like tuples and lists
- Here, *ordered* is easy to understand: The sequence in which letters appear in a word is important, and that relationship needs to be preserved for the word to make sense
- Please listen: *Python indexes the order of elements from zero, rather than one*
- Imagine you have the string "Apple"
- What is the first letter of the word "Apple"? It is "A"
- At what *index* is the letter "A" stored by Python in the string "Apple"? It is stored at 0
- Notice the "boundary conditions": Negative indices "wrap" around the word.

```
x = 'Apple'
print(x[1])
print(x[0])
print(x[-1])
print(x[-2])
print(x[-3])
print(x[-5])
# print(x[5]) # This throws an out-of-range error
# print(x[-6]) # This throws an out-of-range error
```

```
p
A
e
l
p
A
```

Functions

- When a piece of code is so useful that you want to re-use it, you create a *function*
- Functions take any kind of input – numeric, string, boolean, data set, other functions etc. – and return some kind of output

- The syntax for this is `functionName(arguments)`
- For example, the function `len(x)` gives the length of `x`: The number of items in the collection
- Many functions and operators are defined for many types of objects

```
x = 'Apple'
print(len(x))
```

```
5
```

Some Useful Functions

- `type(x)` returns the kind of object that `x` is, like a float or Boolean
- `len(x)` returns the number of elements of `x`
- `sum(x)` returns the sum of the elements of `x`
- `abs(x)` returns the absolute value of `x` (but only if `x` is a number and not a collection)
- `round(x,n)` rounds `x` to the `n`-th decimal place
- `max(x)` and `min(x)` give the largest and smallest values in the collection `x`
- But, there is no `mean(x)` function in base Python. What do we do?

Defining a Function

- Sometimes, the functions we want are not available, or we've written some code that is specific to our work but so valuable we want to reuse it
- Python is very particular about *whitespace*. It has a set of conventions that follow what previous programmers were naturally doing to format their code, which is indenting it to set off blocks of calculations to make it easier to read. This eliminates the ubiquitous `;` and `{}` in C.
- Python is very loose about object typing, so you don't need to be very careful about defining functions and objects: It just assumes things will work.
- If you want to return lots of results, you have to think about how to build the object that you pass back out of the function (e.g. dictionaries or class objects you created)

```
def mean(x):   # Declare the function, mean, and its inputs, x
    m = sum(x)/len(x)  # Perform calculations/evaluate expressions
    return m   # Return an object with your results

x = (2,3,-1,7,11) # Example tuple of values
print(mean(x))
```

```
4.4
```

Functions and Objects

- Objects typically have their own functions that they carry around with them
- These special functions are built when the object is created: This allows us to build abstractions called *classes*
- For example, suppose you want to convert a string `x` to all lowercase letters. You don't use `lower(x)`, you use `x.lower()`
- In general, `x.y` tells Python to look inside `x` for `y`. If you see `x.y()`, then `y()` is a function inside `x` called a **class method**, and if you see `x.y`, it means `.y` is an **class instance attribute** of `x`
- This will come up all the time, since dataframes in Python are objects with many built in functions that operate directly on the data frame, and the notation becomes easier and more natural to follow with experience.

```
x = 'Apple'   # x is an object of class string
print(x.format) # Class instance attribute
print(x.lower())   # Class method called `lower`
```

```
# lower(x)   # This produces an error and doesn't work
'Apple'.upper() # You can call class methods directly on an object as it is constructed
```

```
<built-in method format of str object at 0x000001D6B0922FF0>
apple
```

```
'APPLE'
```

Type Coercion/Casting

- When cleaning data, you often have data that is numeric, but stored as, say, a string, like ' "$10.00" '
- This can be a nuisance: You can easily lose data when an entry like "$12,000" is assumed by the computer to be text and dropped from the dataset as missing when a calculation is attempted
- In order to force an object into a different type, we can use *type coercion/casting*, where we tell Python to try changing from, say, a string to a float
- To coercion an object `x` into a float, you type `y = float(x)`, and similarly for `bool(x)`, `int(x)`, and `str(x)`

```
x = '5' # x starts as a string
print(type(x)) # What type is x?
y = float(x) # Convert x to a float called y
print(type(y)) # What type is y?
print(y**2) # The square of y
print(bool("1")) # Convert a string of the number 1 to Boolean
```

```
<class 'str'>
<class 'float'>
25.0
True
```

Ordered Collections: Indexing

- When you have an ordered object, like a string or tuple or list, you can easily make selections from it by indexing
- Here we run into a bit of an issue: Python starts *indexing* elements in ordered collections from 0, not 1
- *Indexing is not counting* (in Python, C, etc., versus languages that index from 1, like R, FORTRAN, MATLAB, etc.)
- This means that the "first" element in a tuple corresponds to index number 0, the "second" element corresponds to index number 1, and so on.
- You can think of selecting the n-th element from `x` by typing `x[n-1]`, but at this point you have to make the psychological shift to thinking of `x[0]` as the first element and thinking directly of `x[n]` as the $n-1$-st element of `x`

```
x = (3,2,7,-1,10)
print(x[0]) # First element
print(x[1]) # Second element
print(x[-1]) # Notice the boundary conditions: No error
# print(x[(1,3,5)]) # This does not work (fancy indexing)
```

```
3
2
10
```

Ordered Collections: Slicing

- Sometimes you want to select a set of indices all at once, and preserve the order
- This is called *slicing*, where you provide a tuple of values
- To slice, you use the notation `x[a:b]`. This will return the elements `x[a]` to `x[b-1]` as an ordered collection (remember, indexing is not counting)
- If you use `x[:b]` it implicitly sets `a=0`, and if you use `x[a:]` it takes the values from `a` to the end of the ordered object

```
x = (3,2,7,-1,10)
print(x[0:3]) # First, second, third elements
print(x[1:4]) # Second, third, fourth elements
print(x[2:]) # Third item to the end
print(x[:2]) # First item to the third
print(x[2:100]) # Notice the range
```

```
(3, 2, 7)
(2, 7, -1)
(7, -1, 10)
(3, 2)
(7, -1, 10)
```

Slicing From Zero

- Since Python *counts from zero*, when you request `x[1]`, it gives you the *second* item in `x`. The *first* item in `x` is `x[0]`. What happens when you slice?
- If you are indexing from zero and request `2:4`, you get the *third* and *fourth* items in the list, and two items rather than three (you might expected items 2, 3, and 4, but that is not what you get!)
- So everything is shifted to the left by 1 so that the zero-th element corresponds to the first entry in a tuple or list
- I think in terms of "$<$ starting from 0" rather than "$\leq$ starting from 1", but if that's not helpful forget I said anything
- If you are coming from R or MATLAB, this will probably be an adjustment

Object Equivalence/Comparison

- To test if an object x is equivalent to another object y, you can use the `==` operator, which returns `True` or `False`.
- Be careful not to confuse the equivalence operator `==` with the assignment operator `=`.
- Along with `==`, you can test for greater-than `>`, less-than `<`, greater-than-or-equal-to `>=`, less-than-or-equal-to `<=`, or not-equal-to `!=`
- This is done for the object as a whole, not element-by-element, but if you want to check if a value `z` is in a collection `x`, you can use the `in` operator, like `5 in (2,-1,5,8)`

```
x = (2,3,5)
y = (2,3,5)
z = (-2,4,11)
print(x == y) # x and y are the same, pointwise
print(y == z) # x and z are not equal
print(x == 3) # 3 is an element of x, but x is not equal to the float 3
print(5 == '5') # The integer 5 versus the string '5'
x = ['Abby','Amir','Alejandro']
'Amir' in x
'Kyle' not in x
```

```
True
```

```
False
False
False
```

```
True
```

Modules and Packages

- Python is a general purpose computing language, so it doesn't come with a lot of pre-loaded functionality
- A *module* is a .py file that includes a set of user-written functions. A *package* is a directory of modules with an `__init__.py` file that tells the Python interpreter how to load the package.
- To import a package `package`, you type `import package as workingName`. You typically want the working name to be very short but clearly point to the package, to keep code clean but readable.

Modules and Packages

- There are five packages that we will spend time with:
  - Pandas: Data frames and statistics, `import pandas as pd`
  - MatPlotLib: A package that creates plots, `import matplotlib.pyplot as plt`
  - SeaBorn: Slightly nicer looking plots, `import seaborn as sns`
  - Scikit-Learn: A popular machine learning package, `import sklearn as sk`
  - NumPy: Numerical analysis for Python, `import numpy as np`
- By installing and loading packages, you can quickly build a Python environment appropriate for data science

```python
import numpy as np # Load the NumPy package into our workspace
np.random.seed(100) # Set the seed for the random number generator
sample = np.random.randn(10,2) # Create a random sample of normal draws
print(sample)
import matplotlib.pyplot as plt # Load the matplotlib plotter as plt
plt.scatter(sample[:,0],sample[:,1]) # Create a scatterplot to plot the sample
```

```
[[-1.74976547  0.3426804 ]
 [ 1.1530358  -0.25243604]
 [ 0.98132079  0.51421884]
 [ 0.22117967 -1.07004333]
 [-0.18949583  0.25500144]
 [-0.45802699  0.43516349]
 [-0.58359505  0.81684707]
 [ 0.67272081 -0.10441114]
 [-0.53128038  1.02973269]
 [-0.43813562 -1.11831825]]
```

```
<matplotlib.collections.PathCollection at 0x1d6c0c090f0>
```

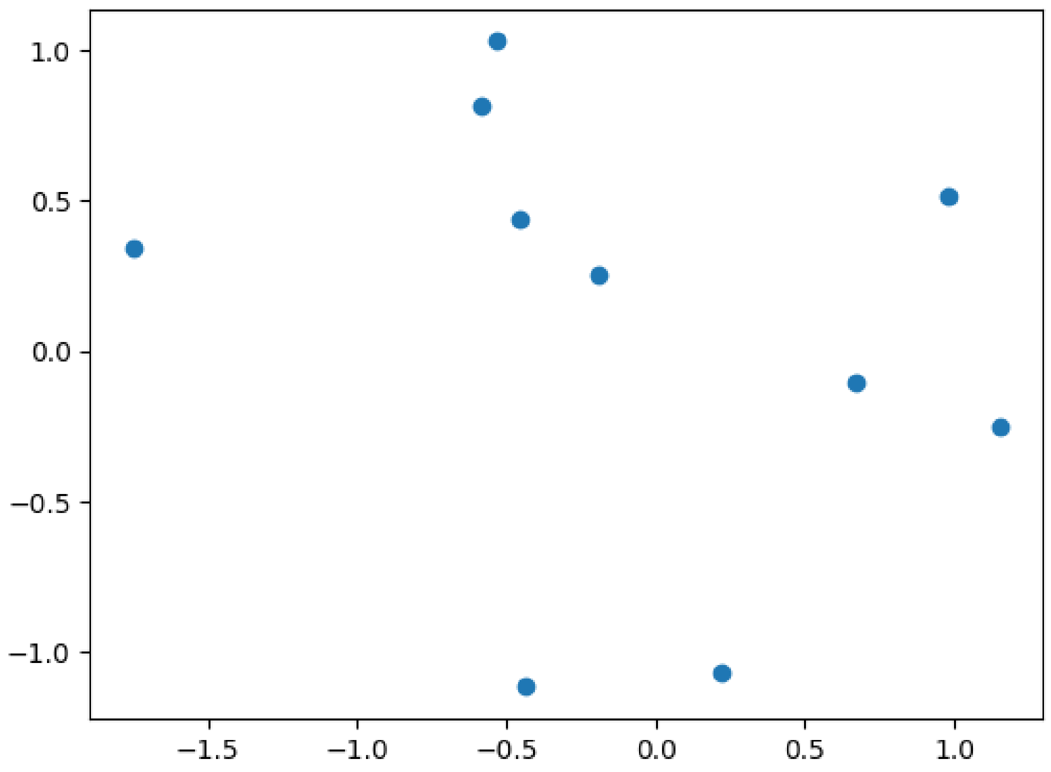

Figure 1: png

NumPy

- Recall that `2*[1,3] = [1,3,1,3]`, not `[2,6]` (concatenation not scalar multiplication)
- You *could* create your own function that implements scalar multiplication, but there is a high quality scientific computing and numerical analysis package called **NumPy** that already does these things
- We will use Numpy a lot but rarely talk about it, since it is typically "under the hood"
- Numpy implements efficient linear algebra and numerical methods in Python, so that you can do scalar and vector multiplication, multiply and invert matrices, and so on
- Numpy has an array class that supports efficient multiplication of vectors and matrices: Convert a Python list `x` to a Numpy array with `np.array(x)`

```
import numpy as np # Import the Numpy package into your workspace
x = [3,5,7] # A standard Python list
print(2*x) # Concatenation of lists
y = np.array(x) # Convert Python list x to Numpy array y
print(2*y) # Scalar multiplication, like we want
z = np.array([-1,-2,-3]) # Initialize a numpy array in one step
print(y+z) # Vector addition, like we want
print(np.inner(y,z)) # Inner product of vectors
print(y/10.0) # Elements of y, divided by 10
```

```
[3, 5, 7, 3, 5, 7]
[ 6 10 14]
[2 3 4]
-34
[0.3 0.5 0.7]
```

NumPy Functions

- Numpy arrays have useful class methods:
  - `x.min()`, `x.max()`, `x.mean()`, `x.median()`, `x.sum()`, `x.sort()`, `x.round()` all do what you would expect
- The Numpy package itself also has some useful functions that are not class methods:
  - `np.where(x == value)` : Determines the indices of x that equal `value`
  - `np.linspace(a,b,N)` makes a grid from `a` to `b` of `N` equally spaced points
  - `np.arange(a,b,step)` makes a grid from `a` to `b-step` of points spaced by `step`

```
import numpy as np
x = np.linspace(2,5,10) # Grid from 2 to 5 with 10 steps
print(x,'\n')
print( np.where( x == 4.0 ), '\n') # Which x equals 4.0?
y = np.arange(2.0,5.0,.1) # Grid from 2 to 5 by .1
print(y)
```

```
[2.         2.33333333 2.66666667 3.         3.33333333 3.66666667
 4.         4.33333333 4.66666667 5.        ]

(array([6], dtype=int64),)

[2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5 3.6 3.7
 3.8 3.9 4.  4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9]
```

NumPy `nan` handling methods

- Since missing values are modeled as NumPy's `nan` value, there are rules for interacting with them:
  - `nan` is a value, initialized as `x=np.nan`
  - Any arithmetic involving a `nan` evaluates to `nan`
  - If you want to check if a variable or value is `nan`, you do not use `==`, but instead use the NumPy method `np.isnan(x)` to check if `x` is a `nan` — this is super important to understand for cleaning data

```
import numpy as np
x = np.nan
y = 1
print(1+x) # nan
print(x*y) # nan
print(x/y) # nan
print(x == np.nan) # False, **probably don't do this**
print(x == y) # False, **probably don't do this**
print(np.isnan(x)) # True
```

```
nan
nan
nan
False
False
True
```

MatPlotLib

- In a lot of situations, we just want to plot some x values against y values: this is called a **scatterplot**
- To make a quick scatterplot, `import matplotlib.pyplot as plt` and use the `plt.scatter(x,y)` method

- To plot functions, `x=np.linspace(min,max,N_steps)` creates a grid from `min` to `max` with a total of `N_steps`
- You can then typically pass the grid `x` to the function `y=f(x)`, and then plot (x, y)

```
import numpy as np # Import the Numpy package into your workspace
x = np.linspace(-5,5,100) # create a grid from -5 to 5 of 100 points
y = x+6*np.sin(x) # Compute the value of the function y = x+6*sin(x)
import matplotlib.pyplot as plt # Import pyplot from matplotlib as plt
plt.scatter(x,y) # Create scatterplot
```

```
<matplotlib.collections.PathCollection at 0x1d6c2e0d5a0>
```

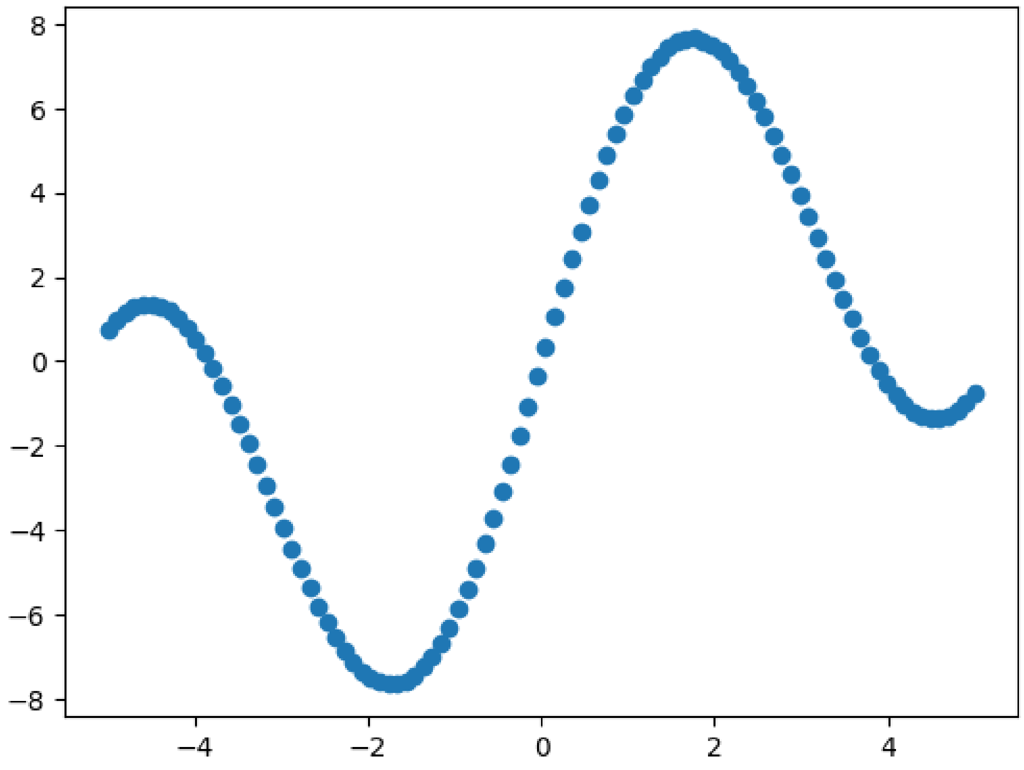

Figure 2: png

Python Review

- If you've done any programming, Python is a pretty easy language to use
  - Dynamic types, interpreted, packages are easy to import, simple control controls with whitespace
- For data science, what matters about a language is mostly the packages it supports: All of the efficient code is written in a compiled language like C++, anyway
- Even if you don't like Python, it's nice to know and the general skills we'll learn easily transfer to R or MATLAB or any other programming language