Introduction to Python

Module II: Advanced - External libraries and virtual environments

Objectives

- Identify the difference between built-in functions, built-in modules and external libraries.
- Import and use common Python external libraries.
- Use the external library NumPy to create arrays and perform mathematical and statistical operations.
- An introduction to another external library in Python: SciPy.
- Set up virtual environments using Conda.

1. Introduction

So far, we have covered the basics of Python. We discussed functions in the first module of Python, and showed how you can simplify complex processes that are repeated throughout your code. One of the main advantages of Python is the extensive amount of open source, and robust functions. These functions can be used throughout your code and greatly simplify the process of data analysis.

A set of functions in a .py file is called a module, and a set of modules is called a library. These pre-made libraries are known as “external libraries”, and we will go over how to use these in your code in the next sections. Here we want to briefly review a useful internal module in Python which can simplify mathematical operations. This module is called Math. The math module has a set of methods and constants for various arithmetic and mathematical functions. To use it, you must import the math module:

Code:

```
# importing Math module
import math
```

When you have imported the math module, you can start using methods and constants of the module. The *math.sqrt()* method for example, returns the square root of a number, and the *math.pi* constant, returns the value of PI (3.14...):

Code:

```
x = math.sqrt(36)
y = math.pi
print(x)
print(y)
```

Output:

```
6.0
3.141592653589793
```

To see the complete list of Math functions, visit the official documentation of the module.

2. External Libraries

To use the extensive external libraries in Python, you must first make sure that they are installed and that they are a part of your active environment. NumPy should be included in your base installation of Anaconda, environments are elaborated more on in Section 7.

Once the library is installed in your active environment, it must be imported to your code. There are various ways to do this. One of the most common external libraries is “NumPy”, meaning “Numerical Python”. This is how you would import the Numpy library under the alias np:

Code:

```
# importing a common Python library
import numpy as np
```

Code:

```
# importing a common Python library

import matplotlib.pyplot as plt
from matplotlib import pyplot as plt
```

This will import the pyplot module from the matplotlib library, under the alias plt.

If you remember from the previous module, we went in depth with functions and discussed the syntax for writing functions. As a quick reminder, the syntax is as follows.

Code:

```
def add_function(a, b):
      result = a + b
      return result

z = add_function(20, 22)
print(z)
```

It can be useful to have multiple associated functions all stored together in a file for easy access later on. A set of functions in a .py file is called a module. Using your knowledge of functions you can easily create a simple module and call it from a script. Here is an example of a module with the filename "example_module.py":

Code:

```
# example_module.py
```

```
def add(a, b):
    # add two numbers
    return(a + b)

def subtract(a, b):
    # subtract two numbers
    return(a - b)
```

This module can then be called from the same directory by using the command "import". You call the function from within the module using the syntax "module.function(inputs)".

Code:

```
import example_module

print(example_module.add(5,3))
print(example_module.subtract(4,2))
```

Exercise 2.1

- Create a new directory with the files math_module.py and math_script.py
- Copy the add and subtract functions from example_module.py and also write functions for multiplication and division
- In script.py import the the math_module
- Choose two numbers and run these through each of the four functions and print the result

It can also be useful to associate many modules together, if there are a large number of functions you would like to access. A library can be thought of as a set of modules. In practice it is a bit more involved to set up a library. The strength of Python is the number of already made and supported libraries, called "external libraries" like we discussed earlier. The syntax of calling a function from a library is "library.module.function(inputs)".

For example, the library “numpy” contains the module “random”. The module “random” then contains functions, such as “normal” to sample from a normal distribution, or “randint” to sample a random integer. Calling these functions looks as follows:

Code:

```
# calling functions from a common Python library
normal_sample = np.random.normal(loc=0.0, scale=1.0, size=10)
integer_sample = np.random.randint(low = 10, high=None, size=5)
print (normal_sample)
print (integer_sample)
```

The first line will return a sample of 10 numbers from a normal distribution with mean zero and variance 1. The second line will return a sample of 5 integers between 0 and 10.

3. NumPy library

In Python, we have lists that are a collection of various data types, but they are slow to process. NumPy (stands for Numerical Python) is a Python library used for working with arrays. An array is a data structure consisting of a collection of elements. NumPy arrays are up to 50x faster than traditional Python lists. NumPy arrays are designed to be homogenous, meaning all the elements in the array should ideally be of the same data type. This allows for efficient storage and operations on the array (technically possible to mix data types but it is generally not recommended for memory efficiency). NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them efficiently. This behavior is called locality of reference in computer science. NumPy is written partially in Python, but most of the parts that require fast computation are written in C or C++ which is faster than Python in nature. These are the main reasons why NumPy arrays are much faster than lists. NumPy is also optimized to work with the latest CPU architectures to efficiently use computer resources (less memory usage). Arrays are frequently used in data science, where computational speed and effective use of resources are very important. NumPy also has functions for working in the domain of linear

algebra, Fourier transform, and matrices. NumPy is automatically included in the Anaconda distribution of Python but if you use any other version of Python, you can install NumPy using: ***conda install numpy*** in your anaconda command line.

To use NumPy, we first need to import the library. NumPy is usually imported under the *np* alias.

Code:

```
import numpy as np
```

The array object type in NumPy is called ndarray. We can create a NumPy array by using the array() function. To create an ndarray, we can pass a list or tuple or any array-like object into the array() method, and it will be converted into a NumPy array:

Code:

```
#using a list to create a NumPy array
import numpy as np
list_array = np.array([1, 2, 3, 4, 5])
print(list_array)
print(type(list_array))
```

Output:

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

Code:

```
#using a tuple to create a NumPy array
tuple_array = np.array((1, 2, 3, 4, 5))
print(tuple_array)
print(type(tuple_array))
```

Output:

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

NumPy arrays can have multiple dimensions. Every array has a dimension. 0-D arrays, or scalars, are the elements in an array.

Code:

```
#0-D array
array_0d = np.array(36)
print(array_0d)
```

Output:

```
36
```

An array that has 0-D arrays or scalars as its elements is called a uni-dimensional or 1-D array. These are the most common and basic arrays.

Code:

```
#This is a 1-D array containing the values 1,2,3,4,5.
array_1d = np.array([1, 2, 3, 4, 5])
print(array_1d)
```

Output:

```
[1 2 3 4 5]
```

An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix or 2nd order tensors.

Code:

```
# Create a 2-D array containing two arrays with the
values 1,2,3 and 4,5,6:

array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(array_2d)
```

Output:

```
[[1 2 3]
 [4 5 6]]
```

An array that has 2-D arrays (matrices) as its elements is called 3-D array. These are often used to represent a 3rd order tensor, but we will skip 3-D arrays in this course.

NumPy Arrays provides the ***ndim*** attribute that returns an integer that tells us how many dimensions the array has.

Code:

```
# using ndim to determine dimensions of an array

array_0d = np.array(36)
array_1d = np.array([1, 2, 3, 4, 5])
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
array_3d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3],
[4, 5, 6]]])

print("number of dimensions for array_0d is",
array_0d.ndim)
print("number of dimensions for array_1d is",
array_1d.ndim)
print("number of dimensions for array_2d is",
array_2d.ndim)
print("number of dimensions for array_3d is",
```

```
array_3d.ndim)
```

Output:

```
number of dimensions for array_0d is 0
number of dimensions for array_1d is 1
number of dimensions for array_2d is 2
number of dimensions for array_3d is 3
```

np.arange() in NumPy is a function used to generate evenly spaced values within a given interval. It's a versatile tool for creating NumPy arrays with numeric sequences.

np.arange(start, stop, step, dtype=None)

- start (optional): This defines the first value in the sequence (inclusive). If not specified, it defaults to 0.
- stop (required): This defines the end of the interval (exclusive). The stop value won't be included in the generated array.
- step (optional): This defines the spacing between consecutive values. Defaults to 1, which means elements will be incremented by 1.
- dtype (optional): This specifies the data type of the elements in the array. By default, it uses the most appropriate type based on the input arguments.

Code:

```
# Creating an array from 0 to 9 (excluding 10):

arr = np.arange(10)
print(arr)
```

Output:

```
[0 1 2 3 4 5 6 7 8 9]
```

np.arange() is generally faster than using a Python for loop to create a list and then converting it to a NumPy array. Numpy function *np.linspace()* is another similar function to create an array of evenly spaced numbers over a specified range with the following format (main arguments):

np.linspace(start, stop, num, endpoint=True/false)

- start: The starting value of the sequence.
- stop: The end value of the sequence.
- num: The number of samples to generate. Default is 50.
- endpoint: If True, stop is the last sample. Otherwise, it is not included. Default is True.

Code:

```
# Generate 20 evenly spaced samples between 0 and 40, not
including 40
samples = np.linspace(0, 40, num=20, endpoint=False)
print(samples)
```

Output:

```
[ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18. 20. 22. 24. 26.
28. 30. 32. 34. 36. 38.]
```

The shape of an array is the number of elements in each dimension. NumPy arrays have an attribute called shape that returns a tuple where each element represents the length of the corresponding dimension. It essentially describes the array's size and structure.

Code:

```
# using shape to find the number of elements in each
dimension
```

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
```

Output:

```
(2, 4)
```

The example above returns (2, 4), indicating 2 rows and 4 columns.Alternatively, we can use the np.shape() Numpy function to find the shape of an array as well. Reshaping means changing the shape of an array. The size of the array can be calculated by multiplying the elements of the shape tuple. In the above example, the total size is 2 * 4 = 8 elements. By reshaping we can add or remove dimensions or change the number of elements in each dimension.

Reshaping a NumPy array involves changing its dimension structure without altering the underlying data. It's a powerful way to manipulate the layout of the data to suit your needs. We can reshape an array into any shape as long as the elements required for reshaping are equal in both shapes. For example, we can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

Code:

```
# using reshape to change dimension structure of an array

# Create a 1D array
data = np.arange(12)
# Array containing 12 elements (0 to 11)

# Reshape into a 2D array with 3 rows and 4 columns
reshaped_data = data.reshape(3, 4)

# Print the original and reshaped arrays
print("Original 1D array:\n", data)
print("\nReshaped 2D array:\n", reshaped_data)
```

Output:

```
Original 1D array:
 [ 0  1  2  3  4  5  6  7  8  9 10 11]

Reshaped 2D array:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Exercise 3.1

Convert the following 1-D array with 12 elements into a 2-D array. The outermost dimension will have 4 arrays, each with 3 elements:

Array_example= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

4. NumPy Array Joining, Indexing and Slicing

Array indexing is the same as accessing an array element. You can access an array element by referring to its index number. The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Code:

```
#Array indexing in 1-D arrays
array_numbers = np.array([1, 2, 3, 4])
print(array_numbers[0])
```

Output:

```
1
```

We can also replace a single value in our array:

Code:

```
#Array value assignment
array_numbers = np.array([1, 2, 3, 4])
array_numbers[0]=6
print(array_numbers)
```

Output:

```
[6 2 3 4]
```

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element. Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

Code:

```
#Array indexing in 2-D arrays
array_numbers2 = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(array_numbers2)
#Access the element on the first row, second column:
print('2nd element on 1st row: ', array_numbers2[0, 1])
#Access the element on the 2nd row, 5th column:
print('5th element on 2nd row: ', array_numbers2[1, 4])
```

Output:

```
2nd element on 1st row:  2
5th element on 2nd row:  10
```

Exercise 4.1

Create a 2-D 5x5 array using np.arange() and then print the element on the third row and second column.

We can also slice NumPy arrays. We pass a slice instead of an index like this: [start:end].

Code:

```
#Slice elements from index 1 to index 5 from the following
array
array_example = np.array([1, 2, 3, 4, 5, 6, 7])
print(array_example[1:5])
```

Output:

```
[2 3 4 5]
```

Code:

```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
#Slice a 2-D array, from the second element, slice
elements from index 1 to index 4
print(arr[1, 1:4])
#From both elements, slice index 1 to index 4 (not
included), this will return a 2-D array:
print(arr[0:2, 1:4])
```

Output:

```
[7 8 9]

[[2 3 4]
 [7 8 9]]
```

Joining arrays means putting contents of two or more arrays in a single array. We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If the axis is not explicitly passed, it is taken as 0.

Code:

```
# Join two 1-D arrays
```

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr_combined = np.concatenate((arr1, arr2))
print(arr_combined)
```

Output:

```
[1 2 3 4 5 6]
```

Finally, arithmetic operators on NumPy arrays apply elementwise. A new array is created and filled with the result.

Code:

```
# Create two NumPy arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Add the arrays element-wise using the + operator
sum_arr = arr1 + arr2
print("Element-wise sum:\n", sum_arr)

# Multiplication is done the same way as addition
product_arr = arr1 * arr2
print("Element-wise product:\n", product_arr)

#If the arrays have different shapes, NumPy might perform
broadcasting to make them compatible for element-wise
operations. Broadcasting extends the smaller array in a
specific way to match the dimensions of the larger array.

arr3 = np.array([1, 2, 3])
arr4 = np.array([10])

# Add arr3 (with 3 elements) to arr4 (with 1 element)
```

```
combined = arr3 + arr4
print("broadcasting example sum is: \n", combined)

combined_product = arr3 * arr4
print("broadcasting example product is: \n",
combined_product)
```

Output:

```
Element-wise sum:
 [5 7 9]
Element-wise product:
 [ 4 10 18]
broadcasting example sum is:
 [11 12 13]
broadcasting example product is:
 [10 20 30]
```

Exercise 4.2

Consider the following NumPy array:

array = [1, 3, 4, 5]

Perform the following arithmetic calculation on the array:

1) 2*array,
2) array*array,
3) array2 = [2,4]*array

5. NumPy mathematical and statistical functions

There are several mathematical and statistical functions available in NumPy which can be very useful in manipulating the data of a matrix (dataset). The function ***mean()*** calculates mean of all the elements of the NumPy array, irrespective of the shape of the array, ***var()*** calculates the variance of the elements of the NumPy array, ***std()*** calculates the standard deviation of the elements of the NumPy array, ***min()*** Returns the minimum element in the NumPy array, ***max()*** returns the maximum element in the NumPy array, ***sum()*** returns the sum of the elements of the NumPy array, ***prod()*** returns product of the elements of the NumPy array.

Code:

```
# NumPy Stats functions

X = np.array( [ [-2.5, 3.1, 7],
                [10, 11, 12] ] )
print("mean = ", X.mean())
print("Variance = ", X.var())
print("Standard Deviation = ", X.std())
print("min = ", X.min())
print("max = ", X.max())
print("sum = ", X.sum())
print("product = ", X.prod())
```

Output:

```
mean = 6.766666666666667
Variance = 25.855555555555554
Standard Deviation = 5.084835843520964
min = -2.5
max = 12.0
sum = 40.6
product = -71610.0
```

5.1. Generate Random Number

NumPy offers the random module to work with random numbers.

Code:

```
# Generate a random integer from 0 to 100:

from numpy import random

x = random.randint(100)

print(x)
```

Output:

```
57
```

The random module's rand() method returns a random float between 0 and 1. In NumPy we work with arrays, and you can use the two methods from the above examples to make random arrays. The randint() method takes a size parameter where you can specify the shape of an array.

Code:

```
# Generate a 1-D array containing 5 random integers from
0 to 100:

x=random.randint(100, size=(5))

print(x)
```

Output:

```
[94 13 13 1 94]
```

Code:

```
# Generate a 2-D array with 3 rows, each row containing 5
random integers from 0 to 100:
```

```
x = random.randint(100, size=(3, 5))

print(x)
```

Output:

```
[[80 54 19 74 65]
 [26 60 69 34 25]
 [50 16 53 84 90]]
```

Data Distribution is a list of all possible values, and how often each value occurs. Such lists are important when working with statistics and data science. The random module offers methods that return randomly generated data distributions. A random distribution is a set of random numbers that follow a certain *probability density function*. We can generate random numbers based on defined probabilities using the ***choice()*** method of the random module. The ***choice()*** method allows us to specify the probability for each value. The probability is set by a number between 0 and 1, where 0 means that the value will never occur and 1 means that the value will always occur.

Code:

```
# Generate a 1-D array containing 100 values, where each
value has to be 3, 5, 7 or 9.

   • The probability for the value to be 3 is set to be
     0.1

   • The probability for the value to be 5 is set to be
     0.3

   • The probability for the value to be 7 is set to be
     0.6

   • The probability for the value to be 9 is set to be 0
```

```
x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0],
size=(100))

print(x)
```

Output:

```
[7 7 7 7 5 7 7 7 5 7 7 7 7 3 5 3 7 7 5 7 7 7 7 5 7 5 5 7
5 7 5 5 5 7 3 5 7
 7 5 7 7 5 5 7 5 7 7 7 7 7 7 5 5 7 5 5 5 5 7 7 7 7 7 7 3
7 7 7 7 3 7 3 3 7
 7 7 5 7 5 7 3 7 3 7 7 3 7 5 5 7 7 7 7 5 3 7 7 3 7 7]
```

The Normal Distribution is one of the most important distributions. It is also called the Gaussian Distribution after the German mathematician Carl Friedrich Gauss. It fits the probability distribution of many events, eg. IQ Scores, Heartbeat etc. Use the random.normal() method to get a Normal Data Distribution. It has three parameters:

- loc - (Mean) where the peak of the bell exists.
- scale - (Standard Deviation) how flat the graph distribution should be.
- size - The shape of the returned array.

Code:

```
# Generate a random normal distribution of size 2x3:

from numpy import random

x = random.normal(size=(2, 3))

print(x)
```

Output:

```
[[ 0.06272742  0.61866698  1.08308449]
 [-0.41735761  0.77582487 -1.27106124]]
```

Code:

```
# Generate a random normal distribution of size 2x3 with
mean at 1 and standard deviation of 2:

from numpy import random

x = random.normal(loc=1, scale=2, size=(2, 3))

print(x)
```

Output:

```
[[ 2.07199561  4.41036069  2.42637975]
 [ 4.23287278 -0.3168145   0.01308563]]
```

There are a lot of different statistical distributions available in NumPy. For more information, you can look at NumPy official documentation.

6. SciPy

SciPy is a scientific computation library that uses NumPy underneath, and it stands for Scientific Python. It provides more utility functions for optimization, statistics and even signal processing. SciPy has optimized and added functions that are frequently used in NumPy and Data Science. SciPy is automatically included in the Anaconda distribution of Python but if you use any other version of Python, you can install SciPy using: **conda install anaconda::scipy** in your anaconda command line, or simply use **pip install scipy**. Once SciPy is installed, import the SciPy module(s) you want to use in your applications by adding the

from scipy import module statement. As SciPy is more focused on scientific implementations, it provides many built-in scientific constants. These constants can be helpful when you are working with Data Science.

Code:

```
# import constants from SciPy library
# Then use pi constant, giga and nano conversion
constants, and foot and yard conversion constants.

from scipy import constants
print(constants.pi)
print(constants.giga)
print(constants.nano)
print(constants.foot)
print(constants.yard)
```

Output:

```
3.141592653589793
1000000000.0
1e-09
0.30479999999999996
0.9143999999999999
```

A list of all units under the constants module can be seen using the dir() function.

Code:

```
print(dir(constants))
```

Output:

```
['Avogadro', 'Boltzmann', 'Btu', 'Btu_IT', 'Btu_th',
'C2F', 'C2K', 'ConstantWarning', 'F2C', 'F2K', 'G',
```

```
'Julian_year', 'K2C', 'K2F', 'N_A', 'Planck', 'R',
'Rydberg', 'Stefan_Boltzmann', 'Tester', 'Wien',
'__all__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__',
'__path__', '__spec__', '_obsolete_constants',
'absolute_import', 'acre', 'alpha', 'angstrom', 'arcmin',
'arcminute', 'arcsec', 'arcsecond', 'astronomical_unit',
'atm', 'atmosphere', 'atomic_mass', 'atto', 'au', 'bar',
'barrel', 'bbl', 'c', 'calorie', 'calorie_IT',
'calorie_th', 'carat', 'centi', 'codata', 'constants',
'convert_temperature', 'day', 'deci', 'degree',
'degree_Fahrenheit', 'deka', 'division', 'dyn', 'dyne',
'e', 'eV', 'electron_mass', 'electron_volt',
'elementary_charge', 'epsilon_0', 'erg', 'exa', 'exbi',
'femto', 'fermi', 'find', 'fine_structure',
'fluid_ounce', 'fluid_ounce_US', 'fluid_ounce_imp',
'foot', 'g', 'gallon', 'gallon_US', 'gallon_imp',
'gas_constant', 'gibi', 'giga', 'golden', 'golden_ratio',
'grain', 'gram', 'gravitational_constant', 'h', 'hbar',
'hectare', 'hecto', 'horsepower', 'hour', 'hp', 'inch',
'k', 'kgf', 'kibi', 'kilo', 'kilogram_force', 'kmh',
'knot', 'lambda2nu', 'lb', 'lbf', 'light_year', 'liter',
'litre', 'long_ton', 'm_e', 'm_n', 'm_p', 'm_u', 'mach',
'mebi', 'mega', 'metric_ton', 'micro', 'micron', 'mil',
'mile', 'milli', 'minute', 'mmHg', 'mph', 'mu_0', 'nano',
'nautical_mile', 'neutron_mass', 'nu2lambda', 'ounce',
'oz', 'parsec', 'pebi', 'peta', 'physical_constants',
'pi', 'pico', 'point', 'pound', 'pound_force',
'precision', 'print_function', 'proton_mass', 'psi',
'pt', 'short_ton', 'sigma', 'speed_of_light',
'speed_of_sound', 'stone', 'survey_foot', 'survey_mile',
'tebi', 'tera', 'test', 'ton_TNT', 'torr', 'troy_ounce',
'troy_pound', 'u', 'unit', 'value', 'week', 'yard',
'year', 'yobi', 'yotta', 'zebi', 'zepto', 'zero_Celsius',
'zetta']
```

SciPy has a lot more functionalities but in this lesson, we only review one of these functionalities called optimizers. Optimizers are a set of procedures

defined in SciPy that either find the minimum value of a function, or the root of an equation. Essentially, all of the algorithms in Machine Learning are nothing more than a complex equation that needs to be minimized with the help of given data, and therefore this functionality of SciPy library can be of significant importance. NumPy is capable of finding roots for polynomials and linear equations, but it can not find roots for non-linear equations, like this one:

x + cos(x)

For that you can use SciPy's optimize.root function. This function takes two required arguments: fun - a function representing an equation, and x0 - an initial guess for the root. The function returns an object with information regarding the solution. The actual solution is given under attribute x of the returned object:

Code:

```
# Find root of the equation x + cos(x):

from scipy.optimize import root
from math import cos

def eqn(x):
  return x + cos(x)

myroot = root(eqn, 0)

print(myroot.x)
print(myroot)
```

Output:

```
[-0.73908513]
fjac: array([[-1.]])
     fun: array([ 0.])
 message: 'The solution converged.'
    nfev: 9
```

```
     qtf: array([ -2.66786593e-13])
       r: array([-1.67361202])
  status: 1
 success: True
       x: array([-0.73908513])
```

Note that fjac in the output is a representation of the Jacobian matrix in a compact format. Minimizing a function can also be done with SciPy. A function, in this context, represents a curve, curves have high points and low points. High points are called maxima. Low points are called minima. The highest point in the whole curve is called global maxima, whereas the rest of them are called local maxima.

The lowest point in the whole curve is called global minima, whereas the rest of them are called local minima. We can use scipy.optimize.minimize() function to minimize the function. The minimize() function takes the following arguments: fun - a function representing an equation, and x0 - an initial guess for the root, and a method which is the name of the method to use. Legal values for the method are: 'CG', 'BFGS', 'Newton-CG', 'L-BFGS-B', 'TNC', 'COBYLA', 'SLSQP'.

Let's try to minimize the function x^2 + x + 2 with BFGS method:

Code:

```
from scipy.optimize import minimize

def eqn(x):
  return x**2 + x + 2

mymin = minimize(eqn, 0, method='BFGS')

print(mymin)
```

Output:

```
message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 1.75
       x: [-5.000e-01]
     nit: 2
     jac: [ 0.000e+00]
hess_inv: [[ 5.000e-01]]
    nfev: 8
    njev: 4
```

In the above example, the line mymin - minimize(eqn, 0, method='BFGS') performs the actual minimization. Let's break down the arguments:
eqn: This is the function we want to minimize.
0: This is the initial guess for the minimum. The optimizer will start its search from this point.
method='BFGS': This specifies the optimization algorithm to use. 'BFGS' is a popular quasi-Newton method, known for its efficiency.

Note: *The minimize function iteratively improves the initial guess by calculating the gradient of the function and using it to move towards a lower point. The 'BFGS' method is particularly efficient here because it approximates the Hessian matrix (matrix of second derivatives) using information from previous iterations, reducing the computational cost.*

As mentioned earlier, SciPy has a lot of various functionalities. You can see a list of these functionalities here at SciPy official documentation.

7. Environments

A further concept in Python related to external libraries that is extremely important are environments, also referred to as virtual environments.

Environments are an essential aspect of any Python project. Let me give an example from graduate school to explain this importance. We were taking a statistics course with a focus on astronomy. This involved an in depth data analysis, using some common Python external libraries (that we will go over soon), but also a much less common astronomy-specific Python external library called AstroML, meaning Astronomy Machine Learning. At the start of the course, the professor told us to create a Python environment specifically for that class. For one assignment, we had to change the version of the Python external library NumPy to a specific older version, to be compatible with AstroML. Those who were using a single base environment for all of their work, whether it was courses or research, had their research code break because of this version of the common library, and had to try to remember which version they previously used, and switch back and forth for working on the class and then research. Those who had a dedicated environment for the course experienced no issues as they could easily switch between environments.

When we attempt to import a library (also known as a package) such as NumPy with import numpy as np, where does this actually come from? The answer is that NumPy is installed in your active environment.

While a library is a set of modules, a package is also a set of modules that also contains extra set up files so that it can be easily installed and run inside of an environment. In other words, all packages are libraries but not all libraries are packages. Any Python library like NumPy or SciPy that can be installed through conda or pip is technically a package. The terms package and library can get used somewhat interchangeably.

A key aspect of data analysis is reproducibility. You may write a collection of code that you then want to run on another computer, or you might want to send it to someone else to run on their computer.

Python code can have many dependencies (meaning installed libraries such as NumPy and SciPy), that are relied upon to run properly. For specific packages, there are also

many different versions of each. For example, NumPy currently goes up to version 1.26.0. To ensure that your code can reliably run, you may want the user to have a set of specific packages installed. For example you may want to specify:

NumPy version 1.25.2

SciPy version 0.12.0

Matplotlib version 3.8.4

An analogy for Python environments can be as follows. A function is a tool like a hammer or screwdriver, and a library/package is a toolbox. An environment is like your workbench with a set of toolboxes. Ensuring that these workbenches are identical across systems is integral to your code running properly.

Anaconda is the currently popular package manager for Python. Just like Git, it is a program that can be run on your terminal (for Linux/Mac), or on the Anaconda prompt for Windows.

Now let's go over setting up an environment to create some NumPy arrays and plot them! Firstly, it is important to make sure that Conda is installed and to check what version you are on.

```
$ conda info # check version number
$ conda update conda # update conda
```

Next we can create our environment.

```
$ conda create –name my_env # creates an environment called my_env
$ conda create -n my_env # equivalent can replace - -name with -n
```

Now to install Python, NumPy and Matplotlib to the environment my_env.

$ conda install -n my_env python=3.7 # add python version 3.7
$ conda install -n my_env numpy # add numpy
$ conda install -n my_env conda-forge::matplotlib # add matplotlib

We can check the packages that are installed in my_env

$ conda list -n my_env

And finally we activate the environment, which allows us to run code using the installed packages:

$ conda activate my_env

OR

$ source activate my_env # this is the older way of doing it but might work better in bash scripts

Now with the environment activated, you can either run your Python scripts from the terminal, or open a Jupyter notebook and that will then run using the packages you have installed in that environment.

You are now starting to see the power of combining environments, to run Python code in a reproducible way.

The key commands for Conda to create and edit environments can be found here:
https://docs.conda.io/projects/conda/en/latest/user-guide/cheatsheet.html

You can also clone environments, export them as files, and more.

6.1. Exporting an environment as a .yml file

Another important set of tools is exporting, sending, and activating Python environments. Environments are saved as .yml files. Using the environment we just created 'my_env', we will go through this process.

Exporting the environment is done using conda env export. Without first activating the environment we can export it and specify the .yml filename:

$ conda env export -n my_env > my_env.yml

This will save the .yml environment file in the directory from which you ran this command. After sending this environment to another machine or user, it can be downloaded as follows, from the directory where you saved my_env.yml:

$ conda env create -f my_env.yml

This demonstrates the portability of Python environments across machines!

References and Further Resources:

Official documentation of Math module - Python
Official documentation of NumPy library - NumPy
Official documentation of SciPy library - SciPy
Intro to NumPy - W3School
Intro to SciPy - W3School