0A_Python_Prerequisites_crsra

October 9, 2023

1 Python Programming Refresher

Python, first released in 1991, is a programming language of increasing popularity and the one that shall be used in this course. Later in this course, you'll learn about a framework called TensorFlow, which has codebases supporting other languages such as C++ and Java, but which is best supported for Python. In addition to this practical consideration, Python boasts such benefits as code-readability, easy semantics, interactivity (i.e., you don't have to recompile all the time!), and it supports such common programming paradigms as object-oriented and functional programming.

This walk-through provides an overview of aspects of the Python programming language you'll need to know for the course, including: lists, tuples, dictionaries, for-loops, if-statements, NumPy, and MatPlotLib. We recommend that you are comfortable with all of these before proceeding with the course.

For those unfamiliar with Python, this will likely not be sufficient to bring you fully up-to-speed. Instead, the free e-book Python Like You Mean It, by Ryan Soklaski, will provide an more comprehensive introduction to Python for scientific computing. Additionally, to further learn about the programming language's features and to view its documentation, the website `http://www.python.org` is a phenomenal resource.

1.1 Variables and Assignments

In Python, we store all pieces of data – numbers, characters, strings, *everything* – as objects, and we refer to these objects using variables. As a simple case, we can *assign* a variable a value using the assignment operator, which is the "equals" sign:

[4]:
```
x = 4
y = 5
z = x + y
print(z)
```

```
9
```

Note that we can perform assignment using specific values – as in assigning 4 to variable x and 5 to variable y – and we can also assign a value with respect to other variables – as in assigning the variable z the sum of x and y.

1.2 Python Collections

Python collections allow us to put multiple data items into a single object that we may then access or operate on all together. There are several built-in types of collections, and we shall discuss three of them: lists, tuples, and dictionaries. Later, we'll introduce an additional way to collect data using arrays from a library called *numpy*.

Lists A list comprises a sequence of objects, usually represented using square brackets with commas between the items in the sequence as is done below:

[6]:
```
my_list = ['a', 'b', 'c', 'd']
```

Above, *my_list* contains a sequence of character objects. Lists, however, accomodate items of varying types of objects:

[7]:
```
varied_list = ['a', 1, 'b', 3.14159] # a list with elements of char, integer,␣
↪and float types
nested_list = ['hello', 'governor', [1.618, 42]] # a list within a list!
```

Lists allow for what is called *indexing*, in which a specified element of the list may be obtained. For instance, say you wanted to grab the second element of *varied_list* above. Then you could index the list as so:

[8]:
```
second_element = varied_list[1] # Grab second element of varied_list
print(second_element)
```

```
1
```

Now is a good time to mention that Python is what's called a *zero-indexed* programming language. This simply means that the "first" element in a list or other collection of data items is indexed using "0" (zero) rather than "1". This is why, above, we grab the second element of *varied_list* using the integer index "1" instead of "2" as some might expect from a *one-indexed* language (like MATLab).

Another feature of python indexing that comes in handy is the use of *negative indexing*. As we discussed above, the "first" element of a python list is denoted by index "0"; thus, it is almost natural to consider the last element of the list as being indexed by "-1". Observe the following examples of negative indexing:

[9]:
```
last_element = my_list[-1] # the last element of my_list
last_element_2 = my_list[len(my_list)-1] # also the last element of my_list,␣
↪obtained differently
second_to_last_element = my_list[-2]
```

Similar to indexing is list slicing, in which a contiguous section of list may be accessed. The colon (:) is used to perform slicing, with integers denoting the positions at which to begin and end the slice. Below, we show that the beginning or ending integer for a slice may be omited when one is slicing from the beginning or to the end of the list. Also note below that the index for slice beginning is included in the slice, but the index for the slice end is not included.

[6]:
```
NFL_list = ["Chargers", "Broncos", "Raiders", "Chiefs", "Panthers", "Falcons",␣
↪"Cowboys", "Eagles"]
AFC_west_list = NFL_list[:4] # Slice to grab list indices 0, 1, 2, 3 --␣
↪"Chargers", "Broncos", "Raiders", "Chiefs"
NFC_south_list = NFL_list[4:6] # Slice list indices 4, 5 -- "Panthers",␣
↪"Falcons"
NFC_east_list = NFL_list[6:] # Slice list indices 6, 7 -- "Cowboys", "Eagles"
```

Tuples A tuple is a Python collection that is extremely similar to a list, with some subtle differences. For starters, tuples are indicated using parentheses instead of square brackets:

[7]:
```
x = 1
y = 2
coordinates = (x, y)
```

The variable *coordinates* above is a tuple containing the variables x and y. This example was chosen to also demonstrate a difference between the typical usage of tuples versus lists. Whereas lists are frequently used to contain objects whose values are similar in some sense, tuples are frequently used to contain attributes of a coherent unit. For instance, as above, it makes sense to treat the coordinates of a point as a single unit. As another example, consider the following tuple and list concerning dates:

[18]:
```
year1 = 2011
month1 = "May"
day1 = 18
date1 = (month1, day1, year1)
year2 = 2017
month2 = "June"
day2 = 13
date2 = (month2, day2, year2)
years_list = [year1, year2]
print(years_list)
print(date1)
```

```
[2011, 2017]
('May', 18, 2011)
```

Notice above that we have collected the attributes of a single date into one tuple: those pieces of information all describe a single "unit". By contrast, in the years list we have collected the different years in the code-snippet: the values in the list have a commonality (they are both years), but they do not describe the same unit.

The distinction I've drawn between tuples and lists here is one that many Python programmers recognize in practice, but not one that is strictly enforced (i.e., you won't get any errors if you break this convention!). Another subtle way in which tuples and lists differ involves what is called *mutability* of Python variables. Mutability is a more complex concept that we need not discuss in our Python introduction, but interested students are encouraged to read further if they like!

Dictionaries Since you've seen parenthesis (for tuples) and square brackets (for lists), you may be wondering what curly braces are used for in Python. The answer: Python dictionaries. The defining feature of a Python dictionary is that it has **keys** and **values** that are associated with each other. When defining a dictionary, this association may be accomplished using the color (:) as is done below:

[9]:
```
book_dictionary = {"Title": "Frankenstein", "Author": "Mary Shelley", "Year":␣
↪1818}
print(book_dictionary["Author"])
```

```
Mary Shelley
```

Above, the keys of the *book_dictionary* are "Title", "Author", and "Year", and each of these keys has a corresponding value associated with it. Notice that the key-value pairs are separated by a comma. Using keys allows us to access a piece of the dictionary by its name, rather than needing to know the index of the piece that we want, as is the case with lists and tuples. For instance, above we could get the author of Frankenstein using the "Author" key, rather than using an index. In fact, unlike in a list or tuple, the order of elements in a dictionary doesn't matter, and dictionaries cannot be indexed using integers, which we see below when we try to access the second element of the dictionary using an integer:

[10]:
```
print(book_dictionary[1])
```

```
      ␣
↪---------------------------------------------------------------------------

      KeyError                                  Traceback (most recent call␣
↪last)

      <ipython-input-10-43bbaea82a52> in <module>()
   ----> 1 print(book_dictionary[1])


      KeyError: 1
```

1.3 Python Control Flow

As with most of the popular programming languages, Python offers several ways to control the flow of execution within a program. Here, we'll introduce looping and conditional statements.

At this point, it is important to make you aware of a unique aspect of the Python language: indentation. While a language like C uses curly braces to contain code statements within loops or conditionals, Python indicates these statements through indentation. This feature lends Python code readability, as you will see in the examples below.

1.3.1 For-Loops

Looping statements allow for the repeated execution of a section of code. For instance, suppose we wanted to add up all of the integers between zero (0) and ten (10), not including ten. We could, of course, do this in one line, but we could also use a loop to add each integer one at a time. Below is the code for a simple accumulator that accomplishes this:

[19]:
```
sum = 0
for i in range(10):
    sum = sum + i
    print(i)
print(sum)
alternative_sum = 0+1+2+3+4+5+6+7+8+9
print(alternative_sum==sum)
```

```
0
1
2
3
4
5
6
7
8
9
45
True
```

The *range*() built-in function generates the sequence of values that we loop over, and notice that range(10) does not include 10 itself. In addition to looping over a sequence of integers using the range() function, we can also loop over the elements in a list, which is shown below:

[12]:
```
ingredients = ["flour", "sugar", "eggs", "oil", "baking soda"]
for ingredient in ingredients:
    print(ingredient)
```

```
flour
sugar
eggs
oil
baking soda
```

Above, the for-loop iterates over the elements of the list *ingredients*, and within the loop each of those elements is referred to as *ingredient*. The use of singular/plural nouns to handle this iteration is a common Python motif, but is by no means necessary to use in your own programming.

1.3.2 Conditionals

Oftentimes while programming, one will want to only execute portions of code when certain conditions are met, for instance, when a variable has a certain value. This is accomplished using conditional statements: **if**, **elif**, and **else**.

[13]:
```
for i in range(10):
    if i % 2 == 0: # % -- modulus operator -- returns the remainder after␣
 ↪division
        print("{} is even".format(i))
    else:
        print("{} is odd".format(i))
```

```
0 is even
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
```

[14]:
```
# Example using elif as well
# Print the meteorological season for each month (loosely, of course, and in␣
 ↪the Northern Hemisphere)
print("In the Northern Hemisphere: \n")
month_integer = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] # i.e., January is 1,␣
 ↪February is 2, etc...
for month in month_integer:
    if month < 3:
        print("Month {} is in Winter".format(month))
    elif month < 6:
        print("Month {} is in Spring".format(month))
    elif month < 9:
        print("Month {} is in Summer".format(month))
    elif month < 12:
        print("Month {} is in Fall".format(month))
    else: # This will put 12 (i.e., December) into Winter
        print("Month {} is in Winter".format(month))
```

```
In the Northern Hemisphere:

Month 1 is in Winter
Month 2 is in Winter
Month 3 is in Spring
Month 4 is in Spring
```

```
Month 5 is in Spring
Month 6 is in Summer
Month 7 is in Summer
Month 8 is in Summer
Month 9 is in Fall
Month 10 is in Fall
Month 11 is in Fall
Month 12 is in Winter
```

1.3.3 List Comprehension

Python allows for list comprehension in which the elements of a list are iterated over all in one line of code. Say, for example, that you wanted to add 1 to each element in a list of integers. You could do so using list comprehension as so:

[15]:
```
even_list = [2, 4, 6, 8]
odd_list = [even+1 for even in even_list]
print(odd_list)
```

```
[3, 5, 7, 9]
```

Note from above the similarities between list comprehension and a for-loop; Python has list comprehension as a compact, "pythonic" way of performing operations that could be done within a for-loop.

1.4 NumPy Library

The NumPy library endows Python with a host of scientific computing capabilities. Chief among these is the Array object, which provides a multidimensional way to organize values of the same type. Numpy arrays allow slicing and indexing similar to lists. Most importantly, Numpy has a formidable number of mathematical operations that can be used to transform arrays and perform computations between arrays. For those familiar with MATLab, these operations should be reminiscent of many matrix operations.

[2]:
```
import numpy as np

x = np.array([2, 4, 6]) # create a rank 1 array
A = np.array([[1, 3, 5], [2, 4, 6]]) # create a rank 2 array
B = np.array([[1, 2, 3], [4, 5, 6]])

print("Matrix A: \n")
print(A)

print("\nMatrix B: \n")
print(B)
```

```
Matrix A:

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

Matrix B:

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

[17]:
```
# Indexing/Slicing examples
print(A[0, :]) # index the first "row" and all columns
print(A[1, 2]) # index the second row, third column entry
print(A[:, 1]) # index entire second column
```

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

[18]:
```
# Arithmetic Examples
C = A * 2 # multiplies every elemnt of A by two
D = A * B # elementwise multiplication rather than matrix multiplication
E = np.transpose(B)
F = np.matmul(A, E) # performs matrix multiplication -- could also use np.dot()
G = np.matmul(A, x) # performs matrix-vector multiplication -- again could also␣
 ↪use np.dot()

print("\n Matrix E (the transpose of B): \n")
print(E)

print("\n Matrix F (result of matrix multiplication A x E): \n")
print(F)

print("\n Matrix G (result of matrix-vector multiplication A*x): \n")
print(G)
```

```
 Matrix E (the transpose of B):

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

 Matrix F (result of matrix multiplication A x E):

[[22 49]
 [28 64]]
```

```
 Matrix G (result of matrix-vector multiplication A*x):

[44 56]
```

[19]:
```
# Broadcasting Examples
H = A * x # "broadcasts" x for element-wise multiplication with the rows of A
print(H)
J = B + x # broadcasts for addition, again across rows
print(J)
```

```
[[ 2 12 30]
 [ 4 16 36]]
[[ 3  6  9]
 [ 6  9 12]]
```

[20]:
```
# max operation examples

X = np.array([[3, 9, 4], [10, 2, 7], [5, 11, 8]])
all_max = np.max(X) # gets the maximum value of matrix X
column_max = np.max(X, axis=0) # gets the maximum in each column -- returns a␣
 ↪rank-1 array [10, 11, 8]
row_max = np.max(X, axis=1) # gets the maximum in each row -- returns a rank-1␣
 ↪array [9, 10, 11]

# In addition to max, can similarly do min. Numpy also has argmax to return␣
 ↪indices of maximal values
column_argmax = np.argmax(X, axis=0) # note that the "index" here is actually␣
 ↪the row the maximum occurs for each column

print("Matrix X: \n")
print(X)
print("\n Maximum value in X: \n")
print(all_max)
print("\n Column-wise max of X: \n")
print(column_max)
print("\n Indices of column max: \n")
print(column_argmax)
print("\n Row-wise max of X: \n")
print(row_max)
```

```
Matrix X:

[[ 3  9  4]
 [10  2  7]
 [ 5 11  8]]

 Maximum value in X:
```

```
11

 Column-wise max of X:

[10 11  8]

 Indices of column max:

[1 2 2]

 Row-wise max of X:

[ 9 10 11]
```

[21]:
```
# Sum operation examples
# These work similarly to the max operations -- use the axis argument to denote␣
 ↪if summing over rows or columns


total_sum = np.sum(X)
column_sum = np.sum(X, axis=0)
row_sum = np.sum(X, axis=1)

print("Matrix X: \n")
print(X)
print("\n Sum over all elements of X: \n")
print(total_sum)
print("\n Column-wise sum of X: \n")
print(column_sum)
print("\n Row-wise sum of X: \n")
print(row_sum)
```

```
Matrix X:

[[ 3  9  4]
 [10  2  7]
 [ 5 11  8]]

 Sum over all elements of X:

59

 Column-wise sum of X:

[18 22 19]

 Row-wise sum of X:
```

```
[16 19 24]
```

[3]:
```
# Matrix reshaping

X = np.arange(16) # makes a rank-1 array of integers from 0 to 15
X_square = np.reshape(X, (4, 4)) # reshape X into a 4 x 4 matrix
X_rank_3 = np.reshape(X, (2, 2, 4)) # reshape X to be 2 x 2 x 4 --a rank-3 array
                                    # consider as two rank-2 arrays with 2 rows␣
 ↪and 4 columns
print("Rank-1 array X: \n")
print(X)
print("\n Reshaped into a square matrix: \n")
print(X_square)
print("\n Reshaped into a rank-3 array with dimensions 2 x 2 x 4: \n")
print(X_rank_3)
```

```
Rank-1 array X:

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]

 Reshaped into a square matrix:

[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]

 Reshaped into a rank-3 array with dimensions 2 x 2 x 4:

[[[ 0  1  2  3]
  [ 4  5  6  7]]

 [[ 8  9 10 11]
  [12 13 14 15]]]
```

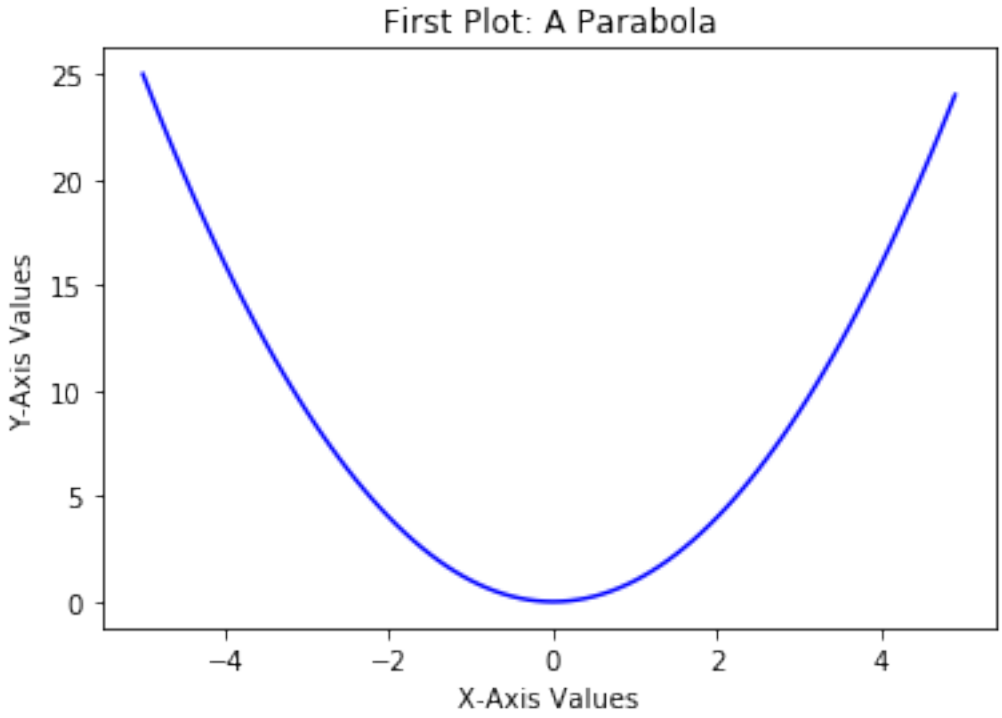
1.5 Plotting

Much of plotting you'll do will be through the Matplotlib library, specifically within the Pyplot module. Aptly named, the *plot* function is used to plot 2-D data, as shown below:

[23]:
```
import matplotlib.pyplot as plt
```

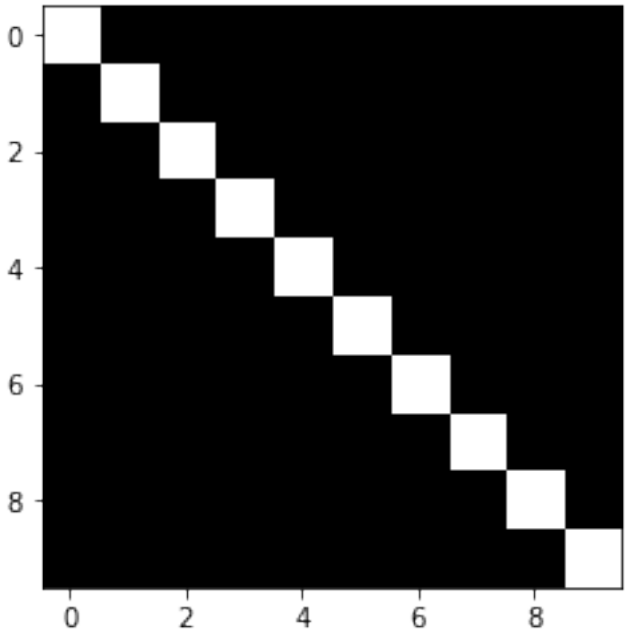
[24]:
```
import matplotlib.pyplot as plt
import numpy as np

# We'll start with a parabola
# Compute the parabola's x and y coordinates
```

```
x = np.arange(-5, 5, 0.1)
y = np.square(x)

# Use matplotlib for the plot
plt.plot(x, y, 'b') # specify the color blue for the line
plt.xlabel('X-Axis Values')
plt.ylabel('Y-Axis Values')
plt.title('First Plot: A Parabola')
plt.show() # required to actually display the plot
```

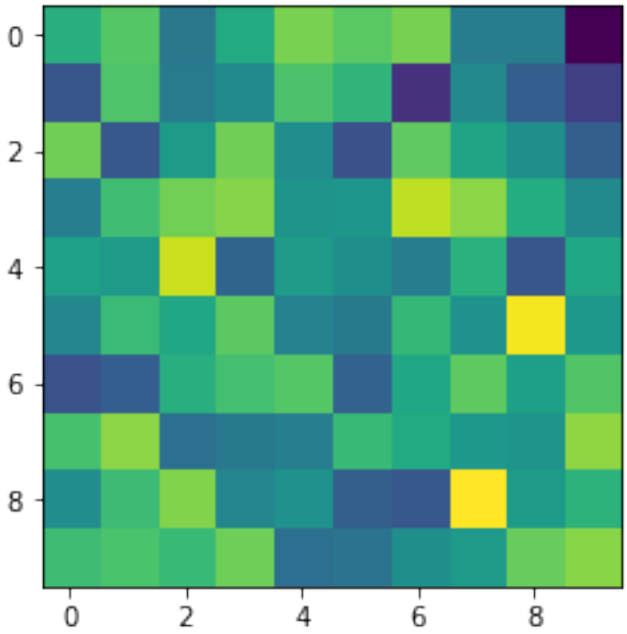


Another Matplotlib function you'll encounter is *imshow* which is used to display images. Recall that an image may be considered as an array, with array elements indicating image pixel values. As a simple example, here is the identity matrix:

[25]:
```
import numpy as np
import matplotlib.pyplot as plt

X = np.identity(10)
identity_matrix_image = plt.imshow(X, cmap="Greys_r")
plt.show()
```



[26]:
```
# Now plot a random matrix, with a different colormap
A = np.random.randn(10, 10)
random_matrix_image = plt.imshow(A)
plt.show()
```



[ ]:

[ ]: