00_Python_intro

September 2, 2024

0.1 Problem 1: Python - the basic syntax

We are going to briefly introduce you to Python in this assignment. This introduction is by no means comprehensive. I highly recommend you brush up on Python through a few tutorials: * https://wiki.python.org/moin/BeginnersGuide * https://www.w3schools.com/python/

Python provides an extensive amount of documentation, e.g., https://docs.python.org/3.10/reference/index.html. Googling a command or question is also quite useful.

You will now go through a basic series of tutorials. Take as long or short as you need to ensure you feel like you know what is going on for the questions below. The tutorials have a fair amount of detail, so **you may want to skim over some of the topics and take note that they exist and come back to them as you need** (e.g., Python Operators are pretty close to c++, you might just scroll through the list and call it good and then come back later as needed). Come back to these tutorials throughout the semester as you need. From https://www.w3schools.com/python/, complete the following tutorials: * Python Intro * Python Syntax * Python Comments * Python Variables * Python Data Types * Python Numbers * Python Strings * Python Booleans * Python Operators

Note that in the code below there is an `import` statement. That statement imports a function from an existing package that allows the variables to be visualized within a Jupyter notebook.

[83]:
```
from IPython.display import display # Used to display variables nicely in␣
↪Jupyter

# Modify the x, y, and z variables to have the number one in a integer, float,␣
↪and string
x = 1 # Should be an integer
display("x = ", x)
y = 1.0 # Should be a float
display("y = ", y)
z = "one" # Should be a string
display("z = ", z)
```

```
'x = '

1

'y = '

1.0
```

'z = '

'one'

[84]:
```python
# Add two to x, y, and z using the "+" operator
x = 0 + 2 # Add number two
display("x = ", x)
y = 0.0 + 2.0
display("y = ", y)
z = "one" + "two" # Add the string "two"
display("z = " , z)
```

'x = '

2

'y = '

2.0

'z = '

'onetwo'

0.2 Problem 2: The list

The list is just what it sounds like. It provides a list of elements of any type. Complete the following tutorial and then coding exercise below.

Tutorial: https://www.w3schools.com/python/python_lists.asp

[85]:
```python
from IPython.display import display # Used to display variables nicely in␣
↪Jupyter

# Create a list with the elements 1, 2., "three"
x = ["1", "2.", "three"]
display("x = ", x)

# Copy the 0th element to the variable zero
zero = x[0]
display("zero = ", zero)

# Copy the final element to the variable "final". Note that the index `-1`␣
↪corresponds to the final element
final = x[-1]
display("final = ", final)

# Append the item 4. to the end of the list and display the list
x.append(4.)
display("x = ", x)
```

'x = '

```
['1', '2.', 'three']

'zero = '

'1'

'final = '

'three'

'x = '

['1', '2.', 'three', 4.0]
```

0.3 Problem 3: The tuple

Tuples are similar to lists, but the collection is unchangeable. Complete the following tutorial and coding exercise.

Tutorial: https://www.w3schools.com/python/python_tuples.asp

[86]:
```
from IPython.display import display # Used to display variables nicely in
 ↪Jupyter

# Create a tuple with the elements 5, 7., and "apple"
x = ("5", "7.", "apple")

# Display the 0th element
zero = x[0]
display("zero = ", zero)

# Display the 1 element
one = x[1]
display("one = ", one)

# Display the last element
last = x[-1]
display("last = ", last)
```

```
'zero = '

'5'

'one = '

'7.'

'last = '

'apple'
```

0.4 Problem 4: The dict

The dictionary provides a mapping data type that you can use to map one item to another. Complete the following tutorial and coding exercise.

Tutorial: https://www.w3schools.com/python/python_dictionaries.asp

[87]:
```
from IPython.display import display # Used to display variables nicely in
 ↪Jupyter

# Create a dictionary that maps "apples" to "oranges" and "1" to "one"
x = {
        "apples": "oranges",
        "one": 1
}
display("x = ", x)

# Lookup the "apples" element from within x and display it
lookup = x["apples"]
display("lookup = ", lookup)

# Add the mapping from "two" to 2. and display the resulting dictionary
x["two"] = 2
display("x = ", x)
```

```
'x = '

{'apples': 'oranges', 'one': 1}

'lookup = '

'oranges'

'x = '

{'apples': 'oranges', 'one': 1, 'two': 2}
```

0.5 Problem 5: Structural programming

Complete the following tutorials from w3schools and the coding exercise. * Python If…Else * Python While Loops * Python For Loops

[88]:
```
# Create a for loop that displays all of the elements in x one at a time
x = ["four", 5., 6]
for i in x:
    display(i)

# Create a for loop that loops through and displays the keys of y one at a time
y = {"four": 4, "five": 5., "six": 6.}
for i in y:
    display(y[i])

# Create a for loop that iterates through the values in x, checks to see if the
 ↪value is a key within y.
# If it is a key within y, then display the mapping to the value. If not, then
 ↪display a string stating that
```

```python
# the particular list value is not in y
for i in x:
    if i in y:
        print(f"{i} maps to {y[i]}")
    else:
        print(f"{i} is not in y")
```

```
'four'
```

```
5.0
```

```
6
```

```
4
```

```
5.0
```

```
6.0
```

```
four maps to 4
5.0 is not in y
6 is not in y
```

0.6 Problem 6: Intro to Numpy

Numpy is an essential package developed for mathematical operations in Python. We will use it for extensively for matrix and general algebraic operations. Complete the following tutorials and coding exercise. * Numpy for beginners * Numpy for Matlab users * From w3schools.com * Getting started * Creating arrays

We will work quite heavily with numpy matrices. A numpy matrix can be created in a host of ways, but the most straight forward is to use the `np.array` initializer. In this case, each row of the matrix is initialized using an array and the matrix is an array of arrays. For example, the following matrix $ex_{mat} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ can be initialized as

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

where the array `[1., 2., 3.]` is the first row and the array `[4., 5., 6.]` is the second.

Perform the following matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$$

There are two multiplication operators that you can utilize. The first is the asterisk, `*`, and the second is the ampersand, `@`. Be careful as they produce severely different results. Perform each multiplication, display the result, and answer the following questions.

0.6.1 Question: What is the difference between `*` and `@`?

> Answer: The `*` operator multiply the numbers in each of the matching columns, and the `@` operator multiplies the matrices like traditional matrices.

[89]:
```python
from IPython.display import display # Used to display variables nicely in␣
 ↪Jupyter
import numpy as np

# Create the matrices (I've provided one for you)
A = np.array([[1, 2],
              [3, 4]])
B = np.array([[2, 1],
              [1, 0]])

# Multiply the matrices together (use the asterisk, i.e., A*B)
res_bad = A*B
display("Bad result from * = ", res_bad)

# Multiply the matrices together (use the ampersand, i.e., A@B)
res_good = A@B
display("Good result from @: ", res_good)
```

```
'Bad result from * = '

array([[2, 2],
       [3, 0]])

'Good result from @: '

array([[ 4,  1],
       [10,  3]])
```

Now, perform the matrix multiplication for

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

Calculate the shape of each matrix and the result and answer the following question:

0.6.2 Question: What do the elements of the .shape tuple correspond to?

Answer: A tuple that describes the dimensions of the array (the dimension of the matrix). Each element of the `.shape` tuple corresponds to the size of the array along with a specific dimension.

[90]:
```python
from IPython.display import display # Used to display variables nicely in␣
 ↪Jupyter
import numpy as np

# Create the matrices (I've provided one for you)
A = np.array([[1, 2, 3],
                          [4, 5, 6],
                          [7, 8, 9]])
B = np.array([[1],
```

```
                          [2],
                          [3]])

# Calculate the matrix multiplication of A and B
result = A@B
display("A times B = ", result)

# Calculate the shape of each
display("Shape of A: ", A.shape)
display("Shape of B: ", B.shape)
display("Shape of result: ", result.shape)
```

```
'A times B = '

array([[14],
       [32],
       [50]])

'Shape of A: '

(3, 3)

'Shape of B: '

(3, 1)

'Shape of result: '

(3, 1)
```

Now, let's extract elements from the matrices. There are two main ways of getting an element out of a matrix. 1. Double indexing: `A[1,2]` gets the element from row 1 and column 2. Remember zero indexing! 2. .item: `A.item(4)` gets the fourth item stored in the matrix

Complete the following exercise and answer the following.

0.6.3 Question: How would you relate the .item to rows and columns?

> Answer: `.item(0)` will give the first element of the flattened array, which corresponds to `A[0, 0]` (the element at row 0, column 0).

You must run the above code prior to running the next code.

[91]:
```
# Display items 0 through 8 of A using the .item function
for k in range(9):
    print(f"Item {k} of A is: {A.item(k)}")

# Use double indexing to extract the number 6 from A
res = A[1, 2]
display("Result: ", res)
```

```
Item 0 of A is: 1
Item 1 of A is: 2
Item 2 of A is: 3
```

```
Item 3 of A is: 4
Item 4 of A is: 5
Item 5 of A is: 6
Item 6 of A is: 7
Item 7 of A is: 8
Item 8 of A is: 9
```

```
'Result: '
```

```
6
```

Be careful with dimensions. Note the following two ways to extract the middle column of A. Calculate the shape of each of the results and answer the following question.

0.6.4 Question: What is the difference between the two methods?

> Answer: The difference is that *Method 1* gives you a **single list of numbers**, while *Method 2* keeps the **column in a 2D format**, retaining its structure as part of a matrix.

[92]:
```python
# Note that you'll need to run the previous two cells before running this cell

# Method 1 for extracting the middle column:
mid_col_1 = A[:, 1]
display("Method 1: ", mid_col_1)

# Method 2 for extracting the middle column:
mid_col_2 = A[:, [1]]
display("Method 2: ", mid_col_2)

# Calculate the shape of each of the results
display("Method 1 shape: ", mid_col_1.shape)
display("Method 2 shape: ", mid_col_2.shape)
```

```
'Method 1: '
```

```
array([2, 5, 8])
```

```
'Method 2: '
```

```
array([[2],
       [5],
       [8]])
```

```
'Method 1 shape: '
```

```
(3,)
```

```
'Method 2 shape: '
```

```
(3, 1)
```

0.7 Problem 7: Vector products

Assume that $\times$ represents a cross-product, $\circ$ represents a dot product, and $\cdot$ represents matrix or scalar multiplication. The notation v_1^T is used to represent the transpose of v_1. Use the following vectors for this problem:

$$v_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, v_2 = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}.$$

Do the following * Evaluate $v_1 \circ v_2$ using the function `np.dot` * Evaluate $v_1^T \cdot v_2$ using the function `v1.transpose()` and matrix multiplication * Evaluate $v_1 \times v_2$ using the function `np.cross`

Answer the following question: ### Question: What is the difference between the result returned from np.dot vs matrix multiplication for the dot product? > Answer: `np.dot` Computes the dot product for 1D arrays (vectors) and performs matrix multiplication for 2D arrays (matrices). For higher-dimensional arrays, it performs a more general sum-product operation. `Matrix Multiplication` (@ or np.matmul): Specifically handles matrix multiplication for 2D arrays and performs batch matrix multiplication for higher-dimensional arrays.

[93]:
```python
from IPython.display import display # Used to display variables nicely in␣
↪Jupyter
import numpy as np

# Define the vectors (ensure you define them as column vectors)

v1 = np.array([[1],
                            [2],
                            [3]])
v2 = np.array([[4],
                            [5],
                            [6]])

# Evaluate $v_1 \circ v_2$ using the function `np.dot`

display(np.dot(np.reshape(v1, (3,)),np.reshape(v2, (3,))))

# Note that the np.dot() function requires each input to
# be a vector and not a matrix. You can reshape the column
# vector into a numpy vector using np.reshape(v1, (3,))


# Evaluate $v_1^T \cdot v_2$ using the function `v1.transpose()` and matrix␣
↪multiplication

display(np.dot(np.reshape(np.transpose(v1), (3,)),np.reshape(v2, (3,))))

# Evaluate $v_1 \times v_2$ using the function `np.cross`
```

```python
display(np.cross(np.reshape(v1, (3,)),np.reshape(v2, (3,))))

# Note that the np.cross() function requires each input to
# be a vector and not a matrix. You can reshape the column
# vector into a numpy vector using np.reshape(v1, (3,))
```

```
32
```

```
32
```

```
array([-3,  6, -3])
```