Review of Python, Lists, & NumPy

"A complex system that works is invariably found to have evolved from a simple system that worked" – John Gall

---

Python

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Python is dynamically typed and garbage-collected.

Assembly language -> C -> Python

The progression from assembly language to C to Python represents a move up the abstraction ladder in the world of programming languages. Each step abstracts away some of the lower-level details, making it easier to write complex software.

In [ ]:
```
# Define a variable
a = 1
print(a)
```

```
1
```

In [ ]:
```
# Dynamically change the type of the variable
print(type(a))
```

```
<class 'float'>
```

In [ ]:
```
# Control Statements
if a == 1:
    print("a is equal to 1")
```

```
a is equal to 1
```

In [ ]:
```
# Functions
def my_function(a):
    print("What value is passed to the my_function:", a)
    # Return a value
    return a + 1

returned_value = my_function(a)

print("The returned value is:", returned_value)
```

```
What value is passed to the my_function: 1
The returned value is: 2
```

Differences between Python Lists and NumPy Arrays

Python, by default, offers a list object that can be used to represent arrays. For more advanced operations, especially for numerical tasks like matrix operations, NumPy, a third-party library, provides a more efficient and feature-rich array object.

NumPy is based on two fundamental objects: an N-dimensional array object (ndarray) and a universal function object (ufunc). An N-dimensional array is a homogeneous collection of "items" indexed by N integers. There two essential pieces of metadata that define an N-dimensional array: 1) the shape of the array, and 2) the kind of items that composes the array.

In this notebook, we'll explore the differences between standard Python and NumPy in terms of math and matrix operations.

In [ ]:
```
# They both used Brackets []

# Python List
list = [1,2,3]
print("Python list:", list)

# Numpy array
import numpy as np
arr = np.array([1,2,3])
print("NumPy array:", arr)
```

```
Python list: [1, 2, 3]
NumPy array: [1 2 3]
```

Both are zero-indexed, meaning that the first element in the array is 0, the second is 1...

Slicing allows us to extract a portion of the array. It's done using the ":" or "..." symbols. The basic syntax is start:stop:step

- start is the index where slicing starts(inclusive)
- stop is the index where slicing stops(exclusive)
- step specifies the interval between elements.

For example, `array[2:5]` would give you elements from index 2 to 4. `array[::2]` would give you every second element.

In [ ]:
```
'''
Both can access the elements with an index. All arrays in both NumPy and Python
meaning that the first element in the array is 0, the second element is 1, and s
'''

# Python List
print("Python list:",list[0])

# Numpy Array
print("Numpy Array:",arr[0])

# And slice/filter out elements
```

```
# Python List
print("Python list:",list[:2])

# Numpy Array
print("Numpy Array:",arr[:2])
```

```
Python list: 1
Numpy Array: 1
Python list: [1, 2]
Numpy Array: [1 2]
```

In [ ]:
```
# Python lists don't support mathematical operations

list = list*2
print("Python list:", list)

# Multiplication repeats the array elements
```

```
Python list: [1, 2, 3, 1, 2, 3]
```

In [ ]:
```
# NumPy array's support mathematical operations.  Very important for us!

arr = arr*2
print("NumPy array:",arr)
```

```
NumPy array: [2 4 6]
```

In [ ]:
```
# NumPy array's can set the data types.   Very import for us!
arr = np.array([1,2,3], np.uint8)
print(arr.dtype)
```

```
uint8
```

In [ ]:
```
# Create an empty array. Return a new array of given shape and type, without ini

# Create an empty array
arr = np.empty((3,3))
print(arr)

# The output are garbage values from memory.  We need to initialize the array.
```

```
[[ 2.60605835e-31 -5.21211670e-31  1.30302917e-31]
 [-5.21211670e-31  1.13363538e-30 -3.51817877e-31]
 [ 1.30302917e-31 -3.51817877e-31  2.01969522e-31]]
```

In [ ]:
```
# Create an 2D array of zeros
arr = np.zeros((2,3))
print("Zeros:\n",arr)

# Create an 2D array of ones
arr = np.ones((2,3))
print("Ones:\n",arr)
```

```
Zeros:
 [[0. 0. 0.]
 [0. 0. 0.]]
Ones:
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

In [ ]:
```
# Array with a range of values
range_array = np.arange(0, 10, 2)
print("Range Array:\n", range_array)

# Array of evenly spaced values over a specified range
linspace_array = np.linspace(0, 1, 5)
print("Linear Space Array:\n", linspace_array)
```

```
Range Array:
 [0 2 4 6 8]
Linear Space Array:
 [0.   0.25 0.5  0.75 1.  ]
```

In [ ]:
```
array = np.arange(60)
print("1D Array:\n", array)
```

```
1D Array:
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59]
```

In [ ]:
```
reshaped_array = array.reshape(3,4,5)
print("3D Array:\n", reshaped_array)
```

```
3D Array:
 [[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]
  [30 31 32 33 34]
  [35 36 37 38 39]]

 [[40 41 42 43 44]
  [45 46 47 48 49]
  [50 51 52 53 54]
  [55 56 57 58 59]]]
```

In [ ]:
```
print(reshaped_array[2,:,:])
```

```
[[40 41 42 43 44]
 [45 46 47 48 49]
 [50 51 52 53 54]
 [55 56 57 58 59]]
```

In [ ]:
```
'''
The first index is the row,
the second index is the column,
and the third index is the depth.
'''
print("3D Array:\n", reshaped_array)


print("\nThe third column of each dimension:\n ",reshaped_array[...,3])
```

```
3D Array:
 [[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]
  [30 31 32 33 34]
  [35 36 37 38 39]]

 [[40 41 42 43 44]
  [45 46 47 48 49]
  [50 51 52 53 54]
  [55 56 57 58 59]]]

The third column of each dimension:
  [[ 3  8 13 18]
 [23 28 33 38]
 [43 48 53 58]]
```

In [ ]:
```
'''
The first index is the row,
the second index is the column,
and the third index is the depth.
'''
print("3D Array:\n", reshaped_array)


print("\nThe third column of each dimension:\n ",reshaped_array[:,:,3])
```

```
3D Array:
 [[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]
  [30 31 32 33 34]
  [35 36 37 38 39]]

 [[40 41 42 43 44]
  [45 46 47 48 49]
  [50 51 52 53 54]
  [55 56 57 58 59]]]

The third column of each dimension:
  [[ 3  8 13 18]
 [23 28 33 38]
 [43 48 53 58]]
```

In [ ]:
```
print(reshaped_array[1,...,3])
```

```
[23 28 33 38]
```

In [ ]:
```
print(reshaped_array[:,:,3])
```

```
[[ 3  8 13 18]
 [23 28 33 38]
 [43 48 53 58]]
```

In [ ]:
```
print(reshaped_array[0,::2,::2])
```

```
[[ 0  2  4]
 [10 12 14]]
```

In [ ]:
```
# Statistical Functions

# Sum
sum_val = np.sum(np.array([1,2,3,4]))
print("Sum:", sum_val)

# Mean
mean_val = np.mean(np.array([1,2,3,4]))
print("\nMean:", mean_val)
```

```
Sum: 10

Mean: 2.5
```

Matrices

In [ ]:
```
import numpy as np

# Create two 2x2 matrices
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [0, 2]])

print("A:\n",A)

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

```
A:
 [[1 2]
 [3 4]]
B:
 [[2 0]
 [0 2]]
```

In [ ]:
```
# Addition
C = A + B
print("Addition:\n", C)

# Subtraction
D = A - B
print("\nSubtraction:\n", D)
```

```
Addition:
 [[3 2]
 [3 6]]

Subtraction:
 [[-1  2]
 [ 3  2]]
```

In [ ]:
```
'''
A
[1, 2]
[3, 4]

B
[2, 0]
[0, 2]

# 1 x 2 + 2 x 0 = 2
# 1 x 0 + 2 x 2 = 4
# 3 x 2 + 4 x 0 = 6
# 3 x 0 + 4 x 2 = 8

'''

print("A:\n",A)

print("B:\n",B)

# Multiplication
E = np.dot(A, B)
print("\nDot:\n", E)
```

```
A:
 [[1 2]
 [3 4]]
B:
 [[2 0]
 [0 2]]

Dot:
 [[2 4]
 [6 8]]
```

In [ ]:
```
print("A:\n",A)

print("B:\n",B)

# Element-wise multiplication
F = A * B
print("\nElement-wise multiplication:\n", F)
```

```
A:
 [[1 2]
 [3 4]]
B:
 [[2 0]
 [0 2]]

Element-wise multiplication:
 [[2 0]
 [0 8]]
```

In [ ]:
```
print("A:\n",A)

# Transpose
G = A.T
print("\nTranspose of A:\n", G)
```

```
A:
 [[1 2]
 [3 4]]

Transpose of A:
 [[1 3]
 [2 4]]
```

Slicing

In [ ]: