NumPy Array Indexing

In this lesson, array indexing in NumPy is explained.

We'll cover the following

Indexing for a 1-D array

Just like normal arrays, elements of a `NumPy` array can also be accessed and changed through indexing. Through `NumPy` indexing, we can also access and change elements in a specific range.

The following code snippet provides an example of all these functionalities:

```
import numpy as np

arr = np.arange(0,10,1) # Generate array with numbers from 0 to 9
print("The Array")
print(arr)

print("\nElement at index 5")
print(arr[5]) # Fetch element at index 5

print("\nElements in a range of 0 to 6")
print(arr[0:6]) # Fetch elements in a range

arr[0:6] = 20 # Assign a value to a range of elements
print("\nNew array after changing elements in a range of 0 to 6")
print(arr)
```

Note: The `:` inside the `[]` defines the range. The value to the left is the

> starting index and is inclusive. Meanwhile, the value on the right defines the
>
> ending index, which is exclusive; exclusive means that the value at the end index will not be considered for the resultant.

Slicing 1-D array

Slicing an array means taking elements from it in a specific range and moving them to another variable. The concept is the same as `string` slicing, but here an array will be sliced. Let's look at an example:

```
import numpy as np

arr = np.arange(0,10,1)
print("The Array")
print(arr)

new_arr1 = arr[1:7] # This command selects a range of elements from the array
print("\nThe new sliced array")
print(new_arr1)

new_arr2 = arr[:] # This command selects all elements in the array
print("\nThe new sliced array with all elements")
print(new_arr2)
```

This can be useful when certain operations need to be performed on a specific portion of the array.

Indexing for a 2-D array

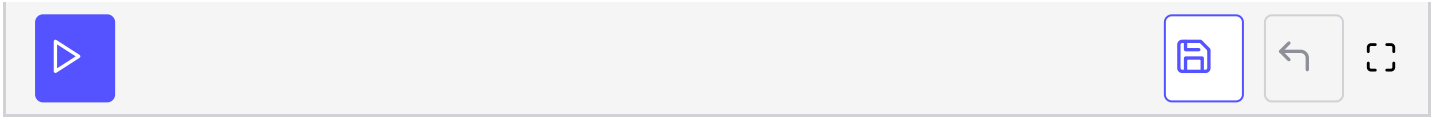
The concept and functionalities are the same as a 1-D array with a little increased complexity due to another added dimension. The elements can be accessed and changed in the same way but now the indexes need to be decided on the basis of the row and column numbers. Let's look at an example:

```
import numpy as np
# Declare a 2-D array
arr2d = np.array([
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
])
print("The Array")
print(arr2d)
```

```
print("\nElement at row 0 and column 1")
print(arr2d[0][1])


print("\nElements in a range of last two rows & columns")
print(arr2d[1:3, 1:3])

arr2d[1:3, 1:3] = 20
print("\nNew 2D array after changing last two rows & columns values")
print(arr2d)
```



The `:` inside the `[]` works the same way as in the 1-D array, but now we can give both the row and column numbers. This `arr2d[1:3, 1:3]` command before the `,` selects the rows starting from `1` to `3` (exclusive) and after the `,` selects the columns starting from `1` to `3` (exclusive).

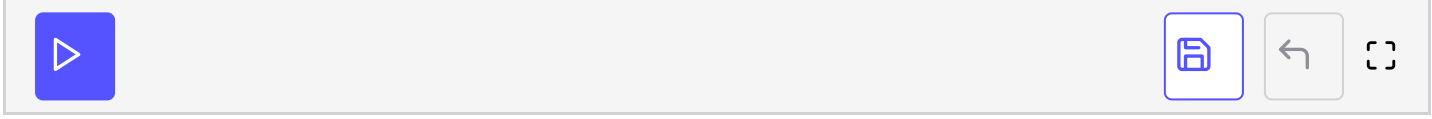
Slicing 2-D array

Slicing a 2-D array is a little more complex task than slicing a 1-D array. Let's look at an example and try understanding it from that.

```
import numpy as np
# Declare a 2-D array
arr2d = np.array([
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
])
print("The Array")
print(arr2d)

print("\nThe new sliced array")
print(arr2d[1:3, 0:2])

print("\nThe new sliced column")
print(arr2d[:, 1:2])
```



In the above examples, the lower-left half and the middle column are sliced from the 2-D array. Inside the `[]`, the range on the left of `,` decides the rows to be included, and the range on the right of `,` decides the columns to be included.

In the first example, the last two rows and the first two columns are selected. The second example selects all the rows, and only the center column to get the values of the middle column.

The memory problem

Whenever a `NumPy` array is sliced into another variable, it does not create another array but only a reference to the original array is created. This means that all the changes made in the newly sliced array are also reflected in the original array.

```
import numpy as np

arr = np.arange(0,10,1)
print("The Array")
print(arr)

new_arr = arr[1:7]
print("\nThe new sliced array")
print(new_arr)

new_arr[:] = 20
print("\nThe Original array")
print(arr)
```

As seen from the above output, the changes made in the sliced array are reflected in the original as well. To create a copy of the array, the `copy()` function needs to be explicitly called, like in the below example.

```
import numpy as np

arr = np.arange(0,10,1)
print("The Array")
print(arr)

new_arr = arr[1:7].copy()
print("\nThe new sliced array")
print(new_arr)

new_arr[:] = 20
print("\nThe Original array")
print(arr)
```

As seen in the output, the changes made to the `new_arr` array are not reflected in the original array.

In the next lesson, transposing a `NumPy` array is explained.