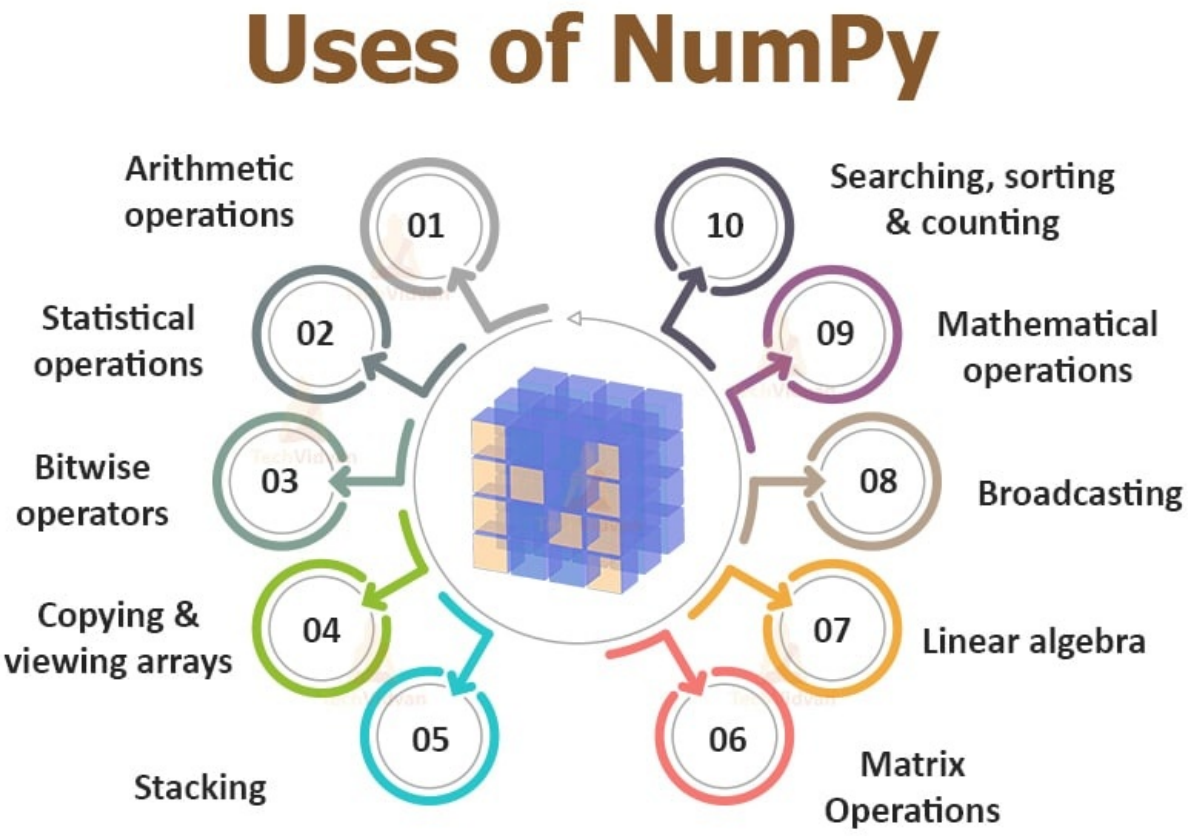
M.A.N.U

2/23/2023

Mathematics Association of Nairobi University
isaak@students.uonbi.ac.ke
https://github.com/Isaakkamau

Introduction to NumPy



`NumPy` stands for Numerical Python and it's a fundamental package for scientific computing in Python. NumPy provides Python with an extensive math library capable of performing numerical computations effectively and efficiently. These lessons are intended as a basic overview of NumPy and introduce some of its most important features.

Download NumPy

NumPy is included with ~Anaconda'. If you don't already have Anaconda installed on your computer, please refer to the Anaconda section to get clear instructions on how to install Anaconda on your PC or Mac.

Why Use NumPy

You may be wondering why people use NumPy - after all, Python can handle lists.

Benefits of using NumPy

Even though Python lists are great on their own, NumPy has a number of key features that give it great advantages over Python lists. Below are a few convincingly strong features:

1. One such feature is `speed`. When performing operations on large arrays NumPy can often perform several orders of magnitude faster than Python lists. This speed comes from the nature of NumPy arrays being memory-efficient and from optimized algorithms used by NumPy for doing arithmetic, statistical, and linear algebra operations.

2. Another great feature of NumPy is that it has `multidimensional array data structures` that can represent vectors and matrices.

3. Another great advantage of NumPy over Python lists is that NumPy has a large number of `optimized built-in mathematical functions`. These functions allow you to do a variety of complex mathematical computations very fast and with very little code (avoiding the use of complicated loops) making your programs more readable and easier to understand.

These are just some of the key features that have made NumPy an essential package for scientific computing in Python. In fact, NumPy has become so popular that a lot of Python packages, such as Pandas, are built on top of NumPy.

In [8]:
```
# Why use NumPy?
import time
import numpy as np
x = np.random.random(100000000)

# Case 1
start = time.time()
sum(x) / len(x)
print('Time taken without numpy is: ',time.time() - start)

# Case 2
start = time.time()
np.mean(x)
print('Time taken by numpy is: ',time.time() - start)
```

```
Time taken without numpy is:  20.245998859405518
Time taken by numpy is:  0.3410041332244873
```

In [10]:
```
print("Sum of 'X' is: ", sum(x))
```

```
Sum of 'X' is:  50002537.17246596
```

In [11]:
```
print("\nThe list 'X' has ", len(x), "Items")
```

```
The list 'X' has  100000000 Items
```

Creating NumPy ndarrays

At the core of `NumPy` is the `ndarray`, where nd stands for `n-dimensional`. An ndarray is a multidimensional array of elements all of the same type. In other words, an ndarray is a grid that can take on many shapes and can hold either numbers or strings.

But before we can dive in and start using NumPy to create ndarrays we need to import it into Python. We can import packages into Python using the import command and it has become a convention to import NumPy as `np`. Therefore, you can import NumPy by typing the following command in your Jupyter notebook:

In [12]:
```
import numpy as np
```

There are several ways to create ndarrays in NumPy. In the following lessons we will see two ways to create ndarrays:

1. Using regular Python lists

2. Using built-in NumPy functions

In this section, we will create ndarrays by providing Python lists to the NumPy `np.array()` function. This can create some confusion for beginners, but it is important to remember that `np.array()` is NOT a class, it is just a function that returns an ndarray. We should note that for the purposes of clarity, the examples throughout these lessons will use small and simple ndarrays. Let's start by creating *1-Dimensional (1D) ndarrays*.

In [13]:
```
# We import NumPy into Python
import numpy as np

# We create a 1D ndarray that contains only integers
x = np.array([1, 2, 3, 4, 5])

# Let's print the ndarray we just created using the print() command
print('x = ', x)
```

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

Rank of an Array (numpy.ndarray.ndim)

Syntax:

```
ndarray.ndim
```

It returns the number of array dimensions.

Let's pause for a second to introduce some useful terminology. We refer to 1D arrays as rank 1 arrays. In general N-Dimensional arrays have rank N. Therefore, we refer to a 2D array as a rank 2 array.

In [16]:
```
# 1-D array
x = np.array([1, 2, 3])
x.ndim
```

Out[16]:
```
1
```

In [18]:
```
# 2-D array
Y = np.array([[1,2,3],[4,5,6],[7,8,9], [10,11,12]])

print('N-Dimensional arrays have rank: ', Y.ndim)
print(Y)
```

```
N-Dimensional arrays have rank:  2
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

In [19]:
```
# Here the`zeros()` is an inbuilt function that you'll study on the next page.
# The tuple (2, 3, 4( passed as an argument represents the shape of the ndarray
y = np.zeros((2, 3, 4))
print(y)
print('N-Dimensional arrays have rank: ', y.ndim)
print('y has dimensions: ', y.shape, 'telling us that y is of rank 3')
```

```
[[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
N-Dimensional arrays have rank:  3
y has dimensions:  (2, 3, 4) telling us that y is of rank 3
```

NumPy Data Types

As mentioned earlier, ndarrays can also hold strings. Let's see how we can create a rank 1 ndarray of strings in the same manner as before, by providing the `np.array()` function a Python list of strings. **Example 1.b - Using 1-D Array of Strings**

In [20]:
```
# We create a rank 1 ndarray that only contains strings
x = np.array(['Hello', 'World'])

# We print information about x
print('x = ', x)
print('x has dimensions:', x.shape)
print('x is an object of type:', type(x))
print('The elements in x are of type:', x.dtype)
```

```
x =  ['Hello' 'World']
x has dimensions: (2,)
x is an object of type: <class 'numpy.ndarray'>
The elements in x are of type: <U5
```

As we can see the shape attribute tells us that x now has only 2 elements, and even though x now holds strings, the `type()` function tells us that x is still an ndarray as before. In this case however, the .dtype attribute tells us that the elements in x are stored in memory as `Unicode strings` of 5 characters.

It is important to remember that one big difference between `Python lists` and `ndarrays`, is that unlike Python lists, **all the elements of an ndarray must be of the same type**. So, while we can create Python lists with both integers and strings, we can't mix types in ndarrays. If you provide the `np.array()` function with a Python list that has both integers and strings, **NumPy will interpret all elements as strings**. We can see this in the next example:

Example 1.c - Using a 1-D Array of Mixed Datatype

In [25]:
```
# We create a rank 1 ndarray from a Python list that contains integers and strings
x = np.array([1, 2, 'World'])

# We print information about x
print('x = ', x)
print('x has dimensions:', x.shape)
print('x is an object of type:', type(x))
print('The elements in x are of type:', x.dtype)
```

```
x =  ['1' '2' 'World']
x has dimensions: (3,)
x is an object of type: <class 'numpy.ndarray'>
The elements in x are of type: <U11
```

We can see that even though the Python list had mixed data types, the elements in x are all of the same type, namely, Unicode strings. We won't be using ndarrays with strings for the remaining of this introduction to NumPy, but it's important to remember that ndarrays can hold strings as well.

Using a 1-D Array to Demonstrate Upcasting in Numeric datatype

Up till now, we have only created ndarrays with integers and strings. We saw that when we create an ndarray with only integers, NumPy will automatically assign the dtype int64 to its elements. Let's see what happens when we create ndarrays with floats and integers.
Example 1.d - Using a 1-D Array of Int and Float

In [26]:
```
# We create a rank 1 ndarray that contains integers
x = np.array([1,2,3])

# We create a rank 1 ndarray that contains floats
y = np.array([1.0,2.0,3.0])

# We create a rank 1 ndarray that contains integers and floats
z = np.array([1, 2.5, 4])

# We print the dtype of each ndarray
print('The elements in x are of type:', x.dtype)
print('The elements in y are of type:', y.dtype)
print('The elements in z are of type:', z.dtype)
```

```
The elements in x are of type: int32
The elements in y are of type: float64
The elements in z are of type: float64
```

We can see that when we create an ndarray with only floats, NumPy stores the elements in memory as 64-bit floating point numbers (float64). However, notice that when we create an ndarray with both floats and integers, as we did with the z ndarray above, NumPy assigns its elements a float64 dtype as well. This is called **upcasting**. Since all the elements of an ndarray must be of the same type, in this case NumPy upcasts the integers in z to floats in order to avoid losing precision in numerical computations.

Using a 1-D Array of Float, and specifying the dtype of each element

Even though NumPy automatically selects the dtype of the ndarray, NumPy also allows you to specify the particular dtype you want to assign to the elements of the ndarray. You can specify the dtype when you create the ndarray using the keyword dtype in the np.array() function. Let's see an example: **Example 1.e - Using a 1-D Array of Float, and specifying the datatype of each element as int64**

In [27]:
```
# We create a rank 1 ndarray of floats but set the dtype to int64
x = np.array([1.5, 2.2, 3.7, 4.0, 5.9], dtype = np.int64)

# We print the dtype x
print('x = ', x)
print('The elements in x are of type:', x.dtype)
```

```
x =  [1 2 3 4 5]
The elements in x are of type: int64
```

We can see that even though we created the ndarray with floats, by specifying the dtype to be int64, NumPy converted the floating point numbers into integers by removing their decimals. Specifying the data type of the ndarray can be useful in cases when you don't want NumPy to accidentally choose the wrong data type, or when you only need certain amount of precision in your calculations and you want to save memory.

numpy.ndarray.size and Creating a 2-D array

Another useful attribute is NumPy.size, which returns the number of elements in the array. Let us now look at how we can create a rank 2 ndarray from a nested Python list. **Example 2 - Using a 2-D Array (Rank #2 Array)**

In [28]:
```
# We create a rank 2 ndarray that only contains integers
Y = np.array([[1,2,3],[4,5,6],[7,8,9], [10,11,12]])

print('Y = \n', Y)

# We print information about Y
print('Y has dimensions:', Y.shape)
print('Y has a total of', Y.size, 'elements')
print('Y is an object of type:', type(Y))
print('The elements in Y are of type:', Y.dtype)
```

```
Y =
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
Y has dimensions: (4, 3)
Y has a total of 12 elements
Y is an object of type: <class 'numpy.ndarray'>
The elements in Y are of type: int32
```

Using Built-in Functions to Create ndarrays

One great time-saving feature of NumPy is its ability to create ndarrays using built-in functions. These functions allow us to create certain kinds of ndarrays with just one line of code. Below we will see a few of the most useful built-in functions for creating ndarrays that you will come across when doing AI programming.

Let's start by creating an ndarray with a specified shape that is full of zeros. We can do this by using the `np.zeros()` function. The function `np.zeros(shape)` creates an ndarray full of zeros with the given shape. So, for example, if you wanted to create a rank 2 array with 3 rows and 4 columns, you will pass the shape to the function in the form of `(rows, columns)`, as in the example below:
Example 1. Create a Numpy array of zeros with a desired shape

In [29]:
```
# We create a 3 x 4 ndarray full of zeros.
X = np.zeros((3,4))

# We print X
print()
print('X = \n', X)
print()

# We print information about X
print('X has dimensions:', X.shape)
print('X is an object of type:', type(X))
print('The elements in X are of type:', X.dtype)
```

```
X =
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

X has dimensions: (3, 4)
X is an object of type: <class 'numpy.ndarray'>
The elements in X are of type: float64
```

As we can see, the np.zeros() function creates by default an array with dtype float64. If desired, the data type can be changed by using the keyword dtype.

Similarly, we can create an ndarray with a specified shape that is full of ones. We can do this by using the np.ones() function. Just like the np.zeros() function, the np.ones() function takes as an argument the shape of the ndarray you want to make. Let's see an example:

Example 2. Create a Numpy array of ones

In [30]:
```
# We create a 3 x 2 ndarray full of ones.
X = np.ones((3,2))

# We print X
print()
print('X = \n', X)
print()

# We print information about X
print('X has dimensions:', X.shape)
print('X is an object of type:', type(X))
print('The elements in X are of type:', X.dtype)
```

```
X =
 [[1. 1.]
 [1. 1.]
 [1. 1.]]

X has dimensions: (3, 2)
X is an object of type: <class 'numpy.ndarray'>
The elements in X are of type: float64
```

As we can see, thenp.ones() function also creates by default an array with dtype float64. If desired, the data type can be changed by using the keyword dtype.

We can also create an ndarray with a specified shape that is full of any number we want. We can do this by using the np.full() function. The np.full(shape, constant value) function takes two arguments. The first argument is the shape of the ndarray you want to make and the second is the constant value you want to populate the array with. Let's see an example:

Example 3. Create a Numpy array of constants

In [31]:
```
# We create a 2 x 3 ndarray full of fives.
X = np.full((2,3), 5)

# We print X
print()
print('X = \n', X)
print()

# We print information about X
```

```
print('X has dimensions:', X.shape)
print('X is an object of type:', type(X))
print('The elements in X are of type:', X.dtype)
```

```
X =
 [[5 5 5]
 [5 5 5]]

X has dimensions: (2, 3)
X is an object of type: <class 'numpy.ndarray'>
The elements in X are of type: int32
```

The np.full() function creates by default an array with the same data type as the constant value used to fill in the array. If desired, the data type can be changed by using the keyword dtype.

A fundamental array in Linear Algebra is the Identity Matrix. An Identity matrix is a square matrix that has only 1s in its main diagonal and zeros everywhere else. The function `np.eye(N)` creates a square `N x N` ndarray corresponding to the Identity matrix. Since all Identity Matrices are square, the `np.eye()` function only takes a single integer as an argument. Let's see an example:

Example 4 a. Create a Numpy array of an Identity matrix

In [32]:
```
# We create a 5 x 5 Identity matrix.
X = np.eye(5)

# We print X
print()
print('X = \n', X)
print()

# We print information about X
print('X has dimensions:', X.shape)
print('X is an object of type:', type(X))
print('The elements in X are of type:', X.dtype)
```

```
X =
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]

X has dimensions: (5, 5)
X is an object of type: <class 'numpy.ndarray'>
The elements in X are of type: float64
```

As we can see, the `np.eye()` function also creates by default an array with dtype float64. If desired, the data type can be changed by using the keyword dtype. You will learn all about Identity Matrices and their use in the Linear Algebra section of this course. We can also create diagonal matrices by using the `np.diag()` function. A diagonal matrix is a square matrix that only has values in its main diagonal. The `np.diag()` function creates an ndarray corresponding to a diagonal matrix , as shown in the example below:

Example 4 b. Create a Numpy array of constants

In [33]:
```
# Create a 4 x 4 diagonal matrix that contains the numbers 10,20,30, and 50
# on its main diagonal
X = np.diag([10,20,30,50])

# We print X
print()
print('X = \n', X)
print()
```

```
X =
 [[10  0  0  0]
 [ 0 20  0  0]
 [ 0  0 30  0]
 [ 0  0  0 50]]
```

numpy.arange

Syntax:

```
numpy.arange([start, ]stop, [step, ]dtype=None)
```

It returns evenly spaced values within a given interval.

NumPy also allows you to create ndarrays that have evenly spaced values within a given interval. NumPy's `np.arange()` function is very versatile and can be used with either one, two, or three arguments. Below we will see examples of each case and how they are used to create different kinds of ndarrays.

Let's start by using `np.arange()` with only one argument. When used with only one argument, `np.arange(N)` will create a rank 1

ndarray with consecutive integers between `0` and `N - 1`. Therefore, notice that if I want an array to have integers between 0 and 9, I have to use N = 10, NOT N = 9, as in the example below: **Example 5. Create a Numpy array of evenly spaced values in a given range, using arange(stop_val)**

In [34]:
```
# We create a rank 1 ndarray that has sequential integers from 0 to 9
x = np.arange(10)

# We print the ndarray
print()
print('x = ', x)
print()

# We print information about the ndarray
print('x has dimensions:', x.shape)
print('x is an object of type:', type(x))
print('The elements in x are of type:', x.dtype)
```

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

x has dimensions: (10,)
x is an object of type: <class 'numpy.ndarray'>
The elements in x are of type: int32
```

When used with two arguments, `np.arange(start,stop)` will create a rank 1 ndarray with evenly spaced values within the half-open interval `[start, stop)`. This means the evenly spaced numbers will include `start` but exclude `stop`. Let's see an example

Example 6. Create a Numpy array using arange(start_val, stop_val)

In [35]:
```
# We create a rank 1 ndarray that has sequential integers from 4 to 9.
x = np.arange(4,10)

# We print the ndarray
print()
print('x = ', x)
print()

# We print information about the ndarray
print('x has dimensions:', x.shape)
print('x is an object of type:', type(x))
print('The elements in x are of type:', x.dtype)
```

```
x =  [4 5 6 7 8 9]

x has dimensions: (6,)
x is an object of type: <class 'numpy.ndarray'>
The elements in x are of type: int32
```

As we can see, the function np.arange(4,10) generates a sequence of integers with 4 inclusive and 10 exclusive.

Finally, when used with three arguments, `np.arange(start,stop,step)` will create a rank 1 ndarray with evenly spaced values within the half-open interval `[start, stop)` with `step` being the distance between two adjacent values. Let's see an example:

Example 7. Create a Numpy array using arange(start_val, stop_val, step_size)

In [36]:
```
# We create a rank 1 ndarray that has evenly spaced integers from 1 to 13 in steps of 3.
x = np.arange(1,14,3)

# We print the ndarray
print()
print('x = ', x)
print()

# We print information about the ndarray
print('x has dimensions:', x.shape)
print('x is an object of type:', type(x))
print('The elements in x are of type:', x.dtype)
```

```
x =  [ 1  4  7 10 13]

x has dimensions: (5,)
x is an object of type: <class 'numpy.ndarray'>
The elements in x are of type: int32
```

numpy.reshape - This is a Function.

Syntax:

```
numpy.reshape(array, newshape, order='C')[source]
```

It gives a new shape to an array without changing its data.

Example 10. Create a Numpy array by feeding the output of arange() function as an argument to the reshape() function.

In [37]:
```
# We create a rank 1 ndarray with sequential integers from 0 to 19
x = np.arange(20)

# We print x
print()
print('Original x = ', x)
print()

# We reshape x into a 4 x 5 ndarray
x = np.reshape(x, (4,5))

# We print the reshaped x
print()
print('Reshaped x = \n', x)
print()

# We print information about the reshaped x
print('x has dimensions:', x.shape)
print('x is an object of type:', type(x))
print('The elements in x are of type:', x.dtype)
```

```
Original x =  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]


Reshaped x =
 [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]

x has dimensions: (4, 5)
x is an object of type: <class 'numpy.ndarray'>
The elements in x are of type: int32
```

NumPy also allows us to create ndarrays with random integers within a particular interval. The function `np.random.randint(start, stop, size = shape)` creates an ndarray of the given `shape` with random integers in the half-open interval `[start, stop)`. Let's see an example:

Example 14. Create a Numpy array using the numpy.random.randint() function.

In [38]:
```
# We create a 3 x 2 ndarray with random integers in the half-open interval [4, 15).
X = np.random.randint(4,15,size=(3,2))

# We print X
print()
print('X = \n', X)
print()

# We print information about X
print('X has dimensions:', X.shape)
print('X is an object of type:', type(X))
print('The elements in X are of type:', X.dtype)
```

```
X =
 [[ 8  4]
 [ 7  7]
 [ 7 14]]

X has dimensions: (3, 2)
X is an object of type: <class 'numpy.ndarray'>
The elements in X are of type: int32
```

Accessing, Deleting, and Inserting Elements Into ndarrays

Now that you know how to create a variety of ndarrays, we will now see how NumPy allows us to effectively manipulate the data within the ndarrays. NumPy ndarrays are mutable, meaning that the elements in ndarrays can be changed after the ndarray has been created. NumPy ndarrays can also be sliced, which means that ndarrays can be split in many different ways. This allows us, for example, to retrieve any subset of the ndarray that we want. Often in Machine Learning you will use slicing to separate data, as for example when dividing a data set into training, cross validation, and testing sets.

We will start by looking at how the elements of an ndarray can be accessed or modified by indexing. Elements can be accessed using indices inside square brackets, [ ]. NumPy allows you to use both positive and negative indices to access elements in the ndarray. Positive indices are used to access elements from the beginning of the array, while negative indices are used to access elements from the end of the array. Let's see how we can access elements in rank 1 ndarrays:

Example 1. Access individual elements of 1-D array

In [39]:
```
# We create a rank 1 ndarray that contains integers from 1 to 5
x = np.array([1, 2, 3, 4, 5])

# We print x
```

```
print()
print('x = ', x)
print()

# Let's access some elements with positive indices
print('This is First Element in x:', x[0])
print('This is Second Element in x:', x[1])
print('This is Fifth (Last) Element in x:', x[4])
print()

# Let's access the same elements with negative indices
print('This is First Element in x:', x[-5])
print('This is Second Element in x:', x[-4])
print('This is Fifth (Last) Element in x:', x[-1])
```

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

This is First Element in x: 1
This is Second Element in x: 2
This is Fifth (Last) Element in x: 5

This is First Element in x: 1
This is Second Element in x: 2
This is Fifth (Last) Element in x: 5
```

Now let's see how we can change the elements in rank 1 ndarrays. We do this by accessing the element we want to change and then using the `=` sign to assign the new value:

Example 2. Modify an element of 1-D array

In [40]:
```
# We create a rank 1 ndarray that contains integers from 1 to 5
x = np.array([1, 2, 3, 4, 5])

# We print the original x
print()
print('Original:\n x = ', x)
print()

# We change the fourth element in x from 4 to 20
x[3] = 20

# We print x after it was modified
print('Modified:\n x = ', x)
```

```
Original:
 x =  [1 2 3 4 5]

Modified:
 x =  [ 1  2  3 20  5]
```

Similarly, we can also access and modify specific elements of rank 2 ndarrays. To access elements in rank 2 ndarrays we need to provide 2 indices in the form `[row, column]`. Let's see some examples.

Example 3. Access individual elements of 2-D array

In [41]:
```
# We create a 3 x 3 rank 2 ndarray that contains integers from 1 to 9
X = np.array([[1,2,3],[4,5,6],[7,8,9]])

# We print X
print()
print('X = \n', X)
print()

# Let's access some elements in X
print('This is (0,0) Element in X:', X[0,0])
print('This is (0,1) Element in X:', X[0,1])
print('This is (2,2) Element in X:', X[2,2])
```

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

This is (0,0) Element in X: 1
This is (0,1) Element in X: 2
This is (2,2) Element in X: 9
```

Example 4. Modify an element of 2-D array

In [42]:
```
# We create a 3 x 3 rank 2 ndarray that contains integers from 1 to 9
X = np.array([[1,2,3],[4,5,6],[7,8,9]])

# We print the original x
print()
print('Original:\n X = \n', X)
print()
```

```
# We change the (0,0) element in X from 1 to 20
X[0,0] = 20

# We print X after it was modified
print('Modified:\n X = \n', X)
```

```
Original:
 X =
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Modified:
 X =
 [[20  2  3]
 [ 4  5  6]
 [ 7  8  9]]
```

numpy.append

Syntax:

```
numpy.append(array, values, axis=None)
```

It appends values to the end of an array. Refer here for more details about additional arguments.

Now, let's see how we can append values to ndarrays. We can append values to ndarrays using the `np.append(ndarray, elements, axis)` function. This function appends the given list of elements to ndarray along the specified axis. Let's see some examples: **Example 6. Append elements**

In [43]:
```
# We create a rank 1 ndarray
x = np.array([1, 2, 3, 4, 5])

# We create a rank 2 ndarray
Y = np.array([[1,2,3],[4,5,6]])

# We print x
print()
print('Original x = ', x)

# We append the integer 6 to x
x = np.append(x, 6)

# We print x
print()
print('x = ', x)

# We append the integer 7 and 8 to x
x = np.append(x, [7,8])

# We print x
print()
print('x = ', x)

# We print Y
print()
print('Original Y = \n', Y)

# We append a new row containing 7,8,9 to y
v = np.append(Y, [[7,8,9]], axis=0)

# We append a new column containing 9 and 10 to y
q = np.append(Y,[[9],[10]], axis=1)

# We print v
print()
print('v = \n', v)

# We print q
print()
print('q = \n', q)
```

```
Original x =  [1 2 3 4 5]

x =  [1 2 3 4 5 6]

x =  [1 2 3 4 5 6 7 8]

Original Y =
 [[1 2 3]
 [4 5 6]]

v =
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

q =
 [[ 1  2  3  9]
 [ 4  5  6 10]]
```

Slicing ndarrays

1. ndarray[start:end]
2. ndarray[start:]
3. ndarray[:end]

The first method is used to select elements between the `start` and `end` indices. The second method is used to select all elements from the start index till the last index. The third method is used to select all elements from the first index till the end index. We should note that in methods one and three, the end index is excluded. We should also note that since ndarrays can be multidimensional, when doing slicing you usually have to specify a slice for each dimension of the array.

We will now see some examples of how to use the above methods to select different subsets of a rank 2 ndarray.

Example 1. Slicing in a 2-D ndarray

In [44]:
```
# We create a 4 x 5 ndarray that contains integers from 0 to 19
X = np.arange(20).reshape(4, 5)

# We print X
print()
print('X = \n', X)
print()

# We select all the elements that are in the 2nd through 4th rows and in the 3rd to 5th columns
Z = X[1:4,2:5]

# We print Z
print('Z = \n', Z)

# We can select the same elements as above using method 2
W = X[1:,2:5]

# We print W
print()
print('W = \n', W)

# We select all the elements that are in the 1st through 3rd rows and in the 3rd to 4th columns
Y = X[:3,2:5]

# We print Y
print()
print('Y = \n', Y)

# We select all the elements in the 3rd row
v = X[2,:]

# We print v
print()
print('v = ', v)

# We select all the elements in the 3rd column
q = X[:,2]

# We print q
print()
print('q = ', q)

# We select all the elements in the 3rd column but return a rank 2 ndarray
R = X[:,2:3]

# We print R
print()
print('R = \n', R)
```

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

Z =
 [[ 7  8  9]
 [12 13 14]
 [17 18 19]]

W =
 [[ 7  8  9]
 [12 13 14]
 [17 18 19]]

Y =
 [[ 2  3  4]
 [ 7  8  9]
 [12 13 14]]

v =  [10 11 12 13 14]

q =  [ 2  7 12 17]

R =
 [[ 2]
 [ 7]
 [12]
 [17]]
```

In [ ]: