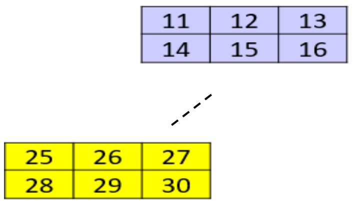
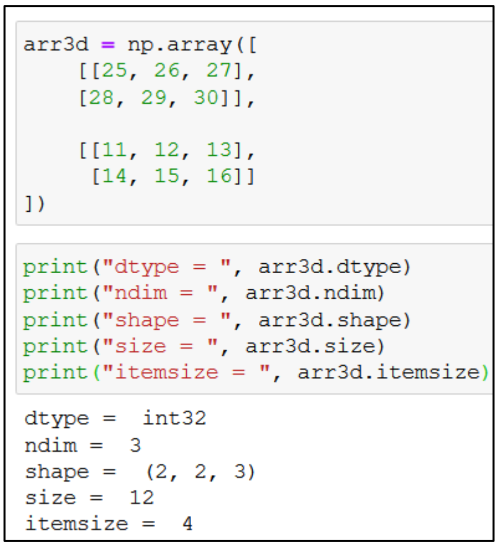
Objective: This worksheet provides different examples of accessing NumPy array elements using the indexing in different ways. It also explores few math functions which are available in the NumPy package.



As shown in the picture above, let us say there is 3-dimesniosnal array of size 2x2x3 with the array elements filled in as shown.

- The array is created in NumPy and its different attributes are verified.



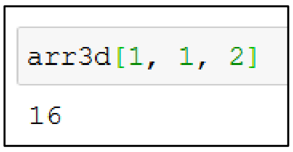
```
arr3d = np.array([
    [[25, 26, 27],
    [28, 29, 30]],

    [[11, 12, 13],
     [14, 15, 16]]
])
```

```
print("dtype = ", arr3d.dtype)
print("ndim = ", arr3d.ndim)
print("shape = ", arr3d.shape)
print("size = ", arr3d.size)
print("itemsize = ", arr3d.itemsize)
```

```
dtype =  int32
ndim =  3
shape =  (2, 2, 3)
size =  12
itemsize =  4
```

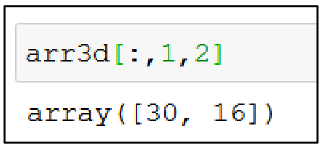
- A specific element is accessed specifying the indices in the square brackets as it is done in lists.



```
arr3d[1, 1, 2]
```

```
16
```

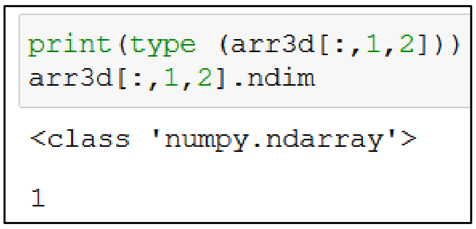
- Colon (:) can be specified to specify all.



```
arr3d[:,1,2]
```

```
array([30, 16])
```

- The return type in this case is also ***ndarray*** and the returned array is 1-dimensional.

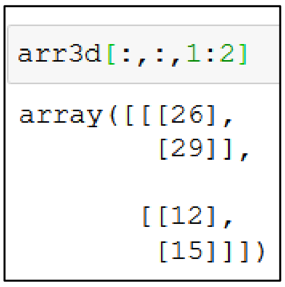


```
print(type (arr3d[:,1,2]))
arr3d[:,1,2].ndim
```

```
<class 'numpy.ndarray'>

1
```

- While accessing the array elements, the range of dimension(s) can also be specified. Remember when specifying the range in x:y form, x is inclusive and y is exclusive.

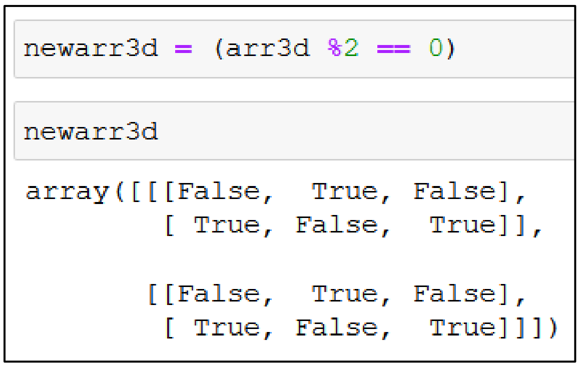


```
arr3d[:,:,1:2]
```

```
array([[[26],
        [29]],

       [[12],
        [15]]])
```

- Notice that in the above case, returned array is still ***ndarray*** but the returned array is 3-dimensional.
- NumPy arrays supports several additional methods by which indexing can be used to obtain interesting outcomes. Let us say, we need to create a separate array based on the above array. This new arrays will have boolean value True if arr3d has an even element, otherwise False to the corresponding locations. It can be achieved in the following way. Notice that the retuned array is 3-dimensional.



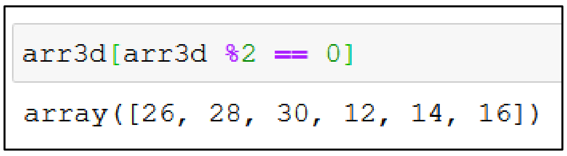
```
newarr3d = (arr3d %2 == 0)
```

```
newarr3d
```

```
array([[[False,  True, False],
        [ True, False,  True]],

       [[False,  True, False],
        [ True, False,  True]]])
```

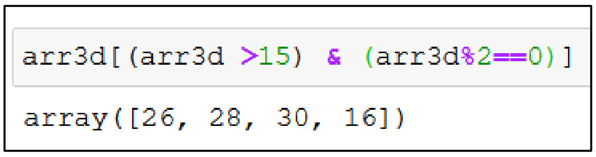
- Notice the expression ***arr3d %2 == 0***. It is not provided as an index, rather arr3d is used as an operand with one operator %. If the expression is provided as an index it will return only the even elements as shown below:



```
arr3d[arr3d %2 == 0]
```

```
array([26, 28, 30, 12, 14, 16])
```

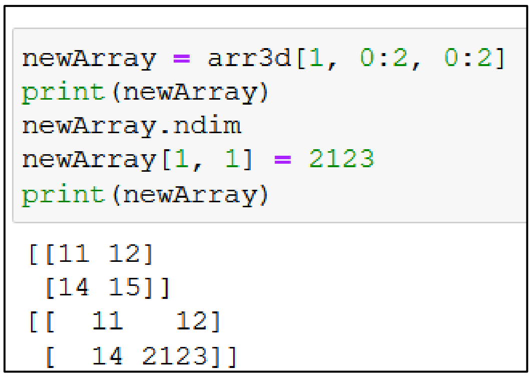
- Let us say we want to create a new array from the elements of arr3d which are greater than 15 and are even. It can be achieved in the following way. Notice that the retuned array is 1-dimensional.



```
arr3d[(arr3d >15) & (arr3d%2==0)]
```

```
array([26, 28, 30, 16])
```

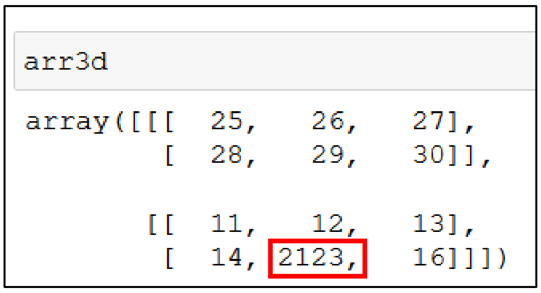
- Let us say, some part of the arr3d is accessed and one of its elements is changed to 2123.



```
newArray = arr3d[1, 0:2, 0:2]
print(newArray)
newArray.ndim
newArray[1, 1] = 2123
print(newArray)
```

```
[[11 12]
 [14 15]]
[[  11   12]
 [  14 2123]]
```

- Now when the original array arr3d is accessed, the change is reflecting in it also, though the change was not made in it explicitly. This is a concept of ***shallow copying*.**

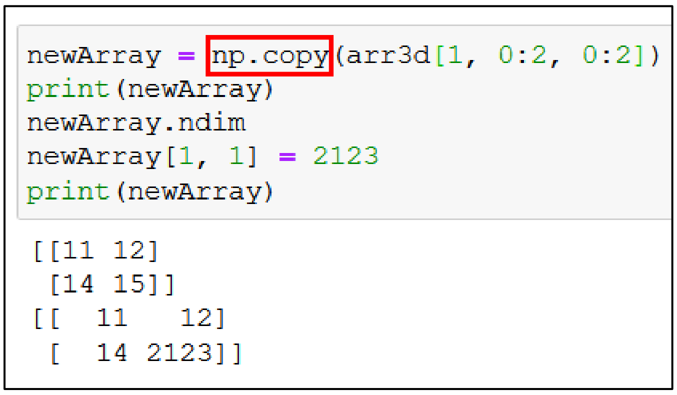


```
arr3d
```

```
array([[[  25,   26,   27],
        [  28,   29,   30]],

       [[  11,   12,   13],
        [  14, 2123,   16]]])
```

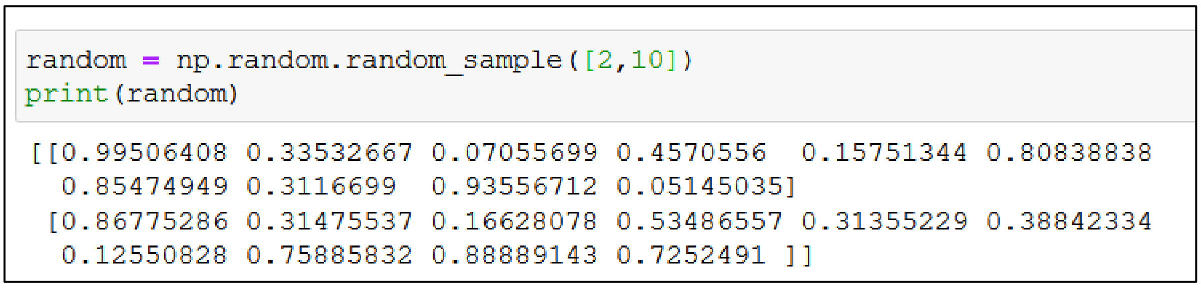
- To avoid this, the ***copy ()*** function of NumPy can be used as shown below for deep copying. Deep copying creates another object, in place of referring to the same object. Verify it yourself.



```
newArray = np.copy(arr3d[1, 0:2, 0:2])
print(newArray)
newArray.ndim
newArray[1, 1] = 2123
print(newArray)
```

```
[[11 12]
 [14 15]]
[[  11   12]
 [  14 2123]]
```

- NumPy provides several variations of generating random numbers which are useful in Data Science and many other fields like Cryptography. For example, there is a requirement to have 2x10 random numbers in the range of [0, 1.0). The function ***random_sample()*** can be used for the purpose.

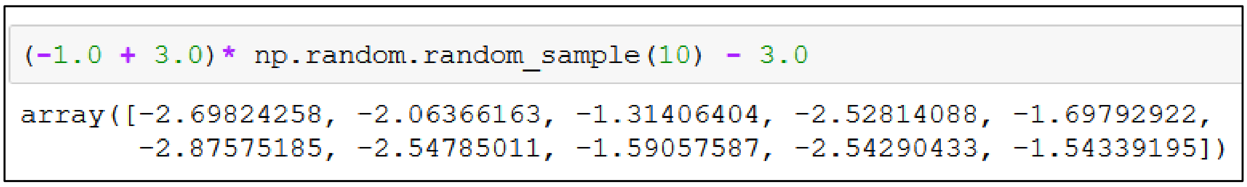


```
random = np.random.random_sample([2,10])
print(random)
```

```
[[0.99506408 0.33532667 0.07055699 0.4570556  0.15751344 0.80838838
  0.85474949 0.3116699  0.93556712 0.05145035]
 [0.86775286 0.31475537 0.16628078 0.53486557 0.31355229 0.38842334
  0.12550828 0.75885832 0.88889143 0.7252491 ]]
```

- If the uniform distribution is to be changed to [a, b) where, b > a in place of [0, 1.0), the function can be used in the following form: ***(b-a) * random_sample() + a***

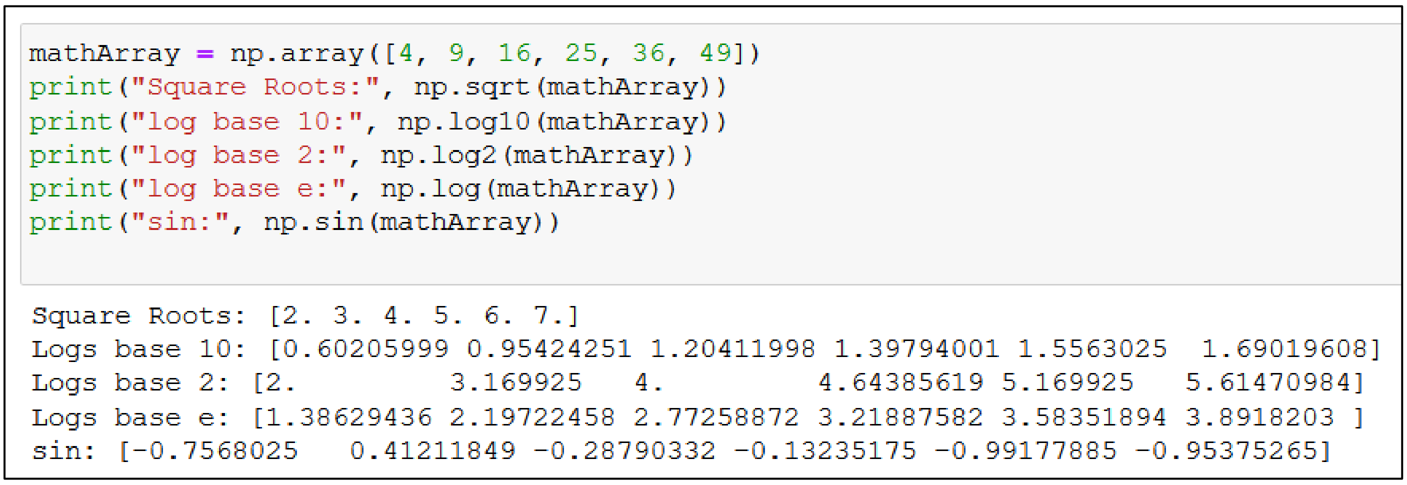
- For example, the code below, generates the random numbers in the range of [-3.0, -1.0)



```
(-1.0 + 3.0)* np.random.random_sample(10) - 3.0
```

```
array([-2.69824258, -2.06366163, -1.31406404, -2.52814088, -1.69792922,
       -2.87575185, -2.54785011, -1.59057587, -2.54290433, -1.54339195])
```

- There are several math functions which are available and can be applied to array elements in NumPy. The interesting point here is to observe that the functions are applied to the whole array in one shot. Few examples are shown below:



```
mathArray = np.array([4, 9, 16, 25, 36, 49])
print("Square Roots:", np.sqrt(mathArray))
print("log base 10:", np.log10(mathArray))
print("log base 2:", np.log2(mathArray))
print("log base e:", np.log(mathArray))
print("sin:", np.sin(mathArray))
```

```
Square Roots: [2. 3. 4. 5. 6. 7.]
Logs base 10: [0.60205999 0.95424251 1.20411998 1.39794001 1.5563025  1.69019608]
Logs base 2: [2.         3.169925   4.         4.64385619 5.169925   5.61470984]
Logs base e: [1.38629436 2.19722458 2.77258872 3.21887582 3.58351894 3.8918203 ]
sin: [-0.7568025   0.41211849 -0.28790332 -0.13235175 -0.99177885 -0.95375265]
```

Exercise:

The figure shows a square of 1 cm x 1 cm and a quadrant of a circle with radius 1 cm embedded in the square. Assume bottom left corner as the origin. Generate several random points (x, y coordinates) in the range of the figure and calculate the Euclidean distance of a point from the origin. Approximately if the distance is more than 1.0, the point does not belong to the circle quadrant. Find out the percentage of the points that belong to the circle quadrant and to the leftover square portion. Does it approximately match with the individual areas?