Arrays

Arrays

Arrays are one of NumPy's most important objects.

An array is a sequence of homogeneous data (each element must be the same data type). NumPy arrays use NumPy specific data types which are listed here.

Though we shall see that arrays can be indexed and sliced similarly to strings, tuples and lists, they behave differently under operations.

Arrays can have any number of dimensions. In this section we will only consider the 1 dimensional case.

Creating Arrays

Arrays can be created using the `np.array()` function with a list, tuple or another array as the argument:

[3]:
```
#Array of integers
np.array([1, 2, 3, 4])
```

[3]:
```
array([1, 2, 3, 4])
```

[4]:
```
#Array pf strings
np.array(('a', 'b', 'c'))
```

[4]:
```
array(['a', 'b', 'c'], dtype='<U1')
```

Remember that arrays are homogeneous:

[7]:
```
#Trying to create an array with different types
np.array([1, 2.3, 'x'])
```

[7]:
```
array(['1', '2.3', 'x'], dtype='<U32')
```

Indexing and Slicing

As said before, arrays can be indexed and sliced similarly to lists and strings

[15]:
```
letters = np.array(['a', 'b', 'c', 'd', 'e'])
print('Letters:', letters)
print('First character:', letters[0])
print('Second character:', letters[1])
print('Last character:', letters[-1])
print('Every second character:', letters[::2])
```

```
Letters: ['a' 'b' 'c' 'd' 'e']
First character: a
Second character: b
Last character: e
Every second character: ['a' 'c' 'e']
```

Mutable But Tricky To Resize

Similarly to lists, arrays are mutable (you can change the array after initializing it). For example, you can change an element of an array:

[9]:
```
arr = np.array((1, 2, 3 ,4))
print('Array:', arr)

print('')
print('Changing element 2')
print('')

arr[2] = 7
print('Array:', arr)
```

```
Array: [1 2 3 4]

Changing element 2

Array: [1 2 7 4]
```

However, unlike lists, it's not easy or efficient to alter the size of an array. It is still possible to resize (with `np.resize()`) and to concatenate (with `np.concatenate()`) arrays, but they don't have certain handy functions for lists like `.append()` and `.insert()`.

In general you should only create an array once you know how big it needs to be. If you need to add elements to an array, consider starting with a list and converting that to an array when you need the array properties.

Iterating Through Arrays

Like strings, tuples and lists, arrays are iterable:

[16]:
```
arr = np.array([1, 2, 3, 4])

for a in arr:
```

```
    print(a)
```

```
1
2
3
4
```

Vectorized Operations

One of the most useful properties of NumPy arrays is their vectorized operations. That is arithmetic operations between an array and array, and an array and scalar are performed element by element.

For example consider the scalar operations:

[10]:
```
2*np.array([1, 2, 3, 4])
```

[10]:
```
array([2, 4, 6, 8])
```

[12]:
```
np.array([1, 4, 5]) + 1
```

[12]:
```
array([2, 5, 6])
```

Array on array operations are also performed element by element:

[13]:
```
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([2, 4, 6, 8])

print(arr2, '-', arr1, 'is', arr2 - arr1)
print(arr2, '/', arr1, 'is', arr2/arr1)
```

```
[2 4 6 8] - [1 2 3 4] is [1 2 3 4]
[2 4 6 8] / [1 2 3 4] is [2. 2. 2. 2.]
```

These vectorized operations are far more efficient than iterating through the arrays and operating on each element individually, i.e.

[14]:
```
#More efficient:
print(arr1, '+', arr2, 'is', arr1 + arr2)
```

```
[1 2 3 4] + [2 4 6 8] is [ 3  6  9 12]
```

[16]:
```
#Less efficient
arr3 = np.array(4*[0])

for  i in range(4):
    arr3[i] = arr1[i] + arr2[i]

print(arr1, '+', arr2, 'is', arr3)
```

```
[1 2 3 4] + [2 4 6 8] is [ 3  6  9 12]
```

Creating Structured Arrays

Often we would like to create a large array with a particular structure. We could create these arrays from lists using list comprehension, but NumPy provides some useful built in functions to use instead.

`np.arange()`

This function is analogous to the `range()` function. It produces a series of values where you can specify the starting value, stopping value and the step size.

The syntax is:

`np.arange(start, stop, step)`

Similar to the `range()` function, you can use 1, 2 or 3 arguments:

[19]:
```
#1 argument: stop
np.arange(5)
```

[19]:
```
array([0, 1, 2, 3, 4])
```

[21]:
```
#2 arguments: start, stop
np.arange(1, 5)
```

[21]:
```
array([1, 2, 3, 4])
```

[22]:
```
#3 arguments: start, stop, step
np.arange(1, 10 ,2)
```

[22]:
```
array([1, 3, 5, 7, 9])
```

Unlike the `range()` function. `np.arange()` also allows for floating point values:

[24]:
```
np.arange(2.3, 3, 0.1)
```

[24]:
```
array([2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3. ])
```

`np.linspace()`

This function creates a series of evenly spaced values between a stopping and starting value. The number of items in the array can also be specified.

The syntax:

`np.linspace(start, stop, number)`

If `number` is not specified an array of length 50 is created.

[7]:
```
np.linspace(0, 1, 10)
```

[7]:
```
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

np.zeroes()

This function creates a uniform array of zeros. It takes the shape of the array you want to generate as an argument.

`np.zeros(shape)`

For a one dimensional array **shape** is just the size of the array:

[11]:
```
np.zeros(5)
```

[11]:
```
array([0., 0., 0., 0., 0.])
```

np.zeros() can be useful if you wish to create an array with a particular size, but will only be filling in the values later.

np.ones()

np.ones() is similar to **np.zeros()**, except it generates a uniform array of ones.

[13]:
```
np.ones(7)
```

[13]:
```
array([1., 1., 1., 1., 1., 1., 1.])
```

Note that, if you want a uniform array of a different value, you can either add that value to an array of zeros or multiply that value with an array of ones.