# The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas are built around the NumPy array. This section will present several examples of using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays. These operations are essential building blocks for many analysis

Some categories of basic array manipulations discusses here are:

- *Attributes of arrays*: Determining the size, shape, memory consumption, and data types of arrays
- *Indexing of arrays*: Getting and setting the value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

## NumPy Array

NumPy arrays have the benefits of a smaller memory consumptions and better runtime behaviour. NumPy arrays also provide the convenience of integrated mathematical operations not availabe in lists.

### Exercise:

Add a value of one (1) to all elements within a `list` of numbers [1, 2, 3, 4] in Python.

In [2]:

```
# todo: Exercise
a = [1,2,3,4]
for num in range(len(a)):
  a[num] = a[num] + 1
print(a)

b = [1,2,3,4]
c = [x+1 for x in b]
print(b)

c = [1,2,3,4]
c = map(lambda x:x+1, c)
print(list(c))
```

```
[2, 3, 4, 5]
[1, 2, 3, 4]
[2, 3, 4, 5]
```

By using NumPy, you can add one (1) to an array of numbers by

In [3]:

```
import numpy as np
a = np.array([1,2,3,4])
a = a + 1
print(a)
```

```
[2 3 4 5]
```

## Create arrays

From the previous example, you can observe a NumPy array can be created as follow

In [4]:

```
import numpy as np
a = np.array([1,2,3,4])
```

3 random arrays (one-dimensional, two-dimensional, and three-dimensional array) are created as examples.

In [5]:

```
import numpy as np
np.random.seed(0)  # seed for reproducibility

a1 = np.random.randint(10, size=6)  # One-dimensional array
a2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
a3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array

print ("1-D array", "\n", a1)
print ("2-D array", "\n", a2)
print ("3-D array", "\n", a3)
```

```
1-D array
 [5 0 3 3 7 9]
2-D array
 [[3 5 2 4]
 [7 6 8 8]
 [1 6 7 7]]
3-D array
 [[[8 1 5 9 8]
  [9 4 3 0 3]
  [5 0 2 3 8]
  [1 3 3 3 7]]

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

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

**Do you know?**

NumPy's random number seed makes the random numbers predictable. Try alternating the seed value with another number in the previous code and observe the numbers generated.

## Attributes of array

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

In [6]:

```
print("a3 ndim: ", a3.ndim)
print("a3 shape:", a3.shape)
print("a3 size: ", a3.size)
```

```
a3 ndim:  3
a3 shape: (3, 4, 5)
a3 size:  60
```

Another useful attribute is the `dtype`, the data type of the array.

In [7]:

```
print("dtype:", a3.dtype)
```

```
dtype: int32
```

Other attributes include `itemsize`, which lists the size (in bytes) of each array element, and `nbytes`, which lists the total size (in bytes) of the array:

In [8]:

```
print("itemsize:", a3.itemsize, "bytes")
print("nbytes:", a3.nbytes, "bytes")
```

```
itemsize: 4 bytes
nbytes: 240 bytes
```

In general, we expect that `nbytes` is equal to `itemsize` times `size`.

### Creating unwritable NumPy arrays

In [9]:

```
array_immutable = np.arange(5)
print (array_immutable)
array_immutable.flags.writeable = False
array_immutable[0] = 1
#RuntimeError: Assignment destination is read-only and not writeable
```

```
[0 1 2 3 4]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-c42c9e495337> in <module>
      2 print (array_immutable)
      3 array_immutable.flags.writeable = False
----> 4 array_immutable[0] = 1
      5 #RuntimeError: Assignment destination is read-only and not writeable

ValueError: assignment destination is read-only
```

**More on creating arrays**

The following are various ways to create arrays

In [10]:

```
# Create an array of ones
np.ones((3,4))
```

Out[10]:

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

`numpy.zeros` and `numpy.empty` differs because `numpy.empty` does not set the array values to zero. Therefore, numpy.empty may be margainally faster. However, having an array of garbage values will usually still require manual setting at a later stage. As such, it is recommended to use numpy.zeros.

In [11]:

```
# Create an array of zeros
np.zeros((2,3,4), dtype=np.int16)
```

Out[11]:

```
array([[[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],

       [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]]], dtype=int16)
```

In [12]:

```
# Create an empty array
np.empty((3,2), dtype=np.int16)
```

Out[12]:

```
array([[257, 257],
       [257, 257],
       [257, 257]], dtype=int16)
```

In [13]:

```
# Create an array with random values
np.random.random((2,2))
```

Out[13]:

```
array([[0.65279032, 0.63505887],
       [0.99529957, 0.58185033]])
```

In [14]:

```
# Create a full array
np.full((2,2),7)
```

Out[14]:

```
array([[7, 7],
       [7, 7]])
```

In [15]:

```
# Create an array of evenly-spaced values
np.arange(10,25,5)
```

Out[15]:

```
array([10, 15, 20])
```

In [16]:

```
# Create an array of evenly-spaced values
np.linspace(0,2,9)
```

Out[16]:

```
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
```

In [17]:

```
# Create an array with ones on the diagonal and zeros elsewhere
np.eye(3, k=1)
```

Out[17]:

```
array([[0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])
```

In [18]:

```
# Create a square array with ones on the main diagonal
np.identity(3)
```

Out[18]:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

## Exercise

Measure the time the taken to multiply a list by itself compared to a numpy array multiplying by itself. Carry out the observation at size = 1000000

In [19]:

```
import numpy as np
import timeit
#todo exercise

# size of arrays and lists
size = 1000000

x = range(size) # declare a list
y = np.arange(size) # declare a numpy array

# NumPy array
startTime = timeit.default_timer()
resultArray = y * y

# print and calculate execution time
print("Time taken by NumPy Array :", (timeit.default_timer() - startTime), "seconds")

# list
startTime = timeit.default_timer()
resultList = [(a * a) for a in x]

# print and calculate execution time
print("Time taken by List :", (timeit.default_timer() - startTime), "seconds")
```

```
Time taken by NumPy Array : 0.0012890000000425061 seconds
Time taken by List : 0.09962069999983214 seconds
```

timeit is more accurate as

- it repeats the tests many times to eliminate the influence of other tasks on your machine, such as disk flushing and OS scheduling.
- it disables the garbage collector to prevent that process from skewing the results by scheduling a collection run at an inopportune moment.
- it picks the most accurate timer for your OS, time.time or time.clock in Python 2 and time.perf_counter() on Python 3. See timeit.default_timer.

In [22]:

```
size = 1000000
x = range(size) # declare a list
%timeit resultList = [(a * a) for a in x]
```

```
117 ms ± 13.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [24]:

```
size = 1000000
y = np.arange(size) # declare a numpy array
%timeit resultArray = y * y
```

```
1.97 ms ± 189 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

## Array Indexing: Accessing Single Elements

In a one-dimensional array, the $i^{th}$ value (counting from zero) can be accessed by specifying the desired index in square brackets.

In [25]:

```
a1
```

Out[25]:

```
array([5, 0, 3, 3, 7, 9])
```

In [26]:

```
a1[0]
```

Out[26]:

```
5
```

In [27]:

```
a1[4]
```

Out[27]:

```
7
```

To index from the end of the array, you can use negative indices:

In [28]:

```
a1[-1]
```

Out[28]:

```
9
```

In [29]:

```
a1[-2]
```

Out[29]:

```
7
```

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

In [30]:

```
a2
```

Out[30]:

```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

In [31]:

```
a2[0, 0]
```

Out[31]:

```
3
```

In [32]:

```
a2[2, 0]
```

Out[32]:

```
1
```

In [33]:

```
a2[2, -1]
```

Out[33]:

```
7
```

Values can also be modified using any of the above index notation:

In [34]:

```
a2[0, 0] = 12
a2
```

Out[34]:

```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

**Important**

Unlike Python lists, NumPy arrays have a fixed type. If you attempt to insert a floating-point value to an integer array, the value will be silently truncated.

In [35]:

```
a1[0] = 3.14159  # this will be truncated!
a1
```

Out[35]:

```
array([3, 0, 3, 3, 7, 9])
```

# Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon ( `:` ) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x` , use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0` , `stop=` *size of dimension* , `step=1` . We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

## One-dimensional subarrays

In [36]:

```
x = np.arange(10)
x
```

Out[36]:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [37]:

```
x[:5]  # first five elements
```

Out[37]:

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

In [38]:

```
x[5:]  # elements after index 5
```

Out[38]:

```
array([5, 6, 7, 8, 9])
```

In [39]:

```
x[4:7]  # middle sub-array
```

Out[39]:

```
array([4, 5, 6])
```

In [40]:

```
x[::2]  # every other element
```

Out[40]:

```
array([0, 2, 4, 6, 8])
```

In [41]:

```
x[1::2]  # every other element, starting at index 1
```

Out[41]:

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

A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

In [42]:

```
x[::-1]  # all elements, reversed
```

Out[42]:

```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

In [43]:

```
x[5::-2]  # reversed every other from index 5
```

Out[43]:

```
array([5, 3, 1])
```

## Multi-dimensional subarrays

NumPy slices can slice through multiple dimensions. Multi-dimensional slices work in the same way, with multiple slices separated by commas. For example:

In [44]:

```
a2
```

Out[44]:

```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

In [45]:

```
a2[:2, :3]  # two rows, three columns
```

Out[45]:

```
array([[12,  5,  2],
       [ 7,  6,  8]])
```

In [46]:

```
a2[:3, ::2]  # all rows, every other column
```

Out[46]:

```
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])
```

Finally, subarray dimensions can even be reversed together:

In [47]:

```
a2[::-1, ::-1]
```

Out[47]:

```
array([[ 7,  7,  6,  1],
       [ 8,  8,  6,  7],
       [ 4,  2,  5, 12]])
```

## Exercise

All arrays generated by NumPy basic slicing are always views of the original array, while slices of lists are shallow copies. Verify that the slicing of NumPy arrays are indeed referencing the original array by changing all values in the obtained slice to zero.

- Generate a 1-dimensional random integer (between 0 and 10) Numpy array of size 10
- Access from index 1 to 4
- Change all values in the slice to zero
- Determine if values change affected original numpy array

In [48]:

```
#todo exericse
a5 = np.random.randint(10, size=10) # One-dimensional array
b5 = a5.tolist()
print("original array is \n", a5, "\n")

a5slice = a5[1:5]
print("obtained slice is \n", a5slice, "\n")

a5slice *= 0
print("modified slice is \n", a5slice,"\n")

print("original array after slice modified is \n", a5, "\n")

print("original list is \n", b5, "\n")

b5slice = b5[1:5]
print("obtained list slice is \n", b5slice, "\n")

for x in range (len(b5slice)):
  b5slice[x] = 0
print("modified list slice is \n", b5slice,"\n")
print("original list after slice modified is \n", b5, "\n")
```

```
original array is
 [4 3 7 5 5 0 1 5 9 3]

obtained slice is
 [3 7 5 5]

modified slice is
 [0 0 0 0]

original array after slice modified is
 [4 0 0 0 0 0 1 5 9 3]

original list is
 [4, 3, 7, 5, 5, 0, 1, 5, 9, 3]

obtained list slice is
 [3, 7, 5, 5]

modified list slice is
 [0, 0, 0, 0]

original list after slice modified is
 [4, 3, 7, 5, 5, 0, 1, 5, 9, 3]
```

### Accessing array rows and columns

One commonly needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon ( `:` ):

In [49]:

```
print(a2)
```

```
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

In [50]:

```
print(a2[:, 0])  # first column of a2
```

```
[12  7  1]
```

In [51]:

```
print(a2[0, :])  # first row of a2
```

```
[12  5  2  4]
```

In the case of row access, the empty slice can be omitted for a more compact syntax:

In [52]:

```
print(a2[0])  # equivalent to a2[0, :]
```

```
[12  5  2  4]
```

## Subarrays as no-copy views

Again, recall that numpy array slices return *views* rather than *copies* of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Let's consider a two-dimensional array in this case:

In [53]:

```
print(a2)
```

```
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Let's extract a $2 \times 2$ subarray from this:

In [54]:

```
a2_sub = a2[:2, :2]
print(a2_sub)
```

```
[[12  5]
 [ 7  6]]
```

Now if we modify this subarray, we'll see that the original array is changed! Observe:

In [55]:

```
a2_sub[0, 0] = 88
print(a2_sub)
```

```
[[88  5]
 [ 7  6]]
```

In [56]:

```
print(a2)
```

```
[[88  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

## Creating copies of arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

In [57]:

```
a2_sub_copy = a2[:2, :2].copy()
print(a2_sub_copy)
```

```
[[88  5]
 [ 7  6]]
```

If we now modify this subarray, the original array is not touched:

In [58]:

```
a2_sub_copy[0, 0] = 42
print(a2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

In [59]:

```
print(a2)
```

```
[[88  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

# Reshaping of Arrays

The reshaping of arrays is a usefule operation that changes the shape of an array without changing the data of the array. The most flexible way of doing this is with the `reshape` method. For example, if you want to put the numbers 1 through 9 in a $3 \times 3$ grid, you can do the following:

In [60]:

```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

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

Note that for this to work, the size of the initial array must match the size of the reshaped array. Where possible, the `reshape` method will use a no-copy view of the initial array, but with non-contiguous memory buffers this is not always the case. A contiguous array is just an array stored in an unbroken block of memory and to access the next value in the array, we just move to the next memory address

Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. This can be done with the `reshape` method, or more easily done by making use of the `newaxis` keyword within a slice operation. The `newaxis` increases the exisitng array by one more dimension, when used once.

In [61]:

```
x = np.array([1, 2, 3])
print(x.shape)
# row vector via reshape
x2 = x.reshape((1, 3))

x3 = x.reshape((1, 3))
#print(x.shape)
```

```
(3,)
```

In [62]:

```
# row vector via newaxis
x[np.newaxis, :]
```

Out[62]:

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

In [63]:

```
# column vector via reshape
x.reshape((3, 1))
```

Out[63]:

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

In [64]:

```
# column vector via newaxis
x[:, np.newaxis]
```

Out[64]:

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

### Exercise

Make a copy of the following Numpy array, x, and reshape the copy of x into a shape of (1,6)

```
x = np.array([[1, 2, 3], [4, 5, 6]])
```

In [66]:

```
#todo: exercise

x = np.array([[1, 2, 3], [4, 5, 6]])
print(x.shape)
x_copy = x.copy()
new_x = x_copy.reshape(1,6)
print(new_x)
```

```
(2, 3)
[[1 2 3 4 5 6]]
```

## Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

## Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

In [67]:

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

Out[67]:

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

You can also concatenate more than two arrays at once:

In [68]:

```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

It can also be used for two-dimensional arrays:

In [69]:

```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
```

In [70]:

```
# concatenate along the first axis
np.concatenate([grid, grid])
```

Out[70]:

```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

In [71]:

```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

Out[71]:

```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

In [72]:

```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])

# vertically stack the arrays
np.vstack([x, grid])
```

Out[72]:

```
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

In [73]:

```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

Out[73]:

```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

Similary, `np.dstack` will stack arrays along the third axis.

In [74]:

```
z = np.array([[1,2,3],
            [5,6,7]])

np.dstack([grid, z])
```

Out[74]:

```
array([[[9, 1],
        [8, 2],
        [7, 3]],

       [[6, 5],
        [5, 6],
        [4, 7]]])
```

## Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split` , `np.hsplit` , and `np.vsplit` . For each of these, we can pass a list of indices giving the split points:

In [75]:

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

Notice that *N* split-points, leads to *N + 1* subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

In [76]:

```
grid = np.arange(16).reshape((4, 4))
grid
```

Out[76]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [77]:

```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

In [78]:

```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Similarly, `np.dsplit` will split arrays along the third axis.

# Array mathematics

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the NumPy module.

In [79]:

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

v = np.array([9,10])
w = np.array([11, 12])
```

Elementwise addition

In [80]:

```
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

Elementwise subtraction

In [81]:

```
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

Elementwise multiplication

In [82]:

```
print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

Elementwise division

In [83]:

```
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

Elementwise square root

In [84]:

```
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Inner product of vectors

In [85]:

```
print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

Vector product

In [86]:

```
print(x.dot(v))
print(np.dot(x, v))
print()
print(x.dot(y))
print(np.dot(x, y))
```

```
[29. 67.]
[29. 67.]

[[19. 22.]
 [43. 50.]]
[[19. 22.]
 [43. 50.]]
```

Sum of all elements

In [87]:

```
print(x, "\n")
print(np.sum(x))
```

```
[[1. 2.]
 [3. 4.]]

10.0
```

Sum of each column

In [88]:

```
print(np.sum(x, axis=0))
```

```
[4. 6.]
```

Sum of each row

In [89]:

```
print(np.sum(x, axis=1))
```

```
[3. 7.]
```

# Structured Data: NumPy's Structured Arrays

While data can be well represented by a homogeneous array of values, this is not always the case. NumPy's *structured arrays* and *record arrays*, can provide efficient storage for compound, heterogeneous data. While the patterns shown here are useful for simple operations, scenarios like this often lend themselves to the use of Pandas `Dataframe` s.

In [90]:

```
import numpy as np
```

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

In [91]:

```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
```

But this is a bit clumsy. There's nothing here that tells us that the three arrays are related; it would be more natural if we could use a single structure to store all of this data. NumPy can handle this through structured arrays, which are arrays with compound data types.

Recall that previously we created a simple array using an expression like this:

In [92]:

```
x = np.zeros(4, dtype=int)
```

We can similarly create a structured array using a compound data type specification:

In [93]:

```
# Use a compound data type for structured arrays
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                          'formats':('U10', 'i4', 'f8')})
print(data.dtype)
```

```
[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

Here `'U10'` translates to "Unicode string of maximum length 10," `'i4'` translates to "4-byte (i.e., 32 bit) integer," and `'f8'` translates to "8-byte (i.e., 64 bit) float." We'll discuss other options for these type codes in the following section.

Now that we've created an empty container array, we can fill the array with our lists of values:

In [94]:

```
data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
```

```
[('Alice', 25, 55. ) ('Bob', 45, 85.5) ('Cathy', 37, 68. )
 ('Doug', 19, 61.5)]
```

As we had hoped, the data is now arranged together in one convenient block of memory.

The handy thing with structured arrays is that you can now refer to values either by index or by name:

In [95]:

```
# Get all names
data['name']
```

Out[95]:

```
array(['Alice', 'Bob', 'Cathy', 'Doug'], dtype='<U10')
```

In [96]:

```
# Get first row of data
data[0]
```

Out[96]:

```
('Alice', 25, 55.)
```

In [97]:

```
# Get the name from the last row
data[-1]['name']
```

Out[97]:

```
'Doug'
```

Using Boolean masking, this even allows you to do some more sophisticated operations such as filtering on age:

In [98]:

```
# Get names where age is under 30
data[data['age'] < 30]['name']
```

Out[98]:

```
array(['Alice', 'Doug'], dtype='<U10')
```

Note that if you'd like to do any operations that are any more complicated than these, you should probably consider the Pandas package. The Pandas package provides a `Dataframe` object, which is a structure built on NumPy arrays that offers a variety of useful data manipulation functionality similar to what we've shown here, as well as much, much more.

## Creating Structured Arrays

Structured array data types can be specified in a number of ways. Earlier, we saw the dictionary method:

In [99]:

```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':('U10', 'i4', 'f8')})
```

Out[99]:

```
dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

For clarity, numerical types can be specified using Python types or NumPy `dtype` s instead:

In [100]:

```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':((np.str_, 10), int, np.float32)})
```

Out[100]:

```
dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f4')])
```

A compound type can also be specified as a list of tuples:

In [101]:

```
np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
```

Out[101]:

```
dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

If the names of the types do not matter to you, you can specify the types alone in a comma-separated string:

In [102]:

```
np.dtype('S10,i4,f8')
```

Out[102]:

```
dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

The shortened string format codes may seem confusing, but they are built on simple principles. The first (optional) character is `<` or `>`, which means "little endian" or "big endian," respectively, and specifies the ordering convention for significant bits. The next character specifies the type of data: characters, bytes, ints, floating points, and so on (see the table below). The last character or characters represents the size of the object in bytes.

| Character | Description | Example |
|---|---|---|
| `'b'` | Byte | `np.dtype('b')` |
| `'i'` | Signed integer | `np.dtype('i4') == np.int32` |
| `'u'` | Unsigned integer | `np.dtype('u1') == np.uint8` |
| `'f'` | Floating point | `np.dtype('f8') == np.int64` |
| `'c'` | Complex floating point | `np.dtype('c16') == np.complex128` |
| `'S'`, `'a'` | String | `np.dtype('S5')` |
| `'U'` | Unicode string | `np.dtype('U') == np.str_` |
| `'V'` | Raw data (void) | `np.dtype('V') == np.void` |

## More Advanced Compound Types

It is possible to define even more advanced compound types. For example, you can create a type where each element contains an array or matrix of values. Here, we'll create a data type with a `mat` component consisting of a $3\times 3$ floating-point matrix:

In [103]:

```
tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
print(X[0])
print(X['mat'][0])
```

```
(0, [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Now each element in the `X` array consists of an `id` and a $3\times 3$ matrix. Why would you use this rather than a simple multidimensional array, or perhaps a Python dictionary? The reason is that this NumPy `dtype` directly maps onto a C structure definition, so the buffer containing the array content can be accessed directly within an appropriately written C program. If you find yourself writing a Python interface to a legacy C or Fortran library that manipulates structured data, you'll probably find structured arrays quite useful!

## RecordArrays: Structured Arrays with a Twist

NumPy also provides the `np.recarray` class, which is almost identical to the structured arrays just described, but with one additional feature: fields can be accessed as attributes rather than as dictionary keys. Recall that we previously accessed the ages by writing:

In [104]:

```
data['age']
```

Out[104]:

```
array([25, 45, 37, 19])
```

If we view our data as a record array instead, we can access this with slightly fewer keystrokes:

In [105]:

```
data_rec = data.view(np.recarray)
data_rec.age
```

Out[105]:

```
array([25, 45, 37, 19])
```

The downside is that for record arrays, there is some extra overhead involved in accessing the fields, even when using the same syntax. We can see this here:

In [106]:

```
%timeit data['age']
%timeit data_rec['age']
%timeit data_rec.age
```

```
237 ns ± 22.3 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
4.55 µs ± 197 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
5.4 µs ± 308 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Whether the more convenient notation is worth the additional overhead will depend on your own application.

## Exercise

Create a structure array for the following table of data

| index | stock | product ID | name |
|---|---|---|---|
| 1 | 200 | Z143 | Batteries |
| 2 | 100 | BOK9 | Books |
| 3 | 250 | C982 | Bicycles |

In [107]:

```
#todo exercise

# Use a compound data type for structured arrays
data = np.zeros(3, dtype={'names':('index','stock', 'product ID', 'name'),'formats':('i4',
'int', 'U10', 'U10')})
name = ['Batteries', 'Books','Bicycles']
productID = ['Z143', 'BOK9', 'C982']
stock = [200, 100, 250]
index = [1,2,3]
data['index'] = index
data['stock'] = stock
data['product ID'] = productID
data['name'] = name
print(data)
```

```
[(1, 200, 'Z143', 'Batteries') (2, 100, 'BOK9', 'Books')
 (3, 250, 'C982', 'Bicycles')]
```

## Sorting

NumPy arrays can be sorted in-place using the sort method:

In [108]:

```
array1 = np.random.randn(5)
array1.sort()
print(array1)
```

```
[-0.45792242 -0.02797118  0.4253934   1.11971196  1.47598983]
```

In [109]:

```
array2 = np.random.randn(5, 5)

array2_a = array2.copy()
array2_b = array2.copy()

print(array2,"\n")

#sort along axis 1
array2_a.sort(0)
array2_b.sort(1)

print (array2_a,"\n")
print (array2_b,"\n")
```

```
[[ 0.6467801  -0.36433431 -0.67877739 -0.35362786 -0.74074747]
 [-0.67502183 -0.13278426  0.61980106  1.79116846  0.17100044]
 [-1.72567135  0.16065854 -0.85898532 -0.20642094  0.48842647]
 [-0.83833097  0.38116374 -0.99090328  1.01788005  0.3415874 ]
 [-1.25088622  0.92525075 -0.90478616  1.84369153  1.52550724]]

[[-1.72567135 -0.36433431 -0.99090328 -0.35362786 -0.74074747]
 [-1.25088622 -0.13278426 -0.90478616 -0.20642094  0.17100044]
 [-0.83833097  0.16065854 -0.85898532  1.01788005  0.3415874 ]
 [-0.67502183  0.38116374 -0.67877739  1.79116846  0.48842647]
 [ 0.6467801   0.92525075  0.61980106  1.84369153  1.52550724]]

[[-0.74074747 -0.67877739 -0.36433431 -0.35362786  0.6467801 ]
 [-0.67502183 -0.13278426  0.17100044  0.61980106  1.79116846]
 [-1.72567135 -0.85898532 -0.20642094  0.16065854  0.48842647]
 [-0.99090328 -0.83833097  0.3415874   0.38116374  1.01788005]
 [-1.25088622 -0.90478616  0.92525075  1.52550724  1.84369153]]
```

**Save arrays on disk in binary format**

In [110]:

```
print(array2)
np.save('array_on_disk',array2)
```

```
[[ 0.6467801  -0.36433431 -0.67877739 -0.35362786 -0.74074747]
 [-0.67502183 -0.13278426  0.61980106  1.79116846  0.17100044]
 [-1.72567135  0.16065854 -0.85898532 -0.20642094  0.48842647]
 [-0.83833097  0.38116374 -0.99090328  1.01788005  0.3415874 ]
 [-1.25088622  0.92525075 -0.90478616  1.84369153  1.52550724]]
```

**Load arrays on disk in binary format**

In [111]:

```
np.load('array_on_disk.npy')
```

Out[111]:

```
array([[ 0.6467801 , -0.36433431, -0.67877739, -0.35362786, -0.74074747],
       [-0.67502183, -0.13278426,  0.61980106,  1.79116846,  0.17100044],
       [-1.72567135,  0.16065854, -0.85898532, -0.20642094,  0.48842647],
       [-0.83833097,  0.38116374, -0.99090328,  1.01788005,  0.3415874 ],
       [-1.25088622,  0.92525075, -0.90478616,  1.84369153,  1.52550724]])
```

**Save array to text file**

In [112]:

```
np.savetxt('array_txt.txt', array2, delimiter=',' )
```

**Load array from text file**

In [70]:

```
np.loadtxt('array_txt.txt', delimiter=',')
```

Out[70]:

```
array([[ 1.14907613, -1.19357825,  1.14104245,  1.50944508,  1.06777513],
       [-0.68658948,  0.01487332, -0.3756659 , -0.03822364,  0.36797447],
       [-0.0447237 , -0.30237513, -2.2244036 ,  0.72400636,  0.35900276],
       [ 1.07612104,  0.19214083,  0.85292596,  0.01835718,  0.42830357],
       [ 0.99627783, -0.49114966,  0.71267817,  1.11334035, -2.15367459]])
```

In [ ]:

```

```