numpy

August 29, 2020

1 NUMPY cheetsheet

Advantages of using Numpy :

NumPy's arrays are more compact than Python lists – a list of lists as you describe, in Python, would take at least 20 MB or so, while a NumPy 3D array with single-precision floats in the cells would fit in 4 MB. Access in reading and writing items is also faster with NumPy.

Maybe you don't care that much for just a million cells, but you definitely would for a billion cells – neither approach would fit in a 32-bit architecture, but with 64-bit builds NumPy would get away with 4 GB or so, Python alone would need at least about 12 GB (lots of pointers which double in size) – a much costlier piece of hardware!

The difference is mostly due to "indirectness" – a Python list is an array of pointers to Python objects, at least 4 bytes per pointer plus 16 bytes for even the smallest Python object (4 for type pointer, 4 for reference count, 4 for value – and the memory allocators rounds up to 16). A NumPy array is an array of uniform values – single-precision numbers takes 4 bytes each, double-precision ones, 8 bytes. Less flexible, but you pay substantially for the flexibility of standard Python lists!

[33]:
```
import numpy as np
import time
import sys

a=np.array([1,2,3])
print (a)
print(a[1])

l=list(range(1000))

print("size of list is ",sys.getsizeof(3)*len(l))#size is 14000

array=np.arange(1000)
print("size of array is ",array.size*array.itemsize)#size is 4000
```

```
[1 2 3]
2
size of list is  28000
size of array is  4000
```

Let us look at the below program which compares NumPy Arrays and Lists in Python in terms of

execution time.

[34]:
```
size=1000000

l1=range(size)
l2=range(size)

a1=np.arange(size)
a2=np.arange(size)

#python list
start=time.time()
result=[(x+y) for x,y in zip(l1,l2)]
print("python list took : ",(time.time()-start)*1000)

#numpy array
start=time.time()
result=a1+a2
print("numpy took : ",(time.time()-start)*1000)
```

```
python list took :  238.36350440979004
numpy took :  26.927471160888672
```

From the output of the above program, we see that the NumPy Arrays execute very much faster than the Lists in Python. There is a big difference between the execution time of arrays and lists.

NumPy Arrays are faster than Python Lists because of the following reasons:

An array is a collection of homogeneous data-types which are stored in contagious memory locations, on the other hand, a list in Python is collection of heterogeneous data types stored in non-contagious memory locations. The NumPy package breakdowns a task into multiple fragments, and then processes all the fragments parallelly. The NumPy package integrates C, C++ and Fortran codes in Python, these programming languages have very less execution time as compared to python.

Below is a program which compares the execution time of different operations on NumPy arrays and Python Lists:

[35]:
```
import numpy
size = 1000000

# declaring lists
list1 = [i for i in range(size)]
list2 = [i for i in range(size)]

# decalring arrays
array1 = numpy.arange(size)
array2 = numpy.arange(size)

# Concatenation
print("\nConcatenation:")
```

```
# list
initialTime = time.time()
list1 = list1 + list2

# calculating execution time
print("Time taken by Lists :", (time.time() - initialTime), "seconds")

# NumPy array
initialTime = time.time()
array = numpy.concatenate((array1, array2), axis = 0)

# calculating execution time
print("Time taken by NumPy Arrays :", (time.time() - initialTime), "seconds")

# Dot Product
dot = 0
print("\nDot Product:")

# list
initialTime = time.time()
for a, b in zip(list1, list2):
    dot = dot + (a * b)

# calculating execution time
print("Time taken by Lists :", (time.time() - initialTime), "seconds")

# NumPy array
initialTime = time.time()
array = numpy.dot(array1, array2)

# calculating execution time
print("Time taken by NumPy Arrays :", (time.time() - initialTime), "seconds")

# Scalar Addtion
print("\nScalar Addtion:")

# list
initialTime = time.time()
list1 =[i + 2 for i in range(size)]

# calculating execution time
print("Time taken by Lists :", (time.time() - initialTime), "seconds")

# NumPy array
initialTime = time.time()
array1 = array1 + 2
```

```
# calculating execution time
print("Time taken by NumPy Arrays :", (time.time() - initialTime), "seconds")

# Deletion
print("\nDeletion: ")

# list
initialTime = time.time()
del(list1)

# calculating execution time
print("Time taken by Lists :", (time.time() - initialTime), "seconds")

# NumPy array
initialTime = time.time()
del(array1)

# calculating execution time
print("Time taken by NumPy Arrays :", (time.time() - initialTime), "seconds")
```

```
Concatenation:
Time taken by Lists : 0.022942781448364258 seconds
Time taken by NumPy Arrays : 0.005986213684082031 seconds

Dot Product:
Time taken by Lists : 0.3061823844909668 seconds
Time taken by NumPy Arrays : 0.0019948482513427734 seconds

Scalar Addtion:
Time taken by Lists : 0.2014610767364502 seconds
Time taken by NumPy Arrays : 0.003990650177001953 seconds

Deletion:
Time taken by Lists : 0.021939754486083984 seconds
Time taken by NumPy Arrays : 0.0009984970092773438 seconds
```

From the above program, we conclude that operations on NumPy arrays are executed faster than Python lists, moreover, the Deletion operation has the highest difference in execution time between an array and a list as compared to other operations in the program.

Some Mathematical operations on numpy array

[36]:
```
a3=np.array([1,2,3])
a4=np.array([4,5,6])
print("Arrays are ",a3,a4)
print(a3+a4)
```

```
print(a4-a3)
print(a4*a3)
print(a4/a3)
```

```
Arrays are  [1 2 3] [4 5 6]
[5 7 9]
[3 3 3]
[ 4 10 18]
[4.  2.5 2. ]
```

Integer type numpy array

[37]:
```
a=np.array([[1,2,3],[4,5,6],[7,8,9]])
print("Array is :\n",a)
print("Dimension of array is :",a.ndim)
print("Item size of the element :",a.itemsize)
print("Data type of the element :",a.dtype)
print("Size of the array :",a.size)
print("Shape of the array :",a.shape)
```

```
Array is :
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Dimension of array is : 2
Item size of the element : 4
Data type of the element : int32
Size of the array : 9
Shape of the array : (3, 3)
```

Float type numpy array

[38]:
```
a=np.array([[1,2,3],[4,5,6],[7,8,9]],dtype=np.float64)
print("Array is :\n",a)
print("Dimension of array is :",a.ndim)
print("Item size of the element :",a.itemsize)
print("Data type of the element :",a.dtype)
print("Size of the array :",a.size)
print("Shape of the array :",a.shape)
```

```
Array is :
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
Dimension of array is : 2
Item size of the element : 8
Data type of the element : float64
Size of the array : 9
Shape of the array : (3, 3)
```

Complex type numpy array

[39]:
```
a=np.array([[1,2,3],[4,5,6],[7,8,9]],dtype=np.complex)
print("Array is :\n",a)
print("Dimension of array is :",a.ndim)
print("Item size of the element :",a.itemsize)
print("Data type of the element :",a.dtype)
print("Size of the array :",a.size)
print("Shape of the array :",a.shape)
```

```
Array is :
 [[1.+0.j 2.+0.j 3.+0.j]
 [4.+0.j 5.+0.j 6.+0.j]
 [7.+0.j 8.+0.j 9.+0.j]]
Dimension of array is : 2
Item size of the element : 16
Data type of the element : complex128
Size of the array : 9
Shape of the array : (3, 3)
```

Some of the ways to create a numpy array

Array of zeros and ones

[40]:
```
x=np.zeros((10))
print(x)
print(x.reshape((2,5)))
x=np.ones((3,4))
print(x)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

Numpy has a method arrange() which is almost similar like range() . We can assign start index, end index and step size also. By default the step size is 1 and the start index is 0

[41]:
```
x=np.arange(10)
print(x)

x=np.arange(1,20,3)
print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 1  4  7 10 13 16 19]
```

There is one more method to create the numpy array

the space between the element is calculated by ((end_index - start_index)/(steps - 1))

[91]:
```
x=np.linspace(0,5,11)#distance is ((5-0)/(11-1))
print(x)
x=np.linspace(0,5,10)
print(x)
x=np.linspace(1,5,5)
print(x)
```

```
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
[0.         0.55555556 1.11111111 1.66666667 2.22222222 2.77777778
 3.33333333 3.88888889 4.44444444 5.        ]
[1. 2. 3. 4. 5.]
```

We can flatten the array and make it an one dimensional array

[76]:
```
a=np.array([[1,2],[3,4],[5,6]])
print(a.ravel())

# #flatting it as it is a single dimensional array then we can loop through it
for elem in a.flat:
    print(elem)
```

```
[1 2 3 4 5 6]
1
2
3
4
5
6
```

Some other matematical operation on numpy array

[43]:
```
a=np.array([[1,2],[3,4],[5,6]])

print("Array :\n",a)
print("Max element in the array : ",a.max())
print("Min element in the array : ",a.min())
print("Sum of all the element : ",a.sum())

print("Sum of element vertically : ",a.sum(axis=0))
print("Sum of element horizontally : ",a.sum(axis=1))

print("Square root of all the element : ",np.sqrt(a))
print("Standard deviation in the array",np.std(a))

a=np.array([[1,2],[3,4]])
b=np.array([[5,6],[7,8]])
print("Sum of the two array : \n",a+b)
```

```
print("matrix dot product of the two array : \n",a.dot(b))
```

```
Array :
 [[1 2]
 [3 4]
 [5 6]]
Max element in the array :  6
Min element in the array :  1
Sum of all the element :  21
Sum of element vertically :  [ 9 12]
Sum of element horizontally :  [ 3  7 11]
Square root of all the element :  [[1.         1.41421356]
 [1.73205081 2.        ]
 [2.23606798 2.44948974]]
Standard deviation in the array 1.707825127659933
Sum of the two array :
 [[ 6  8]
 [10 12]]
matrix dot product of the two array :
 [[19 22]
 [43 50]]
```

We can iterate the numpy array via different techniques

It will follow the C order that is the row major order

for cell in np.nditer(array,order='C'): print(cell)

It will follow the F order that is the column major order

for cell in np.nditer(array,order='F'): print(cell)

there is another type we have which will looping through the array and print the inner array

for x in np.nditer(a,order='F',flags=['external_loop']): print(x)

We can loop through two array sumulataneously. their shape must be same

for x,y in np.nditer([a,b]): print(x,y)

[ ]:

Datasets come from a wide range of sources and formats. it could be collections of numerical measurements, text corpus, images, audio clips, or basically anything. No matter the format, the first step in data science is to transform it into arrays of numbers.

[44]:
```
import numpy as np
heights = [189, 170, 189, 163, 183, 171, 185, 168, 173, 183,173, 173, 175,
           178, 183, 193, 178, 173, 174, 183, 183, 180, 168, 180, 170, 178,
           182, 180, 183, 178, 182, 188, 175, 179, 183, 193, 182, 183, 177, 185,
           188, 188, 182, 185, 191]
heights_arr = np.array(heights)
```

```
print((heights_arr > 188).sum())
```

```
5
```

An array class in Numpy is called an ndarray or n-dimensional array. We can use this to count the number of presidents in heights_arr, use attribute numpy.ndarray.size:

[45]:
```
heights_arr.size
```

[45]:
```
45
```

Note that once an array is created in numpy, its size cannot be changed.

Size tells us how big the array is, shape tells us the dimension. To get current shape of an array use attribute shape:

[46]:
```
heights_arr.shape
```

[46]:
```
(45,)
```

Reshape

Other data we have collected includes the ages of the presidents:

[47]:
```
ages = [57, 61, 57, 57, 58, 57, 61, 54, 68, 51, 49, 64, 50, 48,
        65, 52, 56, 46, 54, 49, 51, 47, 55, 55, 54,42, 51, 56, 55,
        51, 54, 51, 60, 62, 43, 55, 56, 61, 52, 69, 64, 46, 54, 47, 70]
```

Since both heights and ages are all about the same presidents, we can combine them:

[48]:
```
heights_and_ages = heights + ages
# convert a list to a numpy array
heights_and_ages_arr = np.array(heights_and_ages)
heights_and_ages_arr.shape
```

[48]:
```
(90,)
```

This produces one long array. It would be clearer if we could align height and age for each president and reorganize the data into a 2 by 45 matrix where the first row contains all heights and the second row contains ages. To achieve this, a new array can be created by calling numpy.ndarray.reshape with new dimensions specified in a tuple:

[49]:
```
heights_and_ages_arr.reshape((2,45))
```

[49]:
```
array([[189, 170, 189, 163, 183, 171, 185, 168, 173, 183, 173, 173, 175,
        178, 183, 193, 178, 173, 174, 183, 183, 180, 168, 180, 170, 178,
        182, 180, 183, 178, 182, 188, 175, 179, 183, 193, 182, 183, 177,
        185, 188, 188, 182, 185, 191],
       [ 57,  61,  57,  57,  58,  57,  61,  54,  68,  51,  49,  64,  50,
         48,  65,  52,  56,  46,  54,  49,  51,  47,  55,  55,  54,  42,
```

```
       51,  56,  55,  51,  54,  51,  60,  62,  43,  55,  56,  61,  52,
       69,  64,  46,  54,  47,  70]])
```

The reshaped array is now a 2darray, yet note that the original array is not changed. We can reshape an array in multiple ways, as long as the size of the reshaped array matches that of the original.

Numpy can calculate the shape (dimension) for us if we indicate the unknown dimension as -1. For example, given a 2darray `arr` of shape (3,4), arr.reshape(-1) would output a 1darray of shape (12,), while arr.reshape((-1,2)) would generate a 2darray of shape (6,2).

Data Type

Another characteristic about numpy array is that it is homogeneous, meaning each element must be of the same data type.

For example, in heights_arr, we recorded all heights in whole numbers; thus each element is stored as an integer in the array. To check the data type, use numpy.ndarray.dtype heights_arr.dtype

If we mixed a float number in, say, the first element is 189.0 instead of 189:

[50]:
```
heights_float = [189.0, 170, 189, 163, 183, 171, 185, 168, 173, 183, 173, 173,
                 175, 178, 183, 193, 178, 173, 174, 183, 183, 180, 168, 180,␣
 ↪170,
                 178, 182, 180, 183, 178, 182, 188, 175, 179, 183, 193,
                 182, 183, 177, 185, 188,188, 182, 185, 191]
```

Then after converting the list into an array, we'd see all other numbers are coerced into floats:

[51]:
```
heights_float_arr = np.array(heights_float)
print(heights_float_arr)
print(heights_float_arr.dtype)
```

```
[189. 170. 189. 163. 183. 171. 185. 168. 173. 183. 173. 173. 175. 178.
 183. 193. 178. 173. 174. 183. 183. 180. 168. 180. 170. 178. 182. 180.
 183. 178. 182. 188. 175. 179. 183. 193. 182. 183. 177. 185. 188. 188.
 182. 185. 191.]
float64
```

Numpy supports several data types such as int (integer), float (numeric floating point), and bool (boolean values, True and False). The number after the data type, ex. int64, represents the bitsize of the data type.

Indexing

We can use array indexing to select individual elements from arrays. Like Python lists, numpy index starts from 0.

To access the height of the 3rd president Thomas Jefferson in the 1darray 'heights_arr':

[52]:
```
heights_arr[2]
```

[52]: 189

In a 2darray, there are two axes, axis 0 and 1. Axis 0 runs downward down the rows whereas axis 1 runs horizontally across the columns.

In the 2darrary heights_and_ages_arr, recall that its dimensions are (2, 45). To find Thomas Jefferson's age at the beginning of his presidency you would need to access the second row where ages are stored:

[53]:
```
heights_and_ages_arr = heights_and_ages_arr.reshape((2,45))
heights_and_ages_arr[1,2]
```

[53]: 57

In 2darray, the row is axis 0 and the column is axis 1, therefore, to access a 2darray, numpy first looks for the position in rows, then in columns. So in our example heights_and_ages_arr[1,2], we are accessing row 2 (ages), column 3 (third president) to find Thomas Jefferson's age.

Slicing

What if we want to inspect the first three elements from the first row in a 2darray? We use ":" to select all the elements from the index up to but not including the ending index. This is called slicing.

[54]:
```
heights_and_ages_arr[0, 0:3]
```

[54]: array([189, 170, 189])

When the starting index is 0, we can omit it as shown below:

[55]:
```
heights_and_ages_arr[0, :3]
```

[55]: array([189, 170, 189])

What if we'd like to see the entire third column? Specify this by using a ":" as follows

[56]:
```
heights_and_ages_arr[:, 3]
```

[56]: array([163, 57])

Numpy slicing syntax follows that of a python list: arr[start:stop:step]. When any of these are unspecified, they default to the values start=0, stop=size of dimension, step=1.

Assigning an Array to an Array

In addition, a 1darray or a 2darry can be assigned to a subset of another 2darray, as long as their shapes match. Recall the 2darray heights_and_ages_arr: heights_and_ages_arr

If we want to update both height and age of the first president with new data, we can supply the data in a list:

[57]:
```
heights_and_ages_arr[:, 0] = [190, 58]
heights_and_ages_arr
```

[57]:
```
array([[190, 170, 189, 163, 183, 171, 185, 168, 173, 183, 173, 173, 175,
        178, 183, 193, 178, 173, 174, 183, 183, 180, 168, 180, 170, 178,
        182, 180, 183, 178, 182, 188, 175, 179, 183, 193, 182, 183, 177,
        185, 188, 188, 182, 185, 191],
       [ 58,  61,  57,  57,  58,  57,  61,  54,  68,  51,  49,  64,  50,
         48,  65,  52,  56,  46,  54,  49,  51,  47,  55,  55,  54,  42,
         51,  56,  55,  51,  54,  51,  60,  62,  43,  55,  56,  61,  52,
         69,  64,  46,  54,  47,  70]])
```

We can also update data in a subarray with a numpy array as such:

[58]:
```
new_record = np.array([[180, 183, 190], [54, 50, 69]])
heights_and_ages_arr[:, 42:] = new_record
heights_and_ages_arr
```

[58]:
```
array([[190, 170, 189, 163, 183, 171, 185, 168, 173, 183, 173, 173, 175,
        178, 183, 193, 178, 173, 174, 183, 183, 180, 168, 180, 170, 178,
        182, 180, 183, 178, 182, 188, 175, 179, 183, 193, 182, 183, 177,
        185, 188, 188, 180, 183, 190],
       [ 58,  61,  57,  57,  58,  57,  61,  54,  68,  51,  49,  64,  50,
         48,  65,  52,  56,  46,  54,  49,  51,  47,  55,  55,  54,  42,
         51,  56,  55,  51,  54,  51,  60,  62,  43,  55,  56,  61,  52,
         69,  64,  46,  54,  50,  69]])
```

Note the last three columns' values have changed. Updating a multidimensional array with a new record is straightforward in numpy as long as their shapes match.

Combining Two Arrays

Oftentime we obtain data stored in different arrays and we need to combine them into one to keep it in one place. For example, instead of having the ages stored in a list, it could be stored in a 2darray:

[59]:
```
ages_arr = np.array(ages)
print(ages_arr.shape)
print(ages_arr[:3,])
```

```
(45,)
[57 61 57]
```

If we reshape the heights_arr to (45,1), the same as 'ages_arr', we can stack them horizontally (by column) to get a 2darray using 'hstack':

hstack can be done when both of the arrays have same row number. after hstack column no will be incresed. The first parameter in the array.shape represents the number of columns.

[60]:
```
heights_arr = heights_arr.reshape((45,1))
ages_arr = ages_arr.reshape((45,1))
height_age_arr = np.hstack((heights_arr, ages_arr))
print(height_age_arr.shape)
print(height_age_arr[:3,])
```

```
(45, 2)
[[189  57]
 [170  61]
 [189  57]]
```

Now height_age_arr has both heights and ages for the presidents, each column corresponds to the height and age of one president.

Similarly, if we want to combine the arrays vertically (by row), we can use 'vstack'.

vstack can be done when both of the arrays have same column number. after vstack column no will be incresed. The seceond parameter in the array.shape represents the number of columns.

[61]:
```
heights_arr = heights_arr.reshape((1,45))
ages_arr = ages_arr.reshape((1,45))

height_age_arr = np.vstack((heights_arr, ages_arr))
print(height_age_arr.shape)
height_age_arr[:,:3]
```

```
(2, 45)
```

[61]:
```
array([[189, 170, 189],
       [ 57,  61,  57]])
```

To combine more than two arrays horizontally, simply add the additional arrays into the tuple.

Concatenate

More generally, we can use the function numpy.concatenate. If we want to concatenate, link together, two arrays along rows, then pass 'axis = 1' to achieve the same result as using numpy.hstack; and pass 'axis = 0' if you want to combine arrays vertically.

In the example from the previous part, we were using hstack to combine two arrays horizontally, instead:

[62]:
```
height_age_arr = np.concatenate((heights_arr, ages_arr), axis=1)
height_age_arr
```

[62]:
```
array([[189, 170, 189, 163, 183, 171, 185, 168, 173, 183, 173, 173, 175,
        178, 183, 193, 178, 173, 174, 183, 183, 180, 168, 180, 170, 178,
        182, 180, 183, 178, 182, 188, 175, 179, 183, 193, 182, 183, 177,
        185, 188, 188, 182, 185, 191,  57,  61,  57,  57,  58,  57,  61,
         54,  68,  51,  49,  64,  50,  48,  65,  52,  56,  46,  54,  49,
         51,  47,  55,  55,  54,  42,  51,  56,  55,  51,  54,  51,  60,
```

```
       62,  43,  55,  56,  61,  52,  69,  64,  46,  54,  47,  70]])
```

Also you can get the same result as using vstack:

[63]:
```
height_age_arr = np.concatenate((heights_arr, ages_arr), axis=0)
height_age_arr
```

[63]:
```
array([[189, 170, 189, 163, 183, 171, 185, 168, 173, 183, 173, 173, 175,
        178, 183, 193, 178, 173, 174, 183, 183, 180, 168, 180, 170, 178,
        182, 180, 183, 178, 182, 188, 175, 179, 183, 193, 182, 183, 177,
        185, 188, 188, 182, 185, 191],
       [ 57,  61,  57,  57,  58,  57,  61,  54,  68,  51,  49,  64,  50,
         48,  65,  52,  56,  46,  54,  49,  51,  47,  55,  55,  54,  42,
         51,  56,  55,  51,  54,  51,  60,  62,  43,  55,  56,  61,  52,
         69,  64,  46,  54,  47,  70]])
```

You can use np.hstack to concatenate arrays ONLY if they have the same number of rows.

Just like hstack and vstack we have hsplit and vsplit in numpy which can split an array horizonally and vertically

[89]:
```
# Example of hplit
heights_arr_temp = heights_arr.reshape((5,9))
print("Before splitting the shape was : ",heights_arr_temp.shape)
heights_arr_temp = np.array(np.hsplit(heights_arr_temp,3))
print("After splitting the array in horizontanlly the shape is␣
 ↪",heights_arr_temp.shape)

# Example of hplit
heights_arr_temp = heights_arr.reshape((9,5))
print("Before splitting the shape was : ",heights_arr_temp.shape)
heights_arr_temp = np.array(np.vsplit(heights_arr_temp,3))
print("After splitting the array in vertically the shape is ",heights_arr_temp.
 ↪shape)
```

```
Before splitting the shape was :  (5, 9)
After splitting the array in horizontanlly the shape is  (3, 5, 3)
Before splitting the shape was :  (9, 5)
After splitting the array in vertically the shape is  (3, 3, 5)
```

Mathematical Operations on Arrays

Performing mathematical operations on arrays is straightforward. For instance, to convert the heights from centimeters to feet, knowing that 1 centimeter is equal to 0.0328084 feet, we can use multiplication:

[64]:
```
height_age_arr[:,0]*0.0328084
```

[64]:
```
array([6.2007876, 1.8700788])
```

Now we have all heights in feet. Note that this operation won't change the original array, it returns a new 1darray where 0.0328084 has been multiplied to each element in the first column of 'heights_age_arr'. Other mathematical operations for addition, subtraction, division and power (+, -, /, **) work the same way on arrays.

Numpy Array Method

In addition, there are several methods in numpy to perform more complex calculations on arrays. For example, the sum() method finds the sum of all the elements in an array:

[65]:
```
height_age_arr.sum()
```

[65]:
```
10575
```

The sum of all heights and ages is 10575. In order to sum all heights and sum all ages separately, we can specify axis=0 to calculate the sum across the rows, that is, it computes the sum for each column, or column sum. On the other hand, to obtain the row sums specify axis=1. In this example, we want to calculate the total sum of heights and ages, respectively:

[66]:
```
height_age_arr.sum(axis=0)
```

[66]:
```
array([246, 231, 246, 220, 241, 228, 246, 222, 241, 234, 222, 237, 225,
       226, 248, 245, 234, 219, 228, 232, 234, 227, 223, 235, 224, 220,
       233, 236, 238, 229, 236, 239, 235, 241, 226, 248, 238, 244, 229,
       254, 252, 234, 236, 232, 261])
```

The output is the row sums: heights of all presidents (i.e., the first row) add up to 8100, and the sum of ages (i.e., the second row) is 2475.

[67]:
```
height_age_arr.sum(axis=1)
```

[67]:
```
array([8100, 2475])
```

Other operations, such as .min(), .max(), .mean(), work in a similar way to .sum().

Comparisons

In practicing data science, we often encounter comparisons to identify rows that match certain values. We can use operations including "<", ">", ">=", "<=", and "==" to do so. For example, in the height_age_arr dataset, we might be interested in only those presidents who started their presidency younger than 55 years old.

[68]:
```
height_age_arr[:, 1] < 55
```

[68]:
```
array([False, False])
```

The output is a 1darray with boolean values that indicates which presidents meet the criteria. If we are only interested in which presidents started their presidency at 51 years of age, we can use "==" instead.

[69]:
```
height_age_arr[:, 1] == 51
```

[69]:
```
array([False, False])
```

To find out how many rows satisfy the condition, use .sum() on the resultant 1d boolean array, e.g., (height_age_arr[:, 1] == 51).sum(), to see that there were exactly five presidents who started the presidency at age 51. True is treated as 1 and False as 0 in the sum.

Mask & Subsetting

Now that rows matching certain criteria can be identified, a subset of the data can be found. For example, instead of the entire dataset, we want only tall presidents, that is, those presidents whose height is greater than or equal to 182 cm. We first create a mask, 1darray with boolean values:

[70]:
```
mask = height_age_arr[:, 0] >= 182
mask.sum()
```

[70]:
```
1
```

Then pass it to the first axis of `height_age_arr` to filter presidents who don't meet the criteria:

[71]:
```
tall_presidents = height_age_arr[mask, ]
tall_presidents.shape
```

[71]:
```
(1, 45)
```

This is a subarray of height_age_arr, and all presidents in tall_presidents were at least 182cm tall. Masking is used to extract, modify, count, or otherwise manipulate values in an array based on some criterion. In our example, the criteria was height of 182cm or taller.

Multiple Criteria

We can create a mask satisfying more than one criteria. For example, in addition to height, we want to find those presidents that were 50 years old or younger at the start of their presidency. To achieve this, we use & to separate the conditions and each condition is encapsulated with parentheses "()" as shown below:

[72]:
```
height_age_arr=np.hstack((heights_arr.reshape((45,1)), ages_arr.
 ↪reshape((45,1))))
print(height_age_arr[:2])
mask = (height_age_arr[:, 0]>=182) & (height_age_arr[:,1]<=50)
height_age_arr[mask,]
```

```
[[189  57]
 [170  61]]
```

[72]:
```
array([[183,  49],
       [183,  43],
       [188,  46],
       [185,  47]])
```

The results show us that there are four presidents who satisfy both conditions. Data manipulation in Python is nearly synonymous with Numpy array manipulation. Operations shown here are the

building blocks of many other examples used throughout this course. It is important to master them!

[ ]: