Views and Copies

This lesson discusses views and copies, and their uses in NumPy. We'll also find out the difference between indexing and fancy indexing and between ravel and flatten and how and why we make temporary copy.

We'll cover the following

Views and copies are important concepts for the optimization of your numerical computations. Even if we've already manipulated them in the previous section, the whole story is a bit more complex.

Direct and indirect access

While executing a numpy instruction, either a copy of the input array is created or a view is provided. When the contents are physically stored in another location, it is called **Copy**. If on the other hand, a different view of the same memory content is provided, we call it **View**.

Indexing and Fancy Indexing

First, we have to distinguish between indexing and fancy indexing. The first will always return a view while the second will return a copy. This difference is important because in the first case, modifying the view modifies the base array while this is not true in the second case:

```
import numpy as np

Z = np.zeros(9)
```

```
Z_view = Z[:3]
Z_view[...] = 1 #Z_view modifies the base array 'Z'
print(Z)
Z = np.zeros(9)
Z_copy = Z[[0,1,2]]
Z_copy[...] = 1 #Z_copy does not modify the base array 'Z'
print(Z)
```

Thus, if you need fancy indexing, it's better to keep a copy of your array.

```
import numpy as np
Z = np.zeros(9)
index = [0,1,2]
Z[index] = 1 # store 1 at index 0,1,2
print(Z)
```

If you are unsure that the result of your indexing is a view or a copy, you can check the `base` of your result. If it is `None`, then your result is a copy:

```
import numpy as np
Z = np.random.uniform(0,1,(5,5)) #draws sample from a uniform distribution
Z1 = Z[:3,:]
#print("Z1",Z1)
Z2 = Z[[0,1,2], :]
#print("Z2",Z2)
print(np.allclose(Z1,Z2)) #returns True if two arrays are element-wise equal within a tolerar
print(Z1.base is Z)#return true if memory of Z1 is shared with Z and false otherwise
print(Z2.base is Z)#return true if memory of Z2 is shared with Z and false otherwise
print(Z2.base is None) #return true if meory of Z2 is not shared
```

Here we can see in the above code that `Z1.base is Z` returns true because Z1 shares memory of Z and *Z1 creates a copy of Z*. `Z2.base is Z` returns false because *Z2 provides a view of Z*, hence does not share the memory of Z.

Ravel and Flatten

Some NumPy functions return a view when possible e.g. ravel while some others always return a copy e.g flatten The following example explains the

concept of `ravel` and `flatten` through code:

```
import numpy as np

Z = np.zeros((5,5))
print("Z:\n",Z)
print("Z.ravel().base:\n",Z.ravel().base)
print("Z.ravel().base is Z:",Z.ravel().base is Z) #returns true if memory of Z.ravel() is sha

print("\nZ[::2,::2].ravel():\n",Z[::2,::2].ravel())
print("\nZ[::2,::2].ravel().base is Z:",Z[::2,::2].ravel().base is Z)#returns true if memory

print("\nZ.flatten()\n:",Z.flatten())
print("Z.flatten.base is Z:",Z.flatten().base is Z)#returns true if memory of Z.flatten() is
```

Temporary copy

Copies can be made explicitly like in the previous section, but the most general case is the implicit creation of intermediate copies. This is the case when you are doing some arithmetic with arrays:

```
import numpy as np
X = np.ones(10, dtype=np.int)#create an array X of size 10 containing ones
Y = np.ones(10, dtype=np.int)#create an array Y of size 10 containing ones
A = 2*X + 2*Y #store 2*X + 2*Y in A
print("X:",X)
print("Y:",Y)
print("A=2*X + 2*Y :\nA:",A)
```

In the example above, three intermediate arrays have been created. One for holding the result of `2*X`, one for holding the result of `2*Y` and the last one for holding the result of `2*X+2*Y`. In this specific case, the arrays are small enough and this does not really make a difference. However, if your arrays are big, then you have to be careful with such expressions and wonder if you can do it differently. For example, if only the final result matters and you don't need `X` nor `Y` afterwards, an alternate solution would be:

```
import numpy as np
X = np.ones(10, dtype=np.int) #create an array X of size 10 containing ones
```

```
Y = np.ones(10, dtype=np.int) #create an array Y of size 10 containing ones
print("X:",X,"Y:",Y,"\n np.multiply(X, 2, out=X)")
np.multiply(X, 2, out=X) # multiply X with 2 and store the result in X
print("X:",X,"Y:",Y,"\n np.multiply(Y, 2, out=Y)")
np.multiply(Y, 2, out=Y)# multiply Y with 2 and store the result in Y
print("X:",X,"Y:",Y,"\n np.add(X, Y, out=X)")
np.add(X, Y, out=X)# add X and Y and store the result in X
print("X:",X,"Y:",Y)
```

Using this alternate solution, no temporary array has been created. The problem is that there are many other cases where such copies need to be created and this impacts the performance like demonstrated in the example below:

main.py

tools.py

```
import numpy as np
from tools import timeit

def test(X,Y):
  timeit("Z=X + 2.0*Y", globals()) #time taken by Z=X + 2.0*Y
  timeit("Z = X + 2*Y", globals()) #time taken by Z=X + 2*Y
  timeit("np.add(X, Y, out=X); np.add(X, Y, out=X)", globals())
  #time taken by np.add(X, Y, out=X); np.add(X, Y, out=X)

X = np.ones(100000000, dtype=np.int)
Y = np.ones(100000000, dtype=np.int)
test(X,Y)
```

Solve this Quiz!

1 Does the following code return a view or a copy?

```
Z = np.zeros(9)
Z_1 = Z[:4]
Z_1[...] = 1
```

A) View

B) Copy

COMPLETED 0%

1 of 5

Now that we have learned views and copies, let’s move on to a coding challenge.