Smart Array Programming

In this lesson, we will discuss some smart programming tools for arrays.

WE'LL COVER THE FOLLOWING

- Using conditions on arrays
  - Using multiple conditions
  - `any()` and `all()`
- Iterating elements in arrays
- Vectorizing functions
- Copying arrays

Using conditions on arrays

You can perform comparison statements on arrays, and it will return an array of booleans (`True` and `False` values) for each value in the array. For example, let's create an array and find out which values of the array are below 5:

```
import numpy as np

a = np.arange(10)
print('the total array:', a)
print("array of booleans:", a < 5)
print('values less than 5:', a[a < 5])
```

If we want to replace every value that is less than 5 with 666, we can use the following short syntax:

```
import numpy as np

a = np.arange(10)
print(a)
```

```
a[a < 5] = 666
print(a)
```

Using multiple conditions

Multiple conditions can be given as well. When two conditions both have to be true, use the `&` symbol. When at least one of the conditions needs to be true, use the `|` symbol (that is the vertical bar).

Let's use the `&` symbol:

```
import numpy as np

a = np.arange(20)
print(a)
a[(a > 5) & (a < 10)] = 666
print(a)
```

Since only the values $6, 7, 8$ and 9 satisfy the condition $5 < a$ **and** $a < 10$, they are replaced with 666.

Now, let's use the `|` symbol and notice the difference in the output:

```
import numpy as np

a = np.arange(20)
print(a)
a[(a > 5) | (a < 10)] = 666
print(a)
```

Since all the values satisfy the condition $5 > a$ **or** $a < 10$, they are replaced with 666.

`any()` and `all()`

When using the if condition on arrays, it is useful to use the `.any()` and

`.all()` methods.

- `.any()` returns `True` if the conditional statement is satisfied for some elements of the array.
- `.all()` returns `True` if the conditional statement is satisfied for all the elements of the array.

```
import numpy as np

v = np.linspace(-5, 5, 5)
print(v)

if (v > 0).any():
  print("At leat one element is positive.")
else:
   print("No element is positive.")

if (v > 0).all():
  print("All elements are postive.")
else:
   print("Not all elements are positive.")
```

Iterating elements in arrays

To iterate over elements of an array in Python, a `for` loop is the easiest way to do so.

One-Dimensional Arrays

For vectors, we use a single `for` loop:

```
import numpy as np

v = np.array([1, 2, 3, 4])
for x in v:
  print(x)
```

Multi-Dimensional Arrays

For multidimensional arrays, we use nested loops. The following is an example of iterating over a Matrix:

```
import numpy as np

M = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for x in M:
  for y in x:
    print(y)
```

Vectorizing functions

NumPy has some standard mathematical functions for fast operations on a dataset, known as **vectorization**. It is faster than the iterative approach because the code has a quicker run time.

The first step to converting a scalar algorithm into a vectorized algorithm is to make sure that the functions we write work with vector inputs.

```
import numpy as np

def is_negative(x):
  if x < 0:
    return True
  else:
    return False

v = np.linspace(-5, 5, 5)
print(is_negative(v)) # this will throw an error
```

The function `is_negative` did not work because it was not designed to deal with vector inputs. To fix this, NumPy provides the `vectorize` function which will return a vectorized version of the function.

```
import numpy as np

def is_negative(x):
  if x < 0:
    return True
  else:
    return False

vec_is_negative = np.vectorize(is_negative) #vectorizing the function isNegative
```

```
v = np.linspace(-5, 5, 5)
print(v)
print(vec_is_negative(v))
```

Copying arrays

When using the assignment operator with `ndarrays`, the array is not copied but instead, both refer to the same memory location.

For example, if `x` and `y` are `ndarrays` and we do `x = y`, all the values from `y` are not copied to `x`. Instead, the variable `x` shares a reference to the memory space reserved for the variable `y`.

See the example below:

```
import numpy as np

a = np.arange(6)
print("a =", a)
b = a           # a is passed by reference and b is referring to same data as a
print("b =", b)
b[3] = 666      # changing in b will also change the value in a

print("After replacing value in b")
print("a =", a) # value is replaced in a as well
print("b =", b)
```

To avoid this behavior and make `b` a completely independent object, we use the function `copy`.

```
import numpy as np

a = np.arange(6)
print("a =", a)
b = np.copy(a)       # a is copied in b
print("b =", b)
b[3] = 666      # changing in b will not change a

print("After replacing value in b")
print("a =", a)
print("b =", b)
```

Let’s test your knowledge with a quick quiz in the next lesson.