NumPy Arrays

Learn about NumPy arrays and how they're used.

Chapter Goals:

- Learn about NumPy arrays and how to initialize them
- Write code to create several NumPy arrays

A. Arrays

NumPy arrays are basically just Python lists with added features. In fact, you can easily convert a Python list to a Numpy array using the `np.array` function, which takes in a Python list as its required argument. The function also has quite a few keyword arguments, but the main one to know is `dtype`. The `dtype` keyword argument takes in a NumPy type and manually casts the array to the specified type.

The code below is an example usage of `np.array` to create a 2-D matrix. Note that the array is manually cast to `np.float32`.

```
import numpy as np

arr = np.array([[0, 1, 2], [3, 4, 5]],
               dtype=np.float32)
print(repr(arr))
```

When the elements of a NumPy array are mixed types, then the array's type will be *upcast* to the highest level type. This means that if an array input has mixed `int` and `float` elements, all the integers will be cast to their floating-point equivalents. If an array is mixed with `int`, `float`, and `string` elements, everything is cast to strings.

The code below is an example of `np.array` upcasting. Both integers are cast to

their floating-point equivalents.

```
arr = np.array([0, 0.1, 2])
print(repr(arr))
```

B. Copying

Similar to Python lists, when we make a reference to a NumPy array it doesn't create a different array. Therefore, if we change a value using the reference variable, it changes the original array as well. We get around this by using an array's inherent `copy` function. The function has no required arguments, and it returns the copied array.

In the code example below, `c` is a reference to `a` while `d` is a copy. Therefore, changing `c` leads to the same change in `a`, while changing `d` does not change the value of `b`.

```
a = np.array([0, 1])
b = np.array([9, 8])
c = a
print('Array a: {}'.format(repr(a)))
c[0] = 5
print('Array a: {}'.format(repr(a)))

d = b.copy()
d[0] = 6
print('Array b: {}'.format(repr(b)))
```

C. Casting

We cast NumPy arrays through their inherent `astype` function. The function's required argument is the new type for the array. It returns the array cast to the new type.

The code below shows an example of casting using the `astype` function. The `dtype` property returns the type of an array.

```
arr = np.array([0, 1, 2])
print(arr.dtype)
```

```
arr = arr.astype(np.float32)
print(arr.dtype)
```

D. NaN

When we don't want a NumPy array to contain a value at a particular index, we can use `np.nan` to act as a placeholder. A common usage for `np.nan` is as a filler value for incomplete data.

The code below shows an example usage of `np.nan`. Note that `np.nan` cannot take on an integer type.

```
arr = np.array([np.nan, 1, 2])
print(repr(arr))

arr = np.array([np.nan, 'abc'])
print(repr(arr))

# Will result in a ValueError
np.array([np.nan, 1, 2], dtype=np.int32)
```

E. Infinity

To represent infinity in NumPy, we use the `np.inf` special value. We can also represent negative infinity with `-np.inf`.

The code below shows an example usage of `np.inf`. Note that `np.inf` cannot take on an integer type.

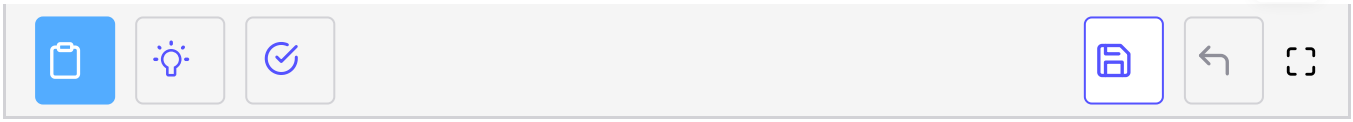
```
print(np.inf > 1000000)

arr = np.array([np.inf, 5])
print(repr(arr))

arr = np.array([-np.inf, 1])
print(repr(arr))

# Will result in an OverflowError
np.array([np.inf, 3], dtype=np.int32)
```

Time to Code!

The first array we'll create comes straight from a list of integers and `np.nan`. The list contains `np.nan` as the first element, and the integers from `2` to `5`, inclusive, as the next four elements.

Set `arr` equal to `np.array` applied to the specified list.

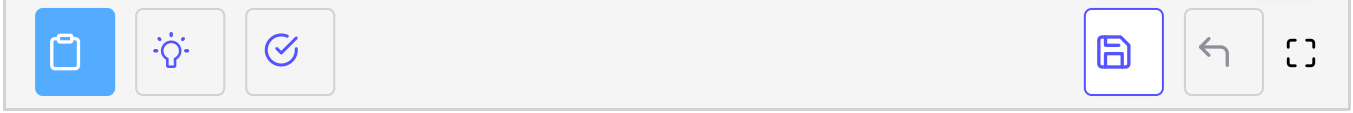
```
# CODE HERE
```



We now want to copy the array so we can change the first element to `10`. This way we don't modify the original array.

Set `arr2` equal to `arr.copy()`, then set the first element of `arr2` equal to `10`.

```
# CODE HERE
```



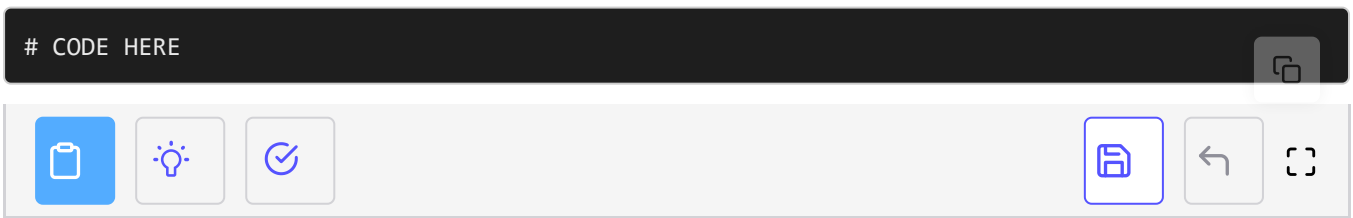
The next two arrays will use floating point numbers. The first array will be upcast to floating point numbers, while we manually cast the second array using `np.float32`.

For manual casting, we use an array's inherent `astype` function, which takes in the new type as an argument and returns the casted array.

Set `float_arr` equal to `np.array` applied to a list with elements `1`, `5.4`, and `3`, in that order.

Set `float_arr2` equal to `arr2.astype`, with argument `np.float32`.

```
# CODE HERE
```



The final array will be a multi-dimensional array, specifically a 2-D matrix. The 2-D matrix will have the integers `1`, `2`, `3` in its first row, and the integers `4`, `5`, `6` in its second row. We'll also manually set its type to `np.float32`.

Set `matrix` equal to `np.array` with a list of lists (representing the specified 2-D matrix) as the first argument, and `np.float32` as the `dtype` keyword argument.

```
# CODE HERE
```