

NumPy Basics

Perform basic operations to create and modify NumPy arrays.

Chapter Goals:

- Learn about some basic NumPy operations
- Write code using the basic NumPy functions

A. Ranged data

While `np.array` can be used to create any array, it is equivalent to hardcoding an array. This won't work when the array has hundreds of values. Instead, NumPy provides an option to create ranged data arrays using `np.arange`. The function acts very similar to the `range` function in Python, and will always return a 1-D array.

The code below contains example usages of `np.arange`.

```
arr = np.arange(5)
print(repr(arr))

arr = np.arange(5.1)
print(repr(arr))

arr = np.arange(-1, 4)
print(repr(arr))

arr = np.arange(-1.5, 4, 2)
print(repr(arr))
```



The output of `np.arange` is specified as follows:

- If only a single number, n , is passed in as an argument, `np.arange` will return an array with all the integers in the range $[0, n)$. **Note:** the lower end is inclusive while the upper end is exclusive.
- For two arguments, m and n , `np.arange` will return an array with all the

integers in the range $[m, n)$.

- For three arguments, m , n , and s , `np.arange` will return an array with the integers in the range $[m, n)$ using a step size of s .
- Like `np.array`, `np.arange` performs upcasting. It also has the `dtype` keyword argument to manually cast the array.

To specify the number of elements in the returned array, rather than the step size, we can use the `np.linspace` function.

This function takes in a required first two arguments, for the start and end of the range, respectively. The end of the range is inclusive for `np.linspace`, unless the keyword argument `endpoint` is set to `False`. To specify the number of elements, we set the `num` keyword argument (its default value is `50`).

The code below shows example usages of `np.linspace`. It also takes in the `dtype` keyword argument for manual casting.

```
arr = np.linspace(5, 11, num=4)
print(repr(arr))

arr = np.linspace(5, 11, num=4, endpoint=False)
print(repr(arr))

arr = np.linspace(5, 11, num=4, dtype=np.int32)
print(repr(arr))
```



B. Reshaping data

The function we use to reshape data in NumPy is `np.reshape`. It takes in an array and a new shape as required arguments. The new shape must exactly contain all the elements from the input array. For example, we could reshape an array with 12 elements to `(4, 3)`, but we can't reshape it to `(4, 4)`.

We are allowed to use the special value of `-1` in at most one dimension of the new shape. The dimension with `-1` will take on the value necessary to allow the new shape to contain all the elements of the array.

The code below shows example usages of `np.reshape`.

```
arr = np.arange(8)

reshaped_arr = np.reshape(arr, (2, 4))

print(repr(reshaped_arr))
print('New shape: {}'.format(reshaped_arr.shape))

reshaped_arr = np.reshape(arr, (-1, 2, 2))
print(repr(reshaped_arr))
print('New shape: {}'.format(reshaped_arr.shape))
```



While the `np.reshape` function can perform any reshaping utilities we need, NumPy provides an inherent function for flattening an array. Flattening an array reshapes it into a 1D array. Since we need to flatten data quite often, it is a useful function.

The code below flattens an array using the inherent `flatten` function.

```
arr = np.arange(8)
arr = np.reshape(arr, (2, 4))
flattened = arr.flatten()
print(repr(arr))
print('arr shape: {}'.format(arr.shape))
print(repr(flattened))
print('flattened shape: {}'.format(flattened.shape))
```



C. Transposing

Similar to how it is common to reshape data, it is also common to transpose data. Perhaps we have data that's supposed to be in a particular format, but some new data we get is rearranged. We can just transpose the data, using the `np.transpose` function, to convert it to the proper format.

The code below shows an example usage of the `np.transpose` function. The matrix rows become columns after the transpose.

```
arr = np.arange(8)
arr = np.reshape(arr, (4, 2))
transposed = np.transpose(arr)
print(repr(arr))
print('arr shape: {}'.format(arr.shape))
print(repr(transposed))
print('transposed shape: {}'.format(transposed.shape))
```





The function takes in a required first argument, which will be the array we want to transpose. It also has a single keyword argument called `axes`, which represents the new *permutation* of the dimensions.

The permutation is a tuple/list of integers, with the same length as the number of dimensions in the array. It tells us where to switch up the dimensions. For example, if the permutation had 3 at index 1, it means the old third dimension of the data becomes the new second dimension (since index 1 represents the second dimension).

The code below shows an example usage of the `np.transpose` function with the `axes` keyword argument. The `shape` property gives us the shape of an array.

```
arr = np.arange(24)
arr = np.reshape(arr, (3, 4, 2))
transposed = np.transpose(arr, axes=(1, 2, 0))
print('arr shape: {}'.format(arr.shape))
print('transposed shape: {}'.format(transposed.shape))
```



In this example, the old first dimension became the new third dimension, the old second dimension became the new first dimension, and the old third dimension became the new second dimension. The default value for `axes` is a dimension reversal (e.g. for 3-D data the default `axes` value is `[2, 1, 0]`).

D. Zeros and ones

Sometimes, we need to create arrays filled solely with 0 or 1. For example, since binary data is labeled with 0 and 1, we may need to create dummy datasets of strictly one label. For creating these arrays, NumPy provides the functions `np.zeros` and `np.ones`. They both take in the same arguments, which includes just one required argument, the array shape. The functions also allow for manual casting using the `dtype` keyword argument.

The code below shows example usages of `np.zeros` and `np.ones`.

```
arr = np.zeros(4)
print(repr(arr))

arr = np.ones((2, 3))
print(repr(arr))

arr = np.ones((2, 3), dtype=np.int32)
print(repr(arr))
```



If we want to create an array of 0's or 1's with the same shape as another array, we can use `np.zeros_like` and `np.ones_like`.

The code below shows example usages of `np.zeros_like` and `np.ones_like`.

```
arr = np.array([[1, 2], [3, 4]])
print(repr(np.zeros_like(arr)))

arr = np.array([[0., 1.], [1.2, 4.]])
print(repr(np.ones_like(arr)))
print(repr(np.ones_like(arr, dtype=np.int32)))
```



Time to Code!

Our initial array will just be all the integers from 0 to 11, inclusive. We'll also reshape it so it has three dimensions.

First, set `arr` equal to `np.arange` with `12` as the only argument.

Then, set `reshaped` equal to `np.reshape` with `arr` as the first argument and `(2, 3, 2)` as the second argument.

CODE HERE

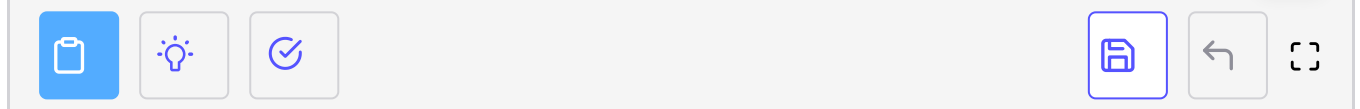


Next we want to get a flattened version of the reshaped array (the flattened version is equivalent to `arr`), as well as a transposed version. For the transposed version of `reshaped`, we use a permutation of `(1, 2, 0)`.

Set `flattened` equal to `reshaped.flatten()`.

Then set `transposed` equal to `np.transpose` with `reshaped` as the first argument and the specified permutation for the `axes` keyword argument.

```
# CODE HERE
```

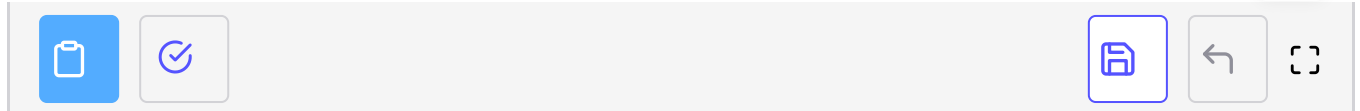


We'll create an array of 5 elements, all of which are `0`. We'll also create an array with the same shape as `transposed`, but containing only `1` as the elements.

Set `zeros_arr` equal to `np.zeros` with `5` as the lone argument.

Then set `ones_arr` equal to `np.ones_like` with `transposed` as the lone argument.

```
# CODE HERE
```



The final array will contain 101 evenly spaced numbers between -3.5 and 1.5, inclusive. Since they are evenly spaced, the difference between adjacent numbers is 0.05.

Set `points` equal to `np.linspace` with `-3.5` and `1.5` as the first two arguments, respectively, as well as `101` for the `num` keyword argument.

```
# CODE HERE
```

