Array slicing and ranges in Python

In Python, data is almost universally represented as NumPy arrays.

In this tutorial, you will discover how to manipulate and access your data correctly in NumPy arrays.

After completing this tutorial, you will know:

- How to convert your list data to NumPy arrays.
- How to access data using Pythonic indexing and slicing.
- How to resize your data to meet the expectations of some machine learning APIs.

1. From List to Arrays

In general, I recommend loading your data from file using Pandas or even NumPy functions.

This section assumes you have loaded or generated your data by other means and it is now represented using Python lists.

Let's look at converting your data in lists to NumPy arrays.

One-Dimensional List to Array

You may load your data or generate your data and have access to it as a list.

You can convert a one-dimensional list of data to an array by calling the array() NumPy function.

```
# one dimensional example

from numpy import array

# list of data

data = [11, 22, 33, 44, 55]

# array of data

data = array(data)

print(data)

print(type(data))
```

Running the example converts the one-dimensional list to a NumPy array.

```
[11 22 33 44 55]
<class 'numpy.ndarray'>
```

Two-Dimensional List of Lists to Array

It is more likely in machine learning that you will have two-dimensional data.

That is a table of data where each row represents a new observation and each column a new feature.

Perhaps you generated the data or loaded it using custom code and now you have a list of lists. Each list represents a new observation.

You can convert your list of lists to a NumPy array the same way as above, by calling the array() function.

```
# two dimensional example
from numpy import array
# list of data
data = [[11, 22],
           [33, 44],
           [55, 66]]
# array of data
data = array(data)
print(data)
print(type(data))
```

Running the example shows the data successfully converted.

```
[[11 22]
 [33 44]
 [55 66]]
<class 'numpy.ndarray'>
```

2. Array Indexing

Once your data is represented using a NumPy array, you can access it using indexing.

Let’s look at some examples of accessing data via indexing.

One-Dimensional Indexing

Generally, indexing works just like you would expect from your experience with other programming languages, like Java, C#, and C++.

For example, you can access elements using the bracket operator [] specifying the zero-offset index for the value to retrieve.

```
# simple indexing

from numpy import array

# define array

data = array([11, 22, 33, 44, 55])

# index data

print(data[0])

print(data[4])
```

Running the example prints the first and last values in the array.

```
11

55
```

Specifying integers too large for the bound of the array will cause an error.

```
# simple indexing

from numpy import array

# define array

data = array([11, 22, 33, 44, 55])

# index data

print(data[5])
```

Running the example prints the following error:

```
IndexError: index 5 is out of bounds for axis 0 with size 5
```

One key difference is that you can use negative indexes to retrieve values offset from the end of the array.

For example, the index -1 refers to the last item in the array. The index -2 returns the second last item all the way back to -5 for the first item in the current example.

```
# simple indexing

from numpy import array

# define array

data = array([11, 22, 33, 44, 55])

# index data

print(data[-1])

print(data[-5])
```

Running the example prints the last and first items in the array.

```
55

11
```

Two-Dimensional Indexing

Indexing two-dimensional data is similar to indexing one-dimensional data, except that a comma is used to separate the index for each dimension.

```
data[0,0]
```

This is different from C-based languages where a separate bracket operator is used for each dimension.

```
data[0][0]
```

For example, we can access the first row and the first column as follows:

```
# 2d indexing

from numpy import array

# define array

data = array([[11, 22], [33, 44], [55, 66]])

# index data

print(data[0,0])
```

Running the example prints the first item in the dataset.

```
11
```

If we are interested in all items in the first row, we could leave the second dimension index empty, for example:

```
# 2d indexing

from numpy import array

# define array

data = array([[11, 22], [33, 44], [55, 66]])

# index data

print(data[0,])
```

This prints the first row of data.

```
[11 22]
```

3. Array Slicing

So far, so good; creating and indexing arrays looks familiar.

Now we come to array slicing, and this is one feature that causes problems for beginners to Python and NumPy arrays.

Structures like lists and NumPy arrays can be sliced. This means that a subsequence of the structure can be indexed and retrieved.

This is most useful in machine learning when specifying input variables and output variables, or splitting training rows from testing rows.

Slicing is specified using the colon operator ':' with a '*from*' and '*to*' index before and after the column respectively. The slice extends from the 'from' index and ends one item before the 'to' index.

```
data[from:to]
```

Let's work through some examples.

One-Dimensional Slicing

You can access all data in an array dimension by specifying the slice ':' with no indexes.

```
# simple slicing

from numpy import array

# define array

data = array([11, 22, 33, 44, 55])

print(data[:])
```

Running the example prints all elements in the array.

```
[11 22 33 44 55]
```

The first item of the array can be sliced by specifying a slice that starts at index 0 and ends at index 1 (one item before the 'to' index).

```
# simple slicing

from numpy import array

# define array

data = array([11, 22, 33, 44, 55])

print(data[0:1])
```

Running the example returns a subarray with the first element.

```
[11]
```

We can also use negative indexes in slices. For example, we can slice the last two items in the list by starting the slice at -2 (the second last item) and not specifying a 'to' index; that takes the slice to the end of the dimension.

```
# simple slicing

from numpy import array

# define array

data = array([11, 22, 33, 44, 55])

print(data[-2:])
```

Running the example returns a subarray with the last two items only.

```
[44 55]
```

Two-Dimensional Slicing

Let's look at the two examples of two-dimensional slicing you are most likely to use in machine learning.

Split Input and Output Features

It is common to split your loaded data into input variables (X) and the output variable (y).

We can do this by slicing all rows and all columns up to, but before the last column, then separately indexing the last column.

For the input features, we can select all rows and all columns except the last one by specifying ':' for in the rows index, and :-1 in the columns index.

```
X = [:, :-1]
```

For the output column, we can select all rows again using ':' and index just the last column by specifying the -1 index.

```
y = [:, -1]
```

Putting all of this together, we can separate a 3-column 2D dataset into input and output data as follows:

```
# split input and output

from numpy import array

# define array

data = array([[11, 22, 33],

           [44, 55, 66],

           [77, 88, 99]])

# separate data

X, y = data[:, :-1], data[:, -1]

print(X)

print(y)
```

Running the example prints the separated X and y elements. Note that X is a 2D array and y is a 1D array.

```
[[11 22]

 [44 55]

 [77 88]]

[33 66 99]
```

Split Train and Test Rows

It is common to split a loaded dataset into separate train and test sets.

This is a splitting of rows where some portion will be used to train the model and the remaining portion will be used to estimate the skill of the trained model.

This would involve slicing all columns by specifying ':' in the second dimension index. The training dataset would be all rows from the beginning to the split point.

```
dataset

train = data[:split, :]
```

The test dataset would be all rows starting from the split point to the end of the dimension.

```
test = data[split:, :]
```

Putting all of this together, we can split the dataset at the contrived split point of 2.

```
# split train and test

from numpy import array

# define array

data = array([[11, 22, 33],

           [44, 55, 66],

           [77, 88, 99]])

# separate data

split = 2

train,test = data[:split,:],data[split:,:]

print(train)

print(test)
```

Running the example selects the first two rows for training and the last row for the test set.

```
[[11 22 33]

[44 55 66]]

[[77 88 99]]
```

4. Array Reshaping

After slicing your data, you may need to reshape it.

For example, some libraries, such as scikit-learn, may require that a one-dimensional array of output variables (y) be shaped as a two-dimensional array with one column and outcomes for each row.

Some algorithms, like the Long Short-Term Memory recurrent neural network in Keras, require input to be specified as a three-dimensional array comprised of samples, timesteps, and features.

It is important to know how to reshape your NumPy arrays so that your data meets the expectation of specific Python libraries. We will look at these two examples.

Data Shape

NumPy arrays have a shape attribute that returns a tuple of the length of each dimension of the array.

For example:

```
# array shape

from numpy import array

# define array

data = array([11, 22, 33, 44, 55])

print(data.shape)
```

Running the example prints a tuple for the one dimension.

```
(5,)
```

A tuple with two lengths is returned for a two-dimensional array.

```
# array shape

from numpy import array

# list of data

data = [[11, 22],

          [33, 44],

          [55, 66]]

# array of data

data = array(data)

print(data.shape)
```

Running the example returns a tuple with the number of rows and columns.

```
(3, 2)
```

You can use the size of your array dimensions in the shape dimension, such as specifying parameters.

The elements of the tuple can be accessed just like an array, with the 0th index for the number of rows and the 1st index for the number of columns. For example:

```
# array shape

from numpy import array

# list of data

data = [[11, 22],

          [33, 44],

          [55, 66]]

# array of data

data = array(data)

print('Rows: %d' % data.shape[0])

print('Cols: %d' % data.shape[1])
```

Running the example accesses the specific size of each dimension.

```
Rows: 3

Cols: 2
```

Reshape 1D to 2D Array

It is common to need to reshape a one-dimensional array into a two-dimensional array with one column and multiple arrays.

NumPy provides the reshape() function on the NumPy array object that can be used to reshape the data.

The reshape() function takes a single argument that specifies the new shape of the array. In the case of reshaping a one-dimensional array into a two-dimensional array with one column, the tuple would be the shape of the array as the first dimension (data.shape[0]) and 1 for the second dimension.

```
data = data.reshape((data.shape[0], 1))
```

Putting this all together, we get the following worked example.

```
# reshape 1D array

from numpy import array

from numpy import reshape

# define array

data = array([11, 22, 33, 44, 55])

print(data.shape)

# reshape

data = data.reshape((data.shape[0], 1))

print(data.shape)
```

Running the example prints the shape of the one-dimensional array, reshapes the array to have 5 rows with 1 column, then prints this new shape.

```
(5,)

(5, 1)
```

Reshape 2D to 3D Array

It is common to need to reshape two-dimensional data where each row represents a sequence into a three-dimensional array for algorithms that expect multiple samples of one or more time steps and one or more features.

A good example is the LSTM recurrent neural network model in the Keras deep learning library.
The reshape function can be used directly, specifying the new dimensionality. This is clear with an example where each sequence has multiple time steps with one observation (feature) at each time step.

We can use the sizes in the shape attribute on the array to specify the number of samples (rows) and columns (time steps) and fix the number of features at 1.

```
data.reshape((data.shape[0], data.shape[1], 1))
```

Putting this all together, we get the following worked example.

```
# reshape 2D array

from numpy import array

# list of data

data = [[11, 22],

          [33, 44],

          [55, 66]]

# array of data

data = array(data)

print(data.shape)

# reshape

data = data.reshape((data.shape[0], data.shape[1], 1))

print(data.shape)
```

Running the example first prints the size of each dimension in the 2D array, reshapes the array, then summarizes the shape of the new 3D array.

```
(3, 2)

(3, 2, 1)
```