Saving and sharing your NumPy arrays

Contents

What you'll learn

You'll save your NumPy arrays as zipped files and human-readable comma-delimited files i.e. *.csv. You will also learn to load both of these file types back into NumPy workspaces.

What you'll do

You'll learn two ways of saving and reading files–as compressed and as text files–that will serve most of your storage needs in NumPy.

- You'll create two 1D arrays and one 2D array
- You'll save these arrays to files
- You'll remove variables from your workspace
- You'll load the variables from your saved file
- You'll compare zipped binary files to human-readable delimited files
- You'll finish with the skills of saving, loading, and sharing NumPy arrays

Skip to main content

What you'll need

- NumPy
- read-write access to your working directory

Load the necessary functions using the following command.

```
import numpy as np
```

In this tutorial, you will use the following Python, IPython magic, and NumPy functions:

- `np.arange`
- `np.savez`
- `del`
- `whos`
- `np.load`
- `np.block`
- `np.newaxis`
- `np.savetxt`
- `np.loadtxt`

Create your arrays

Now that you have imported the NumPy library, you can make a couple of arrays; let's start with two 1D arrays, `x` and `y`, where `y = x**2`.You will assign `x` to the integers from 0 to 9 using `np.arange`.

```
x = np.arange(10)
y = x ** 2
print(x)
print(y)
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 0  1  4  9 16 25 36 49 64 81]
```

Save your arrays with NumPy's `savez`

Now you have two arrays in your workspace,

`x: [0 1 2 3 4 5 6 7 8 9]`

Skip to main content

The first thing you will do is save them to a file as zipped arrays using `savez`. You will use two options to label the arrays in the file,

1. `x_axis = x`: this option is assigning the name `x_axis` to the variable `x`
2. `y_axis = y`: this option is assigning the name `y_axis` to the variable `y`

```
np.savez("x_y-squared.npz", x_axis=x, y_axis=y)
```

Remove the saved arrays and load them back with NumPy's `load`

In your current working directory, you should have a new file with the name `x_y-squared.npz`. This file is a zipped binary of the two arrays, `x` and `y`. Let's clear the workspace and load the values back in. This `x_y-squared.npz` file contains two NPY format files. The NPY format is a native binary format. You cannot read the numbers in a standard text editor or spreadsheet.

1. remove `x` and `y` from the workspaec with `del`
2. load the arrays into the workspace in a dictionary with `np.load`

To see what variables are in the workspace, use the Jupyter/IPython "magic" command `whos`.

```
del x, y
```

```
%whos
```

```
Variable   Type      Data/Info
------------------------------
np         module    <module 'numpy' from '/ho<...>kages/numpy/__init__.py'>
```

```
load_xy = np.load("x_y-squared.npz")

print(load_xy.files)
```

```
['x_axis', 'y_axis']
```

```
whos
```

```
Variable   Type       Data/Info
-------------------------------
load_xy    NpzFile    NpzFile 'x_y-squared.npz'<...>with keys: x_axis, y_axis
np         module     <module 'numpy' from '/ho<...>kages/numpy/__init__.py'>
```

Skip to main content

Reassign the NpzFile arrays to x and y

You've now created the dictionary with an `NpzFile`-type. The included files are `x_axis` and `y_axis` that you defined in your `savez` command. You can reassign `x` and `y` to the `load_xy` files.

```
x = load_xy["x_axis"]
y = load_xy["y_axis"]
print(x)
print(y)
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 0  1  4  9 16 25 36 49 64 81]
```

Success

You have created, saved, deleted, and loaded the variables `x` and `y` using `savez` and `load`. Nice work.

Another option: saving to human-readable csv

Let's consider another scenario, you want to share `x` and `y` with other people or other programs. You may need human-readable text file that is easier to share. Next, you use the `savetxt` to save `x` and `y` in a comma separated value file, `x_y-squared.csv`. The resulting csv is composed of ASCII characters. You can load the file back into NumPy or read it with other programs.

Rearrange the data into a single 2D array

First, you have to create a single 2D array from your two 1D arrays. The csv-filetype is a spreadsheet-style dataset. The csv arranges numbers in rows–separated by new lines–and columns–separated by commas. If the data is more complex e.g. multiple 2D arrays or higher dimensional arrays, it is better to use `savez`. Here, you use two NumPy functions to format the data:

1. `np.block`: this function appends arrays together into a 2D array
2. `np.newaxis`: this function forces the 1D array into a 2D column vector with 10 rows and 1 column.

```
array_out = np.block([x[:, np.newaxis], y[:, np.newaxis]])
print("the output array has shape ", array_out.shape, " with values:")
print(array_out)
```

```
the output array has shape  (10, 2)  with values:
[[ 0  0]
 [ 1  1]
```

Skip to main content

```
 [ 4 16]
 [ 5 25]
 [ 6 36]
 [ 7 49]
 [ 8 64]
 [ 9 81]]
```

Save the data to csv file using `savetxt`

You use `savetxt` with a three options to make your file easier to read:

- `X = array_out`: this option tells `savetxt` to save your 2D array, `array_out`, to the file `x_y-squared.csv`
- `header = 'x, y'`: this option writes a header before any data that labels the columns of the csv
- `delimiter = ','`: this option tells `savetxt` to place a comma between each column in the file

```
np.savetxt("x_y-squared.csv", X=array_out, header="x, y", delimiter=",")
```

Open the file, `x_y-squared.csv`, and you'll see the following:

```
# x, y
0.000000000000000000e+00,0.000000000000000000e+00
1.000000000000000000e+00,1.000000000000000000e+00
2.000000000000000000e+00,4.000000000000000000e+00
3.000000000000000000e+00,9.000000000000000000e+00
4.000000000000000000e+00,1.600000000000000000e+01
5.000000000000000000e+00,2.500000000000000000e+01
6.000000000000000000e+00,3.600000000000000000e+01
7.000000000000000000e+00,4.900000000000000000e+01
8.000000000000000000e+00,6.400000000000000000e+01
9.000000000000000000e+00,8.100000000000000000e+01
```

Our arrays as a csv file

There are two features that you shoud notice here:

1. NumPy uses `#` to ignore headings when using `loadtxt`. If you're using `loadtxt` with other csv files, you can skip header rows with `skiprows = <number_of_header_lines>`.
2. The integers were written in scientific notation. *You can* specify the format of the text using the `savetxt` option, `fmt =`, but it will still be written with ASCII characters. In general, you cannot preserve the type of ASCII numbers as `float` or `int`.

Now, delete `x` and `y` again and assign them to your columns in `x-y_squared.csv`.

```
del x, y
```

```
load_xy = np.loadtxt("x_y-squared.csv", delimiter=",")
```

Skip to main content

```
load_xy.shape
```

```
(10, 2)
```

```
x = load_xy[:, 0]
y = load_xy[:, 1]
print(x)
print(y)
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
[ 0.  1.  4.  9. 16. 25. 36. 49. 64. 81.]
```

Success, but remember your types

When you saved the arrays to the csv file, you did not preserve the `int` type. When loading the arrays back into your workspace the default process will be to load the csv file as a 2D floating point array e.g. `load_xy.dtype == 'float64'` and `load_xy.shape == (10, 2)`.

Wrapping up

In conclusion, you can create, save, and load arrays in NumPy. Saving arrays makes sharing your work and collaboration much easier. There are other ways Python can save data to files, such as pickle, but `savez` and `savetxt` will serve most of your storage needs for future NumPy work and sharing with other people, respectively.

Next steps: you can import data with missing values from Importing with genfromtext or learn more about general NumPy IO with Reading and Writing Files.