Python Tutorial 1

January 12, 2020

This tutorial is for Prof. Xin Tong's DSO 530 class at the University of Southern California in spring 2020. It aims to help you have an idea of the basic statistical elements and implementation in python. It focuses on the detail about the functionality of Numpy and also mentions the function definition of python, the essential libraries, and tools like scikit-learn, Scipy, pandas and matplotlib.

1 Python

1.1 Why Python?

Python has become the lingua franca for many data science applications. It combines the power of general-purpose programming languages with the ease of use of domain-specific scripting languages like MATLAB or R. Python has libraries for data loading, visualization, statistics, natural language processing, image processing, and more. This vast toolbox provides data scientists with a large array of general- and special-purpose functionality. One of the main advantages of using Python is the ability to interact directly with the code, using a terminal or other tools like the Jupyter Notebook, which we'll look at shortly. Machine learning and data analysis are fundamentally iterative processes, in which the data drives the analysis. It is essential for these processes to have tools that allow quick iteration and easy interaction.

As a general-purpose programming language, Python also allows for the creation of complex graphical user interfaces (GUIs) and web services, and for integration into existing systems.

1.2 Supplementary Knowledge of Python

In DSO 545, you have learned about the basic knowledge about python, like variables, lists, loops, dictionaries, etc. Here we will tell you how to define the functions.

Python functions are defined using the def keyword. For example:

[1]:

```python
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
```

```
negative
zero
positive
```

We will often define functions to take optional keyword arguments, like this:

[2]:
```
def hello(name, loud=False):
    if loud:
        print('HELLO, '+ name.upper() + '!' )
    else:
        print('Hello, ' + name + '!')

hello('Bob')
```

```
Hello, Bob!
```

[3]:
```
hello('Fred', loud=True)
```

```
HELLO, FRED!
```

There is a lot more information about Python functions in the following python official documentation website: https://docs.python.org/3.7/tutorial/controlflow.html#defining-functions

2 scikit-learn

scikit-learn is an open-source project, meaning that it is free to use and distribute, and anyone can easily obtain the source code to see what is going on behind the scenes. The scikit-learn project is constantly being developed and improved, and it has a very active user community. It contains a number of state-of-the-art machine learning algorithms, as well as comprehensive documentation about each algorithm. scikit-learn is a very popular tool and the most prominent Python library for machine learning. It is widely used in industry and academia, and a wealth of tutorials and code snippets are available online. scikit-learn works well with a number of other scientific Python tools, which we will discuss later in this chapter.

While reading this, we recommend that you also browse the scikit-learn user guide and API documentation for additional details on and many more options for each algorithm. The online documentation is very thorough, and this tutorial will provide you with all the prerequisites in machine learning to understand it in detail.

2.1 Installing scikit-learn

scikit-learn depends on two other Python packages, NumPy and SciPy. For plotting and interactive development, you should also install matplotlib, IPython, and the Jupyter Notebook. We recommend using the following prepackaged Python distributions, which will provide the necessary packages:

Anaconda

A Python distribution made for large-scale data processing, predictive analytics, and scientific computing. Anaconda comes with NumPy, SciPy, matplotlib, pandas, IPython, Jupyter Notebook, and scikit-learn. Available on Mac OS, Windows, and Linux, it is a very convenient solution and is the one we suggest for people without an existing installation of the scientific Python packages. Anaconda now also includes the commercial Intel MKL library for free. Using MKL (which is done automatically when Anaconda is installed) can give significant speed improvements for many algorithms in scikit-learn.

You can download and install Anaconda (Python 3.7 version) via the official website following the installation documentation in the website:https://www.anaconda.com/distribution/#download-section

If you already have a Python installation set up, you can use pip to install all of these packages in 'Anaconda Prompt':

$ pip install numpy scipy matplotlib ipython scikit-learn pandas

3 Essential Libraries and Tools

Understanding what scikit-learn is and how to use it is important, but there are a few other libraries that will enhance your experience. scikit-learn is built on top of the NumPy and SciPy scientific Python libraries. In addition to NumPy and SciPy, we will be using pandas and matplotlib. Briefly, here is what you should know about these tools in order to get the most out of scikit-learn.

3.1 Numpy

NumPy is one of the fundamental packages for scientific computing in Python. It contains functionality for multidimensional arrays, high-level mathematical functions such as linear algebra operations and the Fourier transform, and pseudorandom number generators. In scikit-learn, the NumPy array is the fundamental data structure.

scikit-learn takes in data in the form of NumPy arrays. Any data you're using will have to be converted to a NumPy array. The core functionality of NumPy is the ndarray class, a multidimensional (n-dimensional) array. All elements of the array must be of the same type. A NumPy array looks like this:

3.1.1 Creating ndarrays

The easiest way to create an array is to use the array function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion:

[4]:

```python
import numpy as np

data1 = [6, 7.5, 8, 0, 1]

arr1 = np.array(data1)

arr1
```

[4]:

```
array([6. , 7.5, 8. , 0. , 1. ])
```

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

[5]:
```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

arr2 = np.array(data2)

arr2
```

[5]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Since data2 was a list of lists, the NumPy array arr2 has two dimensions with shape inferred from the data. We can confirm this by inspecting the ndim and shape attributes:

[6]:
```
arr2.ndim
```

[6]:
```
2
```

[7]:
```
arr2.shape
```

[7]:
```
(2, 4)
```

Unless explicitly specified (more on this later), np.array tries to infer a good data type for the array that it creates. The data type is stored in a special dtype metadata object; for example, in the previous two examples we have:

[8]:
```
arr1.dtype
```

[8]:
```
dtype('float64')
```

[9]:
```
arr2.dtype
```

[9]:
```
dtype('int32')
```

In addition to *np.array*, there are a number of other functions for creating new arrays. As examples, *zeros* and *ones* create arrays of 0s or 1s, respectively, with a given length or shape. *empty* creates an array without initializing its values to any particular value. To create a higher-dimensional array with these methods, pass a tuple for the shape:

[10]:
```
np.zeros(10)
```

[10]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

[11]:
```
np.zeros((3, 6))
```

[11]:
```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

[12]:
```
np.empty((2, 3, 3))
```

```
[12]: array([[[0., 0., 0.],
              [0., 0., 0.],
              [0., 0., 0.]],

             [[0., 0., 0.],
              [0., 0., 0.],
              [0., 0., 0.]]])
```

P.S. It's not safe to assume that *np.empty* will return an array of all zeros. In some cases, it may return uninitialized "garbage" values. For example:

```
[13]: np.empty((2, 3, 4))
```

```
[13]: array([[[1.28222473e-311, 1.28222473e-311, 1.28221236e-311,
               1.28221236e-311],
              [1.28222467e-311, 1.28222467e-311, 1.28222467e-311,
               1.28222467e-311],
              [1.28222467e-311, 1.28222473e-311, 1.28221230e-311,
               1.28222558e-311]],

             [[1.28222467e-311, 1.28221393e-311, 1.28222558e-311,
               1.28221237e-311],
              [1.28220991e-311, 1.28220978e-311, 1.28221206e-311,
               1.28222473e-311],
              [1.28222473e-311, 1.28221207e-311, 1.28221383e-311,
               1.28221384e-311]]])
```

arange is an array-valued version of the built-in Python range function:

```
[14]: np.arange(15)
```

```
[14]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

See Table 3-1 for a shortlist of standard array creation functions. Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be float64 (floating point).

Function	Description
array	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
asarray	Convert input to ndarray, but do not copy if the input is already an ndarray
arange	Like the built-in range but returns an ndarray instead of a list
ones, ones_like	Produce an array of all 1s with the given shape and dtype; ones_like takes another array and produces a ones array of the same shape and dtype
zeros, zeros_like	Like ones and ones_like but producing arrays of 0s instead

Function	Description
empty, empty_like	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
full, full_like	Produce an array of the given shape and dtype with all values set to the indicated “ill value” full_like takes another array and produces a illed array of the same shape and dtype
eye, identity	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

Table 3-1. Array creation functions

3.1.2 Data Types for ndarrays

The data type or dtype is a special object containing the information (or metadata, data about data) the ndarray needs to interpret a chunk of memory as a particular type of data:

[15]:
```
arr1 = np.array([1, 2, 3], dtype=np.float64)

arr2 = np.array([1, 2, 3], dtype=np.int32)

arr1.dtype
```

[15]:
```
dtype('float64')
```

[16]:
```
arr2.dtype
```

[16]:
```
dtype('int32')
```

dtypes are a source of NumPy’s flexibility for interacting with data coming from other systems. In most cases, they provide a mapping directly onto an underlying disk or memory representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like float or int, followed by a number indicating the number of bits per element. A standard double-precision floating-point value (what’s used under the hood in Python’s float object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as float64. See Table 3-2 for a full listing of NumPy’s supported data types.

Type	Type code	Description
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 64-bit integer types
float16	f2	Half-precision loating point

Type	Type code	Description
float32	f4 or f	Standard single-precision floating point; compatible with C float
float64	f8 or d	Standard double-precision floating point; compatible with C double and Python float object
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type; a value can be any Python object
string_	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'
unicode_	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')

Table 3-2. NumPy data types

P.S. Don't worry about memorizing the NumPy dtypes, especially if you're a new user. It's often only necessary to care about the general kind of data you're dealing with, whether floating point, complex, integer, boolean, string, or general Python object. When you need more control over how data are stored in memory and on disk, especially large datasets, it is good to know that you have control over the storage type.

You can explicitly convert or cast an array from one dtype to another using ndarray's astype method:

[17]:
```
arr = np.array([1, 2, 3, 4, 5])

arr.dtype
```

[17]:
```
dtype('int32')
```

P.S. The above output may be different from this because the default type varies according to your python version.

```
[18]: float_arr = arr.astype(np.float64)

      float_arr.dtype
```

```
[18]: dtype('float64')
```

In this example, integers were cast to floating point. If I cast some floating-point numbers to be of integer dtype, the decimal part will be truncated:

```
[19]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

      arr
```

```
[19]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
```

```
[20]: arr.astype(np.int32)
```

```
[20]: array([ 3, -1, -2,  0, 12, 10])
```

If you have an array of strings representing numbers, you can use astype to convert them to numeric form:

```
[21]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

      numeric_strings.astype(float)
```

```
[21]: array([ 1.25, -9.6 , 42.  ])
```

P.S. It’s important to be cautious when using the numpy.string_ type, as string data in NumPy is fixed size and may truncate input without warning. pandas has more intuitive out-of-the-box behavior on non-numeric data.

If casting were to fail for some reason (like a string that cannot be converted to float64), a ValueError will be raised. Here I was a bit lazy and wrote float instead of np.float64; NumPy aliases the Python types to its own equivalent data dtypes.

You can also use another array’s dtype attribute:

```
[22]: int_array = np.arange(10)

      calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

      int_array.astype(calibers.dtype)
```

```
[22]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

There are shorthand type code strings you can also use to refer to a dtype:

[23]:

```
empty_uint32 = np.empty(8, dtype='u4')
# empty creates an array without initializing its values to any particular␣
↪value, using type code to refer to a dtype

empty_uint32
```

[23]:

```
array([         0, 1075314688,          0, 1075707904,          0,
       1075838976,          0, 1072693248], dtype=uint32)
```

Calling astype always creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.

3.1.3 Arithmetic with NumPy Arrays

Arrays are important because they enable you to express batch operations on data without writing any for loops. NumPy users call this vectorization. Any arithmetic operation between equal-size arrays applies the operation element-wise:

[24]:

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])

arr
```

[24]:

```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

[25]:

```
arr + arr
```

[25]:

```
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]])
```

[26]:

```
arr - arr
```

[26]:

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Arithmetic operations with scalars propagate the scalar argument to each element in the array:

[27]:

```
1 / arr
```

[27]:

```
array([[1.        , 0.5       , 0.33333333],
       [0.25      , 0.2       , 0.16666667]])
```

[28]:

```
arr * 0.5
```

[28]:

```
array([[0.5, 1. , 1.5],
       [2. , 2.5, 3. ]])
```

[29]:

```
arr ** 2
```

```
[29]: array([[ 1.,  4.,  9.],
             [16., 25., 36.]])
```

* is elementwise multiplication, not matrix multiplication. We instead use the 'dot' function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

```
[30]: v = np.array([9,10])
      w = np.array([11, 12])
      print('v = \n',v,'\n')
      print('w = \n',w,'\n')
```

```
v =
 [ 9 10]

w =
 [11 12]
```

```
[31]: np.dot(v, w)  # or you can write 'v.dot(w)':9*11 + 10*12 = 219
```

```
[31]: 219
```

```
[32]: x = np.array([[1,2],[3,4]])
      y = np.array([[5,6],[7,8]])
      print('x = \n',x,'\n')
      print('y = \n',y,'\n')
```

```
x =
 [[1 2]
 [3 4]]

y =
 [[5 6]
 [7 8]]
```

```
[33]: np.dot(x, y)  # or you can write 'x.dot(y)': 1*5 + 2*7 = 19; 1*6 + 2*8 = 22;␣
       ↪3*5 +4*7 = 43 ; 3*6 +4*8 = 50
```

```
[33]: array([[19, 22],
             [43, 50]])
```

Comparisons between arrays of the same size yield boolean arrays:

```
[34]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

      arr2
```

```
[34]: array([[ 0.,  4.,  1.],
              [ 7.,  2., 12.]])
```

```
[35]: arr2 > arr
```

```
[35]: array([[False,  True, False],
              [ True, False,  True]])
```

3.1.4 Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
[36]: arr = np.arange(10)

      arr
```

```
[36]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
[37]: arr[5]
```

```
[37]: 5
```

```
[38]: arr[5:8]
```

```
[38]: array([5, 6, 7])
```

```
[39]: arr[5:8] = 12

      arr
```

```
[39]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

As you can see, if you assign a scalar value to a slice, as in arr[5:8] = 12, the value is propagated (or broadcasted henceforth) to the entire selection. An important first distinction from Python's built-in lists is that array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

To give an example of this, I first create a slice of arr:

```
[40]: arr_slice = arr[5:8]

      arr_slice
```

```
[40]: array([12, 12, 12])
```

Now, when I change values in arr_slice, the mutations are reflected in the original array arr:

[41]:
```
arr_slice[1] = 12345

arr
```

[41]:
```
array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
           9])
```

The “bare” slice [:] will assign to all values in an array:

[42]:
```
arr_slice[:] = 64

arr
```

[42]:
```
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

If you are new to NumPy, you might be surprised by this, especially if you have used other array programming languages that copy data more eagerly. As NumPy has been designed to be able to work with very large arrays, you could imagine performance and memory problems if NumPy insisted on always copying data.

P.S. If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array—for example, arr[5:8].copy().

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

[43]:
```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

arr2d[2]
```

[43]:
```
array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent:

[44]:
```
arr2d[0][2]
```

[44]:
```
3
```

[45]:
```
arr2d[0, 2]
```

[45]:
```
3
```

See Figure 3-1 for an illustration of indexing on a two-dimensional array. I find it helpful to think of axis 0 as the “rows” of the array and axis 1 as the “columns.”

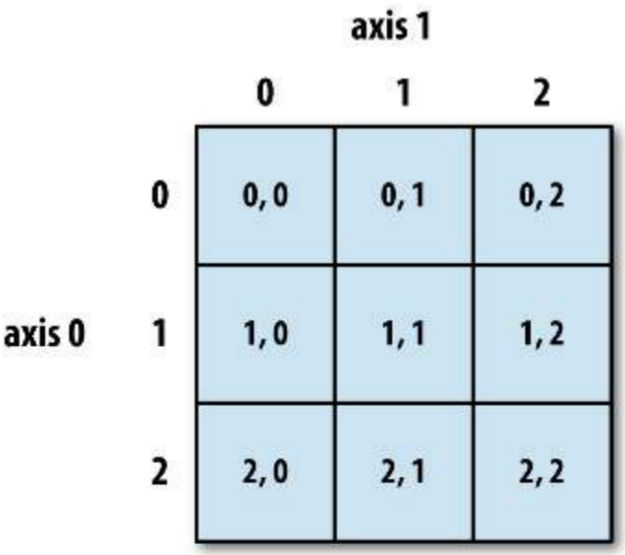


Figure 3-1. Indexing elements in a NumPy array

In multidimensional arrays, if you omit later indices, the returned object will be a lower-dimensional ndarray consisting of all the data along the higher dimensions. So in the 2 × 2 × 3 array arr3d:

```
[46]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

      arr3d
```

```
[46]: array([[[ 1,  2,  3],
              [ 4,  5,  6]],

             [[ 7,  8,  9],
              [10, 11, 12]]])
```

arr3d[0] is a 2 × 3 array:

```
[47]: arr3d[0]
```

```
[47]: array([[1, 2, 3],
             [4, 5, 6]])
```

Both scalar values and arrays can be assigned to *arr3d[0]*:

```
[48]: old_values = arr3d[0].copy()

      arr3d[0] = 42

      arr3d
```

```
[48]: array([[[42, 42, 42],
              [42, 42, 42]],
             [[ 7,  8,  9],
              [10, 11, 12]]])
```

```
[49]: arr3d[0] = old_values

      arr3d
```

```
[49]: array([[[ 1,  2,  3],
              [ 4,  5,  6]],

             [[ 7,  8,  9],
              [10, 11, 12]]])
```

Similarly, textitarr3d[1, 0] gives you all of the values whose indices start with (1, 0), forming a 1-dimensional array:

```
[50]: arr3d[1, 0]
```

```
[50]: array([7, 8, 9])
```

This expression is the same as though we had indexed in two steps:

```
[51]: x = arr3d[1]

      x
```

```
[51]: array([[ 7,  8,  9],
             [10, 11, 12]])
```

```
[52]: x[0]
```

```
[52]: array([7, 8, 9])
```

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

3.1.5 Indexing with slices

Like one-dimensional objects such as Python lists, ndarrays can be sliced with the familiar syntax:

```
[53]: arr
```

```
[53]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

```
[54]: arr[1:6]
```

```
[54]: array([ 1,  2,  3,  4, 64])
```

Consider the two-dimensional array from before, *arr2d*. Slicing this array is a bit different:

```
[55]: arr2d
```

```
[55]: array([[1, 2, 3],
             [4, 5, 6],
             [7, 8, 9]])
```

```
[56]: arr2d[:2]
```

```
[56]: array([[1, 2, 3],
             [4, 5, 6]])
```

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. It can be helpful to read the expression *arr2d[:2]* as "select the first two rows of *arr2d*."

You can pass multiple slices just like you can pass multiple indexes:

```
[57]: arr2d[:2, 1:]
```

```
[57]: array([[2, 3],
             [5, 6]])
```

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower-dimensional slices. For example, I can select the second row but only the first two columns like so:

```
[58]: arr2d[1, :2]
```

```
[58]: array([4, 5])
```

Similarly, I can select the third column but only the first two rows like so:

```
[59]: arr2d[:2, 2]
```

```
[59]: array([3, 6])
```

See Figure 3-2 for an illustration. Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

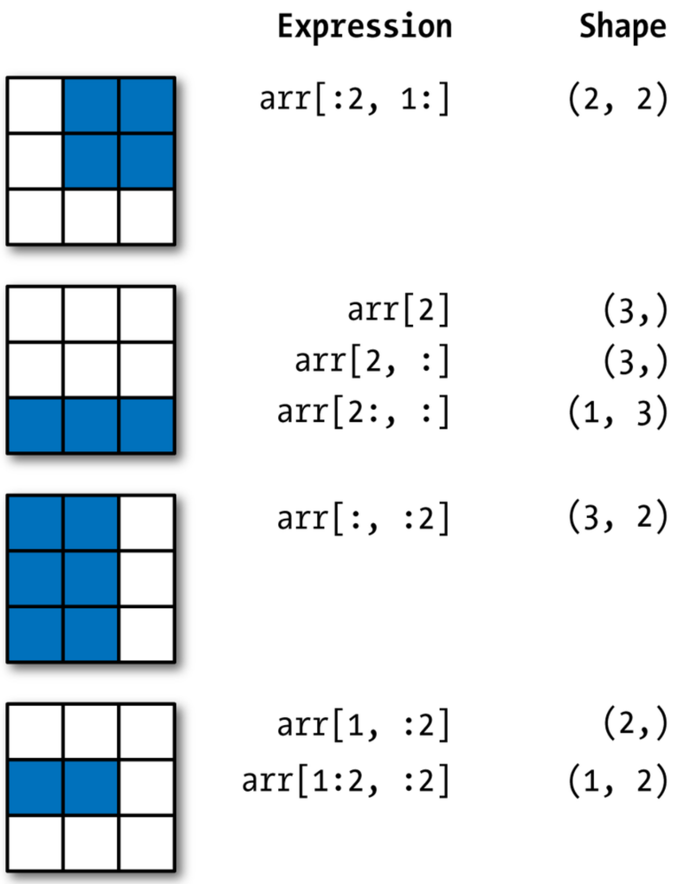


Figure 3-2. Two-dimensional array slicing

3.1.6 Boolean Indexing

Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

[60]:
```
a = np.array([[1,2], [3, 4], [5, 6]])

a
```

[60]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Find the elements of a that are bigger than 2: this returns a numpy array of Booleans of the same shape as a, where each slot of bool_idx tells whether that element of a is > 2.

[61]:
```
bool_idx = (a > 2)

bool_idx
```

```
[61]: array([[False, False],
             [ True,  True],
             [ True,  True]])
```

We use boolean array indexing to construct a rank 1 array consisting of the elements of *a* corresponding to the True values of bool_idx:

```
[62]: a[bool_idx]
```

```
[62]: array([3, 4, 5, 6])
```

We can do all of the above in a single concise statement

```
[63]: a[a > 2]
```

```
[63]: array([3, 4, 5, 6])
```

```
[64]: a[(a==1)|(a==3)]
```

```
[64]: array([1, 3])
```

The boolean array must be of the same length as the array axis it's indexing. You can even mix and match boolean arrays with slices or integers, here is another example and I'm going to use here the randn function in numpy.random to generate some random normally distributed data of mean 0 and variance 1:

```
[65]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

      data = np.random.randn(7, 4)

      names
```

```
[65]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

```
[66]: data
```

```
[66]: array([[ 0.19796297,  1.42372147,  0.85991924,  1.20405872],
             [ 0.79487202, -1.49926438,  0.49605341,  0.17600198],
             [-0.546843  , -0.39666248, -0.68661135, -0.75247445],
             [-0.14268452,  0.30237997,  1.42121026, -0.57963762],
             [ 1.62366709, -1.51326013,  0.52220035,  1.94150386],
             [-0.4605315 , -0.09918563,  1.25007941,  1.06917685],
             [-0.35489868, -0.52299115,  1.35288367,  1.34837318]])
```

```
[67]: names == 'Bob'
```

```
[67]: array([ True, False, False,  True, False, False, False])
```

```
[68]: data[names == 'Bob']
```

```
[68]: array([[ 0.19796297,  1.42372147,  0.85991924,  1.20405872],
              [-0.14268452,  0.30237997,  1.42121026, -0.57963762]])
```

```
[69]: data[names == 'Bob', 2:]
```

```
[69]: array([[ 0.85991924,  1.20405872],
              [ 1.42121026, -0.57963762]])
```

```
[70]: data[names != 'Bob']
      # '!' means 'not'
      # '!=' means 'not equal to'
```

```
[70]: array([[ 0.79487202, -1.49926438,  0.49605341,  0.17600198],
              [-0.546843  , -0.39666248, -0.68661135, -0.75247445],
              [ 1.62366709, -1.51326013,  0.52220035,  1.94150386],
              [-0.4605315 , -0.09918563,  1.25007941,  1.06917685],
              [-0.35489868, -0.52299115,  1.35288367,  1.34837318]])
```

The ~ operator can be useful when you want to invert a general condition:

```
[71]: data[~(names == 'Bob')]
```

```
[71]: array([[ 0.79487202, -1.49926438,  0.49605341,  0.17600198],
              [-0.546843  , -0.39666248, -0.68661135, -0.75247445],
              [ 1.62366709, -1.51326013,  0.52220035,  1.94150386],
              [-0.4605315 , -0.09918563,  1.25007941,  1.06917685],
              [-0.35489868, -0.52299115,  1.35288367,  1.34837318]])
```

3.1.7 Fancy Indexing

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had an 8 × 4 array:

```
[72]: arr = np.empty((8, 4))
```

```
[73]: for i in range(8):
          arr[i] = i

      arr
```

```
[73]: array([[0., 0., 0., 0.],
              [1., 1., 1., 1.],
              [2., 2., 2., 2.],
              [3., 3., 3., 3.],
              [4., 4., 4., 4.],
              [5., 5., 5., 5.],
              [6., 6., 6., 6.],
              [7., 7., 7., 7.]])
```

To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
[74]: arr[[6, 3, 0, 2]]
```

```
[74]: array([[6., 6., 6., 6.],
             [3., 3., 3., 3.],
             [0., 0., 0., 0.],
             [2., 2., 2., 2.]])
```

Hopefully this code did what you expected! Using negative indices selects rows from the end:

```
[75]: arr[[-1, -3, -5]]
```

```
[75]: array([[7., 7., 7., 7.],
             [5., 5., 5., 5.],
             [3., 3., 3., 3.]])
```

Passing multiple index arrays does something slightly different; it selects a one-dimensional array of elements corresponding to each tuple of indices:

```
[76]: arr = np.arange(32).reshape((8, 4))

      arr
```

```
[76]: array([[ 0,  1,  2,  3],
             [ 4,  5,  6,  7],
             [ 8,  9, 10, 11],
             [12, 13, 14, 15],
             [16, 17, 18, 19],
             [20, 21, 22, 23],
             [24, 25, 26, 27],
             [28, 29, 30, 31]])
```

```
[77]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

```
[77]: array([ 4, 23, 29, 10])
```

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

3.1.8 Transposing Arrays and Swapping Axes

Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything. Arrays have the *transpose* method and also the special T attribute:

[78]:
```
arr = np.arange(15).reshape((3, 5))

arr
```

[78]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

[79]:
```
arr.T
```

[79]:
```
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

For higher dimensional arrays, *transpose* will accept a tuple of axis numbers to permute the axes (for extra mind bending):

[80]:
```
arr = np.arange(16).reshape((2, 2, 4))

arr
```

[80]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

[81]:
```
arr.transpose((1, 0, 2))    # change the axes order from 0,1,2 to 1,0,2 :␣
 ↪arr[x][y][z]=arr[y][x][z]
```

[81]:
```
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Here, the axes have been reordered with the second axis first, the first axis second, and the last axis unchanged.

Simple transposing with .T is a special case of swapping axes. ndarray has the method *swapaxes*, which takes a pair of axis numbers and switches the indicated axes to rearrange the data:

[82]:
```
arr
```

[82]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
        [[ 8,  9, 10, 11],
         [12, 13, 14, 15]]])
```

[83]:

```
arr.swapaxes(1, 2)    # swap axe 1 with axe 2 : arr[x][y][z] = arr[x][z][y]
```

[83]:

```
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],

       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

swapaxes similarly returns a view on the data without making a copy.

3.1.9 Universal Functions: Fast Element-Wise Array Functions

A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results. Many ufuncs are simple element-wise transformations, like *sqrt* or *exp*:

[84]:

```
arr = np.arange(10)

arr
```

[84]:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[85]:

```
np.sqrt(arr)
```

[85]:

```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])
```

[86]:

```
np.exp(arr)
```

[86]:

```
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])
```

These are referred to as unary ufuncs. Others, such as *add* or *maximum*, take two arrays (thus, binary ufuncs) and return a single array as the result:

[87]:

```
x = np.random.randn(8)
```

[88]:

```
y = np.random.randn(8)
```

```
[89]: x
```

```
[89]: array([ 1.87304468,  0.10319766,  0.60922039,  0.58539509, -0.17837051,
             -0.15177182, -0.65523098, -0.01261685])
```

```
[90]: y
```

```
[90]: array([ 1.46079153, -2.23912871, -0.1818244 ,  1.24401477, -1.34174993,
              1.24839598, -0.34946606, -0.20187572])
```

```
[91]: np.maximum(x, y)
```

```
[91]: array([ 1.87304468,  0.10319766,  0.60922039,  1.24401477, -0.17837051,
              1.24839598, -0.34946606, -0.01261685])
```

Here, *numpy.maximum* computed the element-wise maximum of the elements in x and y. While not common, a ufunc can return multiple arrays. *modf* is one example, a vectorized version of the built-in Python *divmod*; it returns the fractional and integral parts of a floating-point array:

```
[92]: arr = np.random.randn(7) * 5

      arr
```

```
[92]: array([-4.6656715 ,  3.96619232,  5.68279243,  2.85593404, -0.75357055,
              2.44168874, -2.02588264])
```

```
[93]: remainder, whole_part = np.modf(arr)

      remainder
```

```
[93]: array([-0.6656715 ,  0.96619232,  0.68279243,  0.85593404, -0.75357055,
              0.44168874, -0.02588264])
```

```
[94]: whole_part
```

```
[94]: array([-4.,  3.,  5.,  2., -0.,  2., -2.])
```

See Tables 3-3 and 3-4 for a listing of available ufuncs.

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating point, or complex values. Use fabs as a faster alternative for non-complex-valued data
sqrt	Compute the square root of each element. Equivalent to arr ** 0.5
square	Compute the square of each element. Equivalent to arr ** 2

Function	Description
exp	Compute the exponent ex of each element
log, log10, log2, log1p	Natural logarithm (base e), log base 10, log base 2, and log(1 + x), respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
ceil	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
floor	Compute the floor of each element, i.e. the largest integer less than or equal to each element
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as separate array
isnan	Return boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf	Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
cos, cosh, sin, sinh, tan, tanh	Regular and hyperbolic trigonometric functions
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverse trigonometric functions
logical_not	Compute truth value of not x element-wise. Equivalent to -arr.

Table 3-3. Unary ufuncs

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum. fmax ignores NaN
minimum, fmin	Element-wise minimum. fmin ignores NaN
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators >, >=, <, <=, ==, !=

Function	Description
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation. Equivalent to infix operators & |, ^

Table 3-4. Binary universal functions

3.1.10 Mathematical and Statistical Methods

A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class. You can use aggregations (often called reductions) like *sum*, *mean*, and *std* (standard deviation) either by calling the array instance method or using the top-level NumPy function.

Here I generate some standard normally distributed random data and compute some aggregate statistics:

```
[95]: arr = np.random.randn(5, 4)

      arr
```

```
[95]: array([[-0.24564346, -1.89484331,  0.03209492, -0.45978323],
             [-0.99839059, -1.34623012,  1.30928306, -0.47577643],
             [ 0.28338889, -1.24169591,  0.61659483,  0.83819351],
             [-0.73491145,  0.36279299, -0.2320921 ,  1.40146577],
             [-1.2628713 ,  1.16016504, -1.12397748,  0.58961592]])
```

```
[96]: np.mean(arr)  # or use 'arr.mean()'
```

```
[96]: -0.1711310235122994
```

```
[97]: np.sum(arr) # or use 'arr.sum()'
```

```
[97]: -3.4226204702459877
```

Functions like mean and sum take an optional axis argument that computes the statistic over the given axis, resulting in an array with one fewer dimension:

```
[98]: arr.mean(axis=1)
```

```
[98]: array([-0.64204377, -0.37777852,  0.12412033,  0.1993138 , -0.15926696])
```

```
[99]: arr.sum(axis=0)
```

```
[99]: array([-2.95842791, -2.95981133,  0.60190323,  1.89371554])
```

Here, *arr.mean(1)* means “compute mean across the columns” where *arr.sum(0)* means “compute sum down the rows.” Other methods like *cumsum* and *cumprod* do not aggregate, instead producing an array of the intermediate results:

[100]:
```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

arr.cumsum()
```

[100]:
```
array([ 0,  1,  3,  6, 10, 15, 21, 28], dtype=int32)
```

In multidimensional arrays, accumulation functions like *cumsum* return an array of the same size, but with the partial aggregates computed along the indicated axis according to each lower dimensional slice:

[101]:
```
arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

arr
```

[101]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

[102]:
```
arr.cumsum(axis=0)
```

[102]:
```
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]], dtype=int32)
```

[103]:
```
arr.cumprod(axis=1)
```

[103]:
```
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]], dtype=int32)
```

See Table 3-5 for a full listing.

Method	Description
sum	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
mean	Arithmetic mean. Zero-length arrays have NaN mean.
std, var	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n).
min, max	Minimum and maximum.
argmin, argmax	Indices of minimum and maximum elements, respectively.
cumsum	Cumulative sum of elements starting from 0
cumprod	Cumulative product of elements starting from 1

Table 3-5. Basic array statistical methods

3.1.11 Methods for Boolean Arrays

Boolean values are coerced to 1 (True) and 0 (False) in the above methods. Thus, *sum* is often used as a means of counting True values in a boolean array:

```
[104]: arr = np.random.randn(100)

       (arr > 0).sum()
```

```
[104]: 52
```

There are two additional methods, *any* and *all*, useful especially for boolean arrays. *any* tests whether one or more values in an array is True, while *all* checks if every value is True:

```
[105]: bools = np.array([False, False, True, False])

       bools.any()
```

```
[105]: True
```

```
[106]: bools.all()
```

```
[106]: False
```

These methods also work with non-boolean arrays, where non-zero elements evaluate to True.

3.1.12 File Input and Output with Arrays

NumPy is able to save and load data to and from disk either in text or binary format. In this section I only discuss NumPy's built-in binary format, since most users will prefer pandas or other tools for loading text or tabular data. *np.save* and *np.load* are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension .npy:

```
[107]: arr = np.arange(10)

       np.save('some_array', arr)
```

If the file path does not already end in .npy, the extension will be appended. The array on disk can then be loaded with *np.load*:

```
[108]: np.load('some_array.npy')
```

```
[108]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You save multiple arrays in an uncompressed archive using *np.savez* and passing the arrays as keyword arguments:

[109]:

```
np.savez('array_archive.npz', a=arr, b=arr)
```

When loading an .npz file, you get back a dict-like object that loads the individual arrays lazily:

[110]:

```
arch = np.load('array_archive.npz')

arch['b']
```

[110]:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

If your data compresses well, you may wish to use *numpy.savez_compressed* instead:

[111]:

```
np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

3.1.13 More

If you want to learn more about Numpy, like how to deal with the linear algebra problem, you can go to the official documentation to see more details.

https://numpy.org/devdocs/user/ (You could use 'qiuck search' in the website)

3.2 SciPy(Optional)

SciPy is a collection of functions for scientific computing in Python. It provides, among other functionality, advanced linear algebra routines, mathematical function optimization, signal processing, special mathematical functions, and statistical distributions. scikit-learn draws from SciPy's collection of functions for implementing its algorithms. The most important part of SciPy for us is scipy.sparse: this provides sparse matrices, which are another representation that is used for data in scikitlearn. Sparse matrices are used whenever we want to store a 2D array that contains mostly zeros:

[112]:

```
from scipy import sparse

# Create a 2D NumPy array with a diagonal of ones, and zeros everywhere else
eye = np.eye(4)
print("NumPy array:\n{}".format(eye))
```

```
NumPy array:
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

See the usage of 'format' function via the following website:
https://www.geeksforgeeks.org/python-format-function/

[113]:

```
# Convert the NumPy array to a SciPy sparse matrix in CSR format
# Only the nonzero entries are stored
sparse_matrix = sparse.csr_matrix(eye)
print("\nSciPy sparse CSR matrix:\n{}".format(sparse_matrix))
```

```
SciPy sparse CSR matrix:
  (0, 0)        1.0
  (1, 1)        1.0
  (2, 2)        1.0
  (3, 3)        1.0
```

Sometimes it is not possible to create dense representations of sparse data (as they would not fit into memory), so we need to create sparse representations directly. Here is a way to create the same sparse matrix as before, using the COO format:

[114]:

```
data = np.ones(4)
row_indices = np.arange(4)
col_indices = np.arange(4)
eye_coo = sparse.coo_matrix((data, (row_indices, col_indices)))
print("COO representation:\n{}".format(eye_coo))
```

```
COO representation:
  (0, 0)        1.0
  (1, 1)        1.0
  (2, 2)        1.0
  (3, 3)        1.0
```

More details on SciPy sparse matrices can be found in the SciPy Lecture Notes: http://scipylectures.org/

3.3 pandas

pandas is a Python library for data wrangling and analysis. It is built around a data structure called the DataFrame that is modeled after the R DataFrame. Simply put, a pandas DataFrame is a table, similar to an Excel spreadsheet. pandas provides a great range of methods to modify and operate on this table; in particular, it allows SQL-like queries and joins of tables. In contrast to NumPy, which requires that all entries in an array be of the same type, pandas allows each column to have a separate type (for example, integers, dates, floating-point numbers, and strings). Another valuable tool provided by pandas is its ability to ingest from a great variety of file formats and databases, like SQL, Excel files, and comma-separated values (CSV) files. Here is a small example of creating a DataFrame using a dictionary:

[115]:

```
import pandas as pd

# create a simple dataset of people
data = {'Name': ["John", "Anna", "Peter", "Linda"], 'Location' : ["New York",␣
 ↪"Paris", "Berlin", "London"], 'Age' : [24, 13, 53, 33]
}

data_pandas = pd.DataFrame(data)

print(data_pandas)
```

```
    Name  Location  Age
0   John  New York   24
1   Anna     Paris   13
2  Peter    Berlin   53
3  Linda    London   33
```

There are several possible ways to query this table. For example:

[116]:

```
# Select all rows that have an age column greater than 30
print(data_pandas[data_pandas.Age > 30])
```

```
    Name Location  Age
2  Peter   Berlin   53
3  Linda   London   33
```

Please review 'DSO 545 lab 02: Data Manipulation using Pandas'.

I give the official documentation of pandas as follows if you need it to look up for something: https://pandas.pydata.org/pandas-docs/stable/

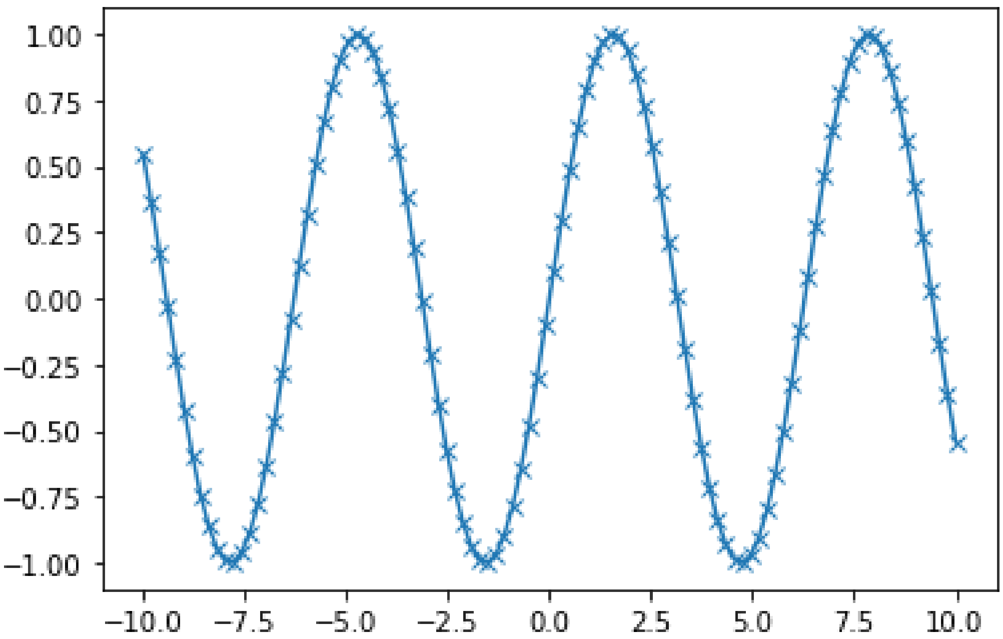
3.4 matplotlib

matplotlib is the primary scientific plotting library in Python. It provides functions for making publication-quality visualizations such as line charts, histograms, scatter plots, and so on. Visualizing your data and different aspects of your analysis can give you important insights, and we will be using matplotlib for all our visualizations. For example, this code produces the plot.

[117]:

```
import matplotlib.pyplot as plt
# Generate a sequence of numbers from -10 to 10 with 100 steps in between
x = np.linspace(-10, 10, 100)
# Create a second array using sine
y = np.sin(x)
# The plot function makes a line chart of one array against another
plt.plot(x, y, marker="x")
plt.show()
```

```
[117]: [<matplotlib.lines.Line2D at 0x25c5a267fc8>]
```



Please review DSO 545 labs about the matplotlib.

I give the official documentation of matplotlib as follows if you need it to look up for something: https://matplotlib.org/users/index.html

References:

Müller, Andreas C; Guido, Sarah. (2017). Introduction to Machine Learning with Python.

McKinney Wes. (2018). Python for data analysis.