

2.1: Sequential Data Structures

In the previous part of the tutorial, we worked through a simple example of a Scipy program which calculates the electric potential produced by a collection of charges in 1D. At the time, we did not explain much about the data structures that we were using to store numerical information (such as the values of the electric potential at various points). Let's do that now.

There are three common data structures that will be used in a Scipy program for scientific computing: arrays, lists, and tuples. These structures store linear sequences of Python objects, similar to the concept of "vectors" in physics and mathematics. However, these three types of data structures all have slightly different properties, which you should be aware of.

2.1.1 Arrays

An [array](#) is a data structure that contains a sequence of numbers. Let's do a quick recap. From a fresh Python command prompt, type the following:

```
>>> from scipy import *
>>> x = linspace(-0.5, 0.5, 9)
>>> x
array([-0.5 , -0.375, -0.25 , -0.125,  0.    ,  0.125,  0.25 ,  0.375,  0.5  ])
```

The first line, as usual, is used to import the `scipy` module. The second line creates an array named `x` by calling [linspace](#), which is a function defined by `scipy`. With the given inputs, the function returns an array of 9 numbers between -0.5 and 0.5, inclusive. The third line shows the resulting value of `x`.

The array data structure is provided specifically by the Scipy scientific computing module. Arrays can only contain numbers (furthermore, each individual array can only contain numbers of one type, e.g. integers or complex numbers; we'll discuss this in the next article). Arrays also support special facilities for doing [numerical linear algebra](#). They are commonly created using one of these functions from the `scipy` module:

- [linspace](#), which creates an array of evenly-spaced values between two endpoints.
- [arange](#), which creates an array of integers in a specified range.
- [zeros](#), which creates an array whose elements are all 0.0.
- [ones](#), which creates an array whose elements are all 1.0.
- [empty](#), which creates an array whose elements are uninitialized (this is usually used when you want to set the elements later).

Of these, we've previously seen examples of the `linspace` and `zeros` functions being used. As another example, to create an array of 500 elements all containing the number -1.2 , you can use the `ones` function and a multiplication operation:

```
x = -1.2 * ones(500)
```

An alternative method, which is *slightly* faster, is to generate the array using `empty` and then use the `fill` method to populate it:

```
x = empty(500); x.fill(-1.2)
```

One of the most important things to know about an array is that its size is fixed at the moment of its creation. When creating an array, you need to specify exactly how many numbers you want to store. If you ever need to revise this size, you must create a *new* array, and transfer the contents over from the old array. (For very big arrays, this might be a slow operation, because it involves copying a lot of numbers between different parts of the computer memory.)

You can pass arrays as inputs to functions in the usual way (e.g., by supplying its name as the argument to a function call). We have already encountered the `len` function, which takes an array input and returns the array's length (an integer). We have also encountered the `abs` function, which accepts an array input and returns a new array containing the corresponding absolute values. Similar to `abs`, many mathematical functions and operations accept arrays as inputs; usually, this has the effect of applying the function or operation to each element of the input array, and returning the result as another array. The returned array

has the same size as the input array. For example, the `sin` function with an array input `x` returns another array whose elements are the sines of the elements of `x` :

```
>>> y = sin(x)
>>> y
array([-0.47942554, -0.36627253, -0.24740396, -0.12467473,  0.
        0.12467473,  0.24740396,  0.36627253,  0.47942554])
```

You can access individual elements of an array with the notation `a[j]`, where `a` is the variable name and `j` is an integer index (where the first element has index 0, the second element has index 1, etc.). For example, the following code sets the first element of the `y` array to the value of its second element:

```
>>> y[0] = y[1]
>>> y
array([-0.36627253, -0.36627253, -0.24740396, -0.12467473,  0.
        0.12467473,  0.24740396,  0.36627253,  0.47942554])
```

Negative indices count backward from the end of the array. For example:

```
>>> y[-1]
0.47942553860420301
```

Instead of setting or retrieving individual values of an array, you can also set or retrieve a *sequence* of values. This is referred to as **slicing**, and is described in detail [in the Scipy documentation](#). The basic idea can be demonstrated with a few examples:

```
>>> x[0:3] = 2.0
>>> x
array([ 2. ,  2. ,  2. , -0.125,  0. ,  0.125,  0.25 ,  0.375,  0.5  ])
```

The above code accesses the elements in array `x`, starting from index 0 up to but not including 3 (i.e. indices 0, 1, and 2), and assigns them the value of `2.0`. This changes the contents of the array `x`.

```
>>> z = x[0:5:2]
>>> z
array([ 2.,  2.,  0.])
```

The above code retrieves a subset of the elements in array `x`, starting from index 0 up to but not including 5, and stepping by 2 (i.e., the indices 0, 2, and 4), and then groups those elements into an array named `z`. Thereafter, `z` can be treated as an array.

Finally, arrays can also be multidimensional. If we think of an ordinary (1D) array as a vector, then a 2D array is equivalent to a matrix, and higher-dimensional arrays are like tensors. We will see practical examples of higher-dimensional arrays later. For now, here is a simple example:

```
>>> y = zeros((4,2))      # Create a 2D array of size 4x2
>>> y[2,0] = 1.0
>>> y[0,1] = 2.0
>>> y
array([[ 0.,  2.],
       [ 0.,  0.]
```

```
[ 1.,  0.],  
[ 0.,  0.]])
```

2.1.2: Lists

There is another type of data structure called a [list](#). Unlike arrays, lists are built into the Python programming language itself, and are not specific to the Scipy module. Lists are general-purpose data structures which are *not* optimized for scientific computing (for example, we will need to use arrays, not lists, when we want to do linear algebra).

The most convenient thing about Python lists is that you can specify them explicitly, using `[...]` notation. For example, the following code creates a list named `u`, containing the integers 1, 1, 2, 3, and 5:

```
>>> u = [1, 1, 2, 3, 5]  
>>> u  
[1, 1, 2, 3, 5]
```

This way of creating lists is also useful for creating Scipy arrays. The [array](#) function accepts a list as an input, and returns an array containing the same elements as the input list. For example, to create an array containing the numbers 0.2, 0.1, and 0.0:

```
>>> x = array([0.2, 0.1, 0.0])  
>>> x  
array([ 0.2,  0.1,  0. ])
```

In the first line, the square brackets create a list object containing the numbers 0.2, 0.1, and 0.0, then passes that list directly as the input to the `array` function. The above code is therefore equivalent to the following:

```
>>> inputlist = [0.2, 0.1, 0.0]  
>>> inputlist  
[0.2, 0.1, 0.0]  
>>> x = array(inputlist)  
>>> x  
array([ 0.2,  0.1,  0. ])
```

Usually, we will do number crunching using arrays rather than lists. However, sometimes it is useful to work directly with lists. One convenient thing about lists is that they can contain arbitrary Python objects, of any data type; by contrast, arrays are allowed only to contain numerical data.

For example, a Python list can store character strings:

```
>>> u = [1, 2, 'abracadabra', 3]  
>>> u  
[1, 2, 'abracadabra', 3]
```

And you can set or retrieve individual elements of a Python list in the same way as an array:

```
>>> u[1] = 0  
>>> u  
[1, 0, 'abracadabra', 3]
```

Another great advantage of lists is that, unlike arrays, you can dynamically increase or decrease the size of a list:

```
>>> u.append(99)                # Add 99 to the end of the list u
>>> u.insert(0, -99)            # Insert -99 at the front (index 0) of the list u
>>> u
[-99, 1, 0, 'abracadabra', 3, 99]
>>> z = u.pop(3)                # Remove element 3 from list u, and name it z
>>> u
[-99, 1, 0, 3, 99]
>>> z
'abracadabra'
```

Aside: About Methods

In the above example, `append`, `insert`, and `pop` are called **methods**. You can think of a method as a special kind of function which is "attached" to an object and relies upon it in some way. Just like a function, a method can accept inputs, and give a return value. For example, when `u` is a list, the code

```
z = u.pop(3)
```

means to take the list `u`, find element index 3 (specified by the input to the method), remove it from the list, and return the removed list element. In this case, the returned element is named `z`. [See here for a summary of Python list methods](#). We'll see more examples of methods as we go along.

2.1.3 Tuples

Apart from lists, Python provides another kind of data structure called a **tuple**. Whereas lists can be constructed using square bracket `[...]` notation, tuples can be constructed using parenthetical `(...)` notation:

```
>>> v = (1, 2, 'abracadabra', 3)
>>> v
(1, 2, 'abracadabra', 3)
```

Like lists, tuples can contain any kind of data type. But whereas the size of a list can be changed (using methods like `append`, `insert`, and `pop`, as described in the previous subsection), the size of a tuple is fixed once it's created, just like an array.

Tuples are mainly used as a convenient way to "group" or "ungroup" named variables. Suppose we want to split the contents of `v` into four separate named variables. We could do it like this:

```
>>> dog, cat, apple, banana = v
>>> dog
1
>>> cat
2
>>> apple
'abracadabra'
>>> banana
3
```

On the left-hand side of the `=` sign, we're actually specifying a tuple of four variables, named `dog`, `cat`, `apple`, and `banana`. In cases like this, it is OK to omit the parentheses; when Python sees a group of variable names separated by commas, it automatically treats that group as a tuple. Thus, the above line is equivalent to

```
>>> (dog, cat, apple, banana) = v
```

We saw a similar example in the previous part of the tutorial, where there was a line of code like this:

```
pmin, pmax = -50, 50
```

This assigns the value -50 to the variable named `pmin` , and 50 to the variable named `pmax` . We'll see more examples of tuple usage as we go along.

This page titled [2.1: Sequential Data Structures](#) is shared under a [CC BY-SA 4.0](#) license and was authored, remixed, and/or curated by [Y. D. Chong](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.