analysis; however, for most engineering problems it is advantageous to use the ubiquitous package NumPy (Harris et al. 2020). The primary reasons this is preferred are that NumPy provides data types, functions, and methods optimized for numerical calculations, which go far beyond Python's built-in modules. In the first several sections of this chapter, we will explore NumPy's data types (most notably the array) and some of its basic functions and methods.

The numerical data represented in NumPy often originates as data from outside the program (e.g., from sensor data gathered via an experiment). Stored in files of various formats, the data must be read from computer memory[1] into the program. This is the most common kind of a program's **inputs**. On the other end, a program can have **outputs**, frequently data files written to computer memory. In this chapter, we will learn how to load input data from files and write output data to files.

Another important kind of program output is a **graphic**—usually a graph, a plot, or a chart. A graphic is often a very important result of a numerical analysis, data visualization being a key component of engineering decision making. In this chapter, we will learn how to use the Python package Matplotlib (Hunter 2007) to generate graphics from data.

## 3.1 Arrays


NumPy arrays are ubiquitous for representing numerical data. Like lists, arrays are mutable and can represent collections of objects. Unlike for lists, the elements of an array must all be of the same (typically numeric) type. In this section, we learn how to create and manipulate basic arrays. Throughout this book, we will assume that the NumPy package is loaded with the following statement:

```
import numpy as np
```

### 3.1.1 Creating Arrays

To construct a basic array (i.e., class `np.ndarray`), we often use the function `np.array()`. Although many types of objects can be passed, a list will often do, as follows:

```
x = np.array([0.29, 0.55, -0.31, -0.84, 0.97])
```

The `shape` attribute of the `np.ndarray` object is an integer tuple representing the size (i.e., length) of each of its **dimensions**. For instance, the shape of the 1-dimensional (1D) array of five elements given the name `x` above is printed with

1. The program typically reads a file stored in "secondary" (i.e., long-term) memory and loads it into "main" memory, which is faster to access for calculations. Similarly, when a program writes to a file, it stores data that is in main memory in secondary memory.

```
print(x.shape)
```

This returns `(5,)`, which indicates the array has a single dimension, called an **axis** in NumPy, with size `5`.

In NumPy, 1D arrays are called **vectors**, 2D arrays are called **matrices**, and higher-dimensional arrays are called **tensors**. The mathematical objects with the same names (i.e., vectors, matrices, and tensors) are usually represented with arrays with corresponding names. A matrix can be created as follows:

```
A = np.array([
    [0, 1, 2, 3],  # First row
    [4, 5, 6, 7],  # Second row
    [8, 9, 10, 11],  # Third row
])
```

So `A.shape` is `(3, 4)` and it represents a $3 \times 4$ mathematical matrix.

We often need to create an array with a specific shape and populate it later. Perhaps the best way to do so is a function call like

```
T = np.full(shape=(5, 3), fill_value=np.nan)
```

This creates a `(5, 3)` matrix with the special `nan` (i.e., not a number) `float` as each element. Related functions `np.zeros()` and `np.ones()` can also be used to create arrays of arbitrary shape filled with `0` and `1` values, respectively. However, for an array that is to be populated subsequently, we prefer `np.full()` with `nan` elements because it is easier to notice if parts of the array have been mistakenly left unpopulated.

The `np.arange()` function is similar to the built-in `range()` function. An array of sequential numbers with integer spacing can be easily created with the `np.arange()` function. For instance,

```
np.arange(start=0, stop=10)
np.arange(0, 10)
np.arange(stop=10)
np.arange(10)
```

All these statements yield an array that prints as follows (printed arrays look like lists):

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

Of course, the `start` argument is necessary for an array that starts at a value other than `0`. There is a `step` argument for `np.arange()`, and it is useful for integer steps other than the default `1`. However, for non-integer steps, we prefer a different function altogether: `np.linspace()`. For instance, the following creates a 1D array of 31 elements from 0 to 3:

```
np.linspace(start=0, stop=3, num=11)
```

This array can be printed to show the following:

```
[0., 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3.]
```

Note that by default `stop` is the last sample; `endpoint=False` would exclude it.

We make extensive use of `np.linspace()` and regular use of the similar `np.logspace()`, which creates a 1D array logarithmically spaced between two powers of the base provided. For instance, in the default base 10, we can generate 6 values from $10^0$ to $10^3$ with

```
np.logspace(start=0, stop=3, num=6)
```

This array can be printed to show the following:

```
[1., 3.98107171, 15.84893192, 63.09573445, 251.18864315, 1000.]
```

Most of the time, we use numeric values (i.e., `dtypes` of `int`, `float`, `complex`, and `bool` types) in Python arrays. It is also possible to create a Python **object array** with `dtype "O"` for object; for instance:

```
A = np.array([{"foo": "bar"}, {"bar": "baz"}])  # An object array
```

Printing the `A.dtype` attribute reveals that it is type `object`. It is occasionally advantageous to use object arrays instead of lists, primarily for the convenience of NumPy's array manipulation capabilities.

### 3.1.2 Accessing, Slicing, and Assigning Elements

Array elements can be accessed via indices in the same way as with lists. For instance,

```
A = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]) # 3x4
A[0, 0]  # => 0
A[0, 3]  # => 3
A[1, 0]  # => 4
A[2, 3]  # => 11
A[1]  #=> [4, 5, 6, 7]
```

Similarly, array slicing has the same syntax as list slicing. For instance,

```
A[0:2]  # => [[0, 1, 2, 3], [4, 5, 6, 7]] (view)
A[:-1]  # => [[0, 1, 2, 3], [4, 5, 6, 7]] (view)
A[:, 1]  # => [1, 5, 9] (view)
A[:, 0:2]  # => [[0, 1], [4, 5], [8, 9]] (view)
```

An important difference between list and array slicing is that, whereas in list slicing the returned list is a *copy* of a portion of the original list, in array slicing, the returned value is a **view** of a portion of the original array. An array view object, just as with `dict` view objects (see section 1.8), uses the same data as the original

object. Therefore, mutating a view object mutates its original object and vice versa. For instance, using the same `A` matrix from above,

```
a = A[:, 0]  # => [0, 4, 8] (first column view)
a[1] = 6  # Assign a new value to view element (second row)
A[0, 0] = 2  # Assign a new value to the original array
print(a)
print(A)
```

This prints

```
[2 6 8]
[[ 2  1  2  3]
 [ 6  5  6  7]
 [ 8  9 10 11]]
```

In other words, the data in `A` and `a` are the same data. To create a copy instead of a view from a slice, simply append the `copy()` method. For instance, the following array `b` is a copy of a portion of `A`, so its data are independent:

```
b = A[:, 0].copy()  # => [0, 4, 8] (first column copy)
```

It is often useful to find the indices of an array that meet some condition. Placing an array in a conditional statement returns Boolean an array of Boolean values for each element that can be used as an index for the array. For instance,

```
A = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]) # 3x4
print(A > 4)
```

prints

```
[[False, False, False, False]
 [False,  True,  True,  True]
 [ True,  True,  True,  True]]
```

This can be used as an index to select those elements that meet the condition. For instance,

```
A[A > 4]
```

returns those elements greater than `4`, as follows:

```
[ 5,  6,  7,  8,  9, 10, 11]
```

The use of the Boolean-valued array resulting from the expression `A > 4` as an index is a type of **advanced indexing** (i.e., slicing), which uses an array with data type Boolean or integer, a non-tuple sequence, or a tuple with at least one sequence object. Unlike basic slicing, which returns a view of the original array, advanced indexing always returns a copy.

Because arrays are mutable, elements can be replaced just as with lists. For instance, the statements

```
A[0, 0] = 2
A[:, 2] = 2
```

mutate `A` such that it now prints as

```
[[ 2,  1,  2,  3],
 [ 4,  5,  2,  7],
 [ 8,  9,  2, 11]]
```

Combining the conditional indexing from above with assignment, we can make assignments based on a condition. For instance, working with the same `A` array, we can coerce values above 5 to 5 as follows:

```
A[A > 5] = 5
```

Now `A` prints as

```
[[2, 1, 2, 3],
 [4, 5, 2, 5],
 [5, 5, 2, 5]]
```

### 3.1.3 Appending To and Concatenating Arrays

Appending an element to an array is possible with the `np.append()` function (there is no `append()` method), but its use in loops is discouraged due to the fact that it creates a new copy of the array at every call. However, in some cases it is just the right function, and it works as shown in the following code:

```
a = np.array([0, 1, 2])
np.append(a, 3)  # => [0, 1, 2, 3]
```

When needing to construct the elements of an array in a loop, it is vastly more efficient to initialize the array with `np.full()` or similar function (see section 3.1.1) before beginning the loop, using index assignment. For instance,

```
a = np.full((5,), np.nan)  # Initialize with nans
for i in range(0, len(a)):
    if i == 0:
        a[i] = 1
    else:
        a[i] = (a[i - 1] + 1) ** 2
print(f"It is {np.any(np.isnan(a))} there are nans in a:\n{a}")
```

prints

```
It is False there are nans in a:
[1.00000e+00 4.00000e+00 2.50000e+01 6.76000e+02 4.58329e+05]
```

The statement `np.any(np.isnan(a))` is a nice idiom for detecting if any `nans` remain in the array. This is a good check that we have in fact replaced all elements of the initialized array with numbers.

Array **concatenation** is the ordered collection of arrays. The `np.concatenate()` function returns a concatenation of arrays given as a tuple to its first argument. For instance,

```
a = np.array([[0, 1], [2, 3]])  # 2x2
b = np.array([[4, 5]])  # 1x2
np.concatenate((a, b))  # => [[0, 1], [2, 3], [4, 5]] (3x2)
```

The `axis` optional argument, `0` by default, determines the dimension along which the array concatenates. For instance, with the same `a` and `b` from above,

```
np.concatenate((a, b), axis=0)  # => [[0, 1], [2, 3], [4, 5]] (3x2)
np.concatenate((a, b.T), axis=1)  # => [[0, 1, 4], [2, 3, 5]] (2x3)
```

Here we have used the **transpose** array attribute, which returns a view of the array with its axes swapped (see section 3.2.1). The arrays to be concatenated must have matching dimensions except in the `axis` dimension.

Box 3.1 Further Reading

- NumPy Developers (2024c), for a basic and short introduction to NumPy

## 3.2 Manipulating, Operating On, and Mapping Over Arrays


In this section, we learn to manipulate, operate on, and map over NumPy arrays.

### 3.2.1 Array Manipulation Functions and Methods

NumPy has many powerful functions and methods for manipulating arrays. We cover only those most frequently useful to us, here; for a full list and documentation, see (NumPy Developers 2024a).

**3.2.1.1 Sorting** To sort an array, the `np.sort(a)` function returns a sorted copy of `a` and the `a.sort()` method will sort (mutate) `a` itself. For instance,

```
a = np.array([6, -3, 0, 9, -6])
np.sort(a)  #  => [-6, -3,  0,  6,  9] (copy)
a.sort()  # a: [-6, -3,  0,  6,  9]
```

The function and the method have the same optional arguments, the most useful of which is `axis: int`, the axis along which to sort. The default is `-1` (i.e., the last dimension).