Chapter 4

Libraries, Arrays and Plots

4.1 Importing libraries

Many Python functions are grouped together into libraries that can be imported into a program. Trig functions, exponentials, logarithms, *etc*, are contained in a library called NumPy. In Chapter 2 we saw how to import and use NumPy:

```
import numpy
# function names must include the prefix numpy
rootseven = numpy.sqrt(7.0)
```

This code computes the square root of 7.0 and assigns the result to the variable `rootseven`.

It can be tiresome to type the prefix `numpy` as part of the name for common mathematical functions. An alternative is to import all NumPy commands with an asterisk:

```
from numpy import *
# function names don't need a prefix
rootseven = sqrt(7.0)
```

This works, but can be dangerous. If we import more than one library in this way, we might have two different functions with the same name. This can lead to conflicts.

The solution is to import the library with a shorter name, like this:

```
import numpy as np
# function names must include the prefix np
rootseven = np.sqrt(7.0)
```

Experienced programmers consider this to be the best way to import a library.

4.2 NumPy arrays

Recall that a Python *list* is a collection of numbers or strings. For example, `[8, -3.2, "tree"]` is a list. A NumPy *array* is similar to a list, but differs in the way it's stored in computer memory. NumPy arrays are used almost exclusively for numerical computation. We will only work with NumPy arrays whose elements are numbers, not strings.

One way to create a NumPy array is with the `linspace()` function. The array `linspace(a,b,N)` consists of `N` evenly spaced numbers beginning with `a` and ending with `b`. Note that `N` must be an integer. The numbers `a` and `b` can be either integer or real. For example, consider the code

```
import numpy as np
myarray = np.linspace(1.0, 3.0, 5)
print(myarray)
```

This array contains 5 evenly spaced numbers beginning with 1.0 and ending with 3.0.

The output of the code above is

```
[1. 1.5 2. 2.5 3. ]
```

This looks a lot like a list of numbers. For comparison, consider

```
mylist = [1, 1.5, 2, 2.5, 3]
print(mylist)
```

which outputs the list

```
[1, 1.5, 2, 2.5, 3]
```

When printed, both arrays and lists are surrounded by square brackets. However, array elements are separated by spaces whereas list elements are separated by commas.

More importantly, NumPy arrays and Python lists differ in their mathematical behavior:

- Addition of two NumPy arrays occurs on an element-by-element basis. Addition of two lists simply concatenates the lists.
- Multiplication of a NumPy array by a number occurs on an element-by-element basis. Multiplication of a list by an integer n concatenates n copies of the list. Multiplication of a list by a real number is not defined.
- Multiplication of two NumPy arrays occurs on an element-by-element basis. Multiplication of two lists is not defined.

Exercise 4.2a

Create a NumPy array `A = [2.  2.2 2.4 2.6 2.8 3.  ]` using the `linspace()` function, and create a Python list `B = [2.0, 2.2, 2.4, 2.6, 2.8, 3.0]`. Find the results of the following calculations:

- `A + A`
- `B + B`
- `3*A`
- `3*B`
- `A*A`
- `B*B`

Note the differences in behavior between NumPy arrays and Python lists.

The advantage of NumPy arrays, as opposed to lists of numbers, is that Python can perform mathematical operations on arrays without cycling over the individual elements one by one. For example, the following program computes the squares of the first 5 integers:

```
for x in [1,2,3,4,5]:
    xsquared = x**2
    print(x, xsquared)
```

The same result is obtained more efficiently using a NumPy array:

```
import numpy as np
x = np.linspace(1,5,5)
xsquared = x**2
print(x, xsquared)
```

Since `x` is a NumPy array, the command `xsquared = x**2` tells Python to create a new NumPy array, called `xsquared`, whose elements are the squares of the elements of `x`.

Exercise 4.2b

We considered projectile motion in Sec. 2.7. Write a program that computes a projectile's position at times $t = 0.0, 0.25, 0.5, \ldots 5.0$. Use `linspace()` to create a NumPy array for t. Compute the arrays x and y from Eqs. (2.1) without using a `for` loop. (You can choose any values of initial position and initial velocity that you like.)

4.3 Creating NumPy arrays

There are many ways to create NumPy arrays.

- `linspace(a,b,N)` creates an array with `N` equally spaced elements starting with `a` and ending with `b`.
- `array([a,b,c])` creates a NumPy array with elements `a`, `b`, and `c`.
- `zeros(N)` creates a NumPy array with `N` elements, all zeros.
- `ones(N)` creates a NumPy array with `N` elements, all ones.

Exercise 4.3a

Create NumPy arrays using each of the functions above. Do they give the expected results?

Another way to create a NumPy array is the `arange()` function, which is similar to the `range()` function discussed in Sec. 3.2. In particular,

- `arange(a,b)` creates a NumPy array that begins at `a`, increments by 1, and ends *before* `b`.
- `arange(a,b,s)` creates a NumPy array that begins at `a`, increments by `s`, and ends *before* `b`.
- `arange(b)` creates an array of integers, beginning with 0, incrementing by 1, and ending *before* `b`.

Remember, `arange()` creates a NumPy array, whereas `range()` creates a list. Another difference is this: for `arange()` the parameters `a`, `b` and `s` can be real numbers; for `range()` the parameters must be integers.

Exercise 4.3b

Create NumPy arrays using `arange()`. Experiment with different real and integer values for `a`, `b`, and `s`. Do they give the expected results?

As a general rule, you should use `linspace(a,b,N)` for creating arrays of real numbers. It can be tricky to use `arange(a,b,s)` if any of the values `a`, `b` or `s` are not integers.

Exercise 4.3c

Create the following arrays:

```
array1 = np.arange(2,7,1)
array2 = np.arange(13/3,11,2/3)
```

Can you predict the outcome in each case?

The command `arange(a,b,s)` is supposed to *exclude* `b`. Occasionally, when dealing with non-integers, `arange(a,b,s)` can make a mistake and include `b`. This is due to machine roundoff error.

4.4 Machine roundoff error

Most modern computers are 64–bit systems—they use 64 bits of memory to approximate a real number. One bit is used for the sign of the number and 11 bits are used for the exponent. The remaining 52 bits encode the significand. With this representation, real numbers are approximated to roughly 15, 16 or 17 significant figures, depending on the number.

To be precise, 64–bit computers encode real numbers as

$$\text{real number} = (-1)^s\,(1 + A)\,2^{B-1023} \ , \tag{4.1}$$

where

$$A = \sum_{i=1}^{52} a_i 2^{-i} \ ,$$
$$B = \sum_{j=0}^{10} b_j 2^{j} \ .$$

Here, s, a_i and b_j denote bits in memory, either 0 or 1. The single bit s determines the sign of the number. The 52 bits $a_1, \ldots, a_{52}$ comprise the signficand, and the 11 bits $b_0, \ldots, b_{10}$ determine the exponent.

As an example, the real number 2/3 is *approximated* by Eq. (4.1) with $s = 0$ and

$$a_i = \begin{cases} 0 & \text{if } i \text{ is odd} \ , \\ 1 & \text{if } i \text{ is even} \ , \end{cases}$$

$$b_j = \begin{cases} 0 & \text{if } j = 0 \text{ or } 10 \ , \\ 1 & \text{otherwise} \ . \end{cases}$$

This number is actually

$$0.66666666666666666296\ldots\ ,$$

slightly less than 2/3. The discrepancy is machine roundoff error.

Exercise 4.4a

Issue the command

```
print(f"x = {x:.20f}")
```

for various real numbers x, such as 2/3. Here, the print function uses f–string formatting to display the results to 20 decimal places. Can 0.3 be represented exactly? 0.1? 0.75?

Due to the finite representation of real numbers, even simple operations are subject to errors.

Exercise 4.4b

Does the conditional

```
1.3 - 1.0 == 0.3
```

evaluate to `True` or `False`?

Machine roundoff errors are present in all numerical calculations. Some consequences of machine error are discussed in later chapters.

Exercise 4.4c

Create a code that assigns variables $x = 0.1$, $y = 0.2$, $z = 0.3$, then computes

```
sum1 = x + (y + z)
sum2 = (x + y) + z
```

Check the equivalence of `sum1` and `sum2` using the comparison operator `==`.

4.5 Plotting graphs

To plot graphs in Python, we need to import the `matplotlib.pyplot` library:

```
import matplotlib.pyplot as plt
```

`plt` is a common abbreviation for `matplotlib.pyplot`.

The code below plots a graph of the function $y = x^3$ from -1 to 1:

```
import numpy as np
import matplotlib.pyplot as plt

# Create a numpy array of x values
x = np.linspace(-1, 1, 100)
# Create a numpy array of y values
y = x**3

# Graph y versus x
plt.close()                 # close plot windows
plt.plot(x,y,"k.")          # plot y versus x using black dots
plt.title("Function x**3")  # create a plot title
plt.xlabel("x")             # create a label for the x-axis
plt.ylabel("y")             # create a label for the y-axis
plt.show()                  # display the results
```

The result is shown in Fig. 4.1.

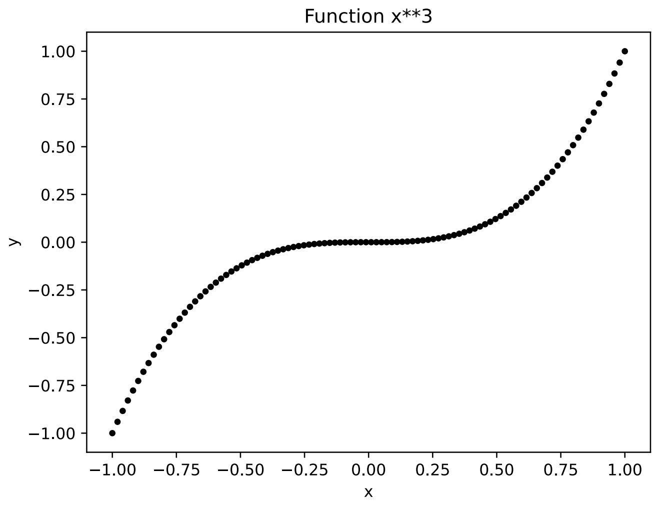

Fig. 4.1: Graph of $y = x^3$ created in Python with Matplotlib.

The function `close()` closes any plot windows that might have been opened previously. The function `show()` displays the figure as directed in the preceding pyplot commands. `close()` and `show()` can be omitted in Jupyter notebooks, but might be needed on other Python platforms.

Observe that `plot(t,y,"k.")` displays each data point as a black dot. If you replace `"k."` with `"k"`, the graph will be a smooth curve through the data. Replace `k` with `r` for red, `b` for blue, `g` for green, `y` for yellow, `c` for cyan, or `m` for magenta.

Exercise 4.5a

Create a new version of your projectile code. Use the `linspace` command to create a time array with 100 or more values. Use a single command (not a `for` loop) to compute the array of x values. Similarly for the y array. Have your code plot the shape of the trajectory, y versus x.

Exercise 4.5b

Plot the functions $f_1(x) = x^3$ and $f_2(x) = x^4$ from -1 to 1. Use `linspace` to create an array of x values, then create arrays `y1 = x**3` and `y2 = x**4`. Plot the two functions on the same graph by using two `plt.plot` commands. You might want to choose different colors for the two curves.