Chapter 8

Random Topics

8.1 Subplots, legends and Mathtext

The basic `plot()` function in Matplotlib is adequate for many purposes. The `subplots()` function opens up a wider range of functionality.

Consider a mass on a spring. The mass executes simple harmonic motion $x(t) = A\sin(\omega t)$ with amplitude A and angular frequency ω. After importing NumPy as `np` and `matplotlib.pyplot` as `plt`, the following code produces two graphs, one for position $x(t)$ and the other for velocity $v(t) = dx/dt = A\omega\cos(\omega t)$:

```
# Choose parameter values
A = 2.0
omega = 5.0

# Create arrays
t = np.linspace(0,5,100)
x = A*np.sin(omega*t)
v = A*omega*np.cos(omega*t)

# Create plots
plt.close()
fig, (ax1, ax2) = plt.subplots(2,1)

ax1.plot(t,x)
ax1.set_xlabel('t')
ax1.set_ylabel('x(t)')

ax2.plot(t,v)
ax2.set_xlabel('t')
ax2.set_ylabel('v(t)')
plt.show()
```

Exercise 8.1a

Run this code and observe the output. What happens if you replace `subplots(2,1)` with `subplots(1,2)`?

The command

```
fig, (ax1, ax2) = plt.subplots(2,1)
```

instructs Python to create a figure named `fig`. The figure consists of two plots, referred to as *axes*. The axes (plots) are named `ax1` and `ax2`. The argument `(2,1)` says that the plots should be arranged into `2` rows and `1` column.

The command `ax1.plot(t,x)` plots x versus t on the first axis. The command `ax2.plot(t,v)` plots v versus t on the second axis. The plot labels are created with `set_xlabel()` and `set_ylabel()`.

You can specify the size of the figure by adding the option `figsize` as an argument to `subplots()`. For example,

```
fig, (ax1, ax2) = plt.subplots(2,1,figsize=(5,7))
```

creates a figure that is 5 inches wide and 7 inches high. The default values are $6.4\,\text{in} \times 4.8\,\text{in}$.

To add figure and plot titles, insert the commands

```
fig.suptitle('Simple Harmonic Motion')
ax1.set_title('position')
ax2.set_title('velocity')
```

somewhere between `plt.close()` and `plt.show()`. You might need to add `plt.tight_layout()` before `plt.show()` to insure proper spacing between plots.

Exercise 8.1b

Add figure and plot titles, and experiment with different figure sizes. How does `plt.tight_layout()` affect your figure?

When there are multiple curves on a single plot, it can be helpful to add a legend. Here is a code that plots functions of the form $ax\sin(x)$ for several values of a, and includes a legend:

```
# Define functions
def f(x,a):
    return a*x*np.sin(x)
```

```
# Choose parameter values
a1 = 2.0
a2 = 3.0
a3 = 4.0

# Create x array
x = np.linspace(0,10,200)

# Create figure
plt.close()
fig, (ax1) = plt.subplots(1,1)
ax1.plot(x,f(x,a1),label='a ='+str(a1))
ax1.plot(x,f(x,a2),label='a ='+str(a2))
ax1.plot(x,f(x,a3),label='a ='+str(a3))
ax1.legend()
plt.show()
```

In this example we have chosen to use `subplots()`, even though there is only one plot. This is a common practice.

Exercise 8.1c

Plot functions of the form $x\cos(ax)$ on a single figure using various values of a. Include a legend.

By default, Matplotlib will display each curve as a solid line and cycle through the colors in this order: blue, orange, green, red, purple, brown, pink, gray, olive, and cyan. You can override these colors by adding a color option to the plot command. The default colors are specified by `'C0'` for blue, `'C1'` for orange, *etc.* For example,

```
ax1.plot(x,f(x,a1),'C2', label='a ='+str(a1))
```

will set the color of the first curve (the curve with parameter value `a1`) to green. Colors can also be specified as `'b'` (blue), `'g'` (green), `'r'` (red), `'c'` (cyan), `'m'` (magenta), `'y'` (yellow), `'k'` (black), or `'w'` (white). Note that `'b'` is not the same shade of blue as `'C0'`; likewise for the other colors.

The linestyle is specified with options such as `'-'` (solid), `'--'` (dashed), and `':'` (dotted). The option `'.'` tells Matplotlib to plot only the data points. The linestyle should be combined with the color option, if both are specified. For example,

```
ax1.plot(x,f(x,a1),'b:', label='a ='+str(a1))
```

creates a blue, dotted curve.

Exercise 8.1d

Using the code from the previous exercise, experiment with different colors and line styles. How do you make a "dash–dot" curve?

One final tip: You can use *Mathtext* in titles, labels and legends. Mathtext is a subset of TeX, used for typsetting mathematical expressions. Strings encompassed by dollar signs `$` are processed as Mathtext. For example,

```
fig.suptitle(r'$\frac{d^2 x}{dt^2} = -\omega^2 x$')
```

creates the title $\frac{d^2x}{dt^2} = -\omega^2 x$. The `r` in front of the string tells Python that what follows is a "raw string."

8.2 Data input and output

Computers are often used to handle large amounts of data. Computer programs should be capable of reading data from a text file, and writing data to a text file. For scientific purposes we usually want to read and write arrays of numbers. NumPy includes the functions `savetxt()` and `loadtxt()` that make reading and writing data straightforward.

To save a one–dimensional array `A` to a file named *filename.txt*, use

```
np.savetxt("filename.txt",A)
```

To read data from a file named *filename.txt* into an array `A`, use

```
A = np.loadtxt("filename.txt")
```

Remember, the file name is a string and should be surrounded by quotation marks.

If you are running Python on your local machine, the program will generally use your local working directory for reading and saving files. That is, the directory where the program itself is saved. If you are running Python in the cloud, reading and writing files to and from your local computer might require some extra steps.

Exercise 8.2a

Run the code

```
A = np.linspace(0,5,20)
np.savetxt("myarray.txt", A)
```

Does the file *myarray.txt* appear in the expected location? Does it have the right content? Now read the file *myarray.txt* into a NumPy array `B`:

```
B = np.loadtxt("myarray.txt")
print(B)
```

Does `B` equal `A`?

For a one–dimensional array, the numbers in a file can be written as either a single row or a single column.

Exercise 8.2b

Using a text editor, create a text file with several numbers in a single column. Create a second text file with the same numbers listed in a row, separated by spaces. Use `loadtxt()` to load these files into NumPy arrays `A` and `B`. Does `A = B`? What happens if you use commas to separate the numbers in the second text file?

The `savetxt()` and `loadtxt()` functions can be used with multicolumn data.

Exercise 8.2c

Create a text file consisting of two columns, each column containing several numbers. Use `loadtxt()` to read the file into an array `C`. What does `print(C)` produce?

In the exercise above, `loadtxt()` reads the data into a two–dimensional NumPy array `C`. Recall that you can reference individual elements of a two–dimensional NumPy array using the syntax `C[i][j]` or, more simply, `C[i,j]`.

If you want each column of the file to occupy a single one–dimensional array, you can copy the two columns of `C` into `A` and `B` using

```
A = C[:,0]    # Copy first column of C into A
B = C[:,1]    # Copy second column of C into B
```

The colon `:` means "all elements."

Exercise 8.2d

Examine the array `C` from the previous exercise. Which element is `C[0,1]`? Which element is `C[1,0]`? Place each column of `C` into a separate array. Place each row of `C` into a separate array.

There is a more direct way to load a multi–column file into separate one–dimensional arrays:

```
a,b = np.loadtxt("filename.txt", unpack=True)
```

This command extracts the first column of *filename.txt* (which has index `0`) and places it into the one–dimensional array `a`. The second column (which has index `1`) is placed into the one–dimensional array `b`.

Exercise 8.2e

Consider once again the multi–column file from the previous exercises. Read the columns of this file directly into one–dimensional arrays and verify the results.

The `savetxt()` function will place the elements of a one–dimensional array into a file with a single column. We can save multiple one–dimensional arrays into the columns of a single file. For example, let's say our code computes the height of a ball as a function of time. Let `t` denote the array of times and `y` denote the array of heights. The command

```
np.savetxt("filename.txt", list(zip(t,y)))
```

will create the file *filename.txt* with `t` in the first column and `y` in the second column.

Exercise 8.2f

Use Eq. (7.1) to compute the height y of a ball as a function of an array of times t. Have your code save the data to a single file with t in the first column and y in the second column.

Exercise 8.2g

Write a program that uses `def` to define the function $y = e^{\sin(x)}$. Have your code compute y for at least 100 values of x from $-\pi$ to π. Output the x and y data to a file. Write a second program that

will read the data from the file and plot a graph of y versus x.

8.3 Random numbers

In scientific programming we sometimes need random numbers. How do we instruct the computer to choose a random number? The NumPy library includes several random number generators. We will use the default generator. Start with the command

```
rnum = np.random.default_rng()
```

This tells Python to prepare a random number generator and call it `rnum`. We can now issue the command

`rnum.integers(`*low*`,`*high*`)`

which yields a random integer from *low* to *high* (including *low*, excluding *high*). Each time this command is used in a program, the random number generator creates a new random integer.

Exercise 8.3a

Write a code that repeatedly produces and prints random integers between *low* `= 1` and *high* `= 20`. Design your code to stop if the random number equals `5`.

The command

`rnum.uniform(`*low*`,`*high*`)`

yields a random real number from *low* to *high* (including *low*, excluding *high*). Each number in the range is equally likely; hence the name `uniform`. Every time this command is used in a program, the random number generator creates a new random number.

Exercise 8.3b

Create a code that produces N random real numbers from *low* `= 0` to *high* `= 1`. Find the average of these numbers. Does the average tend to converge to 0.5 as N increases from 10^2 to 10^4? 10^6?

Random numbers generated by a computer aren't really random. After all, a computer can only follow a prescribed algorithm, and algorithms are fundamentally deterministic. The numbers generated by the NumPy (or

any other) random number generator are more accurately characterized as *pseudorandom*. For many purposes, they are "random enough."

If the computer follows a deterministic algorithm, how does it produce different random numbers each time you run the code? The answer is that the algorithm itself depends on a "seed" number. By default, the seed number is obtained from the operating system using various hardware sources such as mouse movements and fan noise. Instead of using the default seed, we can choose a seed number explicitly:

`rnum = np.random.default_rng(`*seednum*`)`

Note that *seednum* must be an integer. When developing a code that uses random numbers, it can be useful to assign a seed number explicitly. This causes the results to be reproducible.

Exercise 8.3c

Specify a seed number in your code from Exercise 8.3a. Run the code multiple times to make sure the results are reproducible. Try different seed numbers.

8.4 Real number formats

In Sec. 2.9 we introduced f-string formating. Given a variable `x = 23.39128287` the print statement `print(f"x = {x:5f}")` produces the output `x = 23.39128`. The `f` before the opening quotation mark tells Python to interpret the string as an f-string.

The number `23.39128` that appears in the output is produced by the part of the string between curly braces, namely, `x:.5f`. A colon separates the number to be printed, `x`, and the format of the number, `.5f`. This format tells Python to print `x` as a real number (float) rounded to `5` decimal places. That is, with `5` digits after the decimal point.

There are other formats. For example, `x:.5e` specifies scientific notation (exponential notation) with `5` digits after the decimal point. The "general" format `x:.5g` will print `x` to `5` significant figures. Examples of these formatting types are given in Table 8.1. The underbracket shows the digits after the decimal point (for `f` and `e` formats), or the significant figures (for `g` format).

These format types can be used with the NumPy `savetxt()` function as well. For example, `np.savetxt("`*filename.txt*`",A,"%.5f")` will save the contents of the numerical array `A`, with the numbers rounded to `5` decimal

x	x:.5f	x:.5e	x:.5g
23.39128287	23.39128	2.33913e+01	23.391
0.003248124	0.00325	3.24812e−03	0.00324 81
7424681.198	7424681.19800	7.42468e+06	7.4247e+06

Table 8.1: Real numbers `x` from the first column are shown with `.5f` (float), `.5e` (exponential) and `.5g` (general) formatting in the remaining columns. The digits specified by the number `5` are displayed with an underbracket.

places.

Exercise 8.4

Create a NumPy array of random real numbers from 0 to 10. Save the array to a text file with the numbers expressed to 8 significant figures.

8.5 Negative indexing

Let `A` be either a list or an array with `N` elements. The first element is `A[0]`, the second element is `A[1]`, *etc.* The last element is `A[N-1]`. Each of the elements can also be accessed with *negative indexing*. That is, for $i = 0, 1, \ldots, N-1$, element `A[i]` is also denoted `A[i-N]`. The first element of `A` is `A[0]` and `A[-N]`. The second element is `A[1]` and `A[1-N]`. The last element is `A[N-1]` and `A[-1]`. This is depicted in Fig. 8.1.

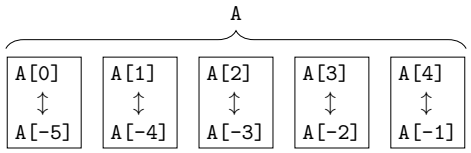

Fig. 8.1: A list or array `A` with $N = 5$ elements. The individual elements are accessed with positive indices 0 through 4, as well as negative indices -1 through -5.

Negative indexing can be useful in various contexts. For example, we might want to access the final element of an array `A`, but we don't immedi-

ately know the size of the array. The last element is always `A[-1]`.

Exercise 8.5

Create an array of numbers of the form n^3 that are less than 2500, where n is a positive integer. After the array has been created, have your code print out the largest element.

8.6 Object oriented programming

Classes, objects and methods form the foundation of "object oriented programming" (OOP). You don't need to be an expert in object oriented programming to understand the principles of scientific computation. But a basic understanding of methods, in particular, will help explain the syntax that we sometimes encounter.

An *object* is a member of a *class*. A *method* is a function that acts on an *object*. Let's look at a very simple example.

A regular polygon is a multi–sided figure with all sides of equal length, and equal interior angles. The code below creates the class of regular polygons. Objects in this class are triangles, squares, pentagons, *etc.* of various sizes. Each object has the following attributes: a name (called `name`), a certain number of sides (called `nsides`), and a particular side length (called `lsides`). Within the polygon class we define a *method* called `area()`, which is a function that computes the area of a polygon object. Here is the code:

```
class polygon:
    def __init__(self,name,nsides,lsides):
        self.name = name
        self.nsides = nsides
        self.lsides = lsides
    def area(self):
        N = self.nsides
        L = self.lsides
        return N*L**2/(4*np.tan(np.pi/L))
```

We can create a "big triangle" object with

```
BT = polygon('big triangle',3,12)
```

and a "small square" object with

```
SS = polygon('small square',4,1/3)
```

Now apply the `area()` method to find the area of each object. We do this by appending `.area()` to the end of the object. Thus,

```
print(f"area of {BT.name} is {BT.area():.2f}")
print(f"area of {SS.name} is {SS.area():.3f}")
```

produces the output

```
area of big triangle is 62.35
area of small square is 0.111
```

Exercise 8.6a

Add a method to the polygon class to compute the circumradius of a polygon object. (The circumradius $R = L/(2\sin(\pi/N)$ is the distance from the center to one of the vertices.) Now create an octagon with sides of length 2. What are the area and circumradius of this octagon?

We previously encountered methods in Sec. 8.3 on random numbers. The command

```
rnum = np.random.default_rng()
```

creates a random number object called `rnum`. The methods `integers()` and `uniform()` act on the object to produce random integers and random real numbers.

Exercise 8.6b

In Sec. 2.8 we introduced the method `split()`, which converts a string into a list of strings. Create a string `myname` consisting of your name. Compare `print(myname)` to `print(myname.split())`. What data type is `myname`? What data type is `myname.split()`?